Mark Summerfield

Python 3

programozás



KISKAPU

Addison-Wesley

Mark Summerfield

Python 3
programozás

Átfogó bevezetés a Python nyelvbe

KISKAPU

Addison-Wesley

A fordítás a következő angol eredeti alapján készült:

Mark Summerfield: Programming in Python 3: A Complete Introduction to the Python Language

Authorized translation from the English language edition, entitled PROGRAMMING IN PYTHON 3: A COMPLETE INTRODUCTION TO THE PYTHON LANGUAGE, 1st Edition, ISBN 0137129297, by SUMMERFIELD, MARK, published by Pearson Education, Inc., publishing as Addison-Wesley Professional.

Copyright © 2009 Pearson Education, Inc.

Translation and Hungarian edition © 2009 Kiskapu Kft.

All rights reserved. No part of this book, including interior design, cover design, and icons, may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording, or otherwise) without the prior written permission of the publisher.

Trademarked names appear throughout this book. Rather than list the names and entities that own the trademarks or insert a trademark symbol with each mention of the trademarked name, the publisher states that it is using the names for editorial purposes only and to the benefit of the trademark owner, with no intention of infringing upon that trademark.

Fordítás és magyar változat © 2009 Kiskapu Kft. Minden jog fenntartva!

A könyv egyetlen része sem sokszorosítható semmilyen módszerrel a Kiadó előzetes írásos engedélye nélkül. Ez a korlátozás kiterjed a belső tervezésre, a borítóra és az ikonokra is. A könyvben bejegyzett védjegyek és márkanevek is felbukkanhatnak. Ahelyett, hogy ezt minden egyes helyen külön jeleznénk, a Kiadó ezennel kijelenti, hogy a műben előforduló valamennyi védett nevet és jelzést szerkesztési célokra, jóhiszeműen, a név tulajdonosának érdekeit szem előtt tartva használja, és nem áll szándékában az azokkal kapcsolatos jogokat megszegni, vagy kétségbe vonni.

A szerzők és a kiadó a lehető legnagyobb körültekintéssel járt el e kiadvány elkészítésekor. Sem a szerző, sem a kiadó nem vállal semminemű felelősséget vagy garanciát a könyv tartalmával, teljességével kapcsolatban. Sem a szerző, sem a kiadó nem vonható felelősségre bármilyen baleset vagy káresemény miatt, mely közvetve vagy közvetlenül kapcsolatba hozható e kiadvánnyal.

Lektor: *Rézműves László*
Fordítás: *Gilicze Bálint, Rézműves László, Varga Péter*
Műszaki szerkesztő: *Csutak Hoffmann Levente*
Tördelés: *Kis Péter*

Felelős kiadó a Kiskapu Kft. ügyvezető igazgatója
© 2009 Kiskapu Kft.
1134 Budapest, Csángó u. 8.
Fax: (+36-1) 303-1619
http://www.kiskapukiado.hu/
e-mail: kiado@kiskapu.hu

ISBN: 978 963 9637 64 1

Készült a debreceni Kinizsi Nyomdában
Felelős vezető: *Bördős János*

Tartalomjegyzék

1. fejezet • Gyors bevezetés az eljárásközpontú programozásba

Python-programok írása és futtatása 2
1. kulcsösszetevő: Adattípusok 7
A Python „gyönyörű szíve” 7
2. kulcsösszetevő: Objektumhivatkozások 9
3. kulcsösszetevő: Gyűjteményes adattípusok 12
4. kulcsösszetevő: Logikai műveletek 15
5. kulcsösszetevő: Vezérlő utasítások 19
6. kulcsösszetevő: Számtani műveletek 24
7. kulcsösszetevő: Bemenet és kimenet 28
8. kulcsösszetevő: Függvények meghatározása és meghívása 31
Példák 33
bigdigits.py 33
generate_grid.py 37
Összefoglalás 39
Gyakorlatok 42

2. fejezet • Adattípusok

Azonosítók és kulcsszavak 45
Egészszerű típusok 49
Egészek 49
Logikai értékek 52
Lebegőpontos típusok 53
Lebegőpontos számok 53

Komplex számok 57
Decimális számok 58
Karakterláncok 60
Karakterláncok összehasonlítása 63
Karakterláncok szeletelése és egyelése 64
Karakterlánc-műveletek és tagfüggvények 66
Karakterláncok formázása az str.format() tagfüggvénnyel 75
Példa: print_unicode.py 83
Karakterkódolások 86
Példák 89
quadratic.py 89
csv2html.py 91
Összefoglalás 96
Gyakorlatok 98

3. fejezet • Gyűjteményes adattípusok

Sorozattípusok 102
Rendezett sorozatok 102
Nevesített rendezett sorozatok 105
Listák 107
Halmaztípusok 115
Halmazok 116
Halmazértelmezések 120
Fagyasztott halmazok 120
Megfeleltetési típusok 121
Szótárak 122
Alapértelmező szótárak 131
Gyűjtemények bejárása és másolása 132
Gyűjtemények másolása 142
Példák 144
generate_usernames.py 144
statistics.py 148
Összefoglalás 152
Gyakorlatok 154

4. fejezet • Vezérlési szerkezetek és függvények

Vezérlési szerkezetek 155
Feltételes elágazások 155
Ciklusok 157
Kivételkezelés 159
Kivételek elfogása és kiváltása 159
Egyedi függvények 167
Nevek és dokumentációs karakterláncok 172
Paraméterek és paraméterértékek kicsomagolása 173
Globális hatókörű változók elérése 176
Lambdafüggvények 178
Feltételezések 180
Példa: make_html_skeleton.py 181
Összefoglalás 187
Gyakorlat 188

5. fejezet • Modulok

Modulok és csomagok 192
Csomagok 196
Egyedi modulok 198
A TextUtil modul 198
A CharGrid modul 203
A Python alapkönyvtárának áttekintése 209
Karakterláncok kezelése 209
Parancssori programozás 210
Matematika és számok 212
Dátumok és időpontok 212
Algoritmusok és gyűjteményes adattípusok 213
Fájlformátumok, kódolások és az adatok megőrzése 215
Fájlok, könyvtárak és folyamatok kezelése 218
Hálózatkezelés és internetes programozás 221
XML 223
Egyéb modulok 225
Összefoglalás 226
Gyakorlat 228

6. fejezet • Objektumközpontú programozás

Az objektumközpontú megközelítés 232
Az objektumközpontú programozás alapelvei és alapfogalmai 233
Egyedi osztályok 236
Jellemzők és tagfüggvények 237
Öröklés és többalakúság 242
Jellemzők elérésének szabályozása tulajdonságokkal 244
Teljes és tökéletesen beágyazott adattípusok létrehozása 247
Egyedi gyűjteményosztályok 260
Gyűjteményeket egyesítő osztályok létrehozása 261
Összefoglalás 281
Gyakorlatok 283

7. fejezet • Fájlkezelés

Bináris adatok írása és olvasása 293
Befőttek tömörítéssel vagy anélkül 293
Nyers bináris adatok tömörítéssel és anélkül 295
Szövegfájlok írása és feldolgozása 307
Szövegírás 308
Szövegfeldolgozás 309
Szövegfeldolgozás szabályos kifejezések használatával 312
XML-fájlok írása és feldolgozása 315
Elemfák 315
DOM (Document Object Model) 319
XML-fájl írása saját kezűleg 322
XML-fájlok feldolgozása a SAX segítségével 323
Véletlen elérésű bináris fájlok 326
Az általános BinaryRecordFile osztály 327
Példa: A BikeStock modul osztályai 335
Összefoglalás 339
Gyakorlatok 340

8. fejezet • Haladó programozási eljárások

Magasabb szintű eljárásközpontú programozás 344
Elágazások létrehozása szótárak segítségével 344
Generátorkifejezések és -függvények 346
Dinamikus kódvégrehajtás és dinamikus betöltés 349
Helyi függvények és önhívó függvények 357
Függvény- és tagfüggvénydíszítők 362
Függvénymagyarázatok 366
Magasabb szintű objektumközpontú programozás 369
A jellemzők elérésének szabályozása 369
Funktorok 373
Környezetkezelők 375
Leírók 379
Osztályszintű díszítők 384
Elvont alaposztályok 387
Többszörös öröklés 395
Metaosztályok 397
Függvényközpontú programozás 402
Részleges függvények alkalmazása 405
Példa: Valid.py 406
Összefoglalás 409
Gyakorlatok 410

9. fejezet • Folyamatok és szálak

Feladatok végrehajtása folyamatokkal 414
Feladatok végrehajtása szálakkal 418
Példa: Többszálú szókereső program 420
Példa: A többször előforduló fájlokat kereső program többszálú változata 423
Összefoglalás 428
Gyakorlatok 429

10. fejezet • Hálózatkezelés

TCP-ügyfél létrehozása 433
TCP-kiszolgáló létrehozása 438
Összefoglalás 446
Gyakorlatok 446

11. fejezet • Adatbázis-programozás

DBM-adatbázisok 450
SQL-adatbázisok 454
Összefoglalás 461
Gyakorlat 462

12. fejezet • Szabályos kifejezések

A Python szabályoskifejezés-nyelve 464
Karakterek és karakterosztályok 464
Mennyiségjelzők 466
Csoportosítás és elfogás 468
Feltételezések és jelzők 470
A Python szabályoskifejezés-modulja 474
Összefoglalás 485
Gyakorlatok 485

13. fejezet • Bevezetés a grafikus felhasználói felület programozásába

Párbeszédablakos programok 490
Főablakos programok 497
Főablak létrehozása 497
Egyéni párbeszédablak létrehozása 508
Összefoglalás 511
Gyakorlatok 512

Tárgymutató 515

Zárszó 537

Bevezetés

A Python valószínűleg a legkönnyebben megtanulható és legegyszerűbben használható a széles körben elterjedt programozási nyelvek közül. A Python-kód könnyen olvasható és megírható, és tömörsége mellett áttekinthető marad. A Python nagyon kifejező nyelv, ami azt jelenti, hogy például a C++ vagy a Java nyelvvel szemben általában sokkal kevesebb Python-kódsor szükséges ugyanannak az alkalmazásnak a megírásához.

A Python rendszerfüggetlen nyelv: ugyanazt a Python-programot általában Windowson és olyan Unix-szerű rendszereken is futtathatjuk, mint a Linux, a BSD vagy a Mac OS X, egyszerűen a programot felépítő fájlnak vagy fájloknak a célszámítógépre történő átmásolásával, a „felépítés" vagy lefordítás szükségessége nélkül. Lehet olyan Python-programokat is írni, amelyek rendszerfüggő szolgáltatásokra támaszkodnak, de erre ritkán van szükség, mivel a háttérben a Pythonnak alapkönyvtárának majdnem az egésze, illetve a külső programkönyvtáraknak a nagy része teljes mértékben rendszerfüggetlen.

A Python egyik nagy erőssége, hogy az alapkönyvtára igen átfogó. Lehetővé teszi például fájlok letöltését az Internetről, tömörített archív fájlok kibontását, vagy éppen egy webkiszolgáló létrehozását – csupán egyetlen vagy néhány kódsor segítségével. Az alapkönyvtár mellett külső könyvtárak ezrei állnak hozzá rendelkezésre, amelyek némelyike az alapkönyvtárnál hatékonyabb és kifinomultabb szolgáltatásokat nyújt – ilyen például a Twisted hálózatkezelési könyvtár vagy a NumPy számítási könyvtár –, míg mások olyan speciális képességekkel rendelkeznek, amelyek az általánosabb alapkönyvtárban nem kaphattak helyet – ilyen például a SimPy szimulációs csomag. A külső könyvtárak többsége elérhető a Python Package Index weboldaláról, a pypi.python.org/pypi címen.

A Pythonnal eljárásközpontú (procedurális), objektumközpontú (objektumorientált) és – kisebb mértékig – függvényközpontú (funkcionális) programokat is írhatunk, bár maga a Python inkább objektumközpontú nyelv. Ebben a könyvben az eljárás- és objektum-központú programok írását egyaránt bemutatjuk, és kitérünk a Python függvényközpontú programozási lehetőségeire is.

Könyvünk célja, hogy megmutassuk, hogyan készíthetők jó Python-programok a Python 3 stílusában, és szeretnénk, ha a kötet az első elolvasás után is hasznos referenciakönyvnek bizonyulna a Python 3 nyelvhez. Bár a Python 3 inkább továbbfejlesztése a Python 2-nek, mintsem forradalmi újítás, egyes régebbi eljárások már nem működnek, vagy nem szükségesek a Python 3-ban, és új megoldások is megjelentek, amelyek kihasználják a Python 3 képességeit. A Python 3 jobb nyelv, mint a Python 2 – a Python 2-vel az évek során szerzett tapasztalatokra épül, és számos új szolgáltatást tartalmaz (valamint elveti a Python 2 tévedéseit), hogy még nagyobb élményt nyújtson a használata, mint a Python 2-é, és hogy kényelmesebb, egyszerűbb és következetesebb legyen.

Ez a könyv a Python *nyelvet* szándékozik megtanítani – bár a Python alapkönyvtárainak többsége szerepel benne, nem az összes. Ez nem jelent gondot, mert a könyv elolvasása után elegendő Python-ismerettel fogunk rendelkezni bármelyik alapkönyvtár vagy külső Python-könyvtár használatához, és saját könyvtármodulokat is képesek leszünk készíteni.

A kötetet többféle közönséget szem előtt tartva készítettük, beleértve az önmagukat képző hobbiprogramozókat, valamint a tanulókat, kutatókat, mérnököket és másokat, akiknek a programozás a munkájuk részét képezi, és természetesen a számítógépes szakembereket is. Annak érdekében, hogy a kötet mindenki számára hasznos lehessen, anélkül, hogy untatná a nagyobb tudással rendelkezőket, vagy magukra hagyná a kevésbé tapasztaltakat, feltételezünk némi programozói jártasságot (bármely nyelven), konkrétan az adattípusok (például a számok és karakterláncok), a gyűjteménytípusok (például a halmazok és listák), a vezérlési szerkezetek (például az if és while utasítások) és a függvények alapszintű ismeretét. Ezen kívül egyes példák és gyakorlatok a HTML jelölőnyelv alapfokú ismeretét is megkövetelik, a kötet végén található, haladóbb témákkal foglalkozó fejezetek pedig az adott témakörben vesznek adottnak némi jártasságot; az adatbázisokat tárgyaló részek például alapszintű SQL-tudást igényelnek.

A könyvet úgy szerkesztettük meg, hogy az Olvasó a lehető leggyorsabban a lehető leghatékonyabb tudást sajátíthassa el. Az első fejezet végére már képesek leszünk kicsi, de hasznos Python-programokat írni. Minden fejezet egy újabb témakört mutat be, gyakran kiszélesítve és elmélyítve a korábbi fejezetekben szerzett tudást. Ez azt jelenti, hogy ha a fejezeteket sorrendben olvassuk, bármely ponton képesek leszünk teljes programokat írni az addig tanultakra építve, majd ha készen vagyunk, természetesen folytathatjuk az olvasást az összetettebb, kifinomultabb megoldások megismerésével. Ez az oka annak, amiért egyes témákat csak bevezetünk egy fejezetben, és egy vagy több későbbi fejezetben ásunk jobban a mélyükre.

Egy új programozási nyelv tanítása során két kulcsfontosságú probléma merül fel. Az első, hogy néha olyan fogalmakat szükséges megtanítani, amelyek megértése más fogalmak ismeretétől függ, amelyek viszont közvetve vagy közvetlenül az első fogalomra épülnek. A második, hogy az elején az Olvasó lehet, hogy semmit sem vagy csak nagyon keveset tud

az adott nyelvről, ezért nagyon nehéz érdekes vagy hasznos példákat és gyakorlatokat bemutatni. Ebben a könyvben igyekeztünk mindkét problémára megoldást találni: először is feltételezünk némi programozó gyakorlatot; másodszor pedig az 1. fejezetben bemutatjuk a Python „szívét" – a Pythonnak azt a nyolc kulcsfontosságú részét, amelyeknek az ismerete elegendő ahhoz, hogy tűrhető programokat írjunk. Ebből a megközelítésből következik, hogy a könyv elején található fejezetek példáinak némelyike kissé mesterkélt, hiszen csak arra a tudásra támaszkodhat, amit az adott pontig megszereztünk. Ez a mesterkéltség azonban fejezetről fejezetre csökken, mígnem a 7. fejezet végén elérünk oda, hogy minden példa teljesen természetes Python 3 stílusban íródik.

Könyvünk megközelítése teljes mértékben gyakorlatias, és arra biztatjuk az Olvasót, hogy próbálja ki a példákat és gyakorlatokat, hogy első kézből szerezzen tapasztalatot. Ha csak lehetséges, példaként kicsi, de teljes programokat szerepeltetünk, hogy valószerű használati eseteket mutassunk be. A példák és a gyakorlatok megoldásai, amelyek mindegyikét kipróbáltuk Windowson, Linuxon és Mac OS X-en is a Python 3.0-val, elérhetők az Interneten, a `www.qtrac.eu/py3book.html` címen.

A könyv szerkezete

Az 1. fejezet a Pythonnak azt a nyolc kulcsfontosságú részét mutatja be, amelyeknek az ismerete elegendő ahhoz, hogy teljes programokat írjunk, ezen kívül ismertet néhányat az elérhető Python-programozási környezetek közül, valamint két aprócska példaprogramot, amelyek a Pythonnak a fejezet korábbi részében tárgyalt nyolc kulcsfontosságú részét használva készültek.

A 2–5. fejezetek a Python eljárásközpontú programozási lehetőségeibe nyújtanak bevezetést, beleértve az alapvető adat- és gyűjteménytípusok, valamint számos hasznos beépített függvény és vezérlési szerkezet, valamint a legegyszerűbb szövegfájl-kezelés ismertetését. Az 5. fejezetben azt mutatjuk be, hogy miként hozhatunk létre egyéni modulokat és csomagokat, valamint áttekintjük a Python alapkönyvtárát, hogy képet kapjunk a Python-csomagban rendelkezésre álló lehetőségekről, és ne kelljen feltalálnunk a melegvizet.

A 6. fejezet a Python objektumközpontú programozási képességeibe ad mélyreható bevezetést. A korábbi fejezetekben az eljárásközpontú programozásról tanultak továbbra is érvényesek maradnak, mivel az objektumközpontú programozás eljárásközpontú alapokra épül, például ugyanazokat az adat- és gyűjteménytípusokat, valamint vezérlési szerkezeteket használja.

A 7. fejezet témája a fájlok írása és olvasása. A bináris fájlokkal kapcsolatban a tömörítésről és a tetszés szerinti („véletlen") hozzáférésről lesz szó, míg a szövegfájlokat illetően a kézi és szabályos kifejezésekkel végzett feldolgozásról. A fejezet azt is meg-

mutatja, hogyan írhatunk és olvashatunk XML-fájlokat, illetve hogyan használhatjuk az elemfákat, a DOM-ot (Document Object Model, dokumentum-objektummodell) és az SAX-et (Simple API for XML, egyszerű alkalmazásprogramozási felület az XML-hez).

A 8. fejezet néhány korábbi fejezet anyagát ismétli át, miközben az adat- és gyűjteménytípusok, a vezérlési szerkezetek, a függvények, valamint az objektumközpontú programozás terén betekintést nyújt a Python számos haladó lehetőségébe. Ebben a fejezetben sok-sok új függvényt, osztályt és haladó eljárást – például a függvényközpontú programozást – megismerünk majd; az anyag kihívást jelent, de a jutalom sem marad el.

A fennmaradó fejezetekben további haladó témakörökkel foglalkozunk. A 9. fejezet arra mutat módszereket, hogy miként oszthatjuk el egy program terhelését több folyamat vagy szál között; a 10. fejezet az ügyfél–kiszolgáló rendszerű alkalmazások készítését mutatja be a Python szabványos hálózatkezelési támogatásának segítségével; a 11. fejezet az adatbázis-programozással foglalkozik (beleértve mind az egyszerű, kulcs–érték alapú „DBM" fájlokat, mind az SQL-adatbázisokat); a 12. fejezet a Python szabályos kifejezéseinek mininyelvét magyarázza el és szemlélteti, valamint a szabályos kifejezések modulját ismerteti; a 13. fejezet pedig a grafikus felület (GUI, Graphical User Interface) programozásába nyújt bevezetést.

A könyv fejezeteinek többsége meglehetősen hosszú, mert így az összefüggő témákat könnyebb fellapozni. A fejezeteket mindazonáltal alfejezetekre, alfejezetek alfejezeteire, azokat pedig sokszor még kisebb részekre osztottuk, hogy a nekünk legmegfelelőbb tempóban – mondjuk egyszerre csak egy alfejezetet elolvasva – haladhassunk.

A Python 3 beszerzése és telepítése

Ha friss, új Mac vagy más Unix-szerű rendszerünk van, a számítógépünkön valószínűleg már ott van a Python 3. Ezt úgy ellenőrizhetjük, hogy beírjuk a `python -V` parancsot (ügyeljünk a nagy V-re!) a konzolon (Mac OS X-en ez a `Terminal.app`): ha a változat száma 3, akkor rendelkezünk a Python 3-mal, tehát nem kell telepítenünk – ha nem ez a helyzet, olvassunk tovább.

A Windows és Mac OS X rendszerekhez könnyen használható grafikus telepítőcsomagok állnak rendelkezésre, amelyek lépésről lépésre végigvezetnek minket a telepítés folyamatán. Ezeket a csomagokat a www.python.org/download oldalról tölthetjük le. Windowshoz három különböző telepítőt biztosítanak – a sima „Windows installer" lesz a megfelelő, hacsak nem tudjuk biztosan, hogy a gépünkben AMD64 vagy Itanium processzor működik, mely esetben az adott processzorhoz készült változatot kell letöltenünk. Ha megszereztük a telepítőt, csak indítsuk el, és kövessük a képernyőn megjelenő utasításokat.

Linux, BSD és más Unix-rendszereken a Python telepítésének legegyszerűbb módja az operációs rendszer csomagkezelő rendszerének használata. A Pythont a legtöbb esetben több önálló csomagban terjesztik. Fedora rendszerre például van egy python csomag a Pythonhoz, és egy python-tools nevű az IDLE-hez (ez egy egyszerű fejlesztőkörnyezet), de megjegyzendő, hogy ezek a csomagok csak akkor Python 3 alapúak, ha a legújabb (10-es vagy annál későbbi változatú) Fedorával rendelkezünk. A Debian alapú terjesztések, például az Ubuntu esetében a szükséges csomagok a `python3` és az `idle3`.

Ha az operációs rendszerünkhöz nem áll rendelkezésre Python 3-csomag, akkor le kell töltenünk a forráskódot a `www.python.org/download` címről, és magunknak kell felépítenünk a Pythont. Ha a gzip-pel tömörített tar csomagot töltjük le, a `tar xvfz Python-3.0.tgz` paranccsal csomagolhatjuk ki, míg ha a bzip2-vel becsomagoltat, akkor a `tar xvfj Python-3.0.bz2` paranccsal. A telepítés és a beállítás ugyanúgy zajlik: először váltsunk az újonnan létrehozott Python-3.0 könyvtárra, és adjuk ki a `./configure` parancsot (ha helyi telepítést szeretnénk, adjuk meg a `--prefix` kapcsolót), majd futtassuk a `make` programot.

Előfordulhat, hogy a felépítés végén olyan üzenetet kapunk, miszerint nem minden modult lehetett felépíteni. Ez általában azt jelenti, hogy nem rendelkezünk néhány szükséges függvénykönyvtárral vagy fejállománnyal a számítógépen. Például ha a readline modult nem lehet felépíteni, akkor a csomagkezelő rendszerrel telepítenünk kell a megfelelő fejlesztési könyvtárat (ez Fedora alapú rendszereken a readline-devel, míg Debian alapú rendszereken a readline-dev – de a csomagnevek sajnos nem mindig ilyen könnyen kitalálhatók). Ha telepítettük a hiányzó csomagokat, adjuk ki újra a `./configure` és a `make` parancsot.

Ha a make sikeresen lefutott, a `make test` paranccsal meggyőződhetünk róla, hogy minden rendben van-e, bár ez nem szükséges, és a teszt jónéhány percig eltarthat.

Ha a `--prefix` segítségével helyi telepítést végeztünk, elég, ha kiadjuk a `make install` parancsot. Általában célszerű közvetett hivatkozást létrehozni a végrehajtható python fájlra (például így: `ln -s ~/local/python3/bin/python3.0 ~/bin/python3`, feltéve, hogy a `--prefix=$HOME/local/python3` kapcsolót használtuk, és a rendszer elérési útjában – `PATH` – szerepel a `$HOME/bin` könyvtár). Ugyanígy az IDLE-hez is érdemes lehet közvetett hivatkozást megadni (a fentieket alapul véve így:
`ln -s ~/local/python3/bin/idle ~/bin/idle3`).

Ha nem használtuk a `--prefix` kapcsolót, és rendszergazdai szintű (root) hozzáféréssel rendelkezünk, jelentkezzünk be root-ként, és adjuk ki a `make install` parancsot.
Az olyan sudo alapú rendszereken, mint az Ubuntu, a `sudo make install` parancsra lesz szükség. Ha a rendszeren a Python 2 is jelen van, az `/usr/bin/python` nem változik meg, tehát a Python 3-at python3-ként érhetjük el, a Python 3 IDLE környezetét pedig idle3 néven.

Köszönetnyilvánítás

Először is a könyv szakmai lektorainak szeretnék köszönetet mondani, köztük is először Jasmin Blanchette számítógéptudósnak, programozónak és szakírónak, akivel közösen már két C++/Qt-könyvet is írtunk. Jasmin részvétele a fejezetek megtervezésében, javaslatai és a példákra vonatkozó bírálatai, valamint az, hogy a szöveget gondosan átnézte, jelentősen növelte a könyv minőségét.

Georg Brandl vezető Python-fejlesztő és Python-dokumentációíró, aki a Python új dokumentációs eszközeinek fejlesztéséért felel. Georgnak számos apró hibát sikerült kiszúrnia, és állhatatos türelemmel addig magyarázott, amíg meg nem értettük és ki nem javítottuk a hibát. Ezen kívül a példákon is számos javítást eszközölt.

Phil Thompson Python-szakértő a PyQt, a jelenleg rendelkezésre álló talán legjobb Python GUI-könyvtár készítője. Phil éles szeme és néha provokatív visszajelzései számos pontot segítettek tisztázni és kijavítani.

Trenton Schulz a Nokia Qt Software-ének (korábban Trolltech) rangidős szoftvermérnöke, aki az összes korábbi könyvemnek értékes lektora volt, és most ismét a segítségemre sietett. Trenton gondosan átolvasta a kéziratot, és számos javaslatot tett, amelyeknek köszönhetően sikerült tisztába tenni néhány kérdést, és jelentősen javítani a szövegen.

A fent említett lektorokon kívül, akik mindnyájan a teljes könyvet elolvasták, köszönettel tartozom még David Boddie-nak, a Nokia Qt Software rangidős dokumentációírójának, aki egyben tapasztalt Python-programozó és nyílt forrású programokat író fejlesztő, és aki hasznos tanácsokkal látott el a könyv egyes részeivel kapcsolatban.

Köszönet illeti Guido van Rossumot, a Python megalkotóját is, valamint a szélesebb Python-közösséget, amelynek tagjai oly sokkal járultak hozzá ahhoz, hogy a Python és különösen annak könyvtárai ilyen hasznosak, a használatuk pedig ilyen élvezetes legyen.

Mint mindig, most is szeretnék köszönetet mondani Jeff Kingstonnak, az általam több mint egy évtizede használt Lout tördelőnyelv megalkotójának.

Külön köszönet jár szerkesztőmnek, Debra Williams Cauley-nak a támogatásáért, és hogy ez alkalommal is a lehető leggördülékenyebbé tette a könyv elkészítésének folyamatát. Köszönet illeti még Anna Popickot, aki olyan jól kézben tartotta a gyártást, és a korrektort, Audrey Doyle-t is, aki ismét kitűnő munkát végzett.

Végül, de nem utolsósorban, szeretnék köszönetet mondani feleségemnek, Andreának, amiért elviselte, hogy néha hajnali négykor felugrottam, hogy lejegyezzem az ötleteimet, vagy kijavítsak néhány kódot, amelyek ragaszkodtak hozzá, hogy azonnal kipróbáljam őket, valamint a szeretetéért, a hűségéért és a támogatásáért.

A szerzőről

Mark Summerfield

Mark a szoftveriparban, elsősorban programozóként szerzett több éves tapasztalat után a számítógép-tudományokból szerezte a diplomáját. Majdnem három évet töltött a Trolltech-nél dokumentációkezelőként, ahol megalapította és szerkesztette a Trolltech műszaki folyóiratát, a *Qt Quarterly*-t. (A Trolltech ma már a Nokia birtokában van, Qt Software néven.) Mark társszerzőként közreműködött a *C++ GUI Programming with Qt 4* című könyv megírásában, és ő írta a *Rapid GUI Programming with Python* és a *Qt: The Definitive Guide to PyQt Programming* című köteteket. Mark a Qtrac Ltd. (`www.qtrac.eu`) tulajdonosa, ahol független szerzőként, szerkesztőként, oktatóként és tanácsadóként dolgozik. Szakterülete a C++, a Qt, a Python és a PyQt.

1

Gyors bevezetés az eljárásközpontú programozásba

- Python-programok írása és futtatása
- A Python „gyönyörű szíve”

Ez a fejezet elegendő információval szolgál ahhoz, hogy megírhassuk első Python-programjainkat. Ha még nem telepítettük a Pythont, akkor most már legfőbb ideje lesz, hiszen csak így tehetünk szert a tanultakat megerősítő, első kézből származó tapasztalatokra. (A Python beszerzését és telepítését a jelentősebb operációs rendszereken a *Bevezetés* tárgyalja.)

A fejezet első része bemutatja a Python-programok létrehozásának és futtatásának módját. A Python nyelvű programkód megírására használhatjuk kedvenc, egyszerű szövegfájlokkal dolgozó szövegszerkesztőnket, bár az itt ismertetett IDLE fejlesztőkörnyezet a kódszerkesztésen felül további szolgáltatásokat is nyújt, például a Python nyelvű kóddal való kísérletezgetésre, illetve a kódban történő hibakeresésre szolgáló eszközöket.

A második rész a Python nyolc olyan kulcsfontosságú összetevőjét mutatja be, amelyek önmagukban is elegendőek használható programok írásához. Az egyes összetevőket a későbbi fejezetekben részletesen tárgyaljuk, és a könyv olvasásával haladva mellékük sorakozik a Python többi része is, vagyis a könyv végére érve a teljes nyelvet megismerjük, és programjainkban képesek leszünk mindazt használni, amit a nyelv kínál.

A fejezet utolsó részében megismerünk két rövid programot, amelyek a Python nyelvnek a második részben bemutatott elemei közül használnak néhányat – így azonnal belekóstolhatunk a Python nyelvű programok írásába.

Python-programok írása és futtatása

Karakterkódolások
☞ 86

Python nyelvű kód írható bármely olyan, egyszerű szövegfájlokkal dolgozó szövegszerkesztőben, amely képes ASCII vagy UTF-8 kódolású szöveges állományok betöltésére és mentésére. Alapértelmezés szerint a Python nyelvű fájlokról feltételezzük, hogy UTF-8-as karakterkódolásúak, vagyis az ASCII kódkészletnél tágabb halmazát kezelik a karaktereknek, olyat, amelyben ábrázolható szinte az összes nyelv összes betűje. A Python nyelvű fájlok kiterjesztése általában `.py` – annak ellenére, hogy néhány Unix-szerű rendszeren, például Linuxon és Mac OS X-en az alkalmazások normális esetben nem kapnak kiterjesztést –, a Python nyelven írt, grafikus felhasználói felületű programok pedig rendszerint a `.pyw` kiterjesztést kapják, főleg Windows és Mac OS X operációs rendszeren. Ebben a könyvben minden esetben a `.py` kiterjesztést használjuk a konzolon futó, illetve a `.pyw` kiterjesztést a grafikus felhasználói felületű, Python nyelvű programoknál. A könyvben bemutatott valamennyi példa változtatás nélkül futtatható azokon a rendszereken, amelyekre létezik a Python 3 környezet.

Hogy megbizonyosodjunk arról, hogy minden helyesen van beállítva – valamint azért, hogy a klasszikus első példán is túlessünk –, hozzuk létre bármelyik, egyszerű szövegfájlokat kezelő szövegszerkesztővel (a Windows Jegyzettömbje is megteszi – mindjárt kerítünk magunknak egy jobb szövegszerkesztőt) a következőket tartalmazó, `hello.py` nevű állományt:

```
#!/usr/bin/env python3

print("Hello", "World!")
```

Az első sor egy megjegyzés. A Python nyelvű programokban a megjegyzések a kettőskereszt (#) karakterrel kezdődnek, és a sor végéig tartanak. (A meglehetősen titokzatos megjegyzést nemsokára megmagyarázzuk.) A második sor üres – hacsak nem idézőjelek közötti szövegről van szó, a Python figyelmen kívül hagyja az üres sorokat, amelyek azonban az emberi lények számára gyakran lehetnek igen hasznosak, hiszen a hosszabb részleteket üres sorokkal tagolva könnyebben olvashatóvá válik a kód. A harmadik sorban Python nyelvű kódot találunk. A sorban a `print()` függvényt két paraméterrel hívjuk meg, amelyek közül mindkettő `str` (string, vagyis karakterlánc) típusú.

Futtatáskor a `.py` kiterjesztésű fájl minden sorát végrehajtja a gép, mégpedig az elsővel kezdve, majd egyesével folytatva. Ez eltérést jelent néhány egyéb nyelvhez – például a C++-hoz, illetve a Javához képest –, amelyeknél a program végrehajtása egy adott névvel rendelkező, megkülönböztetett szerepű függvénynél vagy tagfüggvénynél kezdődik. A program végrehajtásának egyenes vonala természetesen eltéríthető, ahogy azt majd a következő részben, a Python nyelv vezérlési szerkezeteinek tárgyalásánál látni fogjuk.

Abból fogunk kiindulni, hogy a Windowst használó olvasók a Python nyelvű programjaik fájljait a `C:\py3eg` könyvtárban, a Unix (vagyis: Unix, Linux és Mac OS X) operációs rendszert használók pedig a `$HOME/py3eg` könyvtárban helyezik el. A `hello.py` fájlt mentsük a `py3eg` könyvtárba, és zárjuk be a szövegszerkesztőt.

Most, hogy van egy programunk, akár le is futtathatnánk. A Python-programokat a Python-értelmező értelmezi, és mindezt rendszerint egy konzolablakban teszi. Windows operációs rendszeren az ablak neve „Parancssor", vagy „DOS Prompt", vagy „MS-DOS Prompt", vagy valami hasonló, és rendszerint a Start, All programs, Accessories (Start, Minden program, Kellékek) menüpontot választva érhető el. Mac OS X operációs rendszeren a konzolt az alapértelmezés szerint az Applications (Alkalmazások) menü Utilities (Segédprogramok) részéről megnyitható, de a Finder használatával is megtalálható `Terminal.app` program biztosítja, Unix rendszereken pedig használhatjuk az `xterm`-et, vagy az ablakkezelő környezet által kínált konzolt, például a `konsole`-t vagy a `gnome-terminal`-t.

Indítsunk el egy konzolt, és Windowson – feltételezve, hogy a Pythont az alapértelmezés szerinti helyre telepítettük – adjuk ki az alábbi parancsokat (a konzol kimenetét **`vastagon`**, az általunk beírtakat `vékonyan` szedtük):

```
C:\>cd c:\py3eg
C:\py3eg\>C:\Python30\python.exe hello.py
```

Minthogy a `cd` (change directory, könyvtárváltás) parancsnak teljes elérési utat adtunk, nincs jelentősége, hogy melyik könyvtárban adtuk ki a parancsot.

Unix környezetben a fentiek helyett az alábbi parancsot kell kiadnunk (feltételezve, hogy a Python 3 a `PATH` környezeti változóban tárolt könyvtárak valamelyikében van):

```
$ cd $HOME/py3eg
$ python3 hello.py
```

A kimenet elvileg mindkét esetben azonos:

```
Hello World!
```

Felhívjuk a figyelmet arra, hogy hacsak másként nem jelezzük, a Python viselkedése Mac OS X-en megegyezik bármely egyéb Unix rendszerrel. Az a helyzet, hogy valahányszor „Unix"-ot olvasunk, érthetünk alatta Linuxot, BSD-t, Mac OS X-et, és bármilyen egyéb Unixot, illetve Unix-szerű rendszert.

Bár a programnak mindössze egyetlen végrehajtható sora van, a program futtatásával több, a `print()` függvénnyel kapcsolatos következtetést is levonhatunk. Egyfelől azt, hogy a `print()` függvény a Python nyelv beépített részét képezi – a használatát megelőzően nem kell az „import" vagy az „include" kulcsszóval betöltenünk valamilyen függvénykönyvtárból –, másfelől pedig azt, hogy a kiírt karakterláncokat egy szóközzel választja el,

és az utolsóként kiírt karakterlánc után új sort kezd. Mindkettő olyan alapértelmezett viselkedés, amely – mint ahogy azt később látni fogjuk – megváltoztatható. Szintén említésre méltó, hogy a `print()` függvény az átadott paraméterek számára nem túl kényes: átadhatunk nagyon sok és nagyon kevés paramétert is.

`print()`
☞ 177

Az előzőhöz hasonló parancssorok begépelése hamar unalmassá válik. Szerencsére mind Windows, mind Unix rendszeren használhatunk kényelmesebb megoldást is. Feltételezve, hogy a `py3eg` könyvtárban vagyunk, Windowson egyszerűen kiadhatjuk az alábbi parancsot:

```
C:\py3eg\>hello.py
```

A Windows a fájltípusokhoz tartozó, a rendszerleíró adatbázisban (új nevén: beállításjegyzékben) lévő bejegyzések alapján minden olyan alkalommal, amikor `.py` kiterjesztésű fájlnév kerül a konzolba, automatikusan elindítja a Python-parancsértelmezőt.

Ha a Windows az alábbi kimenettel tér vissza…

```
('Hello', 'World!')
```

…az azt jelenti, hogy a rendszerre a Python 2 is telepítve van, és a Windows azt hívja meg a Python 3 helyett. Erre a problémára az lehet a megoldás, ha a `.py` kiterjesztésű fájlok társítását megváltoztatjuk, Python 2-ről Python 3-ra. További, kevésbé kényelmes, ámde biztonságosabb megoldást jelent, ha a Python 3 parancsértelmezőjét felvesszük az elérési útba (a példa feltételezi, hogy az alapértelmezett könyvtárba telepítettük), és minden egyes alkalommal mi magunk indítjuk el:

```
C:\py3eg\>path=c:\python30;%path%
C:\py3eg\>python hello.py
```

Kényelmesebbé tehetjük a dolgunkat, ha létrehozunk egy `py3.bat` fájlt, amelybe egyetlen sorként a `path=c:\python30;%path%` utasítás kerül, s amelyet a `C:\Windows` könyvtárba mentünk. Ezek után, ha azért indítjuk el a konzolt, mert Python 3 programokat akarunk futtatni, kezdhetjük a `py3.bat` futtatásával. Vagy akár megoldhatjuk azt is, hogy a `py3.bat` automatikusan lefusson. Ezt úgy tehetjük meg, hogy megváltoztatjuk a konzol tulajdonságait (a Start menü programjai között keressük meg a parancssort, és az egér jobb gombjával rákattintva nyissuk meg a hozzá tartozó tulajdonságablakot), és a Shortcut (Parancsikon) lap Target (Cél) mezőjében lévő beállítást egészítsük ki a „ /u /k c:\windows\py3.bat" bejegyzéssel. Figyeljünk az „/u" és a „/k" kapcsolók előtt, között és után lévő szóközökre, és arra, hogy az egész kiegészítés a „cmd.exe" után következzen.

Unix rendszeren először végrehajthatóvá kell tennünk az állományt, és csak ez után válik lehetővé a futtatása:

```
$ chmod +x hello.py
$ ./hello.py
```

A `chmod` parancsot természetesen csak egyszer kell kiadnunk, a továbbiakban elegendő a `./hello.py` parancsot begépelnünk, és a program lefut.

Amikor egy Unix rendszer konzoljából programot indítunk, a fájlnak csak az első két bájtját olvassa be a parancshéj. Ha ez a két bájt a kettőskereszt és a felkiáltójel (#!) karakter, akkor a parancshéj feltételezi, hogy a fájl végrehajtásához parancsértelmezőre van szükség, és a fájl első sora adja meg a használandó parancsértelmezőt. A sor neve *shebang*, és ha ilyet írunk egy fájlba, akkor az csak az első sorba kerülhet.

A shebang-sor rendszerint az alábbi két forma egyikében használatos:

```
#!/usr/bin/python3
```

vagy:

```
#!/usr/bin/env python3
```

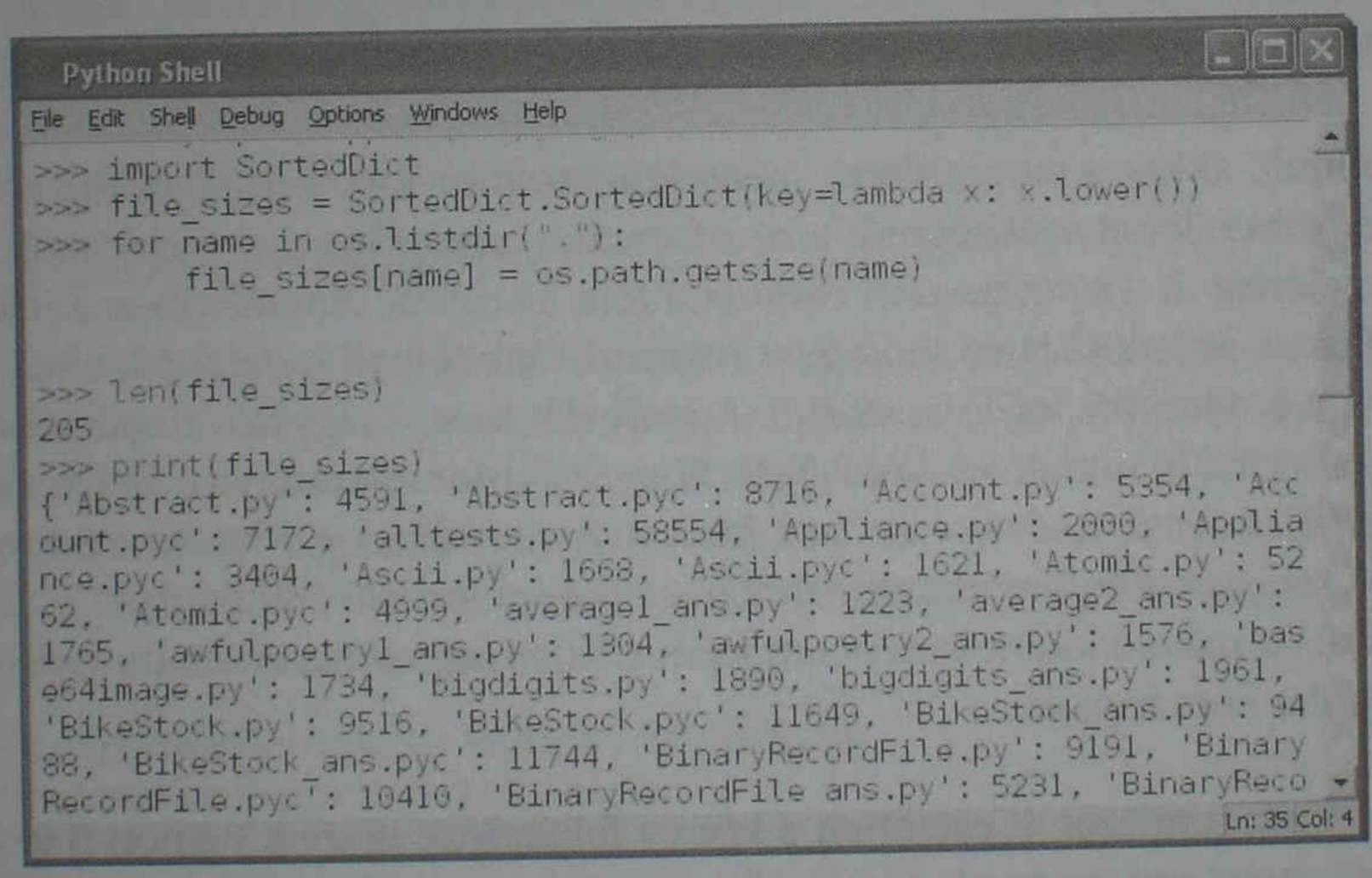
Ha az első formában írjuk, akkor a parancshéj a megadott értelmezőt használja. Olyan Python-programok esetében lehet szükségünk erre a formára, amelyeket egy webkiszolgáló futtat – a pontos elérési út természetesen eltérhet a fent közölttől. Amennyiben a második formát használjuk, a parancshéj a jelenlegi környezeti változóinak megfelelő, elsőként megtalált Python 3-értelmezőt fogja használni. A második forma nagyobb rugalmasságot biztosít, hiszen nem zárja ki annak a lehetőségét, hogy a Python 3-értelmező helye nem az `/usr/bin`, hanem például az `/usr/local/bin` könyvtár, vagy esetleg a `$HOME` könyvtár alá telepítettük. A shebang-sor Windows rendszereken szükségtelen, de ártalmatlan.
A könyv minden példája a másodikként ismertetett formájú shebang-sort tartalmazza, de kiírni itt a könyvben soha nem fogjuk.

Figyeljünk arra, hogy Unix rendszerek esetében a könyv feltételezi, hogy a Python 3 végrehajtható állománya – vagy egy a végrehajtható állományra mutató közvetett hivatkozás (symbolic link) – `python3` néven megtalálható a `PATH` környezeti változóban megadott elérési úton. Ha nem így lenne, akkor a példák shebang-sorát úgy kell megváltoztatnunk, hogy a helyes nevet – illetve a shebang-sor első formájának használatakor a helyes nevet és elérési utat – tartalmazza. Szintén jó megoldás, ha a `PATH` környezeti változóban feltüntetett elérési úton létrehozunk egy a Python 3 végrehajtható állományára mutató, `python3` nevű közvetett hivatkozást.

Sok hatékony szövegszerkesztő tartalmazza a Python nyelvű programok szerkesztésének beépített támogatását – ilyenek például a Vim és az Emacs. Ez a támogatás rendszerint a nyelv elemeinek színezett jelölésében, illetve a behúzások helyes kezelésében nyilvánul meg. További lehetőségként adott az IDLE nevű Python programozási környezet használata. Windows és Mac OS X rendszereken az IDLE az alapértelmezett telepítés során felkerül a számítógépre, egyéb Unix rendszereken pedig, ahogy a *Bevezetésben* már leírtuk, gyakran önálló csomagként szerezhető be.

A Python beszerzése és telepítése *Bevezetés* ☜

Ahogy az 1.1. ábrán lévő képernyőképen látható, az IDLE kinézete meglehetősen régies, a Unix rendszerek Motif nevű felhasználói felületét, illetve a Windows 95-öt idéző. Ennek oka abban keresendő, hogy az IDLE a Tk alapú, Tkinter nevű (a 13. fejezetben tárgyaljuk), grafikus felhasználói felületek készítésére való függvénykönyvtárat használja, nem pedig a manapság elterjedt, több lehetőséget nyújtó függvénykönyvtárak, például a PyGtk, a PyQt vagy a wxPython valamelyikét. A Tkinter használatának okai részint történelmiek, részint a Tkinter liberális felhasználási engedélyeivel kapcsolatosak, valamint azzal a ténnyel, hogy a Tkinter sokkal kisebb az egyéb, grafikus felhasználói felületek kialakításához használható függvénykönyvtáraknál. A Tkinter előnyös tulajdonságai közül megemlítjük, hogy a Python környezetnek alapértelmezett részét képezi, és a használata nagyon egyszerűen megtanulható.



1.1. ábra
Az IDLE Python-héja

Az IDLE három igen fontos eszközt ad a kezünkbe. Lehetővé teszi azt, hogy közvetlenül a Python-héjba (Python Shell) írjunk Python nyelvű kifejezéseket és kódot, illetve lássuk ezek eredményét. Van benne egy olyan kódszerkesztő, amely a Python-programok elemeit megfelelően színezi, és kezeli a behúzásokat is. Kapunk vele továbbá egy olyan hibakeresőt, amellyel végiglépkedhetünk a kód sorain, és azonosíthatjuk, illetve megszüntethetjük a programjainkban lévő hibákat. A Python Shell különösen egyszerű algoritmusok, kódrészletek és szabályos kifejezések kipróbálásakor hasznos, továbbá igen fejlett és rugalmas számológépként is használható.

Számos egyéb Python-fejlesztőkörnyezet van még, de a szerző azt tanácsolja, hogy maradjunk az IDLE használatánál – legalábbis kezdetben. További lehetőség, hogy a programjainkat az általunk választott, egyszerű szövegfájlokkal dolgozó szövegszerkesztőben írjuk meg, a hibakeresést pedig a `print()` függvény segítségével végezzük.

A Python-értelmező meghívása Python nyelvű program megjelölése nélkül is lehetséges. Ilyen esetekben a parancsértelmező interaktív módban indul el. Ebben a módban lehetőség nyílik arra, hogy pontosan úgy írjunk be Python nyelvű kifejezéseket, és pontosan úgy láthassuk a kifejezések eredményét, mintha az IDLE Python Shell ablakában dolgoznánk. A készenléti jel (prompt) is azonos az IDLE-ben megszokottal. Ugyanakkor az IDLE használata lényegesen egyszerűbb, így a szerző az javasolja, hogy használjuk az IDLE-t a kódrészletekkel való kísérletezgetésre. A könyvben szereplő rövid interaktív példák ismertetésekor feltételezzük, hogy vagy egy interaktív Python-értelmezőben, vagy az IDLE Python-héjában adjuk ki a parancsokat.

Mostanra tudjuk, hogy miként hozhatók létre, illetve miként futtathatók Python nyelvű programok, mindazonáltal egyetlen függvény ismeretével nyilvánvalóan nem jutunk túlzottan messzire. A következőkben jelentősen bővülni fognak a Python nyelvvel kapcsolatos ismereteink, ezáltal képessé válunk rövid, ámde használható programok írására, amit a fejezet utolsó szakaszában meg is teszünk.

A Python „gyönyörű szíve"

Ebben a részben megismerjük a Python nyolc kulcsfontosságú összetevőjét, a következőkben pedig bemutatjuk, hogy miként használható ez a nyolc összetevő néhány kicsi, ámde valós helyzetekben használható program megírására. Az itt tárgyaltakról még igen sok mindent el lehet mondani, ezért ha úgy érezzük, hogy valami hiányzik a Python nyelvből, vagy túl unalmas módszerrel oldunk meg problémákat, akkor a szövegben lévő hivatkozások, a tartalomjegyzék, illetve a tárgymutató használatával bátran ugorjunk előre. Szinte biztosan arra a megállapításra jutunk, hogy a Python mégiscsak rendelkezik a hiányolni vélt képességgel – gyakran az itt bemutatott kifejezéseknél lényegesen tömörebb formában is –, sőt még annál is többet tud.

1. kulcsösszetevő: Adattípusok

Az adatok tárolása és ábrázolása olyan alapvető dolog, amit minden programozási nyelvnek el kell végeznie. A Python jó pár beépített adattípussal rendelkezik, de mi ebből most csak néhányat tekintünk át. A Python az egész számokat – a pozitívokat és a negatívokat is – az `int`, a karakterláncokat – a Unicode kódolású karakterek sorozatait – pedig az `str` típussal ábrázolja. Íme néhány példa egész és karakterlánc típusú értékekre:

```
-973
21062458333711437339583605536734086463779019080109822508621955072
0
"Infinitely Demanding"
'Simon Critchley'
'positively áâãЄ÷©'
''
```

Megemlítenénk, hogy a listában lévő második szám a 2^{217} – azt, hogy a Python mekkora egész számok tárolására képes, csak a számítógép memóriája korlátozza, nem pedig valamiféle bájtban megadott szám. A karakterláncokat aposztrófok és idézőjelek egyaránt határolhatják, már amennyiben a karakterlánc egyik végén ugyanaz áll, mint a másikon. Ezen felül – lévén a Python nyelv a Unicode karakterkódolást használja – a karakterek nem csak az ASCII kódtáblában szereplők közül kerülhetnek ki, ahogy azt az utolsó előtti karakterlánc példáján is láthatjuk. Az üres karakterláncokat a karakterláncot határoló jelek között álló semmi jelképezi.

A Python nyelv a szögletes zárójelet (`[]`) használja a sorozatokon – például a karaktersorozatokon – belüli elemek elérésére. Példának okáért, ha a Python-héjban vagyunk, legyen szó a parancsértelmező interaktív módjáról vagy az IDLE-ről, beírhatjuk az alábbiakat (a Python Shell kimenetét **vastag**, az általunk beírtakat vékony betűkkel szedtük):

```
>>> "Hard Times"[5]
'T'
>>> "giraffe"[0]
'g'
```

A Python-héjak hagyományosan a három „nagyobb mint” (`>>>`) jelet használják készenléti jelként, ámbár ez megváltoztatható. A szögletes zárójelek minden sorozat vagy sorozatként felfogható adattípus esetében használhatók, így a karaktersorozatok és a listák esetében is. A programozási nyelv elemeinek egységes használata az egyik olyan dolog, ami csodálatosan széppé teszi a Python nyelvet. Jegyezzük meg, hogy a Python nyelv a sorszámozást mindig nullával kezdi.

A Pythonban mind az `str`, mind az alapvető, számértékeket tároló típusok, mint amilyen az `int` típus, *állandó értékűek* (immutable), vagyis az egyszer beállított értékükön nem lehet változtatni. Első hallásra ez meglehetősen furcsa korlátozásnak tűnhet, de a Python nyelvtana miatt ez a gyakorlatban nem fog gondot jelenteni. Egyetlen ok miatt említjük meg az egészet: a szögletes zárójel használatával megkaphatjuk egy karaktersorozat adott sorszámú helyén álló karakterét, de a zárójel nem használható arra, hogy más karakterre cseréljük a jelenleg ott lévőt. (Jegyezzük meg magunknak azt is, hogy a Python nyelvben a karakter nem egyéb, mint egy egyetlen karakter hosszúságú karakterlánc.)

Ha egy adatelemet egy adattípusról egy másikra szeretnénk átalakítani, az *adattípus*`(`*elem*`)` formát használhatjuk. Például:

```
>>> int("45")
45
>>> str(912)
'912'
```

Az `int()` utasítást használó átalakítás megbirkózik az érték előtti és mögötti szóközökkel, illetve tabulátorokkal, így az előző utasítás működött volna `int(" 45 ")` formában is. Az `str()` utasítással megvalósuló átalakítás majdnem minden adatelem esetében alkalmazható. Ahogy a 6. fejezetben látni fogjuk, a saját egyedi adattípusaink esetében is könnyedén megvalósítható az `str()`, az `int()`, illetve az egyéb adattípusokká való átalakítás támogatása, amennyiben a művelet értelmezhető. Ha az átalakítás nem sikerül, kivétel keletkezik – a kivételkezelést röviden alább, az ötödik pontnál mutatjuk majd be, mindenre kiterjedő tárgyalására pedig a 4. fejezetben kerül sor.

A karakterláncokat és az egészeket mindenre kiterjedően a 2. fejezetben tárgyaljuk, az egyéb beépített adattípusok, illetve a Python alapkönyvtáraiból származó néhány adattípus tárgyalása szintén itt kapott helyet. A fejezet az állandó értékű sorozatokon, például a karakterláncokon végezhető műveletekkel is foglalkozik.

2. kulcsösszetevő: Objektumhivatkozások

Sekély és mély másolás
☞ 142

Most, hogy van néhány adattípusunk, a következő szükséges dolgot az adattípusok tárolására használható változók jelentik. A Python nyelvben nem találkozunk a szó megszokott értelmében vett változókkal, helyettük *objektumhivatkozásokkal* dolgozhatunk. Ha állandó értékű, például `int` vagy `str` típusú objektumokról van szó, akkor objektumhivatkozás és változó között nincs észrevehető különbség. A megváltoztatható objektumok esetében pedig van ugyan különbség, de ez a gyakorlatban ritkán érvényesül. A könyvben a *változó* és az *objektumhivatkozás* kifejezéseket egyenértékűként fogjuk használni. Lássunk néhány aprócska példát, majd beszéljünk meg néhány részletet.

```
x = "blue"
y = "green"
z = x
```

A nyelvtani forma egyszerű, *`objektumHivatkozás = érték`* alakú. Nincs szükség az objektumhivatkozások előzetes bevezetésére (deklarációjára), és az érték típusát sem kell megadnunk. Amikor a Python az első kifejezést végrehajtja, létrehoz egy `str` típusú, a „blue” szöveget tartalmazó objektumot, és létrehozza az `str` típusú objektumra mutató objektumhivatkozást is. Gyakorlati szempontból nyugodtan mondhatjuk azt, hogy „a *blue* karakterlánchoz rendeljük az `x` változót”. A második kifejezés hasonló. A harmadik kifejezés `z` néven új objektumhivatkozást hoz létre, majd a hivatkozás céljául ugyanazt az objektumot állítja be, amire az `x` nevű objektumhivatkozás is hivatkozik – esetünkben a „blue” szöveget tartalmazó `str` típusú objektumot.

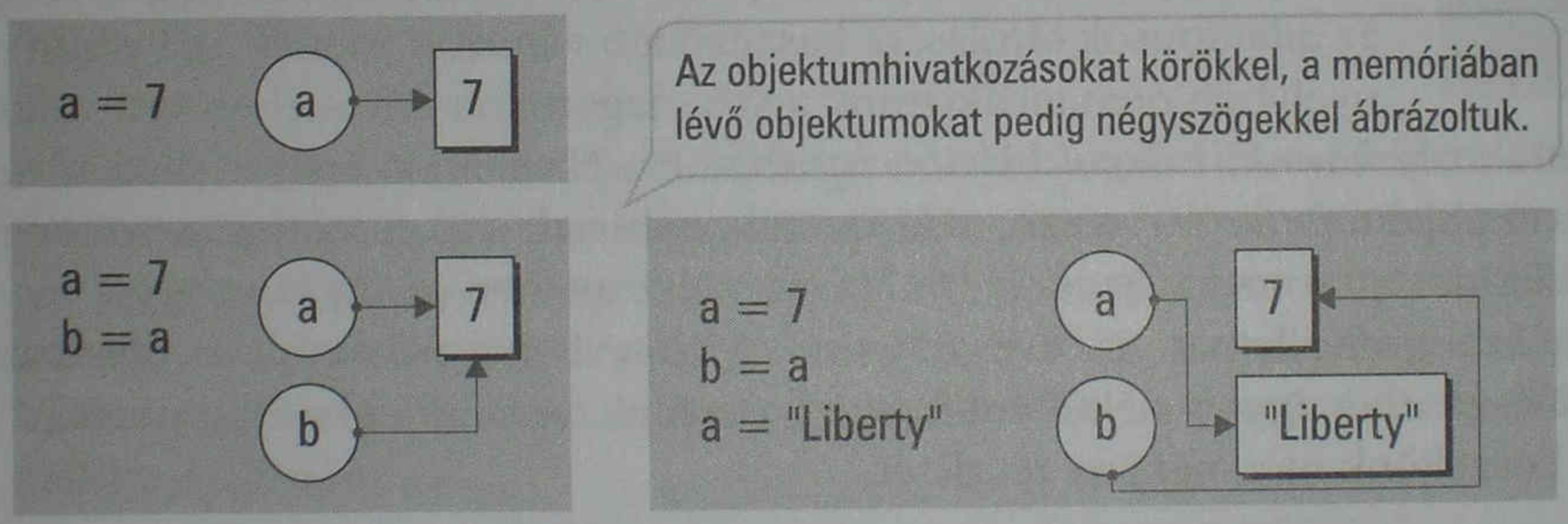
Az egyenlőségjel nem egyenértékű más programozási nyelvek változóknak értéket adó műveleti jelével. Az egyenlőségjel egy objektumhivatkozást kapcsol a memóriában lévő objektumhoz. Ha az objektumhivatkozás már eddig is létezett, akkor a hivatkozás lecserélésével mostantól egy másik, az egyenlőségjel jobb oldalán álló objektumra fog hivatkozni, ha pedig eddig még nem létezett, akkor az egyenlőségjel hozza létre.

Az előző, x, y és z hivatkozást tartalmazó példát folytatva adjunk ki néhány, a tárolt hivatkozásokat lecserélő utasítást (korábban már jeleztük, hogy a kettőskereszt karakter a sor végéig tartó megjegyzés kezdetét jelenti):

```
print(x, y, z) # a kiírt szöveg: blue green blue
z = y
print(x, y, z) # a kiírt szöveg: blue green green
x = z
print(x, y, z) # a kiírt szöveg: green green green
```

A negyedik (`x = z`) kifejezés után mind a három hivatkozás ugyanarra az `str` típusú objektumra hivatkozik. Lévén a „blue" karakterláncra már nem mutat egyetlen hivatkozás sem, a Python szemétgyűjtője nyugodtan felhasználhatóként jelölheti meg az általa elfoglalt memóriát.

Az objektumok és az objektumhivatkozások kapcsolati rendszerét az 1.2. ábra mutatja be.



1.2. ábra

Objektumok és objektumhivatkozások

Az objektumhivatkozások nevei (vagyis az *azonosítók*) tekintetében szólnunk kell pár korlátozásról. Az azonosítók nem egyezhetnek meg a Python nyelv kulcsszavaival; betűvel vagy aláhúzással kell kezdődniük, amelyet betűk, számok és aláhúzások szóközt vagy tabulátort nem tartalmazó sorozata követhet. Az azonosítók hossza nem korlátozott. A betűk és a számok a Unicode karakterkódolással leírható karakterek közül kerülhetnek ki, vagyis tartalmazzák az ASCII kódkészlet betűit és számait, („a", „b", ..., „z", „A", „B", ..., „Z", „0", „1",..., „9"), és ezen felül még sok mindent. A Python nyelv azonosítóiban megkülönböztetjük a kis- és a nagybetűket, vagyis az `AZONOSÍTÓ`, az `Azonosító` és az `azonosító` más-más azonosító neve. A 2. fejezetben további részleteket ismerhetünk meg, és bemutatunk néhány, némileg egzotikus példát is.

Azonosítók és kulcsszavak
☞ 45

A Python *dinamikusan típusos* nyelv, ami annyit tesz, hogy az objektum, amire egy objektumhivatkozás mutat, bármikor lecserélhető egy másik objektumra, és ez a másik objektum akár eltérő adattípusba is tartozhat. Az erősen típusos nyelvek (mint a C++ és a Java) csak

az érintett adattípusokra megadott műveletek végrehajtását engedélyezik. A Python nyelv is alkalmazza ezt a megszorítást, a Pythont mégsem soroljuk az erősen típusos nyelvek közé, ugyanis az érvényes műveletek köre megváltozhat, például olyankor, amikor egy objektumhivatkozás célobjektumát lecseréljük valamilyen más adattípusba tartozó objektumra. Például:

```
route = 866
print(route, type(route)) # a kiírt szöveg: 866 <class 'int'>
route = "North"
print(route, type(route)) # a kiírt szöveg: North <class 'str'>
```

A példában egy `route` nevű új objektumhivatkozást hozunk létre, és úgy állítjuk be, hogy egy `int` típusú, `866` értékű adatobjektumra hivatkozzon. Ekkor a `route` objektumhivatkozással még használhatnánk a „/” műveleti jelet, lévén az osztás az egész számok esetében érvényes művelet. Ez után a `route` objektumhivatkozást ismét felhasználjuk, és ezúttal egy másik, `str` típusú értékre, a „North”-ra hivatkozunk vele. Az `int` típusú objektumra pedig a szemétgyűjtő vár, hiszen immáron egyetlen objektumhivatkozás sem hivatkozik rá. Ha ekkor hajtanánk végre a `route` hivatkozás objektumán a „/” műveleti jellel jelölt műveletet, egy `TypeError` kivétel keletkezne, hiszen az osztás a karakterláncok esetében érvénytelen művelet.

A `type()` függvény a számára paraméterül átadott adatérték adattípusát (más szóval: „objektumosztályát”) adja vissza – a függvény igen hasznos lehet teszteléskor és hibakeréskor, de kész programkódban csak ritkán fordul elő, hiszen van jobb, hasonló feladatot ellátó eszköz is, ahogy azt a 6. fejezetben látni fogjuk.

`isinstance()`
☞ 166, 241, 398

Ha interaktív parancsértelmezőben vagy egy Python-héjban (amilyet az IDLE is nyújt) kísérletezgetünk Python-kóddal, akkor elég egyszerűen az objektumhivatkozás nevét beírnunk, és a Python kiírja a hozzá tartozó értéket. Például:

```
>>> x = "blue"
>>> y = "green"
>>> z = x
>>> x
'blue'
>>> x, y, z
('blue', 'green', 'blue')
```

Ez sokkal kényelmesebb módszer, mintha minden alkalommal a `print()` függvényt kellene hívnunk, de csak akkor működik, ha interaktív módban használjuk a Pythont, vagyis minden általunk írt programnak és modulnak a `print()` vagy egy ahhoz hasonló függvényt kell meghívnia, ha kimenetet akarunk megjeleníteni. Figyeljük meg, hogy az utolsó esetben a Python a kimenetet zárójelben, vesszőkkel elválasztva jeleníti meg, vagyis egy *rendezett sorozat* (tuple) formájában. A rendezett sorozat névvel az objektumokból képzett, rendezett, állandó értékű sorozatokat illetjük, és bővebben a következő pontban tárgyaljuk őket.

3. kulcsösszetevő: Gyűjteményes adattípusok

Gyakran bizonyulhat kényelmesnek, ha adatértékek egész gyűjteményeit tudjuk tárolni. A Python nyelv számos adattípust kínál gyűjtemények tárolására, ilyen például a társításos tömb (asszociatív tömb), illetve a halmaz. Itt és most a sok lehetőség közül mindössze kettőt ismertetünk, az egyik a `tuple`, a másik a `list`. A Python nyelv rendezett sorozatai (tuple) és listái bármilyen típusba tartozó adatokból képesek akárhányat tárolni. A `tuple` *állandó értékű* típus, vagyis ha egyszer létrehoztunk egyet, a későbbiekben nem változtathatunk rajta. A lista ezzel szemben *változtatható* (mutable) típus, tehát értékek beszúrására és eltávolítására bármikor lehetőségünk van.

Egy `tuple` létrehozása vesszők használatával lehetséges, ahogy az alábbi példákban is látható:

```
>>> "Denmark", "Norway", "Sweden"
('Denmark', 'Norway', 'Sweden')
>>> "one",
('one',)
```

Függvények írása és meghívása
☞ 31

A Python a `tuple` típusú szerkezetek tartalmát zárójelben jeleníti meg. Sok programozó ezt a viselkedést lemásolva a rendezett sorozat elemeit a program írásakor is zárójelbe teszi. Ha egy egyetlen értéket tároló rendezett sorozatunk van, és zárójelet szeretnénk használni, akkor a vesszőt is ki kell tennünk, például így: `(1,)`. Üres rendezett sorozat üres zárójellel hozható létre: `()`. A függvények hívásakor az egyes paraméterek elválasztására is vesszőt használunk, vagyis amennyiben `tuple` típusú értéket adunk át paraméterként, a bizonytalanság megszüntetése végett zárójelet kell használnunk.

Íme néhány lista:

```
[1, 4, 9, 16, 25, 36, 49]
['alpha', 'bravo', 'charlie', 'delta', 'echo']
['zebra', 49, -879, 'aardvark', 200]
[]
```

Listákat létrehozhatunk szögletes zárójelek (`[]`) használatával, ahogy itt is tettük, a későbbiekben pedig a listák létrehozásának egyéb módjait is meg fogjuk ismerni. A negyedikként bemutatott lista üres lista.

A színfalak mögött persze sem a listák, sem a rendezett sorozatok nem tárolnak semmiféle adatot, viszont objektumhivatkozásokat annál inkább. Amikor lista vagy rendezett sorozat jön létre (illetve a listák esetében új elem beszúrására kerül sor), akkor az átadott objektumhivatkozások másolatait tároljuk bennük. Az olyan literális elemek esetében, mint az egészek és a karakterláncok, a memóriában létrejön a megfelelő adattípusba tartozó objektum, megkapja a kezdőértékét, ezt követően létrejön az objektumra mutató objektumhivatkozás, és ez az objektumhivatkozás kerül majd a listába, illetve a rendezett sorozatba.

Mint ahogy minden egyéb, a Python nyelvben az adatszerkezetek is objektumok, lehetséges tehát a gyűjteményeket tároló adatszerkezetek más adatszerkezetekben történő tárolása, példának okáért létrehozható listákat tároló lista, és ez a művelet semmiféle különleges nyelvtani ismeretet nem igényel. Léteznek olyan helyzetek, amikor jelentőséggel bír az a tény, hogy a listák, a rendezett sorozatok, illetve a Python nyelv egyéb, gyűjtemények tárolására szolgáló adatszerkezetei objektumok helyett az objektumokra mutató hivatkozásokat tárolnak – az ilyen eseteket a 3. fejezetben tárgyaljuk.

Sekély és mély másolás
☞ 142

Az eljárásközpontú (procedurális) programozás függvényhívásokkal dolgozik, és a függvények gyakran kapnak adatértékeket paraméterül. Példaként tekintsük a már ismert `print()` függvényt. A `len()` függvény, amely egyetlen adatértéket vár paraméterül, és az átadott adat „hosszát” `int` típusú értékként adja vissza, szintén gyakran használatos a Python nyelvben. Az alábbiakban néhány, a `len()` függvényt hívó utasítás következik (ezúttal nem használtunk félkövér betűket az értelmező kimenetének jelzésére, lévén véleményünk szerint az Olvasó mostanra el tudja különíteni egymástól a kimenetet és a begépelendő szöveget):

```
>>> len(("one",))
1
>>> len([3, 5, 1, 2, "pause", 5])
6
>>> len("automatically")
13
```

A rendezett sorozatok és a listák mérettel is rendelkeznek, azaz esetükben van értelme méretről beszélni, márpedig az ilyen adatszerkezetek azok, amelyek a `len()` függvény számára értelmezhető paraméterként átadhatók. (Amennyiben a `len()` függvény mérettel nem rendelkező adatot kap paraméterül, kivétel keletkezik.)

Sized
☞ 390

A Python nyelv minden adatértéke adott adattípusba (ha tetszik: *objektumosztályba*) tartozó objektum (ha tetszik: *objektumpéldány*). Az *adattípus*, az *osztály*, illetve az *objektumosztály* kifejezéseket egyenértékűként fogjuk használni. Az objektumpéldányok és a más nyelvekben használt sima adatértékek (példaként említhetjük a C++ és a Java beépített, számokat tároló típusait) közötti egyik döntő különbség, hogy az objektumok rendelkezhetnek *tagfüggvényekkel*. A tagfüggvény (method) lényegében véve nem más, mint egy adott objektumra meghívott függvény. Példának okáért a `list` típusnak van egy `append()` nevű tagfüggvénye, amellyel a következőképpen helyezhetünk el egy újabb objektumot a lista végén:

```
>>> x = ["zebra", 49, -879, "aardvark", 200]
>>> x.append("more")
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more']
```

Az `x` nevű objektum tudja magáról, hogy ő egy `list` típusú objektum (a Python minden objektuma ismeri a saját típusát), ezért nem kellett külön megadnunk az adattípust. Az `append()` függvény megvalósításában az első paraméter maga az `x` objektum – a Python nyelv tagfüggvény-támogatásának köszönhetően nekünk ezt nem kell kiírnunk.

Az `append()` függvény a működése során megváltoztatja az eredeti listát. Ez azért történhet meg, mert a listák változtatható objektumok. A művelet hatásosabb ebben a formában, mint ha új, az eredetinél egy elemmel hosszabb listát hoznánk létre, majd az objektumhivatkozás célpontjául ezt az új listát állítanánk be. Mindez a nagyon hosszú listák esetében különösen igaz.

Egy eljárásközpontú nyelvben ugyanezt az eredményt érhetjük el az alábbi, a Python nyelvtani szabályainak tökéletesen megfelelő utasítással, ahol a `list.append()` önálló utasításként szerepel:

```
>>> list.append(x, "extra")
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more', 'extra']
```

Itt megadjuk az adatszerkezetet, illetve az adatszerkezet tagfüggvényét, amelynek első paraméterül egy olyan adattípusba tartozó adatot adunk át, amilyenre a tagfüggvényt meg szeretnénk hívni. Ezt további paraméterek követik. (Az öröklődés tekintetében a két nyelvtani forma között hangyányi eltérést figyelhetünk meg; a gyakorlatban túlnyomó részben az első formát használjuk. Az öröklést a 6. fejezet tárgyalja.)

Amennyiben az objektumközpontú programozásban nem mozgunk otthonosan, az előbb említettek első hallásra esetleg idegenül hangzanak. Egyelőre elég, ha elfogadjuk, hogy a Python nyelvben megvannak a hagyományos értelemben vett függvények, amelyeket *`függvényNév(paraméterek)`* formában hívunk, továbbá megvannak az objektumok tagfüggvényei is, amelyeket viszont *`objektumNév.tagfüggvényNév(paraméterek)`* formában hívunk. (Az objektumközpontú programozást a 6. fejezetben tárgyaljuk.)

A pont műveleti jel által jelölt művelet egy objektum jellemzőihez történő hozzáférésre használatos. Az objektumjellemzők bármilyen objektumtípusba tartozhatnak, bár eleddig csak tagfüggvényként használt jellemzőket mutattunk be. Minthogy egy jellemző lehet jellemzőkkel rendelkező objektum is, amelynek a jellemzői szintén bírhatnak jellemzőkkel, és így tovább, az elérni kívánt jellemző jelölésére olyan sok pontot használhatunk, amennyit csak kell.

A `list` osztályú adatszerkezeteknek sok további tagfüggvényük is van: megtaláljuk köztük az `insert()` és a `remove()` tagfüggvényeket is. Az előző arra való, hogy adott sorszámú helyen új elemet szúrjunk be a listába, az utóbbi pedig eltávolítja a lista adott sorszámú elemét. Ahogy korábban már megjegyeztük, a Python mindig nullával kezdi a sorszámozást.

Az előzőekben már láttuk, hogy a szögletes zárójellel karakterláncokból nyerhetünk ki karaktereket, és a példa kapcsán azt is megjegyeztük, hogy a műveleti jel bármely sorozaton használható. A listák sorozatok, így hát bátran elvégezhetünk ezekhez hasonló műveleteket:

```
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more', 'extra']
>>> x[0]
'zebra'
>>> x[4]
200
```

A `tuple`-ok is sorozatok, vagyis amennyiben az `x` egy rendezett sorozat neve lett volna, a szögletes zárójellel ugyanúgy megkaptuk volna a karaktereket, mint ahogy az `x` nevű lista esetében meg is kaptuk. Minthogy azonban a listák változtathatók (a karakterláncoktól és a rendezett sorozatoktól eltérően, amelyek állandók), a szögletes zárójel a lista elemeinek megadására is használható.

```
>>> x[1] = "forty nine"
>>> x
['zebra', 'forty nine', -879, 'aardvark', 200, 'more', 'extra']
```

Ha olyan sorszámot adunk meg, ami túl nagy vagy túl kicsi, kivétel keletkezik – a kivételkezelésről röviden szólunk az 5. pontban, a kivételek teljeskörű tárgyalására pedig a 4. fejezetben kerül sor.

A *sorozat* kifejezést az eddigiekben többször használtuk, anélkül, hogy a jelentését pontosan megadtuk volna, és egyelőre ezen a gyakorlaton nem is változtatunk. Mindazonáltal a Python nyelv pontosan megadja, hogy a sorozatnak, illetve a többi használható adattípusnak milyen tulajdonságokkal kell rendelkeznie – ahogy erről majd a 8. fejezetben olvashatunk.

A listákat, a rendezett sorozatokat és a Python nyelv többi beépített, gyűjtemények tárolására szolgáló adattípusát a 3. fejezetben tárgyaljuk.

4. kulcsösszetevő: Logikai műveletek

A programozási nyelvek alapvető képességei közé tartozik a logikai műveletek végrehajtása. A Python nyelv logikai műveletei négy csoportba sorolhatók, és az alábbiakban áttekintjük a legfontosabb tudnivalókat mind a négy csoporttal kapcsolatban.

Az azonossági művelet

Minthogy a Python nyelvben minden változó valójában objektumhivatkozás, néha van értelme feltenni azt a kérdést, hogy két vagy akár több objektumhivatkozás ugyanarra az objektumra hivatkozik-e. Az `is` műveleti jel (operátor) olyan kétparaméteres műveletet jelöl, amelynek eredménye `True` (igaz), ha a művelet bal oldali objektumhivatkozása ugyanarra az objektumra hivatkozik, mint a jobb oldali objektumhivatkozás. Íme néhány példa:

```
>>> a = ["Retention", 3, None]
>>> b = ["Retention", 3, None]
>>> a is b
False
>>> b = a
>>> a is b
True
```

Megjegyeznénk, hogy rendszerint nincs értelme az `int`, az `str` és a legtöbb egyéb adattípus `is` művelettel való összehasonlításának, hiszen inkább az értékek összehasonlításával tudunk kezdeni valamit. Az a helyzet, hogy az adatelemek `is` művelettel való összehasonlítása olyan eredményhez is vezethet, amire nem számít az ember. Ilyet láthattunk az előző példában is, ahol – bár az `a` és `b` listák ugyanazokat az elemeket kapták értékül – maguk a listák külön-külön objektumként jöttek létre, és eképpen az `is` művelet `False` (hamis) eredményt adott vissza az első használatakor.

Az azonosság-összehasonlító műveletek egyik előnyös tulajdonsága, hogy a végrehajtásuk igen gyors. Ennek oka abban keresendő, hogy az objektumhivatkozások által hivatkozott objektumokat nem kell összehasonlítani. Az `is` művelet során csak az objektumok memóriacímeinek összehasonlítása történik meg: azonos memóriacím azonos objektumot jelent.

Az `is` műveletet legelterjedtebben az adatelemeknek a beépített `None` nevű, a semmit jelölő objektummal való összehasonlítására használjuk. A `None` (nincs) érték gyakran jelöli helyőrzőként az „ismeretlen” vagy „nem létező” értékeket:

```
>>> a = "Something"
>>> b = None
>>> a is not None, b is None
(True, True)
```

Az azonosságvizsgálat értelmének megfordítására az `is not` műveleti jelet használjuk.

Az azonossági művelet arra való, hogy kideríthessük, hogy két objektumhivatkozás ugyanarra az objektumra hivatkozik-e, vagy pedig azt, hogy egy objektum `None` értékű, vagyis nem létező-e. Ha objektumok összehasonlítása a cél, akkor inkább valamelyik összehasonlító műveletet érdemes használnunk.

Összehasonlító műveletek

A Pythonban is megtaláljuk a megszokott kéttényezős összehasonlító műveleteket, amelyeknek a jelentése is a szokásos: < (kisebb, mint), <= (kisebb vagy egyenlő), == (egyenlő), != (nem egyenlő), >= (nagyobb vagy egyenlő) és végül > (nagyobb, mint). Ezek a műveletek az objektumok értékét hasonlítják össze, pontosabban az összehasonlító műveletben szereplő objektumhivatkozások által hivatkozott objektumokét. Íme néhány, a Python Shellbe beírt példa:

```
>>> a = 2
>>> b = 6
>>> a == b
False
>>> a < b
True
>>> a <= b, a != b, a >= b, a > b
(True, True, False, False)
```

Minden pontosan úgy történik, ahogy egész számok esetén arra számítunk.
A karakterláncok összehasonlításával is minden rendben lévőnek tűnik:

```
>>> a = "many paths"
>>> b = "many paths"
>>> a is b
False
>>> a == b
True
```

Bár `a` és `b` különböző (külön-külön létező) objektumok, az értékük megegyezik, az összehasonlító műveletek szerint egyenlők. Mindazonáltal legyünk tudatában, hogy a Python Unicode kódolással ábrázolja a karaktereket, és emiatt az ASCII karakterkészleten kívüli karaktereket tartalmazó karakterláncok összehasonlítása összetettebb, körmönfontabb dolog annál, mint amilyennek elsőre hinni vélnénk – ezt a problémakört a 2. fejezetben tárgyaljuk részletesen.

Karakterláncok összehasonlítása
☞ 63

A Python nyelv összehasonlító műveleteinek egyik különösen remek tulajdonsága, hogy összeláncolhatók egymással. Például:

```
>>> a = 9
>>> 0 <= a <= 10
True
```

Ezzel a módszerrel sokkal elegánsabban dönthető el, hogy egy adott adatelem benne van-e egy adott tartományban, mint ha a más nyelvekben megszokott módon két önálló, de logikai és művelettel összekapcsolt összehasonlító műveletet kellene elvégeznünk. A módszernek megvan az a további előnye, hogy az adatelem kiértékelését csak egyszer kell elvégezni – hiszen a kifejezésben csak egy helyen szerepel –, és ez olyasmi, ami fontos különbséget jelent, ha az adatelem értékének kiszámítása műveletigényes folyamat, vagy ha az adatelem elérésének nemkívánatos mellékhatásai lehetnek.

A Python dinamikusan típusos, és a dinamikusság „erős felének" köszönhetően az értelmetlen összehasonlítások kivételt váltanak ki. Például:

```
>>> "three" < 4
Traceback (most recent call last):
...
TypeError: unorderable types: str() < int()
```

Ha egy létrejött kivételt nem kezelünk, a Python egy a kivétel útját visszakövető üzenettel nyugtázza az eseményt, majd kiírja a kivételhez tartozó hibaüzenetet. Az olvashatóság kedvéért a kimenetből elhagytuk a visszakövetést, helyette három pont szerepel.* Ugyanilyen, `TypeError` kategóriájú kivételt kaptunk volna akkor is, ha a `"3" < 4` összehasonlítást írtuk volna, a Python ugyanis nem bocsátkozik találgatásokba a szándékainkat illetően. A probléma helyes megközelítése az, ha a Pythont utasítjuk a típusátalakítás elvégzésére, például így: `int("3") < 4`. Ha nem ezt tesszük, akkor összehasonlítható objektumokat kell használnunk: két egész vagy két karakterlánc típusút.

Másik lehetőség: `FuzzyBool`
☞ 255

A Python nyelvben egyszerű az olyan egyedi adattípusok létrehozása, amelyek jól illeszkednek a beépítettek közé. Egyszerű például a saját, egyedi, számok tárolására használható típus létrehozása, amely a beépített `int` típusba tartozó objektumokkal összehasonlítható lesz, de a karakterláncokkal vagy egyéb nem számokat tároló típusba tartozó objektumokkal nem.

A tagsági művelet

Az olyan adattípusok esetében, amelyek sorozatok vagy gyűjtemények – karakterláncok, listák vagy rendezett sorozatok –, az `in` művelettel vizsgálható a tagsági viszony megléte, a `not in` művelettel pedig a tagsági viszony meg nem léte. Például:

```
>>> p = (4, "frog", 9, -33, 9, 2)
>>> 2 in p
True
>>> "dog" not in p
True
```

Listák és rendezett sorozatok esetében az `in` művelet lineáris keresést végez, amely nagyobb (több tízezres nagyságrendű vagy ennél is nagyobb) gyűjtemények esetében esetleg nagyon lassú lehet. Másfelől, az `in` művelet nagyon gyors, ha szótár- vagy halmazszerkezetet kell vizsgálnia – ezt a két gyűjteményes adattípust a 3. fejezetben ismertetjük. Most pedig íme egy példa az `in` művelet karakterlánccal történő használatára:

```
>>> phrase = "Peace is no longer permitted during Winterval"
>>> "v" in phrase
True
>>> "ring" in phrase
True
```

---

* A visszakövetés (angol nyelvű szakirodalomban a *traceback* és a *backtrace* kifejezés is használatos rá) a kezeletlen kivétel létrejöttétől a hívásokat tartalmazó verem aljáig listázza a hívásokat.

A tagsági művelet karakterláncok esetében arra használható, hogy kívánt hosszúságú karaktersorozatok jelenlétét vizsgálja egy adott karakterláncban. (Ahogy korábban már rámutattunk, a karakterek tulajdonképpen egyetlen elem hosszúságú karakterláncok.)

Logikai műveletek

A Python nyelv három logikai műveletet bocsát a rendelkezésünkre: az `and` (és), az `or` (vagy) és a `not` (nem) műveleteket. Az `and` és az `or` művelet egyaránt szereti rövidre zárni a dolgokat, vagyis a művelet eredményét meghatározó tényezőt adják vissza, ahelyett, hogy logikai értéket kapnánk tőlük (kivéve, ha a művelet tényezői is logikai értékek). Lássuk, mit jelent ez a gyakorlatban:

```
>>> five = 5
>>> two = 2
>>> zero = 0
>>> five and two
2
>>> two and five
5
>>> five and zero
0
```

Ha a kifejezés megfelelő környezetben szerepel, akkor a kiértékelés eredménye logikai érték lesz. Ennek értelmében – mondjuk egy `if` kifejezésen belül szerepeltetve az előző vizsgálatokat – az eredmények `True` (igaz), `True` és `False` (hamis) lennének.

```
>>> nought = 0
>>> five or two
5
>>> two or five
2
>>> zero or five
5
>>> zero or nought
0
```

Az `or` művelet hasonlóan működik. Logikai kiértékelést eredményező környezetben az előző kifejezések eredménye ismét `True`, `True`, illetve `False` lenne. A `not` művelet egyetlen tényezőt vár, és a művelet eredménye minden esetben egy logikai érték. Az előző példát továbbgondolva a `not (zero or nought)` kifejezés kiértékelésének eredménye `True`, a `not two` kifejezésé pedig `False`.

5. kulcsösszetevő: Vezérlő utasítások

Korábban már említettük, hogy a Python-programokat tartalmazó fájlokban lévő minden egyes kifejezés végrehajtódik, kezdve az elsővel, majd sorról sorra továbbhaladva. A programsorok lefutásának sorrendje megváltoztatható, mégpedig egy függvény vagy tagfüggvény meghívásával, illetve egy vezérlési szerkezettel – ilyenek a feltételes elágazások vagy a ciklusok. A program futásának menete akkor is más irányt vesz, ha kivétel keletkezik.

Ebben a szakaszban röviden áttekintjük a Python nyelv `if` kifejezését, illetve a `while` és a `for` ciklusszervező utasításokat, míg a függvények működésének tárgyalását a nyolcadik pontra, a tagfüggvények ismertetését pedig a 6. fejezetre halasztjuk. Áttekintjük továbbá a kivételkezeléssel kapcsolatos alapvető tudnivalókat – a tárgykör részletes ismertetésére a 4. fejezetben kerül sor. Mindenekelőtt azonban tisztázni fogjuk néhány szakkifejezés értelmét.

Logikai kifejezésnek nevezünk minden olyan kifejezést, amelynek a kiértékelése logikai eredményre (`True` vagy `False`) vezet. A Python nyelvben a kifejezés eredménye `False`, ha a `False` beépített állandóról, illetve a `None` nevű, megkülönböztetett szerepű objektumról van szó, valamint ha üres sorozatot, gyűjteményt (például üres karakterlánc, üres lista, illetve üres rendezett sorozat), vagy 0 értékű, szám típusú adatot vizsgálunk. Minden más `True` értéket kap. Ha saját egyedi adatszerkezetet hozunk létre (példát a 6. fejezetben láthatunk), akkor mi dönthetjük el azt is, hogy logikai kiértékeléskor mit kell eredményül kapnunk.

A Python szakzsargonjában a kódblokkok, vagyis a nullától különböző számú kifejezésből álló sorozatok neve *suite*, a magyar fordításban azonban megmaradunk az elterjedtebb kódblokk kifejezés használatánál. A Python nyelvtana bizonyos esetekben, bizonyos helyeken *megköveteli* egy kódblokk jelenlétét. Ha az ilyen helyekre semmit sem szeretnénk írni, vagyis semmilyen tennivalót nem kell elvégezni, akkor használhatjuk a Python nyelv `pass` kulcsszavát, amely éppen erre utasítja a parancsértelmezőt. A `pass` kulcsszót használhatjuk olyan esetekben is, ahol csak annyit akarunk jelezni, hogy számításba vettük a szóban forgó lehetőséget is.

Az if utasítás

A Python nyelvben az if utasítást tartalmazó kifejezések általános formája a következő:*

```
if logikai_kifejezés1:
    kódblokk1
elif logikai_kifejezés2:
    kódblokk2
...
elif logikai_kifejezésN:
    kódblokkN
else:
    else_kódblokk
```

Az `elif` záradékok használata nem kötelező, és a záró `else` záradék is elhagyható. Amennyiben meg akarunk különböztetni egy bizonyos esetet, de semmilyen műveletet nem szeretnénk az eset előfordulásakor elvégezni, akkor az adott ág kódblokkjaként használhatjuk a `pass` kulcsszót.

---

* A könyvben három pont helyettesíti a hiányzó sorokat.

Az első dolog, ami a C++-hoz, illetve a Javához szokott programozó szemébe ötlik, az a zárójelek (legyen szó sima vagy kapcsos zárójelekről) hiánya. Másodikként a kettőspontok használata szokott feltűnni – a kettőspontokat a nyelvtani szabályok követelik meg, és eleinte igen könnyű elfeledkezni a használatukról. Kettőspontot használunk az `else` és az `elif` kulcsszó után, és lényegében minden olyan helyen, ahol kódblokknak kell következnie.

A legtöbb programozási nyelvtől eltérően a Python nyelv a behúzásokat a blokkszerkezet jelzésére használja. Ez néhány programozóból nemtetszést vált ki – főleg, még mielőtt kipróbálná a módszert –, másokat egyenesen érzelmi kitörésre sarkall. Ugyanakkor mindössze pár napig tart, amíg megszokja az ember, néhány hét, illetve hónap elteltével pedig hajlamossá válunk a zárójeleket tartalmazónál szebbnek, áttekinthetőbbnek értékelni a zárójelmentes kódot.

Lévén a kódblokkokat behúzások jelzik, azonnal felmerül a kérdés: „Milyen behúzást kell használni?" A Python nyelv megfelelő irányelvei négy szóköz használatát javasolják behúzási szintenként, illetve a tabulátorok használatától való tartózkodásra intenek. A legtöbb modern szövegszerkesztőben beállítható, hogy ezt automatikusan kezelje – természetesen ilyen az IDLE szerkesztője is, és ilyen a legtöbb, Pythonra felkészített szövegszerkesztő. A Python remekül működik más számú szóköz, illetve tabulátor használata mellett is, és a kétféle jelet akár keverve is használhatjuk – a lényeg, hogy a behúzások egyértelműek legyenek. Ebben a könyvben a hivatalos Python-irányelvekhez tartjuk magunkat.

Íme egy igen egyszerű, `if` utasítást tartalmazó kifejezést használó példa:

```
if x:
    print("x is nonzero")
```

Ebben az esetben a feltétel (az `x`) kiértékelése `True` eredményt ad, és sor kerül a kódblokk (a `print()` függvény) végrehajtására.

```
if lines < 1000:
    print("small")
elif lines < 10000:
    print("medium")
else:
    print("large")
```

A fenti kód pedig egy lényegesen összetettebb, `if` utasítást tartalmazó kifejezés, amely egy a `lines` változó értékét leíró szót jelenít meg a képernyőn.

A while utasítás

A `while` utasítás arra való, hogy egy kódblokkot többször is végrehajtsunk (a „több" lehet egy, sőt nulla is). A végrehajtások száma a `while` ciklusszervező utasításhoz rendelt logikai kifejezés állapotától függ. Íme az utasítás formája:

```
while logikai_kifejezés:
    kódblokk
```

Az az igazság, hogy a `while` utasítás, illetve az általa megvalósított ciklusok nyelvtana ennél azért kifinomultabb, lévén mind a `break`, mind a `continue` utasítások támogatottak, sőt egy egyébiránt elhagyható `else` záradék használatára is van lehetőség – minderről a 4. fejezetben szólunk részletesen. A `break` („kitörés", megszakítás) utasítás az utasítást tartalmazó legbelső cikluson kívülre helyezi át a vezérlést – vagyis a ciklusból való „kitörést" teszi lehetővé. A `continue` (folytatás) utasítás hatására a program futása a ciklus elejétől folytatódik. Általában mind a `break`, mind a `continue` utasítást egy `if` utasítást tartalmazó kifejezés belsejében helyezzük el, eképp kötve feltételhez a ciklus viselkedését.

```
while True:
    item = get_next_item()
    if not item:
        break
    process_item(item)
```

Ennek a `while` utasítással megvalósított ciklusnak meglehetősen általános a szerkezete. A ciklus addig fut, amíg van feldolgozandó elem. (A `next_item()` és a `process_item()` utasítások a program más részén megvalósított egyedi függvényeket jelölnek.) A fenti példában a `while` utasítás belsejében lévő kódblokknak része egy feltételvizsgálat, amelynek belsejében szintén egy kódblokkra lelünk – a Python nyelvben ez követelmény –; ez a kódblokk most mindössze egyetlen `break` utasításból áll.

A for ... in utasítás

A Python nyelv `for` utasításra épülő ciklusa is használja az `in` kulcsszót (amely más környezetben a tagsági művelet jelzésére szolgál), mégpedig a következő formában:

```
for változó in bejárható_objektum:
    kódblokk
```

A `while` utasítással megvalósított ciklushoz hasonlatosan a `for` utasítással működő is támogatja a `break` és a `continue` utasítás használatát, és szintén rendelkezik elhagyható `else` záradékkal. A *változó* a ciklus ismétlődő lefutásai során a *bejárható_objektum*ban tárolt egyes objektumokra hivatkozik. A *bejárható_objektum* olyan objektumot jelöl, amelynek az elemei egyesével végigjárhatók. Ilyen bejárható objektumok például a karakterláncok (ezeket karakterenként lehet végigjárni), a listák, a rendezett sorozatok és a Python nyelv további, gyűjteményként kezelt adatok tárolására szolgáló adattípusai.

```
for country in ["Denmark", "Finland", "Norway", "Sweden"]:
    print(country)
```

Ezúttal meglehetősen egyszerű módját választottuk az országlista megjelenítésének. A gyakorlatban egy változó használata lényegesen elterjedtebb módszer:

```
countries = ["Denmark", "Finland", "Norway", "Sweden"]
for country in countries:
    print(country)
```

Valójában a `print()` függvénnyel közvetlenül írathatók ki teljes listák, illetve rendezett sorozatok, például a `print(countries)` utasítás is működik, mégis gyakori eset, hogy inkább egy `for` ciklus segítségével (vagy listaértelmezéssel, lásd később) írjuk ki a gyűjtemények tartalmát, és így megtartjuk a kiírás formázása feletti teljes ellenőrzést.

```
for letter in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
    if letter in "AEIOU":
        print(letter, "is a vowel")
    else:
        print(letter, "is a consonant")
```

Listaértelmezés
☞ 113

A fenti kódrészletben első alkalommal használjuk az `in` kulcsszót a `for` utasítás kiegészítőjeként. A `letter` nevű változó a ciklus egymás utáni futásainak során egyesével veszi fel az „A", a „B", illetve a többi értéket, egészen „Z"-ig. A kódrészlet második sorában ismét az `in` kulcsszót használjuk, ezúttal azonban tagsági műveletben. Figyeljük meg, hogy a példában egymásba ágyazott kódblokkok találhatók. A `for` utasítással szervezett ciklushoz tartozó kódblokk az `if...else` kifejezés, de mind az `if`, mind az `else` ágakon egy-egy újabb kódblokk kapott helyet.

Kivételkezelés alapfokon

A Python nyelv sok függvénye, illetve tagfüggvénye kivétel létrehozásával jelzi, ha hiba vagy egyéb jelentős esemény történt. A kivételek éppen olyan objektumok, mint a Python nyelv többi objektuma, és ha karakterlánccá alakítjuk őket – például képernyős megjelenítéshez –, akkor a kivételek szöveges üzenetként jelennek meg. A kivételkezelők egyszerű alakját az alábbi sorok szemléltetik:

```
try:
    try_kódblokk
except kivétel1 as változó1:
    kivételkezelő_kódblokk1
...
except kivételN as változóN:
    kivételkezelő_kódblokkN
```

Nem árt megjegyeznünk, hogy az `as változó` rész elhagyható; elképzelhető, hogy csak annyiban érdekel bennünket a dolog, hogy létrejött egy bizonyos kivétel, és nem érdekes a hozzá tartozó üzenetszöveg.

A teljeskörű utasításforma kifinomultabb; példának okáért minden egyes `except` záradék kezelhet több kivételt is, és egy elhagyható `else` záradék is létezik. Mindezt a 4. fejezet tárgyalja részletesen.

Az alapelv a következő: ha a `try` kódblokkjában lévő összes kifejezés kivétel létrejötte nélkül fut le, akkor az `except` ágak kimaradnak; ha pedig a `try` kódblokkjában kivétel keletkezik, a vezérlést azonnal az első, a létrejött kivételnek megfelelő `except` kódblokkja kapja meg. Ez annyit tesz, hogy a Python nem hajtja végre a `try` kódblokkjában lévő kifejezések közül azokat, amelyek a kivétel létrejöttét okozó után következnek. Ilyen esetekben, amennyiben megadtuk az `as` *`változó`* részt, a változóba bekerül a kivételobjektumra mutató objektumhivatkozás.

Amennyiben a kivételt kezelő `except` kódblokkon belül újabb kivétel jön létre, vagy amennyiben olyan kivétel keletkezik, amelynek egyik `except` utasítás sem feleltethető meg, a Python újabb próbálkozást tesz a megfelelő `except` utasítás fellelésére, ezúttal tágabb hatókörben. A megfelelő kivételkezelő megtalálására irányuló erőfeszítések egyre táguló hatókörben, illetve az egymásból hívott függvényekben visszafelé haladva folytatódnak, egészen addig, amíg megfelelő kivételkezelőre lelve a kivétel kezelése megtörténik, vagy megfelelő kivételkezelő hiányában a program futása a kezeletlen kivétel miatt megszakad. Kezeletlen kivétel miatti leálláskor a Python megjeleníti a visszakövetést, illetve a kivételhez tartozó üzenetszöveget.

Íme egy példa:

```
s = input("enter an integer: ")
try:
    i = int(s)
    print("valid integer entered:", i)
except ValueError as err:
    print(err)
```

Ha a felhasználó „3.5” értéket ad meg, a kimenet az alábbi lesz:

```
invalid literal for int() with base 10: '3.5'
```

De ha 13-at írunk be, akkor ez:

```
valid integer entered: 13
```

Sok könyv nehéz témának tekinti a kivételkezelést, és igyekszik a tárgyalását addig halogatni, ameddig csak lehetséges. A Python működése szempontjából a kivételek létrejötte – a kivételek kezelése pedig különösen – alapvető jelentőségű, így kezdettől fogva figyelembe vesszük. Ahogy majd látni fogjuk, azzal, hogy a „kivételes” eseteket elválasztjuk a számunkra igazán érdekes folyamatoktól, a kivételkezelők használata lényegesen olvashatóbbá teszi a kódot.

6. kulcsösszetevő: Számtani műveletek

A Python a számtani műveletek (aritmetikai műveletek) teljes skáláját a rendelkezésünkre bocsátja. Megtaláljuk a négy matematikai alapművelet kéttényezős műveleteit: a + az összeadás, a – a kivonás, a * a szorzás, a / pedig az osztás műveletét jelzi. Ezen felül a Python

adattípusai közül sok használható az értéknövelést és értékadást egy művelettel megvalósító műveletekkel – ilyenek például a += és a *=. A +, a - és a * műveleti jelek mindegyike a várt módon viselkedik – amennyiben mindkét tényező egész szám.

```
>>> 5 + 6
11
>>> 3 - 7
-4
>>> 4 * 8
32
```

Vegyük észre, hogy a – jel egy- és kéttényezős műveletként egyaránt használható. Előbbi esetben a negatív számok, utóbbiban a kivonás jelzésére szolgál. Amikor azonban az osztáshoz érünk, a Python kilóg a tömegből:

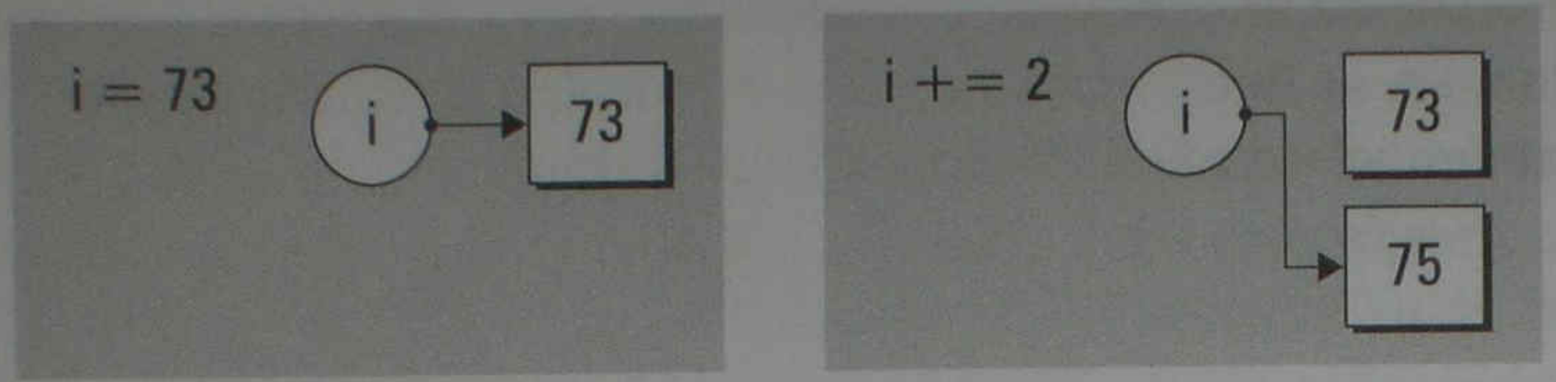
```
>>> 12 / 3
4.0
>>> 3 / 2
1.5
```

Az osztás műveletének eredménye lebegőpontos érték, nem pedig egész szám, holott sok más nyelv egész számot ad értékül, csonkolva az esetleges törtrészt. Ha egész eredményre van szükségünk, bármikor átalakíthatjuk az értéket az `int()` függvénnyel – vagy használhatjuk a későbbiekben tárgyalt, csonkoló osztást végző műveletet (`//`).

Számműveletek és -függvények
☞ 50

Első pillantásra az előző kifejezések kevés meglepetésre adnak okot, különösképpen azoknak, akik ismerősen mozgolódnak a C és a hozzá hasonló nyelvek körében. Az ilyen nyelvekben ugyanis az értéknövelést és értékadást egységként megvalósító műveletek (bővített értékadó műveletek) a kód rövidítésének eszközei – például az `a += 8` és az `a = a + 8` műveletek alapvetően egyenértékűek. Mindazonáltal volna itt két említésre méltó apróság. Az egyik csak a Python nyelv esetében igaz, a másik azonban valamennyi programozási nyelv ilyesféle műveleténél fennáll.

Az első dolog, ami figyelmet érdemel, az, hogy az `int()` adattípusba tartozó objektumok állandó értékűek, vagyis egy ilyen objektum értéke az értékadást követően nem módosítható. A színfalak mögött tehát valójában a következő történik, amikor az értéknövelést és értékadást egyszerre megvalósító műveletek állandó értékű objektumokat kapnak tényezőül: végrehajtódik a művelet, majd az eredmény egy újonnan létrejövő objektumba kerül, ezután az eredeti objektumra mutató objektumhivatkozás új értéket kap, és így már az eredményt tartalmazó objektumra hivatkozik az eredeti helyett. Az előzőleg ismertetett esetben tehát az `a += 8` sorhoz érve a Python kiszámolja, hogy mennyi az `a + 8`, az eredményt elraktározza egy új `int` típusú objektumban, majd az `a` objektumhivatkozás céljául megjelöli az újonnan létrehozott objektumot (és amennyiben az `a` objektumhivatkozás által eredetileg hivatkozott objektumra többé nem mutat érvényes objektumhivatkozás, az objektum helyét a memóriában a szemétgyűjtő felhasználhatóként fogja megjelölni). Az 1.3. ábrán az előzőekben elmondottakat követhetjük nyomon.



1.3. ábra

Egyidejű értéknövelés és értékadás állandó értékű objektum esetében

A második apróság lényege pedig az, hogy az `a` *`műveleti_jel`*`= b` kifejezés nem pontosan ugyanaz, mint az `a = a` *`műveleti_jel`* `b`. A programnak az első esetben az `a` értékét csak egyszer kell a memóriából lehívnia, a művelet végrehajtása tehát gyorsabb lehet. Ezen felül, amikor az `a` összetett kifejezést jelöl (például egy listaelem helyét menet közben kiszámolót, mondjuk *`elemek[eltolás + index]`* alakban), az első formában végrehajtott művelet hibatűrőbb lehet, mert a kód karbantartójának a hely kiszámolását végző kifejezést szükség esetén elég egyetlen helyen javítania.

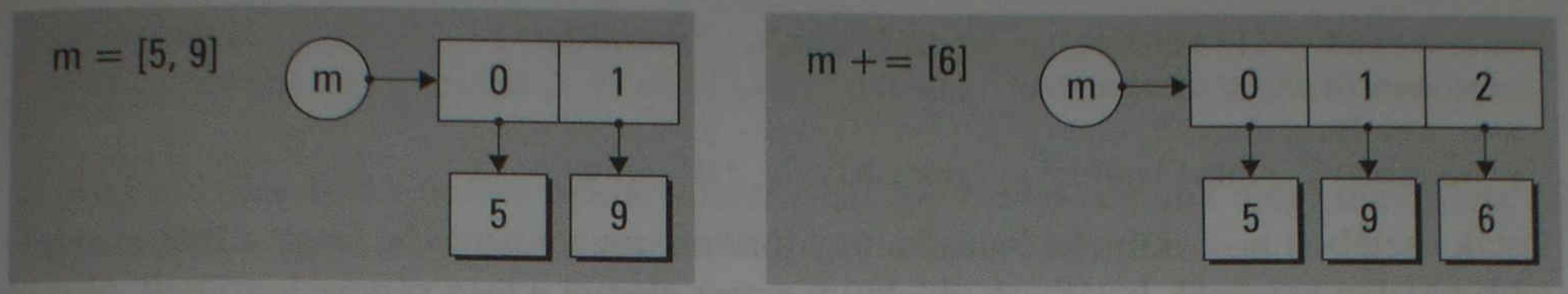
A `+` és a `+=` műveleti jeleket a Python a karakterláncok és listák esetében túlterheli (vagyis más adattípusnál más értelemben is használja). Az első jel az összefűzés (konkatenáció) műveletét hajtja végre, a második pedig karakterláncok esetében hozzáfűzést, listák esetében bővítést (másik lista hozzáfűzését) jelent.

```
>>> name = "John"
>>> name + "Doe"
'JohnDoe'
>>> name += " Doe"
>>> name
'John Doe'
```

A karakterláncok az egész számokat tartalmazó objektumokhoz hasonlóan állandó értékűek, ezért aztán a `+=` műveleti jel hatására új objektum jön létre, és a kifejezésben baloldalt álló objektumhivatkozás ettől kezdve erre az új objektumra hivatkozik – pontosan úgy, mint ahogy az `int` típusnál leírtuk. A listák esetében az utasításforma ugyanez, de a színfalak mögött más zajlik:

```
>>> seeds = ["sesame", "sunflower"]
>>> seeds += ["pumpkin"]
>>> seeds
['sesame', 'sunflower', 'pumpkin']
```

Lévén a listák változtathatók, a `+=` műveleti használatakor az eredeti objektum változik meg, és így nincs szükség a `seeds` objektumhivatkozás célobjektumának megváltoztatására. Az 1.4. ábrán áttekinthetjük, hogy mindez miként történik.



1.4. ábra

Egyidejű értéknövelés és értékadás változtatható objektum esetében

Ha már a Python nyelvtana okosan elrejti az állandó és a változó értékű adattípusokon alapuló objektumok kezelésének különbségeit, miért van egyáltalán szükség kétféle típusra? Az okot főként a programok teljesítményének alakulásában kell keresnünk. Az állandó értékű típusok megvalósítása lényegesen hatékonyabb lehet (éppen azért, mert sosem változnak), mint a változó értékűeké. Sőt, néhány gyűjteményt megvalósító adattípus, például a halmaz, csak állandó értékű objektumokkal dolgozik. Másfelől azonban a változtatható típuson alapuló objektumok használata kényelmesebb lehet. Ahol számít a kétféle típus közötti különbség, ott megbeszéljük a témát – például a 4. fejezetben, ahol az egyedi függvények alapértelmezett paramétereiről lesz szó, a 3. fejezetben, ahol a listákat, halmazokat és egyéb adatszerkezet-típusokat tárgyaljuk, és persze a 6. fejezetben, ahol az egyedi adattípusok létrehozását mutatjuk be.

Listák esetében a `+=` műveleti jel jobb oldalán egy bejárható típusba tartozó objektumnak kell állnia. Ha nem így lenne, kivétel keletkezik:

```
>>> seeds += 5
Traceback (most recent call last):
...
TypeError: 'int' object is not iterable
```

Egy lista bővítésére a megfelelő módszer az, ha bejárható objektumot használunk, mondjuk egy listát:

```
>>> seeds += [5]
>>> seeds
['sesame', 'sunflower', 'pumpkin', 5]
```

Természetesen a lista növelésére használt bejárható objektum több elemet is tartalmazhat:

```
>>> seeds += [9, 1, 5, "poppy"]
>>> seeds
['sesame', 'sunflower', 'pumpkin', 5, 9, 1, 5, 'poppy']
```

Ha a lista végéhez egyszerű karakterláncot fűzünk – például azt, hogy `"durian"` –, és nem egy karakterláncot tartalmazó listát – vagyis azt, hogy `["durian"]` –, akkor meglepő, bár végső soron logikus eredményt kapunk:

```
>>> seeds = ["sesame", "sunflower", "pumpkin"]
>>> seeds += "durian"
>>> seeds
['sesame', 'sunflower', 'pumpkin', 'd', 'u', 'r', 'i', 'a', 'n']
```

Listák esetében a += művelet hatására úgy történik meg a növelés, hogy a műveleti jel jobb oldalán szereplő, bejárható objektum egyes elemei a lista végére kerülnek. Minthogy a karakterláncok bejárható objektumok, a karakterlánc minden egyes karaktere külön elemként kerül a lista végére. Ha a lista `append()` tagfügvényét használjuk, a paraméter mindig egy elemként kerül a lista végére.

7. kulcsösszetevő: Bemenet és kimenet

Ha remekül használható programokat szeretnénk írni, meg kell oldanunk az adatok bevitelét – például a felhasználótól egy konzolon, vagy fájlokból –, és a kimenet kiírását (konzolra vagy fájlokba). Már használatba vettük a Python nyelvbe beépített `print()` függvényt, bár a teljeskörű tárgyalását a 4. fejezetig halogatjuk. Ebben a részben a konzolon megvalósuló bemenetre, illetve kimenetre összpontosítunk, állományok olvasására és írására pedig a parancshéjban lévő átirányítási lehetőséget fogjuk használni.

A felhasználótól adatokat bekérni a Python nyelv beépített `input()` függvényével tudunk. A függvény elhagyható paramétere egy karakterlánc, aminek a tartalmát a konzolon jeleníti meg. A függvény a hívását követően arra vár, hogy a felhasználó begépelje a kért adatot, és lenyomja az ENTER (RETURN) billentyűt. Az `input()` függvény üres karakterláncot ad vissza, ha a felhasználó anélkül nyomja le az ENTER-t, hogy bármit beírt volna. Egyéb esetekben a felhasználó által beírtakat tartalmazó karakterláncot adja vissza a függvény, mégpedig a sor végét jelző karakterek nélkül.

Íme az első teljes, „használható" programunk, amely az előző pontokban megszerzett ismereteink közül sokat hasznosít, és egyedül az `input()` függvény lehet benne újdonság.

```
print("Type integers, each followed by Enter; or just Enter to finish")
total = 0
count = 0
while True:
    line = input("integer: ")
    if line:
        try:
            number = int(line)
        except ValueError as err:
            print(err)
            continue
        total += number
        count += 1
    else:
        break
```

```
if count:
print("count =", count, "total =", total, "mean =", total / count)
```

A program (amelyet a könyv példái között `sum1.py` néven találunk meg) mindössze 17 végrehajtható sorral bír. Futtatásakor az alábbihoz hasonló képet kapunk:

```
Type integers, each followed by Enter; or just Enter to finish
number: 12
number: 7
number: 1x
invalid literal for int() with base 10: '1x'
number: 15
number: 5
number:
count = 4 total = 39 mean = 9.75
```

A könyv példái
Bevezetés ☜

Egy nagyon rövid, mégis meglehetősen hatékony programot alkottunk. Ha a felhasználó egész számmá nem alakítható karakterláncot ad meg, a problémát a képernyőre megfelelő hibaüzenetet író, a vezérlést pedig a ciklus elejének átadó kivételkezelővel kezeljük – vagyis „folytatjuk a ciklust". Az utolsó `if` kifejezés annak biztosítására szolgál, hogy ha a felhasználó egyáltalán nem ír be számot, akkor ne írjuk ki a szokásos összefoglalót, és ezzel elkerüljük a nullával való osztást.

A fájlkezelést teljes részletességgel a 7. fejezetben ismertetjük, de addig is képesek vagyunk fájlok létrehozására, mégpedig a `print()` függvény kimenetének a konzolból történő átirányításával. Például a

```
C:\>test.py > results.txt
```

utasítás a képzeletbeli `test.py` programban előforduló `print()` függvényhívások kimenetét a `results.txt` fájlba irányítja. A fenti utasításforma a Windows és Unix operációs rendszerek konzoljában is működik. Ha a gépen a Python 2 az alapértelmezett Python-változat, Windowson a parancsot a `C:\Python30\python.exe test.py > results.txt` alakban kell kiadnunk, és a `python test.py > results.txt` alakot kell használnunk, ha a `PATH` környezeti változóban a Python 3 található meg hamarabb (de ezt most mondjuk el utoljára). Unix rendszereken először futtathatóvá kell tennünk a programot (`chmod +x test.py`), ezt követően pedig a `./test.py` parancs kiadásával indíthatjuk el – hacsak a programot tartalmazó könyvtár nincs benne a `PATH` környezeti változóban.

Az adatok beolvasása úgy is megvalósítható, hogy az adatokat tartalmazó fájlt a kimenet átirányításának mintájára átirányítjuk. Mindazonáltal, ha a `sum1.py` program esetében átirányítást használnánk, a program hibásan futna. Ennek az az oka, hogy az `input()` függvény kivételt hoz létre, amennyiben EOF (end of file, vagyis fájl vége) karaktert kap bemenetül. Íme egy, az előzőnél megbízhatóbb programváltozat, a `sum2.py`, amely már egyaránt képes a felhasználó billentyűzetéről és fájlátirányításból származó bemenet kezelésére:

```
print("Type integers, each followed by Enter; or ^D or ^Z to finish")
total = 0
count = 0
while True:
    try:
        line = input()
        if line:
            number = int(line)
            total += number
            count += 1
    except ValueError as err:
        print(err)
        continue
    except EOFError:
        break
if count:
    print("count =", count, "total =", total, "mean =", total / count)
```

A `sum2.py < data\sum2.dat` parancssor esetében (ahol a `sum2.dat` fájl soronként egyetlen számot tartalmaz, maga a fájl pedig a példák `data` alkönyvtárában található) a konzolon a következő kimenet jelenik meg:

```
Type integers, each followed by Enter; or ^D or ^Z to finish
count = 37 total = 1839 mean = 49.7027027027
```

Számos kis módosítást végeztünk annak érdekében, hogy a program mind interaktívan, mind átirányítással kényelmesen használható legyen. Először is megváltoztattuk a kilépés feltételét: most már nem üres sort, hanem EOF karaktert (Unixokon CTRL+D, Windowson CTRL+Z vagy ENTER) vár a program. Ez megbízhatóbbá teszi a programot olyan esetekben is, amikor a bemeneti állomány üres sorokat tartalmaz. A továbbiakban nem írjuk ki a készenléti jelet minden egyes szám előtt, hiszen értelmetlen, ha a bemenetet fájlból kapjuk. Mindezen felül egyetlen `try`-blokkot használunk, két kivételkezelővel.

Figyeljük meg, hogy amennyiben érvénytelen bemenet érkezik (vagy a billentyűzetről, vagy az átirányított bemenő fájl egy „rossz" sorából), az `int()` függvény az átalakítást megkísérelve `ValueError` kivételt vált ki, és a folyamatvezérlés azonnal átkerül a megfelelő `except` blokkhoz. Mindez annyit tesz, hogy érvénytelen adat esetén sem a `total`, sem a `count` változó értéke nem nő – és éppen ezt szerettük volna elérni.

Ugyanilyen egyszerű lett volna két önálló kivételkezelő blokk használata:

```
while True:
    try:
        line = input()
        if line:
```

```
            try:
                number = int(line)
            except ValueError as err:
                print(err)
                continue
            total += number
            count += 1
    except EOFError:
        break
```

Végül mégis a kivételkezelések csoportba foglalása mellett döntöttünk, így ugyanis a fő feldolgozó rész egységesebb képet mutat.

8. kulcsösszetevő: Függvények meghatározása és meghívása

Minden további nélkül lehetséges olyan programok írása, amelyek csak az előzőekben ismertetett adattípusokat és vezérlési szerkezeteket tartalmazzák. Ugyanakkor gyakran fordul elő, hogy ugyanazt a feldolgozási folyamatot szeretnénk újra meg újra megismételni, valami pirinyó eltéréssel – mondjuk eltérő kiindulási értékkel. A Python nyelvben létezik megoldás a kódblokkok egységbe zárhatóságára – mégpedig az átadott paraméterek segítségével különböző kiindulási értékekkel végrehajtott függvények formájában. A függvények általános formája a következő:

```
def függvényNév(paraméterek):
    kódblokk
```

A *paraméterek* elhagyhatók, több paraméter egymástól való elválasztása pedig vesszők használatával történik. A Python nyelvben minden függvénynek van visszatérési *értéke*, ez alapértelmezés szerint a `None` érték. Ha a függvényből a `return` *érték* utasítással térünk vissza, akkor az *érték*-et adja vissza a függvény. A visszaadott érték lehet egyszerű érték, vagy értékekből álló rendezett sorozat. A hívónak nem kötelező figyelembe vennie a visszaadott értéket, ilyenkor az egyszerűen elvész.

`return`
☞ 157, 169

Figyeljük meg, hogy a `def` utasítás hasonlóan viselkedik, mint az értékadó művelet. A `def` utasítás végrehajtásakor függvényobjektum jön létre, és a függvénynévként megadott nevet a függvényobjektumra mutató objektumhivatkozás kapja. Lévén a függvények is objektumok, elhelyezhetők gyűjtemények tárolására létrehozott objektumokban, illetve átadhatók más függvények paraméteréül, ahogy azt a későbbi fejezetekben látni fogjuk.

Interaktív, konzolos alkalmazások írása során gyakran merül fel az igény egész szám bekérésére a felhasználótól. Az alábbi függvény éppen ezt a feladatot látja el:

```
def get_int(msg):
    while True:
        try:
```

```
            i = int(input(msg))
            return i
        except ValueError as err:
            print(err)
```

A függvény egyetlen paramétert vár, az `msg` (message, vagyis üzenet) nevűt. A `while`-ciklus belsejében a felhasználót felszólítjuk egy egész szám megadására. Ha a felhasználó érvénytelen adatot ír be, `ValueError` kivétel keletkezik, hibaüzenet kerül a képernyőre, és a ciklus újra kezdődik. Ha érvényes egész számot kap a függvény, akkor a kapott számot visszaadja a hívójának. A függvény az alábbi módon hívható meg:

```
age = get_int("enter your age: ")
```

A példában szereplő függvény egyetlen paramétere nem elhagyható, más néven kötelező paraméter, ugyanis nem adtunk meg helyette alapértelmezett értéket. Az a helyzet, hogy a Python igen kifinomultan és rugalmasan, az alapértelmezett, a hely szerinti és a kulcsszóval azonosított paramétereket egyaránt kezelve támogatja a függvények írását. A függvények nyelvtanát a 4. fejezet tárgyalja kimerítően.

Bár saját függvényeket írni felemelő érzés, nem minden esetben szükséges. Ennek oka abban keresendő, hogy a Python sok beépített függvénnyel bír, az alapkönyvtárában lévő modulok pedig ennél is többet tartalmaznak, így előfordulhat, hogy a megírni szándékozott függvény már a rendelkezésünkre áll.

A Python nyelv moduljai nem egyebek, mint `.py` kiterjesztésű, Python nyelvű kódot – egyedi függvények és osztályok (egyedi adattípusok) meghatározásait, és néha változókat – tartalmazó állományok. A modulokban tárolt képességekkel úgy ruházható fel a programunk, hogy a modult betöltjük a programba, például így:

```
import sys
```

A modul betöltését az `import` utasítással végezzük el, amit a `.py` kiterjesztésű fájl neve követ, mégpedig a kiterjesztés nélkül.* A modul betöltését követően a modulban tárolt valamennyi függvény, osztály és változó használható. Például:

```
print(sys.argv)
```

A `sys` modulban található az `argv` változó, vagyis egy olyan lista, amelynek első eleme az a név, amellyel a programot indítottuk, a második és az azt követő elemei pedig a program parancssori paramétereit tartalmazzák. Az előzőekben szereplő két sornyi kód alkotja az `echoargs.py` program teljes egészét. Ha a programot az `echoargs.py -v` parancssorral indítjuk, akkor az `['echoargs.py', '-v']` üzenetet fogja a képernyőre írni. (Unixon az első érték esetleg `'./echoargs.py'` lesz.)

---

* A sys és néhány egyéb beépített modul, valamint a C nyelven írt modulok esetében nem feltétlenül van meg a megfelelő, `.py` kiterjesztésű fájl – de ugyanúgy használjuk őket, mint azokat, amelyeknél megvan.

A modulokban használt függvényekre vonatkozó utasításforma *`modulNév.függvényNév(paraméterek)`* alakú. Az utasításforma a *Gyűjteményes adattípusok* című részben már bemutatott pont (`.`) műveleti jelre épül. Az alapkönyvtár (standard library) rengeteg modulból áll, amelyekből a könyv során sokat fogunk használni. Az alapkönyvtár moduljainak kisbetűvel írt nevük van, a programozók ezért a saját moduljaik nevében lévő szavakat nagybetűvel szokták kezdeni (például: `KedvencModulom`), hogy könnyebb legyen őket megkülönböztetni.

A pont (.) műveleti jel 14 ☜

Nézzünk meg egy példát, a `random` (véletlen) modult (az alapkönyvtár `random.py` állományát), amely sok hasznos függvényt ad a kezünkbe:

```
import random
x = random.randint(1, 6)
y = random.choice(["apple", "banana", "cherry", "durian"])
```

A kifejezések végrehajtását követően az `x` objektum egy 1 és 6 közötti egész számot fog tartalmazni, az `y` objektum tartalma pedig a `random.choice()` függvénynek átadott karakterláncok egyike lesz.

shebang-sor (#!) 5 ☜

Az `import` utasítások a Python nyelvű fájlokban hagyományosan elöl, a *shebang* sor és a modul dokumentációja után kapnak helyet. (A modulok dokumentációjával az 5. fejezet foglalkozik.)

Azt tanácsoljuk, hogy elsőként az alapkönyvtárból származó modulok betöltését végezzük el, utána következzenek a mások által készített, „külső" modulok, és a saját moduljaink betöltése maradjon utolsónak.

Példák

Az előzőekben eleget megtanultunk a Python nyelvről ahhoz, hogy igazi programok írásába fogjunk. Ebben a részben két olyan program tanulmányozásával folytatjuk a tanulást, amelyek a Python nyelvből csak az eddig megtanultakat használják. Mindkét program elsődleges feladata a lehetőségek bemutatása, és az eddig megtanultak rendszerezésének elősegítése.

A következő fejezetekben egyre többet fogunk megismerni a Python nyelvből, illetve a Python nyelv alapkönyvtárából, képessé válva az itt bemutatottaknál tömörebb és hatékonyabb programok írására – de először el kell sajátítanunk a későbbi építmények alapjául szolgáló ismereteket.

bigdigits.py

Az első megvizsgált program meglehetősen rövid, bár van benne egy-két érdekesség, többek között egy listákat tartalmazó lista. Hogy mit csinál a programunk? Megadunk egy számot a parancssorban, és ugyanazt a számot visszaírja a képernyőre, de „nagy" számokkal.

Azokon a számítógépes munkahelyeken, ahol sok felhasználó dolgozott egyetlen nagy sebességű sornyomtatón osztozva, általánosan elterjedt gyakorlat volt, hogy az egyes felhasználók által kiküldött nyomtatási feladatokat a nyomtató egy olyan fedőlappal választotta el, amelyen a most bemutatásra váró módszerrel volt feltüntetve a felhasználónév és pár egyéb azonosító adat.

A programkódot három részletben fogjuk áttekinteni: elsőként az adatbevitelt, másodikként a program által használt adatokat tartalmazó listák kialakítását, végül magát a kiírást. De mindenekelőtt lássunk egy példát a program futtatására:

```
bigdigits.py 41072819
   *     *     ***   *****  ***    ***    *    ****
  **    **    *   *      * *  *  *  *  * **   *    *
 * *     *   *     *    *   *  *  *  *  *  *   *    *
*  *     *   *     *   *    *  *  *  *  *  *   *    *
*  *     *   *     *   *      *     *  *  *   *    *
****** *   *     *   *      *     *  *  *  *       *
   *   *   *   *   *      *     *  *  *  *       *
   *  ***   ***   *      *****  ***  ***       *
```

Nem tüntettük fel a konzol készenléti jelét (illetve a Unixokat használók esetében a ./ jelpárt sem), és mostantól kezdve a későbbiekben is eltekintünk ettől.

```
import sys
```

Minthogy a parancssorból kell beolvasnunk egy paramétert (a kiírni kívánt számot), tudnunk kell olvasni a `sys.argv` listából. Ezért a munkát a `sys` modul betöltésével kezdjük.

Minden számot karakterláncokból álló listaként ábrázolunk. A nullát például így:

```
Zero = ["  *** ",
        " *   * ",
        "*     *",
        "*     *",
        "*     *",
        " *   * ",
        "  *** "]
```

a `set` típus
☞ 115

a `dict` típus
☞ 122

Időzzünk el kicsit ennél a `Zero` nevű listánál, amely ugyebár karakterláncokat tartalmaz, és több sorra is kiterjed. A Python nyelv utasításai a legtöbb esetben egyetlen sorba kerülnek, de amennyiben zárójelezett kifejezésről van szó – tehát listáról, halmazról, szótár típusú objektumban tárolt szövegről, függvényhívás paraméterlistájáról, vagy olyan többsoros kifejezésről, amelyben az utolsó kivételével minden sor fordított perjellel (\) zárul –, több sort is elfoglalhatnak. A felsorolt esetek mindegyikében annyi sort foglalhat el a kifejezés, amennyire csak szükség van, a behúzások pedig a kifejezés második sorától kezdve nem számítanak.

Minden számot egy hét karakterláncból álló listával ábrázolunk, a karakterláncok pedig egy számon belül egyforma szélességűek. A nullát követő számok listái is a már megismert módon épülnek fel, bár ezeknél a tömör elrendezést az olvashatóságnál fontosabb szempontként kezeltük.

```
One = [" * ", "** ", " * ", " * ", " * ", " * ", "***"]
Two = [" *** ", "*  *", "  * ", " *  ", "*   ", "*   ", "****"]
# ...
Nine = [" ****", "*  *", "*  *", " ****", "   *", "   *", "   *"]
```

Az utolsó szükséges adat a számjegyek listáit tartalmazó lista:

```
Digits = [Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine]
```

A `Digits` listát közvetlenül, további változók igénybe vétele nélkül is létrehozhattuk volna, a következőképpen:

```
Digits = [
    [" ***  ", "*   *", "*    ", "* **", "* **",
     "*   *", " ***  "], # Zero
    [" * ", "** ", " * ", " * ", " * ", " * ", "***"], # One
    # ...
    [" ****", "*  *", "*  *", " ****", "   *", "    *",
     "   *"] # Nine
    ]
```

Annak oka, hogy mindezek ellenére mégis úgy döntöttünk, hogy minden számhoz külön változót használunk, a megértés könnyítése, illetve az így kapott programkód szebb látványa volt.

A kód fennmaradó részét egyszerre mutatjuk be, így a kedves Olvasó a kódot követő magyarázat elolvasása előtt áttekintheti a program működését.

```
try:
    digits = sys.argv[1]
    row = 0
    while row < 7:
        line = ""
        column = 0
        while column < len(digits):
            number = int(digits[column])
            digit = Digits[number]
            line += digit[row] + " "
            column += 1
        print(line)
        row += 1
except IndexError:
    print("usage: bigdigits.py <number>")
except ValueError as err:
    print(err, "in", digits)
```

A kód egésze a kivételkezelőn belül kapott helyet, így annak módjában áll elbánni azzal a két dologgal, ami hibát okozhat. Elsőként a parancssori paraméter kinyerését végezzük el. A `sys.argv` sorszámozása a Python nyelv többi listájához hasonlóan nullával kezdődik. A nulla sorszámú helyen tehát a program indításához használt név található, így futás közben a listának legalább egy eleme biztosan van. Ha a felhasználó nem adott meg paramétert, akkor olyan listából fogjuk megkísérelni a második elem olvasását, amelyben csak egyetlen elem található, ez pedig `IndexError` kivétel létrejöttéhez fog vezetni. Ebben az esetben a folyamatvezérlést azonnal a megfelelő kivételkezelő kódblokk kapja meg, ahol egyszerűen kiírunk egy üzenetet a program használatának módjáról. A program végrehajtása ezt követően a `try` blokk végén folytatódik, és minthogy ott programkód nem található, a program futása egyszerűen megszakad.

Ha nem következik be `IndexError` kivételhez vezető esemény, a parancssori paraméter – ami remélhetőleg számjegyek sorozata – a `digits` karakterláncba kerül. (*Az Objektumhivatkozások* című részből emlékezhetünk rá, hogy az azonosítókban megkülönböztetjük a kis- és nagybetűket, a `digits` és a `Digits` név tehát két különböző azonosítót jelöl.) Minden nagy számjegyet hét karakterlánc ábrázol, így aztán a szám helyes megjelenítése úgy oldható meg, hogy először kiírjuk az összes szám első sorát, majd a másodikat, és ezt folytatjuk egészen addig, amíg mind a hét sor a képernyőre kerül. Az egyes sorok kiíratását egy `while` ciklussal végezzük el. Ugyanilyen egyszerű lett volna a `for row in (0, 1, 2, 3, 4, 5, 6):` kifejezés használata, a későbbiekben pedig egy lényegesen jobb, a `range()` nevű beépített függvény használatán alapuló módszert is meg fogunk ismerni.

`range()`
☞ 136

A `line` nevű karakterláncot használjuk a számjegyekből származó sorok tartalmának tárolására. A ciklus az oszlopokon, vagyis a parancssori paraméterben soron következő karaktereken halad végig. A `digits[column]` kifejezés adja vissza az egyes karaktereket, amelyeket egész számmá alakítunk, és a `number` nevű változóban tárolunk. Ha az átalakítás sikertelen, `ValueError` kivétel keletkezik, és a folyamatvezérlést azonnal a megfelelő kivételkezelő kódblokk kapja meg. Ilyenkor kiírunk egy hibaüzenetet, majd a program végrehajtása a `try` blokk végén folytatódik. Ahogy korábban már rámutattunk, ott programkód nem található, így a program futása egyszerűen megszakad.

Ha az átalakítás sikeres volt, a `number` változó értékét a `Digits` lista sorszámául fogjuk használni, és kinyerjük belőle a `digit` nevű, karakterláncokból álló lista értékét. Ennek a listának a `row`-adik karakterláncát hozzáfűzzük az éppen kialakításban lévő sor végéhez, és utána elhelyezünk két szóközt is, hogy a számjegyeket egymástól vízszintesen elválasszuk.

Minden alkalommal, amikor a belső `while` ciklus véget ér, kiírjuk a felépített sort. A program megértésének kulcsa abban a részben rejlik, ahol az egyes számjegyek adott sorát tartalmazó karakterláncot a kialakítás alatt lévő sor végéhez illesztjük. A programot futtatva kicsit megízlelhetjük a dolgokat. A gyakorlatok során még visszatérünk ehhez a programhoz, és kicsit javítunk a kimenetén.

generate_grid.py

A programozói munka során időről időre felmerül az igény tesztadatok előállítására. A feladat elvégzésére nincs általánosan használható alkalmazás, hiszen a tesztadatok iszonyató mértékben különbözőek az egyes programoknál. Tesztadatok előállítására gyakran használják a Pythont, lévén a Python nyelvű programok megírása és átalakítása igen egyszerű dolog. A most következő részben véletlenszámokként előállított egészekből hozunk létre mátrixot. A sorok és oszlopok kívánt számát, valamint a véletlenszámok alsó és felső határát a felhasználó adhatja meg. Kezdetnek vessünk egy pillantást a program futtatásának egy lehetséges eredményére:

```
generate_grid.py
rows: 4x
invalid literal for int() with base 10: '4x'
rows: 4
columns: 7
minimum (or Enter for 0): -100
maximum (or Enter for 1000):
      554       720       550       217       810       649       912
      -24       908       742       -65       -74       724       825
      711       968       824       505       741        55       723
      180       -60       794       173       487         4       -35
```

A program interaktív, és a futás elejénél, a sorok megadásakor gépelési hibát vétettünk. A program hibaüzenet kiírásával válaszolt, majd megkért bennünket, hogy adjuk meg újra a sorok számát. A maximális érték megadásakor az ENTER billentyűvel elfogadtuk az alapértelmezett értéket.

A kód áttekintését négy lépésben végezzük el: elsőként a betöltést, másodikként a `get_int()` függvény (itt a *Függvények meghatározása és meghívása* című részben közöltnél kifinomultabb változatot találunk) megírását, harmadikként a felhasználói interakciót, illetve a használandó értékek bekérését, végül pedig magát az adatfeldolgozást vizsgáljuk meg.

```
import random
```

`random.randint()` 33 ☜

A `random` modulra azért van szükségünk, hogy használhassuk a `random.randint()` függvényt.

```
def get_int(msg, minimum, default):
    while True:
        try:
            line = input(msg)
            if not line and default is not None:
                return default
            i = int(line)
            if i < minimum:
```

```
            print("must be >=", minimum)
        else:
            return i
    except ValueError as err:
        print(err)
```

A függvény három paramétert vár: egy karakterláncot az üzenet szövegével, egy minimális és egy alapértelmezett értéket. Ha a felhasználó egyszerűen lenyomja az ENTER billentyűt, akkor két dolog történhet. Ha a `default` (alapértelmezett) nevű változó értéke a `None` megkülönböztetett szerepű objektum, vagyis nem adtunk át alapértelmezett értéket a függvénynek, a folyamatvezérlés eljut az `int()` függvényt tartalmazó sorhoz, majd az átalakítás hibába ütközik (lévén a " üres karakterlánc nem alakítható át egész számmá), és `ValueError` kivétel lép fel. Amennyiben a `default` nevű változó értéke nem `None`, akkor azt az értéket fogjuk visszaadni. Egyéb esetekben a függvény meg fogja kísérelni a felhasználó által megadott adat egész számmá való alakítását, és az átalakítás sikere esetén ellenőrizni fogja, hogy az egész szám eléri-e a megadott minimális értéket.

Így a függvény mindig visszaad egy számot: vagy a `default` változó értékét, ha a felhasználó egyszerűen lenyomta az ENTER billentyűt, vagy egy érvényes, a megadott minimális értéket elérő egész számot.

```
rows = get_int("rows: ", 1, None)
columns = get_int("columns: ", 1, None)
minimum = get_int("minimum (or Enter for 0): ", -1000000, 0)

default = 1000
if default < minimum:
    default = 2 * minimum
maximum = get_int("maximum (or Enter for " + str(default) + "): ",
                  minimum, default)
```

A `get_int()` függvényünk használatával egyszerű a felhasználó által kért sor- és oszlopszámnak, illetve a véletlenszám minimális értékének a bekérése. A sorok és oszlopok esetében a `None` alapértelmezett értéket adjuk meg, vagyis nincs alapértelmezett érték, és a felhasználónak meg kell adnia egy-egy egész számot. A minimum esetében 0, a maximum esetében pedig 1000 lesz az alapértelmezett érték – illetve a minimum értékének kétszerese, olyan esetekben, amikor a minimum legalább 1000.

Ahogy az előző példa ismertetésekor rámutattunk, a függvényhívás paraméterlistája akárhány sorra kiterjedhet, a második sortól kezdődően pedig nincs jelentősége a behúzásnak.

Ha végre tudjuk, hogy a felhasználó hány sort és oszlopot szeretne, illetve ismerjük a kívánt véletlenszámok minimális és maximális értékét, készen állunk a szükséges számítások elvégzésére.

```
row = 0
while row < rows:
    line = ""
    column = 0
    while column < columns:
        i = random.randint(minimum, maximum)
        s = str(i)
        while len(s) < 10:
            s = " " + s
        line += s
        column += 1
    print(line)
    row += 1
```

A mátrix kialakításához három `while` ciklust használunk. Az egyik a sorokon, a második az oszlopokon, a harmadik pedig a karaktereken halad végig. A középső ciklusban előállítunk egy véletlenszámot, majd karakterlánccá alakítjuk. A belső `while` ciklus feladata, hogy a karakterláncokat a lánc elejére szóköz karaktereket illesztve tíz karakter hosszúságúvá alakítsa. A `line` nevű karakterláncot használjuk az egyes sorokba kerülő számok összegyűjtésére – a megjelenítésére akkor kerül sor, ha már az utolsó oszlop számát is elhelyeztük benne. Ezzel végére értünk a második példának.

A Python nyelv igen kifinomult képességeket tud felmutatni a karakterláncok formázásának terén, és remek támogatással rendelkezik a `for...in` ciklusok támogatottságát illetően is. Mind a `bigdigits.py`, mind a `generate_grid.py` programok `for...in` ciklusokat használnának a valós életben, a szóközökkel való berhelés helyett pedig a Python nyelv karakterlánc-formázó képességeit aknáznánk ki. Egyelőre azonban tartottuk magunkat a fejezetben elsajátítottak használatához – ennyi is elég teljesen használható programok írásához. A következő fejezetekben a Python nyelv újabb és újabb elemeit sajátítjuk majd el, vagyis a könyvben előrehaladva egyre kifinomultabb programokat ismerhetünk meg, és programírói képességeink sem maradnak meg a mostani szinten.

Összefoglalás

`str.format()`
☞ 75

Ebben a fejezetben megtanultuk a Python nyelvű programok szerkesztésének és futtatásának módját, valamint áttekintettünk néhány rövid, de mégis teljes programot. A fejezet nagy részét a „Python gyönyörű szíve” megismerésének szenteltük – és ezzel eléggé megismertük a nyelvet ahhoz, hogy igazi programokat írhassunk.

A munkát a Python nyelv két alapvető adattípusának, az `int`-nek és az `str`-nek a megismerésével kezdtük. Az egész adatok jelölése a legtöbb egyéb programozási nyelvhez hasonlóan történik. A karakterlánc típusú értékek aposztrófok vagy idézőjelek közé kerülnek – mindegy melyiket használjuk, amennyiben az érték mindkét végén ugyanazt használjuk.

Karakterláncok és egészek között lehetséges az átalakítás, például az `int("250")` és az `str(125)` utasításokkal. Ha egy egész számmá alakítás sikertelen, `ValueError` kivétel lép fel – karakterlánccá szinte bármi átalakítható.

A karakterláncok sorozatok, vagyis a sorozatokkal egyébként használható függvények és műveletek itt is működnek. Lehetséges például egy adott karakter elérése, a sorozatok elemeinek elérésére használatos, szögletes zárójeleket (`[]`) formázó műveleti jellel. Karakterláncok összefűzése a `+`, egyik karakterlánc hozzáfűzése a másikhoz pedig a `+=` műveleti jel használatával történik. Minthogy a karakterláncokat tároló objektumok állandó értékűek, két karakterlánc összefűzésekor a színfalak mögött új, a két adott karakterláncot összefűzve tartalmazó karakterlánc jön létre, a műveleti jel bal oldalán álló objektumhivatkozás pedig az eredményt tartalmazó karakterláncot kapja meg új célként.
A karakterláncok egy `for...in` ciklussal be is járhatók (karakterenként), a `len()` nevű beépített függvény pedig a karakterláncban lévő karakterek számának kiderítésére használható.

Az olyan állandó értékű objektumok esetében, mint a karakterláncok, az egész számok és a rendezett sorozatok (tuple), a programozás szempontjából az objektumhivatkozások változóknak tekinthetők, mintha az objektumhivatkozás és az általa hivatkozott objektum egyazon dolog volna. A probléma a változtatható (mutable) objektumok esetében is kezelhető így, de ilyenkor az objektum bármilyen megváltozása az objektum minden előfordulását – a rá mutató valamennyi objektumhivatkozást – érinteni fogja. A kérdéskörre a 3. fejezetben térünk vissza.

A Python nyelv alapkönyvtára számos beépített, a gyűjtemények kezelését megvalósító és néhány egyéb adattípussal rendelkezik. Itt a lista és a rendezett sorozat típusokat ismertük meg, különös tekintettel arra, hogy miként lehetséges egyszerű adatokból listákat létrehozni (például: `even = [2, 4, 6, 8]`). A listák, mint a Pythonban minden egyéb dolog, objektumok. Meghívhatjuk tehát a tagfüggvényeiket: az `even.append(10)` hívás például tíz újabb elemmel bővíti a listát. A listák és a rendezett sorozatok is a karakterláncokhoz hasonló sorozatok, vagyis egy `for...in` ciklussal elemenként bejárhatók, a `len()` nevű függvénnyel pedig megismerhetjük az elemszámukat. A listák és a rendezett sorozatok adott elemét is a szokásos, szögletes zárójeleket (`[]`) formázó műveleti jellel kaphatjuk meg. Összefűzésük a `+`, az egyik hozzáfűzése a másikhoz pedig a `+=` műveleti jel használatával történik. Ha egy lista végére egyetlen elemet szeretnénk hozzáfűzni, akkor vagy a `list.append()` tagfüggvényt, vagy a `+=` műveleti jelet kell használnunk – de utóbbi esetben a hozzáfűzni kívánt elemet előbb egyelemű listává kell alakítanunk, például így: `even += [12]`. Minthogy a listák változtatható objektumok, a `[]` műveleti jellel egyetlen elem megváltoztatására is van lehetőség: `even[1] = 16`.

A gyors működésű `is` és `is not` műveletek használhatók annak ellenőrzésére, hogy két objektumhivatkozás ugyanarra az objektumra hivatkozik-e – ez különösen akkor hasznos, ha az ellenőrzést a különleges szerepű, beépített `None` objektumhoz viszonyítva kérjük.

Használhatjuk valamennyi megszokott összehasonlító műveletet, (`<`, `<=`, `==`, `!=`, `>=`, `>`), de csak egymással összehasonlítható adattípuson alapuló objektumok esetében, és csak olyankor, ha az adott művelet támogatott. Az eddig megismert adattípusok – `int`, `str`, `list`, és `tuple` – közül az összes támogatja az összehasonlító műveletek mindegyikét. Az egymással össze nem hasonlítható – például `int` és `str`, illetve `int` és `list` típusú – objektumok összehasonlítása azonnal `TypeError` kivételhez vezet.

A Python nyelv támogatja a szokványos logikai műveleteket: az `and`-et, az `or`-t és a `not`-ot. Az `and` és az `or` műveletek egyaránt szeretik rövidre zárni a dolgokat, és azonnal a művelet eredményét meghatározó tényezőt adják vissza – ami esetleg nem a logikai típusba tartozik (bár átalakítható logikai típusúvá). A `not` művelet eredménye minden esetben `True` vagy `False`.

Az `in` és a `not in` művelettel az ellenőrizhető, hogy egy objektum tagja-e egy másik, sorozat típusú objektumnak, például karakterláncnak, listának vagy rendezett sorozatnak. A tagság ellenőrzése listák és rendezett sorozatok esetében lassú lineáris keresést használ, karakterláncok esetében pedig egy esetleg lényegesen gyorsabb keverék algoritmust – a művelet sebessége azonban a nagyon hosszú karakterláncok, listák és rendezett sorozatok kivételével ritkán jelent gondot. A 3. fejezetben megismerjük a Python nyelv társításos tömb és halmaz gyűjteményes adattípusait, amelyek esetében egyaránt igen gyorsan lehetséges a tagsági viszony ellenőrzése. Van lehetőség annak megállapítására is, hogy egy adott változó (pontosabban: objektumhivatkozás által hivatkozott objektum) melyik típusba tartozik, de a `type()` függvényt a legtöbb esetben csak hibakereséskor és teszteléskor használjuk.

A Python nyelv számos vezérlési szerkezetet ismer. Ide tartozik az `if...elif...else` kifejezéssel megvalósított feltételes elágazás, a `while` utasításon alapuló feltételes, illetve a `for...in` kifejezéssel működő, sorozatokat bejáró ciklusszervezés, valamint a `try...except` blokkot használó kivételkezelés. A `break` utasítással mind a `while`, mind a `for...in` ciklus idő előtt megszakítható, a `continue` utasítással pedig mindkettő esetében a ciklus elejére helyezhető a vezérlés.

Támogatott a szokásos számtani műveletek (`+`, `-`, `*` és `/`) használata, bár a Python nyelv viselkedése szokatlan, amennyiben az osztás eredménye két egész (számként ábrázolt) tényező esetében is lebegőpontos lesz. (A sok más nyelvben használt csonkoló osztás a Python nyelvben a // műveleti jellel valósul meg.) A Python nyelvben megtalálhatók az értéknövelést és értékadást egy lépésben megvalósító műveletek (`+=` és `*=`) is; amennyiben a műveleti jel bal oldalán lévő objektum állandó értékű, akkor az említett művelet a színfalak mögött új objektumot hoz létre, és elvégzi az objektumhivatkozás megváltoztatását. Ahogy korábban már jeleztük, az `str` és a `list` típusok esetében a számtani műveletek túlterheltek.

A konzolról adatokat az `input()` függvénnyel kaphatunk, adatok kiírásához pedig a `print()` függvény használható. Konzolon dolgozva a fájlokból való olvasás és a fájlokba való írás ugyanezekkel a beépített függvényekkel, valamint a fájlok átirányításával oldható meg.

A Python gazdag beépített képességein felül használhatjuk az ugyancsak kiterjedt tudású alapkönyvtárát is, amelynek a moduljait használat előtt be kell töltenünk a programunkba. A `sys` modul betöltésére általában sor kerül, és ebben a modulban találjuk meg a parancssori paramétereket tároló `sys.argv` listát. Amikor pedig a Python nem rendelkezik a szükséges függvénnyel, könnyedén megírhatjuk a nekünk pontosan megfelelőt – ehhez a `def` kifejezést kell használnunk.

A fejezetben leírtak felhasználásával lehetséges rövid, ámde hasznos Python-programokat írni. A következő fejezetben újabb ismereteket szerzünk a Python adattípusairól: mélyebbre merülünk az `int` és az `str` típus tárgyalásában, és néhány teljesen új típus is előkerül. A 3. fejezetben a rendezett sorozatokkal és a listákkal, illetve a Python nyelv egyéb gyűjteményes adattípusaival kapcsolatos tudásunkat bővítjük. Ezt követően a 4. fejezetben a mostaninál lényegesen részletesebben tárgyaljuk a Python vezérlési szerkezeteit. Megtanuljuk azt is, hogy saját függvények írásával miként bontható működési egységekre egy program, így segítve a több helyen előforduló kódrészletek használatának elkerülését, és bátorítva a későbbiekben újrahasznosítható kódrészletek írását.

Gyakorlatok

A könyv példái
Bevezetés ☜

A gyakorlatok célja itt és a könyv más részein egyaránt az, hogy az Olvasót a Pythonnal való kísérletezgetésre bátorítsuk, és közvetlen tapasztalatok megszereztetésével segítsük a fejezet anyagának elsajátítását. A példák és gyakorlatok a számok és a szövegek feldolgozását egyaránt érintik, hogy az olvasóközönség lehető legszélesebb körének érdeklődését kielégítsék. Mind a példákat, mind a gyakorlatokat igyekszünk igen rövidre fogni, a hangsúlyt a gépelés helyett a gondolkodásra és a megismerésre helyezve. A könyv példái között megtaláljuk az egyes gyakorlatok megoldását is.

1. A `bigdigits.py` program egy igen szép változata az, amikor csillagok helyett az adott számjegy számaival írjuk ki a számjegyet. Például így:

```
bigdigits_ans.py 719428306
77777  1   9999    4    222   888   333    000    666
   7  11  9   9   44   2   2 8   8 3   3  0   0  6
  7   1  9   9  4 4   2  2  8   8     3 0     0 6
 7    1   9999 4  4      2    888    33  0     0 6666
7     1      9 444444   2    8   8     3 0     0 6   6
7     1      9    4    2     8   8 3   3  0   0  6   6
7    111     9    4    22222  888   333    000    666
```

Két lehetséges megközelítéssel élhetünk. Az egyszerűbb az, ha lecseréljük a listákban lévő csillagokat, de ez igen rugalmatlan megoldása a problémának, ezért érdemes lenne egy másikat keresnünk. Változtassuk meg inkább úgy a feldolgozást végző kódot, hogy a számjegyek sorait ne egészben vegyük, hanem karakterenként adjuk a kiírásra váró sorhoz, és közben minden csillagot cseréljünk a megfelelő számjegyre. Ennek elvégzése a `bigdigits.py` fájl lemásolásán felül nagyjából öt sor átírását jelenti. Nem nehéz, de némi ravaszkodást igénylő művelet. A feladat megoldása a `bigdigits_ans.py` fájlban található.

2. Az IDLE igen rugalmas és ügyes számológépként is használható. Néha azonban nem árt, ha készítünk egy a célfeladathoz jobban illő számológépet. Írjunk olyan programot, amely a felhasználótól egy `while` ciklusban kér be számokat, fokozatosan kialakítva a beírt számok listáját. Ha a felhasználó az ENTER billentyű lenyomásával befejezi a számok beírását, írjuk a képernyőre a beírt számokat, azt, hogy hány számot kaptunk, a számok összegét, a legalacsonyabb és legmagasabb beírt számot, valamint a számok számtani közepét (összeg/számosság). Íme egy próbafuttatás eredménye:

```
average1_ans.py
enter a number or Enter to finish: 5
enter a number or Enter to finish: 4
enter a number or Enter to finish: 1
enter a number or Enter to finish: 8
enter a number or Enter to finish: 5
enter a number or Enter to finish: 2
enter a number or Enter to finish:
numbers: [5, 4, 1, 8, 5, 2]
count = 6 sum = 25 lowest = 1 highest = 8 mean = 4.16666666667
```

A szükséges változók kezdőértékének beállítása nagyjából négy sor (az üres listát a `[]` jelöli), a `while` ciklus magja 15 sornál rövidebb, és akkor még az alapvető hibakezelést is számoltuk. A program végén lévő kiírás pár sorban elintézhető, vagyis az egész program, az olvashatóságot megkönnyítő üres sorokkal együtt körülbelül 25 soros. A feladat megoldása az `average_ans.py` fájlban található.

3. Vannak helyzetek, amikor tesztszöveget kell előállítanunk – például egy webes oldalterv feltöltésére, még mielőtt az igazi tartalom rendelkezésre állna, vagy tesztadatként, valamilyen jelentésíró program fejlesztésekor. Ez utóbbi ötlet nyomán elindulva írjunk olyan programot, amelyik bikkfaverseket írogat (olyasmit, amit a vogonok is szívesen hallgatnának).

`random.randint()` és `random.choice()` 33 ☜

Hozzunk létre pár olyan listát, ami szavakat tartalmaz. Egyet mondjuk a névelőknek („the”, „a” stb.), egyet a főneveknek („cat”, „dog”, „man”, „woman”), egyet az igéknek („sang”, „ran”, „jumped”), egyet pedig határozói igeneveknek („loudly”, „quietly”, „well”, „badly”). Ha ezzel megvagyunk, készítsünk egy ötször ismétlődő ciklust, amely minden alkalommal választ egy névelőt, főnevet, igét és határozót, mindezt véletlenszerűen,

a `random.choice()` függvény használatával. A `random.randint()` függvénnyel azt döntsük el, hogy melyik mondatformában alkotjuk meg a következő sort: névelő, főnév, ige és határozó, vagy csak névelő, főnév és ige legyen benne. Íme egy példafuttatás kimenete:

```
awfulpoetry1_ans.py
her man heard politely
his boy sang
another woman hoped
her girl sang slowly
the cat heard loudly
```

Be kell töltenünk a `random` modult. A listák 4-10 sort foglalhatnak el a bennük lévő szavak számának függvényében, a ciklus maga pedig kevesebb, mint tíz sor, vagyis pár üres sorral számolva a program nagyjából húsz soros lehet. A feladat megoldása az `awfulpoetry1_ans.py` fájlban található.

4. A bikkfaverseket farigcsáló programunk rugalmasabbá tehető úgy, hogy némi programozással bekérünk a felhasználó parancssorából egy 1 és 10 közötti számot, és a program ennyi sorból álló verseket írogat. Ha nincs parancssori bemenet, akkor alapértelmezés szerint írjon ötsoros verseket, mint az előbb. A fő ciklust le kell cserélnünk egy `while` ciklusra. Ne feledjük, hogy a Python összehasonlító műveletei láncolhatók, tehát nincs szükség logikai „és” műveletre, amikor azt ellenőrizzük, hogy a paraméter megfelelő határok között van-e. A program bővítése nagyjából tíz új programsorral valósítható meg. A feladat megoldása az `awfulpoetry2_ans.py` fájlban található.
5. Jó lenne, ha a második gyakorlatban lévő átlagszámító program nem csak a számtani közepet, hanem a mediánt is meg tudná határozni, ehhez azonban a lista rendezésére van szükség. A Python nyelvben a listák rendezése a `list.sort()` tagfüggvénnyel könnyedén megoldható, de a módszer használatától eltekintünk, minthogy eddig nem volt szó róla. Az átlagszámító programot bővítsük egy a számokat sorba rendező kódblokkal – a hatékonyság ezúttal nem számít, használjuk a legegyszerűbb eszünkbe jutó módszert. Ha a lista rendezésével megvagyunk, azt is tudnunk kell, hogy a medián páratlan számú elem esetén a középső értékkel, páros számú elem esetén pedig a két középső érték átlagának értékével egyezik meg. Számoljuk ki a mediánt, és írjuk ki a többi információval együtt.

Az utolsó feladat főleg tapasztalatlan programozók számára okozhat nehézségeket. Akkor is jelenthet kihívást, ha már rendelkezünk némi tapasztalattal a Python nyelvű programozás terén – főleg, ha tartjuk magunkat ahhoz a megszorításhoz, hogy csak azokat az eszközöket használjuk, amelyekről már volt szó a könyvben. A rendezés kivitelezése körülbelül egy tucat új sort jelent, a medián kiszámítása (a maradékképzés művelet használata nélkül, hiszen azt még nem tárgyaltuk) pedig további négyet. A feladat megoldása az `average2_ans.py` fájlban található.

2

Adattípusok

- Azonosítók és kulcsszavak
- Egészszerű típusok
- Lebegőpontos típusok
- Karakterláncok

Ebben a fejezetben megkezdjük a Python programnyelv eddigieknél jóval részletesebb bemutatását. Először az objektumhivatkozások elvnevezésénél használatos szabályokkal ismerkedünk meg, ehhez kapcsolódóan felsorolva a Python saját kulcsszavait. Ezt követően részletesen bemutatjuk a nyelv legfontosabb adattípusait – a gyűjteményes típusok kivételével, amelyekről a 3. fejezetben ejtünk szót. A tárgyalásban szereplő adattípusok a nyelv sajátjai; kivételt ez alól csak azok jelentenek, amelyek az alapkönyvtárból valók. A kettő között a különbség mindössze annyi, hogy az utóbbiak használatához először be kell töltenünk a megfelelő modult, és ezzel minősítenünk az adattípus nevét – minderről az 5. fejezetben olvashatunk.

Azonosítók és kulcsszavak

Objektumhivatkozások 9 ☜

Amikor létrehozunk egy adatelemet, két lehetőségünk van: hozzárendelhetjük egy változóhoz, vagy elhelyezhetjük egy gyűjteményben. (Amint az előző fejezetben említettük, a hozzárendelés – értékadás – során egy objektumhivatkozást csatolunk az adatot a memóriában tároló objektumhoz.) Az objektumhivatkozásoknak adott elnevezések az *azonosítók*, de gyakran egyszerűen *névként* hivatkozunk rájuk.

A Python azonosítói olyan nem üres karaktersorozatok, amelyek egy „kezdőkarakterből" és nulla vagy több „folytatáskarakterből" állnak. Mindez azonban nem elég – az azonosítónak emellett számos szabálynak és elnevezési elvnek kell megfelelnie.

Az első szabály a kezdő- és folytatáskarakterekre vonatkozik. Az előbbiek helyén bármely betűnek minősülő Unicode-karakter állhat, így az ASCII-betűk („a", „b", ..., „z", „A", „B", ...,"Z"), az aláhúzás karakter (_), valamint a legtöbb angoltól eltérő nyelv betűi.

A folytatáskarakterek között szerepelhet bármely, kezdőként megengedett karakter, illetve szinte minden Unicode nem szóköz karakter, így a számjegyek („0”, „1”, ..., „9”), vagy a katalán „·” karakter. Az azonosítók érzékenyek a kis- és nagybetűk különbözőségére, így a `TAXRATE`, a `Taxrate`, a `TaxRate`, a `taxRate` és a `taxrate` öt különböző azonosítónak minősül.

A kezdő- és folytatáskarakterek megengedett értékeinek pontos listáját megtalálhatjuk a nyelv dokumentációjában (Python language reference, Lexical analysis, Identifiers and keywords rész), valamint a PEP 3131-ben* (Supporting Non-ASCII Identifiers)

A második alapszabály szerint az azonosítóknak nem adhatunk olyan nevet, ami megegyezik a Python valamelyik kulcsszavával – ez azt jelenti, hogy nem használhatjuk a 2.1. táblázatban szereplő neveket.

2.1. táblázat *A Python kulcsszavai*

`and`	`continue`	`except`	`global`	`lambda`	`pass`	`while`
`as`	`def`	`False`	`if`	`None`	`raise`	`with`
`assert`	`del`	`finally`	`import`	`nonlocal`	`return`	`yield`
`break`	`elif`	`for`	`in`	`not`	`True`	
`class`	`else`	`from`	`is`	`or`	`try`	

E kulcsszavak többségével már találkoztunk az előző fejezetben, jóllehet tizenegy bemutatása – `assert`, `class`, `del`, `finally`, `from`, `global`, `lambda`, `nonlocal`, `raise`, `with` és `yield` – még várat magára.

Első elnevezési elvünk a következő: ne használjuk a Python előre meghatározott azonosítóit a sajátjaink elnevezésénél, vagyis kerüljük a `NonImplemented` és az `Ellipsis` neveket, a Python beépített adattípusainak neveit (mint az `int`, a `float`, a `list`, az `str` vagy a `tuple`), valamint a beépített függvények és kivételek neveit. Azonnal felmerül a kérdés: honnan tudjuk, hogy pontosan mely nevek esnek a fenti kategóriák valamelyikébe? Nos, a Python rendelkezik egy `dir()` nevű beépített függvénnyel, amely egy adott objektum tulajdonságainak listáját adja vissza. Amennyiben paraméterek nélkül használjuk, a Python beépített tulajdonságait jeleníti meg. Lássunk hát, miként is használhatjuk:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
```

* PEP, vagyis Python Enhancement Proposal. Amennyiben valaki módosítani vagy bővíteni szeretné a Pythont, és ehhez megfelelő támogatást kap a programozói közösségtől, egy PEP-et kell készítenie, hogy hivatalosan is elfogadhassák. Egyes esetekben a bővítések beépülnek a nyelvbe – ez történt a PEP 3131 esetében is. A `www.python.org/dev/peps` címen az összes PEP-et megtekinthetjük.

A `__builtins__` tulajdonság valójában egy modul, amely a Python beépített tulajdonságait tartalmazza. Ha ezt adjuk meg a `dir()` paramétereként, az alábbi eredményt kapjuk:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
...
'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

A lista több mint 130 elemből áll, ezért nem mutatjuk be a maga teljességében. Annyit érdemes tudnunk, hogy a nagybetűvel kezdődő elemek a Python beépített kivételeinek nevei, a kisbetűsek pedig függvényeket és adattípusokat azonosítanak.

Ha túlzottan fárasztónak találjuk, hogy megjegyezzük, vagy újra és újra átnézzük ezeket az elkerülendő neveket, alkalmazhatjuk a Python valamelyik kódellenőrző eszközét – ilyen például a PyLint (`www.logilab.org/project/name/pylint`). Ezzel a segédeszközzel egyébként más meglevő hibákat vagy hibalehetőségeket is kiszűrhetünk.

Második elnevezési elvünk az aláhúzások (_) használatát érinti. Ne adjunk olyan neveket, amelyek két aláhúzásjellel kezdődnek és végződnek (mint például az `__lt__`). Az ilyen alakú neveket ugyanis a Python használja különleges tagfüggvények és változók azonosítására, magunk azonban kerüljük az alkalmazásukat. (A tagfüggvényeket magunk is megvalósíthatjuk, vagyis elkészíthetjük belőlük a saját változatunkat.) Ezekről a nevekről bővebben a 6. fejezetben szólunk majd. Mindemellett érdemes tudnunk, hogy bizonyos helyzetekben azok a nevek is különleges bánásmódot kapnak, amelyek egy vagy két aláhúzással kezdődnek (de nem végződnek két aláhúzással). Az előbbiek használatáról az 5., az utóbbiakról pedig a 6. fejezetben ejtünk szót.

Az aláhúzásjelet önmagában is használhatjuk azonosítóként – az interaktív értelmezőkben, illetve a Python Shellben pedig ez tárolja az utolsóként kiértékelt kifejezés eredményét. Egy egyszerű, futó programban ez az azonosító nem létezik, hacsak magunk használatba nem vesszük. Egyes programozók előszeretettel alkalmazzák olyan `for...in` ciklusokban, ahol nem lényegesek a sorra vett elemek. Ilyen például az alábbi is:

```
for _ in (0, 1, 2, 3, 4, 5):
    print("Hello")
```

`import` 34 ☜

Fontos tudnunk azonban, hogy a nemzetközivé tett programokban az aláhúzásjel gyakorta használatos a fordítófüggvény jelölésére.

`import` ☞ 192

Így a `gettext.gettext("Fordíts le")` helyett a `_("Fordíts le")` áll. (Ahhoz persze, hogy ez a kód működjön, előbb be kell töltenünk a `gettext` modult, hogy elérhessük a `gettext()` függvényét.)

A példa kedvéért nézzük meg, hogy milyen azonosítók jelenhetnek meg egy spanyolajkú programozó kódjában. Feltételezzük, hogy a korábbiakban végrehajtottuk az `import math` utasítást, és létrehoztuk a `radio` és a `vieja_área` változókat:

```
ð = math.pi
å = 0.0000001
nueva_área = ð * radio * radio
if abs(nueva_área - vieja_área) < å:
    print("las áreas han convergido")
```

A `math` modul segítségével az epszilon (`å`) értékét egy igen kicsiny lebegőpontos számban állapítottuk meg, és az `abs()` függvénnyel jutottunk hozzá a területek különbségéhez – mindezekről bővebben is szólunk a fejezet későbbi részében. Ami a számunkra ebben a pillanatban lényeges, hogy nyugodt lélekkel használhatunk ékezetes karaktereket, sőt görög betűket is a változóink azonosítására. Ugyanilyen könnyedséggel járhattunk volna el, ha arab, kínai, héber, japán vagy cirill betűket szerettünk volna használni – de bármilyen más nyelvet is választhatunk, amelyet a Unicode szabvány támogat.

Arról, hogy egy karakterlánc valóban érvényes azonosító, a legegyszerűbben úgy győződhetünk meg, ha megpróbálunk valamit hozzárendelni egy interaktív Python-értelmezőben, vagy az IDLE Python Shelljében. Íme néhány példa:

```
>>> stretch-factor = 1
SyntaxError: can't assign to operator (...)
>>> 2miles = 2
SyntaxError: invalid syntax (...)
>>> str = 3 # Legal but BAD
>>> l'impőt31 = 4
SyntaxError: EOL while scanning single-quoted string (...)
>>> l_impőt31 = 5
>>>
```

Ha érvénytelen azonosítót használunk, a rendszer `SyntaxError` kivételt vált ki. A zárójelekben álló hibaüzenet változó, így három ponttal helyettesítettük. Az első értékadás azért hibás, mert a „-” karakter nem Unicode-betű, számjegy vagy aláhúzás. A második esetben azért kapunk hibaüzenetet, mert az azonosító kezdőkaraktere számjegy – márpedig tudjuk, hogy számjegyek csak folytatáskarakterek lehetnek. Nem keletkezik kivétel, ha az azonosító alakja helyes – még akkor sem, ha egyébként megegyezik egy beépített adattípus, függvény vagy kivétel nevével –, így a harmadik értékadás működik, jóllehet ellenjavallt. A negyedik utasítás hibáját az okozza, hogy az aposztróf nem Unicode-betű, számjegy vagy aláhúzás. Az ötödik értékadással nincs semmi gond.

Egészszerű típusok

A Pythonban két egészszerű típus – az `int` és a `bool` – áll a rendelkezésünkre*. Mind az egész, mind a logikai típus állandó értékű (immutable), de – köszönhetően a Python bővített értékadó műveleteinek, ebből vajmi keveset érzékelünk. Amikor logikai kifejezésekkel dolgozunk, a 0 felel meg a `False` értéknek, a `True` szerepét pedig tetszőleges nullától különböző egész szám, illetve maga a `True` játszhatja. Számértékű kifejezésekben a `True` értéke 1, míg a `False`-nak a 0 felel meg. Mindez azt jelenti, hogy meglehetősen különös utasításokat is összeállíthatunk, amelyek mindazonáltal értelmesek maradnak – így például az `i += True` utasítással eggyel növelhetjük az `i` egész értékét. Természetesen egyébként ezt az `i += 1` alakban tennénk meg.

Egészek

A programjainkban tárolható egészek méretét csak a számítógép memóriakapacitása korlátozza, így nyugodtan létrehozhatunk több száz jegyű egész számokat, és dolgozhatunk is velük, jóllehet a kezelésük több időt vesz igénybe, mint azoké a számoké, amelyeket a számítógép processzora maga is képes ábrázolni.

A literális egészeket alapértelmezés szerint tízes számrendszerben adhatjuk meg, de más számrendszereket is használhatunk, amennyiben ez a munkánk szempontjából kedvezőbbnek látszik:

```
>>> 14600926                      # decimális
14600926
>>> 0b110111101100101011011110    # bináris
14600926
>>> 0o67545336                    # oktális
14600926
>>> 0xDECADE                      # hexadecimális
14600926
```

A bináris számokat a `0b`, az oktálisakat a `0o`**, a hexadecimálisakat pedig a `0x` előtaggal kell kezdenünk. Jó tudni, hogy ez utóbbi esetben nagybetűket is használhatunk a számok felírásánál.

Egészekkel minden megszokott matematikai műveletet és műveleti jelet alkalmazhatunk – ezt mutatja a 2.2. táblázat is. Egyes lehetőségek megvalósítására beépített függvények szolgálnak (ilyen például az `abs()`, ahol az `abs(i)` az `i` egész abszolútértékét adja vissza), míg másokat `int` műveletek biztosítanak (így az `i+j` az `i` és a `j` összegét adja).

---

* Az alapkönyvtár (standard library) a rendelkezésünkre bocsátja a `fractions.Fraction` típust is, így korlátlan pontossággal ábrázolhatunk racionális számokat, ami igen hasznos lehet bizonyos matematikai és tudományos feladatoknál.

** A C stílusú programozási nyelvek felhasználóinak külön kiemeljük, hogy az oktális számok jelzésére nem elég egy 0 karaktert helyezni a szám elé – a Pythonban előtagként a `0o` (nulla és egy o betű) használatos.

A bináris számműveletek (`+`, `-`, `/`, `//`, `%` és `**`) rendelkeznek értékadó változattal is (`+=`, `-=`, `/=`, `//=`, `%=` és `**=`), ahol az `x op= y` logikailag egyenértékű az `x = x op y` kifejezéssel, amennyiben az `x` kiolvasásának nincsenek egyéb mellékhatásai.

Objektumokat létrehozhatunk literális értékek változókhoz rendelésével (például: `x = 17`), de azt is megtehetjük, hogy függvényként meghívjuk a megfelelő adattípust (például: `x = int(17)`). Egyes objektumok (például a `decimal.Decimal` típusúak) csak ez utóbbi módszerrel hozhatók létre, mivel nincs literális alakjuk. Amennyiben az adattípus meghívásával hozzuk létre az objektumot, három lehetséges esetet érdemes különválasztanunk.

Az első esetben az adattípust paraméterek nélkül hívjuk meg. Ilyenkor a keletkező objektum az adattípus alapértelmezett értékét kapja – így az `x = int()` eredményeképpen egy 0 értékű egészet kapunk. Bármely beépített típust meghívhatjuk paraméter nélkül.

A második esetben egyetlen paraméterrel hívjuk meg az adattípust. Amennyiben megegyezű típusú értéket adunk meg, az eredmény az átadott objektum sekély másolata (shallow copy) lesz. (A sekély másolásról bővebben a 3. fejezetben szólunk.) Ha eltérő típusú paramétert adunk meg, típusátalakításra kerül sor. Ennek menetéről a 2.3. táblázat tájékoztat. Amennyiben a paraméter olyan típusból való, amelyet át lehet alakítani az adott típussá, de az átalakítás kudarcot vall, `ValueError` kivételt kapunk, egyébként pedig hozzájutunk az eredményobjektumhoz. Ha a paraméter típusát nem lehet a másik típussá alakítani, `TypeError` kivétel lép fel. A beépített `float` és `str` típusok lehetővé teszik az egésszé alakítást, emellett ilyen és egyéb átalakításokra lehetőségünk van az egyedi adattípusaink esetében is, amint arról a 6. fejezetben beszámolunk.

2.2. táblázat *Számműveletek és -függvények*

Használati alak	Leírás
`x + y`	Összeadja az `x` és az `y` számokat.
`x - y`	Kivonja `y`-t `x`-ből.
`x * y`	Megszorozza `x`-et `y`-nal.
`x / y`	Elosztja `x`-et `y`-nal. Az eredmény mindig `float` (illetve `complex`, ha `x` vagy `y` `complex` típusú).
`x // y`	Elosztja `x`-et `y`-nal. A művelet levágja az eredmény törtrészét, így mindig `int` típusú értéket kapunk (lásd a `round()` függvényt).
`x % y`	Visszaadja az `x/y` osztásnál kapott maradékot.
`x ** y`	`x`-et az `y`-adik hatványra emeli (lásd még a `pow()` függvényt).
`-x`	`x` ellentettjét hozza létre, vagyis ha `x` nem nulla, megváltoztatja az előjelét, ha nulla, nem tesz semmit.
`+x`	Nem tesz semmit. Többnyire a kód világossá tételére használják.
`abs(x)`	Visszaadja `x` abszolútértékét.

2.2. táblázat *Számműveletek és -függvények* *(folytatás)*

Használati alak	Leírás
Rendezett sorozatok 12 ☞ `divmod(x, y)`	Az `x/y` osztás eredményét és osztási maradékát két egész szám rendezett párjaként adja vissza. Rendezett sorozatok ☞ 102
`pow(x, y)`	`x`-et az `y`-adik hatványra emeli – ugyanazt teszi, mint a `**` művelet.
`pow(x, y, z)`	Az `(x ** y) % z` egyszerűbb alakja.
`round(x, n)`	Visszaadja az `x` `float` érték kerekítése során kapott egész számot (illetve az *n* tizedesjegyű törtet, ha *n* is adott).

2.3. táblázat *Egészek átalakítófüggvényei*

Használati alak	Leírás
`bin(i)`	Visszaadja `i` egész szám bináris alakját egy karakterlánc képében, például: `bin(1980) == '0b11110111100'`.
`hex(i)`	Visszaadja `i` egész szám hexadecimális alakját egy karakterlánc képében, például: `hex(1980) == '0x7bc'`.
`int(x)`	Egésszé alakítja az `x` objektumot. Amennyiben nem sikerül, `ValueError` kivételt vált ki, ha pedig `x` adattípusa nem támogatja az átalakítást, `TypeError` kivételt kapunk. Ha `x` lebegőpontos szám, a művelet az egész részére csonkolja.
`int(s, alap)`	Az `s` karakterláncot egésszé alakítja – ha nem jár sikerrel, `ValueError` kivételt ad. Az *`alap`* paraméter feltüntetése nem kötelező, ha megadjuk, egy 2 és 36 közti egészet kell választanunk (a határokat is beleértve).
`oct(i)`	Visszaadja `i` egész szám oktális alakját egy karakterlánc képében, például: `oct(1980) == '0o3674'`.

2.4. táblázat *Bitenkénti műveletek egészek között*

Használati alak	Leírás
`i | j`	Bitenkénti OR művelet `int i` és `int j` között – a negatív számokat kettes komplemensként kezeli.
`i ^ j`	Bitenkénti XOR (kizáró vagy) művelet `i` és `j` között.
`i & j`	Bitenkénti AND `i` és `j` között.
`i << j`	Balra eltolja `i`-t `j` bittel, vagyis `i*(2**j)` túlcsordulás-vizsgálat nélkül.
`i >> j`	Jobbra eltolja `i`-t `j` bittel, vagyis `i//(2**j)` túlcsordulás-vizsgálat nélkül.
`~i`	`i` bitjeit az ellentettjükre váltja.

Gyűjtemények másolása
☞ 132, 142

A harmadik esetben két vagy több paramétert adunk meg – nem minden típus támogatja ezt, ha pedig igen, eltérő eredményt kaphatunk. Az `int` esetében lehetőségünk van két paraméter használatára: az elsőben egy egész számot leíró karakterláncot adhatunk meg, a másodikban pedig a karakterlánc-alak alapját. Így például az `int("A4", 16)` alkalmazásánál az eredmény egy egész szám lesz, amelynek az értéke 164. Ezt a használati alakot a 2.3. táblázatban láthattuk.

A bitenkénti műveleteket a 2.4. táblázatban találhatjuk meg. Minden bináris bitenkénti műveletnek (`|`, `^`, `&`, `<<` és `>>`) létezik értékadó változata is (`|=`, `^=`, `&=`, `<<=` és `>>=`), ahol az `i op= j` logikailag azonos az `i = i op j` kifejezéssel, amennyiben az `i` értékének kiolvasása nem jár mellékhatásokkal.Ha igaz/hamis jelzőket szeretnénk használni, alkalmazhatunk egyetlen egész számot, amelynek a bitjeit a bitenkénti műveletekkel vizsgálhatjuk. Ugyanezt megtehetjük kevésbé tömör, de kényelmesebb alakban, logikai értékek listájával is.

Logikai értékek

Két beépített logikai (Boolean, Bool-féle) objektum létezik: a `True` és a `False`. Más Python-adattípusokhoz hasonlóan (legyen szó beépített, könyvtári vagy egyéni típusokról) a `bool` adattípust is meghívhatjuk függvényként – ha nem adunk meg semmilyen paramétert, az eredmény `False` lesz, `bool` paraméter megadásakor ennek másolatához jutunk, más típusú paraméter esetén pedig a rendszer megpróbálja `bool` típusúvá alakítani az adott értéket. A beépített, valamint az alapkönyvtárból származó adattípusok kivétel nélkül átalakíthatók `bool` típusúvá, az egyedi adatípusok esetében pedig nem nehéz megadnunk az átalakítás módját. Lássunk most néhány logikai értékadást és kifejezést:

```
>>> t = True
>>> f = False
>>> t and f
False
>>> t and True
True
```

Logikai műveletek
☞ 19

Amint a korábbiakban is említettük, a Python három logikai műveletet – `and`, `or` és `not` – bocsát a rendelkezésünkre. Mind az `and`, mind az `or` rövidzáras logikát alkalmaz, tehát azt a tényezőt adja vissza, amelyik meghatározza az eredményt, míg a `not` viszatérési értéke `True` vagy `False` lehet.

A Python régebbi változatait használó programozók olykor az 1 és a 0 értéket alkalmazták a `True` és a `False` helyett, ami a legtöbb esetben nagyszerűen működik is, egy új kódban azonban minden olyan helyen ragaszkodjunk a beépített logikai objektumokhoz, ahol logikai értékekre van szükség.

Lebegőpontos típusok

A Python háromféle lebegőpontos típussal rendelkezik: ide tartozik a beépített `float` és `complex`, valamint az alapkönyvtár `decimal.Decimal` típusa. Mindhárom állandó értékű típus. A `float` típus kétszeres pontosságú lebegőpontos számokat tárol, amelyek értéktartománya attól függ, hogy milyen C- (C#- vagy Java-) fordítóval építettük fel a Pythont, a pontosságuk korlátozott, továbbá nem vizsgálhatjuk megbízhatóan az egyenlőségüket. A `float` értékeket ábrázolhatjuk tizedesponttal vagy exponenciális jelölésmóddal: `0.0`, `4.`, `5.7`, `-2.5`, `-2e9`, `8.9e-4`.

A számítógépek a lebegőpontos számokat kettes számrendszerben ábrázolják, így egyes tizedes törtek (mint a 0.5) pontosan megjeleníthetők, míg mások (mint a 0.1 vagy a 0.2) csak közelítőleg. Ráadásul ez az ábrázolásmód rögzített számú bitet használ, így létezik egy korlát a szám jegyek számára nézve is. Íme egy példa az IDLE-ben, ami mindent megvilágít:

```
>>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.400000000000004, -2.5, 0.0008899999999999995)
```

A pontatlanság nem csak a Pythonra jellemző – az összes programozási nyelv ugyanezzel a gonddal küzd a lebegőpontos számok ábrázolásánál.

Ha valóban nagy pontosságra vágyunk, a Python `decimal` moduljának `decimal.Decimal` típusa segíthet. Az ilyen típusú számok olyan pontosságúak, amilyet csak szeretnénk (alapértelmezés szerint 28 tizedesjegyig pontosak), és a típus képes a periodikus törtrészű számok (mint a 0.1) pontos ábrázolására is – hátránya azonban, hogy e számok feldolgozása jelentősen lassabb egyszerű `float` típusú társaikénál. Pontosságuknak köszönhetően a `decimal.Decimal` számok alkalmasak üzleti számítások végrehajtására.

A nyelv támogatja a kevert módú aritmetikát, vagyis ha egy kifejezésben egy `int` és egy `float` értéket is használunk, az eredmény `float` lesz, a `float` és a `complex` közös használata pedig `complex` értéket eredményez. Mivel a `decimal.Decimal` értékek pontossága rögzített, csak másik `decimal.Decimal`, illetve `int` értékekkel alkalmazhatók együtt – utóbbi esetben az eredmény `decimal.Decimal` típusú lesz. Amennyiben össze nem egyeztethető típusok között kísérlünk meg valamilyen műveletet végrehajtani, a rendszer `TypeError` kivételt vált ki.

Lebegőpontos számok

A 2.2 táblázat számműveletei és -függvényei – beleértve az értékadó változatokat is – egytől egyig alkalmazhatók `float` típusú értékeken. Magát a `float` típust meghívhatjuk függvényként is – ha nem adunk át paramétert, az eredmény `0.0` lesz, `float` típusú paraméter esetén ennek másolatát kapjuk, míg más típusnál a rendszer megkísérli az átalakítást `float`-tá. Ha a célunk az, hogy a szám karakterlánc-alakjából számértéket kapjunk, a kiin-

dulási értéket egyaránt megadhatjuk decimális vagy exponenciális alakban. A `float` értékekkel végzett számítások során olykor a NaN („not a number" – nem szám), vagy az „infinity" (végtelen) értékekhez juthatunk – sajnos ezek a nyelv megvalósításától és az alkalmazott matematikai könyvtártól függően eltérően viselkedhetnek.

Íme egy egyszerű függvény, amely két `float` érték egyenlőségét vizsgálja a számítógép pontosságán belül:

```
def equal_float(a, b):
    return abs(a - b) <= sys.float_info.epsilon
```

Mindehhez szükségünk van a `sys` modul betöltésére. A `sys.float_info` objektum számos tulajdonsággal rendelkezik, a `sys.float_info.epsilon` pedig nem más, mint a legkisebb különbség, ami a gépen ábrázolható lebegőpontos számok között előfordulhat. A szerző egyik 32 bites gépén ez az érték éppen egy kicsivel nagyobb 0.0000000000000002-nél. (Ezt az értéket hagyományosan epszilonnak hívják.) A Python lebegőpontos számai jobbára megbízható pontosságot adnak 17 értékes számjegyig.

Ha az IDLE-ben beírjuk a `sys.float_info` utasítást, hozzájutunk az objektum összes tulajdonságához, köztük a gép által megjeleníthető legkisebb és legnagyobb lebegőpontos számokhoz, a `help(sys.float_info)` pedig némi tájékoztatást ad magáról az objektumról.

A lebegőpontos számokat egészekké alakíthatjuk az `int()` függvénnyel, amely mindössze az egész részt tartja meg, a törtrészt pedig eldobja, a `round()` függvénnyel, amely a kerekítésnél tekintettel van a törtrészre is, illetve a `math.floor()` és a `math.ceil()` függvényekkel, amelyek a lebegőpontos számhoz legközelebbi, nála nem nagyobb, illetve nem kisebb egészt adják vissza. A `float.is_integer()` tagfüggvény `True` értékkel tér vissza, ha a megadott lebegőpontos szám törtrésze 0. A lebegőpontos számot törtalakban a `float.as_integer_ratio()` tagfüggvénnyel kaphatjuk meg. Így például, ha `x = 2.75`, az `x.as_integer_ratio()` a `(11, 4)` eredménnyel tér vissza. Egészeket a `float()` alkalmazásával alakíthatunk lebegőpontos számokká.

2.5. táblázat *A math modul függvényei és állandói, 1. rész*

Használati alak	Leírás
`math.acos(x)`	x árkusz koszinuszát adja vissza radiánban.
`math.acosh(x)`	x árkusz hiperbolikusz koszinuszát adja vissza radiánban.
`math.asin(x)`	x árkusz szinuszát adja vissza radiánban.
`math.asinh(x)`	x árkusz hiperbolikusz szinuszát adja vissza radiánban.
`math.atan(x)`	x árkusz tangensét adja vissza radiánban.
`math.atan2(y, x)`	y / x árkusz tangensét adja vissza radiánban.
`math.atanh(x)`	x árkusz hiperbolikusz tangensét adja vissza radiánban.

2.5. táblázat *A math modul függvényei és állandói, 1. rész* *(folytatás)*

Használati alak	Leírás
`math.ceil(x)`	Eredménye ⌈x⌉, vagyis a legkisebb, x-nél nem kisebb egész szám. `math.ceil(5.4) == 6`
`math.copysign(x,y)`	x-et adja vissza y előjelével.
`math.cos(x)`	A radiánban megadott x koszinuszát adja vissza.
`math.cosh(x)`	A radiánban megadott x koszinusz hiperbolikuszát adja vissza.
`math.degrees(r)`	Az `r float` típusú számot radiánból fokra váltja.
`math.e`	Az `e` állandó, amelynek az értéke közelítőleg 2.7182818284590451
`math.exp(x)`	Az eredmény e^x, vagyis `math.e ** x`
`math.fabs(x)`	Az `x float` típusú abszolútértékét adja vissza.
`math.factorial(x)`	Az eredmény `x!`
`math.floor(x)`	Eredménye ⌊x⌋ vagyis a legnagyobb, x-nél nem nagyobb egész szám. `math.floor(5.4) == 5`
`math.fmod(x, y)`	Az `x/y` osztás maradékát adja meg – lebegőpontos számok esetén jobb eredményt ad, mint a `%`.
`math.frexp(x)`	Egy számpárt ad vissza, a mantisszával (`float`) és az exponenssel (`int`).
`math.fsum(i)`	Az `i` bejárható objektum értékeinek összegét adja vissza `float` alakban.
`math.hypot(x, y)`	Az eredmény gyök alatt x^2+y^2.
`math.isinf(x)`	Az eredmény `True`, ha `float x` értéke ± `inf` (± ∞).
`math.isnan(x)`	Az eredmény `True`, ha `float x` értéke `nan` („not a number” – nem szám)
`math.ldexp(m, e)`	Az eredmény $m \times 2^e$ – gyakorlatilag a `math.frexp()` inverze.
`math.log(x, b)`	Az eredmény $\log_b x$, a b megadása nem kötelező, alapértelmezett értéke `math.e`.
`math.log10(x)`	Az eredmény $\log_{10} x$.
`math.log1p(x)`	Az eredmény $\log_e(1+x)$ – akkor pontos, ha x 0-hoz közeli szám.
`math.modf(x)`	Visszaadja x egész és törtrészét két `float` típusú szám alakjában.

Rendezett sorozatok 11 ☜

Rendezett sorozatok ☞ 102

2.6. táblázat *A math modul függvényei és állandói, 2. rész*

Használati alak	Leírás
`math.pi`	A ð állandó, értéke közelítőleg 3.1415926535897931.
`math.pow(x, y)`	Az eredmény x^y mint `float` érték.
`math.radians(d)`	A `float` `d` számot fokból radiánná alakítja.
`math.sin(x)`	A radiánban megadott `x` szinuszát adja vissza.
`math.sinh(x)`	A radiánban megadott `x` szinusz hiperbolikuszát adja vissza.
`math.sqrt(x)`	Az eredmény √x.
`math.sum(i)`	Az `i` bejárható objektum elemeinek összegét adja vissza `float` alakban.
`math.tan(x)`	A radiánban megadott `x` tangensét adja vissza.
`math.tanh(x)`	A radiánban megadott `x` tangens hiperbolikuszát adja vissza.
`math.trunc(x)`	Az `x` egész részét adja vissza `int` alakban – ugyanaz, mint az `int(x)`.

A lebegőpontos számok hexadecimális alakú karakterláncokkal is megjeleníthetők a `float.hex()` tagfüggvénnyel. Az ellentétes irányú átalakítást a `float.fromhex()` segítségével végezhetjük el.* Lássunk is erre egy példát:

```
s = 14.25.hex()          # str s == '0x1.c800000000000p+3'
f = float.fromhex(s)     # float f == 14.25
t = f.hex()              # str t == '0x1.c800000000000p+3'
```

Az exponenst itt a `p` betű (mint „power”, vagyis hatvány) jelzi, hiszen az `e` maga is hexadecimális számjegy.

A nyelv beépített lebegőpontos lehetőségei mellett a `math` modul számos olyan függvényt bocsát a rendelkezésünkre, amelyek `float` típusú értékekkel végeznek műveleteket (lásd a 2.5. és a 2.6. táblázatot). Használatukra az alábbiakban mutatunk néhány példát:

```
>>> import math
>>> math.pi * (5 ** 2)
78.53981633974831
>>> math.hypot(5, 12)
13.0
>>> math.modf(13.732)
(0.7319999999999932, 13.0)
```

A `math.hypot()` függvény az origó és az `(x, y)` koordinátákkal megadott pont távolságát számítja ki, eredménye tehát `math.sqrt((x ** 2) + (y ** 2))`.

---

* Megjegyzés objektumközpontú programozók számára: a `float.fromhex()` osztálytagfüggvény.

A `math` modul jelentős mértékben arra a háttérben meghúzódó matematikai könyvtárra épít, amelyet a Python fordításánál alkalmaztunk. Mindez azt is jelenti, hogy egyes hibajelenségek és határesetek különbözőképpen viselkedhetnek eltérő rendszereken.

Komplex számok

A `complex` állandó értékű adattípus, amely nem más, mint egy `float` értékekből álló pár – egyikük adja a komplex szám valós, a másik pedig a képzetes részét. A literális komplex számok valós és képzetes részét egy +, illetve – jel választja el egymástól, és a képzetes rész után egy `j` betű áll.* Így komplex számok a következők: `3.5+2j`, `0.5j`, `4+0j`, `-1-3.7j`. Vegyük észre, hogy ha a valós rész értéke nulla, teljesen el is hagyható.

A komplex szám részeit a `real`, illetve `imag` tulajdonságain keresztül érhetjük el, például így:

```
>>> z = -89.5+2.125j
>>> z.real, z.imag
(-89.5, 2.125)
```

A `//`, a `%`, a `divmod` és a háromparaméteres `pow()` kivételével a 2.2. táblázat összes számműveletе és -függvénye – értékadó változataikkal egyetemben –alkalmazható komplex számokra. Nem szabad továbbá megfeledkeznünk a `complex` típus `conjugate()` tagfüggvényéről, amely megváltoztatja a képzetes rész előjelét, valahogy így:

```
>>> z.conjugate()
(-89.5-2.125j)
>>> 3-4j.conjugate()
(3+4j)
```

Vegyük észre, hogy a tagfüggvényt egy literális `complex` számra hívtuk meg. Ez nem tekinthető szokatlannak, hiszen a Python lehetővé teszi, hogy bármely literális értéken elérjünk egy tagfüggvényt vagy tulajdonságot – már amennyiben a literális érték adattípusa rendelkezésünkre bocsátja azt. Mindazonáltal ez nem vonatkozik a különleges tagfüggvényekre, mivel ezek helyett a megfelelő műveleteket – mint például a + – használjuk. Így a `4j.real` eredménye `0.0`, a `4j.imag` eredménye `4.0`, a `4j + 3 + 2j` eredménye pedig `3 + 6j`.

A `complex` adattípus függvényként is meghívható – ha ezt paraméterek nélkül tesszük meg, a `0j` értéket kapjuk vissza, `complex` típusú érték esetén az eredmény maga a paraméter lesz, egyébként pedig a rendszer kísérletet tesz a megadott objektum átalakítására. Amennyiben a célunk az átalakítás, a `complex()` számára átadhatunk egy karakterláncot vagy egy `float` értékekből álló párt. Ha csak egyetlen `float` értéket adunk meg, a rendszer a képzetes részt `0j`-nek tekinti.

---

* A matematikusok az *i* betűt használják a $\sqrt{-1}$ jelölésére, a Python azonban a mérnökök világában elterjedt *j* jelölést vette át.

A `math` modul függvényei nem képesek `complex` típusú számok kezelésére. Mindez józan meggondolás eredménye, hiszen a modul fejlesztői úgy vélték, jobb, ha egyértelmű kivételhez jutunk, mintha számításaink közepén egyszer csak feltűnik egy váratlan komplex szám.

Akik komolyabb műveleteket szeretnének végezni komplex számokkal, a `cmath` modult tölthetik be, amely megvalósítja a legtöbb trigonometrikus és logaritmikus függvény komplex változatát, emellett további, kifejezetten a komplex számokkal használatos függvényt is elérhetővé tesz, mint a `cmath.phase()`, a `cmath.polar()`, valamint a `cmath.rect()`. Elérhető emellett a `cmath.pi` és a `cmath.e` állandó is, amelyek ugyanazokat a `float` értékeket tartalmazzák, mint `math` modulbeli megfelelőik.

Decimális számok

A `float` típusra jellemző számítási pontatlanság számos alkalmazásnál teljességgel lényegtelen, illetve hatásait messze felülmúlja a kapott sebesség. Adódnak azonban helyzetek, amikor a pontosság létfontosságú, még akkor is, ha tudjuk, hogy jelentős sebességcsökkenésnek nézünk elébe. Ilyenkor segít a `decimal` modul állandó értékű `Decimal` típusa, amelynél a pontosságot mi magunk határozhatjuk meg. Igaz ugyan, hogy a számítások lassabbak, mintha `float` értékeket használnánk, de az, hogy a sebességcsökkenést valóban érezzük-e, az adott alkalmazástól függ.

Amennyiben létre szeretnénk hozni egy decimális értéket, mindenekelőtt be kell töltenünk a `decimal` modult:

```
>>> import decimal
>>> a = decimal.Decimal(9876)
>>> b = decimal.Decimal("54321.012345678987654321")
>>> a + b
Decimal('64197.012345678987654321')
```

A decimális számok létrehozásáért a `decimal.Decimal()` függvény felelős. Paraméterként egész vagy karakterlánc értéket adhatunk át neki, `float` értéket azonban nem, hiszen e számok pontossága eleve változó, míg a decimális típusúaké meghatározott. Amennyiben karakterlácot adunk át, egyaránt használhatunk tizedespontos, illetve exponenciális jelölést. A `decimal.Decimal` értékek pontos ábrázolása a nagyobb számítási pontosság mellett azt is eredményezi, hogy e számok egyenlőségvizsgálata megbízhatóan elvégezhető.

A 2.2. táblázatban felsorolt számműveletek és -függvények – beleértve az értékadó változatokat is – mind alkalmazhatók `decimal.Decimal` értékekre, jóllehet néhány szabályt be kell tartanunk. Ha a `**` művelet bal oldali tényezője `decimal.Decimal` típusú, akkor a jobb oldali tényező egész kell legyen. Hasonlóan, ha a `pow()` függvény első paramétere `decimal.Decimal` típusú, a másodiknak és a harmadiknak (amelyeket nem kötelező megadnunk) szintén egész értéket kell választanunk.

A `math` és `cmath` modulok nem használhatók `decimal.Decimal` értékekkel – ez alól csak a `math` modul néhány függvénye kivétel, amely `decimal.Decimal` értékű tagfüggvényként is elérhető. Így, ha például az `x float` típusú érték esetében ki szeretnénk számítani az e^x értékét, a `math.exp(x)` alakot használjuk, ha azonban `x decimal.Decimal` típusú, az `x.exp()` kifejezéssel juthatunk eredményre. Az előző fejezetből már tudjuk, hogy az `x.exp()` ez esetben valójában a `decimal.Decimal.exp(x)` egyszerűbb nyelvtani alakja.

A `decimal.Decimal` adattípus rendelkezésünkre bocsátja az `ln()` függvényt is, amely egy adott szám természetes (e alapú) logaritmusát számítja ki (éppúgy, mint a `math.log()`, amennyiben egyetlen paraméterrel használjuk), továbbá elérhető a `log10()` és az `sqrt()` is, valamint számos olyan tagfüggvény, amely kifejezetten erre a típusra jellemző.

A `decimal.Decimal` típusú számok viselkedése a megadott *környezet* szerint alakul – ez a környezet nem más, mint azoknak a beállításoknak a halmaza, amelyek a `decimal.Decimal` értékek viselkedését meghatározzák. Ide tartozik a pontosság (alapértelmezés szerint 28 tizedesjegy), a kerekítési módszer, valamint egyéb adatok.

Adódhatnak helyzetek, amikor a `float` és a `decimal.Decimal` közötti pontosságbeli különbség szembetűnővé válik:

```
>>> 23 / 1.05
21.904761904761905
>>> print(23 / 1.05)
21.9047619048
>>> print(decimal.Decimal(23) / decimal.Decimal("1.05"))
21.904761904761904761904761905
>>> decimal.Decimal(23) / decimal.Decimal("1.05")
Decimal('21.904761904761904761904761905')
```

Jóllehet a `decimal.Decimal` értékekkel végzett osztás pontosabb, mintha a `float` típust használnánk, ez esetben (egy 32 bites gépen) a különbség mindössze a tizenötödik tizedesjegynél jelentkezik. Az ilyen apró eltérés számos esetben teljesen lényegtelen – így például könyvünk minden olyan példájában, ahol lebegőpontos számokra volt szükség, megelégedtünk a `float` típussal.

Példánk azonban nemcsak erre irányítja rá a figyelmet, hanem arra is, hogy az objektumok képernyőre írását valamiféle háttérben lezajló formázás előzi meg. Amikor a `print()` függvényt a `decimal.Decimal(23) / decimal.Decimal("1.05")` eredményére hívjuk meg, a képernyőre egy egyszerű szám kerül – a kimenetet ilyenkor *karakterlánc alakban* kapjuk meg. Ha azonban egyszerűen csak a kifejezést írjuk le, egy `decimal.Decimal` típusú érték jelenik meg – ezt hívjuk ábrázolási alaknak. Ez általában is igaz a Python-objektumokra – mindegyikük két kimeneti alakkal rendelkezik. A karakterlánc alak emberi értelmezésre alkalmas, míg az *ábrázolási alakban* megadott értékből a Python-értelmező (amennyiben lehetséges) képes előállítani magát az ábrázolt objektumot. Erre a témakörre

a karakterláncok kapcsán hamarosan visszatérünk, majd pedig ismét tárgyaljuk a 6. fejezetben, amikor megtanuljuk, hogy miként határozzuk meg az egyedi adattípusaink karakterlánc és ábrázolási alakját.

A `decimal` modulnak a Library Reference-ben megtalálható leírása részletekbe menően tájékoztat mindenről, ami a hely hiánya vagy a téma különlegessége miatt a fenti tárgyalásból kimaradt – itt példákat is kapunk a modul alkalmazására, és a gyakran ismétlődő kérdések is választ nyernek.

Karakterláncok

Karakterkódolások
☞ 86

A karakterláncokat a `str` nevezetű, állandó értékű adattípussal ábrázolhatjuk, amely Unicode-karakterek sorozatát képes tárolni. Magát az `str` adattípust függvényként is meghívhatjuk – ha nem adunk meg paramétert, az üres karakterláncot kapjuk vissza, nem karakterlánc paraméterek esetén hozzájutunk az átadott érték karakterlánc alakjához, ha pedig karakterláncot adunk át, egy másolatát kapjuk meg. Az `str()` átalakítófüggvényként is alkalmazható – ilyenkor az első paraméter egy karakterlánc kell legyen, illetve olyan érték, amely karakterlánccá alakítható, és megadhatunk két további karakterlánc értékű paramétert is, amelyekkel a kódolást, valamint a kódolási hibák kezelésének módját határozhatjuk meg.

A korábbiakban említettük, hogy literális karakterláncokat könnyedén létrehozhatunk az idézőjelek segítségével, ráadásul az idézőjel és az aposztróf ugyanazt az eredményt adja (fontos azonban, hogy ne használjuk vegyesen a kettőt). Mindezek mellett létrehozhatunk *hármas idézőjelek közé zárt* karakterláncot is – ilyenkor a karakterláncot három idézőjellel (vagy aposztróffal) kezdjük, és így is zárjuk le. Így teszünk az alábbiakban is:

```
text = """A hármas idézőjelek közé zárt karakterláncban minden további
nélkül szerepelhetnek 'aposztrófok' és "idézőjelek". Emellett az új
sorok kezdését \
is levédhetjük, így ez a karakterlánc mindössze két sorból áll."""
```

Ha egy egyszerű idézőjelekkel vagy aposztrófokkal határolt karakterláncban szeretnénk újfent idézőjeleket vagy aposztrófokat használni, minden további nélkül megtehetjük, ha a belül használt típus eltér a kívül állótól – egyébként le kell védenünk őket:

```
a = "Az 'aposztrófok' használhatók, az \"idézőjeleket\"
le kell védenünk."
b = 'Az \'aposztrófokat\' le kell védenünk,
az "idézőjelek" használhatók.'
```

A Pythonban a sor vége egyúttal az utasítás végét is jelenti, kivéve a zárójeleken (`()`), a szögletes zárójeleken (`[]`), a kapcsos zárójeleken (`{}`) belül, valamint a hármas idézőjelekkel határolt karakterláncokban. Új sorokat minden további nélkül használhatunk hár-

mas idézőjelekkel határolt karakterláncokban, az egyszerű literális karakterláncokban pedig az \n védőkód segítségével helyezhetünk el új sorokat. A Python összes védőkódját megtalálhatjuk a 2.7. táblázatban. Bizonyos esetekben – például amikor szabályos kifejezéseket alkotunk – számos literális fordított perjelet kell alkalmaznunk (a szabályos kifejezésekről bővebben a 12. fejezetben szólunk). Ez bizony meglehetősen kényelmetlen lehet, hiszen mindegyiküket le kell védenünk:

```
import re
phone1 = re.compile("^((?:[(]\\d+[)])?\\s*\\d+(?:-\\d+)?)$")
```

2.7. táblázat *A Python védőkódjai*

Védőkód	Jelentés
\	Új sor
\\	Fordított perjel (\)
\'	Aposztróf (')
\"	Idézőjel (")
\a	ASCII csengő (BEL)
\b	ASCII fordított perjel (BS)
\f	ASCII lapdobás (FF)
\n	ASCII újsor (LF)
\N{*név*}	Adott nevű Unicode-karakter
\ooo	Oktális jegyekkel megadott karakter
\r	ASCII kocsivissza (CR)
\t	ASCII tabulátor (TAB)
\u*hhh*	16 bites hexadecimális értékkel megadott Unicode-karakter
\U*hhhhhhhh*	32 bites hexadecimális értékkel megadott Unicode-karakter
\v	ASCII függőleges tabulátor (VT)
\x*hh*	8 bites hexadecimális értékkel megadott karakter

A megoldást a *nyers* karakterláncok jelentik, amelyek lehetnek egyszerű vagy hármas idézőjelekkel határoltak, fontos viszont, hogy az első idézőjelet egy *r* betű előzze meg. Ilyen karakterláncokban a rendszer minden karaktert a literális alakjában értelmez, nincs szükség védelemre. Lássuk korábbi, phone nevezetű szabályos kifejezésünket nyers karakterlánc alakjában felírva:

```
phone2 = re.compile(r"^((?:[(]\d+[)])?\s*\d+(?:-\d+)?)$")
```

Amennyiben hosszú literális karakterláncot szeretnénk létrehozni, amely több sort is tartalmaz, és nem akarunk hármas idézőjeleket használni, kétféleképpen is eljárhatunk:

```
t = "Ez bizony nem a legjobb módszer két karakterlánc " + \
    "összekötésére, mivel szükség van az új sorok kihagyására."
s = ("Ezzel a módszerrel valóban szépen köthetünk "
     "össze karakterláncokat, literális alakok összefűzésével.")
```

Fontos, hogy a második esetben zárójeleket kell alkalmaznunk – ha ezt nem tennénk, az *s* értéke mindössze az első karakterlánc lenne, a második pedig `IndentationError` kivételt váltana ki. A Python leírásának *Idioms and Anti-Idioms* HOWTO-dokumentuma azt tanácsolja, hogy akármilyen helyzetben is szeretnénk több sort összekötni, az új sorok levédése helyett használjuk ezt a zárójeles módszert – könyvünkben megfogadjuk a tanácsot.

Mivel a `.py` fájlok alapértelmezés szerint az UTF-8 Unicode-kódolást alkalmazzák, tetszőleges Unicode-karaktert feltüntethetünk a literális karakterláncokban mindenféle külön jelölés nélkül. Arra is módunk van, hogy a kívánt Unicode-karaktert a hexadecimális védőkódjával, illetve a nevével azonosítsuk, valahogy így:

```
>>> euros = "€ \N{euro sign} \u20AC \U000020AC"
>>> print(euros)
€€€€
```

Karakterkódolások
☞ 86

Ez esetben nem alkalmazhatunk hexadecimális védőkódot, mivel ezek legfeljebb két számjegyűek lehetnek, így az értékük nem lehet nagyobb `0xFF`-nél. Érdemes tudnunk, hogy a Unicode-nevekben nem számít a kis- és nagybetűk különbözősége, és a bennük álló szóközök elhagyhatók.

Amennyiben hozzá szeretnénk jutni egy karakterlánc valamelyik karakterének Unicode-kódpontjához (vagyis a Unicode kódolásban neki megfelelő egész számhoz), a beépített `ord()` függvényt használhatjuk:

```
>>> ord(euros[0])
8364
>>> hex(ord(euros[0]))
'0x20ac'
```

Rendelkezésünkre áll a beépített `chr()` függvény is, amellyel bármely olyan egész számból, amely létező Unicode-kódpontot azonosít, megkaphatjuk a hozzá tartozó karaktert:

```
>>> s = "anarchists are " + chr(8734) + chr(0x23B7)
>>> s
'anarchists are ¥Ö'
>>> ascii(s)
"'anarchists are \u221e\u23b7'"
```

str.format
☞ 75, 83, 309

Amennyiben az `s`-et önmagában beírjuk az IDLE parancssorába, a rendszer karakterlánc alakjában írja ki, vagyis a kimeneti karakterek idézőjelek közé kerülnek. Ha csak ASCII-karakterekre van szükségünk,

alkalmazhatjuk a beépített `ascii()` függvényt, amely visszaadja paraméterének 7 bites ASCII-karakterekből álló alakját – ahol pedig ez nem lehetséges, az `\xhh`, `\uhhhh`, illetve `\Uhhhhhhhh` védőkódok legrövidebbikét választja. A karakterláncok kimenetének pontos szabályozásáról még szólunk a fejezet későbbi részében.

Karakterláncok összehasonlítása

A karakterláncok támogatják a szokásos összehasonlító műveleteket (<, <=, ==, !=, > és >=), amelyek a karakterláncok memóriában tárolt bájtjait vetik össze. Sajnos ez az összehasonlítási módszer két komoly nehézséget is okoz, például a karakterlánc-listák rendezése során. Mindkettő általánosságban érinti a Unicode-karakterláncokat támogató nyelveket – egyik sem a Python sajátja.

Karakterkódolások
☞ 86

Az első gondot az okozza, hogy a Unicode-karaktereket gyakran két vagy több eltérő bájtsorozat is megjelenítheti. Jó példa erre az *Å* (Unicode-kódpontja `0x00C5`), amely UTF-8 kódolású bájtokkal háromféleképpen is ábrázolható: `[0xE2, 0x84, 0xAB]`, `[0xC3, 0x85]` és `[0x41, 0xCC, 0x8A]`. Szerencsére ezen a gondon könnyedén úrrá lehetünk. Ha betöltjük a `unicodedata` modult, majd meghívjuk a `unicodedata.normalize()` függvényt, első paraméterének az „NFKD" értéket adva (ezzel a normalizálási eljárást adjuk meg, jelentése „Normalization Form Compatibility Decomposition", vagyis kompatibilis normalizált formára bontás), második paramétereként pedig átadva azt a karakterláncot, amely az *Å* valamelyik érvényes alakját tartalmazza, az eredmény egy olyan karakterlánc lesz, amelyet UTF-8 kódolású bájtokkal ábrázolva az *Å* helyén mindig a `[0x41, 0xCC, 0x8A]` bájtsorozatot kapjuk.

A második, komolyabb gondot az adja, hogy egyes karakterek rendezése nyelvfüggő. Ilyen például az *ä*, amely a svéd ábécében a *z* után áll, míg a németben úgy tekintik, mintha az *ae* betűpár állna a helyén. Hasonlóképpen, az angolban az *ø* betűt a rendezés szempontjából egyszerű *o*-nak tekintik, míg a norvég és a dán ábécében a *z* mögött áll. Mindez igen-igen sok gondot okozhat, ráadásul egyes alkalmazásokat különféle nyelvi környezetben élő felhasználók is futtathatnak (eltérő rendezési igényekkel), sőt némely karakterláncok különböző nyelvekből származó szavakat is tartalmazhatnak (például spanyol és angol tagot egyaránt). Az már csak hab a tortán, hogy bizonyos karakterek (nyilak, szimbólumok vagy éppen matematikai jelek) nem igazán illeszkednek semmilyen rendezési eljárásba.

A Python fejlesztői itt elvi döntést hoztak az apró, de kellemetlen hibák elkerülésére: nem tették lehetővé a rendszer számára, hogy találgasson. Esetünkben ez a nyelv kiválasztására vonatkozik, ezért a rendszer megelégszik a karakterláncok memóriabeli bájtjainak összehasonlításával. Így a rendezés alapjául a Unicode-kódpontok szolgálnak, ami az angol nyelv ASCII szerinti rendezésének felel meg. A normalizálásra jobbára nincs szükség, kivéve ha a karakterláncok valamilyen külső forrásból (fájlból vagy hálózati csatolótól) származnak – de még ilyenkor sem érdemes elvégeznünk, hacsak nem győződünk meg teljes bizonyos-

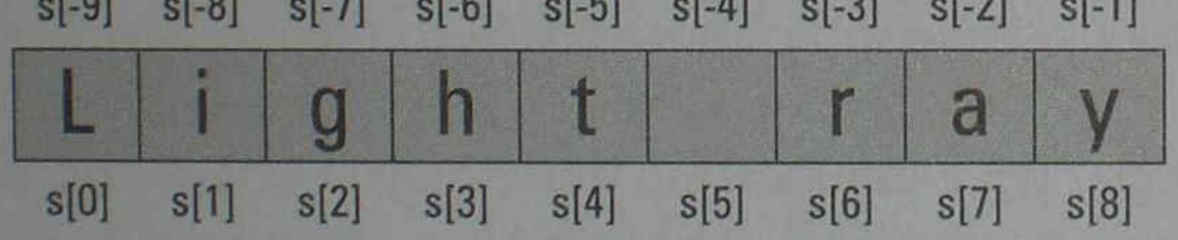
sággal a szükségességéről. Természetesen, amint a 3. fejezetben majd láthatjuk, a Python rendezési eljárásait igényeink szerint testreszabhatjuk. Ha bővebben szeretnénk olvasni a Unicode-karakterláncok rendezéséről, a Unicode Collation Algorithm dokumentum (`unicode.org/reports/tr10`) kielégítheti a tudásvágyunkat.

Karakterláncok szeletelése és egyelése

Gyűjteményes adattípusok 12 ☜

Az előző fejezetből tudhatjuk, hogy egy sorozat elemeit – így egy karakterlánc karaktereit is – kiolvashatjuk az elemelérési művelettel (`[]`). Nos, ez a művelet ennél sokkal többre is képes, hiszen nem csak egy elemet vagy karaktert, de egy teljes szeletet (részsorozatot) is kiolvashatunk vele – ebben a szerepben a neve: szeletelő művelet.

Először azonban foglalkozzunk az egyes karakterek kiolvasásával. A karakterláncok sorszámozása 0-val kezdődik, és a karakterlánc hossza mínusz 1-nél ér véget. Használhatunk azonban emellett negatív sorszámokat is – ilyenkor a számozás az utolsó karaktertől visszafelé folyik. Tegyük fel, hogy elvégeztük az `s = "Light ray"` értékadást. A 2.1. ábrán az `s` karakterlánc elérhető sorszámait láthatjuk.



2.1. ábra

Sorszámok egy karakterláncban

A negatív sorszámok meglepően hasznosak, különösképpen a –1, amely a karakterlánc utolsó karakterét adja meg. Amennyiben túllépünk az elérhető sorszámok halmazán (vagy bármely sorszámra hivatkozunk egy üres karakterlánc esetében), `IndexError` kivételhez jutunk.

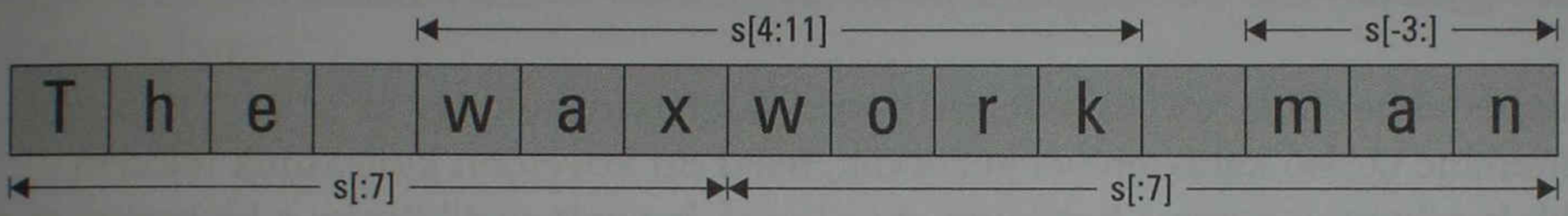
A szeletelő műveletet háromféle alakban használhatjuk:

```
sor[kezdet]
sor[kezdet:vég]
sor[kezdet:vég:lépés]
```

A *sor* bármilyen sorozat lehet – tehát lista, karakterlánc vagy rendezett számsorozat. A *kezdet*, *vég* és *lépés* értékek egész számok (illetve egészeket tartalmazó változók) kell legyenek. Az első alakot már használatba is vettük: így olvashatjuk ki a *kezdet*-edik elemet a sorozatból. A második alak kicsit izgalmasabb: használatával kiolvashatjuk a *kezdet*-től a *vég*-ig terjedő elemeket, nem beleértve az utóbbit. Minderről hamarosan még szót ejtünk.

Amennyiben a második (egy kettőspontot tartalmazó) alakot használjuk, bármely (akár mindkét) sorszámot elhagyhatjuk. Ha a *kezdet*-től szabadulunk meg, a rendszer az alapértelmezett 0 értéket alkalmazza helyette, míg a *vég* elhagyása esetén a `len(sor)` lép a helyébe. Mindez azt is jelenti, hogy ha mindkét sorszámot elhagyjuk, a teljes sorozat másolatát kapjuk – `s[:]` esetén tehát az `s[0:len(s)]` eredményt.

Éljünk most az `s = "The waxwork man"` értékadással, és figyeljünk meg néhány szeletet a 2.2. ábrán.



2.2. ábra

Sorozat szeletelése

Amennyiben egy karakterláncba egy részt szeretnénk beilleszteni, megtehetjük, ha a szeletelést és az összefűzést együtt használjuk – valahogy így:

```
>>> s = s[:12] + "wo" + s[12:]
>>> s
'The waxwork woman'
```

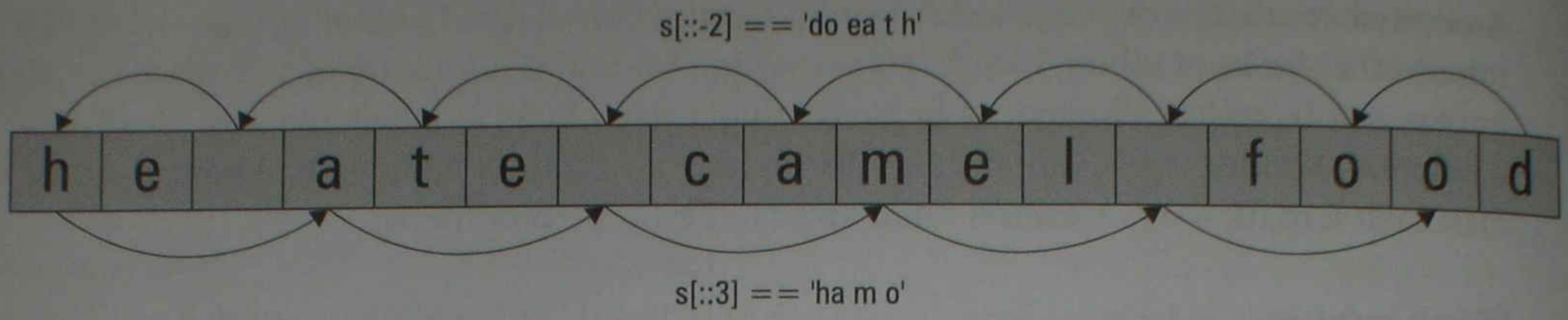
Mivel a „wo” az eredeti karakterláncban is megjelenik, ugyanezt az eredményt elérhettük volna az `s[:12] + s[7:9] + s[12:]` hozzárendelésével is.

Karakterlánc-műveletek és tagfüggvények
☞ 66

Amennyiben sok karakterlánc összefűzését kell elvégeznünk, a + és a += műveletek nem igazán hatékonyak – ilyenkor érdemes a hamarosan bemutatott `str.join()` tagfüggvényre támaszkodnunk.

Most azonban ismerkedjünk meg az `[]` művelet harmadik használati alakjával, amely hasonlóan működik, mint a második, de ezúttal csak minden *lépés*-edik karaktert olvasunk ki. Ha elhagyjuk a kezdősorszámot, a 0 alapértelmezett érték lép a helyébe – kivéve, ha negatív lépést adtunk meg, ilyenkor ugyanis az alapértelmezett kezdősorszám a –1. Amennyiben a *vég* sorszámot nem adjuk meg, ennek helyére a `len(sor)` érték lép, de ha a lépés értéke negatív, itt is változik a helyzet, és az alapértelmezett sorszám a karakterlánc kezdete előtti pozíciót jelzi. A *lépés* értékét nem hagyhatjuk el, és 0-t sem adhatunk itt át a műveletnek – ha nincs szükségünk erre a beállításra, használjuk a második (egy kettőspontot tartalmazó) alakot.

A következő ábrán néhány kiegyelt szeletet láthatunk az `s = "he ate camel food"` karakterláncból.



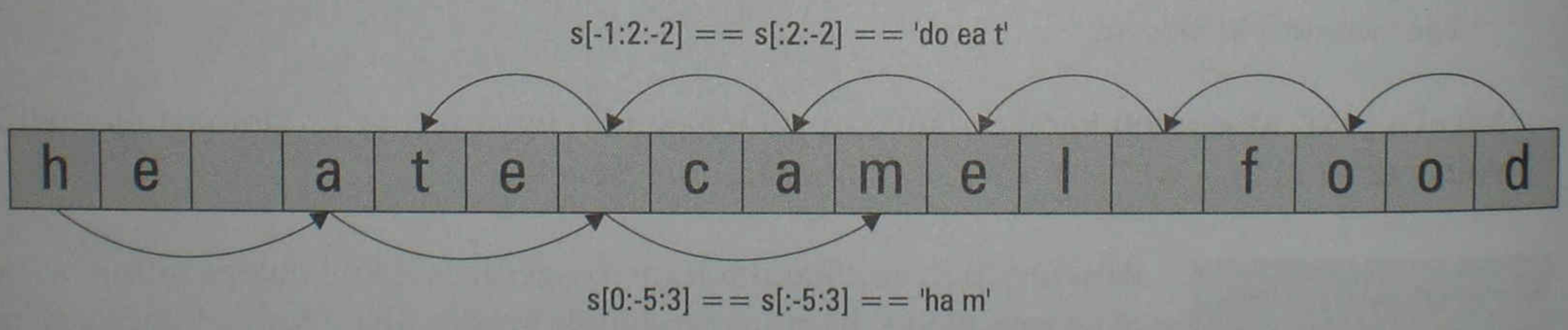
2.3. ábra

Sorozatok egyelése

Itt az alapértelmezett kezdő és befejező sorszámot használtuk, így az `s[::-2]` eredménye a karakterlánc utolsó karakterével kezdődik, majd ezt követően kiolvasunk minden második karaktert egészen a kezdetiig. Az `s[::3]` hasonlóan áll össze: a kiolvasást az első karakternél kezdjük, majd hármasával lépkedünk a karakterlánc végéig.

Az egyelés módszerét jobbára más típusú sorozatok esetében alkalmazzák, de van egy kifejezetten a karakterláncokra jellemző felhasználása:

```
>>> s, s[::-1]
('The waxwork woman', 'namow krowxaw ehT')
```



2.4. ábra

Egyelés és szeletelés – egyszerre

A –1-es lépésköz azt jelenti, hogy minden karaktert kiolvasunk a karakterlánc végétől az eleje felé haladva – vagyis hozzájutunk a karakterlánc tükörképéhez.

Karakterlánc-műveletek és tagfüggvények

Bejárható objektumok műveletei és függvényei ☞ 133

Mivel a karakterláncok állandó értékű sorozatok, minden olyan lehetőséget alkalmazhatunk, ami általában elérhető az ilyen sorozatok esetében. Ez azt jelenti, hogy a vizsgálhatjuk a „tagságot” az `in` segítségével, összefűzést végezhetünk a `+`, hozzáfűzést a `+=`, többszörözést a `*` és értékadó többszörözést a `*=` művelettel. Ezek mindegyikéről szót ejtünk a következőkben, emellett bemutatjuk a karakterláncok számos tagfüggvényét is. A 2.8., 2.9. és 2.10. táblázatok áttekintést adnak a karakterláncok összes tagfüggvényéről. Mindössze két, meglehetősen különleges függvény (az `str.maketrans()` és az `str.translate()`) maradt ki, ezekről azonban említést teszünk a későbbiekben.

Sorozatok lévén a karakterláncok „mérhető" objektumok, így a `len()` függvény számára átadhatjuk őket paraméterként. Az eredmény az adott karakterlánc karaktereinek száma lesz (üres karakterlánc esetén 0-t kapunk).

Láthattuk, hogy a + művelet túlterhelt, így alkalmas a karakterláncok összefűzésére is. Ha azonban az összefűzésben sok karakterlánc vesz részt, az `str.join()` tagfüggvény számottevően jobb megoldást jelent. A tagfüggvény paraméterként egy sorozatot vár (karakterláncok listáját vagy rendezett sorozatát), amelynek tagjait egyetlen karakterlánccá egyesíti azzal a karakterlánccal összekötve, amelyre meghívtuk. Lássuk, hogyan is fest ez a valóságban:

```
>>> treatises = ["Arithmetica", "Conics", "Elements"]
>>> " ".join(treatises)
'Arithmetica Conics Elements'
>>> "-<>-".join(treatises)
'Arithmetica-<>-Conics-<>-Elements'
>>> "".join(treatises)
'ArithmeticaConicsElements'
```

Az első példa talán a legismertebb – itt az egyesítésnél egyetlen karaktert használunk, ami jelen esetben a szóköz. A harmadik példa egyszerű összefűzést ad, köszönhetően annak, hogy a tagfüggvényt az üres karakterláncra hívtuk meg.

2.8. táblázat *Karakterláncok tagfüggvényei, 1. rész*

Használati alak	Leírás
`s.capitalize()`	Az eredmény az `s str` másolata, nagy kezdőbetűvel; lásd még az `str.title()` tagfüggvényt.
`s.center(`*`szélesség`*`,`*`karakter`*`)`	Az eredmény az `s` másolata egy *`szélesség`* hosszú karakterláncban középre igazítva, ahol a fennmaradó helyet szóközök, illetve, ha megadjuk, *`karakter`* karakterek (1 hosszú karakterláncok) töltik ki; lásd még az `str.ljust()`, az `str.rjust()` és az `str.format()` tagfüggvényeket.
`s.count(t,`*`kezdet, vég`*`)`	Megadja, hányszor szerepel a `t` karakterlánc az `s`-ben (illetve az `s` *`kezdet:vég`* szeletében).
`s.encode(`*`kódolás, hiba`*`)`	Egy `bytes` objektumot ad vissza, amely az alapértelmezett (illetve, amennyiben megadjuk, a *`kódolás`*) kódolással jeleníti meg a karakterláncot, a hibákat pedig a *`hiba`* paraméter szerint kezeli, amelynek a megadása nem kötelező.
`s.endswidth(x,`*`kezdet, vég`*`)`	Az eredmény `True`, ha az `s` (illetve az `s` *`kezdet:vég`* szelete) az `x` karakterlánccal, illetve az `x tuple` valamelyik elemével végződik – egyébként pedig `False`. Lásd még: `str.startswith()`.

bytes típus ☞ 297

Karakterkódolások ☞ 86

2.8. táblázat *Karakterláncok tagfüggvényei, 1. rész* *(folytatás)*

Használati alak	Leírás
`s.expandtabs(`*méret*`)`	Az `s` másolatát adja vissza, amelyben a tabulátorok helyett szóközök állnak nyolcasával, illetve a `size` paraméterrel megadható számban.
`s.find(t,` *kezdet, vég*`)`	Visszaadja `t` balról nézve legelső előfordulását az `s`-ben (illetve az `s` *kezdet*`:`*vég* szeletében) – ha nincs találat, az eredmény –1 lesz. Ha a jobb szélső előfordulást szeretnénk megkeresni, alkalmazzuk az `str.rfind()` tagfüggvényt. Lásd még: `str.index()`.
`s.format(...)`	Az `s` karakterlánc másolatát adja vissza a paraméterekkel meghatározott formázással. A tagfüggvényt és paramétereinek jelentését bővebben tárgyaljuk a fejezet későbbi részében.
`s.index(t,` *kezdet, vég*`)`	Megkeresi a `t` bal szélső előfordulását az `s`-ben (illetve az `s` *kezdet*`:`*vég* szeletében), ha pedig nem talál semmit, `ValueError` kivételt vált ki. Ha a jobb szélső előfordulást keressük, alkalmazzuk az `str.rindex()` tagfüggvényt. Lásd még: `str.find()`.
`s.isalnum()`	Az eredmény `True`, ha az `s` nem üres, és minden karaktere betű vagy szám.
`s.isalpha()`	Az eredmény `True`, ha az `s` nem üres, és minden karaktere betű.
`s.isdecimal()`	Az eredmény `True`, ha az `s` nem üres, és minden karaktere tízes számrendszerű Unicode-számjegy.
`s.isdigit()`	Az eredmény `True`, ha az `s` nem üres, és minden karaktere ASCII-számjegy.
`s.isidentifier()`	Az eredmény `True`, ha az `s` nem üres, és érvényes azonosító.
`s.islower()`	Az eredmény `True`, ha az `s` rendelkezik legalább egy olyan karakterrel, amelynek van kisbetűs alakja, és minden ilyen karakter esetében a kisbetűs alak szerepel a karakterláncban; lásd még: `str.isupper()`.

`str.format`
☞ 75, 83

2.9. táblázat *Karakterláncok tagfüggvényei, 2. rész*

Használati alak	Leírás
`s.isnumeric()`	Az eredmény `True`, ha az `s` nem üres, és minden karaktere számértékű Unicode-karakter, számjegy vagy tört, vagy más hasonló.
`s.isprintable()`	Az eredmény `True`, ha az `s` üres, illetve ha minden karaktere nyomtatható – ide tartozik például a szóköz, az újsor azonban nem.
`s.isspace()`	Az eredmény `True`, ha az `s` nem üres, és minden karaktere üreshely.

2.9. táblázat *Karakterláncok tagfüggvényei, 2. rész* *(folytatás)*

Használati alak	Leírás
`s.istitle()`	Az eredmény `True`, ha az `s` nem üres, és minden szava nagybetűvel kezdődik. Lásd még: `str.title()`.
`s.isupper()`	Az eredmény `True`, ha az `s` rendelkezik legalább egy olyan karakterrel, amelynek van nagybetűs alakja, és minden ilyen karakter esetében a nagybetűs alak szerepel a karakterláncban; lásd még: `str.islower()`.
`s.join(`*sor*`)`	Az eredmény a *sor* sorozat elemeinek összefűzése, ahol az egyes elemek között az `s` karakterlánc (ami lehet üres is) példányai állnak.
`s.ljust(`*szélesség*`,` *karakter*`)`	Az eredmény `s` másolata egy szélesség hosszú karakterláncban balra igazítva, ahol a fennmaradó helyet szóközök, illetve, ha megadjuk, *karakter* karakterek (1 hosszú karakterláncok) töltik ki. Jobbra igazításhoz az `str.rjust()`, középre igazításhoz pedig az `str.center()` tagfüggvényt használhatjuk.
`s.lower()`	Visszaadja `s` kisbetűs alakú másolatát. Lásd még: `str.upper()`.
`s.maketrans()`	Az `str.translate()` társa – bővebben lásd a szövegben.
`s.partition(t)`	Egy három karakterláncból álló rendezett sorozatot ad vissza. Az első `s`-nek az a része, amely `t` `s`-beli bal szélső előfordulását megelőzi, a második rész maga a `t` karakterlánc, a harmadik pedig az `s` fennmaradó része. Amennyiben a `t` nem szerepel az `s`-ben, az eredményben `s` és két üres karakterlánc jelenik meg. Amennyiben a `t` jobb szélső előfordulása szerint kívánjuk az `s`-et felbontani, használjuk az `str.rpartition()` tagfüggvényt.
`s.replace(t, u, n)`	Visszaadja `s` másolatát, amelyben a `t` karakterlánc minden (illetve, amennyiben megadtuk, legfeljebb `n` számú) előfordulása helyett `str u` áll.
`s.split(t, n)`	Az eredmény azoknak a karakterláncoknak a listája, amelyeket az `s` `t` szerinti szétvágásával kaphatunk. Ha az `n` paramétert megadjuk, ez korlátozza a vágások számát, ha pedig a `t`-t elhagyjuk, a tagfüggvény a szóközök mentén végzi el a feldarabolást. Amennyiben a műveletet jobbról szeretnénk kezdeni, alkalmazzuk az `str.rsplit()` tagfüggvényt – ennek azonban csak akkor van jelentősége, ha az `n` paramétert megadjuk, és ez kisebb, mint a hasítások lehetséges száma.
`s.splitlines(f)`	Visszaadja azokat a sorokat, amelyeket úgy kaphatunk, ha az `s`-t felhasítjuk a sor vége karakterek mentén. Ezek a karakterek csak abban az esetben maradnak benne az eredményként kapott darabokban, ha az `f` értéke `True`.
`s.startswith(x,` *kezdet*`,` *vég*`)`	Az eredmény `True`, ha az `s` (illetve `s` *kezdet*`:` *vég* szelete) az `x` karakterlánccal, illetve az `x` `tuple` valamelyik elemével kezdődik – egyébként pedig `False`. Lásd még: `str.endswith()`.

2.10. táblázat *Karakterláncok tagfüggvényei, 3. rész*

Használati alak	Leírás
`s.strip(` *karakterek*`)`	Az eredmény `s` másolata, amelyből hiányoznak a kezdő és befejező szóköz karakterek (illetve a *karakterek* karakterlánc elemei). Az `str.lstrip()` csak a karakterlánc elejéről, míg az `str.rstrip()` a végéről vágja le a „felesleget”.
`s.swapcase()`	Az eredmény `s` másolata, amelyben a nagybetűk helyett kisbetűk, a kisbetűk helyett pedig nagybetűk állnak. Lásd még: `str.lower()` és `str.upper()`.
`s.title()`	Az eredmény `s` másolata, amelyben minden szó kezdőbetűje nagybetű, az összes többi pedig kisbetű. Lásd: `str.istitle()`.
`s.translate()`	Az `str.maketrans()` tagfüggvény társa – a részleteket lásd a szövegben.
`s.upper()`	Visszaadja `s` nagybetűs alakú másolatát. Lásd még: `str.lower()`.
`s.zfill(w)`	Visszaadja `s` másolatát, amelyet – amennyiben rövidebb `w`-nél – az elejére írt 0 karakterek sorozatával `w` hosszúságúra egészít ki.

Az `str.join()` tagfüggvényt alkalmazhatjuk a beépített `reversed()` függvénnyel együtt, amennyiben egy karakterlánc tükörképét szeretnénk előállítani (`"".join(reversed(s))`), jóllehet ugyanezt az eredményt tömörebb alakban is elérhetjük az egyelés révén (`s[::-1]`).

A * művelettel másolatokat készíthetünk a karakterláncokról:

```
>>> s = "=" * 5
>>> print(s)
=====
>>> s *= 10
>>> print(s)
==================================================
```

Amint azt a példa jól mutatja, a művelet értékadó változatát is alkalmazhatjuk.*

Amennyiben az `in` tagsági műveletet karakterláncokra alkalmazzuk, a visszatérési értéke `True`, ha a bal oldali karakterlánc-paramétere tartalmazza, illetve megegyezik a jobb oldalival. Ha egy karakterlánc helyzetét szeretnénk megállapítani egy másikon belül, kétféleképpen járhatunk el. Egyrészt alkalmazhatjuk az `str.index()` tagfüggvényt, amely visszaadja a kívánt helyzetet, illetve, amennyiben nem jár sikerrel, `ValueError` kivételt vált ki. A másik lehetőséget az `str.find()` adja, amely szintén a keresett helyzettel tér vissza, ha azonban a karakterlánc nem található, nem kapunk kivételt, ellenben

* A karakterláncok formázására használhatjuk a `%` műveletet is, amely azonban elavult, csupán a Python 2 és a Python 3 közötti megfelelőséget biztosítja, könyvünk példáiban ezért nem szerepel.

a visszatérési érték –1 lesz. Az első paraméter mindkét esetben a keresett karakterlánc, és mindkét tagfüggvény képes egyéb paraméterek fogadására is. A második paraméterrel megadhatjuk a keresés kezdetét, a harmadikkal pedig a végét.

A két tagfüggvény közötti választás jobbára ízlés és a körülmények kérdése, jóllehet ha több előfordulást is keresünk, az `str.index()` általában tisztább kódot eredményez, amint azt az alábbi két egyenértékű kódrészlet is mutatja:

```
def extract_from_tag(tag, line):
    opener = "<" + tag + ">"
    closer = "</" + tag + ">"
    try:
        i = line.index(opener)
        start = i + len(opener)
        j = line.index(closer, start)

        return line[start:j]
    except ValueError:
        return None
```

```
def extract_from_tag(tag, line):
    opener = "<" + tag + ">"
    closer = "</" + tag + ">"
    i = line.find(opener)
    if i != -1:
        start = i + len(opener)
        j = line.find(closer,
                      start)
        if j != -1:
            return line[start:j]
        return None
```

Az `extract_from_tag()` függvény két változata pontosan ugyanazt a viselkedést mutatja. Így ha például az `extract_from_tag("red", "what a <red>rose</red> this is")` alakban hívjuk meg őket, az eredmény mindkét esetben a „rose” lesz. A kivételkezelést alkalmazó változat azonban elválasztja azt a viselkedést, amelyet helyesnek tartunk, a hibák kezelésétől, míg a hibaértéket visszaadó változatnál a kettő összekeveredik.

Az `str.count()`, `str.endswith()`, `str.find()`, `str.rfind()`, `str.index()`, `str.rindex()` és `str.startswith()` tagfüggvények további két paramétert kaphatnak: a keresés kezdetét és végét. Íme két egyenlőség, amelyek mindezt jól mutatják (feltételezzük, hogy `s` karakterlánc):

```
s.count("m", 6) == s[6:].count("m")
s.count("m", 5, -3) == s[5:-3].count("m")
```

Láthatjuk, hogy azok a tagfüggvények, amelyek a *kezdet* és a *vég* paramétereket fogadják, valójában az ezek által meghatározott szeletben keresnek.

Nézzünk most egy újabb egyenlőséget, amely az `str.partition()` viselkedését segít tisztázni:

```
result = s.rpartition("/")
```

```
i = s.rfind("/")
if i == -1:
    result = s, "", ""
else:
    result = s[:i], s[i], s[i + 1:]
```

A bal és a jobb oldalon található kódrészletek nem teljesen egyenértékűek, hiszen az utóbbi létrehoz egy új, `i` nevezetű változót. Vegyük észre, hogy a rendezett sorozatok értékadását mindenféle bűvészkedés nélkül elvégezhettük, és mindkét esetben a `/` jobb szélső előfordulását kerestük. Ha `s` a `"/usr/local/bin/firefox"` karakterlánc, mindkét kódrészlet a `('/usr/local/bin', '/', 'firefox')` eredményt adja.

Az `str.endswith()` (és az `str.startswith()`) függvényt használhatjuk egyetlen karakterlánc paraméterrel, de karakterláncok rendezett sorozatát is átadhatjuk neki. Az alábbi utasításban az `str.endswith()` és az `str.lower()` segítségével képernyőre írjuk a megadott fájl nevét, amennyiben az JPEG típusú:

```
if filename.lower().endswith((".jpg", ".jpeg")):
    print(filename, "is a JPEG image")
```

Az olyan `is*()` tagfüggvények, mint az `isalpha()` és az `isspace()`, `True`-t adnak vissza, ha a karakterlánc, amelyre meghívtuk őket, legalább egy karakterből áll, és minden karaktere eleget tesz a feltételnek. Például:

```
>>> "917.5".isdigit(), "".isdigit(), "-2".isdigit(), "203".isdigit()
(False, False, False, True)
```

Az `is*()` tagfüggvények a Unicode-karakterosztályozás alapján működnek, így ha az `str.isdigit()` tagfüggvényt az `"\N{circled digit two}03"`, illetve a `"②03"` karakterláncokra hívjuk meg, az eredmény mindkét esetben `True` lesz. Ezért semmiféleképpen sem feltételezhetjük, hogy egy karakterlánc egész számmá alakítható csak azért, mert az `isdigit()` tagfüggvénye `True` értékkel tér vissza.

Amikor külső forrásból jutunk hozzá egy karakterlánchoz (más programból, fájlból, a hálózatról, vagy éppen programunk felhasználójától), gyakran előfordul, hogy nem kívánt szóköz karaktereket találunk a kapott karakterlánc elején és végén. Szerencsére könnyen megszabadulhatunk tőlük: a bal oldaliak eltávolítására az `str.lstrip()`, a jobb oldaliakéra az `str.rstrip()` szolgál, ha pedig a karakterlánc mindkét végéről le szeretnénk faragni a felesleget, az `str.strip()` tagfüggvényt használhatjuk. Amennyiben ezeket a tagfüggvényeket egy karakterlánc paraméterrel alkalmazzuk, eltávolítják a benne található karakterek összes előfordulását az eredeti karakterlánc megfelelő (illetve mindkét) végéről. Lássunk erre egy példát:

```
>>> s = "\t no parking "
>>> s.lstrip(), s.rstrip(), s.strip()
('no parking ', '\t no parking', 'no parking')
>>> "<[unbracketed]>".strip("[](){}<>")
'unbracketed'
```

Lehetőségünk van arra is, hogy kicseréljük a karakterlánc egyes részeit – erre szolgál az `str.replace()`, amely két karakterláncot vár paraméterként, és visszatérési értékként az eredeti karakterlánc egy másolatát adja vissza, amelyben az első paraméter minden

előfordulását a másodikkal cseréli fel. Amennyiben a második karakterlánc üres, a művelet eredményeként töröljük az első paraméter összes előfordulását az eredeti karakterláncból. Az `str.replace()` és más karakterlánc-kezelő tagfüggvények használatát szemlélteti a fejezet vége felé bemutatott `csv2html.py` nevű példaprogramunk.

`csv2-html.py` example
☞ 91

Gyakran van szükségünk egy karakterlánc részekre darabolására. Tegyük fel például, hogy van egy adatokat tartalmazó szövegfájlunk, amelyben minden sor egy bejegyzést tartalmaz, a bejegyzések mezőit pedig csillagok választják el egymástól – ebből a fájlból szeretnénk kiolvasni az egyes adatmezőket. A feladatra nagyszerűen alkalmas az `str.split()` tagfüggvény, amelynek első paramétereként a mezőket elválasztó karakterláncot kell megadnunk, valamint – amennyiben erre szükségünk van – második paraméterként meghatározhatjuk a hasítások számának felső határát. Ha ezt nem tesszük meg, a tagfüggvény annyi részre bontja a karakterláncot, amennyi hasítás csak elvégezhető. Lássunk most egy példát:

```
>>> record = "Leo Tolstoy*1828-8-28*1910-11-20"
>>> fields = record.split("*")
>>> fields
['Leo Tolstoy', '1828-8-28', '1910-11-20']
```

A születés és a halál dátumára ismét alkalmazhatjuk az `str.split()` tagfüggvényt, így kiszámíthatjuk a szóban forgó személy hozzávetőleges életkorát (legfeljebb egy évnyi eltéréssel):

```
>>> born = fields[1].split("-")
>>> born
['1828', '8', '28']
>>> died = fields[2].split("-")
>>> print("lived about", int(died[0]) - int(born[0]), "years")
lived about 82 years
```

Az `int()` függvény használatán kívül – amire annak érdekében volt szükség, hogy megkapjuk az évek számának egész értékét a karakterlánc alakból – a kódban nincs semmilyen csavar. Mindazonáltal, érdemes megjegyeznünk, hogy az évszámokat közvetlenül is megkaphattuk volna a `fields` listából, valahogy így: `year_born = int(fields[1].split("-")[0])`.

A 2.8., 2.9. és 2.10. táblázatokból kihagytuk az `str.maketrans()` és az `str.translate()` tagfüggvényeket. Az előbbi használatával fordítótáblákat készíthetünk, amelyek karaktereket feleltetnek meg karaktereknek. A tagfüggvény egy-, két- és háromparaméteres alakban is alkalmazható, most azonban csak a legegyszerűbb (kétparaméteres) hívását mutatjuk be, ahol az első paraméter az átváltandó karaktereket tartalmazza, a második pedig a nekik megfeleltetetteket. Fontos, hogy a két karakterlánc hossza azonos legyen. Az `str.translate()` paraméterként egy fordítótáblát vár, és az eredeti karakterlánc eszerint átalakított változatát adja

vissza. Az alábbiakban bemutatjuk, miként alakíthatunk át bengáli számjegyeket tartalmazó karakterláncokat olyanokká, amelyek az angol számjegyeket alkalmazzák:

```
table = "".maketrans("\N{bengali digit zero}"
    "\N{bengali digit one}\N{bengali digit two}"
    "\N{bengali digit three}\N{bengali digit four}"
    "\N{bengali digit five}\N{bengali digit six}"
    "\N{bengali digit seven}\N{bengali digit eight}"
    "\N{bengali digit nine}", "0123456789")
print("20749".translate(table))                    # az eredmény: 20749
print("\N{bengali digit two}07\N{bengali digit four}"
    "\N{bengali digit nine}".translate(table))     # az eredmény: 20749
```

Vegyük észre, hogy az `str.maketrans()` hívása során, valamint a második `print()` utasításban kihasználtuk a Python literális karakterláncoknál alkalmazott összefűzési eljárását, így a karakterláncokat nyugodt lélekkel külön sorba írhatjuk anélkül, hogy ki kellene hagynunk az új sorokat, vagy kifejezett összefűzést kellene alkalmaznunk.

Az `str.maketrans()` tagfüggvényt azért üres karakterláncra hívtuk meg, mert ez valójában egyáltalán nem számít – egyszerűen feldolgozza a paramétereit, és visszaadja a fordítótáblát.* Az `str.maketrans()` és az `str.translate()` tagfüggvények alkalmazhatók karakterek törlésére is, csak adjuk át a nem kívánt karaktereket egy karakterlánc formájában az `str.maketrans()` harmadik paramétereként. Ha ennél kifinomultabb karakterátalakításra van szükségünk, készíthetünk egyéni kodeket is – erről bővebben a `codecs` modul leírásában olvashatunk.

A Python rendelkezik néhány további könyvtármodullal, amelyek szintén tartalmaznak a karakterláncok kezelésére alkalmas függvényeket. A korábbiakban már röviden említettük a `unicodedata` modult, amelynek a használatát hamarosan be is mutatjuk. Érdemes megismerkednünk továbbá a `difflib` modullal, amellyel különbségeket kereshetünk fájlok, illetve karakterláncok között, valamint az `io` modul `io.StringIO` osztályával, amely lehetővé teszi, hogy úgy írjuk és olvassuk a karakterláncok adatait, mintha fájlok lennének. Nem szabad szó nélkül elmennünk a `textwrap` modul mellett sem, amellyel lehetőséget kapunk karakterláncok tördelésére és kitöltésére, továbbá itt a `string` modul is, amelyben hozzájuthatunk néhány hasznos állandóhoz, mint az `ascii_letters` vagy az `ascii_lowercase`. Ezeknek a moduloknak a használatára az 5. fejezetben láthatunk majd néhány példát. Mindezek mellett a Python `re` modulja jelentős támogatást ad a szabályos kifejezések használatához – róluk a 12. fejezetben szólunk bővebben.

*Objektumközpontú programozók figyelmébe: az `str.maketrans()` statikus tagfüggvény.

Karakterláncok formázása az str.format() tagfüggvénnyel

Az `str.format()` tagfüggvény igen rugalmas és hatékony módot ad karakterláncok létrehozására. A használata nem okoz gondot az egyszerűbb esetekben, de ha bonyolultabb formázást szeretnénk alkalmazni, fontos, hogy tisztában legyünk a pontos használati alakjával.

Az `str.format()` visszatérési értéke az eredeti karakterlánc, amelyben a *cseremezők* helyén a megadott paraméterek szerepelnek a megfelelő formázással:

```
>>> "The novel '{0}' was published in {1}".format("Hard Times", 1854)
"The novel 'Hard Times' was published in 1854"
```

A cseremezőket kapcsos zárójelekben elhelyezett mezőnevekkel azonosítjuk. Amennyiben a mezőnév egyszerű egész szám, a rendszer az `str.format()` megfelelő sorszámú paraméterét teszi a helyére. Így esetünkben a 0 névvel azonosított mező helyére az első paraméter kerül, az 1-gyel azonosított helyére pedig a második.

Ha a formázott karakterláncban kapcsos zárójeleket is fel szeretnénk tüntetni, egyszerűen duplázzuk meg őket – így teszünk az alábbi példában is:

```
>>> "{{{0}}} {1} ;-}}".format("I'm in braces", "I'm not")
"{I'm in braces} I'm not ;-}"
```

Ha megpróbálunk összefűzni egy karakterláncot és egy számot, a Python teljesen jogosan `TypeError` kivételt vált ki. Az `str.format()` használatával azonban könnyedén elérhetjük, amit szerettünk volna:

```
>>> "{0}{1}".format("The amount due is $", 200)
'The amount due is $200'
```

Az `str.format()` lehetőséget ad karakterláncok összefűzésére is (jóllehet erre az `str.join()` a legalkalmasabb):

```
>>> x = "three"
>>> s ="{0} {1} {2}"
>>> s = s.format("The", x, "tops")
>>> s
'The three tops'
```

A fenti példában használatba vettünk néhány karakterlánc-változót, de a következőkben jobbára literális értékekkel dolgozunk az `str.format()` bemutatásánál, egyszerűen a kényelem kedvéért. Mindazonáltal sose feledjük, hogy ahol literális értéket használunk, ugyanúgy állhatna karakterlánc-változó is.

A cseremező az alábbiak közül bármelyik alakot felveheti:

```
{mezőnév}
{mezőnév!átalakítás}
{mezőnév:formátumleírás}
{mezőnév!átalakítás:formátumleírás}
```

Fontos megjegyeznünk továbbá, hogy maguk a cseremezők is tartalmazhatnak további cseremezőket. A beágyazott cseremezők nem rendelkezhetnek semmilyen formázással, a szerepük az automatizált formázási eljárásokban mutatkozik meg. Minderre példát is láthatunk majd a formátumok meghatározásának részletes tárgyalásánál. A következőkben a cseremező egyes részeit vesszük górcső alá – elsőként a mezőneveket.

Mezőnevek

A mezőnevek lehetnek az `str.format()` paramétereinek megfelelő egészek vagy a tagfüggvény kulcsszó-paramétereinek nevei. Ez utóbbiakról a 4. fejezetben szólunk bővebben, de a kezelésük egyszerű, így a teljesség kedvéért itt is bemutatjuk a használatukat:

```
>>> "{who} turned {age} this year".format(who="She", age=88)
'She turned 88 this year'
>>> "The {who} was {0} last week".format(12, who="boy")
'The boy was 12 last week'
```

Első példánkban két kulcsszó-paraméter – a `who` és az `age` – szerepel, míg a másodikban egy helyzeti paraméter (ilyeneket használtunk a korábbiakban) és egy kulcsszó-paraméter. Vegyük észre, hogy a paraméterek listájában a helyzeti paraméterek mindig megelőzik a kulcsszó-paramétereket, jóllehet a formázott karakterláncban a sorrendjük természetesen tetszőleges lehet.

A mezőnevek utalhatnak gyűjteményes adattípusokra – így például listákra – is. Ilyenkor egy sorszámmal (nem szelettel!) azonosíthatjuk az adott elemet:

```
>>> stock = ["paper", "envelopes", "notepads", "pens", "paper clips"]
>>> "We have {0[1]} and {0[2]} in stock".format(stock)
'We have envelopes and notepads in stock'
```

A 0 érték helyzeti paraméterre utal, így a `{0[1]}` a `stock` listaparaméter második, a `{0[2]}` pedig a harmadik elemét jelöli.

`dict` típus
☞ 121

A későbbiekben megismerkedünk majd a Python-szótárakkal, amelyek kulcs-érték párokat tartalmaznak. Mivel alkalmazhatók az `str.format()` tagfüggvényben, ismét csak a teljesség kedvéért bemutatunk egy példát a használatukra. Ha mindez jelen pillanatban nem világos, ne aggódjunk: a 3. fejezetben kitisztul a kép.

```
>>> d = dict(animal="elephant", weight=12000)
>>> "The {0[animal]} weighs {0[weight]}kg".format(d)
'The elephant weighs 12000kg'
```

Miként a listák és a rendezett sorozatok elemeit egész értékű sorszámaikkal érhetjük el, a szótárak elemeihez a kulcsok révén juthatunk hozzá. Elérhetjük továbbá a névvel ellátott tulajdonságokat is. Feltéve, hogy korábban betöltöttük a `math` és `sys` modulokat, végrehajthatjuk az alábbi utasítást:

```
>>> "math.pi={0.pi} sys.maxunicode={1.maxunicode}".format(math, sys)
'math.pi=3.14159265359 sys.maxunicode=65535'
```

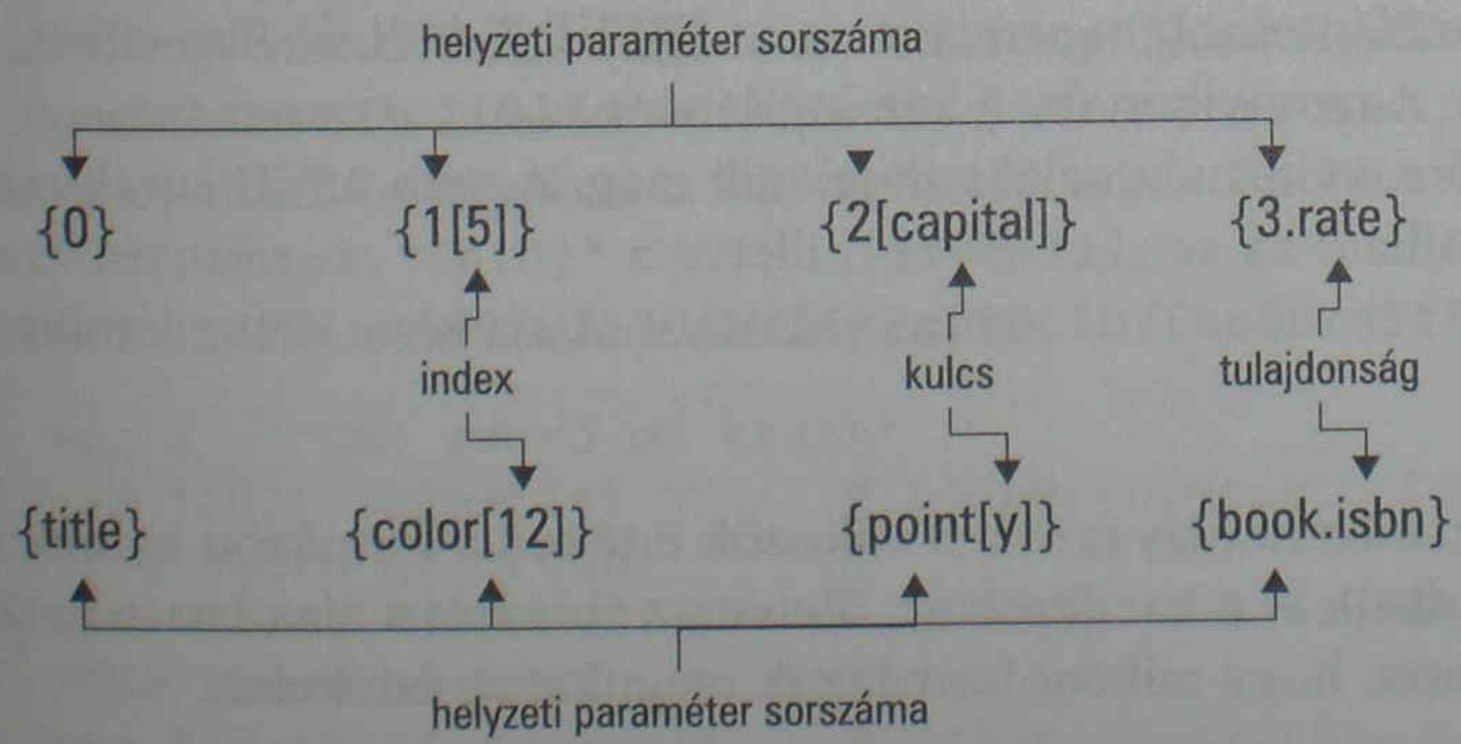
Összegezve a látottakat elmondhatjuk, hogy a mezőnevek használati alakja lehetővé teszi, hogy hivatkozzunk bármely helyzeti, illetve kulcsszó-paraméterre, amelyet az `str.format()` tagfüggvénynek átadtunk. Amennyiben a paraméterek gyűjtemény típusúak – például listák vagy szótárak –, illetve rendelkeznek tulajdonságokkal, a kívánt részt a `[]` vagy a `.` jelölésmóddal érhetjük el. Minderről áttekintést kaphatunk a 2.5. ábrán.

Átalakítások

decimális számok
☞ 58

A `decimal.Decimal` számok tárgyalásánál láthattuk, hogy az ilyen típusú értékek kétféleképpen írhatók ki. Frissítsük fel az emlékezetünket két rövidke példával:

```
>>> decimal.Decimal("3.4084")
Decimal('3.4084')
>>> print(decimal.Decimal("3.4084"))
3.4084
```



2.5. ábra
Formátumparaméterek hivatkozásai mezőnevekben

Az első példában a `decimal.Decimal` értékünket az ábrázolási alakjában írtuk ki – ebből az alakból a Python értelmezője képes újra előállítani magát az objektumot. A Python-programok képesek Python-kódrészletek, sőt teljes programok értelmezésére, így ez a lehetőség olykor hasznosnak bizonyulhat. Nem minden objektum képes olyan ábrázolást biztosítani, amelyből valóban vissza is állítható – ezt jelzi, ha az eredmény kisebb-nagyobb jelek (csúcsos zárójelek) között áll. Így például a `sys` modul ábrázolási alakja a `"<module 'sys' (built-in)>"` karakterlánc.

eval()
☞ 349

A `decimal.Decimal` értékeket emellett karakterlánc alakban is megjeleníthetjük. Ez az alak emberi „fogyasztásra" készült, vagyis elvárható, hogy könnyen értelmezhető legyen. Ha egy adattípus nem rendelkezik karakterlánc alakkal, de valahol szükség lenne erre, a Python az ábrázolási alakot használja.

A Python beépített adattípusai ismerik az `str.format()` tagfüggvényt, így ha valamelyiküket paraméterként átadjuk nekik, a megfelelő karakterláncot adják vissza saját maguk megjelenítésére. Nem nehéz feladat az sem, hogy az egyedi adattípusainkat kiegészítsük az `str.format()` támogatásával – erről a 6. fejezetben szólunk bővebben. Ráadásul az adattípusok eredeti viselkedése is felülbírálható, vagyis magunk határozhatjuk meg, hogy ilyen esetben a karakterlánc vagy az ábrázolási alakot adják vissza. Minderre a mezőben az átalakítás beállításával van lehetőségünk. Jelenleg három érték közül választhatunk: az `s` a karakterlánc alakot adja, az `r` az ábrázolási alakot, az `a` pedig szintén ez utóbbit, amelyben azonban csak ASCII-karakterek szerepelnek. Íme egy példa:

```
>>> "{0} {0!s} {0!r} {0!a}".format(decimal.Decimal("93.4"))
"93.4 93.4 Decimal('93.4') Decimal('93.4')"
```

Ez esetben a `decimal.Decimal` érték karakterlánc alakja ugyanazt az eredményt adja, mint ami amúgy is megjelent volna az `str.format()` hívásakor. Mindemellett ebben a példában az ábrázolási alak eredeti és ASCII formátuma között sincs különbség, hiszen mindkettőben csak ASCII-karakterek szerepelnek.

A következő példánk már kacifántosabb, ugyanis most a „翻訳で失われる" film címét tároltuk a `movie` változóban. Amennyiben ezt a karakterláncot `"{0}".format(movie)` kifejezéssel írjuk a képernyőre, változatlan alakban jelenik meg. A nem ASCII-karakterek elkerülésére egyaránt használható az `ascii(movie)`, illetve a `"{0!a}".format(movie)` kifejezés – mindkettő a `'\u7ffb\u8a33\u3067\u5931\u308f\u308c\u308b'` karakterláncot adja eredményül.

Az eddigekben megtanultuk, miként helyezzük el változók értékeit a formázott karakterláncban, és hogyan kényszerítsük ki a karakterlánc, illetve az ábrázolási alak használatát. Mindezek után rátérhetünk arra, hogy miként formázzuk magukat az értékeket.

Formátumleírások

Az egészek, lebegőpontos számok és karakterláncok alapértelmezett formázása jobbára tökéletesen megfelel az igényeinknek. Ha azonban finomabb szabályozásra vágyunk, a formátumleírással könnyedén célt érhetünk. A tisztább átláthatóság érdekében külön tárgyaljuk a karakterláncok, az egészek, valamint a lebegőpontos számok formázását – mielőtt azonban elmélyednénk a részletekben, a 2.6. ábrán bemutatjuk a formázás általános alakját.

:	*kitöltés*	*igazítás*	*előjel*	*#*	*0*	*szélesség*	*.pontosság*	*típus*
	Tetszőleges karakter, kivéve }	< balra > jobbra ^ középre = számok esetén kitöltés az előjel és a számjegyek között	+ előjel kikényszerítése – előjel szükség esetén „ „ szóköz vagy – (amelyik megfelelő)	egészek 0b, 0o vagy 0x előtagja	számok kitöltése nullákkal	legkisebb mezőszélesség	Karakterláncoknál a legnagyobb mezőszélesség, lebegőpontos számoknál a tizedesjegyek száma	int b, c, d, n, o, x, X; float e, E, f, g, G, n, %

2.6. ábra

A formátumleírás általános alakja

Karakterláncok esetén a kitöltő karaktert, a mezőn belüli igazítást, valamint a legkisebb és legnagyobb mezőszélességet határozhatjuk meg.

A karakterláncok formátumleírása egy kettősponttal (:) veszi kezdetét, ezt követi egy (nem kötelező) karakterpár – egy kitöltő karakter (ami nem lehet }), valamint egy igazítási karakter (< a balra, ^ a középre és > a jobbra igazításhoz). Ezt követi két (szintén nem kötelezően megadandó) egész szám: a legkisebb és egy pont után a legnagyobb mezőszélesség értéke.

Fontos megjegyeznünk, hogy ha megadunk egy kitöltő karaktert, az igazítás meghatározásával sem maradhatunk adósak. Az előjel és a típus megadását elhagytuk, hiszen ezek nincsenek hatással a karakterláncokra. Mivel a formátumot meghatározó egyik elem megadása sem kötelező, önmagában is feltüntethetjük a kettőspontot – ez nem árt, de nem is használ.

A sok elmélet után lássunk néhány példát:

```
>>> s = "The sword of truth"
>>> "{0}".format(s)          # alapértelmezett formázás
'The sword of truth'
>>> "{0:25}".format(s)       # a legkisebb szélesség 25
'The sword of truth '
>>> "{0:>25}".format(s)      # jobbra igazítás, a legkisebb szélesség 25
' The sword of truth'
>>> "{0:^25}".format(s)      # középre igazítás, a legkisebb szélesség 25
' The sword of truth '
>>> "{0:-^25}".format(s)     # kitöltés a - karakterrel, középre igazítás,
                             # a legkisebb szélesség 25
'---The sword of truth----'
>>> "{0:.<25}".format(s)     # kitöltés a - karakterrel, balra igazítás,
                             # a legkisebb szélesség 25
'The sword of truth.......'
>>> "{0:.10}".format(s)      # a legnagyobb szélesség 10
'The sword '
```

Az utolsó előtti példában mindenképpen meg kellett adnunk a balra igazítást – jóllehet ez az alapértelmezés. Ha a < jelet kihagytuk volna, a formátumleírás mindössze a `:.25` lenne, ami nem jelent egyebet 25 karakteres legnagyobb szélességnél. Amint a korábbiakban megjegyeztük, lehetőségünk van arra is, hogy cseremezőket helyezzünk el a formátumleírásokon belül is, ami alkalmat ad arra, hogy számított formázást használjunk. Az alábbiakban két módot is bemutatunk arra, hogy miként állíthatjuk be egy karakterlánc legnagyobb szélességét a `maxwidth` változóval:

```
>>> maxwidth = 12
>>> "{0}".format(s[:maxwidth])
'The sword of'
>>> "{0:.{1}}".format(s, maxwidth)
'The sword of'
```

Első példánkban a szokásos szeletelési módszert láthatjuk viszont, a másodikban azonban beágyazott cseremezőt alkalmazunk.

Egészek esetén a formátumleírással szabályozhatjuk a kitöltő karaktert, a mezőn belüli igazítást, az előjelet, a mező legkisebb szélességét, valamint a számábrázolás alapját. Az egészek formátumleírása egy kettősponttal veszi kezdetét, ezt követi egy (nem kötelező) karakterpár – egy kitöltő karakter (ami nem lehet }), valamint egy igazítási karakter (< a balra, ^ a középre, > a jobbra igazításhoz, valamint = az előjel és a szám közötti tér kitöltéséhez). Ezt követi egy (szintén nem kötelező) előjelkarakter – + megadásánál a rendszer mindig kiírja az előjelet, - esetén csak a negatív előjelet tünteti fel, míg szóköz megadásánál a pozitív előjelnél szóközt, a negatív előjelnél pedig – jelet ad. Ez után következhet a legkisebb szélességet megadó egész szám – ezt megelőzheti egy # karakter a számábrázolás alapjának feltüntetésére (bináris, oktális vagy hexadecimális számoknál), valamint a 0 a nullákkal való kitöltéshez („0-kitöltés"). Ha a kimenet számábrázolását nem tízes alapon képzeljük el, meg kell adnunk egy típuskaraktert is – ez bináris számok esetén `b`, oktálisoknál `o`, kisbetűs hexadecimálisoknál `x`, nagybetűs hexadecimálisoknál `X`, sőt a teljesség kedvéért a tízes számrendszert jelző `d` is megengedett. Létezik két további típuskarakter is: a `c` megadása esetén a rendszer a számhoz tartozó Unicode-karaktert írja ki, az `n` esetében pedig a kimenet igazodik a helyi viszonyokhoz. A 0-kitöltéshez kétféleképpen is hozzájuthatunk:

```
>>> "{0:0=12}".format(8749203)  # a kitöltés karaktere 0, a legkisebb
                                # szélesség 12
'000008749203'
>>> "{0:0=12}".format(-8749203) # a kitöltés karaktere 0, a legkisebb
                                # szélesség 12
'-00008749203'
>>> "{0:012}".format(8749203)   # 0-kitöltés, a legkisebb szélesség 12
'000008749203'
>>> "{0:012}".format(-8749203)  # 0-kitöltés, a legkisebb szélesség 12
'-00008749203'
```

Az első két példában a kitöltő karakter a 0 volt, emellett az előjel és a szám közötti kitöltést alkalmaztuk, míg a második két példában 0-kitöltést használtunk 12 karakteres legkisebb szélességgel. Lássunk most néhány példát az igazításra is:

```
>>> "{0:*<15}".format(18340427)  # * kitöltés, balra igazítás,
                                 # a legkisebb szélesség 15
'18340427*******'
>>> "{0:*>15}".format(18340427)  # * kitöltés, jobbra igazítás,
                                 # a legkisebb szélesség 15
'*******18340427'
>>> "{0:*^15}".format(18340427) # * kitöltés, középre igazítás,
                                # a legkisebb szélesség 15
'***18340427****'
>>> "{0:*^15}".format(-18340427) # * kitöltés, középre igazítás,
                                 # a legkisebb szélesség 15
'***-18340427***'
```

Most pedig következzék pár példa az előjelkarakterek szerepére:

```
>>> "[{0: }] [{1: }]".format(539802, -539802)  # szóköz vagy - jel
'[ 539802] [-539802]'
>>> "[{0:+}] [{1:+}]".format(539802, -539802)  # minden esetben előjel
'[+539802] [-539802]'
>>> "[{0:-}] [{1:-}]".format(539802, -539802)  # - előjel, amennyiben
                                               # szükség van rá
'[539802] [-539802]'
```

Lássunk néhány példát a típuskarakterek alkalmazására is:

```
>>> "{0:b} {0:o} {0:x} {0:X}".format(14613198)
'110111101111101011001110 67575316 deface DEFACE'
>>> "{0:#b} {0:#o} {0:#x} {0:#X}".format(14613198)
'0b110111101111101011001110 0o67575316 0xdeface 0XDEFACE'
```

Az egészek esetében nem adhatunk meg legnagyobb szélességet, ez ugyanis számjegyek esetleges levágásával járna, ami tökéletesen értelmetlenné tenné a szám használatát.

Utolsóként említjük az egészek (nemkülönben a lebegőpontos számok) formázási karakterei között az `n`-t, amely egészek esetén ugyanazzal a hatással jár, mint a `d` karakter, lebegőpontos számok esetén pedig a `g` használható helyette. Az `n` jelentőségét az adja, hogy figyelembe veszi a helyi beállításokat („lokál", locale), eszerint alkalmazza a megfelelő tizedeskaraktert, és a számjegyeket is ennek figyelembe vételével csoportosítja. Az alapértelmezett beállítás a „C lokál", amelyben a tizedeskarakter a pont, a csoportosítás karaktere pedig egy üres karakterlánc. Felhasználóink helyi beállításait tiszteletben tarthatjuk, ha programjainkban az alábbi két sort futtatjuk elsőként:*

```
import locale
locale.setlocale(locale.LC_ALL, "")
```

* Többszálas programokban jobb, ha a `locale.setlocale()` függvényt csak egyszer hívjuk meg, méghozzá a program indításakor, minden további szál születése előtt, mivel a függvény általában nem szálbiztos.

Ha lokálként egy üres karakterláncot adnánk át, a Python megpróbálja automatikusan meghatározni a felhasználó nyelvi környezetét (például a LANG környezeti változó vizsgálatával), illetve ha ez nem sikerül, a C lokált használja. Lássunk néhány példát, amelyek a különböző lokálok hatását mutatják egy egész és egy lebegőpontos számon:

```
x, y = (1234567890, 1234.56)
locale.setlocale(locale.LC_ALL, "C")
c = "{0:n} {1:n}".format(x, y) # c == "1234567890 1234.56"
locale.setlocale(locale.LC_ALL, "en_US.UTF-8")
en = "{0:n} {1:n}".format(x, y) # en == "1,234,567,890 1,234.56"
locale.setlocale(locale.LC_ALL, "de_DE.UTF-8")
de = "{0:n} {1:n}".format(x, y) # de == "1.234.567.890 1.234,56"
```

Jóllehet az `n` igen hasznos lehet egész számok esetében, lebegőpontos értékeknél sokat veszt a hasznosságából, hiszen mihelyt e számok „megnőnek”, amúgy is exponenciális alakot vesznek fel.

Lebegőpontos számok esetén a formátumleírás lehetővé teszi, hogy szabályozzuk a kitöltő karaktert, a mezőn belüli igazítást, az előjelet, a mező legkisebb szélességét, a tizedesjel után álló számjegyek számát, valamint azt, hogy a számot hagyományos, exponenciális vagy százalék alakban tüntessük fel.

A lebegőpontos számok formátumleírása csaknem megegyezik az egészekéével – mindössze a végén találhatunk két különbséget. A legkisebb szélesség (nem kötelezően megadandó) értékét követően egy pont után megadhatjuk, hogy hány tizedesjegyet szeretnénk feltüntetni. A formátumleírás végén feltüntethetünk egy típuskaraktert is – az `e` esetén exponenciális alakot kapunk kis *e* betűvel, `E` esetén szintén exponenciális alakhoz jutunk nagy *E* betűvel, `f` esetén a szabványos lebegőpontos alakot kapjuk, míg a `g` az „általános” alakot jelzi – ez megegyezik az `f` hatásával, hacsak a szám nem túlzottan nagy, ilyenkor viszont az `e` esetéhez jutunk –, a `G` pedig ennek egy alig eltérő változatát (`f`, illetve `E`). Használhatjuk még a `%` jelet is – ez esetben a szám 100-szorosát kapjuk `f` formátumban, és az eredmény után egy `%` jel áll. Íme néhány példa az exponenciális és az egyszerű alakok bemutatására:

```
>>> amount = (10 ** 3) * math.pi
>>> "[{0:12.2e}] [{0:12.2f}]".format(amount)
'[    3.14e+03] [     3141.59]'
>>> "[{0:*>12.2e}] [{0:*>12.2f}]".format(amount)
'[****3.14e+03] [*****3141.59]'
>>> "[{0:*>+12.2e}] [{0:*>+12.2f}]".format(amount)
'[***+3.14e+03] [****+3141.59]'
```

Első példánkban a legkisebb szélesség 12 karakter, és a tizedespont után két számjegy áll. A második példa az elsőre épít, kiegészítve a * kitöltéssel. Márpedig ha kitöltést alkalmazunk, igazítást is meg kell adnunk – ez esetben a jobbra igazítás mellett döntöttünk (annak ellenére meg kell adnunk, hogy számok esetében ez az alapértelmezés). A harmadik példa az előző kettőre épít: a + karakterrel kikényszeríti az előjel megjelenítését. Könyvünk írásának idején a Python nem támogatta közvetlen módon a komplex számok formázását,

ezen az apróságon azonban könnyedén túllendülhetünk, ha külön lebegőpontos számként formázzuk a valós és a képzetes részt, valahogy így:

```
>>> "{0.real:.3f}{0.imag:+.3f}j".format(4.75917+1.2042j)
'4.759+1.204j'
>>> "{0.real:.3f}{0.imag:+.3f}j".format(4.75917-1.2042j)
'4.759-1.204j'
```

A komplex szám részeit külön-külön értük el, és olyan lebegőpontos alakban írtuk ki, amelyben három tizedesjegy szerepel. Kikényszerítettük továbbá a képzetes rész előjelének megjelenítését, valamint feltüntettük a *j* karaktert is.

Példa: print_unicode.py

Az előzőekben közelebbről is megismerkedtünk az `str.format()` tagfüggvény formátumleírásaival, emellett számos kódrészletet láthattunk, amelyek segítettek a tanultak jobb megértésében. A következőkben egy apró, ám hasznos példaprogramban mutatjuk be az `str.format()` használatát, így a formátumleírásokat végre életszagú környezetben is viszontláthatjuk. Példánkban megjelenik néhány, a korábbiakban bemutatott karakterlánc-tagfüggvény, sőt a `unicodedata` modul egy függvényét is használatba vesszük.*

7. fejezet: *Fájlkezelés* ☞ 287

Programunk mindössze 25 futtatható kódsorból áll. Betölti a `sys` és a `unicodedata` modulokat, továbbá meghatároz egy egyedi függvényt, amelynek a neve `print_unicode_table()`. Első lépésként kipróbáljuk a programot, hogy lássuk, mit is tesz, majd megvizsgáljuk a program végén álló kódot, ahol az adatfeldolgozás valójában elkezdődik, végül pedig bemutatjuk az egyedi függvényt is.

```
print_unicode.py spoked
decimal    hex    chr                name
-------  -----  ---  --------------------------------
  10018   2722   ✢   Four Teardrop-Spoked Asterisk
  10019   2723   ✣   Four Balloon-Spoked Asterisk
  10020   2724   ✤   Heavy Four Balloon-Spoked Asterisk
  10021   2725   ✥   Four Club-Spoked Asterisk
  10035   2733   ✳   Eight Spoked Asterisk
  10043   273B   ✻   Teardrop-Spoked Asterisk
  10044   273C   ✼   Open Centre Teardrop-Spoked Asterisk
  10045   273D   ✽   Heavy Teardrop-Spoked Asterisk
  10051   2743   ❃   Heavy Teardrop-Spoked Pinwheel Asterisk
  10057   2749   ❉   Balloon-Spoked Asterisk
  10058   274A   ❊   Eight Teardrop-Spoked Propeller Asterisk
  10059   274B   ❋   Heavy Eight Teardrop-Spoked Propeller Asterisk
```

* A program feltételezi, hogy a konzolunk Unicode UTF-8 kódolást alkalmaz. Sajnos a Windows konzol UTF-8 támogatása nem a legjobb, a MacOS X konzol pedig alapértelmezés szerint az Apple Roman kódolást alkalmazza. Áthidaló megoldásként elkészítettük a program `print_unicode_uni.py` nevű változatát is, amely a kimenetét egy fájlba írja, amelyet azután megnyithatunk egy az UTF-8-at értelmezni képes szerkesztőben, mondjuk az IDLE-ben.

Programunk, amennyiben paraméterek nélkül futtatjuk, egy az összes Unicode-karaktert tartalmazó táblázatot jelenít meg, a szóköz karaktertől kezdve egészen a legmagasabb elérhető kódponttal rendelkező karakterig. Ha egy paramétert is megadunk – így tettünk a példában is – a program csak azokat a sorokat írja ki a táblázatból, amelyekben a Unicode-karakternév kisbetűs változata tartalmazza a megadott paramétert.

```
word = None
if len(sys.argv) > 1:
    if sys.argv[1] in ("-h", "--help"):
        print("usage: {0} [string]".format(sys.argv[0]))
        word = 0
    else:
        word = sys.argv[1].lower()
if word != 0:
    print_unicode_table(word)
```

A betöltés és a `print_unicode_table()` függvény létrehozása után a futtatás során elérkezünk a fenti kódrészlethez. Kezdetben feltételezzük, hogy a felhasználó nem adott meg semmilyen paramétert a parancssorban. Amennyiben a parancssorban szerepelt paraméter, és ez a `-h` vagy a `--help`, tájékoztatást adunk a program használatáról, és a `word` változónak a `0` értéket adjuk, jelezve, hogy mindennel elkészültünk. Ha a paraméter értéke ettől eltérő, kisbetűs változatát tároljuk a `word` változóban. Ez után, ha a `word` értéke nem 0, kiírjuk a táblázatot.

A felhasználói tájékoztató kiírásakor olyan formátumleírást használunk, amelyben csak a formátum neve áll – esetünkben a paraméter sorszáma. Az ott alkalmazott sor helyett ezt is írhattuk volna:

```
print("usage: {0[0]} [string]".format(sys.argv))
```

Itt az első `0` a kívánt paraméter sorszáma, a `[0]` pedig a paraméteren *belüli* sorszám, hiszen a `sys.argv` egy lista.

```
def print_unicode_table(word):
    print("decimal  hex  chr  {0:^40}".format("name"))
    print("-------- ----- ---  {0:-<40}".format(""))
    code = ord(" ")
    end = sys.maxunicode
    while code < end:
        c = chr(code)
        name = unicodedata.name(c, "*** unknown ***")
        if word is None or word in name.lower():
            print("{0:7} {0:5X} {0:^3c} {1}".format(
                  code, name.title()))
        code += 1
```

A jobb átláthatóság érdekében elhelyeztünk a kódban néhány üres sort is. A függvény törzsének első két sora a címsorokat írja ki. Az első `str.format()` hívás a „name" feliratot jeleníti meg egy 40 karakter széles mezőben középre igazítva, míg a második egy szintén 40 karakter széles üres karakterláncot ír ki, a „–" karakterrel kitöltve, balra igazítva. (Mivel kitöltő karaktert használunk, mindenképpen meg kell adnunk az igazítást is.) A második sort így is írhattuk volna:

```
print("-------  -----  ---  {0}".format("-" * 40))
```

Itt a * többszöröző műveletet alkalmaztuk a megfelelő karakterlánc létrehozására, és az eredményt egyszerűen beillesztettük a formázott karakterláncba. Harmadik megoldásként egyszerűen beírhattunk volna 40 darab „–" karaktert, és az így kapott literális karakterláncot használhattuk volna.

A Unicode-kódpontokat a `code` változóval követjük nyomon – elsőként a szóköz kódpontját (0x20) tároljuk benne. Az `end` változóban a legnagyobb elérhető Unicode-kódpontot tároljuk – ez változó lehet attól függően, hogy a Python fordításakor az UCS-2 vagy az UCS-4 karakterkódolás használata mellett döntöttünk.

Karakterkódolások
☞ 86

A `while` ciklus belsejében a `chr()` függvénnyel hozzájutunk ahhoz a Unicode-karakterhez, amelyik megfelel az aktuális kódpontnak. A `unicodedata.name()` függvénnyel megkaphatjuk az átadott karakter Unicode-nevét. A függvény képes egy második paraméter fogadására is, amelyben meghatározhatjuk az alkalmazandó nevet, amennyiben a Unicode maga cserben hagy.

Ha a felhasználó nem adott meg semmilyen szót (a `word` értéke `None`), illetve megadott, és ez szerepel valamelyik Unicode-név kisbetűs változatában, kiírjuk a megfelelő sort.

Jóllehet az `str.format()` tagfüggvénynek mindössze egyszer adjuk át a `code` változót, a formázott karakterláncban háromszor is használjuk – először akkor, amikor a `code` értékét egy 7 karakter széles mezőben szeretnénk megjeleníteni (a kitöltő karakter alapértelmezés szerint a szóköz, így nem kell külön megadnunk), másodszor akkor, amikor a `code` értékét egy 5 karakter széles mezőben nagybetűs hexadecimális számként írjuk ki, végül harmadszor akkor, amikor a `code` értékéhez tartozó Unicode-karaktert írjuk ki. Ez utóbbi esetben a „c" formátumleírót használjuk, és az eredményt egy legalább 3 karakter széles mezőben középre igazítjuk. Vegyük észre, hogy az első formátumleírásban nem kellett megadnunk a „d" beállítást, ugyanis egész paraméterek esetén ez az alapértelmezés. A második paraméter a karakter Unicode-neve a „title" formátummal ábrázolva, ahol a szavak első betűi nagybetűk, az összes többi pedig kisbetű.

Ennyi gyakorlás és példa módot adott arra, hogy valóban kiismerjük az `str.format()` tagfüggvényt – meglátjuk majd, hogy ennek a tudásnak nagy hasznát vesszük a könyv további részében.

Karakterkódolások

A számítógépek végső soron csak bájtok tárolására képesek – ezek 8 bites értékek, amelyek előjel nélkül a `0x00-0xFF` tartományt képesek lefedni. Valamiképpen tehát minden karaktert bájtok formájában kell tárolnunk. A számítástechnika hőskorában a programozás úttörői olyan kódolási módszereket dolgoztak ki, amelyek a karaktereket egyes bájtokhoz rendelik. Így például az ASCII kódolásban az *A* karaktert a `0x41`, a *B* karaktert a `0x42` jeleníti meg, és így tovább. Nyugat-Európában gyakran használták a Latin-1 kódolást, amelyben az első 127 karakter megegyezett a 7 bites ASCII-ben találhatókkal, a fennmaradó részen pedig helyet kaptak az ékezetes karakterek és más, az európaiak számára fontos jelek. Az évek során számos egyéb kódolás is napvilágot látott, amelyek közül sok ma is használatos.

Sajnos ez a rengeteg különböző kódolás meglehetősen kényelmetlen helyzeteket okozhat, különösen, ha nemzetközi felhasználásra szánt programot készítünk. Az egyik, szinte teljes körben elfogadott megoldás a Unicode kódolás használata. A Unicode minden karaktert egy egész számhoz (kódponthoz) rendel – csakúgy, mint a korábbi kódolások, de itt karakterenként nem csak egy bájtot használhatunk, következésképpen egyetlen kódolással leírhatjuk az összes nyelv minden elképzelhető karakterét. A kompatibilitást segítendő az első 127 Unicode karakter megegyezik a 7 bites ASCII kódolás első 127 karakterével.

A kérdés már csak az, miként tárolja a rendszer a Unicode-karaktereket? Nos, jelenleg kicsit több, mint egymillió Unicode-karaktert határoztak meg, így a 32 bites egészek – még ha előjelet használunk is – bőségesen elegendőek a tárolásukhoz. Így hát a Unicode-karakterek tárolásának legegyszerűbb módja, ha 32 bites egészeket sorolunk fel – karakterenként egyet. A memóriát tekintve ez igen kényelmes módszer, hiszen így a karaktersorozatok közvetlenül megfeleltethetők 32 bites egészekből álló tömböknek. Ha azonban fájlokat használunk, vagy hálózati kapcsolaton szeretnénk átküldeni a szöveget, ez a tárolási mód nem kielégítő, főként ha a szöveg többnyire 7 bites ASCII-karakterekből áll – ilyenkor ugyanis a karaktereket leíró négy bájtból három értéke `0x00` lesz. Ezt a tárhelypocsékolást elkerülendő a Unicode számos ábrázolásmóddal rendelkezik.

A memóriában a Unicode-kódok jobbára UCS-2, illetve UCS-4 formátumban kapnak helyet. Az előbbi esetében lényegében előjel nélküli 16 bites egészeket tárolunk, amelyek az első 65 535 kódpontot jelenítik meg, az utóbbiban pedig az összes kódpontot (jelenleg 1 114 111 van belőlük) tárolhatjuk 32 bites egészekkel. Arról, hogy melyiket használjuk, a Python fordításakor hozhatunk döntést. (Amennyiben a `sys.maxunicode` értéke 65535, az USC-2-t választottuk.)

A mezőkben tárolt, illetve hálózati kapcsolaton át továbbított adatok esetében a helyzet messze bonyolultabb. Ha Unicode kódolást alkalmazunk, a kódpontokat az UTF-8 kódolásal tárolhatjuk – itt az első 127 kódpontot egy, a többit pedig két vagy több bájton tároljuk. Ez a kódolás angol szöveg esetén igen tömör eredményt ad, és ha csak a 7 bites ASCII-karaktereket használjuk, az UTF-8 fájlok megkülönböztethetetlenek az ASCII-fájloktól.

Elterjedt ezenkívül az UTF-16 kódolás is, amely a karakterek többségét két bájton, a többit pedig négyen tárolja. Az utóbbi egyes ázsiai nyelvek esetében tömörebb eredményt ad, de az UTF-8-tól eltérően itt egy bájtsorrendjellel kell kezdenünk az adatsort, hogy a kiolvasó program tudja, hogy az adott bájtpár rövid vagy hosszú kódhoz tartozik-e. Mindezek mellett a régebbi kódolásokat – mint a GB2312, az ISO-8859-1 vagy a Latin-1 – továbbra is előszeretettel használják.

Az `str.encode()` tagfüggvény egy bájtsorozatot – valójában egy `bytes` objektumot, amelyről bővebben a 7. fejezetben szólunk – ad vissza a kódolási paraméter szerint kódolva. A használatával némi képet kaphatunk a kódolások közti különbségekről, és világossá válhat, hogy miként eredményezhet hibát, ha nem a megfelelő kódolást használjuk:

```
>>> artist = "Tage Åsén"
>>> artist.encode("Latin1")
b'Tage \xc5s\xe9n'
>>> artist.encode("CP850")
b'Tage \x8fs\x82n'
>>> artist.encode("utf8")
b'Tage \xc3\x85s\xc3\xa9n'
>>> artist.encode("utf16")
b'\xff\xfeT\x00a\x00g\x00e\x00 \x00\xc5\x00s\x00\xe9\x00n\x00'
```

A nyitó idézőjel előtti `b` arra utal, hogy nem karakterláncot, hanem egy literális `bytes` objektumot adunk meg. A kényelmesebb írásmód kedvéért a literális `bytes` értékek felírásánál vegyesen használunk nyomtatható ASCII-karaktereket és hexdecimális kódokat.

Tage Åsén nevét sajnos nem írhatjuk le ASCII kódolással, ugyanis ebben nem szerepel sem az *Å*, sem egyéb ékezetes karakter, ha mégis megpróbálkozunk vele, `UnicodeEncodeError` kivételhez jutunk. A 8 bites Latin-1 (vagy más néven ISO-8859-1) kódolás azonban már minden karakterrel rendelkezik, amelyre szükségünk van. Szegény Bánk Ernő azonban már nem ilyen szerencsés, hiszen az *ő* karakter kimaradt a Latin-1 karakterkészletből, így a nevét nem írhatjuk le ezzel a kódolással. Természetesen a Unicode kódolás alkalmas mindkét név tárolására. Vegyük észre, hogy az UTF-16 esetében az első két bájt a bájtsorrendjel, amely segít a visszafejtő függvénynek annak megállapításában, hogy kis vagy nagy endián kódról van-e szó.

Érdemes pár további megjegyzést is tennünk az `str.encode()` tagfüggvénnyel kapcsolatban. Az első paraméter (a kódolás neve) nem érzékeny a kis- és nagybetűk különbözőségére, továbbá a rendszer az elválasztójeleket és az aláhúzásokat egyenértékűként kezeli, vagyis a „us-ascii” és a „US_ASCII” lényegében azonos. Sok esetben egyazon kódoláshoz több névváltozatot is találunk – így a „latin”, a „latin1”, a „latin_1”, az „ISO-8859-1”, a „CP819” és még sokan mások mind-mind a „Latin-1” megfelelői. A tagfüggvény fogadhat emellett egy második paramétert is, amellyel meghatározhatjuk, hogyan kezelje a hibákat. Így például tetszőleges karakterláncot ASCII formátumúvá alakíthatunk – természetesen

esetleges adatvesztés mellett – az „ignore” vagy a „replace” paraméterrel, vagy veszteségmentesen a „backslashreplace” paraméterrel, amely a nem ASCII-karaktereket `\x`, `\u`, illetve `\U` védőkódokkal helyettesíti. Így az `artist.encode("ascii", "ignore")` a `b'Tage sn'` eredményt adja, az `artist.encode("ascii", "replace")` a `b'Tage ?s?n'`, az `artist.encode("ascii", "backslashreplace")` pedig a `b'Tage \xc5s\xe9n'` `bytes` objektumot. (Szintén ASCII-karakterlánchoz juthatunk a `"{0!a}".format(artist)` utasítással, amely a `'Tage \xc5s\xe9n'` eredményt adja.)

Az `str.encode()` függvény ellentettje a `bytes.decode()` (illetve a `bytearray.decode()`), amely a megadott kódolás alapján visszafejtett karakterláncot adja vissza, valahogy így:

```
>>> print(b"Tage \xc3\x85s\xc3\xa9n".decode("utf8"))
Tage Åsén
>>> print(b"Tage \xc5s\xe9n".decode("latin1"))
Tage Åsén
```

A 8 bites Latin-1, a CP850 (egy IBM PC-kódolás), valamint az UTF-8 közötti különbségek nyilvánvalóvá teszik, hogy a találgatás a kódolások esetében nem célravezető. Szerencsére az UTF-8 a gyakorlatban szabvánnyá vált az egyszerű szövegfájlok körében, így a következő nemzedékek talán nem is tudnak majd róla, hogy más kódolások is léteztek a világon.

A Python `.py` fájljai UTF-8 kódolást alkalmaznak, így a nyelv mindig pontosan tudja, hogy milyen kódolást használjon a literális karakterláncok esetében. Ez azt jelenti, hogy tetszőleges Unicode-karaktereket bemásolhatunk a karakterláncainkba – amennyiben ezt a szerkesztő is támogatja.*

Amikor a Python külső forrásokból, például csatolóktól jut hozzá az adatokhoz, nem tudhatja, hogy milyen kódolással tárolták azokat – ezért egyszerűen bájtonként adja át az adatokat, és nekünk kell gondoskodnunk a visszafejtésről. A szövegfájlok esetében a nyelv némileg finomabb megközelítést alkalmaz: a helyi kódolást használja, hacsak magunk mást nem adunk meg.

Szerencsére egyes fájlformátumok meghatározzák a saját kódolásukat. Így például nyugodtan feltételezhetjük, hogy az XML-fájlok UTF-8 kódolást alkalmaznak, hacsak a `<?xml?>` utasítás kifejezetten másként nem rendelkezik. Ha tehát XML-adatokat olvasunk ki, először érdemes hozzájutnunk mondjuk az első 1000 bájthoz, és utánanézni a kódolás beállításának. Ha megtaláljuk, végezzük el a visszafejtést eszerint, ha nem, térjünk vissza a jó öreg UTF-8-hoz. Ez a megközelítés alkalmazható minden olyan XML- vagy egyszerű szöveges fájlra, amely a Python által támogatott egybájtos kódolások valamelyikét alkalmazza – kivéve az EBCDIC alapú kódolásokat (CP424, CP500), és néhány egyéb esetet (CP037, CP864,

* Más kódolásokat is alkalmazhatunk – lásd a Python Tutorial *Source Code Encoding* témakörét.

CP865, CP1026, CP1140, HZ, SHIFT-JIS-2004, SHIFT-JISX0213). Sajnos ez az eljárás nem használható több bájtos kódolások (mint az UTF-16 vagy az UTF-32) esetén. Ha felkeressük a Python Package Index weboldalát (`pypi.python.org/pypi`), biztosan találunk legalább két olyan Python-csomagot, amely képes a fájlok kódolásának automatikus felderítésére.

éldák

A következőkben arra építünk, amit fejezetünk eddigi részében, valamint az előző fejezetben megtanultunk. Így két apró, de teljes értékű programot készítünk, amelyek vizsgálatával áttekinthetjük az eddig tanultakat. Az első programocska némiképp elvisz a matematika irányába, jóllehet meglehetősen rövid – hossza mindössze 35 sor körüli. A második, amely szöveges adatokat kezel, már mélyebbre hatol tudásanyagunkban – hét függvényt írunk, összesen nagyjából 80 sornyi kóddal.

quadratic.py

A másodfokú egyenletek, amelyek alakja $ax^2 + bx + c = 0$, ahol $a \neq 0$, valójában parabolákat írnak le.

Megoldásaikat a $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ képlettel kaphatjuk meg, ahol a $b^2 - 4ac$ tagot az egyenlet *diszkriminánsának* hívják – ha az értéke pozitív, az egyenletnek két valós megoldása van, ha nulla, egyetlen megoldás létezik, amely valós, ha pedig negatív, két komplex megoldást kapunk. A következőkben egy olyan programot írunk, amely a felhasználótól várja az *a*, *b* és c paramétereket (ahol *b* és *c* lehet 0 is), majd kiszámítja és kiírja a megoldást, illetve megoldásokat.*

Először kipróbáljuk a programot, ezt követően pedig áttekintjük a kódot.

```
quadratic.py
ax² + bx + c = 0
enter a: 2.5
enter b: 0
enter c: -7.25
2.5x² + 0.0x + -7.25 = 0 □  x = 1.70293863659 or x = -1.70293863659
```

A 1.5, –3 és 6 paraméterekkel a kimenet a következő lesz (néhány számjegyet levágtunk):

```
1.5x² + -3.0x + 6.0 = 0 □  x = (1+1.7320508j) or x = (1-1.7320508j)
```

Nos, a kimenet nem olyan csinos, mint amilyennek szerettük volna – így például a `+ -3.0x` helyett jobban tetszene a `-3.0x`, és a 0 értékű paraméterek tagjait fel sem szeretnénk tüntetni. Ne aggódjunk, a feladatok sorában módot kapunk rá, hogy csiszolgassuk a kódot!

---

* Mivel a Windows konzolja meglehetősen gyenge UTF-8 támogatást ad, a Mac OS X konzolja pedig alapértelmezés szerint az Apple Roman kódolást használja, gondok adódnak a `quadratic.py` programban használt egyes karakterekkel (mint a ² és a □). Ennek tudatában elérhetővé tettük a program `quadratic_uni.py` nevű változatát, amely a megfelelő jeleket jeleníti meg Linuxon, más konzolokon pedig helyettesítő jeleket (`^2` és `->`) használ helyettük.

Most azonban térjünk rá a kód vizsgálatára, amely három betöltéssel veszi kezdetét:

```
import cmath
import math
import sys
```

Egyaránt szükségünk van a `float` és a `complex` matematikai könyvtárakra, hiszen a valós és a komplex számok négyzetgyökfüggvénye eltérő, emellett a `sys.float_info.epsilon` miatt, amellyel összehasonlíthatjuk a lebegőpontos számokat a 0-val, szükségünk van a `sys` modulra is.

Írnunk kell továbbá egy függényt, amely képes arra, hogy a felhasználótól egy lebegőpontos számot fogadjon:

```
def get_float(msg, allow_zero):
    x = None
    while x is None:
        try:
            x = float(input(msg))
            if not allow_zero and abs(x) < sys.float_info.epsilon:
                print("zero is not allowed")
                x = None
        except ValueError as err:
            print(err)
    return x
```

A függvény mindaddig bent marad a ciklusban, amíg a felhasználó egy érvényes lebegőpontos számot nem ad meg (például 0.5, –9, 21, 4.92) – 0-t csak akkor fogad el, ha az `allow_zero` értéke `True`.

A program, miután meghatározta a `get_float()` függvényt, végrehajtja a fennmaradó kódot. Ezt három részben mutatjuk be – kezdjük a felhasználói bemenet fogadásával:

```
print("ax\N{SUPERSCRIPT TWO} + bx + c = 0")
a = get_float("enter a: ", False)
b = get_float("enter b: ", True)
c = get_float("enter c: ", True)
```

Hála a `get_float()` függvénynek, az *a*, *b* és *c* paraméterek fogadása pofonegyszerű. A második, logikai paraméterrel határozhatjuk meg, hogy elfogadjuk-e a 0-t.

```
x1 = None
x2 = None
discriminant = (b ** 2) - (4 * a * c)
if discriminant == 0:
    x1 = -(b / (2 * a))
else:
    if discriminant > 0:
```

```
    root = math.sqrt(discriminant)
else: # discriminant < 0
    root = cmath.sqrt(discriminant)
x1 = (-b + root) / (2 * a)
x2 = (-b - root) / (2 * a)
```

Kódunk némiképp eltér a fentebb bemutatott képlettől, mivel a diszkrimináns kiszámításával kezdjük a munkát. Amennyiben a diszkrimináns értéke 0, tudjuk, hogy egyetlen megoldás létezik, és ezt közvetlenül ki is számítjuk. Egyébként hozzájutunk a diszkrimináns valós vagy komplex négyzetgyökéhez, és meghatározzuk a két megoldást.

```
equation = ("{0}x\N{SUPERSCRIPT TWO} + {1}x + {2} = 0"
            " \N{RIGHTWARDS ARROW} x = {3}").format(a, b, c, x1)
if x2 is not None:
    equation += " or x = {0}".format(x2)
print(equation)
```

Nem volt szükségünk semmilyen különleges formázásra, mivel a példában a Python lebegőpontos számok megjelenítésénél alkalmazott alapértelmezései tökéletesen megfeleltek. Mindazonáltal a különleges karakterek ábrázolásához Unicode-karakterneveket használtunk.

csv2html.py

Gyakran van szükség arra, hogy egy adathalmazt HTML-ben jelenítsünk meg, így az alábbiakban egy korántsem szokatlan feladattal foglalkozunk – készítünk egy programot, amely beolvassa egy CSV (comma separated values – vesszővel elválasztott értékek) formátumú fájl tartalmát, és a benne található adatokat egy HTML-táblázatban jeleníti meg. A Python rendelkezik egy `csv` nevezetű modullal, amely hatékony és kifinomult módszereket biztosít a CSV és hasonló formátumok kezelésére, de most eltekintünk a használatától, és magunk írjuk meg a szükséges kódot.

Az általunk elfogadott CSV formátumban soronként egy bejegyzés áll, és a bejegyzésekeket vesszők osztják mezőkre. A mezőkről annyit tudunk, hogy karakterláncok vagy számok szerepelhetnek bennük – az előbbiek aposztrófokon, illetve idézőjeleken belül állnak, az utóbbiak pedig idézőjelek nélkül szerepelnek, hacsak nem tartalmaznak vesszőt. A karakterláncokon belül állhatnak vesszők, így fontos, hogy ezeket ne tekintsük mezőelválasztónak. Feltételezzük, hogy az első bejegyzés tartalmazza a mezők címkéit. Kimenet gyanánt egy HTML-táblázatot hozunk létre balra igazított szöveggel (ez az alapértelmezés a HTML-ben) és jobbra igazított számokkal, bejegyzésenként egy sorral és mezőnként egy cellával.

A programnak tehát először ki kell írnia a HTML-táblázat nyitócímkéjét, beolvasnia az adatok sorait, majd HTML-táblázatsorokban kiírnia azokat, végül pedig, ha nincs több sor, kiírnia a HTML-táblázat zárócímkéjét. Szeretnénk továbbá, ha a táblázat első sorának (amelyben a mezők címkéi állnak majd) háttere világoszöld lenne, az adatsoroké pedig felváltva fehér és világossárga. Fontos az is, hogy megfelelően levédjük a HTML különleges karaktereit (&, < és >), valamint a karakterláncok külalakjára is próbáljunk ügyelni.

Először is lássunk néhány adatot a kísérletezéshez:

```
"COUNTRY","2000","2001",2002,2003,2004
"ANTIGUA AND BARBUDA",0,0,0,0,0
"ARGENTINA",37,35,33,36,39
"BAHAMAS, THE",1,1,1,1,1
"BAHRAIN",5,6,6,6,6
```

Feltételezve, hogy az adatok a `data/co2-sample.csv` fájlban találhatók, és a `csv2html.py < data/co2-sample.csv > co2-sample.html` parancsot adtuk ki, a `co2-sample.html` fájl tartalma valahogy így alakul:

```
<table border='1'><tr bgcolor='lightgreen'>
<td>Country</td><td align='right'>2000</td><td align='right'>2001</td>
<td align='right'>2002</td><td align='right'>2003</td>
<td align='right'>2004</td></tr>
...
<tr bgcolor='lightyellow'><td>Argentina</td>
<td align='right'>37</td><td align='right'>35</td>
<td align='right'>33</td><td align='right'>36</td>
<td align='right'>39</td></tr>
...
</table>
```

Hozzá kell tennünk, hogy kissé kicsinosítottuk a kimenetet, és néhány sor helyett három pontot írtunk. A HTML igen egyszerű változatát, a HTML 4 transitionalt alkalmaztuk, stíluslapok nélkül. A 2.7. ábrán láthatjuk, hogyan fest az eredmény egy böngészőben.

Country	2000	2001	2002	2003	2004
Antigua and Barbuda	0	0	0	0	0
Argentina	37	35	33	36	39
Bahamas, The	1	1	1	1	1
Bahrain	5	6	6	6	6

2.7. ábra

A csv2html.py programmal kapott táblázat egy böngészőben

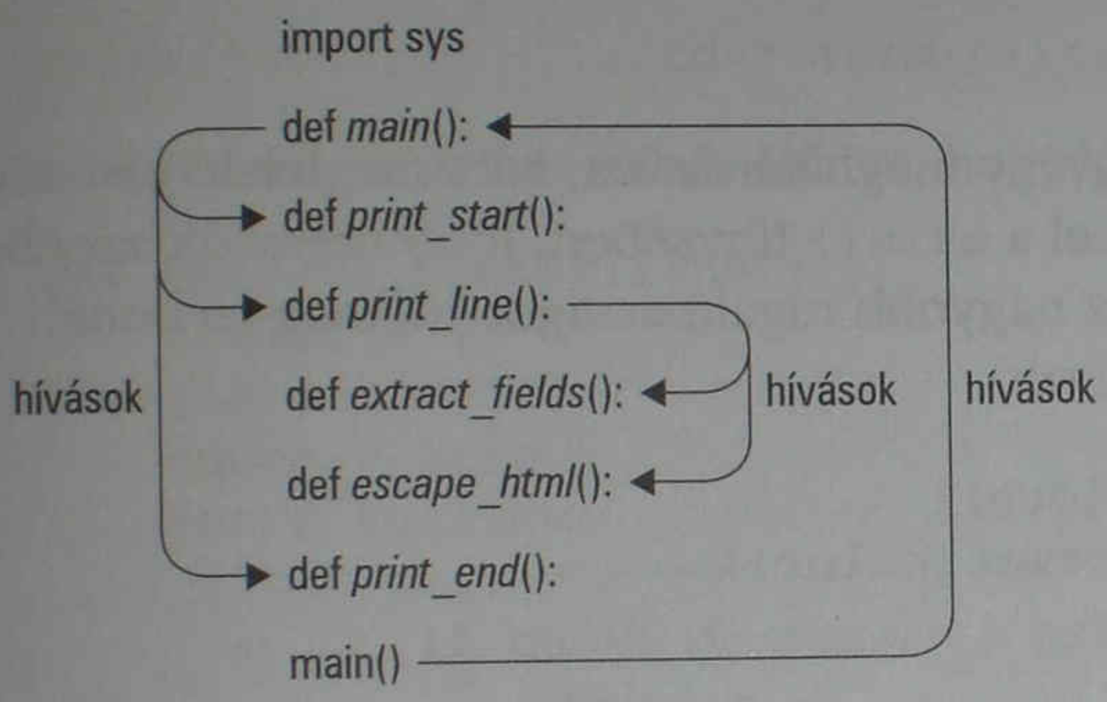
Láttuk hát, hogyan használhatjuk a programot, és milyen eredményt kaphatunk, így nekiláthatunk a kód áttekintésének. A program a `sys` modul betöltésével veszi kezdetét – ezt és az egyéb betöltéseket a következőkben nem mutatjuk be, hacsak nem valamilyen szokatlan vagy figyelemreméltó esetről van szó. a program utolsó utasítása egy egyszerű függvényhívás:

```
main()
```

Jóllehet a Pythonban nincs szükség a számos más nyelv által megkövetelt belépési pontra, mégis gyakori a Python-programokban a `main()` függvény használata, amelynek a meghívásával megkezdődik a kód végrehajtása. Mivel egyetlen függvényt sem hívhatunk meg,

mielőtt meghatároztuk volna, fontos, hogy a `main()`-t csak akkor hívjuk meg, ha a működéséhez szükséges minden függvényt meghatároztunk. Ugyanakkor a függvények megjelenési sorrendje a fájlban (vagyis a létrehozásuk sorrendje) egyáltalán nem számít.

A `csv2html.py()` programban először a `main()` függvényt hívjuk meg, amely ugyanakkor meghívja a `print_start()` és a `print_line()` függvényeket. Utóbbi viszont az `extract_fields()` és az `escape_html()` függvényeket veszi használatba. A program szerkezetét a 2.8. ábra mutatja.



2.8. ábra
A csv2html.py program szerkezete

A fájl beolvasását a Python felülről kezdi, tehát először elvégzi a betöltést, ezt követően létrehozza a `main()` függvényt, majd a többit is, abban a sorrendben, ahogy a fájlban megjelennek. Amikor végre eléri a fájl végén található `main()` hívást, minden olyan függvény, amelyet a `main()` meghív (sőt azok is, amelyeket ezek a függvények meghívnak) létezik már. A program futtatása, legalábbis a mi fogalmaink szerint, a `main()` hívásával kezdődik.

Vegyük most sorra a függvényeket – elsőként álljon itt a `main()`.

```
def main():
    maxwidth = 100
    print_start()
    count = 0
    while True:
        try:
            line = input()
            if count == 0:
                color = "lightgreen"
            elif count % 2:
                color = "white"
            else:
                color = "lightyellow"
            print_line(line, color, maxwidth)
            count += 1
        except EOFError:
            break
    print_end()
```

A `maxwidth` változóval korlátozzuk az egy cellán belül álló karakterek számát – ha a mező tartalma nagyobb lenne ennél, csonkoljuk, és három pontot teszünk a kapott szöveg végére. A `print_start()`, a `print_line()` és a `print_end()` függvényekre pillanatokon belül visszatérünk. A `while` ciklus sorra veszi a bemenet sorait – ezek származhatnak felhasználói bemenetről is, de most úgy vesszük, hogy egy fájlba kerültek. Beállítjuk a kívánt színt, és meghívjuk a `print_line()` függvényt a HTML-táblázat adott sorának kiírásához.

```
def print_start():
    print("<table border='1'>")

def print_end():
    print("</table>")
```

Könnyedén elkerülhettük volna e két függvény meghatározását, ha a megfelelő `print()` függvényhívásokat közvetlenül helyezzük el a `main()` törzsében. Jobb szeretjük azonban a kódot logikai részekre bontani, hiszen ez nagyobb rugalmasságot ad, még ha ennél a rövid példánál ez nem is annyira nyilvánvaló.

```
def print_line(line, color, maxwidth):
    print("<tr bgcolor='{0}'>".format(color))
    fields = extract_fields(line)
    for field in fields:
        if not field:
            print("<td></td>")
        else:
            number = field.replace(",", "")
            try:
                x = float(number)
                print("<td align='right'>{0:d}</td>".format(round(x)))
            except ValueError:
                field = field.title()
                field = field.replace(" And ", " and ")
                field = escape_html(field)
            if len(field) <= maxwidth:
                print("<td>{0}</td>".format(field))
            else:
                print("<td>{0:.{1}} ...</td>".format(field, maxwidth))
    print("</tr>")
```

A sorok mezőkre bontásánál nem használhatjuk egyszerűen az `str.split(",")` utasítást, hiszen vesszők az idézőjelben álló karakterláncokon belül is előfordulhatnak. Ezért az itt elvégzendő műveleteket kihelyeztük az `extract_fields()` függvénybe. Ha rendelkezünk a mezők listájával (karakterláncok alakjában, határoló idézőjelek nélkül), sorra vehetjük őket, táblázatcellát létrehozva mindegyikükhöz.

Ha egy mező üres, üres cellát írunk ki. Amennyiben egy mező tartalma idézőjelek között áll, lehet karakterlánc, de lehet szám is, amelyet azért tettünk idézőjelbe, hogy feltüntethessünk benne egy vesszőt, mint az `"1,566"` esetében. Ez utóbbi eset kezelésére másolatot

készítünk a mezőről, amelyből eltávolítjuk az idézőjeleket, és megkíséreljük `float` típusúvá alakítani. Amennyiben az átalakítás sikeres, a szám értékét a legközelebbi egészre kerekítjük, és jobbra igazítva kiírjuk egy cellában. Ha nem sikerül, a mezőt karakterláncként jelenítjük meg. Ez esetben az `str.title()` tagfüggvény használatával nagy kezdőbetűket alkalmazunk, és még arra is ügyelünk, hogy a függvény hatásának kiigazítására az *And* helyett az angol címek helyesírási szabályai szerint kisbetűs *and* szót írjunk. A HTML különleges karaktereit levédjük, és vagy kiírjuk a teljes mezőt, vagy a csonkolt alakot adjuk vissza három ponttal kiegészítve. A belső cseremező használata helyett egyszerűbben is eljárhatunk, ha szeleteljük a karakterláncot:

```
print("<td>{0} ...</td>".format(field[:maxwidth]))
```

Ez utóbbi megközelítés másik előnye, hogy kissé kevesebbet kell bajlódnunk a gépeléssel.

```
def extract_fields(line):
    fields = []
    field = ""
    quote = None
    for c in line:
        if c in "\"'":
            if quote is None: # az idézőjelben álló karakterlánc eleje
                quote = c
            elif quote == c:  # az idézőjelben álló karakterlánc vége
                quote = None
            else:
                field += c    # további idézőjel az idézőjelben álló
                              # karakterláncban
            continue
        if quote is None and c == ",":  # mező vége
            fields.append(field)
            field = ""
        else:
            field += c          # mező összeállítása
    if field:
        fields.append(field)  # az utolsó mező hozzáadása
    return fields
```

Ez a függvény karakterenként kiolvassa a neki átadott sort, felépítve a mezők listáját, amelynek az elemei már nem tartalmazzák a határoló idézőjeleket. A függvény elboldogul mind az idézőjelbe vagy aposztrófok közé tett, mind az idézőjel nélküli mezőkkel, továbbá megfelelően kezeli a vesszőket és az idézőjeleket (aposztrófokat az idézőjeles karakterláncokon belül és idézőjeleket az aposztrófok közé helyezett karakterláncokban).

```
def escape_html(text):
    text = text.replace("&", "&amp;")
    text = text.replace("<", "&lt;")
    text = text.replace(">", "&gt;")
    return text
```

Utolsóként bemutatott függvényünk egyszerűen helyettesíti a HTML egyes különleges karaktereit a megfelelő HTML-elemmel. Természetesen először az & jeleket kell helyettesítenünk, a < és a > sorrendje azonban nem számít. A Python alapkönyvtárában megtalálhatjuk ennek a függvénynek egy némiképp kifinomultabb változatát – ezt használhatjuk a feladatok megoldásánál, a 7. fejezetben pedig újra találkozhatunk vele.

Összefoglalás

A fejezet elején bemutattuk a Python kulcsszavainak listáját, és leírtuk, hogy milyen szabályok érvényesek a nyelvben az azonosítókra. A Python Unicode-támogatásának köszönhetően az azonosítókat nem kell egy kisebb karakterkészletre – mint az ASCII vagy a Latin-1 – korlátoznunk.

Megismerkedtünk az `int` adattípussal, amely lényegesen eltér más nyelvek hasonló típusaitól, ugyanis nem rendelkezik méretkorlátozással – nagyságát kizárólag az elérhető memória mennyisége korlátozza, így a típus tökéletesen alkalmas arra, hogy több száz jegyű számokkal dolgozzunk. A Python alapvető adattípusai egyaránt állandó értékűek, de ez alig észrevehető, hiszen az értékadó műveletekkel (mint a `+=`, a `*=`, a `-=`, a `/=` és mások) a lehető legtermészetesebb nyelvtani alakokat használhatjuk, a Python pedig a háttérben létrehozza a megfelelő objektumokat, és hozzájuk köti a változókat. A literális egészeket többnyire tízes számrendszerbeli alakjukban írjuk fel, de használhatunk bináris alakot is a `0b`, oktális alakot a `0o` és hexadecimális alakot a `0x` előtaggal.

Ha egy egész számot elosztunk egy másikkal a `/` művelettel, az eredmény mindig `float` típusú lesz. Ez a viselkedés eltér más, széles körben elterjedt programozási nyelvekétől, de segít néhány alattomos hiba elkerülésében, amelyek akkor következnek be, amikor egy osztásnál a rendszer szó nélkül csonkolja az eredményt. (Ha pedig valóban egészosztásra van szükségünk, a `//` műveletet használhatjuk.)

A Pythonban rendelkezésre áll a `bool` adattípus, amely `True` vagy `False` értéket tartalmazhat. A nyelv három logikai művelettel – `and`, `or` és `not` – rendelkezik, amelyek közül a binárisak (`and` és `or`) rövidzáras logikát alkalmaznak.

Háromféle lebegőpontos számot használhatunk: a `float`, a `complex`, illetve a `decimal.Decimal` típust. Leggyakrabban a `float` típussal találkozunk – ez egy kétszeres pontosságú lebegőpontos adattípus, amelynek a pontos jellemzői attól a C-, C#- vagy Java-könyvtártól függnek, amellyel a Pythont felépítették. A komplex számokat két `float` értékkel ábrázoljuk: az egyik a valós, a másik pedig a képzetes részt adja meg.
A `decimal.Decimal` típust a `decimal` modul révén érhetjük el. Az ilyen típusú számok alapértelmezés szerint 28 tizedesjegyig pontosak, de ezt a pontosságot igényeink szerint növelhetjük vagy csökkenthetjük.

Mindhárom lebegőpontos típust használhatjuk a megfelelő beépített matematikai műveletekkel és függvényekkel. Mindemellett a `math` modul számos trigonometrikus, hiperbolikus és logaritmikus függvényt biztosít a `float` értékek számára, a `cmath` pedig hasonló függvényeket tesz elérhetővé a komplex számokhoz.

Fejezetünk nagy részét a karakterláncoknak szenteltük. A Python literális karakterláncait létrehozhatjuk aposztrófok vagy idézőjelek közé zárva, de használhatunk hármas idézőjeleket is – ilyenkor mindenféle elkerülő manőver nélkül használhatunk új sorokat és idézőjeleket a karakterláncban. Számos védőkód ismeretes, amelyekkel különleges karaktereket – mint a tabulátor (`\t`) vagy az újsor (`\n`) – vehetünk használatba, a Unicode-karakterek beillesztésére pedig a hexadecimális védőkódjuk, illetve a nevük megadásával is lehetőségünk van. Jóllehet a karakterláncok támogatják a más Python-adattípusoknál megismert összehasonlító műveleteket, meg kellett jegyeznünk, hogy a rendezésük bizony akadályokba ütközhet, amennyiben nem csak az angol ábécé karaktereit tartalmazzák.

Mivel a karakterláncok sorozatok, a szeletelő művelet (`[]`) segítségével igen egyszerű alakban szeletelhetjük, illetve egyelhetjük őket. A karakterláncok összefűzhetők a `+` és többszörözhetők a `*` művelettel, sőt ezek értékadó változatait (`+=` és `*=`) is használhatjuk, jóllehet az `str.join()` gyakrabban használatos az összefűzéshez. A karakterláncok kezelésére számos tagfüggvény áll rendelkezésünkre – egyesekkel a tulajdonságaikat vizsgálhatjuk (mint az `str.isspace()` vagy az `str.isalpha()`), másokkal a kis- és nagybetűk között válthatunk (mint az `str.lower()` vagy az `str.title()`), megint másokkal keresést végezhetünk (mint az `str.find()` vagy az `str.index()`), és még sok más lehetőség áll a rendelkezésünkre.

A Python karakterlánc-támogatása valóban nagyszerű – lehetővé teszi, hogy könnyedén megkeressünk, kiolvassunk és összehasonlítsunk karakterláncokat vagy azok részeit; karaktereket vagy karakterláncok részeit lecseréljük; karakterláncokat részekre bontsunk; illetve a részek listájából egyetlen karakterláncot állítsunk elő.

Mind közül talán a legrugalmasabb az `str.format()` tagfüggvény. Használatával egy cseremezőket tartalmazó karakterláncból kiindulva, és ezeket a mezőket megfelelően formázott változókkal helyettesítve végül a kívánságainknak megfelelő alakú eredményt kaphatunk. A cseremezők lehetővé teszik, hogy a tagfüggvény paramétereit a sorszámuk (indexük), illetve (kulcsszó-paraméterek esetén) a nevük szerint elérjük, és a megfelelő sorszámmal, kulccsal vagy tulajdonságnévvel hozzájussunk a megfelelő paraméterelemhez vagy tulajdonsághoz. A formátumleírásban megadhatjuk a mező kitöltő karakterét, igazítását, valamint a legkisebb szélességét. Számok esetén arra is befolyást gyakorolhatunk, hogy miként jelenjen meg a kimenet előjele, lebegőpontos értékeknél pedig megadhatjuk a tizedesjegyek számát, valamint azt, hogy egyszerű vagy exponenciális jelölésmódot kívánunk-e alkalmazni.

Szót ejtettünk továbbá a karakterkódolások kényes témaköréről is. A Python `.py` fájljaiban alapértelmezés szerint a Unicode UTF-8 kódolás használatos, így a megjegyzéseket, változókat és egyéb adatokat szinte tetszőleges nyelven megadhatjuk. Lehetőségünk van arra, hogy az `str.encode()` használatával egy karakterláncot adott kódolás mentén bájtsorozattá alakítsunk, ha pedig egy bájtsorozatból szeretnénk a kódolás ismeretében a megfelelő karakterlánchoz hozzájutni, a `bytes.decode()` tagfüggvény segíthet. A jelenleg használatban levő milliónyi karakterkódolás megnehezíti az egyszeri programozó életét, de ahogy az UTF-8 egyre szélesebb körben elfogadottá válik az egyszerű szövegfájlok esetében (az XML formátumnál már eleve ez az alapértelmezés), a gondok az évek során bizonyára mérséklődnek.

A fejezetünkben bemutatott adattípusokon kívül a Python még további kettőt – nevezetesen a `bytes` és a `bytearray` típust – tartogat a számunkra, amelyekről a 7. fejezetben szólunk bővebben. Létezik emellett számos gyűjteményes adattípus is, egyesek beépítettek, másokhoz az alapkönyvtárban juthatunk hozzá. A következő fejezetben bemutatjuk közülük a legfontosabbakat.

Gyakorlatok

1. Módosítsuk a `print_unicode.py` programot úgy, hogy a felhasználó több szót is megadhasson a parancssorban, és csak azokat a sorokat írjuk ki, amelyekben a Unicode-karakternév ezek mindegyikét tartalmazza. Mindez azt jelenti, hogy az alábbihoz hasonló parancsokat szeretnénk használni:

   ```
   print_unicode_ans.py greek symbol
   ```

   Az egyik lehetséges megoldás, ha a `word` változót (ami eddig a `0` vagy a `None` értéket, illetve egy karakterláncot tartalmazhatott) egy `words` nevű listával helyettesítjük. Ne felejtsük el a kód mellett frissíteni a használati utasítást sem. A módosítás kevesebb mint 10 új sor beírásával és másik 10 módosításával jár. Egy lehetséges megoldást megtekinthetünk a `print_unicode_ans.py` fájlban. (A Windowst, illetve több rendszert használók számára a feladat a `print_unicode_uni.py` módosítása, a megoldást pedig a `print_unicode_uni_ans.py` fájlban találhatják meg.)
2. Módosítsuk a `quadratic.py` programot úgy, hogy ne írja ki a 0.0 paraméterű tagokat, a negatívakat pedig `+ -`*n* helyett egyszerűen `-` *n* alakban jelenítse meg. Mindehhez az utolsó öt sort nagyjából tizenöttel kell helyettesítenünk. Egy lehetséges megoldást megtekinthetünk a `quadratic_ans.py` fájlban. (A Windowst, illetve több rendszert használók számára a feladat a `quadratic_uni.py` módosítása, a megoldást pedig a `quadratic_uni_ans.py` fájlban találhatják meg.)
3. Töröljük az `escape_html()` függvényt a `csv2html.py` programból, és használjuk helyette az `xml.sax.saxutils` modul `xml.sax.saxutils.escape()` függvényét. Mindez igen egyszerű, hiszen csak egyetlen új sorra (a betöltésre), öt sor (a nem kívánt függény) törlésére, valamint egy sor módosítására (az `excape_html()` helyett használatba vesszük az `xml.sax.saxutils.escape()` függvényt) van szükség. A megoldást a `csv2html1_ans.py` fájlban találhatjuk meg.

4. Módosítsuk ismét a `csv2html.py` programot, ezúttal egy új, `process_options()` nevű függvénnyel kiegészítve, amelyet a `main()` függvény törzséből hívunk majd meg. Az új függvénynek két értéket kell visszaadnia egy rendezett pár alakjában: egy `maxwidth` nevezetű `int`, valamint egy `format` nevezetű `str` értéket.
A `process_options()`-nak a meghívásakor a `maxwidth` számára az alapértelmezett 100, a `format` számára pedig a „.0f" értéket kell megadnia – számok kiírásánál ezt alkalmazzuk a formátumleírásban.
Ha a felhasználó a `-h` vagy a `--help` kapcsolót adta meg a parancssorban, írjunk a képernyőre egy használati utasítást, a függvény visszatérési értéke pedig legyen `(None, None)`. (Ez esetben a `main()` ne tegyen semmit.) Egyébként a függvény olvasson be minden parancssori paramétert, és végezze el a megfelelő hozzárendeléseket. Mindez azt jelenti, hogy be kell állítanunk a `maxwidth` értékét, ha a `maxwidth=n` paraméterrel találkozunk, a `format=s` esetében pedig a `format` értékét kell a felhasználó kívánsága szerint meghatároznunk. Lássuk most, miként festhet egy használati utasítás:

```
csv2html2_ans.py -h
usage:
csv2html.py [maxwidth=int] [format=str] < infile.csv > outfile.html

maxwidth is an optional integer; if specified, it sets the maximum
number of characters that can be output for string fields,
otherwise a default of 100 characters is used.

format is the format to use for numbers; if not specified it
defaults to ".0f".
```

Ez pedig a parancssor, mindkét kapcsoló beállításával:

```
csv2html2_ans.py maxwidth=20 format=0.2f < mydata.csv > mydata.html
```

Ne felejtsük el módosítani a `print_line()` függvényt, hogy használatba vegyük a `format` értékét a számok megjelenítésénél – ehhez szükségünk lesz egy újabb paraméter átadására, egy új sor beillesztésére és egy másik módosítására. Mindez némiképp hatással lesz a `main()` függvényre is. A `process_options()` mindent összevetve nagyjából 25 sorból áll majd (ebbe beleszámítottuk a használati utasítás kilenc sorát is). Fontos megjegyeznünk, hogy ez a gyakorlat komoly kihívást jelenthet a kevésbé gyakorlott programozóknak.
A program kipróbálásához két adatfájl – a `data/co2-sample.csv` és a `data/co2-fromfossilfuels.csv` – áll a rendelkezésünkre, egy lehetséges megoldást pedig a `csv2html2_ans.py` fájlban találhatunk. Az 5. fejezetben megismerkedünk majd az `optparse` modullal, amellyel egyszerűsíthetjük a parancssori feldolgozást.

3

Gyűjteményes adattípusok

- Sorozattípusok
- Halmaztípusok
- Megfeleltetési típusok
- Gyűjtemények bejárása és másolása

Az előző fejezetben megismertük a Python legfontosabb alapvető adattípusait. Ebben a fejezetben programírói eszköztárunkat az adatelemek együttes tárolásának irányában bővítjük, és ehhez a Python nyelv gyűjteményes adattípusait fogjuk használni. Szó lesz a rendezett sorozatokról és a listákról, valamint bemutatunk új gyűjteményes adattípusokat is, többek közt a halmazokat és a szótárakat. Mindegyiket kimerítően fogjuk tárgyalni.*

A gyűjteményeken felül azt is látni fogjuk, hogy miként hozhatók létre olyan adatelemek, amelyek több más adatelemet tartalmaznak (a C és a C++ nyelv struktúráihoz, illetve a Pascal nyelv rekordjaihoz hasonlók); az ilyen elemeket kezelhetjük egységként, ha éppen úgy kényelmes, de a bennük tárolt elemek külön-külön is elérhetők maradnak. Természetesen az egyéb adatelemeknél megszokott módon az ilyen, több adatelemet tartalmazó elemek is gyűjteménybe szervezhetők.

Ha gyűjteményben tároljuk az adatainkat, akkor lényegesen könnyebb az olyan műveletek elvégzése, amelyeket minden adatelemen végre kell hajtani, és ezzel a módszerrel a fájlokban tárolt adatok beolvasása is egyszerűbb. Ismertetni fogjuk a szövegfájlok kezelésének alapvető fogásait is – olyan mértékben, ahogy a fejezetben szükség van rá –, de a részletek nagyja (a hibakezelés is) a 7. fejezetre marad.

Az egyes gyűjteményes adattípusok tárgyalásának végeztével a gyűjtemények bejárására összpontosítunk – a Python nyelv valamennyi gyűjteményes adattípusánál azonos utasításformát használunk. Meg fogunk ismerkedni a gyűjtemények másolásának módszereivel, és a másolás kapcsán felmerülő problémákkal is.

---

* A sorozattípusok, a halmaztípusok, illetve a megfeleltetési típusok e fejezetben használt meghatározásai gyakorlati szempontból megfelelők, de nem formálisak. Formális meghatározások a 8. fejezetben fordulnak elő nagyobb számban.

Sorozattípusok

A *sorozat* adattípus támogatja a tagsági műveletet (`in`), vizsgálható a mérete (`len()`), képezhető részsorozata (`[]`), valamint bejárható. A sorozatok tárolására a Python nyelv öt beépített típust – `bytearray`, `bytes`, `list`, `str` és `tuple` – bocsát a rendelkezésünkre. Az első kettőt eltérő helyen, a 7. fejezetben tárgyaljuk. Az alapkönyvtár (standard library) további sorozattípusokat is kínál, amelyek közül a `collections.namedtuple` a legjelentősebb. Az előbb említett típusok mindegyikére igaz, hogy az elemek bejárása adott sorrendben történik.

Karakterláncok
60 ☞

A karakterláncokról az előző fejezetben volt szó. Ebben a részben a rendezett sorozatok (tuple), a nevesített rendezett sorozatok, és a listák tárgyalására kerítünk sort.

Rendezett sorozatok

Karakterláncok szeletelése és egyelése
64 ☞

Rendezett sorozat (tuple) alatt nulla vagy annál több objektumhivatkozás rendezett sorát értjük. A rendezett sorozatok bejárása és szeletelése (részsorozat képzése) a karakterláncokkal azonos utasításformát követ, miáltal egyszerűvé válik a rendezett sorozatok elemeinek kinyerése. A karakterláncokhoz hasonlóan a rendezett sorozatok is állandó értékűek (immutable), vagyis az elemek törlése, illetőleg cseréje nem lehetséges. Ha rendezett sor módosítására van szükségünk, egyszerűen listát használunk rendezett sorozat helyett, amennyiben pedig egy már létező rendezett sorozatot szeretnénk módosítani, akkor először a `list()` átalakítófüggvénnyel listává alakítjuk a rendezett sorozatot, és a módosításokat az így létrejövő listán végezzük el.

Sekély és mély másolás
☞ 143

Lehetőség van a `tuple` adattípus függvényként való meghívására is: `tuple()`. A paraméterek nélküli hívás visszatérési értéke egy üres rendezett sorozat, amennyiben pedig az átadott paraméter egy már létező, tuple típusú objektum objektumhivatkozása, a függvény a paraméter sekély másolatát adja vissza. A függvény legfeljebb egy paramétert fogad. Rendezett sorozat létrehozására a `tuple()` függvény meghívása nélkül is van lehetőség. Üres zárójel, vagyis `()` használatakor üres rendezett sorozat jön létre, vesszők használatával pedig a felsorolt elemekből képzett rendezett sorozat. Vannak olyan esetek, amikor a rendezett sorozatokat az egyértelműség kedvéért zárójelbe kell tennünk. Például, ha az `1, 2, 3` elemekből álló rendezett sorozatot átadjuk egy függvénynek, a *`függvény`*`((1,2,3))` formát használjuk.

A 3.1. ábra a `t = "venus", -28, "green", "21", 19.74` rendezett sorozatot, valamint a rendezett sorozat elemeinek sorszámait (index) mutatja be. A karakterláncok elemeinek sorszámozása ugyanígy történik, de amíg a karakterláncok egyes elemei karakterek, a rendezett sorozatokban objektumhivatkozásokat találunk.

t[-5]	t[-4]	t[-3]	t[-2]	t[-1]
'venus'	-28	'green'	'21'	19.74
t[0]	t[1]	t[2]	t[3]	t[4]

3.1. ábra

Rendezett sorozat elemeinek sorszámai

A `tuple` adattípus mindössze két tagfüggvényt bocsát a rendelkezésünkre. Az egyik a `t.count(x)`, amely az `x` objektum `t` rendezett sorozaton belüli előfordulásainak számát adja meg, a másik pedig a `t.index(x)` amely az `x` objektum `t` rendezett sorozaton belüli bal szélső előfordulási helyének sorszámát adja vissza – és `ValueError` kivételt vált ki, ha a rendezett sorozatban nem található meg az `x` objektum. (Ezek a tagfüggvények a listák esetében is a rendelkezésünkre állnak.)

Ezen felül a rendezett sorozatokon értelmezzük a `+` (összefűzés), a `*` (szorzás) és a `[]` (szeletelés) műveleteket, illetve az `in` és a `not in` műveleteket, amelyek a tagsági viszony vizsgálatára valók. Használhatók az értéknövelést és értékadást egy műveletként megvalósító (bővített) `+=` és `*=` műveletek is, annak ellenére, hogy a `tuple` típusú objektumok állandó értékűek. A színfalak mögött ugyanis a Python új rendezett sorozatot hoz létre az eredmény tárolására, majd a műveleti jel bal oldalán álló objektumhivatkozást úgy módosítja, hogy az új rendezett sorozatra hivatkozzon. A karakterláncok is ugyanígy működnek az említett két művelet esetében. A szokásos összehasonlító műveletekkel (`<`, `<=`, `==`, `!=`, `>=`, `>`) lehetőség nyílik a rendezett sorozatok összehasonlítására is, mégpedig elemenkénti összehasonlítással (egymásba ágyazott elemek, például rendezett sorozaton belüli rendezett sorozatok esetében önhívó összehasonlítással).

Tekintsünk át néhány szeletelési példát, kezdve egyetlen elem kinyerésével, majd egy részsorozat képzésével folytatva:

```
>>> hair = "black", "brown", "blonde", "red"
>>> hair[2]
'blonde'
>>> hair[-3:] # same as: hair[1:]
('brown', 'blonde', 'red')
```

Ezek a műveletek karakterláncok, listák és bármilyen egyéb sorozat típusú szerkezet esetében azonosan működnek.

```
>>> hair[:2], "gray", hair[2:]
(('black', 'brown'), 'gray', ('blonde', 'red'))
```

A fenti példában egy ötelemű rendezett sorozat létrehozásával próbálkoztunk, de egy olyan háromelemű rendezett sorozat lett az eredmény, amely két darab kételemű rendezett sorozatot is tartalmaz. Mindez azért történt, mert a három elemet vesszővel kapcsoltuk össze. Ha egyetlen, minden elemet tartalmazó rendezett sorozat a cél, akkor összefűzést kell alkalmaznunk:

```
>>> hair[:2] + ("gray",) + hair[2:]
('black', 'brown', 'gray', 'blonde', 'red')
```

Ha egy egyetlen elemből álló rendezett sorozatot szeretnénk kialakítani, akkor mindenképp kell vesszőt használnunk, de ha csak vesszőt használnánk, akkor `TypeError` kivétel lenne az eredmény – lévén a Python azt hinné, hogy egy karakterláncot és egy rendezett sorozatot akarunk összefűzni – így aztán a fenti példában a vesszőre *és* a zárójelre is szükség van.

A könyvben mostantól rendezett sorozatok használatakor egy bizonyos kódolási formához tartjuk magunkat. Ha egy kétényezős művelet bal oldalán, illetve egy egyténYezős művelet jobb oldalán áll a rendezett sorozat, a zárójelet elhagyjuk, de használni fogjuk minden más esetben. Íme néhány példa:

```
a, b = (1, 2)                                  # kétértékű művelet bal oldala
del a, b                                       # egyértékű művelet jobb oldala
def f(x):
    return x, x ** 2                           # egyértékű művelet jobb oldala
for x, y in ((1, 1), (2, 4), (3, 9)):          # kétértékű művelet bal oldala
    print(x, y)
```

Nem kötelező ezt a formát követni. Vannak programozók, akik inkább mindenütt használják a zárójelet – a rendezett sorozatok ábrázolási alakjához igazodva –, mások pedig kizárólag olyankor használnak zárójelet, ha az mindenképp szükséges.

```
>>> eyes = ("brown", "hazel", "amber", "green", "blue", "gray")
>>> colors = (hair, eyes)
>>> colors[1][3:-1]
('green', 'blue')
```

Ezúttal két rendezett sorozatot ágyaztunk be egy harmadik belsejébe. Minden további formalitás nélkül hozhatunk létre beágyazott gyűjteményeket, méghozzá akárhány szint mélységben. A szeletelés művelet jele (`[]`) részsorozatra is alkalmazható, egymás után annyiszor, ahányszor csak szükséges.

```
>>> things = (1, -7.5, ("pea", (5, "Xyz"), "queue"))
>>> things[2][1][1][2]
'z'
```

Alaposan tekintsük át a fenti példa működését. Kezdjük a `things[2]` kifejezéssel, amely a rendezett sorozat harmadik elemét adja vissza (hiszen az első elem a 0. sorszámot kapja), ami saját maga is egy rendezett sorozat `("pea", (5, "Xyz"), "queue")`. A `things[2][1]` kifejezés a `things[2]` kifejezés második elemét adja vissza, és ez szintén egy rendezett sorozat `(5, "Xyz")`. A `things[2][1][1]` kifejezés ennek a rendezett sorozatnak a második elemét adja vissza, amely az `"Xyz"` karakterlánc. Végül a `things[2][1][1][2]` kifejezés a karakterlánc harmadik elemét (karakterét), vagyis a `"z"` karaktert adja.

A rendezett sorozatok alkalmasak akárhány, akármelyik típusból származó elem tárolására, beleértve a gyűjteményes adattípusokat, vagyis a rendezett sorozatokat és a listákat is, lévén valójában objektumhivatkozásokat tartalmaznak. Az ehhez hasonló összetett adatszerkezetek használatakor könnyen elveszíthetjük a fonalat. A probléma egyik lehetséges megoldása, ha az adott sorszámú helyeknek nevet adunk:

```
>>> MANUFACTURER, MODEL, SEATING = (0, 1, 2)
>>> MINIMUM, MAXIMUM = (0, 1)
>>> aircraft = ("Airbus", "A320-200", (100, 220))
>>> aircraft[SEATING][MAXIMUM]
220
```

Ez a kifejezés kétségtelenül több jelentést hordoz, mint az `aircraft[2][1]` formájú megfelelője, de sok változó létrehozásával jár, ami pedig elég ronda dolog. A következőkben más megoldását ismerhetjük meg a problémának.

Az „aircraft" kódrészlet első két sorában rendezett sorozatoknak adtunk értéket. Ha az értékadás jobb oldalán sorozat típusú objektum áll (itt éppen egy rendezett sorozat), a balon pedig rendezett sorozat, akkor azt mondjuk, hogy a jobb oldalt *kicsomagoltuk*. A sorozatok kicsomagolása értékek felcserélésére is használható:

```
a, b = (b, a)
```

A jobb oldali zárójelre nincs feltétlenül szükség, de ahogy korábban megjegyeztük, a könyv kódolási formájában a kéttényezős műveletek bal oldali, illetve az egytényezős műveletek jobb oldali tényezőinél elhagyjuk, minden egyéb esetben pedig használjuk a zárójelet.

A sorozatok kicsomagolására láttunk már példát, mégpedig a `for...in` ciklusok esetében. Egy kis emlékeztető:

```
for x, y in ((-3, 4), (5, 12), (28, -45)):
    print(math.hypot(x, y))
```

A példában a ciklus kételemű rendezett sorozatokból álló rendezett sorozatot jár végig, az egyes kételemű rendezett sorozatokat pedig az `x` és az `y` változóba csomagolja ki.

Nevesített rendezett sorozatok

A nevesített rendezett sorozat pontosan úgy viselkedik, mint egy egyszerű rendezett sorozat, ugyanolyan teljesítményt nyújt, és egyéb jellemzői is azonosak. Annyival jelent többet, hogy a sorszámon kívül név szerint is hivatkozhatunk az elemeire – így pedig lehetővé válik adatelem-csoportok kialakítása.

A `namedtuple()` függvény a `collections` modulban található. A függvény egyedi adattípusok létrehozására szolgál. Például:

```
Sale = collections.namedtuple("Sale", "productid
                                       customerid date quantity price")
```

A `collections.namedtuple()` függvény első paramétere a létrehozni kívánt egyedi rendezettsorozat-típus neve, a második pedig egy szóközzel elválasztott nevekből álló karakterlánc, amelyben az egyes nevek az egyedi `tuple` típusú objektumokban tárolt adatokat fogják jelölni. Az első paraméternek, illetve a második paraméterben lévő neveknek a Python nyelv azonosítóira vonatkozó szabályokat kell követniük. A fenti példa esetében a `Sale` éppen úgy viselkedik, mint a Python nyelv bármely egyéb osztálya (mondjuk a `tuple`), eképp létrehozhatunk `Sale` típusú objektumokat.* Például:

```
sales = []
sales.append(Sale(432, 921, "2008-09-14", 3, 7.99))
sales.append(Sale(419, 874, "2008-09-15", 1, 18.49))
```

A példában egy olyan listát hoztunk létre, amely két `Sale` típusú elemből áll, vagyis két egyedi rendezett sorozatból. A rendezett sorozatok belsejében lévő elemekre a sorszámaikkal is hivatkozhatunk – például az első eladott tétel ára a `sales[0][-1]` kifejezés szerint 7.99 lesz – de használhatunk neveket is, ami a kódot lényegesen olvashatóbbá teszi:

```
total = 0
for sale in sales:
    total += sale.quantity * sale.price
print("Total ${0:.2f}".format(total)) # prints: Total $42.46
```

Az az olvashatóság és kényelem, amit a nevesített rendezett sorozatok adnak a kezünkbe, gyakran bizonyul hasznosnak. Példának kínálkozik mindjárt a korábbi „aircraft", amely így szépül meg:

```
>>> Aircraft = collections.namedtuple("Aircraft",
...                                   "manufacturer model seating")
>>> Seating = collections.namedtuple("Seating", "minimum maximum")
>>> aircraft = Aircraft("Airbus", "A320-200", Seating(100, 220))
>>> aircraft.seating.maximum
220
```

Bár a nevesített rendezett sorozatok használata igen kényelmes lehet, a 6. fejezetben bemutatjuk az objektumközpontú programozást, és az egyszerű nevesített rendezett sorozatok használatán túllépve megtanuljuk az egyedi adattípusok létrehozását, amelyek azon felül, hogy adatelemeket tárolnak, saját egyedi tagfüggvényekkel is rendelkeznek.

---

* Megjegyezzük, hogy az objektumközpontú programozásban az így létrejött objektumosztályok a `tuple` osztály alosztályai.

Listák

Lista alatt nulla vagy annál több objektumhivatkozás rendezett sorozatát értjük. A listák bejárása és szeletelése a karakterláncokkal azonos utasításformát követ. Ezáltal egyszerűvé válik a listák elemeinek kinyerése. A karakterláncoktól és a rendezett sorozatoktól eltérően a listák változtathatók (mutable), vagyis lehetőség nyílik elemek törlésére, illetőleg cseréjére. Lehetőség van továbbá részsorozatok beillesztésére, cseréjére és törlésére is.

Karakterláncok szeletelése és egyelése 64 ☞

Lehetőség van a `list` adattípus függvényként való meghívására is: `list()`. A paraméterek nélküli hívás visszatérési értéke egy üres lista, amennyiben pedig az átadott paraméter egy már létező, `list` típusú objektum objektumhivatkozása, a függvény a paraméter sekély másolatát adja vissza. Bármely más paraméter esetében kísérlet történik az adott objektum `list` típusúvá alakítására. Lista létrehozására a `list()` függvény meghívása nélkül is van lehetőség. Üres szögletes zárójel, vagyis `[]` használatakor üres lista jön létre, a szögletes zárójelen belül vesszők használatával pedig a felsorolt elemekből képzett lista. Listák létrehozására további módszert a listaértelmezések használata kínál – a témakört az alfejezet későbbi részében tárgyaljuk.

Sekély és mély másolás ☞ 143

Listaértelmezések ☞ 113

Minthogy a listák elemei valójában objektumhivatkozások, listák a rendezett sorozatokhoz hasonlóan bármely adattípusba tartozó elemekből kialakíthatók, értve ezalatt a gyűjteményes adattípusokat, például a listákat és a rendezett sorozatokat is. A szokásos összehasonlító műveletekkel (<, <=, ==, !=, >=, >) lehetőség nyílik a listák összehasonlítására is, mégpedig elemenkénti összehasonlítással (egymásba ágyazott elemek, például listán belüli rendezett sorozatok, illetve listák esetében önhívó összehasonlítással).

Az `L = [-17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]` értékadó utasítás kiadásakor a 3.2. ábrán látható lista jön létre.

L[-6]	L[-5]	L[-4]	L[-3]	L[-2]	L[-1]
-17.5	'kilo'	49	'V'	['ram', 5, 'echo']	7
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

3.2. ábra

A lista elemeinek sorszáma és helye

A fenti `L` nevű listán már végrehajtható a szeletelés művelete –szükség esetén többször is –, így lehetőség nyílik a lista elemeinek elérésére. A következő egyenlőségek is ezt mutatják:

```
L[0] == L[-6] == -17.5
L[1] == L[-5] == 'kilo'
L[1][0] == L[-5][0] == 'k'
L[4][2] == L[4][-1] == L[-2][2] == L[-2][-1] == 'echo'
L[4][2][1] == L[4][2][-3] == L[-2][-1][1] == L[-2][-1][-3] == 'c'
```

A listák egymásba ágyazhatók, végigjárhatók, képezhetők belőlük részsorozatok – akárcsak a rendezett sorozatok. Az a helyzet, hogy az előző alfejezet rendezett sorozatokkal kapcsolatos példái a rendezett sorozatok helyett listákat használva ugyanúgy működnének. A listák támogatják a tagsági viszonyt vizsgáló `in` és `not in` műveletet. Az összefűzés a `+`, a lista bővítése (a műveleti jeltől jobbra álló elemek hozzáfűzése a bal oldali listához) pedig a `+=` jelű művelettel lehetséges. Szorzásra a `*` és a `*=` jelű műveletek használhatók. Listák paraméterül szolgálhatnak a beépített `len()` függvény, valamint az alább, illetve a fejezet *Elemek törlése a del utasítással* című részében tárgyalt `del` utasítás számára. A listák ezen felül a 3.1. táblázatban tárgyalt tagfüggvényeket is a rendelkezésünkre bocsátják.

Bár a szeletelés műveletével kinyerhetők elemek a listákból, vannak helyzetek, amikor egy listának akár több elemét egyetlen lépésben szeretnénk megkapni. Ez a sorozatkicsomagolás műveletével valósítható meg. Bármely bejárható típus (lista, rendezett sorozat stb.) kicsomagolható a sorozatkicsomagolás műveletével. A művelet jele a csillag (`*`). Ha egy értékadó művelet bal oldalán legalább kettő változó áll, és ezek egyikét csillag előzi meg, akkor az egyes változókhoz a megfelelő elemeket rendeljük, míg a megcsillagozott változó a megmaradó elemeket kapja értékül. Íme néhány példa:

```
>>> first, *rest = [9, 2, -4, 8, 7]
>>> first, rest
(9, [2, -4, 8, 7])
>>> first, *mid, last = "Charles Philip Arthur George
Windsor".split()
>>> first, mid, last
('Charles', ['Philip', 'Arthur', 'George'], 'Windsor')
>>> *directories, executable = "/usr/local/bin/gvim".split("/")
>>> directories, executable
(['', 'usr', 'local', 'bin'], 'gvim')
```

Ha a sorozatot kicsomagoló műveletet a fent ismertetett módon használjuk, a `*rest` kifejezésre és a hasonló kifejezésekre *megcsillagozott kifejezés* névvel utalunk.

3.1. táblázat *A lista adattípus tagfüggvényei*

Utasításforma	Leírás
`L.append(x)`	Az `x` elem az `L` nevű lista végére kerül.
`L.count(x)`	Az `x` elem `L` nevű listán belüli előfordulásainak számát adja vissza.
`L.extend(m)` `L += m`	Az `m` bejárható típusú objektum elemeit az `L` nevű lista végére fűzi, a `+=` művelet ugyanezt teszi.
`L.index(x,` *`kezdet, vég`*`)`	Az `x` elem `L` nevű listán belüli (illetve a lista *`kezdet:vég`* részsorozatán belüli) bal szélső előfordulásának sorszámát adja vissza; vagy `ValueError` kivételt vált ki.
`L.insert(i, x)`	Az `x` elemet az `L` nevű lista `i` egész típusú értékkel jelölt helyére szúrja be.

3.1. táblázat *A lista adattípus tagfüggvényei* *(folytatás)*

Utasításforma	Leírás
`L.pop()`	Az `L` nevű lista jobb szélén álló elemét adja vissza, és eltávolítja az elemet a listából.
`L.pop(i)`	Az `L` nevű lista végén álló `i` egész típusú értékkel jelölt helyén álló elemet adja vissza, az elemet a listából eltávolítva.
`L.remove(x)`	Az `x` elem `L` nevű listán belüli bal szélső előfordulását megszünteti; vagy ha nem talál `x` elemet a listában, `ValueError` kivételt vált ki.
`L.reverse()`	Az `L` nevű listát helyben megfordítja.
`L.sort(...)`	Az `L` nevű listát helyben rendezi; a tagfüggvény ugyanazokat a *`key` és `reverse`* elhagyható paramétereket használja, mint a `sorted()` nevű beépített függvény.

`sorted()`
☞ 112, 139, 178

A Python nyelven belül létezik egy hasonló, *megcsillagozott paraméter* nevű dolog is. Például a következő függvény három paramétert vár:

```
def product(a, b, c):
    return a * b * c # a * itt a szorzást jelöli
```

A függvény három paraméterrel vagy megcsillagozott paraméterekkel hívható meg:

```
>>> product(2, 3, 5)
30
>>> L = [2, 3, 5]
>>> product(*L)
30
>>> product(2, *L[1:])
30
```

Az első híváskor a három paramétert a megszokott módon adjuk át. A második hívásnál megcsillagozott paramétert használunk – annyi történik, hogy a háromelemű listát a * művelet kicsomagolja, vagyis amennyire a függvényt érinti a dolog, megkapja a három várt paramétert. Ugyanezt az eredményt egy háromelemű rendezett sorozat használatával is elérhettük volna. A harmadik függvényhíváskor az első paramétert a hagyományos módon adjuk át, a maradék kettőt pedig az `L` listából kicsomagolt kételemű részsorozatként. A függvények és a paraméterátadás kimerítő tárgyalására a 4. fejezetben kerül sor.

Elemek törlése a del utasítással

Bár a `del` utasítás neve a *delete* (törlés) szóból származik, a `del` utasítás nem feltétlenül végez bármiféle törlést. Amikor nem gyűjteményben lévő objektumhivatkozással kapcsolatosan használjuk, a `del` megszünteti az objektumhivatkozás és az objektum kapcsolatát, és az *objektumhivatkozást* törli. Például:

```
>>> x = 8143 # létrejön az x objektumhivatkozás;
             # létrejön a 8143 értékű, int típusú objektum
>>> x
8143
>>> del x # az x objektumhivatkozás törlése;
          # az int típusú értékre szemétgyűjtés vár
>>> x
Traceback (most recent call last):
...
NameError: name 'x' is not defined
```

Ha egy objektumhivatkozást törlünk, és a hivatkozás céljaként szereplő objektumra más hivatkozás nem mutat, a Python az objektumot kijelöli szemétgyűjtésre. Azt nem tudhatjuk, hogy a szemétgyűjtés mikor történik meg, ha megtörténik egyáltalán – ez ugyanis a Python-megvalósítástól függ – tehát ha mindenképpen törölni akarunk, magunknak kell intézni a dolgot. A Python két megoldást kínál. Az egyik a `try...finally` blokk használata a törlés kikényszerítésére, a másik pedig a `with` kifejezés alkalmazása (lásd a 8. fejezetben).

Ha a `del` utasítást gyűjteményes adattípusokon – például rendezett sorozaton vagy listán – használjuk, csak a gyűjteményre mutató objektumhivatkozás törlése történik meg. A gyűjteményre és az elemeire a szemétgyűjtő vár, hacsak nincs másik, rájuk mutató objektumhivatkozás (az olyan elemek esetében, amelyek saját maguk is gyűjtemények, mindez önhívással történik).

A változtatható gyűjtemények, például a listák esetében a `del` utasítás egyedi elemekre vagy szeletekre is vonatkozhat – mindkét esetben a szeletelés (`[]`) műveletét kell használnunk. Ha a hivatkozott elemet, illetve elemeket eltávolítottuk a gyűjteményből, és rájuk más objektumhivatkozás nem hivatkozik, megint csak a szemétgyűjtés lesz a sorsuk.

A csillag művelet esetében sohasem lehet szó bizonytalanságról – sohasem kétséges, hogy kicsomagolást vagy szorzást végez. Ha értékadó művelet bal oldalán szerepel, akkor kicsomagol, ha máshol (például függvényhívásban) egytényezős műveletben szerepel, akkor is kicsomagol, ha pedig kéttényezős műveletben szerepel, akkor szoroz.

Láttuk már, hogy a listák a `for` *`elem`* `in` *`L`* utasításformával bejárhatók. Ha a lista elemeit meg szeretnénk változtatni, a következő szerkezetet kell használnunk:

```
for i in range(len(L)):
    L[i] = feldolgozás(L[i])
```

A bejáráshoz szükséges egész értékeket a `range()` beépített függvény adja meg. Ha egy *n* egész paramétert adunk át a `range()` függvénynek, akkor a 0, 1, …, *n-1* sorozatot kapjuk.

Ezzel a módszerrel egy egész számokból álló lista értékeit is növelhetjük. Például:

```
for i in range(len(numbers)):
    numbers[i] += 1
```

range()
☞ 136

Minthogy a lista típus támogatja a szeletek használatát, több esetben azonos hatás érhető el szeleteléssel, illetve a lista típus egyik tagfüggvényének használatával. Példának okáért legyen adott a `woods = ["Cedar", "Yew", "Fir"]` lista, amelynek a bővítése az alábbi módszerek bármelyikével lehetséges:

```
woods += ["Kauri", "Larch"]
```

```
woods.extend(["Kauri", "Larch"])
```

Az eredmény mindkét esetben a `['Cedar', 'Yew', 'Fir', 'Kauri', 'Larch']` lista lesz.

A lista végére egyes elemeket a `list.append()` tagfüggvénnyel helyezhetünk. A `list.insert()` tagfüggvénnyel adott sorszámú helyre új elemet szúrhatunk be, de a beszúrás úgy is megvalósítható, hogy egy nulla hosszúságú szeletnek adjuk értékül a beszúrandó elemet. Például, ha adott a `woods = ["Cedar", "Yew", "Fir", "Spruce"]` lista, az új érték beszúrása a második sorszámú helyre (vagyis a lista harmadik helyére) az alábbi utasítások bármelyikével elvégezhető:

```
woods[2:2] = ["Pine"]
```

```
woods.insert(2, "Pine")
```

Az eredmény mindkét esetben a `['Cedar', 'Yew', 'Pine', 'Fir', 'Spruce']` lista lesz.

A listában egyedi elemek cseréje az adott sorszámú elem újbóli értékadásával történik, például: `woods[2] = "Redwood"`. A szeletek cseréje úgy történik, hogy valamilyen bejárható objektumot adunk értékül a szeletnek, például: `woods[1:3] = ["Spruce", "Sugi", "Rimu"]`. A szeletnek és az átadott bejárható objektumnak nem kell azonos elemszámúnak lennie. Mindig az történik, hogy a lista elemeit eltávolítjuk, és utána beszúrjuk az átadott elemeket. Így amennyiben az átadott elemek kevesebben vannak a listából kivett elemeknél, a lista rövidebb lesz; ha pedig több elemet adtunk át, mint amennyit a listából kivettünk, a lista hossza megnő.

Hogy igazán egyértelművé tegyük azt, hogy mi történik, amikor a listaszeletnek egy bejárható objektumot adunk értékül, tekintsük át a folyamatot egy további példa segítségével. Tegyük fel, hogy van egy `L = ["A", "B", "C", "D", "E", "F"]` listánk, amely egy szeletének az `L[2:5] = ["X", "Y"]` utasítással értékül adunk egy bejárható objektumot (ami esetünkben szintén egy lista). Először a szelet eltávolítására kerül sor, vagyis a színfalak mögött a lista ekkor az `['A', 'B', 'F']` formát ölti. Ennek végeztével a bejárható objektum értékeinek beszúrása következik, mégpedig a szelet elejétől kezdve, vagyis az eredményül kapott lista az `['A', 'B', 'X', 'Y', 'F']` lesz.

Elemek eltávolítására számos módszer kínálkozik. A `list.pop()` tagfüggvénnyel a lista utolsó eleme távolítható el – ráadásul az eltávolított értéket a függvény visszatérési értékeként visszakapjuk. Ha a `list.pop()` tagfüggvény egy egész értéket kap paraméterül, akkor a megfelelő sorszámú helyen álló elemet távolítja el (és adja vissza a hívónak). Az elemek eltávolításának további módja, ha a `list.remove()` tagfüggvényt hívjuk meg, az eltávolítandó elemet paraméterként megadva. Az elemek egyesével történő eltávolítására a `del` utasítás is használható – például a `del woods[4]` formában –, és az utasítás szeletek eltávolítására is alkalmas. Szeletek eltávolítása úgy is lehetséges, hogy a szeletnek egy üres listát adunk értékül, vagyis az alábbi két kódrészlet egyenértékű:

```
woods[2:4] = []
```

```
del woods[2:4]
```

A bal oldali kódrészletben egy bejárható objektumot (egy üres listát) adtunk értékül a szeletnek, vagyis először törlődött a szelet, ezt követően pedig – lévén a beszúrandó objektum üres volt – nem szúrtunk be semmit.

Amikor először foglalkoztunk a szeleteléssel és az egyeléssel (végigjárással), a karakterláncoknál időztünk, vagyis olyan helyen, ahol az egyelés csak kevéssé volt érdekes. A listák esetében azonban az egyelés lehetővé teszi minden *n*-edik elem elérését, ami sokszor lehet hasznos. Például tegyük fel, hogy van egy `x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` listánk, amelynek minden páratlan sorszámú elemét (vagyis az `x[1]`, `x[3]` stb. elemet) 0 értékűvé akarjuk tenni. Egyeléssel például így érhető el minden második elem: `x[::2]`. De így a 0,2,4, stb. sorszámú elemeket fogjuk megkapni. Ezen úgy segíthetünk, hogy megadunk egy induló sorszámot, vagyis az egyelés immáron az `x[1::2]` formát ölti, és éppen a szükséges szeletet adja. Ha a szelet minden értékét nullázni akarjuk, szükségünk van egy nullákból álló listára, amelyben pontosan annyi nulla van, mint ahány tagú a szelet.

Szeletelés és egyelés 64 ☜

Íme a teljes megoldás: `x[1::2] = [0] * len(x[1::2])`. Az utasítás végrehajtása után az `x` lista a következő lesz: `[1, 0, 3, 0, 5, 0, 7, 0, 9, 0]`. A szorzás (`*`) műveletet a megfelelő számú nullából álló sorozat előállítására használtuk, a „megfelelő számot" pedig a szelet hossza (vagyis az elemeinek száma) adja meg. A dolog érdekessége, hogy amikor a `[0, 0, 0, 0, 0]` listát az *egyeléssel képzett, a második elemekből álló* listának adjuk értékül, a Python az `x[1]` értékéül az első, az `x[3]` értékéül a második nullát adja, és így tovább.

A listák ugyanúgy, ahogy bármely más bejárható adattípus, megfordíthatók és rendezhetők. A műveletek elvégzésére *A bejárók és a bejárható adattípusok műveletei és függvényei* című részben ismertetett `reversed()`, illetve `sorted()` függvények használhatók. A függvényeknek megfelelő tagfüggvények – `list.reverse()` és `list.sort()` – is megtalálhatók a lista típusnál; mindkettő helyben dolgozik, vagyis nem adnak vissza semmit. A `list.sort()` tagfüggvény és a `sorted()` függvény ugyanazokat az elhagyható paramétereket várják. Elterjedt gyakorlat, hogy a karakterláncokból álló listát a kis- és nagybetűkre való tekintet nélkül rendezzük – például a `woods` nevű listát rendezhetnénk így:

`woods.sort(key=str.lower)`. A `key` paramétert használjuk az olyan függvények megadására, amelyeket a lista minden elemére alkalmazni kell, és amelyeknek a visszatérési értékét használjuk a rendezés végrehajtásakor. Ahogy azt az előző fejezetben, a karakterláncok összehasonlítását tárgyalva megjegyeztük, az angoltól eltérő nyelvű karakterláncok értelmes rendezése meglehetős kihívásokat rejt magában.

`sorted()`
☞ 139, 178

Az elemek beszúrását tekintve a listaműveletek olyankor a leggyorsabbak, ha a beszúrás, illetve a törlés a lista végén történik (`list.append()`, `list.pop()`). A leglassabb műveletek azok, amelyeknél keresni kell a lista elemeit; példaként a `list.remove()` és a `list.index()` tagfüggvények, illetve az `in` tagsági művelet használatát említhetjük. Ha a keresést, illetve a tagság eldöntését gyorsítani akarjuk, a `set` vagy a `dict` (mindkettőről lesz még szó a fejezetben) használata jobb eredménnyel kecsegtet. Érdemes tudnunk, hogy ha a listáinkat rendben, rendezetten tartjuk, a további rendezések igen gyorsak lehetnek – a Python nyelv rendezési algoritmusát részlegesen rendezett listákra különösen jól optimalizálták. A bináris keresések – a `bisect` modullal használhatjuk őket – szintén gyorsak. (A 6. fejezetben létre fogunk hozni egy belsőleg rendezett egyedi listaosztályt.)

Listaértelmezések

Rövidebb listákat gyakran hozunk létre az egyes értékek megadásával, de a hosszabb listák megadása rendszerint programból történik. Egész számokból álló listák létrehozására a `list(range(n))` utasítást használhatjuk – ha csak magukra az egész számokra van szükség, akkor a `range()` függvény is elegendő – másfajta listák előállításakor pedig a `for...in` ciklus használata a bevett gyakorlat. Tegyük fel például, hogy egy adott időszak szökőéveinek listáját szeretnénk elkészíteni. Valahogy így fogunk a dologhoz:

```
leaps = []
for year in range(1900, 1940):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        leaps.append(year)
```

Ha a `range()` beépített függvénynek átadunk két egész (*n* és *m*) paramétert, akkor visszatérési értékül az *n*, *n+1*, ..., *m-1* egész számokból álló sorozatot kapjuk.

`range()`
☞ 136

Természetesen amennyiben előzőleg ismerjük az időszak értékét, használhatunk saját kezűleg megadott listát is, például: `leaps = [1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936]`.

A *listaértelmezés* (list comprehension) egy kifejezés és egy ciklus olyan együttese, amely elhagyható, szögletes zárójelben megadott feltétellel is bírhat. A ciklus feladata a listaelemek előállítása, és a feltétel felelős a nemkívánatos elemek kiszűréséért. A listaértelmezés legegyszerűbb formája az alábbi:

```
[elem for elem in bejárható_objektum]
```

Ez a forma a *bejárható_objektum* elemeiből képzett listával fog visszatérni, és végeredményét tekintve nem különbözik a `list(`*bejárható_objektum*`)` utasítástól. Két olyan dolog van, ami a listaértelmezéseket érdekesebbé és használhatóbbá teszi. Az egyik az, hogy lehetőségünk nyílik kifejezések használatára, a másik pedig az, hogy feltételt adhatunk meg – és ezzel el is jutottunk a listaértelmezés általános utasításformáihoz:

```
[kifejezés for elem in bejárható_objektum]
[kifejezés for elem in bejárható_objektum if feltétel]
```

A második forma egyenértékű az alábbiakkal:

```
temp = []
for elem in bejárható_objektum:
    if feltétel:
        temp.append(kifejezés)
```

Az esetek nagy részében a *kifejezés* vagy megegyezik az *elem*-mel, vagy az értékét befolyásolja az *elem* értéke. Természetesen a listaértelmezésnek nincs szüksége a `for...in` ciklussal megvalósított változat `temp` változójára.

Immáron újraírhatjuk a `leaps` listát előállító kódrészletet, mégpedig listaértelmezés használatával. A kód megírását három lépésben végezzük el. Először egy olyan listát hozunk létre, amely az adott időszak minden évét tartalmazza:

```
leaps = [y for y in range(1900, 1940)]
```

Ezt az eredményt értük volna el a `leaps = list(range(1900, 1940))` utasítással is. Második lépésként megfogalmazunk egy egyszerű feltételt, hogy csak minden negyedik év maradjon a listában:

```
leaps = [y for y in range(1900, 1940) if y % 4 == 0]
```

Végül előáll a kész változat:

```
leaps = [y for y in range(1900, 1940)
    if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
```

A listaértelmezés használatával ebben az esetben négysorosról kétsorosra csökkent a kód hossza – a megtakarítás kicsi, de nagy projekteknél a sok kicsi is számít.

Minthogy a listaértelmezések listákat, vagyis bejárható objektumokat állítanak elő, a listaértelmezések utasításformájában pedig szintén bejárható objektumot használunk, lehetséges a listaértelmezések egymásba ágyazása. Ez egyenértékű az egymásba ágyazott `for...in` ciklusok használatával. Például, ha elő akarjuk állítani az összes lehetséges ruhacímke-változatot adott nemek, méretek és színek tekintetében, kivéve a divatipar által rutinszerűen mellőzött molett alkatú hölgyek esetét, a feladatot megoldhatjuk egymásba ágyazott `for...in` ciklusokkal:

```
codes = []
for sex in "MF": # Male, Female
    for size in "SMLX": # Small, Medium, Large, eXtra large
        if sex == "F" and size == "X":
            continue
        for color in "BGW": # Black, Gray, White
            codes.append(sex + size + color)
```

A kód egy huszonegy elemű listát `['MSB', 'MSG', ..., 'FLW']` ad. Ugyanez az eredmény elérhető pár sornyi listaértelmezést használó kóddal is:

```
codes = [s + z + c for s in "MF" for z in "SMLX" for c in "BGW"
         if not (s == "F" and z == "X")]
```

Ebben az esetben a listát az `s + z + c` kifejezés hozza létre. A listaértelmezés logikája némileg abban is különbözik, hogy itt a legbelső ciklusban hagyjuk ki az érvénytelen nem-méret kombinációkat, míg a `for...in` ciklusokon alapuló változatban ezt a középső ciklusban végezzük el. Bármely listaértelmezést használó kódrészlet átírható `for...in` ciklus vagy ciklusok használatával.

Generátorok
☞ 346

Ha az előállított lista túlságosan nagy, az egész lista egy lépésben történő előállítása helyett esetleg hatékonyabb, ha az egyes elemeket csak akkor állítjuk elő, ha már szükség van rájuk. Ez úgy valósítható meg, hogy listaértelmezés helyett generátort használunk. A generátorokról később, a 8. fejezetben lesz szó.

Halmaztípusok

A *halmaz* (set) típus támogatja a tagsági műveletet (`in`), vizsgálható a mérete (`len()`), valamint bejárható. A halmaztípusoknak ezen felül rendelkezniük kell `set.isdisjoint()` tagfüggvénnyel, és támogatniuk kell az összehasonlító, valamint a bitenkénti műveleteket (a halmazok esetében ez utóbbiakat unióképzésre, metszetképzésre és hasonlókra használjuk). A Python nyelv két beépített halmaztípussal rendelkezik: a változtatható `set` (halmaz) és az állandó értékű `frozenset` (fagyasztott halmaz) típussal. Bejáráskor a halmaz típusú objektumok elemeinek sorrendje nem befolyásolható.

Halmazba csak *kivonatolható* (hashable) objektumokat helyezhetünk. A kivonatolható objektumok olyan objektumok, amelyek rendelkeznek a `__hash__()` megkülönböztetett szerepű tagfüggvénnyel – ennek visszatérési értéke az objetum élete során változatlan marad –, és más objektumokkal való egyezésük vizsgálható a `__eq__()` megkülönböztetett szerepű tagfüggvénnyel. (A megkülönböztetett szerepű tagfüggvények – ezeknek a neve elején és végén egyaránt két aláhúzás áll – tárgyalására a 6. fejezetben kerül sor.)

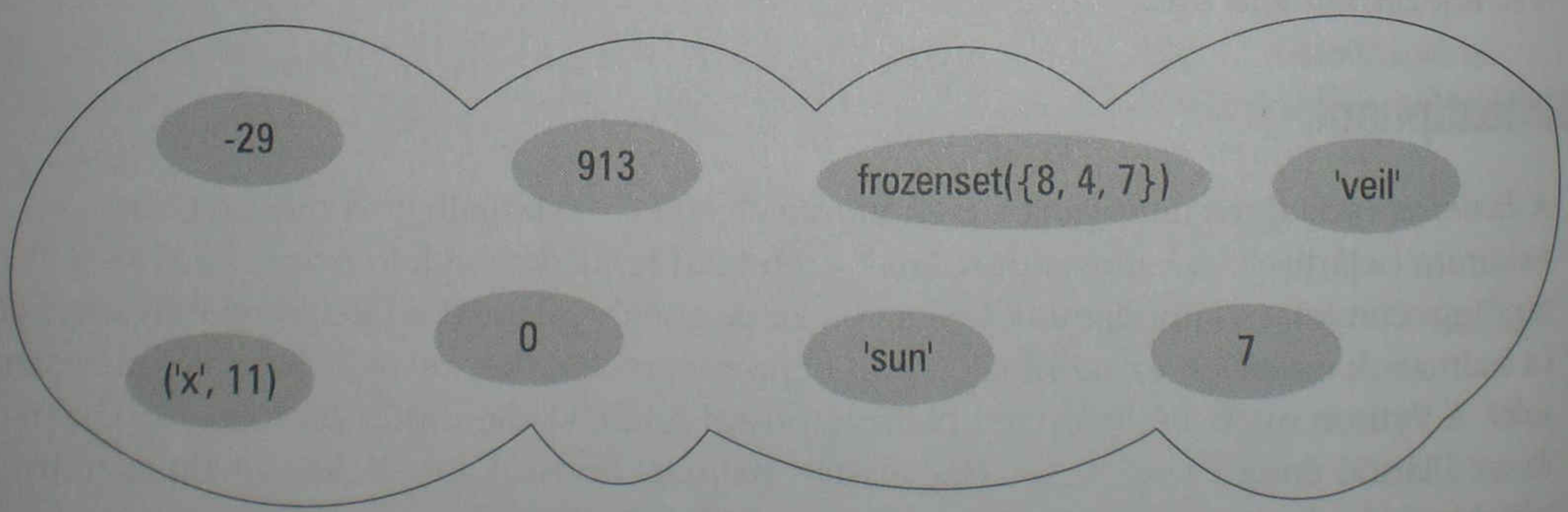
A beépített állandó értékű adattípusok (mint a `float`, a `frozenset`, az `int`, az `str` és a `tuple`) mindegyike kivonatolható, és így halmazba helyezhető. A beépített változtatható adattípusok (a `dict`, a `list` és a `set`) nem kivonatolhatók, hiszen a kivonatértékük (hash) a bennük tárolt objektumok változásával megváltozik. Így nem lehetnek halmaz tagjai azok az objektumok, amelyek ezekbe az adattípusokba tartoznak.

A halmaztípusok a megszokott összehasonló műveletekkel (<, <=, ==, !=, >=, >) hasonlíthatók össze. Érdemes megjegyeznünk, hogy bár az == és a != műveletek értelmezése a megszokott, az összehasonlítás elemről elemre (a halmazokon belüli rendezett sorozatok, fagyasztott halmazok és egyéb beágyazott elemek esetében pedig önhívással) történik, a többi összehasonlító művelet pedig halmaz–részhalmaz viszonyokat vizsgál – amint azt hamarosan látni fogjuk.

Halmazok

A halmaz (`set`) típus nulla vagy több kivonatolható objektumra mutató objektumhivatkozás rendezetlen gyűjteménye. A halmazok megváltoztathatók, vagyis egyszerűen adhatunk új elemet egy halmazhoz, és az elemek eltávolítása sem bonyolult. Minthogy a halmazok rendezetlenek, az elemeik sorszáma nincs nyilvántartva, nem lehetséges belőlük szeletet (részsorozatot) képezni, és az egyelésükre sincs mód. A 3.3. ábrán az alábbi kódrészlet által létrehozott halmazt tekinthetjük meg:

```
S = {7, "veil", 0, -29, ("x", 11), "sun", frozenset({8, 4, 7}), 913}
```



3.3. ábra

A halmaz egyedi elemek rendezetlen gyűjteménye

Lehetőség van a `set` adattípus függvényként való meghívására is: `set()`. A paraméterek nélküli hívás visszatérési értéke egy üres halmaz, amennyiben pedig az átadott paraméter egy `set` típusú objektum objektumhivatkozása, a függvény a paraméter sekély másolatát adja vissza. Bármely más paraméter esetében kísérlet történik az adott objektum `set` típusúvá alakítására. Elemekkel rendelkező halmaz létrehozására a `set()` függvény meghívása

Sekély és mély másolás ☞ 143

Halmazértelmezések ☞ 120

nélkül is van lehetőség, de az üres halmazokat a `set()` függvénnyel kell létrehoznunk, az üres kapcsos zárójel erre *nem* használható.* Nullától eltérő elemszámú halmaz létrehozására az elemek kapcsos zárójelbe tett, vesszővel tagolt felsorolása használható. Halmazok létrehozására további módszert a halmazértelmezések használata kínál – a témakört az alfejezet későbbi részében tárgyaljuk.

A halmazok értékei mindig egyediek – egy elem többszöri elhelyezése a halmazban baj nélkül kivitelezhető, ám értelmetlen. A következő három halmaz például azonos: `set("apple")`, `set("aple")`, és `{'e','p', 'a', 'l'}`. Erre való tekintettel a halmazokat gyakran használják elemek többszöri előfordulásának megszüntetésére. Például ha az `x` egy karakterláncokból álló listát jelöl, az `x = list(set(x))` utasítás végrehajtását követően az `x` lista minden karakterlánca egyedi lesz – de a sorrendjük is felborul.

A halmazokban támogatott a `len()` nevű, beépített függvény használata, illetve a tagsági viszony gyors, az `in` és a `not in` műveleteken alapuló vizsgálata. Ahogy a 3.4. ábrán megfigyelhető, használhatók a szokásos műveletek is. A halmazok tagfüggvényeinek és műveleteinek teljes listáját a 3.2. táblázatban adjuk meg. Az „update”-tagfüggvények (a `set.update()`, a `set.intersection_update()` és társaik) paraméterül kaphatnak bármely bejárható objektumot – a megfelelő műveletek (`|=`, `&=` és a többiek) megkövetelik, hogy mindkét tényezőjük halmaz legyen.

A halmazokat elterjedten használják a tagsági viszony fennállásának gyors eldöntésére is. Például lehetséges, hogy ha a felhasználó nem ad meg parancssori paramétert, illetve a `-h` vagy a `--help` paramétert adja meg, akkor tájékoztatni szeretnénk a program használatáról:

```
if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
```

A halmazokat szintén általánosan használjuk annak biztosítására, hogy nem dolgozunk fel egy adatot többször is. Például tegyük fel, hogy valamilyen bejárható objektum – mondjuk egy lista – tartalmazza a webkiszolgálónk naplófájljában fellelhető IP-címeket, amelyeket egyesével szeretnénk feldolgozni. Feltételezve, hogy az IP-címek kivonatolhatók, és az `ips` nevű, bejárható objektum tárolja őket, illetve hogy mindegyik esetében a `process_ip()` nevű, már megadott függvény hívásával szeretnénk elvégezni a feldolgozást, a következő kódrészletek egyformán azt teszik, amit szeretnénk, de némileg eltérő viselkedéssel:

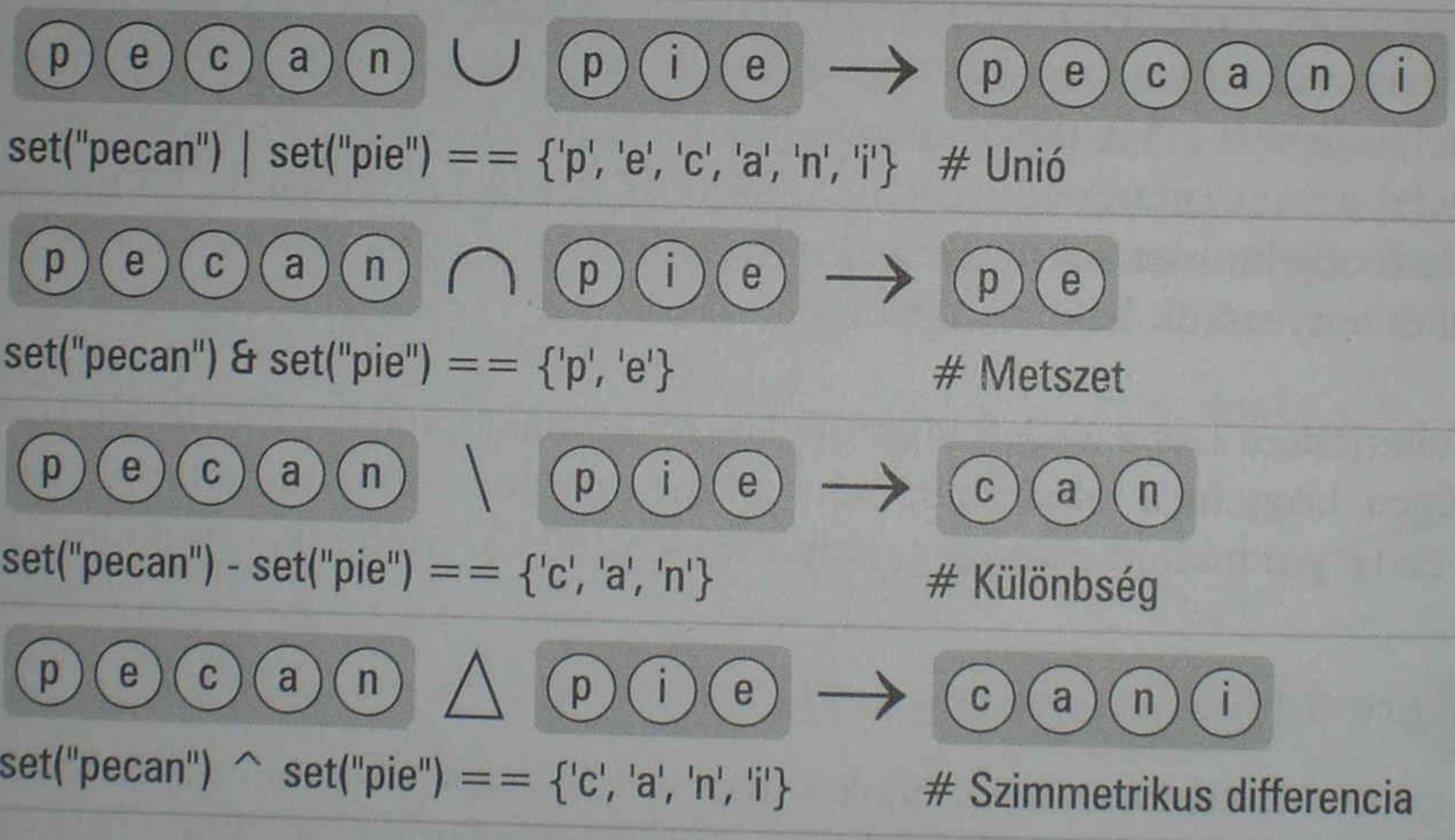
```
seen = set()
for ip in ips:
if ip not in seen:
seen.add(ip)
process_ip(ip)
```

```
for ip in set(ips):
process_ip(ip)
```

* Az üres kapcsos zárójel, a `{}`, üres `dict` típusú objektumot hoz létre, ahogy az a következő szakaszból kiderül.

A bal oldali kódrészletben a még fel nem dolgozott IP-címet behelyezzük a `seen` halmazba, majd feldolgozzuk; a már feldolgozott címeket pedig átlépjük. A jobb oldali kódrészletben a feldolgozásig minden IP-cím csak egyszer jut el. A két kódrészlet különbsége egyrészt abban rejlik, hogy a bal oldali létrehozza a `seen` halmazt, ami a jobb oldali esetében szükségtelen, másrészt pedig abban, hogy a bal oldali kódrészlet az `ips` objektumban elfoglalt sorrendjükben dolgozza fel az IP-címeket, míg a jobb oldalinál a sorrend megjósolhatatlan.

A jobb oldali módszer könnyebben kódolható, amennyiben azonban az `ips` objektumban elfoglalt sorrend jelentőséggel bír, akkor vagy a bal oldali módszert kell használnunk, vagy a jobb oldal első sorának ilyesféle átírása válik szükségessé: `for ip in sorted(set(ips)):` – ami esetleg nem elég hatékony a szükséges sorrend kialakítását tekintve. Ha az `ips` objektumban lévő elemek száma nagyon nagy, a jobb oldali módszer elvileg lassabb lesz, hiszen a halmazt folyamatos bővítés helyett egyetlen lépésben állítja elő.



3.4. ábra

A szokásos halmazműveletek

3.2. táblázat *A halmaz adattípus tagfüggvényei és műveletei*

Utasításforma	Leírás
`s.add(x)`	Az `x` elemet az `s` halmazba helyezi, ha még nincs benne.
`s.clear()`	Minden elemet eltávolít az `s` halmazból.
`s.copy()`	Az `s` halmaz sekély másolatát adja vissza*.
`s.difference(t)` `s - t`	Új halmazt ad vissza, amelyben az `s` halmaz minden eleme benne van, ami a `t` halmazban nincs benne.*
`s.difference_update(t)` `s -= t`	Eltávolítja az `s` halmaz minden olyan elemét, amely a `t` halmaznak is eleme.

Sekély és mély másolás ☞ 143

3.2. táblázat *A halmaz adattípus tagfüggvényei és műveletei* *(folytatás)*

Utasításforma	Leírás
`s.discard(x)`	Eltávolítja az `x` elemet az `s` halmazból, ha benne van; lásd még a `set.remove()` tagfüggvényt.
`s.intersection(t)` `s & t`	Új halmazt ad vissza, amelyben az `s` és a `t` halmazokban egyaránt előforduló elemek vannak.*
`s.intersection_update(t)` `s &= t`	Az `s` halmazt a saját magának a `t` halmazzal képzett metszetéből származó elemekkel tölti fel.
`s.isdisjoint(t)`	`True` értékkel tér vissza, ha az `s` és a `t` halmaznak nincs közös eleme.
`s.issubset(t)` `s <= t`	`True` értékkel tér vissza, ha az `s` halmaz egyenlő a `t` halmazzal, vagy annak részhalmaza; az `s < t` művelet használatával dönthető el, hogy az `s` halmaz valódi részhalmaza-e a `t` halmaznak.
`s.issuperset(t)` `s >= t`	`True` értékkel tér vissza, ha az `s` halmaz egyenlő a `t` halmazzal, vagy bővebb nála; az `s > t` művelet használatával dönthető el, hogy az `s` halmaz valódi bővített halmaza-e a `t` halmaznak.
`s.pop()`	Az `s` halmaz egy véletlenszerűen kiválasztott elemével tér vissza, az elemet a halmazból eltávolítja; ha az `s` halmaz üres, `KeyError` kivétel keletkezik.
`s.remove(x)`	Eltávolítja az `x` elemet az `s` halmazból, ha benne van, különben `KeyError` kivételt vált ki; lásd még a `set.discard()` tagfüggvényt.
`s.symmetric_difference(t)` `s ^ t`	Új halmazzal tér vissza, amelyben az `s` és a `t` halmaz minden eleme benne van, kivéve a mindkét halmazban megtalálható elemeket.
`s.symmetric_difference_update(t)` `s ^= t`	Az `s` halmaznak a saját maga és a `t` halmaz szimmetrikus differenciáját adja új értékül.
`s.union(t)` `s | t`	Új halmazzal tér vissza, amelyben az `s` halmaz minden eleme benne van, illetve a `t` halmaz olyan elemei, amelyek az `s` halmaznak nem elemei.
`s.update(t)` `s |= t`	Minden olyan elemet behelyez az `s` halmazba, amelyek a `t` halmaznak elemei, de az `s` halmaznak nem.

Halmazokat a szükségtelen elemek eltávolítására is szokás használni. Példának okért, ha van egy fájlnevekből álló listánk, de szeretnénk a „makefile” nevűeket eltávolítani belőle (mondjuk azért, mert azokat nem mi írtuk, hanem valamelyik parancsfájlunk hozza őket létre), akkor ilyen kódot írhatunk:

```
filenames = set(filenames)
for makefile in {"MAKEFILE", "Makefile", "makefile"}:
    filenames.discard(makefile)
```

* Ez a tagfüggvény és a művelete (ha van neki) a fagyasztott halmaz típusú objektumok esetében is használható.

Ez a kód eltávolít a listából minden elterjedt írásmódú, „makefile" nevű fájlt. Ha a listában nincs ilyen nevű fájl, semmit nem tesz. Azonos eredmény egy sorban is elérhető, ha a halmazok különbségét képező műveletet (-) használjuk:

```
filenames = set(filenames) - {"MAKEFILE", "Makefile", "makefile"}
```

Az elemek eltávolítására a `set.remove()` tagfüggvény is használható, ám a tagfüggvény olyan esetekben, amikor a kért elem nincs a halmazban, `KeyError` kivételt vált ki.

Halmazértelmezések

Halmazok létrehozására a `set()` függvény meghívásán, illetve a halmazliterálokon kívül a *halmazértelmezések* (set comprehension) is használhatók. A halmazértelmezések egy kifejezésből, illetve egy elhagyható feltételt tartalmazó ciklusból állnak; a halmazértelmezést kapcsos zárójelek közé írjuk. A listaértelmezésekhez hasonlóan ezúttal is kétféle utasításforma létezik:

```
{kifejezés for elem in bejárható_objektum}
{kifejezés for elem in bejárható_objektum if feltétel}
```

A fenti utasításformák használatával szűrést végezhetünk (már ha a sorrend nem számít). Íme egy példa:

```
html = {x for x in files if x.lower().endswith((".htm", ".html"))}
```

Amennyiben a `files` objektum fájlnevek listáját tartalmazza, a fenti halmazértelmezés a `html` nevű halmaznak úgy ad értéket, hogy a halmazba csak a `.htm`, illetve a `.html` végű fájlnevek kerüljenek; a művelet nem veszi figyelembe a kis-, illetve a nagybetűk használatát.

A listaértelmezésekhez hasonlóan a halmazértelmezésekben használt bejárható objektum maga is lehet halmazértelmezés – vagy bármilyen egyéb értelmezés –, vagyis lehetőségünk van igen kifinomult halmazértelmezések létrehozására.

Fagyasztott halmazok

A fagyasztott halmaz olyan halmaz, amely a létrehozása után nem módosítható. Természetesen arra azért lehetőségünk van, hogy a fagyasztott halmazra hivatkozó változónak más objektumot adjunk hivatkozási célul. A fagyasztott halmazok létrehozásának egyetlen módja, ha a `frozenset` adattípust függvényként hívjuk meg. A `frozenset()` függvény paraméterek nélküli híváskor üres fagyasztott listával tér vissza; ha `frozenset` típusú objektumot adunk át paraméterül, akkor a paraméter sekély másolatát kapjuk; bármilyen egyéb típusú objektum esetében pedig kísérlet történik a paraméter fagyasztott listává történő alakítására. A függvény legfeljebb egy paramétert kezel.

Minthogy a fagyasztott halmaz típusú objektumok állandók, csak azokat a tagfüggvényeket és műveleteket használhatjuk velük, amelyek a szóban forgó fagyasztott halmaz, illetve halmazok megváltoztatása nélkül

Sekély és mély másolás ☞ 143

adnak eredményt. A 3.2. táblázatban a halmazok minden tagfüggvényét bemutattuk; a fagyasztott halmazokkal az alábbiakat használhatjuk: `frozenset.copy()`, `frozenset.difference()` (-), `frozenset.intersection()` (&), `frozenset.isdisjoint()`, `frozenset.issubset()` (<=; illetve < a valódi részhalmazokhoz), `frozenset.issuperset()` (>=; illetve > a valódi bővített halmazokhoz), `frozenset.union()` (|) és `frozenset.symmetric_difference()` (^). Az itt feltüntetett tagfüggvényeket a táblázatban csillag jelzi.

Ha egy kéttényezős művelet egyik tényezője halmaz, a másik pedig fagyasztott halmaz, az eredmény adattípusa a bal oldali tényező adattípusával fog megegyezni. Vagyis ha `f` fagyasztott halmazt, `s` pedig halmazt jelöl, akkor az `f & s` művelet eredménye egy fagyasztott halmaz, az `s & f` műveleté pedig egy halmaz lesz. A `==` és a `!=` műveletek esetében a tényezők sorrendje nem számít, és az `f == s` művelet `True` értéket ad eredményül, ha mindkét halmazban ugyanazok az elemek vannak.

A fagyasztott halmaz típusú objektumok állandóságának további következménye, hogy megfelelnek a halmazok elemeivel szemben támasztott kivonatolhatósági követelménynek, így a halmazok és a fagyasztott halmazok elemei lehetnek fagyasztott halmazok.

A halmazok használatára a fejezet következő, illetve a *Példák* című részében további példákat láthatunk.

Megfeleltetési típusok

A *megfeleltetési típusok* támogatják a tagsági műveletet (`in`), vizsgálható a méretük (`len()`), valamint bejárhatók. A megfeleltetések kulcs-érték párokból álló gyűjtemények, amelyeknek léteznek tagfüggvényeik az elemek, az elemek kulcsának és az elemek értékének elérésére. Bejáráskor az elemek sorrendje megjósolhatatlan. A Python két megfeleltetési típussal rendelkezik, a beépített `dict` (szótár) típussal, illetve az alapkönyvtár `collections.defaultdict` típusával. A *szótár* kifejezést olyan esetekben, amikor a különbség nem számít, úgy használjuk, hogy az említett két típus bármelyike érthető alatta.

Egy szótár kulcsa csak kivonatolható objektum lehet, tehát egy szótár kulcsaként az állandó értékű típusok – például `float`, `frozenset`, `int`, `str` és `tuple` – használhatók, a változtatható típusok – például `dict`, `list` és `set` – nem. Másfelől, az egyes kulcsokhoz tartozó értékek lehetnek bármilyen típusú objektumra hivatkozó objektumhivatkozások, ideértve a számokat, karakterláncokat, listákat, halmazokat, szótárakat, függvényeket és így tovább.

Kivonatolható objektumok 115 ☜

A szótárak összehasonlítására a szokásos összehasonlító műveletekkel (<, <=, ==, !=, >=, >) nyílik lehetőség, mégpedig elemenkénti összehasonlítással (egymásba ágyazott elemek, például szótáron belüli

rendezett sorozatok, illetve szótárak esetében pedig önhívó összehasonlítással). Nyilvánvaló okok miatt a szótárak esetében csak az == és a != összehasonlító műveletek elvégzésének van értelme.

Szótárak

A `dict` típus nulla vagy több kulcs-érték pár rendezetlen gyűjteménye, ahol a kulcsok kivonatolható objektumra mutató objektumhivatkozások, az értékekként tárolt objektumhivatkozások viszont bármilyen típusú objektumra hivatkozhatnak. A szótár típusú objektumok változtathatók, vagyis egyszerűen megvalósítható új elem felvétele vagy egy már meglévő elem törlése. Minthogy a szótárak rendezetlenek, az elemek helyét nem jelöli sorszám, és nincs mód sem szeletelésre, sem egyelésre.

Lehetőség van a `dict` adattípus függvényként való meghívására is: `dict()`. A paraméterek nélküli hívás visszatérési értéke egy üres szótár, amennyiben pedig az átadott paraméter egy megfeleltetési típusba tartozó objektum, a paraméteren alapuló szótárat kapunk vissza – szótár paraméter esetében például a szótár sekély másolatát. Lehetséges sorozat típusú paraméter használata is, amennyiben a sor minden tagja maga is kételemű sorozat. Ilyenkor a kételemű sorozat első értéke lesz a kulcs, a második az érték. Ezen felül, olyan szótárak esetében, ahol a kulcsok a Python nyelvben megengedett azonosítók, kulcsszó-paraméterek is használhatók: a kulcsszó lesz a kulcs, a kulcsszó értéke pedig az érték. A szótárak kapcsos zárójel használatával is létrehozhatók – az üres kapcsos zárójel, a {} üres szótárat hoz létre. Egyéb esetekben a kapcsos zárójelek között vesszővel elválasztott elemeknek kell lenniük – legalább egynek –, amelyek mindegyike egy kulcsból, az elem részét képező kettőspontból, illetve egy értékből áll. Szótárak létrehozására további módszert a szótárértelmezések használata kínál – a témakört az alfejezet későbbi részében tárgyaljuk.

Sekély és mély másolás ☞ 143

Kulcsszó-paraméterek ☞ 170

Szótárértelmezések ☞ 130

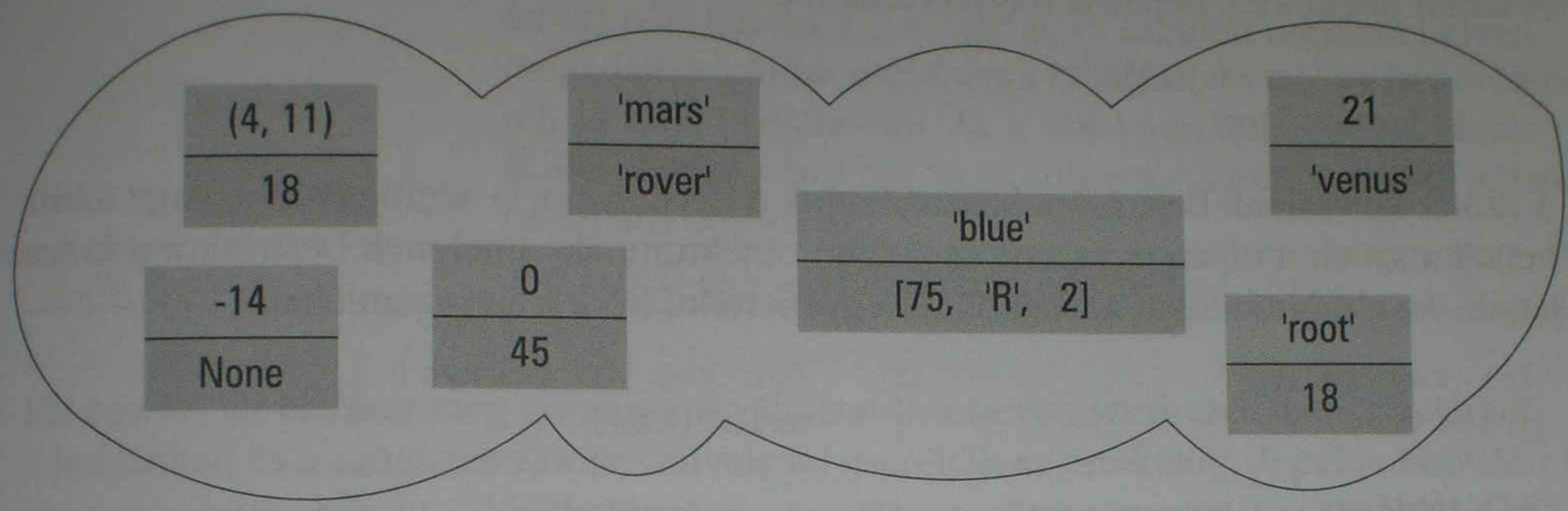
Íme néhány példa a különféle utasításformák szemléltetésére – mindegyik ugyanazt a szótárat hozza létre:

```
d1 = dict({"id": 1948, "name": "Washer", "size": 3})
d2 = dict(id=1948, name="Washer", size=3)
d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])
d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3)))
d5 = {"id": 1948, "name": "Washer", "size": 3}
```

A `d1` szótár értékét szótárliterállal adjuk meg. A `d2` szótár esetében kulcsszó-paramétereket használunk. A `d3` és `d4` szótárat sorozatokból hozzuk létre, a `d5` szótárat ismét literálként adjuk meg. A `d4` szótár létrehozásakor használt `zip()` beépített függvény olyan rendezett sorozatokból álló listát ad vissza, ahol az első rendezett sorozat tartalmazza a `zip()` függvény bejárható paramétereinek első

zip() ☞ 138

elemeit, a második a másodikakat, és így tovább. A kulcsszó-paraméterek esetében – a `d2` szótárnál – használt utasításforma rendszerint a legrövidebb és a legkényelmesebb, amennyiben a kulcsok a Python nyelvben megengedett azonosítók.



3.5. ábra

A szótár egyedi kulcsokkal rendelkező kulcs-érték párok rendezetlen gyűjteménye

A 3.5. ábra az alábbi kódrészlettel létrehozott szótárat szemlélteti:

```
d = {"root": 18, "blue": [75, "R", 2], 21: "venus", -14: None,
     "mars": "rover", (4, 11): 18, 0: 45}
```

A szótárak kulcsai egyediek, vagyis amikor egy olyan kulcs–érték párból álló elemet veszünk fel a szótárba, amelyiknek a kulcsával már van megegyező kulcsunk, a kulcs párjaként tárolt érték cseréje fog megtörténni. Az egyes értékek elérésére szögletes zárójelet használunk – például a `d["root"]` kifejezés visszatérési értéke `18`, a `d[21]` kifejezésé a `"venus"` karakterlánc, a `d[91]` kifejezés pedig `KeyError` kivételt vált ki, amennyiben a 3.5. ábrán bemutatott szótárról van szó.

A szótáraknál szögletes zárójelet használunk új elemek felvételére, illetve a meglévők eltávolítására. Elem felvételéhez az `=` műveletet használjuk, például `d["X"] = 59`. Elemek törlésére a `del` utasítás való – a `del d["mars"]` kifejezés például a „mars” kulcsú elemet törli a szótárból, illetve `KeyError` kivételt vált ki, amennyiben nincs ilyen kulcsú elem. Elemek szótárból való eltávolítására (és függvény visszatérési értékekénti beolvasására) a `dict.pop()` tagfüggvény használható.

A szótárak támogatják a `len()` beépített függvény használatát, kulcsaik esetében az `in` és a `not in` tagsági viszonyt vizsgáló műveletet is. A szótár típusú objektumok valamennyi tagfüggvénye szerepel a 3.3. táblázatban.

Minthogy a szótárak kulcsokkal és értékekkel egyaránt bírnak, kulcs-érték párok alapján éppúgy bejárhatók, mint kulcsonként vagy értékenként. Példának okáért íme két egyenértékű megközelítés a kulcs-érték párok alapján végrehajtott bejárásra:

```
for item in d.items():
print(item[0], item[1])
```

```
for key, value in d.items():
print(key, value)
```

A szótár értékeinek bejárása nagyon hasonló:

```
for value in d.values():
   print(value)
```

A szótár kulcsainak bejárására használhatjuk a `dict.keys()` tagfüggvényt, vagy tekinthetjük a szótárat olyan egyszerű, bejárható objektumnak, amelynek bejáráskor a kulcsait látjuk. Az elmondottakat az alábbi két egyenértékű kódrészlet szemlélteti:

```
for key in d:
print(key)
```

```
for key in d.keys():
print(key)
```

3.2. táblázat *A halmaz adattípus tagfüggvényei és műveletei*

Utasításforma	Leírás
`d.clear()`	A `d` szótár valamennyi elemét törli.
`d.copy()`	A `d` szótár sekély másolatát adja vissza.
`d.fromkeys(s, v)`	Olyan szótár típusú objektumot ad vissza, amelynek kulcsai az `s` sorból, az értékei pedig a `v` sorból származnak. Ha `v` nem adott, az értékek helyére `None` kerül.
`d.get(k)`	A `k` kulcshoz rendelt értéket adja vissza, illetve `None` értéket, ha a `d` szótárnak nincs `k` kulcsa.
`d.get(k, v)`	A `k` kulcshoz rendelt értéket adja vissza, illetve `v` értéket, ha a `d` szótárnak nincs `k` kulcsa.
`d.items()`	A `d` szótár (kulcs, érték) párjait tartalmazó nézetet* ad vissza.
`d.keys()`	A `d` szótár kulcsait tartalmazó nézetet* ad vissza.
`d.pop(k)`	A `k` kulcshoz rendelt értéket adja vissza, majd eltávolítja a `k` kulcsú értéket. Ha a `d` szótárban nincs `k` kulcs, `KeyError` kivételt vált ki.
`d.pop(k, v)`	A `k` kulcshoz rendelt értéket adja vissza, majd eltávolítja a `k` kulcsú értéket. Ha a `d` szótárban nincs `k` kulcs, `v` értéket ad vissza.
`d.popitem()`	A `d` szótárból véletlenszerű (kulcs, érték) párt ad vissza. Ha a `d` szótár üres, `KeyError` kivételt vált ki.
`d.setdefault(k, v)`	Ugyanaz, mint a `dict.get()` tagfüggvény, kivéve, hogy ha a kulcs nem található meg a `d` szótárban, új elemet vesz fel `k` kulccsal és `None` értékkel, illetve `v` értékkel, ha `v` adott.

Sekély és mély másolás ☞ 143

3.2. táblázat *A halmaz adattípus tagfüggvényei és műveletei* *(folytatás)*

Utasításforma	Leírás
`d.update(a)`	Minden `a`-ban szereplő, de a `d` szótárban nem szereplő (kulcs, érték) párt felvesz a `d` szótárba, és minden mindkét helyen megtalálható kulcs esetében a `d` szótárban tárolt értéket lecseréli az `a`-ban találhatóra. Az `a` objektum lehet szótár, illetve (kulcs, érték) párokból vagy kulcsszó-paraméterekből képzett bejárható objektum.
`d.values()`	A `d` szótár elemeit tartalmazó nézetet* ad vissza.

Ha egy szótár értékeit meg szeretnénk változtatni, a bevett szerkezet szerint bejárjuk a kulcsokat, és a szögletes zárójel művelettel cseréljük az értékeket. A példa kedvéért bemutatjuk, hogy miként valósítható meg a `d` szótár minden egyes értékének eggyel való növelése – feltételezve, hogy minden érték szám:

```
for key in d:
    d[key] += 1
```

A `dict.items()`, a `dict.keys()`, és a `dict.values()` tagfüggvény egyaránt *szótárnézetet* ad vissza. A szótárnézet gyakorlatilag olyan, csak olvasható bejárható objektumnak tekinthető, amely látszólag a szótár elemeit, illetve a szótár kulcsait vagy értékeit tartalmazza – attól függően, hogy mire utasítottuk a gépet.

Általában kezelhetjük a nézeteket egyszerűen úgy, mint bejárható objektumokat. Az átlagos bejárható objektumoktól azonban két dolog különbözteti meg a nézeteket. Az egyik az, hogy ha a nézet alapjául szolgáló szótár megváltozik, a nézet tükrözi a változásokat. A másik pedig az, hogy a kulcs- és az elemnézetek támogatnak néhány, a halmazokra emlékeztető műveletet. Ha adott `v` szótárnézet és adott `x`, ami halmaz és szótárnézet egyaránt lehet, az alábbi műveletek hajthatók végre:

```
v & x   # Metszet
v | x   # Unió
v - x   # Különbség
v ^ x   # Szimmetrikus differencia
```

Annak megállapítására, hogy a szótárban megtalálható-e egy bizonyos kulcs, az `in` tagsági művelet használható, például: `x in d`. Az unióképzés műveletét pedig annak megállapítására használhatjuk, hogy egy adott halmazból mely kulcsok találhatók meg egy szótárban. Például:

```
d = {}.fromkeys("ABCD", 3)    # d == {'A': 3, 'B': 3, 'C': 3, 'D': 3}
s = set("ACX")                # s == {'A', 'C', 'X'}
matches = d.keys() & s        # matches == {'A', 'C'}
```

* A szótárnézetekre úgy gondolhatunk (és úgy is használhatjuk őket), mintha bejárható objektumok lennének (a szövegben hamarosan tárgyaljuk őket).

Figyeljük meg, hogy a kódrészlet megjegyzéseiben az elemek ábécésorrendben vannak – ez azonban pusztán az olvashatóság megkönnyítése miatt van így, lévén mind a szótárak, mind a halmazok rendezetlenek.

A szótárakat gyakran használjuk egyedi elemek számának nyilvántartására. Egy lehetséges példa, amikor egy fájlban előforduló különféle szavak előfordulásait számoljuk meg. Íme egy kész program (`uniquewords1.py`), amely kiírja a parancssorban felsorolt fájlokban lévő egyes szavakat, illetve az előfordulásaik számát:

```
import string
import sys

words = {}
strip = string.whitespace + string.punctuation + string.digits +
"\"'"
for filename in sys.argv[1:]:
    for line in open(filename):
        for word in line.lower().split():
            word = word.strip(strip)
            if len(word) > 2:
                words[word] = words.get(word, 0) + 1
for word in sorted(words):
    print("'{0}' occurs {1} times".format(word, words[word]))
```

A program egy üres, `words` nevű szótár létrehozásával kezdődik. Ez után a `string` modul által rendelkezésre bocsátott hasznos karakterláncok összefűzésével létrehozunk egy olyan újabb karakterláncot, amely a figyelmen kívül hagyandó karaktereket tartalmazza. Egyesével végighaladunk a parancssorban megadott fájlneveken. Az `open()` függvényről a *Szövegfájlok olvasása és írása* című részben olvashatunk. Karakterkódolást nem adunk meg – merthogy nem tudjuk, hogy mi lesz az egyes fájlok karakterkódolása –, hanem úgy intézzük, hogy a Python az éppen alapértelmezett karakterkódolás használatával nyissa meg az egyes fájlokat. A sorokat kisbetűssé alakítjuk, majd szavakra bontjuk, és a szavak végeiről eltávolítjuk azokat a karaktereket, amelyeket figyelmen kívül akarunk hagyni. Ha az így kapott szó legalább három karakter hosszú, frissítjük a szótárat.

Szövegfájlok olvasása és írása

A fájlok megnyitására az `open()` beépített függvényt használjuk, amely egy „fájl objektummal”, azaz szövegfájlok esetében egy `io.TextIOWrapper` típusú objektummal tér vissza. Az `open()` függvénynek egy kötelező paramétere van: a fájlnév, amely tartalmazhatja az elérési utat. A függvény ezen felül hat elhagyható paraméterrel is bír; az alábbiakban ezek közül kettő rövid áttekintésére vállalkozunk. A második paraméter, a *mód*, annak megadására való, hogy az állományt szövegfájlként vagy bináris fájlként kezelje a program, illetve hogy olvasás, írás, vagy hozzáfűzés végett szeretnénk-e megnyitni a fájlt, netán az előző műveletek valamilyen kombinációjára kell felkészülni.

7. fejezet: *Fájlkezelés* ☞ 287

Karakterkódolások
☞ 86

A szövegfájlok esetében a Python az operációs rendszertől függő karakterkódolást használ. Amikor csak lehetséges, a legokosabb az `open()` függvény karakterkódolást szabályozó paraméterét is megadnunk, vagyis a fájlok megnyitására használatos leggyakoribb utasításformák az alábbiak:

```
fin = open(filename, encoding="utf8")          # szöveg olvasásakor
fout = open(filename, "w", encoding="utf8")    # szöveg írásakor
```

Minthogy az `open()` függvény a *mód* paraméter alapértelmezett beállításával a szöveg olvasásra való megnyitását végzi el, és mert a kódolást kulcsszóval azonosított, nem pedig hely szerinti paraméterátadással állítottuk be, a fájlok olvasásra történő megnyitásakor a további paramétereket elhagyhatjuk. Ehhez hasonlóan az írásra való megnyitáskor is elég azokat a paramétereket megadnunk, amelyeket ténylegesen használni szeretnénk. (A paraméterátadással részletesen a 4. fejezet foglalkozik.)

Amikor a fájlt olvasásra, szöveges módban megnyitottuk, a fájl objektum `read()` tagfüggvényével a teljes állomány beolvasható egyetlen karakterláncba. Karakterláncokból álló listába a `readlines()` tagfüggvény olvassa be a fájl sorait. A soronkénti beolvasást általában a következő, igen elterjedt szerkezettel végezzük, amelyben a fájl objektumot bejárható adatként kezeljük:

```
for sor in open(filename, encoding="utf8"):
    feldolgozás(sor)
```

A szerkezet azért működik, mert a fájl, akár egy sorozat, bejárható, és a fájl egyes sorai jelentik a bejárás egymásra következő elemeit. A visszakapott sorok végén szerepel a soremelést végző „`\n`” karakter.

Ha „w” módot adunk meg, a fájl szöveges írható állományként nyílik meg. Fájlokba a fájl objektum `write()` tagfüggvényével szokás írni. A tagfüggvény egyetlen paramétere egy karakterlánc. Minden sornak tartalmaznia kell a „`\n`” karaktert. A Python a „`\n`” karakter helyére automatikusan a gazdarendszer sorainak végén szereplő karaktert, illetve karaktereket helyettesíti.

Amikor a fájl objektum használatát befejeztük, meghívhatjuk az objektum `close()` tagfüggvényét – a hívás hatására a gyorsítótárban lévő írási műveletek valóban megtörténnek. A kis Python-programokban általánosan elterjedt gyakorlat a `close()` tagfüggvény hívásának elhagyása – a Python ugyanis automatikusan elvégzi a hívással kapcsolatos teendőket, mihelyt a fájl objektum kikerül a hatókörből. Probléma felmerülése esetén kivétel keletkezik.

A `words[word] += 1` formát nem használhatjuk, mert így minden olyan alkalommal, amikor új szóra lelünk, `KeyError` kivétel keletkezik – természetesen nem lehetséges az olyan elemek értékének növelése, amelyek még nem is léteznek a szótárban. Így aztán némileg eltérő módszerrel közelítjük meg a problémát. A `dict.get()` tagfüggvényt 0 alapértelme-

zett értékkel hívjuk meg. Ha a szó már benne van a szótárban, a `dict.get()` tagfüggvény megadja a hozzá tartozó számot, és ennél eggyel nagyobb számot állítunk be új értékként. Ha a szó nincs a szótárban, a `dict.get()` tagfüggvény a megadott 0 értékkel tér vissza, és az ennél eggyel nagyobb számot (azaz az 1-et) állítjuk be a `word` karakterláncban tárolt kulcsú elem új értékeként. Hogy minden világos legyen, íme két kódrészlet, amelyek működése megegyezik, ámbár a `dict.get()` tagfüggvényt használó hatékonyabb:

```
words[word] = words.get(word, 0) + 1
```

```
if word not in words:
    words[word] = 0
words[word] += 1
```

A következő alfejezetben, ahol az alapértelmező szótárakról lesz szó, újabb megoldást keresünk a problémára.

Ha a szavakat tartalmazó szótárat feltöltöttük, a kulcsok (a szavak) rendezett bejárásával kiírjuk az egyes szavakat, és az előfordulásaik számát.

A `dict.get()` tagfüggvény használata egyszerű elemek, például a számok és a karakterláncok esetében lehetővé teszi a szótárak egyszerű frissítését. De mi a helyzet olyankor, amikor az egyes értékek maguk is gyűjtemények? Az ilyen helyzetek kezelését egy program segítségével szemléltetjük. A program beolvassa a parancssorban megadott, HTML nyelvű fájlokat, majd kiír egy listát, ami azokat a webhelyeket tartalmazza, amelyekre a fájlok valamelyike hivatkozik, majd a webhely nevéhez képest behúzva kiírja azoknak a fájloknak a nevét, amelyek az adott webhelyre hivatkoznak. Szerkezetét tekintve az `external_sites.py` program nagyon hasonlít a szavakat számláló, az előbbiekben megvizsgált programhoz. Íme a programkód fő része:

```
sites = {}
for filename in sys.argv[1:]:
    for line in open(filename):
        i = 0
        while True:
            site = None
            i = line.find("http://", i)
            if i > -1:
                i += len("http://")
                for j in range(i, len(line)):
                    if not (line[j].isalnum() or line[j] in ".-"):
                        site = line[i:j].lower()
                        break
                if site and "." in site:
                    sites.setdefault(site, set()).add(filename)
                i = j
            else:
                break
```

Először egy üres szótárat hozunk lére. Ez után egyesével végighaladunk a parancssorban megadott fájlokon, illetve azok egyes sorain. Figyelembe kell vennünk annak lehetőségét, hogy egyetlen sor akárhány webhelyre is hivatkozhat – ezért addig futtatjuk a `str.find()` tagfüggvényt, amíg hibával nem tér vissza. Ha felleljük a „http://" karakterláncot, a karakterlánc hosszának megfelelő értékkel növeljük az indulási helyet jelző `i` változó értékét. Ez után megvizsgáljuk a következő karaktereket, egészen addig, amíg olyat nem találunk, ami nem fordulhat elő webhely nevében. Amikor megtalálunk egy webhelyet, egyszerű hibaellenőrzésként megnézzük, hogy van-e pont a névben, és amennyiben igen, felvesszük a szótárba.

A `sites[site].add(filename)` utasításformát nem használhatjuk, mert ez `KeyError` kivételt váltana ki minden olyan alkalommal, amikor új webhelyre lelünk – természetesen nem adhatunk új elemet olyan halmazhoz – vagyis a szótár elemének értékéhez –, ami még nem is létezik a szótárban. Így a problémát más oldalról kell megközelítenünk. A `dict.setdefault()` tagfüggvény a szótárnak az adott kulcsú (az első paraméterben megadott) elemére hivatkozó objektumhivatkozással tér vissza. Ha ilyen elem nincs, a tagfüggvény az adott kulccsal új elemet hoz létre, amelynek értékéül vagy a `None` objektumot, vagy a második paraméterként átadott alapértelmezett értéket adja meg. Ebben az esetben az alapértelmezett értékként a `set()` függvényt, vagyis egy új halmazt adunk meg. Így a `dict.setdefault()` tagfüggvény meghívása mindig egy értékre hivatkozó objektumhivatkozással tér vissza, vagy egy olyannal, ami már korábban is létezett, vagy egy újjal. (Természetesen, ha az adott kulcs nem kivonatolható, `TypeError` kivétel lép fel.)

Ebben a példában a visszaadott objektumhivatkozás mindig egy halmazra hivatkozik (amikor az egyes kulcsok, vagyis webhelyek először kerülnek elő, a halmaz még üres), ez után pedig a webhelyre hivatkozó fájl nevét hozzáadjuk a webhelyre hivatkozó fájlneveket tároló halmazhoz. A halmaz használatával biztosítjuk, hogy csak egyszer jegyezzük fel a fájl nevét olyan esetekben is, amikor egy fájl több alkalommal hivatkozik ugyanarra a webhelyre.

A `dict.setdefault()` tagfüggvény működését jobban megérthetjük, ha megnézzük a következő két egyenértékű kódrészletet:

```
sites.setdefault(site, set()).add(fname)
```

```
if site not in sites:
    sites[site] = set()
sites[site].add(fname)
```

A teljesség kedvéért íme a program további része:

```
for site in sorted(sites):
    print("{0} is referred to in:".format(site))
    for filename in sorted(sites[site], key=str.lower):
        print("    {0}".format(filename))
```

sorted()
☞ 139, 178

Az egyes webhelyek alatt behúzással kiírva megtaláljuk az adott webhelyre hivatkozó fájlok neveit. A külső `for...in` ciklusban lévő `sorted()` függvény meghívásával rendezzük a szótár kulcsait – minden olyan helyzetben, ahol a szótárnak bejárható objektumként kell szerepelnie, a kulcsok bejárása történik meg. Ha azt szeretnénk, hogy a (kulcs, érték) párokat vagy az értékeket járja be a program, a `dict.items()`, illetve a `dict.values()` tagfüggvényt kell használnunk. A belső `for...in` ciklus járja be az aktuális webhely fájlneveinek rendezett halmazát.

Szótárértelmezések

A *szótárértelmezések* egy kifejezésből, illetve egy elhagyható feltételt tartalmazó ciklusból állnak. A szótárértelmezést kapcsos zárójelek közé írjuk. A lista- és a halmazértelmezésekhez hasonlóan ezúttal is kétféle utasításforma létezik:

```
{kulcskifejezés: értékkifejezés for kulcs, érték in bejárható_objektum}
{kulcskifejezés: értékkifejezés for kulcs, érték in bejárható_objektum
                                                    if feltétel}
```

A következő kódsor bemutatja, hogy szótárértelmezéssel miként állítható elő olyan szótár, ahol az egyes kulcsok az aktuális könyvtárban lévő fájlok nevei, az értékek pedig az adott fájl bájtban mért hosszának felelnek meg:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir(".")}
```

os és os.path modul
☞ 219

Az `os` („operating system”, vagyis operációs rendszer) modul `os.listdir()` tagfüggvénye a neki átadott elérési úton található fájlok és könyvtárak listájával tér vissza – a lista soha nem tartalmazza a „.” és a „..” könyvtárt.
Az `os.path.getsize()` tagfüggvény a neki átadott fájl bájtban vett méretét adja vissza. Egy feltétel hozzáadásával biztosítható, hogy csak a fájlok méretét kapjuk vissza:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir(".")
              if os.path.isfile(name)}
```

Az `os` modul `os.path.isfile()` tagfüggvénye `True` értékkel tér vissza, ha az átadott elérési út fájlt jelöl, egyébként (könyvtárak, hivatkozások és társaik esetében) pedig `False` értéket kapunk.

Szótárértelmezés használható fordított szótár létrehozására is. Ha például adott a `d` szótár, előállíthatunk egy olyan új szótárat, amelynek a `d` szótár értékei lesznek a kulcsai, és a `d` szótár kulcsai lesznek az értékei:

```
inverted_d = {v: k for k, v in d.items()}
```

Az új szótár visszaalakítható az eredetivé, ha az eredeti szótár értékei mind egyediek voltak – a megfordítási művelet `TypeError` kivételt eredményez, ha az értékek között olyanra lel, ami nem kivonatolható.

A lista- és a halmazértelmezésekkel megegyező módon a szótárértelmezés bejárható objektuma is lehet egy másik értelmezés, vagyis az egymásba ágyazott értelmezések mindenféle variációja elképzelhető.

Alapértelmező szótárak

Az alapértelmező szótárak is szótárak – rendelkeznek a szótárak valamennyi műveletével és tagfüggvényével.* Ami az alapértelmező szótárakat a mezei szótáraktól megkülönbözteti, az annak a módja, ahogy a hiányzó kulcsokat kezelik – minden más esetben a szótárakkal azonos módon viselkednek.

Ha egy szótár nem létező („hiányzó") kulcsára hivatkozunk, `KeyError` kivétel lép fel. Ez azért hasznos, mert sokszor szeretnénk tudni, hogy a keresett kulcs létezik-e vagy sem. Vannak azonban olyan esetek, amikor azt szeretnénk, ha minden keresett kulcs létezne, akár annak árán is, hogy az első olyan esetben, amikor keressük, fel kell vennünk a szótárba.

Vegyünk például egy olyan `d` szótárat, amelyben *nincs* `m` kulcsú elem, vagyis az `x = d[m]` kód `KeyError` kivételt vált ki. Ha a `d` szótárat a létrehozásakor kicsit másképp, alapértelmező szótárként alkotjuk meg, és van benne `m` kulcsú elem, akkor az előző kifejezés visszatérési értéke ugyanaz lesz, mint ha `d` egyszerű szótár lenne. Amennyiben az `m` kulcs nem létezik az alapértelmező szótárban, új, `m` kulcsú elem jön létre valamilyen alapértelmezett értékkel, és az előző kifejezés az újonnan létrejött elem értékét adja vissza.

Nemrég írtunk egy rövid programot, amely a parancssorban megadott fájlokban lévő különféle szavakat számolta meg. A szavakat tartalmazó szótárat így hoztuk létre:

```
words = {}
```

A `words` szótár minden egyes kulcsa egy szó volt, a megfelelő érték pedig egy egész, ami azt tárolta, hogy az adott szó hányszor fordult elő a beolvasott fájlokban összesen. A következő módszerrel növeltük az értéket egy-egy megfelelő szó megtalálásakor:

```
words[word] = words.get(word, 0) + 1
```

A `dict.get()` tagfüggvényt kellett minden olyan alkalommal használnunk, amikor egy szót először találtunk meg – ilyenkor új elemet hoztunk létre, 1 értékkel –, és olyankor is, amikor az adott szóból további példányra leltünk – ilyenkor a megtalálások számát eggyel növeltük.

Az alapértelmező szótár létrehozásakor lehetőségünk van *gyártófüggvény* átadására. A gyártófüggvény olyan függvény, amely a meghívásakor egy adott típusú objektumot ad vissza. A Python nyelv valamennyi beépített adattípusa használható gyártófüggvényként,

---

* Megjegyzés objektumközpontú programozóknak: a `defaultdict` a `dict` alosztálya.

például a `str` adattípust hívhatjuk, mint függvényt: `str()`. Ha nem adtunk át paramétert, akkor új karakterlánc típusú objektummal tér vissza. Az alapértelmező szótárnak átadott gyártófüggvény arra való, hogy a hiányzó kulcsokhoz tartozó értékeket előállítsa.

Ne feledjük, hogy a függvény *neve* a függvényre hivatkozó objektumhivatkozás, vagyis amikor függvényt kívánunk paraméterként átadni, elég a függvény nevét átadnunk. A Python onnan tudja, hogy a függvényt meg kell hívnia, hogy a nevet zárójelek követik.

A `uniquewords2.py` nevű program egyetlen sorral (`import collections`) hosszabb az eredeti, `uniquewords1.py` nevűnél, ezen felül a szótárat létrehozó és frissítő sorokban van eltérés. Az alapértelmező szótárat a következőképp állítjuk elő:

```
words = collections.defaultdict(int)
```

A `words` nevű alapértelmező szótár soha nem fog `KeyError` kivételt kiváltani. Ha kiadjuk az `x = words["xyz"]` utasítást, és az utasítás kiadásának pillanatában nincs `"xyz"` kulcsú elem, és így nem találjuk meg a megfelelő kulcsot, az alapértelmező szótár azonnal felvesz egy új, `"xyz"` kulcsú elemet, majd az `int()` függvényt meghívva ellátja a 0 értékkel, amit aztán az `x` változó is felvesz.

```
words[word] += 1
```

Így immáron nem kell a `dict.get()` tagfüggvényt meghívnunk; helyette egyszerűen növeljük az elem értékét. Amikor egy szót először lelünk fel, új, 0 (de azonnal 1-gyel növelt) értékű elem jön létre, minden további alkalommal pedig 1-gyel növeljük az értéket, akármi legyen is az.

Befejeztük a Python nyelv beépített gyűjteményes adattípusainak áttekintését, és áttekintettünk párat az alapkönyvtár gyűjteményes adattípusai közül is. A következőkben pár, a gyűjteményes adattípusokkal kapcsolatban rendszeresen felmerülő problémát veszünk szemügyre.

Gyűjtemények bejárása és másolása

Ha már elkészült az adatelemeket tároló gyűjtemény, természetesen be akarjuk járni a benne lévő valamennyi elemet. Ebben a részben először a Python nyelv néhány bejárható objektumát, a velük kapcsolatos műveleteket és függvényeket mutatjuk be.

Szintén általános az igény a gyűjtemények másolására. Van pár ezzel kapcsolatos apróság, ami abból ered, hogy a Python nyelv a hatékonyság érdekében objektumhivatkozásokkal dolgozik. A második részben ezért a gyűjtemények másolásának és a kívánt viselkedés elérésének módját mutatjuk be.

A bejárók és a bejárható adattípusok műveletei és függvényei

Bejárhatónak (iterable) nevezzük az olyan adattípusokat, amelyek egyesével vissza tudják adni az összes értéküket. Minden olyan objektum, amely rendelkezik `__iter__()` tagfüggvénnyel, illetve bármely sorozat (vagyis olyan objektum, amely rendelkezik a 0-tól induló egész számokat paraméterként elfogadni képes `__getitem__()` tagfüggvénnyel) bejárható, és belőle *bejárók* (iterator) származtathatók. A bejárók olyan objektumok, amelyek rendelkeznek `__next__()` tagfüggvénnyel, amely minden hívás alkalmával a soron következő elemet adja vissza, ha pedig azok elfogytak, `StopIteration` kivételt vált ki. A bejárható objektumok esetében használható műveleteket és függvényeket a 3.4. táblázat mutatja be.

`__iter()__`
☞ 279

Az elemek visszaadásának sorrendje az objektum alapjául szolgáló bejárható típustól függ. Listák és rendezett sorozatok esetében az elemeket rendszerint a sorszámuk által meghatározott rendben kapjuk vissza, az első – 0. sorszámú – elemtől kezdve. Néhány bejárónál a visszaadás sorrendje nem jósolható meg – ilyenek a szótár- és a halmazbejárók.

A beépített `iter()` függvénynek két meglehetősen eltérő viselkedési formája van.
Ha gyűjteményes típusú adatot vagy sorozatot adunk át neki, akkor az átadott objektumon alapuló bejárót ad vissza – illetve `TypeError` kivételt vált ki, ha az átadott objektum nem bejárható. Ezzel a felhasználással találkozunk olyankor, amikor egyedi gyűjteménytípust hozunk létre, máskor azonban igen ritkán. Az `iter()` függvény másik viselkedését akkor figyelhetjük meg, ha a függvény egyik paramétere valamilyen meghívható típusba tartozó objektum (függvény vagy tagfüggvény), a másik pedig egy célérték. Ilyenkor a bejárás során az `iter()` függvény minden visszakapott értéknél meghívja a neki átadott függvényt, és a hívott függvény által visszaadott értéket adja vissza. Ha a visszatérési érték megegyezik a célértékkel, az `iter()` függvény `StopIteration` kivételt vált ki.

Olyan alkalmakkor, amikor `for` *`elem`* `in` *`bejárható_objektum`* ciklust használunk, a Python valójában az `iter(`*`bejárható_objektum`*`)` hívással jut hozzá a bejáróhoz. Ennek a bejárónak a `__next__()` tagfüggvénye hívódik meg a ciklus minden ismétlődésekor a következő elem kiolvasásához, amikor pedig `StopIteration` kivétel keletkezik, a függvény elkapja a kivételt, és befejezi a ciklust. A bejárható objektum következő értéke a `__next__()` tagfüggvény hívásával is megkapható. Íme két egyenértékű kódrészlet (amelyek egy lista értékeit szorozzák össze) – az egyik egy `for...in` ciklust használ, a másik közvetlenül egy bejáróval dolgozik.

```
product = 1
for i in [1, 2, 4, 8]:
product *= i
print(product) # eredmény: 64
```

```
product = 1
i = iter([1, 2, 4, 8])
while True:
    try:
        product *= next(i)
    except StopIteration:
        break
print(product) # eredmény: 64
```

Bármely (véges) *i* bejárható adat rendezett sorozattá alakítható a `tuple(i)` függvény meghívásával, a listává alakítás pedig a `list(i)` függvény hívásával történik.

Az `all()` (mind) és az `any()` (bármelyik) függvények használhatók bejárókkal, és a függvények kiterjedt használatán alapuló programokban gyakran használják is őket. Íme néhány, az `all()`, az `any()`, a `len()` (hossz), a `min()`, a `max()`, és a `sum()` (összeg) függvény használatát bemutató példa:

Függvényközpontú programozás ☞ 402

```
>>> x = [-2, 9, 7, -4, 3]
>>> all(x), any(x), len(x), min(x), max(x), sum(x)
(True, True, 5, -4, 9, 13)
>>> x.append(0)
>>> all(x), any(x), len(x), min(x), max(x), sum(x)
(False, True, 6, -4, 9, 13)
```

A fenti egyszerű függvények közül valószínűleg a `len()` függvényt használják a legelterjedtebben.

Az `enumerator()` függvény paramétere egy bejáró, a visszatérési értéke pedig egy sorszámozott objektum (enumerator, felsoroló objektum). A visszaadott objektum bejáróként viselkedik, a bejáráskor visszaadott elemei pedig kételemű rendezett sorozatok. A rendezett sorozat első eleme a sorszám (alapértelmezés szerint 0-től indulóan), a második elem pedig abból a bejáróból származik, amely az `enumerate()` függvény paramétere volt. Nézzük meg az `enumerate()` függvény használatát egy apró, de teljes program példáján keresztül!

A `grepword.py` program paraméterül egy szót és egy vagy több fájlnevet vár a parancssorban, kimenetként pedig a fájl nevét, valamint annak a sornak a számát, illetve magát a sort adja, ahol a szót megtalálta.* Íme egy példafuttatás:

```
grepword.py Dom data/forenames.txt
data/forenames.txt:615:Dominykas
data/forenames.txt:1435:Dominik
data/forenames.txt:1611:Domhnall
data/forenames.txt:3314:Dominic
```

A `data/forenames.txt` és a `data/surnames.txt` adatállomány nevek rendezetlen listáit tartalmazza, minden sorban egy névvel.

A `sys` modul betöltésétől eltekintve a program mindössze tíz sor hosszú:

```
if len(sys.argv) < 3:
    print("usage: grepword.py word infile1 [infile2 [... infileN]]")
    sys.exit()
word = sys.argv[1]
```

* A 9. fejezetben a program két további megvalósítását is látni fogjuk, `grepword-p.py` és `grepword-t.py` néven, ahol a munkát több folyamat, illetve több szál között osztjuk el.

```
for filename in sys.argv[2:]:
    for lino, line in enumerate(open(filename), start=1):
        if word in line:
            print("{0}:{1}:{2:.40}".format(filename, lino,
                                           line.rstrip()))
```

3.4. táblázat *Bejárható objektumok szokványos műveletei és függvényei*

Utasításforma	Leírás
`s + t`	Az `s` és a `t` sorozatok összefűzéseként előállított sorozatot adja vissza.
`s * n`	Az `s` sorozat `n` egész számú összefűzéseként előállított sorozatot adja vissza.
`x in i`	`True` értéket ad vissza, ha az `x` tagja az `i` bejárható objektumnak; a vizsgálat megfordítására a `not in` művelet használható.
`all(i)`	`True` értéket ad vissza, ha az `i` bejárható objektumban minden érték kiértékelése `True` eredményt ad.
`any(i)`	`True` értéket ad vissza, ha az `i` bejárható objektumban bármely érték kiértékelése `True` eredményt ad.
`enumerate(i, kezdet)`	Általában `for...in` ciklusokban használva állítunk elő vele (*sorszám*, *elem*) alakú rendezett sorozatokból álló sort; a sorszámozás 0-tól vagy a *kezdet*-től indul; lásd a szöveget.
`len(x)`	Az `x` objektum „hosszát” adja vissza. Ha az `x` gyűjtemény típusú, akkor az elemek, ha az `x` karakterlánc, akkor a karakterek száma a visszatérési érték.
`max(i, kulcs)`	Az `i` bejárható objektum legnagyobb elemét adja vissza, illetve a legnagyobb *kulcs(elem)* értéket, ha a kulcs függvényt megadtuk.
`min(i, kulcs)`	Az i bejárható objektum legkisebb elemét adja vissza, illetve a legkisebb *kulcs(elem)* értéket, ha a *kulcs* függvényt megadtuk.
`range(kezdet, vég, lépés)`	Egészeket tartalmazó bejárót ad vissza. Ha egyetlen paramétert adunk meg (*vég*), akkor a bejáró 0-tól `stop -1`-ig halad; két érték esetén (*kezdet*, *vég*) az elemek a *kezdet*-nél kezdődnek és vég `-1`-ig tartanak; három paraméterrel az értékek a *kezdet*-től indulnak, vég `-1`-ig tartanak, és a *lépés* lépésközzel változnak.
`reversed(i)`	Az `i` bejáró elemeit fordított sorrendben tartalmazó bejárót ad vissza.
`sorted(i, kulcs, megfordítás)`	Az `i` bejáró elemeiből képzett rendezett listát ad vissza; a *kulcs* érték a DSU (Decorate, Sort, Undecorate) rendezést állítja be. Ha a *megfordítás* értéke `True`, fordított sorrendet kapunk.
`sum(i, kezdet)`	Az `i` bejárható objektum elemeinek összegét adja vissza, az alapértelmezés szerint 0 értékkel bíró *kezdet*-tel növelve; `i` nem tartalmazhat karakterláncokat.
`zip(i1, ..., iN)`	Rendezett sorozatokból álló bejárót ad vissza az `i1...` *iN* bejárók felhasználásával; lásd a szöveget.

Először azt ellenőrizzük, hogy van-e legalább két parancssori paraméterünk. Ha nincs, kiíratjuk a használati utasítást, majd megszakítjuk a programot. A `sys.exit()` függvény azonnali és tiszta megszakítást eredményez, miután az esetleg nyitva talált fájlokat lezárja. A neki átadott, elhagyható, egész típusú paramétert átadja a hívó héjnak.

Feltételezzük, hogy az első paraméter az a szó, amelyet a felhasználó keres, és azt is, hogy a további paraméterek azok a fájlok, amelyekben a keresést végre kell hajtani. Nem véletlenül hívjuk az `open()` függvényt a karakterkódolás megadása nélkül – a felhasználó helyettesítő karakterekkel akárhány megvizsgálandó fájlt megadhat, és ezek mindegyike eltérő karakterkódolású lehet. Így aztán engedjük, hogy a Python az operációs rendszernek megfelelő kódolást használja.

Szövegfájlok olvasása és írása 126 ☜

Az `open()` függvénytől visszakapott fájl objektum szöveges módban olyan bejáróként használható, amely egyszerre egy sornyi szöveget ad vissza. Amikor ezt a bejárót átadjuk az `enumerate()` függvénynek, olyan sorszámozott bejárót kapunk, amely az éppen bejárt sor számát a `lino` („line number", azaz sor száma) változóba írja. Ha a felhasználó által keresett szó megtalálható a sorban, kiírjuk a fájl nevét, a sor számát, és a sor első 40 karakterét, a sor végén lévő soremelés (`\n`) nélkül. Az `enumerate()` függvénynek van egy elhagyható kulcsszó-paramétere, a *`kezdet`*, amelynek az alapértelmezett értéke 0. Ezt a paramétert ebben az esetben 1 értékűre állítottuk, ugyanis a szövegfájlok sorainak számozása hagyományosan 1-től indul.

Meglehetősen gyakori, hogy nem sorszámozott objektumra van szükségünk, hanem olyan bejáróra, amelyben egymás utáni egész számokat tárolunk. A `range()` függvény éppen ilyen objektumok létrehozására való. Ha egész számokból álló listára vagy rendezett sorozatra van szükségünk, akkor a `range()` függvénytől visszakapott bejárót a megfelelő átalakítófüggvénnyel alakíthatjuk át. Íme néhány példa:

```
>>> list(range(5)), list(range(9, 14)), tuple(range(10, -11, -5))
([0, 1, 2, 3, 4], [9, 10, 11, 12, 13], (10, 5, 0, -5, -10))
```

A `range()` függvényt leginkább a következő két célra szokás használni: egész számokból álló lista vagy rendezett sorozat létrehozására, illetve a `for...in` ciklusok ciklusváltozójának előállítására. Példaként álljon itt két egyenértékű kódrészlet, amelyek arról gondoskodnak, hogy az `x` értékei pozitívak legyenek:

```
for i in range(len(x)):
    x[i] = abs(x[i])
```

```
i = 0
while i < len(x):
    x[i] = abs(x[i])
    i += 1
```

Ha az `x` lista mindkét esetben mondjuk `[11, -3, -12, 8, -1]` volt, az átalakítás után `[11, 3, 12, 8, 1]` lesz.

Minthogy a bejárható objektumok a * művelettel kicsomagolhatók, a `range()` függvénytől visszakapott objektumot is kicsomagolhatjuk. Például, ha van egy négy paramétert váró `calculate()` függvényünk, az alábbi módszerek bármelyikével átadható neki az 1, a 2, a 3 és a 4 paraméter:

```
calculate(1, 2, 3, 4)
t = (1, 2, 3, 4)
calculate(*t)
calculate(*range(1, 5))
```

Mindhárom híváskor négy paramétert adunk át. A második hívás egy négyelemű rendezett sorozatot csomagol ki, a harmadik pedig a `range()` függvénytől visszakapott bejáróval teszi ugyanezt.

Az eddig megismert dolgok közül párat az alábbi rövid, ám teljes program áttekintésével rendszerezünk. Ez lesz az első olyan alkalom is, amikor közvetlenül írunk egy fájlba. A `generate_test_names1.py` program egy keresztneveket és egy vezetékneveket tartalmazó fájlt olvas be, majd létrehozza a `test-names1.txt` nevű fájlt, és beleír száz véletlenszerűen összeállított nevet.

A `random_choice()` függvényt fogjuk használni, amely egy sorozatból véletlenszerűen választ ki egy elemet, tehát lehet, hogy pár név többször is előfordul majd. Először a nevek listáját visszaadó függvényt nézzük át, utána pedig a program többi részét.

```
def get_forenames_and_surnames():
    forenames = []
    surnames = []
    for names, filename in ((forenames, "data/forenames.txt"),
                            (surnames, "data/surnames.txt")):
        for name in open(filename, encoding="utf8"):
            names.append(name.rstrip())
    return forenames, surnames
```

A külső `for...in` ciklusban két kételemű rendezett sorozatot járunk be. Mindkét rendezett sorozatot két változóba csomagoljuk ki. Bár mindkét lista meglehetősen hosszú lehet, visszanyerésük a függvénytől mégis hatékony lépés, lévén a Python objektumhivatkozásokat használ, vagyis valójában egyedül a két objektumhivatkozást tartalmazó rendezett sorozatot kapjuk vissza a függvénytől.

A Python nyelvű programok belsejében mindig a Unix-féle rendszereken használt módszert érdemes használni az elérési utak megadásakor, ugyanis az ilyeneket levédés nélkül írhatjuk be, ezen felül pedig minden operációs rendszeren működnek – Windowson is. Ha egy elérési utat, amelyet mondjuk a `path` változó tárol, ki akarunk írni a felhasználónak, bármikor betölthetjük az `os` modult, majd a `path.replace("/",os.sep)` tagfüggvény hívásával lecserélhetjük a perjeleket az operációs rendszerre jellemző könyvtárelválasztó karakterre.

```
forenames, surnames = get_forenames_and_surnames()
fh = open("test-names1.txt", "w", encoding="utf8")
for i in range(100):
    line = "{0} {1}\n".format(random.choice(forenames),
                              random.choice(surnames))
    fh.write(line)
```

Miután megkaptuk a két listát, a kimeneti fájlt megnyitjuk írásra, a fájl objektumot pedig az `fh` („file handle", azaz fájlleíró) nevű változóban tároljuk. Ezek után egy százszor ismétlődő ciklus minden ismétlődésekor létrehozzuk a fájlba írandó sort, nem lefelejtve a soremelést végző karaktert a sorok végéről. Az `i` ciklusváltozót nem használjuk; mindössze azért van rá szükség, hogy megfeleljünk a `for...in` ciklus utasításformájának. A programot így mindössze az előző kódrészlet, a `get_forenames_and_surnames()` függvény és a betöltést végző utasítás alkotja.

Szövegfájlok olvasása és írása 126 ☜

A `generate_test_names1.py` programban két különálló állományból beolvasott elemekből alkotott párokat helyeztünk el karakterláncokban. A két, illetve több listából vagy más bejárható objektumból származó elemek egymáshoz rendelésének másik módját a `zip()` függvény jelenti. A `zip()` függvény nullától különböző számú bejárható objektumot vár paraméterül, és egy olyan bejárót ad vissza, amely viszont rendezett sorozatok visszaadására képes. Az első rendezett sorozat a különböző átadott bejárható objektumok első elemeit tartalmaza, a második a második elemeket, és így tovább, egészen addig, amíg valamelyik objektum elemei elfogynak. Íme egy példa:

```
>>> for t in zip(range(4), range(0, 10, 2), range(1, 10, 2)):
...     print(t)
(0, 0, 1)
(1, 2, 3)
(2, 4, 5)
(3, 6, 7)
```

Bár a `range()` függvény második és harmadik hívásából származó bejárók közül mindkettő öt elemet ad, az első hívásból származó csak négyet, így ez lesz az, amelyik a `zip()` függvény által visszaadható rendezett sorozatok számát négyre korlátozza.

Alább a neveket előállító program módosított változata olvasható. Ezúttal minden név 25 karakteres, és mindegyiket egy véletlenszerű évszám követi. A program neve `generate_test_names2.py`, a kimeneti állományé pedig `test-names2.txt`. A `get_forenames_and_surnames()` és az `open()` függvények hívását nem mutatjuk be, ugyanis a fájlneveket kivéve mindenben megegyeznek az előzőekkel.

```
limit = 100
years = list(range(1970, 2013)) * 3
for year, forename, surname in zip(
        random.sample(years, limit),
```

```
            random.sample(forenames, limit),
            random.sample(surnames, limit)):
        name = "{0} {1}".format(forename, surname)
        fh.write("{0:.<25}.{1}\n".format(name, year))
```

Először az előállított nevek számának felső határát adjuk meg. Ezt követően előállítjuk az 1970-től 2012-ig terjedő évszámok listáját, amelyet háromszor lemásolunk, így az utolsó listában minden év háromszor szerepel. Erre azért van szükség, mert a `random.choice()` függvény helyett a `random.sample()` függvényt használjuk. Ez viszont paraméterül egy bejárható objektumon kívül egy számot is vár, ami megmutatja, hogy hány elemet kell a függvénynek előállítania. A szám viszont nem lehet kisebb az átadott bejárható objektum elemeinek számánál. A `random.sample()` függvény olyan bejárható objektumot ad vissza, amely az átadott bejárható adat alapján a megadott számú elemet állítja elő – ismétlődés nélkül. A program ezen változata így minden esetben egyedi neveket állít elő.

A `for...in` ciklusban kicsomagolunk minden egyes, a `zip()` függvénytől visszakapott rendezett sorozatot. Az egyes nevek hosszát legfeljebb 25 karakterre szeretnénk korlátozni, és ezt úgy végezzük el, hogy először létrehozunk egy a teljes nevet tároló karakterláncot, amelynek utóbb az `str.format()` függvénnyel beállítjuk a maximális hosszát. Az egyes neveket bal oldalra rendezzük, a 25 karakternél rövidebbek esetében pedig a fennmaradó helyet pontokkal töltjük ki. A huszonhatodik helyen álló pont biztosítja, hogy a legnagyobb megengedett hosszúságot elérő nevek is elkülönüljenek az évszámtól.

A szakasz végéhez közeledve megemlítünk még két olyan függvényt – `sorted()` és `reversed()` –, amelyek használata egyaránt a bejárható objektumokhoz kapcsolódik. A `sorted()` függvény az elemek rendezett listáját adja vissza, a `reversed()` függvény visszatérési értéke pedig nem egyéb, mint a neki paraméterül átadott bejáró elemeit fordított sorrendben tartalmazó másik bejáró. Íme egy a `reversed()` függvény használatát bemutató példa:

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>> list(reversed(range(6)))
[5, 4, 3, 2, 1, 0]
```

A `sorted()` függvény az alábbi példák tanúsága szerint kifinomultabb működésű:

```
>>> x = []
>>> for t in zip(range(-10, 0, 1), range(0, 10, 2), range(1, 10, 2)):
...     x += t
>>> x
[-10, 0, 1, -9, 2, 3, -8, 4, 5, -7, 6, 7, -6, 8, 9]
>>> sorted(x)
[-10, -9, -8, -7, -6, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sorted(x, reverse=True)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -6, -7, -8, -9, -10]
>>> sorted(x, key=abs)
[0, 1, 2, 3, 4, 5, 6, -6, -7, 7, -8, 8, -9, 9, -10]
```

Az előző kódrészletben a `zip()` függvény háromelemű rendezett sorozatokat ad vissza: `(-10, 0, 1)`-et, `(-9, 2, 3)`-at, és így tovább. A `+=` művelet a lista bővítését végzi, vagyis a neki átadott sorozat minden elemét a lista végére fűzi.

A `sorted()` függvény első hívása a lista hagyományos sorrenden alapuló rendezett másolatát, a második hívás pedig a lista hagyományos sorrenden alapuló fordított rendezési sorrendű másolatát adja vissza. A `sorted()` függvény utolsó hívásában egy „kulcsfüggvényt" adunk meg, amelyre azonnal visszatérünk.

Figyeljük meg, hogy a Python nyelv függvényei – mint ahogy minden egyéb dolog a nyelvben – objektumok, és így paraméterül adhatók más függvényeknek, valamint gyűjteményekben is éppen úgy tárolhatók, mint a hagyományos értelemben vett objektumok. Gondoljunk arra, hogy a függvénynév nem egyéb, mint a függvényre mutató objektumhivatkozás; a Python a függvény nevét követő zárójelből értesül arról, hogy a függvényt meg kell hívnia.

Amikor egy függvénynek kulcsfüggvényt adunk át – esetünkben az `abs()` függvényt –, az átadott függvényt a lista minden elemére meghívjuk (a függvény egyetlen paramétere a lista aktuális eleme), és így úgynevezett „díszített" lista jön létre. Ennek végeztével a díszített lista rendezése következik, amit a rendezett lista díszítés nélküli, eredményként való visszaadása követ. Kulcsfüggvényként módunkban áll saját egyedi függvényeket használni – amint azt hamarosan látni fogjuk.

Van például lehetőségünk karakterláncokból álló lista kis- és nagybetűk figyelembe vétele nélküli rendezésére – ilyenkor kulcsként a `str.lower()` függvényt adjuk át. Ha az `x` lista a következő: `["Sloop", "Yawl", "Cutter", "schooner", "ketch"]`, a kis- és nagybetűk figyelembe vétele nélküli DSU (Decorate, Sort, Undecorate, azaz Díszít, Rendez, Lecsupaszít) rendezés megvalósítható egyetlen sornyi, átadott kulcsfüggvényt tartalmazó kóddal, vagy a pontos DSU-algoritmussal. A következő két kódrészlet ezt mutatja be:

```
x = sorted(x, key=str.lower)
```

```
temp = []
for item in x:
    temp.append((item.lower(), item))
x = []
for key, value in sorted(temp):
    x.append(value)
```

Mindkét kódrészlet új listát hoz létre: `["Cutter", "ketch", "schooner", "Sloop", "Yawl"]`. Az elvégzett számítások azonban nem azonosak, ugyanis a jobb oldali részlet létrehoz egy `temp` nevű listát.

A Python rendezőalgoritmusa egy a feladathoz idomuló stabil összefésülő rendezést valósít meg, amely szép is, jó is, okos is. Igen jól hozzáigazították az egyébként igen gyakran előforduló, részlegesen rendezett listákhoz.* A „feladathoz idomuló" alatt azt értjük, hogy a rendezőalgoritmus kihasználja a körülményekből – például a részleges rendezettségből – eredő előnyöket. A „stabil" jelző arra vonatkozik, hogy azok az elemek, amelyek a rendezés során együtt mozoghatnának, nem változtatják meg az egymáshoz képest elfoglalt helyüket – ez ugyanis szükségtelen. Az összefésülő rendezés pedig a használt rendezőalgoritmus alapformájának a neve. Amikor egész számokat vagy egyéb egyszerű típusba tartozó objektumokat tartalmazó gyűjtemények rendezése folyik, az objektumosztály „kisebb, mint" (<) műveletét használjuk. A Python képes gyűjteményekből álló gyűjtemények bármekkora mélységig történő rekurzív rendezésére. Például:

```
>>> x = list(zip((1, 3, 1, 3), ("pram", "dorie", "kayak", "canoe")))
>>> x
[(1, 'pram'), (3, 'dorie'), (1, 'kayak'), (3, 'canoe')]
>>> sorted(x)
[(1, 'kayak'), (1, 'pram'), (3, 'canoe'), (3, 'dorie')]
```

A Python a rendezett sorozatok rendezését úgy végezte, hogy összehasonlította az összes rendezett sorozat első elemét, ha pedig egyezést talált, a második elemnél folytatta. Ezzel a rendezés az egész számokon alapult, az egyezések esetében pedig a karakterláncok döntöttek. Kikényszeríthetjük, hogy a rendezés alapját a karakterláncok képezzék, az egészeket pedig csak az egyezések esetén használjuk. Íme:

```
def swap(t):
    return t[1], t[0]
```

A `swap()` függvény paramétere egy kételemű rendezett sorozat, visszatérési értéke pedig az átadott rendezett sorozat elemeinek megfordításával létrehozott új rendezett sorozat. Felételezve, hogy a `swap()` függvényt az IDLE-ben adtuk meg, kiadhatjuk a következő parancsot:

```
>>> sorted(x, key=swap)
[(3, 'canoe'), (3, 'dorie'), (1, 'kayak'), (1, 'pram')]
```

A listák a `list.sort()` tagfüggvény használatával helyben is rendezhetők. A függvény a `sorted()` függvénnyel azonos elhagyható paramétereket kezeli.

A rendezés csak olyan gyűjtemények esetében használható, amelyeknek a tagjai összehasonlíthatók egymással:

```
sorted([3, 8, -7.5, 0, 1.3])              # eredmény: [-7.5, 0, 1.3, 3, 8]
sorted([3, "spanner", -7.5, 0, 1.3])      # TypeError kivételt vált ki
```

* Az algoritmust Tim Peters írta. A Python forráskódjában lévő `listsort.txt` állományban található hozzá egy érdekes leírás.

Bár az első lista különböző típusú – egész, illetve lebegőpontos – számokat tárol, a tárolt objektumok típusai összehasonlíthatók egymással, így a lista rendezése gond nélkül lefut. A második lista egyik eleme azonban karakterlánc, ami nem hasonlítható össze úgy egy számmal, hogy az összehasonlításnak értelme legyen, így aztán `TypeError` kivétel keletkezik. Ha olyan listát szeretnénk rendezni, amelyben egész és lebegőpontos számok, valamint karakterláncként tárolt számok egyaránt megtalálhatók, a `float()` függvényt kell átadnunk kulcsfüggvényként:

```
sorted(["1.3", -7.5, "5", 4, "-2.4", 1], key=float)
```

Ez a függvényhívás a `[-7.3, '-2.4', 1, '1.3', 4, '5']` listával tér vissza. Figyeljük meg, hogy a lista értékeiben nem történik változás – a karakterláncok karakterláncok maradnak. Ha a függvény olyan karakterláncba botlik, amely nem alakítható számmá (mondjuk abba, hogy „sárgarigófészek"), `TypeError` kivétel lép fel.

Gyűjtemények másolása

Objektumhivatkozások 9 ☜

Minthogy a Python nyelv objektumhivatkozásokkal dolgozik, az értékadó művelet (=) nem jár másolással. Ha a jobb oldali tényező egy konkrét érték, például karakterlánc vagy szám, a bal oldali tényező értéke egy a memóriában lévő és az értéket tároló objektumra mutató objektumhivatkozás lesz. Ha a jobb oldali tényező egy objektumhivatkozás, a bal oldali tényező értéke egy olyan objektumhivatkozás lesz, amely a jobb oldalon állóval azonos objektumra hivatkozik. Mindezeknek az az egyik folyománya, hogy az értékadás nagyon hatékony művelet.

Amikor terjedelmes gyűjteményeket, például hosszú listákat adunk értékül, a hatékonyság igen szembeszökő. Íme egy példa:

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs
>>> beatles, songs
(['Because', 'Boys', 'Carol'], ['Because', 'Boys', 'Carol'])
```

A fenti kódrészlettel létrehozunk egy új objektumhivatkozást (`beatles`), így két, ugyanarra a listára hivatkozó objektumhivatkozást tartalmazó objektumunk lesz – másolás pedig nem történik.

Minthogy a listák változtathatók, foganatosítunk is egy változtatást. Például:

```
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Cayenne'])
```

A változtatást a `beatles` változó használatával ejtettük meg, ami azonban egy objektumhivatkozás, és ugyanarra a listára hivatkozik, amire a `songs` változó is. Vagyis minden olyan változtatás, amelyet az egyik objektumhivatkozás használatával végeztünk, látható

a másik számára is. Leggyakrabban pontosan ez az a viselkedés, amit megkívánunk, lévén a terjedelmes gyűjtemények másolásának esetleg magas a költsége. Mindez azzal is együtt jár, hogy amennyiben listát vagy egyéb változtatható objektumot adunk át egy olyan függvénynek, amely megváltoztatja az objektum értékét, a függvényhívás végeztével a módosított gyűjtemény hozzáférhető marad.

Mindazonáltal előfordulnak olyan helyzetek, amikor a gyűjtemény – vagy más változtatható objektum – ténylegesen önálló másolatát szeretnénk használni. A sorok szeletelésekor – például a `songs[:2]` utasítással – a részsorozat minden esetben az átmásolt elemek független másolatát tartalmazza majd. Ha teljes sorozatról kívánunk másolatot létrehozni, az alábbi módszer szerint érdemes eljárnunk:

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs[:]
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Carol'])
```

Szótárak és halmazok esetében a másolás a `dict.copy()` és a `set.copy()` függvények használatával végezhető el. Ezen felül a `copy` modulban megtalálható a `copy.copy()` függvény is, amely a neki átadott objektum másolatával tér vissza. A beépített gyűjteményes adattípusokba tartozó objektumok másolásának további módja, ha a típust függvényként használjuk, és a másolandó objektumot adjuk át neki paraméterül. Íme néhány példa:

```
copy_of_dict_d = dict(d)
copy_of_list_L = list(L)
copy_of_set_s = set(s)
```

Vegyük azonban észre, hogy ezek a másolási módszerek *sekély másolások* – vagyis csak az objektumhivatkozásokat másolják, magukat az objektumokat nem. Állandó értékű objektumok – számok, karakterláncok – tekintetében ez ugyanazt a hatást éri el, mint a másolás (csak hatékonyabb). A változtatható objektumok esetében – ilyenek az egymásba ágyazott gyűjtemények – mindez annyit tesz, hogy az eredeti és a másolt gyűjtemény objektumhivatkozásai ugyanazokra az objektumokra hivatkoznak. A következő kódrészlet is ezt szemlélteti:

```
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = x[:] # shallow copy
>>> x, y
([53, 68, ['A', 'B', 'C']], [53, 68, ['A', 'B', 'C']])
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['Q', 'B', 'C']])
```

Ha az x listának egy sekély másolatát hozzuk létre, valójában a beágyazott `["A", "B", "C"]` listára hivatkozó objektumhivatkozást másoljuk. Ez annyit tesz, hogy az x és az y lista egyaránt egy az előző listára mutató objektumhivatkozást tartalmaz harmadik elemként, vagyis a beágyazott listában bekövetkező változások az x és az y lista számára egyaránt láthatók. Ha az így egymásba ágyazott gyűjteményekről valóban független másolatokat szeretnénk készíteni, mély másoláshoz kell folyamodnunk:

```
>>> import copy
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = copy.deepcopy(x)
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['A', 'B', 'C']])
```

Ezúttal az x és y listák, illetve a bennük tárolt elemek egymástól tökéletesen függetlenek lesznek.

Nem árt megjegyeznünk, hogy a könyv mostantól a *másolás* és a *sekély másolás* fogalmakat egymással felcserélhetően használja – *mély másolásra* csak akkor gondoljon a kedves Olvasó, ha a mély másolat kifejezést olvassa.

Példák

Végére értünk a Python beépített adattípusait, illetve az alapkönyvtár két gyűjteményes adattípusát (`collections.namedtuple` és `collections.defaultdict`) tárgyaló áttekintésünknek. A Python nyelvben megtalálható a kétvégű sort megvalósító `collections.deque` típus is, más forrásokból, például a Python Package Index weboldalról (`pypi.python.org/pypi`) pedig továbbiak is hozzáférhetők. Most mindenesetre áttekintünk néhány, az eddigieknél kissé hosszabb példát, amelyek a mostani és az előző fejezetben elsajátított ismeretekre támaszkodnak.

Az első program nagyjából hetven soros, és a szövegfeldolgozás tárgykörébe kalauzol bennünket. A második program körülbelül kilencven sorból áll, és legfőképp a matematika kedvelőinek érdeklődésére tarthat számot. A programok szótárakat, listákat, nevesített rendezett sorozatokat és halmazokat használnak, és mindkettő erősen támaszkodik az előző fejezetben megismert `str.format()` tagfüggvényre.

generate_usernames.py

Képzeljük el, hogy éppen egy új rendszert állítunk fel, és a cégünk valamennyi dolgozója számára felhasználóneveket kell meghatároznunk. Rendelkezésünkre áll egy sima, UTF-8 kódolású szöveget tartalmazó fájl, amelyben egy-egy sor tartalmazza az egyes adatrekordokat, a rekordokban lévő mezőket pedig kettőspontok határolják. Egy rekordban egy dolgozó adatait tároljuk, a mezők pedig a következők: egyedi dolgozóazonosító (staff

ID); keresztnév (forename); középső név (middle name), amely akár üres is maradhat; vezetéknév (surname) és a részleg neve (department name). Álljon itt néhány sor a példaként szolgáló `data/users.txt` adatállományból.

```
1601:Albert:Lukas:Montgomery:Legal
3702:Albert:Lukas:Montgomery:Sales
4730:Nadelle::Landale:Warehousing
```

A programnak be kell olvasnia a parancssorban megadott valamennyi adatfájlt, minden sor (rekord) esetében el kell végeznie a mezőkre bontást, és megfelelő felhasználónevekkel kell visszatérnie. Minden felhasználónévnek egyedinek és a személy nevén alapulónak kell lennie. A kimenetet a konzolra írt szöveg formájában várjuk, vezeték- és keresztnév szerinti ábécésorrendben, vagyis így:

```
Name                              ID   Username
--------------------------------- ------ ---------
Landale, Nadelle.................. (4730) nlandale
Montgomery, Albert L.............. (1601) almontgo
Montgomery, Albert L.............. (3702) almontgo1
```

Minden rekordban pontosan öt mező található, és bár hivatkozhatnánk rájuk szám szerint is, neveket használunk, hogy a kód könnyebben olvasható legyen:

```
ID, FORENAME, MIDDLENAME, SURNAME, DEPARTMENT = range(5)
```

A Python-programozók körében hagyomány, hogy a nagybetűs nevű változókat állandóként kezelik.

Szükségünk van továbbá egy az egyes felhasználók adatait tartalmazó nevesített rendezett sorozatra:

```
User = collections.namedtuple("User",
                              "username forename middlename surname id")
```

Ha egy pillantást vetünk a kód fennmaradó részére, megértjük az állandók és a `Users` nevű nevesített rendezett sorozat használatát.

A program fő részét a `main()` függvény hordozza:

```
def main():
    if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
        print("usage: {0} file1 [file2 [... fileN]]".format(
              sys.argv[0]))
        sys.exit()

    usernames = set()
    users = {}
    for filename in sys.argv[1:]:
        for line in open(filename, encoding="utf8"):
```

```
                line = line.rstrip()
                if line:
                user = process_line(line, usernames)
                users[(user.surname.lower(), user.forename.lower(),
                       user.id)] = user
    print_users(users)
```

Ha a felhasználó a parancssorban nem ad meg fájlnevet, illetve a parancssorba `-h` vagy `--help` kerül, egyszerűen kiírjuk a használati utasítást, és megszakítjuk a program futását.

Minden beolvasott sor végéről eltávolítjuk a szóköz, a tabulátor és a soremelés (`\n`) jelét, és csak azokat a sorokat dolgozzuk fel, amelyek nem üresek. Ez annyit tesz, hogy amennyiben az adatfájlban üres sorok vannak, minden további nélkül feldolgozatlanul hagyjuk őket.

A már felhasznált felhasználóneveket a `usernames` halmazban tartjuk nyilván, így ügyelve arra, hogy ne fordulhasson elő többször is ugyanaz a felhasználónév. Maga az adat a `users` nevű szótárba kerül. Minden felhasználó (vagyis dolgozó) önálló elemet alkot a szótárban. Az elem kulcsa a felhasználó vezetéknevéből, keresztnevéből és dolgozóazonosítójából álló rendezett sorozat, az elem értéke pedig a `User` típusú nevesített `tuple` objektum. Az, hogy a felhasználó vezetéknevéből, keresztnevéből és dolgozóazonosítójából álló rendezett sorozatot használunk a szótár kulcsaként, annyit tesz, hogy ha a szótárra meghívjuk a `sorted()` függvényt, a visszaadott bejárható objektum az általunk kívánt sorrendbe (vagyis vezetéknév, keresztnév, dolgozóazonosító) kerül, és ehhez nem kell kulcsfüggvényt átadnunk.

```
def process_line(line, usernames):
    fields = line.split(":")
    username = generate_username(fields, usernames)
    user = User(username, fields[FORENAME], fields[MIDDLENAME],
                fields[SURNAME], fields[ID])
    return user
```

Minthogy az egyes rekordok adatformátuma ilyen egyszerű, és mert a sorok végén lévő eltávolítandó karaktereket már eltávolítottuk, a mezők kinyerése egyszerűen a kettőspontok mentén történő vágásnak felel meg. A `generate_usernames()` függvénynek átadjuk a mezőket és a `usernames` halmazt, majd létrehozunk egy a `User` nevesített rendezett sorozat típuson alapuló objektumpéldányt, amelyet visszaadunk a hívó függvénynek – a `main()`-nek –, ami aztán a kiírásra kész felhasználót elhelyezi a `users` szótárban.

Ha a sorszámok tárolására nem hoztunk volna létre megfelelő állandókat, számokat kellene megadnunk, például így:

```
user = User(username, fields[1], fields[2], fields[3], fields[0])
```

Bár ez rövidebb megoldás, mégis ellenjavallt. Először is, a kód jövőbeni karbantartója számára nem világos az egyes mezők szerepe, másodszor pedig kiteszi a programot az adatfájl formátumának változásából adódó problémáknak – ha a rekord mezői más sorrendbe kerülnek, a programunk minden olyan helyen hibássá válik, ahol a mezőkre hivatkozunk. Azzal, hogy a rekord szerkezetét nevesített állandókkal ábrázoljuk, ilyen esetekben a program működőképességének helyreállításához elegendő az állandók értékének megváltoztatása.

```
def generate_username(fields, usernames):
    username = ((fields[FORENAME][0] + fields[MIDDLENAME][:1] +
                 fields[SURNAME]).replace("-", "").replace("'", ""))
    username = original_name = username[:8].lower()
    count = 1
    while username in usernames:
        username = "{0}{1}".format(original_name, count)
        count += 1
    usernames.add(username)
    return username
```

A felhasználónév előállítását célzó első kísérletünk lényege, hogy vesszük a vezetéknév és a középső név első betűjét és a teljes keresztnevet, majd a létrejött karakterláncból töröljük a kötőjeleket és az aposztrófokat. A középső név első karakterének kinyeréséhez használt kódban egy kis trükköt alkalmazunk. Ha a `fields[MIDDLENAME][0]` utasítást használnánk, akkor `IndexError` kivétel keletkezne minden olyan esetben, amikor nincs középső név. Azáltal, hogy szeletet képzünk, megkapjuk az első betűt, ha van első betű, és üres karakterláncot kapunk, ha nincs.

Ennek végeztével kisbetűssé alakítjuk a felhasználónevet, és nyolc karakterre korlátozzuk a hosszát. Ha már van ilyen felhasználónév (vagyis benne van a `usernames` halmazban), megpróbálkozunk azzal, hogy a végére odaírunk egy 1-est; ha ez is használt már, akkor egy 2-est, és így tovább. Ezek után elhelyezzük a felhasználónevet a `usernames` halmazban, végül a felhasználónevet visszaadjuk a hívónak:

```
def print_users(users):
    namewidth = 32
    usernamewidth = 9
    print("{0:<{nw}} {1:^6} {2:{uw}}".format(
          "Name", "ID", "Username", nw=namewidth, uw=usernamewidth))
    print("{0:-<{nw}} {0:-<6} {0:-<{uw}}".format(
          "", nw=namewidth, uw=usernamewidth))
    for key in sorted(users):
        user = users[key]
        initial = ""
        if user.middlename:
            initial = " " + user.middlename[0]
        name = "{0.surname}, {0.forename}{1}".format(user, initial)
        print("{0:.<{nw}} ({1.id:4}) {1.username:{uw}}".format(
              name, user, nw=namewidth, uw=usernamewidth))
```

Ha minden rekord feldolgozásával végeztünk, meghívjuk a `print_users()` függvényt, a `users` szótárat adva neki paraméterül.

`str.format()`
75 ☜

Az első `print()` utasítás kiírja az oszlopfejléceket, a második pedig kötőjelekkel húzza alá azokat. A második utasítás `str.format()` hívása is érdekes. Az átadott, kiírandó karakterlánc a `""`, vagyis egy üres karakterlánc. A kötőjeleket úgy kapjuk, hogy az üres karakterláncot kötőjelekkel a megfelelő hosszúságúra nyújtjuk.

Alább egy `for...in` ciklus használatával, illetve a felhasználókat tároló szótár kulcsának kicsomagolásával rendezett sorrendben kiírjuk az egyes felhasználók adatait. A `user` változót kényelmi okból hozzuk létre: így a függvény hátralévő részében nem kell mindig kiírnunk a `users[key]` kifejezést. Amikor a függvény először hívja meg a `str.format()` tagfüggvényt, a `name` változót vezetéknév, keresztnév (és az esetleges középső név kezdőbetűje) formátumúra állítjuk. A `user` nevesített rendezett sorozat elemeit név szerinti hivatkozással érjük el. Amikor a felhasználó neve rendelkezésünkre áll egyetlen karakterláncként, kiírjuk a felhasználó adatait, mégpedig úgy, hogy az egyes oszlopok (név, azonosító, felhasználónév) szélessége a nekünk tetsző legyen.

A teljes program, ami egyébként csak annyiban különbözik attól, amit most áttekintettünk, hogy van az elején néhány sornyi megjegyzés és modulbetöltés, `generate_usernames.py` néven található meg. A program szerkezete – adatfájl beolvasása, rekordok feldolgozása, kimenet képernyőre írása – igen gyakran használatos, és a következő példában is újra előkerül.

statistics.py

Tegyük fel, hogy van egy adag adatfájlunk, bennük olyan számokkal, amelyek valamilyen általunk korábban elvégzett adatfeldolgozással kapcsolatosak, és az adatok áttekintéséhez valamilyen alapvető statisztikát szeretnénk készíteni. Valamennyi fájl egyszerű, ASCII karakterkódolású szöveges állomány, amelynek soraiban szóközzel elválasztva egy vagy több szám áll.

Az alábbi példa mutatja az előállítani kívánt kimenetet, benne a számosság, a középérték, a medián, a módusz és a standard deviáció értékeivel:

```
count      =     183
mean       =     130.56
median     =      43.00
mode       = [5.00, 7.00, 50.00]
std. dev.  =     235.01
```

A fenti példában 183 számot olvastunk be. Az 5, a 7 és az 50 volt a három leggyakrabban előforduló szám, a minta standard deviációja pedig 235.01.

Maga a statisztika a `Statistics` nevű nevesített rendezett sorozatba kerül:

```
Statistics = collections.namedtuple("Statistics",
                                    "mean mode median std_dev")
```

A `main()` függvény a program szerkezetének áttekintésére is szolgál:

```
def main():
    if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
        print("usage: {0} file1 [file2 [... fileN]]".format(
              sys.argv[0]))
        sys.exit()
    numbers = []
    frequencies = collections.defaultdict(int)
    for filename in sys.argv[1:]:
        read_data(filename, numbers, frequencies)
    if numbers:
        statistics = calculate_statistics(numbers, frequencies)
        print_results(len(numbers), statistics)
    else:
        print("no numbers found")
```

Az összes fájlban előforduló összes számot a `numbers` listában tároljuk. A módusz, vagyis a leggyakrabban előforduló számok megállapításához tudnunk kell, hogy az egyes számok hányszor fordulnak elő, így az előfordulások tárolására létrehozunk egy az `int()` gyártófüggvényt használó alapértelmező szótárat.

Egyesével beolvassuk az egyes fájlok adatait. A `read_data()` függvény további paraméterként megkapja a listát és az alapértelmező szótárat, így képessé válik a frissítésükre. Ha az adatok beolvasásával végeztünk, feltételezve, hogy számok is előfordultak közöttük, a `calculate_statistics()` függvény hívásával folytatjuk a feldolgozást. A függvény a `Statistics` típusba tartozó nevesített rendezett sorozattal tér vissza, amit aztán az eredmények kiírására használunk.

```
def read_data(filename, numbers, frequencies):
    for lino, line in enumerate(open(filename, encoding="ascii"),
                                 start=1):
        for x in line.split():
            try:
                number = float(x)
                numbers.append(number)
                frequencies[number] += 1
            except ValueError as err:
                print("{0}:{1}: skipping {2}: {3}".format(
                      filename, lino, x, err))
```

A sorokat a szóközöknél osztjuk fel, és minden így kapott elemet megpróbálunk lebegőpontos számmá alakítani. Ha az átalakítás sikeres – vagyis az egész számoknál, illetve a lebegőpontos számoknál, tizedestört- és normálalakban egyaránt –, a szám bekerül a `numbers` listába, és frissítjük a `frequencies` alapértelmező szótárat. (Ha egyszerű szótárat használtunk volna, a frissítést végző kód a következő lenne: `frequencies[number] = frequencies.get(number, 0) + 1`.) Ha az átalakítás sikertelen, kiírjuk a sor számát (a szövegfájloknál megszokott módon a sorok számozását 1-gyel kezdjük), azt a szöveget, amelyet megpróbáltunk átalakítani, illetve a `ValueError` kivétel hibaüzenetét.

```
def calculate_statistics(numbers, frequencies):
    mean = sum(numbers) / len(numbers)
    mode = calculate_mode(frequencies, 3)
    median = calculate_median(numbers)
    std_dev = calculate_std_dev(numbers, mean)
    return Statistics(mean, mode, median, std_dev)
```

A fenti függvényt használjuk a statisztikai adatok összegyűjtésére. Minthogy a középérték – az „átlag” – kiszámítása igen egyszerű, a műveletet közvetlenül itt végezzük el. Az egyéb statisztikai adatok esetében az adott számításnak szentelt függvényt hívjuk meg. A fenti függvény végezetül a négy kiszámított statisztikai adatot tartalmazó, `Statistics` típusú nevesített rendezett sorozattal tér vissza.

```
def calculate_mode(frequencies, maximum_modes):
    highest_frequency = max(frequencies.values())
    mode = [number for number, frequency in frequencies.items()
            if math.fabs(frequency - highest_frequency) <=
               sys.float_info.epsilon]
    if not (1 <= len(mode) <= maximum_modes):
        mode = None
    else:
        mode.sort()
    return mode
```

Elképzelhető, hogy nem csak egy „leggyakrabban előforduló” számot találunk, vagyis a fenti függvény esetében az előfordulási adatokat tartalmazó szótáron felül a hívó függvénynek az elfogadható móduszok maximális számát is meg kell adnia. (A hívó függvény a `calculate_statistics()`, és legfeljebb három módusz engedélyezett.)

Az előfordulási gyakoriságokat tartalmazó szótárban a legmagasabb érték megkeresését a `max()` függvény végzi. Ha a legmagasabb értéket megleltük, egy listaértelmezéssel állítjuk elő azoknak a számoknak a listáját, amelyek előfordulásának száma egyenlő ezzel a legmagasabb számmal. Minthogy a számok törtszámok is lehetnek, a különbségek abszolút értékét a `math.fabs()` tagfüggvénnyel hasonlítjuk össze, ez ugyanis a gép által ábrázolható legkisebb szám tekintetében jobb eredményt ad, mint az `abs()` függvény.

Ha a móduszok száma nulla, illetve nagyobb az elfogadhatónál, `None` értéket kapunk vissza móduszként, egyéb esetben pedig a móduszok rendezett listája lesz a visszaadott érték.

A medián (a „középső érték") az az érték, amely a számok rendezett sorának közepén helyezkedik el, kivéve, ha a számok száma páros szám, ilyenkor ugyanis a közép kettő szám közé esik, ilyenkor tehát a középső két érték középértéke a medián.

Először növekvő sorrendbe rendezzük a számokat. Ennek végeztével csonkoló (egész-) osztással megállapítjuk a középső szám sorszámát, amit aztán kinyerünk a sorozatból, és mediánként tárolunk. Ha a számok száma páros szám, a mediánt a két középső szám középértékével tesszük egyenlővé.

```
def calculate_median(numbers):
    numbers = sorted(numbers)
    middle = len(numbers) // 2
    median = numbers[middle]
    if len(numbers) % 2 == 0:
        median = (median + numbers[middle - 1]) / 2
    return median
```

A minta standard deviációja az eloszlás jellemzésére szolgál, és azt mutatja meg, hogy a számok mennyire különböznek a középértéktől. A fenti programfüggvény a következő matematikai függvény használatával számítja ki a minta standard deviációját: $s=\sqrt{\frac{\Sigma(x-\bar{x})^2}{n-1}}$ A képletben az `x` jelöli az aktuális számot, az $\bar{x}$ a középértéket, az `n` pedig a minta számosságát.

```
def print_results(count, statistics):
    real = "9.2f"
    if statistics.mode is None:
        modeline = ""
    elif len(statistics.mode) == 1:
        modeline = "mode = {0:{fmt}}\n".format(
                statistics.mode[0], fmt=real)
    else:
        modeline = ("mode       = [" +
                    ", ".join(["{0:.2f}".format(m)
                    for m in statistics.mode]) + "]\n")
    print("""\
count      = {0:6}
mean       = {1.mean:{fmt}}
median     = {1.median:{fmt}}
{2}\
std. dev.  = {1.std_dev:{fmt}}""".format(
        count, statistics, modeline, fmt=real))
```

str.format()
75

A fenti függvény legnagyobb részének az a feladata, hogy a móduszok listájából kialakítsa a `modeline` karakterláncot. Ha nincs módusz, az értéket megjelenítő sort sem írjuk a képernyőre. Ha csak egy módusz van,

akkor a móduszok listája egyelemű (`mode[0]`), amit a többi statisztikai adat esetében is használt formátumban írunk ki. Ha több móduszértékünk van, listaként írjuk ki azokat, a megfelelő formázást egyesével végezve el. Ennek érdekében egy listaértelmezéssel előállítjuk a móduszok karakterláncait, majd a karakterláncokból listát képzünk, az egyes elemek közti elválasztásként egy vesszőt és egy szóközt használva. A függvény végén lévő kiírás a nevesített rendezett sorozat használatának köszönhetően egyszerű. Így lehetővé válik, hogy a `statistics` objektumban tárolt statisztikai adatot sorszámok helyett nevek alapján érjük el, és köszönhetően a Python nyelv háromszoros idézőjeleinek, a szöveg elrendezését érthető alakban adhatjuk meg.

Egyetlen apróságra hívnánk még fel a figyelmet. A móduszokat `{2}` elem formában írjuk ki, amit egy fordított perjel követ. A fordított perjel levédi az újsor karaktert, vagyis akkor sem írunk üres sort, ha a módusz üres karakterlánc. A `modeline` karakterlánc végére pedig azért kell nem üres karakterlánc esetén kitennünk az új sort kezdő `\n` kódot, mert előzőleg levédtük az újsor karaktert.

Összefoglalás

A fejezetben szó volt a Python nyelv valamennyi beépített gyűjteményes adattípusáról, és az alapkönyvtár néhány gyűjteményes adattípusáról is. A sorozattípusok közül áttekintettük a `tuple`, a `collections.namedtuple` és a `list` típust, amelyek egyaránt támogatják a karakterláncokéval egyező szeletelési és egyelési műveleteket. Megbeszéltük a sorozatkicsomagoló művelet (`*`) használatát, és érintettük a függvényhívásokban szereplő megcsillagozott paraméterek témakörét. Tárgyaltuk továbbá a halmaztípusokat – `set` és `frozenset` –, illetve a megfeleltetési típusokat – `dict` és `collections.defaultdict`.

Láttuk, hogy a Python alapkönyvtárából származó nevesített `tuple` típus miként használható olyan egyszerű, egyedi `tuple` adattípus létrehozására, amelynek az elemei sorszám alapján, vagy kényelmesebb módszerrel, név alapján is elérhetők. Láttuk az „állandók" létrehozásának módját is, amikor csak nagybetűkből álló változóneveket választottunk.

A listák tárgyalása során megállapítottuk, hogy mindaz, ami elvégezhető rendezett sorozatokkal, elvégezhető listákkal is. Annak köszönhetően, hogy a `list` típus objektumai változtathatók, a rendezett sorozatoknál lényegesen szélesebb skálán használhatók. Vannak olyan függvényeik, amelyek a listát módosítják (például a `list.pop()`), amelyekkel lehetővé válik a szeleteknek az értékadó művelet bal oldalán történő használata, illetve amelyekkel végrehajtható szeletek beszúrása, cseréje vagy törlése. A listák ideális eszközei az elemsorozatok tárolásának, főleg ha sorszám alapján történő gyors hozzáférésre van szükség.

Amikor a `set` és a `frozenset` adattípusokat tárgyaltuk, megemlítettük, hogy mindkettő csak kivonatolható típusokba tartozó objektumokat tartalmazhat. A halmazok esetében gyors a tagsági viszony vizsgálata, ezért remekül alkalmasak adatok többszöri előfordulásának kiszűrésére.

A szótárak bizonyos tekintetben hasonlítanak a halmazokra – például a kulcsaiknak kivonatolhatónak és egyedieknek kell lenniük, mint egy halmaz elemeinek. A szótárak azonban kulcs-érték párokat tárolnak, az érték típusa pedig bármi lehet. A szótárak tárgyalása során szóba került a `dict.get()` és a `dict.setdefault()` tagfüggvény, az alapértelmező szótárak tárgyalása során pedig bemutattuk két tagfüggvény használatának egy a szokásostól eltérő módját. A halmazokhoz hasonlóan a szótárak esetében is gyors a tagsági viszony vizsgálata. A kulcs alapján történő hozzáférés szintén gyors művelet.

A listák, a halmazok és a szótárak esetében egyaránt elérhető a tömör „értelmezés” utasításforma. Ennek az utasításformának a használatával lehetőségünk nyílik az említett adattípusok előállítására bejárható adatok alapján – amelyek elemei maguk is lehetnek értelmezések. A bejárható adatokhoz szükség esetén feltétel is kapcsolható. Gyűjtemények létrehozása során gyakran használjuk a `range()` és a `zip()` függvényt, a hagyományos `for..in` ciklusokban éppúgy, mint az értelmezésekben.

Az elemek törlése a változtatható gyűjtemények megfelelő tagfüggvényeinek – mint a `list.pop()` és a `set.discard()` – használatával, illetve a `del` utasítás kiadásával lehetséges; a `d` szótár `k` kulcsú elemének törlését például a `d[k]` utasítás végzi.

Az a tény, hogy a Python objektumhivatkozásokkal dolgozik, különösen hatékonnyá teszi az értékadás műveletét, ugyanakkor azt is jelenti, hogy az értékadás művelete (=) során nem történik meg az objektumok másolása. Láttuk a sekély és a mély másolás közötti különbséget, később pedig azt is, hogy miként lehetséges a listák sekély másolása a teljes listát tartalmazó szelet képzésével `L[:]`, és miként lehetséges a szótárak sekély másolása a `dict.copy()` tagfüggvénnyel. A `copy` modul függvényeinek használatával bármilyen másolható objektum másolása megvalósítható. A `copy.copy()` tagfüggvény való a sekély másolásra, a `copy.deepcopy()` tagfüggvény pedig a mély másolásra.

Bemutattuk a Python magas szinten optimalizált `sorted()` függvényét. A függvényt elterjedten használják Python nyelvű programok írásakor, a Python nyelv ugyanis nem rendelkezik olyan adattípussal, amely önmagában képes lenne megvalósítani az adatok rendezését. Így ha gyűjtemények rendezett bejárását kell elvégeznünk, a `sorted()` függvényt használjuk.

A Python beépített gyűjteményes adattípusai – a rendezett sorozat, a lista, a halmaz, a fagyasztott halmaz és a szótár – elegendőek minden feladat ellátására. Mindazonáltal az alapkönyvtár kínál néhány további gyűjteményes adattípust, és egyéb forrásokból sok más típushoz is hozzáférhetünk.

Gyakran van szükségünk gyűjtemények fájlokból való olvasására, illetve fájlokba való írására. A fejezetben éppen csak érintettük a szövegfájlok kezelésének tárgykörét, amikor megvalósítottuk a fájlok soronkénti írását és olvasását. A fájlkezelés kimerítő tárgyalására a 7. fejezetben kerül sor, az adatkezelés egyéb kérdéseit pedig a 11. fejezetben vizsgáljuk.

A következő fejezetben a Python nyelv vezérlési szerkezeteit vesszük közelebbről szemügyre, és megismerkedünk egy új szerkezettel is. Mélyebbre ássuk magunkat a kivételkezelés témakörébe, és pár eddig nem tárgyalt kifejezés, például az `assert` használatába. Ezen felül szó lesz még az egyedi függvények írásáról, különös tekintettel a Python elképesztően rugalmas paraméterkezelésére.

Gyakorlatok

1. Módosítsuk úgy az `external_sites.py` programot, hogy alapértelmező szótárat használjon. Ez a változtatás egyszerű, mindössze egyetlen betöltést, azon felül pedig két sor megváltoztatását igényli. A megoldást az `external_sites_ans.py` fájlban találjuk.
2. Módosítsuk úgy a `uniquewords2.py` programot, hogy a kimenet az ábécésorrend helyett a szavak előfordulási gyakoriságát vegye figyelembe. Be kell járnunk a szótár elemeit, meg kell alkotnunk azt a rövidke, kétsoros függvényt, amely kinyeri az egyes elemek értékeit, és amikor elkészült, mint kulcsfüggvényt kell átadnunk a `sorted()` függvénynek. Megfelelően módosítanunk kell a `print()` függvény hívását is – a művelet nem nehéz, de egy kis trükkre van szükség. A megoldást a `uniquewords2_ans.py` fájlban találjuk.
3. Módosítsuk úgy a `generate_usernames.py` programot, hogy soronként két felhasználó adatait írja ki. A nevek hosszát korlátozzuk tizenhét karakterre, minden hatvannegyedik sor után pedig szúrjunk be egy lapdobást. Az oszlopfejléceknek minden oldal tetején szerepelniük kell. Íme a várt kimenetet bemutató minta:

```
Name                ID    Username  Name                ID    Username
------------------ ------ -------- ------------------ ------ --------
Aitkin, Shatha...  (2370) saitkin   Alderson, Nicole.  (8429) nalderso
Allison, Karma...  (8621) kallison  Alwood, Kole E...  (2095) kealwood
Annie, Neervana..  (2633) nannie    Apperson, Lucyann  (7282) leappers
```

Ez a feladat már igazi kihívást rejt magában. Az oszlopfejléceket változókban kell tárolnunk, hogy a szükséges helyeken kiírhassuk őket. A formátumleírást is úgy kell megváltoztatnunk, hogy a szűkebb neveknek megfelelő legyen. Az oldalakra tördelés megvalósításának egyik módja, ha a kiírásra szánt adatok egy listába kerülnek, majd végigjárva a listát, megkapjuk a jobb, illetve a bal oldalra kerülő elemeket, amelyek párba állítását a `zip()` függvénnyel oldjuk meg. A feladat egy megoldását a `generate_usernames_ans.py` fájl tartalmazza. A `data/users2.txt` állományt használhatjuk adatmintaként.

4

Vezérlési szerkezetek és függvények

- Vezérlési szerkezetek
- Kivételkezelés
- Egyedi függvények

Fejezetünk első két részében a Python vezérlési szerkezeteit vesszük górcső alá – elsőként az elágazásokkal és a ciklusokkal megismerkedünk meg, majd a kivételkezeléssel foglalkozunk. Mindezek alapjairól az első fejezetben szóltunk, de a következő oldalakon alaposabb tárgyalásukra is mód nyílik, emellett megismerkedünk a vezérlési szerkezetek eddigiekben be nem mutatott használati alakjaival, továbbá megtanuljuk, hogyan váltsunk ki kivételeket, és szót ejtünk az egyedi kivételek létrehozásáról is.

A fejezet harmadik, a hosszát tekintve legtekintélyesebb részében az egyedi függvényekkel foglalkozunk, részletesen bemutatva a Python különösen rugalmas paraméterkezelési lehetőségeit. Az egyedi függvények lehetővé teszik, hogy elkülönítsük és paraméterekkel szabályozhatóvá tegyük a programok egyes részeit – ezzel a program mérete is csökken, hiszen így megszabadulhatunk egyes ismétlődő kódrészletektől, továbbá lehetővé válik a kód újrahasznosítása. (A következő fejezetben megtanuljuk, hogyan készíthetünk egyedi modulokat, így a függvényeinket több programban is felhasználhatjuk.)

Vezérlési szerkezetek

A Python a feltételes elágazásokat az `if`, a ciklusok használatát pedig a `while` és a `for...in` utasítások révén teszi lehetővé, emellett létezik a nyelvben egy úgynevezett *feltételes kifejezés* is – ez egyfajta `if` utasítás, amely megfelel a C stílusú nyelvekben létező háromtényezős műveletnek (`:?`).

Feltételes elágazások

Amint az első fejezetben láthattuk, a Python feltételes elágazási utasításának használati alakja valahogy így fest:

```
if logikai_kifejezés1:
    kódblokk1
elif logikai_kifejezés2:
    kódblokk2
...
elif logikai_kifejezésN:
    kódblokkN
else:
    else_kódblokk
```

Az `elif` kódblokkok száma tetszőleges – akár 0 is – lehet, továbbá a záró `else` kódblokk feltüntetése sem kötelező. Ha meg szeretnénk emlékezni egy esetről, de nem kívánunk semmit tenni, amennyiben bekövetkezik, helykitöltőként a `pass` kulcsszót használhatjuk.

Egyes esetekben az `if...else` utasítást egyetlen *feltételes kifejezéssé* zsugoríthatjuk össze, méghozzá így:

```
kifejezés1 if logikai_kifejezés else kifejezés2
```

Amennyiben a *`logikai_kifejezés`* értéke `True`, a feltételes kifejezés eredménye *`kifejezés1`* lesz, ellenkező esetben pedig *`kifejezés2`*.

Gyakori programozási módszer egy alapértelmezett érték beállítása akár például felhasználói kérésre, akár annak érdekében, hogy megfeleljünk a program környezetét adó rendszer követelményeinek. Mindezt a következőképpen érhetjük el egy hagyományos `if` utasítással:

```
offset = 20
if not sys.platform.startswith("win"):
    offset = 10
```

A `sys.platform` változó tárolja az aktuális rendszer nevét; az értéke lehet például „win32” vagy „linux2”. Ugyanezt az eredményt azonban elérhetjük egyetlen sorral is, ha feltételes kifejezést használunk:

```
offset = 20 if sys.platform.startswith("win") else 10
```

Itt nincs szükség zárójelekre, de a használatuk segíthet elkerülni egy igen alattomos csapdát. Tegyük fel például, hogy a `width` változó értékét 100-ra szeretnénk állítani, hozzáadva 10-et, amennyiben a `margin` értéke `True`. Próbálkozhatunk az alábbi kóddal:

```
width = 100 + 10 if margin else 0 # HIBÁS!
```

Figyelem! *A jelenség különösen becsapós, hiszen a kód tökéletesen működik, ha a margin értéke True, eredményként a várt 110-es értéket adva. Ha azonban a margin értéke False, a várt 100 helyett 0-t kapunk. Ennek az az oka, hogy a Python értelmezésében a 100 + 10 felel meg a feltételes kifejezés `kifejezés1` tagjának. A megoldást a zárójelek használata jelenti:*

```
width = 100 + (10 if margin else 0)
```

A zárójelek emellett az emberi szem számára is tisztábbá teszik a kódot.

A feltételes kifejezésekkel javíthatunk a felhasználóknak átadott üzenetek minőségén is. Így ha például a feldolgozott fájlok számát szeretnénk feltüntetni (angol környezetben) a „0 file(s)", „1 file(s)" és hasonlók kiírása helyett jobban járunk, ha feltételes kifejezéseket alkalmazunk:

```
print("{0} file{1}".format((count if count != 0 else "no"),
("s" if count != 1 else "")))
```

Az eredmény így „no files", „1 file", „2 files" stb. lesz, ami sokkal meggyőzőbb benyomást kelt a programunkról.

Ciklusok

A Python két ciklustípust (`while` és `for...in`) bocsát a rendelkezésünkre, amelyek egyaránt több lehetőséggel rendelkeznek az 1. fejezetben bemutatottaknál.

while ciklusok

A `while` ciklus teljes és általános alakja a következő:

```
while logikai_kifejezés:
    while_kódblokk
else:
    else_kódblokk
```

Az `else` rész megadása nem kötelező. Amíg a *`logikai_kifejezés`* értéke `True`, a rendszer végrehajtja a `while` blokkot – amint az értéke `False` lesz, a ciklus véget ér, és következik az `else` kódblokk végrehajtása, amennyiben jelen van. Ha a rendszer a `while` cikluson belül egy `continue` utasítással találkozik, azonnal visszaugrik a ciklus elejére, és ismét kiértékeli a *`logikai_kifejezés`*-t. Amennyiben a ciklus nem a megszokott módon ér véget, a program kihagyja az `else` kódblokkot.

Az `else` kulcsszó használata meglehetősen félrevezető, ugyanis amennyiben a ciklus rendben véget ér, a program mindenképpen végrehajtja az `else` blokkot. Ez csak akkor marad el, ha a ciklus egy `break` utasítás miatt megszakad, egy `return` utasítással véget ér (amennyiben egy függvény vagy tagfüggvény belsejében van), illetve ha időközben kivétel következik be. (Ez utóbbi esetben a Python kihagyja az `else` kódblokkot, és a megfelelő kivételkezelő után néz – erről hamarosan még bővebben szólunk.) Az `else` kellemes meglepetéseket is tartogat, ugyanis a viselkedése tökéletesen megegyező a `while` és `for...in` ciklusok, valamint a `try...except` kódblokkok esetében.

Lássuk most az `else` működését egy valódi példában! Az `str.index()` és a `list.index()` tagfüggvények megadják egy adott karakterlánc, illetve elem sorszámát, ha pedig nem találják, `ValueError` kivételt váltanak ki. Az `str.find()` ugyanezt teszi, de ha kudarcot vall a keresésben, a kivétel kiváltása helyett a -1 sorszámot adja vissza. A listák esetében nem kapunk ilyen tagfüggvényt, de egy `while` ciklussal kitölthetjük ezt az űrt:

```
def list_find(lst, target):
    index = 0
    while index < len(lst):
        if lst[index] == target:
            break
        index += 1
    else:
        index = -1
    return index
```

A fenti függvény végighalad a listán a megadott elemet keresve. Ha megtalálja, a `break` utasítás megszakítja a ciklust, így a függvény a megfelelő sorszámot adja vissza. Ha nincs találat, a ciklus rendben véget ér, így a rendszer végrehajtja az utána álló `else` utasítást, amely a -1 értéket rendeli a visszaadott `index` változóhoz.

for ciklusok

A `while`-hoz hasonlóan a `for...in` ciklus teljes alakja is tartalmazhat `else` kódblokkot:

```
for kifejezés in bejárható_objektum
    for_kódblokk
else:
    else_kódblokk
```

A kifejezés rendesen egyetlen változó vagy változók (többnyire rendezett) sorozata. Ha rendezett sorozatot vagy listát használunk, ezek elemei a kifejezés elemeibe kerülnek.

Ha a rendszer egy `continue` utasítással találkozik a `for..in` ciklus törzsében, azonnal a blokk elejére ugrik, és kezdetét veszi a ciklus következő ismétlése. Amennyiben a ciklus befejeződik, a rendszer végrehajtja az esetleges `else` kódblokkot – ha azonban megszakad (egy `break` vagy `return` utasítás hatására), a program a ciklust követő utasításra ugrik, kihagyva az `else` blokkot. Hasonlóképpen, ha kivétel keletkezik, a Python átugorja az `else` részt, és megkeresi a megfelelő kivételkezelőt (erről hamarosan bővebben is szólunk).

`enumerate()`
136 ☜

Lássuk most a `list_find()` függvény `for...in` ciklust alkalmazó változatát, amely a korábbiakhoz hasonlóan bemutatja az `else` használatát is:

```
def list_find(lst, target):
    for index, x in enumerate(lst):
        if x == target:
            break
    else:
        index = -1
    return index
```

Amint a fenti kódblokk jól mutatja, a `for...in` ciklus *kifejezés*-ében létrehozott változók tovább élnek azután is, hogy a ciklus befejeződött. Helyi változóként az életük akkor ér véget, amikor a program eléri az őket tartalmazó hatókör határát.

Kivételkezelés

A Python a hibákat, valamint a különleges körülményeket kivételek kiváltásával jelzi, jóllehet egyes külső fejlesztőktől származó Python-könyvtárak ennél régebbi módszereket – például „hiba" értékek visszaadását – is alkalmazhatnak.

Kivételek elfogása és kiváltása

A kivételek elfogására a `try...except` blokk használatos, amelynek az általános felépítése így fest:

```
try:
    try_kódblokk
except kivételcsoport1 as változó1:
    kivétel_kódblokk1
...
except kivételcsoportN as változóN:
    kivétel_kódblokkN
else:
    else_kódblokk
finally:
    finally_kódblokk
```

A szerkezetben szerepelnie kell legalább egy `except` kódblokknak, az `else` és a `finally` blokkok feltüntetése azonban nem kötelező. A program akkor hajtja végre az `else` kódblokkot, ha a `try` blokk kódja rendben lefut – ha azonban kivétel keletkezik, kihagyja. Amennyiben a `finally` blokkot is megadjuk, a rendszer ezt minden esetben végrehajtja, ha a teljes szerkezet végére ér.

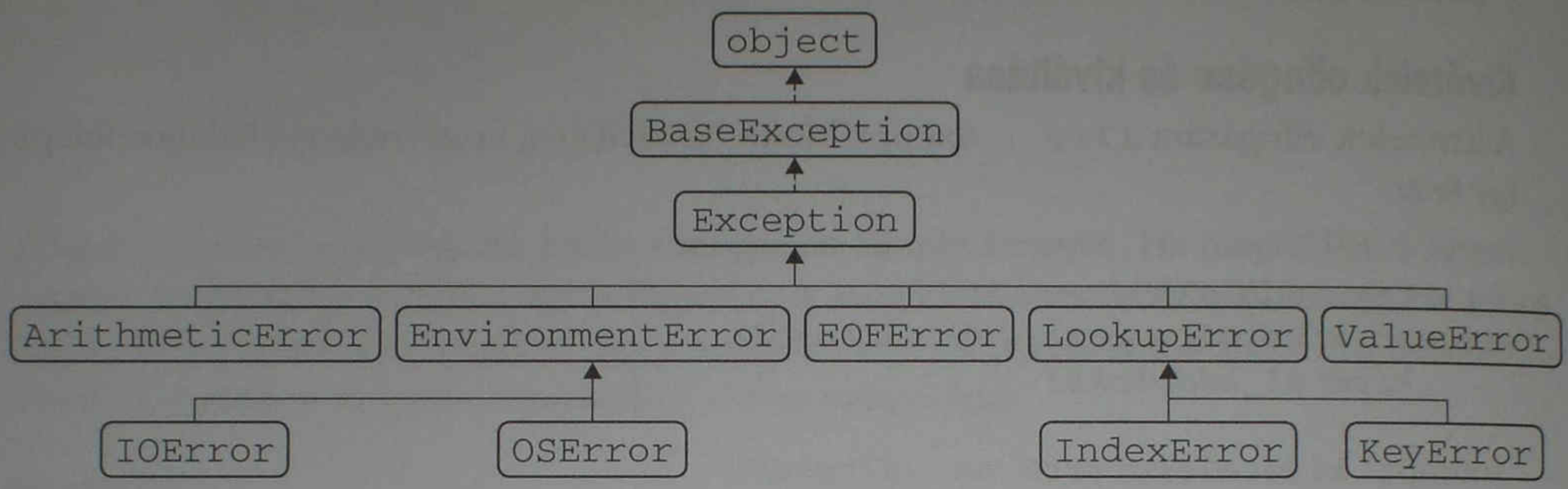
Az `except` utasítások kivételcsoportjaiban szerepelhet egyetlen kivétel, vagy kivételek rendezett sorozata. A változó rész feltüntetése nem kötelező – amennyiben használatba vesszük, ezen keresztül hivatkozhatunk a megfelelő blokkban magára a kivételre.

Amennyiben kivétel következik be a `try` blokkban, a rendszer megvizsgálja az összes általunk megadott `except` utasítást – ha pedig a kivétel megfelel valamelyik kivételcsoportnak, végrehajtja azt a blokkot. Az egyezéshez az szükséges, hogy a kivétel típusa megegyezzen a kivételcsoport (vagy annak valamelyik eleme) típusával, illetve e típusok valamelyikének egy alosztályával.*

Ha például a `KeyError` kivétel egy szótárkeresés során jelenik meg, az első `except` blokk, amely az `Exception` osztályra hivatkozik, egyezést ad, hiszen a `KeyError` az `Exception` (közvetett) leszármazottja. Ha egyetlen csoportban sem bukkan fel az `Exception` (általában

* Amint a 6. fejezetben majd láthatjuk, az objektumközpontú programozásban gyakran találkozhatunk osztályhierarchiákkal, amelyekben egyes osztályok – adattípusok – mások leszármazottai. A Pythonban a hierarchia csúcsán az `object` osztály áll; minden egyéb ennek közvetett vagy közvetlen leszármazottja. Az alosztályok jellemzője, hogy más osztályok leszármazottai, így (az `object` kivételével) a Pythonban minden osztály alosztály, lévén az `object` leszármazottja.

ez a helyzet), valamelyikükben azonban szerepel a `LookupError`, a `KeyError` itt egyezést ad, hiszen alosztálya a `LookupError`-nak. Ha pedig egyetlen csoport sem sorolja fel a `LookupError` vagy az `Exception` típust, de valahol megjelenik maga a `KeyError`, itt kapunk egyezést. A 4.1. ábrán a kivételhierarchia egy részletét láthatjuk.



4.1. ábra

Részlet a Python kivételhierarchiájából

Íme egy rövid példa a kivételkezelési szerkezet hibás használatára:

```
try:
    x = d[5]
except LookupError: # HIBÁS SORREND
    print("Lookup error occurred")
except KeyError:
    print("Invalid key used")
```

Figyelem! *Ha a* `d` *szótár nem rendelkezik olyan elemmel, amelyhez az 5 kulcs tartozik, a lehető legpontosabb kivételt (ez a* `KeyError`*) szeretnénk kiváltani, nem pedig az általánosabb* `LookupError`*-t. Esetünkben azonban a program soha nem éri el a* `KeyError` *blokkját. Ha a rendszer egy* `KeyError` *kivételt vált ki, a* `LookupError` *blokkja egyezést ad, hiszen a* `LookupError` *alaposztálya a* `KeyError`*-nak, vagyis felette jelenik meg az osztályhierarchiában. Ha tehát több* `except` *blokkot is feltüntetünk, a kivételeket mindig a legpontosabbtól (amely a hierarchiában legalul található) a legáltalánosabb felé (amely a hierarchiában legfelül áll) haladva soroljuk fel.*

```
try:
    x = d[k / n]
except Exception: # HELYTELEN GYAKORLAT
    print("Something happened")
```

Figyelem! *Fontos megjegyeznünk, hogy az* `except Exception` *utasítás használata általában ellenjavallt, ugyanis minden kivételt elfog, így könnyen elrejtheti a kódban maradt logikai hibákat. A példánkban lehet, hogy a* `KeyError` *kivételeket szerettük volna elfogni, de ha az* `n` *értéke 0, akaratlanul – és észrevétlenül – elfogjuk a* `ZeroDivisionError` *kivételt is.*

Figyelem! *Használhatjuk az* `except:` *alakot is – ilyenkor nem adunk meg egyetlen kivételcsoportot sem. Az ilyen* `except` *blokk minden olyan kivételt elfog, amely a* `BaseException` *leszármazottja, de az* `Exception` *osztályé nem (ezek a kivételek nem szerepelnek a 4.1. ábrán). Itt hasonló vagy még komolyabb gondoknak nézünk elébe, mint az* `except Exception` *esetében, így ha tehetjük, tartózkodjunk e szerkezet használatától.*

Ha egyik `except` blokk sem képes elfogni a kivételt, a Python elindul felfelé a hívásveremben, megfelelő kivételkezelőt keresve. Ha így sem jár sikerrel, a program futása véget ér, és a rendszer kiírja a kivételt, valamint a visszakövetést a konzolra.

Ha nem következik be kivétel, a program végrehajtja az `else` kódblokkot (amennyiben megadtuk). Végezetül pedig, akármi is történjék – vagyis ha nem következik be kivétel; ha bekövetkezik, és egy `except` blokk kezeli; vagy ha bekövetkezik, és a program továbbküldi felfelé a hívásveremben – a rendszer végrehajtja a `finally` blokkot, amennyiben rendelkezésre áll. Ha nem következik be kivétel; vagy ha bekövetkezik, és egy `except` blokk kezeli, a `finally` blokk végrehajtása a kivételkezelés végén kerül sorra. Ha azonban nincs egyezés, a rendszer csak a `finally` kódblokk futtatása után adja tovább a kivételt a hívásverem felsőbb szintjei felé. Ezzel minden körülmények között biztosítjuk a `finally` blokk lefutását, ami nagy segítség az erőforrások felszabadításának biztosításához. A 4.2. ábrán bemutatjuk, miként hajt végre a rendszer egy `try...except...finally` szerkezetet.

Normál lefutás

```
try:
    # feldolgozás
except kivétel:
    # kezelés
finally:
    # tisztogatás
# folytasd itt
```

Kezelt kivétel

```
try:
    # feldolgozás
except kivétel:
    # kezelés
finally:
    # tisztogatás
# folytasd itt
```

Kezeletlen kivétel

```
try:
    # feldolgozás
except kivétel:
    # kezelés
finally:
    # tisztogatás
# menj fel a hívásveremben
```

4.2. ábra

A try...except...finally szerkezet végrehajtási útjai

Íme a `list_find()` függvény végső változata, amely ezúttal már kivételkezelést alkalmaz:

```
def list_find(lst, target):
    try:
        index = lst.index(target)
    except ValueError:
        index = -1
    return index
```

Ebben a kódblokkban valóban kihasználtuk a `try...except` szerkezet lehetőségeit, hiszen így a kivételt visszatérési értékké alakítottuk. Ugyanezzel a módszerrel megtehetjük azt is, hogy elfogunk egy kivételt, és egy másik típusút váltunk ki – erről hamarosan bővebben is szólunk.

A Python lehetővé teszi az egyszerű `try...finally` szerkezet alkalmazását is, amely olykor hasznos lehet:

```
try:
    try_kódblokk
finally:
    finally_kódblokk
```

Akármi is történjék a `try` blokkban (a számítógép, illetve a program összeomlása kivételével), a rendszer mindenképpen végrehajtja a `finally` blokkot. A `try...finally` szerkezet helyettesíthető egy megfelelő környezetkezelővel együtt alkalmazott `with` utasítással (mindkettőről a 8. fejezetben beszélünk bővebben).

A `try...except...finally` szerkezet alkalmazásának egyik gyakori példája a fájlhibák kezelése. Vegyük például a `noblanks.py` nevű programot, amely fájlnevek listáját olvassa be a parancssorról, és mindegyikükhöz létrehoz egy azonos nevű, `.nb` kiterjesztésű fájlt, amelynek a tartalma csak annyiban tér el az eredetitől, hogy nincsenek benne üres sorok. Íme a program `read_data()` függvénye:

```
def read_data(filename):
    lines = []
    fh = None
    try:
        fh = open(filename, encoding="utf8")
        for line in fh:
            if line.strip():
                lines.append(line)
    except (IOError, OSError) as err:
        print(err)
        return []
    finally:
        if fh is not None:
            fh.close()
    return lines
```

Az `fh` nevű fájlobjektum értékét `None`-ra állítjuk, ugyanis lehet, hogy az `open()` hívás kudarcot vall, így semmit nem rendelünk az `fh` változóhoz (vagyis az értéke `None` marad), és egy kivételt váltunk ki. Ha az általunk meghatározott kivételek valamelyike következik be (`IOError` vagy `OSError`), a hibaüzenet kiírása után egy üres listát adunk vissza. Vegyük azonban észre, hogy a `finally` blokkot még a visszatérés *előtt* végrehajtjuk, így a fájlt is biztosan bezárjuk – amennyiben korábban sikeresen megnyitottuk.

Fontos észrevennünk azt is, hogy ha kódolási hiba következik be, a fájlt még akkor is be tudjuk zárni, ha nem fogjuk el a megfelelő kivételt (`ValueError`). Ilyen esetekben a rendszer végrehajtja a `finally` blokkot, majd felküldi a kivételt a hívásveremben – ezúttal nincs visszatérési érték, hiszen a függvény futása egy kezeletlen kivétel miatt ért véget. Programunk, mivel nem rendelkezik megfelelő `except` blokkal a kódolási hibák kivételeinek elfogására, megszakítja a futását, és visszakövetési üzenetet ír ki.

Az `except` blokkot kicsit szűkszavúbban is írhattuk volna:

```
except EnvironmentError as err:
    print(err)
    return []
```

Ezt azért tehetjük meg, mert az `EnvironmentError` alaposztálya mind az `IOError`, mind az `OSError` osztályoknak.

Környezetkezelők
☞ 375

A 8. fejezetben bemutatunk egy némiképp tömörebb alakú megoldást is a fájlok bezárásának biztosítására, amiben nincs szükség `finally` blokkra.

Kivételek kiváltása

A kivételek nagyszerű lehetőséget adnak a program menetének befolyásolására. Ezt kihasználhatjuk beépített vagy egyedi kivételek alkalmazásával, amelyeket egyaránt kiválthatunk. A kivételek kiváltását kétféle alakban végezhetjük el:

```
raise kivétel(paraméterek)
raise
```

Az első alaknál alkalmazhatunk beépített vagy az `Exception` osztályból származó egyedi kivételt. Ha valamilyen szöveget is feltüntetünk a kivétel paramétereként, a rendszer ezt a képernyőre írja a kivétel elfogásakor. A második alak használatánál – vagyis amikor nem adunk meg kivételt – a `raise` újra kiváltja az éppen aktív kivételt, ha pedig nincs ilyen, a `TypeError`-t választja.

Egyedi kivételek

Az egyedi kivételek valójában egyedi adattípusok (osztályok). Az osztályok létrehozásáról a 6. fejezetben szólunk bővebben, de mivel az egyszerű egyedi kivételtípusok létrehozása nem okoz különösebb nehézséget, itt is bemutatjuk a szükséges alakot:

```
class kivételNeve (szülőKivétel): pass
```

Az alaposztály (szülőosztály) kizárólag az `Exception` vagy annak leszármazottja lehet.

Az egyedi kivételek egyik gyakori alkalmazása, amikor ki szeretnénk ugrani a mélyen egymásba ágyazott ciklusokból. Tegyük fel például, hogy van egy `table` objektumunk, amely bejegyzéseket (`record`) tartalmaz, amelyekben mezők (`field`) állnak, ezek pedig többféle értéket (`item`) vehetnek fel. Egy adott érték után az alábbi kóddal kutathatunk:

```
found = False
for row, record in enumerate(table):
    for column, field in enumerate(record):
        for index, item in enumerate(field):
            if item == target:
                found = True
                break
        if found:
            break
if found:
    print("found at ({0}, {1}, {2})".format(row, column, index))
else:
    print("not found")
```

Ezt a 15 sornyi kódot meglehetősen bonyolulttá teszi, hogy mind a három ciklusból ki kell törnünk a `break` utasítással. Alternatív megoldásként használhatunk egyedi kivételt is:

```
class FoundException(Exception): pass
try:
    for row, record in enumerate(table):
        for column, field in enumerate(record):
            for index, item in enumerate(field):
                if item == target:
                    raise FoundException()
except FoundException:
    print("found at ({0}, {1}, {2})".format(row, column, index))
else:
    print("not found")
```

A kód hossza így 10-11 sorra mérséklődik (attól függően, hogy beleszámítjuk-e a kivétel meghatározását), és sokkal könnyebben áttekinthető, mint az elődje. Ha megtaláljuk a keresett elemet, kiváltjuk az egyedi kivételt, így az `except` blokk végrehajtása következik, az `else` kódblokkét pedig kihagyjuk. Ha azonban nem találjuk az elemet, nem váltunk ki kivételt, így végül a program végrehajtja az `else` blokkot.

Figyeljük meg most egy újabb példán, hogy milyen egyéb módjai lehetnek a kivételkezelésnek. Minden kódblokk a `checktags.py` programból származik, amely beolvassa a parancssorban megadott HTML fájlokat, és elvégez néhány egyszerű vizsgálatot, meggyőződve arról, hogy a < karakterrel kezdődő címkék > karakterrel végződnek, és az egységek megfelelő alakban állnak. A program négy egyedi kivételt határoz meg:

```
class InvalidEntityError(Exception): pass
class InvalidNumericEntityError(InvalidEntityError): pass
class InvalidAlphaEntityError(InvalidEntityError): pass
class InvalidTagContentError(Exception): pass
```

A második és a harmadik az első leszármazottja – a későbbiekben, a kivételkezelő kód tárgyalásánál látjuk majd, miért is hasznos ez. A kivételeket alkalmazó `parse()` függvény több mint 70 sor hosszú, így csak a kivételkezelésben érintett részeit mutatjuk be:

```
fh = None
try:
    fh = open(filename, encoding="utf8")
    errors = False
    for lino, line in enumerate(fh, start=1):
        for column, c in enumerate(line, start=1):
            try:
```

A kód hagyományosan indul, `None` értékre állítva egy fájlobjektumot, és a fájlkezelés kódját egy `try` blokkba helyezve. A program egymás után beolvassa a fájl sorait, a sorok tartalmához pedig karakterenként jut hozzá.

Vegyük észre, hogy a kódban két `try` blokkot is alkalmazunk – a külső kezeli a fájlobjektum kivételeit, a belső pedig az értelmezésnél (feldolgozásnál) megjelenő kivételeket.

```
...
elif state == PARSING_ENTITY:
    if c == ";":
        if entity.startswith("#"):
            if frozenset(entity[1:]) - HEXDIGITS:
                raise InvalidNumericEntityError()
            elif not entity.isalpha():
                raise InvalidAlphaEntityError()
...
```

A függvény különféle állapotokkal rendelkezik. Így például egy & jel beolvasása után `PARSING_ENTITY` állapotba kerül, és az `entity` karakterláncban tárolja az & jel és a pontosvessző közé eső karaktereket.

set type 115 ☜

A kód itt bemutatott része azt az esetet kezeli, amikor egy egység olvasása közben egy pontosvesszővel találkozunk. Amennyiben az egység számértékű (vagyis a kezdetét a &# jelöli, majd hexadecimális számjegyek, végül pedig egy pontosvessző áll, például: &#20AC;), halmazzá alakítjuk, amelyből elvesszük a hexadecimális számjegyeket – ha bármi bent marad a halmazban, az azt jelenti, hogy legalább egy érvénytelen karakter szerepel az egységben, ezért egyedi kivételt váltunk ki. Ha az egység alfabetikus (vagyis a kezdetét a & jelöli, majd betűk, végül pedig egy pontosvessző áll, például: ©), kivételt váltunk ki, amennyiben találunk legalább egy nem alfabetikus karaktert a betűk között.

```
...
except (InvalidEntityError,
        InvalidTagContentError) as err:
    if isinstance(err, InvalidNumericEntityError):
        error = "invalid numeric entity"
```

```
        elif isinstance(err, InvalidAlphaEntityError):
            error = "invalid alphabetic entity"

        elif isinstance(err, InvalidTagContentError):
            error = "invalid tag"
        print("ERROR {0} in {1} on line {2} column {3}"
              .format(error, filename, lino, column))
        if skip_on_first_error:
            raise
...
```

isinstance()
☞ 241

Ha értelmezési kivétel keletkezett, ebben az `except` blokkban fogjuk el. Az `InvalidEntityError` szülőosztály alkalmazásával egyszerre elfoghatjuk az `InvalidNumericEntityError` és az `InvalidAlphaEntityError` kivételeket. Ezek után az `isinstance()` segítségével megvizsgáljuk, hogy valójában melyikről van szó, és ennek megfelelően írjuk ki a hibaüzenetet. A beépített `isinstance()` függvény visszatérési értéke `True`, ha az első paraméterének a típusa megegyezik a másodikéval (vagy annak valamelyik alaptípusával).

Persze létrehozhattunk volna külön `except` blokkot is a három értelmezési kivétel számára, de ezzel a megoldással elkerültük az utolsó négy sor (a `print()` hívástól a `raise` utasításig) felesleges megismétlését.

A program két felhasználási móddal rendelkezik. Ha a `skip_on_first_error` értéke `False`, a program tovább vizsgálja a fájlt akkor is, ha már találkozott értelmezési hibával – így egy fájl esetében több hibaüzenetet is kaphatunk. Amennyiben a `skip_on_first_error` értéke `True`, és a program értelmezési hibát talál, kiírja a megfelelő hibaüzenetet, a `raise` segítségével újra kiváltja az értelmezési kivételt, és a külső (fájlonkénti) `try` blokkra bízza az elfogását.

```
...
elif state == PARSING_ENTITY:
raise EOFError("missing ';' at end of " + filename)
```

A fájl értelmezésének befejezésekor ellenőriznünk kell, hogy nem maradtunk-e egy egység kellős közepén. Amennyiben ez a helyzet, a fájl végét jelző beépített `EOFError` kivételt váltjuk ki, de a hibaüzenetet magunk adjuk meg. Persze nem lenne nehezebb saját kivételt kiváltani sem:

```
except (InvalidEntityError, InvalidTagContentError):
    pass # Already handled
except EOFError as err:
    print("ERROR unexpected EOF:", err)
except EnvironmentError as err:
    print(err)
finally:
    if fh is not None:
        fh.close()
```

A külső `try` blokk esetében önálló `except` kódblokkokat használtunk, hiszen mindegyiküknél eltérő viselkedésre van szükségünk. Ha értelmezési kivétellel találkozunk, tudhatjuk, hogy már kiírtunk egy hibaüzenetet, így csak arra van szükségünk, hogy abbahagyjuk a fájl olvasását, és továbblépjünk a következőre – ezért hát semmit sem kell tennünk a kivételkezelőben. Ha `EOFError` kivételhez jutunk, ez jelentheti azt, hogy valóban túl korán elértük a fájl végét, de az is előfordulhat, hogy magunk váltottuk ki a kivételt. Akár így, akár úgy, kiírunk egy hibaüzenetet, valamint a kivétel szövegét. Ha `EnvironmentError` (vagyis `IOError` vagy `OSError`) kivétel lép fel, egyszerűen kiírjuk a hozzá tartozó szöveget. Végül pedig, akármi is történt a korábbiakban, bezárjuk a fájlt – amennyiben megnyitottuk.

Egyedi függvények

Függvények segítségével egységbe zárhatjuk és paraméterekkel láthatjuk el a programjaink lehetőségeit. A Pythonban négyféle típusukat használhatjuk – léteznek globális függvények, helyi függvények, lambdafüggvények és tagfüggvények.

Az eddig elkészített függvényeink mind-mind *globálisak* voltak. A globális objektumok (beleértve a függvényeket is) bármely kódblokk számára elérhetők, amely ugyanabban a modulban (vagyis ugyanabban a `.py` fájlban) található, amelyben az adott objektumot létrehozták – sőt, amint a következő fejezetben láthatjuk majd, a globális objektumok más modulokból is elérhetők.

A *helyi* (vagy beágyazott) függvények jellemzője, hogy más függvényeken belül határozzák meg őket. Így csak e függvény belsejében érhetők el, ami igen hasznossá teszi őket az olyan apró segédfüggvények készítésénél, amelyeket másutt nem használunk. Először a 7. fejezetben találkozhatunk majd velük.

Elektronikus dokumentáció

Jóllehet könyvünk meglehetősen részletes leírást ad a Python 3 nyelvről, beépített függvényeiről, valamint az alapkönyvtár leggyakrabban használt moduljairól, a Python elektronikus dokumentációja mégis hasznunkra lehet, hiszen részletes leírást ad a nyelvről, különös tekintettel a Python kiterjedt alapkönyvtárának bemutatására. A dokumentációt elérhetjük a `docs.python.org` címen, de megkapjuk magával a nyelvvel is.

Windowson a dokumentációhoz a Windows súgóformátumában jutunk hozzá. A Start, All Programs, Python 3.x, Python Manuals (Start, Minden program, Python 3.x, Python Manuals) menüpontra kattintva megnyithatjuk a súgóböngészőt – benne az áhított dokumentációval. Munkánkat nagyban segíti a keresés lehetősége és a tárgymutató. A Unix-felhasználók számára a segítség HTML formátumban érkezik, amelyben a hivatkozások mellett különféle mutatókat kapunk a könnyebb eligazodás érdekében. A böngészőben a bal oldalon elérhetjük továbbá a Quick Search (Gyors keresés) lehetőséget, ami szintén nagy segítséget jelenthet a keresésben.

Az új felhasználók leggyakrabban a Library Reference oldalon keresgélnek, a tapasztaltabbak pedig jobbára a Global Module Indexben. Mindkettőben találhatunk hivatkozásokat olyan oldalakra, amelyek lefedik a Python teljes alapkönyvtárát – a Library Reference esetében pedig azokat az oldalakat is elérhetjük, amelyek a Python beépített lehetőségeit tárgyalják.

Érdemes átfutnunk a dokumentációt, különösképpen a Library Reference-t, illetve a Global Module Indexet, így képet kaphatunk arról, hogy mit kínál a Python alapkönyvtára, ha pedig egy témakör felkelti az érdeklődésünket, részletesebben is foglalkozhatunk vele. Így némi áttekintést kapunk a Python lehetőségeiről, és nagyjából tudjuk majd, merre kell keresgélnünk, ha valamilyen segítségre lesz szükségünk a későbbiekben. (A Python alapkönyvtárának rövid összefoglaló leírását megtalálhatjuk az 5. fejezetben.)

Segítséget magától az értelmezőtől is kaphatunk. Ha paraméterek nélkül meghívjuk a beépített `help()` függvényt, bekerülünk az elektronikus súgórendszerbe – egyszerűen kövessük a program útmutatásait, és könnyedén eljuthatunk a keresett tartalomhoz. Ha ki szeretnénk lépni, a „q” vagy a „quit” begépelésével megtehetjük, és így visszajutunk az értelmezőhöz. Ha pontosan tudjuk, hogy melyik adattípusról vagy modulról szeretnénk tájékoztatást kapni, csak adjuk meg a nevét a `help()` paramétereként, és máris célt érünk. Így a `help(str)` például a `str` adattípusról ad leírást, minden tagfüggvényével együtt, a `help(dict.update)` a `dict` gyűjteménytípus `update()` tagfüggvényét mutatja be, a `help(os)` pedig az `os` modulról ad tájékoztatást (amennyiben betöltöttük).

Ha már otthonosan mozgunk a Pythonban, gyakran elég, ha csak emlékeztetőt kapunk arról, hogy milyen tulajdonságai (tagfüggvényei) vannak az adott adattípusnak. Erre szolgál a `dir()` függvény – így például a `dir(str)` felsorol minden karakterlánc-kezelő tagfüggvényt, a `dir(os)` pedig megadja az `os` modul állandóit és függvényeit (ismét csak feltéve, hogy betöltöttük a modult).

A *lambdafüggvények* valójában kifejezések, így a használatuk helyén létrehozhatók – a képességeik azonban jelentősen korlátozottak a rendes függvényekhez képest.

A *tagfüggvények* (method) olyan függvények, amelyek valamilyen adattípushoz köthetők, és így csak azzal együtt használhatók – bemutatásukra a 6. fejezetben kerül sor, amikor megismerkedünk az objektumközpontú programozás alapjaival.

A Python számos beépített függvényt bocsát a rendelkezésünkre, az alapkönyvtárban és egyéb külső könyvtárakban pedig több százat (ha a tagfüggvényeket is beleszámítjuk, több ezret) találhatunk, így nem ritka, hogy a céljainknak megfelelő függvényt már megírták. Ezért érdemes gyakran lapozgatni az elektronikus dokumentációt, hátha találunk valami kedvünkre valót (lásd az *Elektronikus dokumentáció* című keretes írásunkat).

A függvények (legyen szó akár globális, akár helyi függvényről) létrehozásának általános alakja a következő:

```
def függvényNeve(paraméterek):
    kódblokk
```

Paraméterek megadása nem kötelező, ha pedig egynél több van belőlük, vesszővel elválasztva soroljuk fel őket azonosítók vagy *azonosító = érték* párok alakjában, amelyekről hamarosan bővebben szólunk. Az alábbi függvény például egy háromszög területét számítja ki a Hérón-képlet alkalmazásával:

```
def heron(a, b, c):
    s = (a + b + c) / 2
    return math.sqrt(s * (s - a) * (s - b) * (s - c))
```

A függvény belsejében a paraméterváltozók – `a`, `b` és `c` – az átadott paraméterértékeket kapják kezdőértékként. A függvény hívásakor meg kell adnunk ezeket a paraméterértékeket, például így: `heron(3, 4, 5)`. Ha túl kevés vagy túl sok paramétert adunk meg, a rendszer `TypeError` kivételt vált ki. Az ilyen hívások esetében *helyzeti paraméterekről* beszélünk, hiszen a paraméterértékek a nekik megfelelő helyzetben álló paraméterbe kerülnek. Esetünkben ez azt jelenti, hogy a függvény hívásakor az `a` paraméter a 3, a `b` a 4, a `c` pedig az 5 értéket kapja.

A Pythonban minden függvény rendelkezik visszatérési értékkel, jóllehet ennek figyelmen kívül hagyása elfogadható (és igen gyakran alkalmazott) gyakorlat. A visszatérési érték lehet önálló érték vagy értékek rendezett sorozata, és mivel gyűjteményeket is visszaadhatunk, gyakorlatilag semmiféle korlátozás nem érvényes a visszatérési értékekre nézve. A függvény törzsét bármely ponton elhagyhatjuk egy `return` utasítással. Amennyiben ezt paraméterek nélkül tüntetjük fel, vagy egyáltalán nem használunk `return` utasítást, a függvény visszatérési értéke `None` lesz. (A 6. fejezetben bemutatjuk a `yield` utasítást, amely bizonyos típusú függvényeknél a `return` helyett használható.)

Egyes függvények olyan paraméterekkel rendelkeznek, amelyeknél érdemes alapértelmezett értékeket meghatároznunk. Íme egy függvény, amely megszámolja a megadott karakterlánc betűit – alapértelmezés szerint az ASCII-betűket véve figyelembe:

```
def letter_count(text, letters=string.ascii_letters):
    letters = frozenset(letters)
    count = 0
    for char in text:
        if char in letters:
            count += 1
    return count
```

A `letters` paraméter alapértelmezett értékét a *paraméter=alapértelmezés* alakban határoztuk meg. Mindez lehetővé teszi, hogy a `letter_count()` függvényt egyetlen paraméterrel hívjuk meg, például így: `letter_count("Maggie and Hopey")`. Ez esetben

a függvény törzsében a `letters` értéke az alapértelmezésként megadott karakterlánc lesz. Ha azonban úgy tartja kedvünk, eltérhetünk az alapértelmezéstől egy helyzeti paraméter értékének feltüntetésével (`letter_count("Maggie and Hopey", "aeiouAEIOU")`) vagy egy kulcsszó-paraméterrel (`letter_count("Maggie and Hopey", letters="aeiouAEIOU")`, lásd a következőkben).

A paraméterek használati alakja nem teszi lehetővé, hogy az alapértelmezett értékkel rendelkező paraméterek után olyanokat is feltüntessünk, amelyeknél nem határoztunk meg alapértelmezést – így a `def rossz(a, b=1, c):` hibás alak. Ugyanakkor a paraméterek átadásánál nem kell feltétlenül követnünk a meghatározás sorrendjét, hiszen alkalmazhatunk *kulcsszó-paramétereket*, az értékeket `név=érték` alakban átadva.

Íme egy aprócska függvény, amely visszaadja az átadott karakterláncot, amennyiben az rövidebb egy megadott értéknél, egyébként pedig csonkolja, és egy jelzőt biggyeszt a végére:

```
def shorten(text, length=25, indicator="..."):
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    return text
```

Lássunk pár példát a használatára:

```
shorten("The Road")                       # visszatérési érték: 'The Road'
shorten(length=7, text="The Road")        # visszatérési érték: 'The ...'
shorten("The Road", indicator="&", length=7) # visszatérési érték:
                                          # 'The Ro&'
shorten("The Road", 7, "&")               # visszatérési érték: 'The Ro&'
```

Mivel mind a `length`, mind az `indicator` rendelkezik alapértelmezett értékkel, egyenként vagy egyszerre is elhagyhatók – ez utóbbira került sor az első hívásnál. A második esetben két kulcsszó-paramétert használtunk, így a sorrendjük tetszőleges lehet. A harmadik hívásban keverednek a helyzeti és a kulcsszó-paraméterek – az első helyzeti, míg a második kettő kulcsszó-paraméter (fontos, hogy a helyzeti paraméterek mindig megelőzzék a kulcsszó-paramétereket). A negyedik hívásban egyszerű helyzeti paramétereket alkalmaztunk.

Szövegfájlok olvasása és írása 126

A kötelező és a nem kötelező paraméterek közötti különbség igen egyszerűen megmagyarázható – ha meghatározzuk a paraméter alapértelmezett értékét, a megadása a hívásban nem kötelező (hiszen a Python „vészhelyzetben” tudja, hogy mit tegyen), ha azonban nem határozunk meg alapértelmezett értéket, mindenképpen meg kell adnunk a paramétert a hívásban (a Python önmagában tanácstalan). Az alapértelmezett értékek körültekintő használatával egyszerűsíthetjük a kódot, és átláthatóbbá tehetjük a függvényhívásokat. Emlékezzünk vissza, hogy a beépített `open()` függvény egyetlen kötelező paraméterrel

(a fájlnév), valamint hat nem kötelezővel rendelkezik. A helyzeti és a kulcsszó-paraméterek együttes használatával megadhatjuk azokat a paramétereket, amelyek lényegesek a számunkra, a többiről pedig egyszerűen elfeledkezhetünk. Mindennek eredményeképpen nyugodtan alkalmazhatjuk az `open(filename, encoding="utf8")` hívást ahelyett, hogy az `open(filename, "r", None, "utf8", None, None, True)` alakot kellene használnunk. A kulcsszó-paraméterek használatának lehetősége még egy előnnyel jár: számottevően olvashatóbbá teszi a függvényhívásokat, különösképpen logikai paraméterek esetében.

Figyelem!

Ha alapértelmezett értékeket alkalmazunk, a rendszer ezeket a `def` *utasítás végrehajtásakor hozza létre (vagyis a függvény létrehozása pillanatában), nem pedig akkor, amikor meghívjuk a függvényt. Állandó értékű (immutable) paraméterek – mint a számok vagy a karakterláncok – használata esetén ennek nincs semmi jelentősége, de változtatható (mutable) paraméterek esetén egy alattomos csapdába sétálhatunk:*

```
def append_if_even(x, lst=[]): # HIBÁS!
    if x % 2 == 0:
        lst.append(x)
    return lst
```

A függvény létrehozásakor az `lst` paramétert a rendszer egy új listához rendeli hozzá. Ahányszor csak meghívjuk a függvényt, mindössze az első paramétert adva meg, az alapértelmezett lista az lesz, amelyet a rendszer ugyanakkor hozott létre, mint a függvényt – nem kapunk újabb és újabb üres listát. Nos, valószínűleg nem ez a viselkedés volt a szívünk vágya: minden bizonnyal új, üres listát szerettünk volna kapni minden olyan hívásnál, ahol a második paraméter nincs jelen. Íme a függvény új változata, amely ezúttal az állandó értékű paramétereknél helyesen viselkedő alakot használja:

```
def append_if_even(x, lst=None):
    if lst is None:
        lst = []
    if x % 2 == 0:
        lst.append(x)
    return lst
```

Ezúttal minden alkalommal új listát kapunk, ahányszor csak a második paraméter nélkül hívjuk meg a függvényt. Ha pedig megadjuk a listát, ugyanúgy használhatjuk, mint a függvény korábbi változatában. A `None` alapértelmezett érték használata és az új objektum létrehozása a követendő eljárás a szótárak, listák, halmazok és egyéb változtatható adattípusok használata esetén, amelyeket alapértelmezett paraméterértékként szeretnénk használni. Lássuk most a függvény kissé rövidebb változatát, amely azonban az előzővel megegyező viselkedést ad:

```
def append_if_even(x, lst=None):
    lst = [] if lst is None else lst
    if x % 2 == 0:
        lst.append(x)
    return lst
```

```
        lst.append(x)
    return lst
```

A feltételes kifejezések használatával minden olyan paraméter esetében megtakaríthatunk egy sort, amelynek az alapértelmezett értéke változtatható.

Nevek és dokumentációs karakterláncok

Érdemes a függvényeinknek és paramétereiknek találó nevet adnunk, így a rendeltetésük és a használatuk könnyebben érthetővé válik más programozók számára, arról nem is beszélve, hogy saját magunknak is jó szolgálatot teszünk ezzel azokra az időkre gondolva, amikor a feledés köde leszáll agyunk barázdált völgyeire. Az alábbiakban néhány olyan alapszabályt sorolunk fel, amelyek megkönnyítik a munkánkat:

- Válasszunk egy elnevezési rendszert, és alkalmazzuk következetesen. Könyvünkben az állandókat *`NAGYBETŰVEL`* írjuk, az osztályokat (ideértve a kivételeket is) *`NagyKezdőbetűkkel`*, a grafikus felület (GUI) függvényeit és tagfüggvényeit *`púposAlakban`*, végül pedig minden egyebet *`kisbetűvel`* vagy *`kisbetűvel_és_aláhúzásokkal`*.
- Bármilyen nevet is adjunk, tartózkodjunk a rövidítések használatától, kivéve, ha egyaránt szabványosak és széles körben használatosak.
- Alkalmazzunk arányos méretű változó- és paraméterneveket: az `x` nagyszerű választás az x-koordináta tárolására, mint ahogy az `i` is tökéletes ciklusváltozó lehet, általában azonban törekedjünk arra, hogy a név elég hosszú legyen ahhoz, hogy leírja az azonosító szerepét. Fontos, hogy az adat jelentését, ne pedig a típusát írjuk le a névvel (vagyis a *`tartozás_összege`* messze alkalmasabb név, mint a *`pénz`*), ha csak a felhasználás nem az adott típus általános tulajdonságait érinti – lásd a `shorten()` példa `text` paraméterét (a következő oldalon).
- A függvényeknek és a tagfüggvényeknek olyan nevet adjunk, amelyek leírják, hogy milyen feladatot végeznek el, illetve mit adnak vissza (attól függően, hogy melyik a hangsúlyosabb) – sose kössük azonban a nevet ahhoz, hogyan végzik el ezt a munkát, hiszen ez az idők során változhat.

Lássunk néhány elnevezési példát:

```
def find(l, s, i=0):                                    # HELYTELEN
def linear_search(l, s, i=0):                           # HELYTELEN
def first_index_of(sorted_name_list, name, start=0):    # HELYES
```

Mindhárom függvény egy adott név első előfordulásának sorszámát adja vissza egy neveket tartalmazó listában, megadott kezdősorszámtól indulva, és olyan algoritmust alkalmazva, ami eleve feltételezi, hogy a lista rendezett.

Az első meghatározás azért helytelen, mert a név nem ad semmiféle támpontot arra nézve, hogy mit is találunk majd meg, a paraméterek pedig (valószínűsíthetően) az elvárt típusokat (`list`, `string`, `integer`) jelzik, de semmi nem utal a valódi jelentésükre. A második meghatározást azért nem tartjuk jónak, mert a függvény neve az eredetileg használt algoritmusra utal, ami azóta megváltozhatott. Mindez persze nem okoz gondot a függvény felhasználói számára, a fejlesztőket azonban összezavarhatja, hiszen lineáris keresésre utal, holott esetleg valójában már bináris keresést használ. A harmadik meghatározás helyes, hiszen a függvény neve a visszatérési értékre utal, a paraméterek nevei pedig tisztán mutatják a szerepüket.

Egyik függvény neve sem utal azonban arra, hogy mi történik, ha nem találja a keresett nevet – valamilyen különleges visszatérési értéket kapunk (mint például a -1), vagy kivétel keletkezik? Ezt pedig valahogyan jeleznünk kellene a felhasználóknak.

Tetszőleges függvény kiegészíthető leírással a *dokumentációs karakterláncok* (docstring) segítségével – ezek a `def` sor után, de még a függvény kódja előtt állhatnak. Íme a korábban már látott `shorten()` függvény teljes kódja:

```
def shorten(text, length=25, indicator="..."):
    """Returns text or a truncated copy with the indicator added
    text is any string; length is the maximum length of the returned
    string (including any indicator); indicator is the string added at
    the end to indicate that the text has been shortened
    >>> shorten("The Road")
    'The Road'
    >>> shorten("No Country for Old Men", 20)
    'No Country for Ol...'
    >>> shorten("Cities of the Plain", 15, "*")
    'Cities of the *'
    """
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    return text
```

A programok világában nem különösebben meglepő, ha egy függvény leírása hosszabb magánál a függvénynél. A közmegegyezés szerint a dokumentációs karakterlánc első sora rövid leírást ad a függvényről, ezt egy üres sor követi, majd a teljes leírás, végül pedig néhány való életbeli példa. Az 5. fejezetben láthatjuk majd, miként alkalmazhatjuk a függvényleírásokban szereplő példákat egységtesztek készítésénél.

Paraméterek és paraméterértékek kicsomagolása

Karakterkódolások 86 ☜

Az előző fejezetben láthattuk, miként használhatjuk a sorozatkicsomagoló műveletet (*) helyzeti paraméterek biztosítására. Így ha például egy háromszög területét szeretnénk kiszámítani, és egy listában rendelkezésünkre állnak az oldalhosszak, ezzel a hívással célt érhetünk: `heron(sides[0], sides[1], sides[2])` – a kicsomagoló művelettel azonban a jelentősen egyszerűbb

`heron(*sides)` alakkal is elérhetjük ugyanazt az eredményt. Ha pedig a lista (vagy egyéb sorozat) több elemmel rendelkezik, mint ahány paramétert a függvény fogad, kivághatjuk a megfelelő nagyságú szeletet.

A sorozatkicsomagoló műveletet a függvények paraméterlistájában is alkalmazhatjuk. Ez olyankor hasznos, ha a függvényünkkel változó számú paramétert szeretnénk fogadni. Ilyen az alábbi, `product()` nevezetű függvény, amely kiszámítja a paramétereinek a szorzatát:

```
def product(*args):
    result = 1
    for arg in args:
        result *= arg
    return result
```

A függvény egyetlen paramétere az `args`. A neve elé írt * azt jelzi, hogy a paraméter rendezett sorozatként viselkedik a függvény belsejében – annyi elemmel, ahány helyzeti paramétert megadunk. Íme néhány hívás példaképpen:

```
product(1, 2, 3, 4)          # args == (1, 2, 3, 4); az eredmény: 24
product(5, 3, 8)             # args == (5, 3, 8); az eredmény: 120
product(11)                  # args == (11,); az eredmény: 11
```

A helyzeti paraméterek után állhatnak kulcsszó-paraméterek is. Erre ad példát az alábbi függvény, amely adott kitevőre emeli a paramétereit, majd kiszámítja az eredményként kapott számok összegét:

```
def sum_of_powers(*args, power=1):
    result = 0
    for arg in args:
        result += arg ** power
    return result
```

A függvényt meghívhatjuk kizárólag helyzeti paraméterekkel, mint a `sum_of_powers(1, 3, 5)` esetében, de használhatunk egyszerre helyzeti és kulcsszó-paramétereket is, mint a `sum_of_powers(1, 3, 5, power=2)` hívásnál.

Alkalmazhatjuk továbbá a * műveletet önmagában álló „paraméterként" is. Ezzel azt jelölhetjük, hogy a * után nem állhatnak helyzeti paraméterek, de kulcsszó-paraméterek még megjelenhetnek. Az alábbiakban bemutatjuk a `heron()` függvény egy módosított változatát, amely ezúttal pontosan három helyzeti és egy (nem kötelező) kulcsszó-paramétert fogad.

```
def heron2(a, b, c, *, units="meters"):
    s = (a + b + c) / 2
    area = math.sqrt(s * (s - a) * (s - b) * (s - c))
return "{0} {1}".format(area, units)
```

Lássuk a függvény viselkedését néhány hívás példáján:

```
heron2(25, 24, 7)                    # az eredmény: '84.0 meters'
heron2(41, 9, 40, units="inches")    # az eredmény: '180.0 inches'
heron2(25, 24, 7, "inches")          # HIBÁS! TypeError kivételt vált ki
```

Az utolsó hívásnál megkíséreltünk átadni egy negyedik helyzeti paramétert, de a * ezt nem teszi lehetővé, ami a `TypeError` kivételhez vezet.

Amennyiben a * műveletet az első paraméter helyére írjuk, egyetlen helyzeti paramétert sem használhatunk, így csak kulcsszó-paraméterekkel hívhatjuk meg a függvényt. Íme egy ilyen (képzeletbeli) függvény aláírása:

```
def print_setup(*, paper="Letter", copies=1, color=False):
```

A `print_setup()` függvényt meghívhatjuk paraméterek nélkül, elfogadva az alapértelmezéseket, de módosíthatjuk is valamelyik vagy akár az összes alapértelmezés értékét, mint a `print_setup(paper="A4", color=True)` hívásnál. Ha azonban helyzeti paramétereket próbálunk használni, mint a `print_setup("A4")` esetében, `TypeError` kivételhez jutunk.

Ahogy egy sorozatot kicsomagolhatunk helyzeti paramétereket átadva egy függvénynek, a megfeleltetésekkel is megtehetjük ugyanezt a ** művelet alkalmazásával.* Ezzel egy szótárat adhatunk át a `print_setup()` függvénynek, valahogy így:

```
options = dict(paper="A4", color=True)
print_setup(**options)
```

A rendszer kicsomagolja az `options` szótár kulcs-érték párjait, és az értékeket hozzárendeli a megfelelő nevű paraméterekhez. Amennyiben a szótárban olyan kulcs is szerepel, amelynek nem felel meg egyetlen paraméter sem, `TypeError` kivételhez jutunk. Ha egy paraméter nem szerepel a szótárban, az alapértelmezett értékét veszi fel – ha azonban nincs ilyen alapértelmezés, megint csak `TypeError` kivételhez jutunk.

A megfeleltetés-kicsomagoló műveletet a paraméterek meghatározásánál is használhatjuk – a segítségével olyan függvényeket készíthetünk, amelyek képesek tetszőleges számú kulcsszó-paraméter fogadására. Jó példa erre az alábbi `add_person_details()` nevű függvény, amely az `ssn` és a `surname` helyzeti paraméterek mellett tetszőleges számú kulcsszó-paramétert képes fogadni:

```
def add_person_details(ssn, surname, **kwargs):
print("SSN =", ssn)
print("     surname =", surname)
for key in sorted(kwargs):
    print("     {0} = {1}".format(key, kwargs[key]))
```

* A 2. fejezetben láthattuk, hogy bináris műveletként használva a ** a `pow()` műveletnek felel meg.

A függvényt meghívhatjuk egyszerűen két helyzeti paraméterrel, de kiegészíthetjük a hívást további adatokkal is, mint az `add_person_details(83272171, "Luther", forename="Lexis", age=47)` esetében. Mindez igen jelentős rugalmasságot biztosít, különösen ha azt is észrevesszük, hogy olyan függvényt is meghatározhatunk, amely tetszőleges számú helyzeti és kulcsszó-paramétert képes fogadni:

```
def print_args(*args, **kwargs):
    for i, arg in enumerate(args):
        print("positional argument {0} = {1}".format(i, arg))
    for key in kwargs:
        print("keyword argument {0} = {1}".format(key, kwargs[key]))
```

A függvény nem tesz mást, mint kiírja a paramétereit. Meghívhatjuk akár paraméterek nélkül is, de tetszőleges számú helyzeti és kulcsszó-paramétert is megadhatunk.

Globális hatókörű változók elérése

Olykor jól jöhet, ha van néhány globális változónk, amelyet a programunk különféle függvényei elérhetnek. Ez a módszer általában megfelel az „állandók" esetében, de a változóknál jobbára ellenjavallt – bár rövidke programoknál nem mindig elvetendő módszer.

A `digit_names.py` nevezetű program képes egy nyelvi beállítást („en" vagy „fr", a megadása nem kötelező), valamint egy számot fogadni a parancssorban, majd kiírja a számjegyeket az adott nyelven. Így ha az „123" paramétert adjuk meg a számára a parancssorban, az eredmény a „one two three" karakterlánc lesz. A program három globális változóval rendelkezik:

```
Language = "en"
ENGLISH = {0: "zero", 1: "one", 2: "two", 3: "three", 4: "four",
           5: "five", 6: "six", 7: "seven", 8: "eight", 9: "nine"}
FRENCH = {0: "zéro", 1: "un", 2: "deux", 3: "trois", 4: "quatre",
          5: "cinq", 6: "six", 7: "sept", 8: "huit", 9: "neuf"}
```

A kódban követtük a közmegegyezést, miszerint a teljesen nagybetűs változónevek állandókat takarnak, az alapértelmezett nyelv pedig az angol lett. (A Pythonban közvetlenül nem hozhatunk létre állandókat – változókat használhatunk, amelyek érinthetetlenségét a programozónak kell biztosítania.) A program egy másik részén elérjük a `Language` változót, és az értéke szerint választjuk ki a megfelelő szótárat:

```
def print_digits(digits):
    dictionary = ENGLISH if Language == "en" else FRENCH
    for digit in digits:
        print(dictionary[int(digit)], end=" ")
    print()
```

Amikor a Python találkozik a `Language` változóval, először a helyi (függvény-) hatókörben keresi, de nem találja. A következő lépés a globális hatókör (vagyis a teljes `.py` fájl) vizsgálata, ahol sikerrel jár. Az `end` kulcsszó-paraméter szerepéről, amelyet az első `print()` hívásban alkalmazunk, *A print() függvény* című keretes írásunkban szólunk bővebben.

A print() függvény

A `print()` függvény tetszőleges számú helyzeti és három kulcsszó-paramétert (`sep`, `end` és `file`) fogad. Az utóbbiak mindegyike rendelkezik alapértelmezett értékkel. A `sep` paraméter alapértelmezés szerint a szóköz – ha két vagy több helyzeti paramétert adunk meg, a függvény egymás után írja ki őket, a `sep` karakterláncot helyezve el közöttük. Amennyiben csak egy helyzeti paramétert adunk meg, a `sep` nem kap szerepet az eredményben. Az `end` paraméter alapértelmezés szerint `\n`, ezért kerül a képernyőre egy új sor a `print()` függvény hívása után. A `file` paraméter alapértelmezése a `sys.stdout`, így az eredmény a szabványos kimeneti folyamba, vagyis jobbára a konzolra kerül.

Amennyiben el szeretnénk térni az alapértelmezéstől, mindhárom kulcsszó-paraméter értékét megváltoztathatjuk. Így a `file` értékeként megadhatunk egy fájlobjektumot, amelyet írásra vagy hozzáfűzésre nyitottunk meg, a `sep` és az `end` pedig tetszőleges karakterlánc (akár üres is) lehet.

Ha több elemet szeretnénk ugyanabban a sorban kiírni, gyakori, hogy olyan `print()` utasítások sorozatát alkalmazzuk, amelyekben az `end` az egyes elemeket elválasztó karakterláncot adja meg. Végül pedig paraméterek nélkül hívjuk meg a függvényt, így egyszerűen egy új sort kapunk. Minderre jó példát ad a `print_digits()` függvény (az előző oldalon).

Íme, programunk `main()` függvényének kódja, amely szükség esetén módosítja a `Language` változó értékét, és meghívja a `print_digits()` függvényt az eredmény kiírásához:

```
def main():
    if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
        print("usage: {0} [en|fr] number".format(sys.argv[0]))
        sys.exit()
    args = sys.argv[1:]
    if args[0] in {"en", "fr"}:
        global Language
        Language = args.pop(0)
    print_digits(args.pop(0))
```

A kódban egyetlen újdonság van: a `global` utasítás használata. Ezzel tájékoztathatjuk a Pythont arról, hogy a változó létezik a globális (fájlszintű) hatókörben, és a hozzárendeléseket is ezzel a globális változóval kell elvégezni – nincs szükség egy ilyen nevű helyi változó létrehozására.

Figyelem! *Ha nem használtuk volna a* `global` *utasítást, a program működne ugyan, de amikor a Python eljut a* `Language` *változóhoz az* `if` *utasításban, a helyi (függvénybeli) hatókörben keresné, és mivel nem találná, egy helyi változót hozna létre ezzel a névvel, érintetlenül hagyva a globális* `Language` *értékét.*

Ez az apró tévedés kizárólag akkor vezet hibához, ha a programot az „fr" paraméterrel futtatjuk, ugyanis ilyenkor a program a frissiben létrehozott helyi `Language` *változóhoz rendeli az „fr" értéket, nem pedig a globális megfelelőjéhez. Ez utóbbi, amelyet a* `print_digits()` *függvény elér, továbbra is az „en" értéket tartalmazza.*

A globális változók használata kicsit is összetettebb programok esetén ellenjavallt – az állandók esetét kivéve, ahol azonban nincs szükség a `global` utasításra.

Lambdafüggvények

A lambdafüggvények az alábbi alakban határozhatók meg:

```
lambda paraméterek: kifejezés
```

Generátor függvények ☞ 277, 346

A *paraméterek* rész megadása nem kötelező – amennyiben megadjuk, jobbára egyszerűen vesszőkkel elválasztott változónevekből, vagyis helyzeti paraméterekből áll, jóllehet a `def` utasítás által támogatott teljes alakot is alkalmazhatjuk. A *kifejezés* rész nem tartalmazhat elágazást vagy ciklust (bár feltételes kifejezések szerepelhetnek itt), továbbá nem állhat itt `return` (vagy `yield`) utasítás sem. A `lambda` kifejezés eredménye nem más, mint egy névtelen függvény. A meghívását követően a *kifejezés* kiértékelésének eredményét adja vissza – ha ez egy rendezett sorozat, zárójelek közé kell tennünk.

Íme egy egyszerű lambdafüggvény, amely az *s* értéket adja vissza, kivéve, ha a neki átadott paraméter 1:

```
s = lambda x: "" if x == 1 else "s"
```

A `lambda` kifejezés egy névtelen függvényt ad vissza, amelyet az `s` változóhoz rendelünk. Tetszőleges (hívható) változót meghívhatunk a zárójelek használatával, így ha rendelkezünk például egy művelet során feldolgozott fájlok számával (`count`), az `s()` függvénnyel a következőképpen írhatunk ki egy üzenetet: `print("{0} file{1} processed".format(count, s(count))`.

A lambdafüggvényeket gyakran alkalmazzák a beépített `sorted()` függvény és a `list.sort()` tagfüggvény kulcsfüggvényeként. Tegyük fel, hogy rendelkezünk a kémiai elemek listájával, amelyben az adatok rendezett hármasok (csoport, rendszám, név) alakban szerepelnek, és szeretnénk különböző szempontok szerint rendezni ezt a listát. Íme példaképpen egy ilyen lista:

```
elements = [(2, 12, "Mg"), (1, 11, "Na"), (1, 3, "Li"), (2, 4, "Be")]
```

A lista rendezése nyomán az alábbi eredményt kapjuk:

```
[(1, 3, 'Li'), (1, 11, 'Na'), (2, 4, 'Be'), (2, 12, 'Mg')]
```

sorted()
139

Korábban, a `sorted()` függvény bemutatásakor láthattuk, hogy a rendezés módosításához kulcsfüggvényeket is igénybe vehetünk. Így ha a listát a rendszám és a név alapján szeretnénk rendezni, nem pedig a természetesen adódó csoport, rendszám és név kulcsokkal, írhatunk egy aprócska függvényt (`def ignore0(e): return e[1], e[2]`), amelyet kulcsfüggvényként alkalmazhatunk a céljaink elérésére. Ha sok ilyen kis függvényt kell írnunk, a hagyományos eljárás körülményesnek bizonyulhat – ilyenkor segítenek a lambdafüggvények:

```
elements.sort(key=lambda e: (e[1], e[2]))
```

A kulcsfüggvényünk ez esetben a `lambda e: (e[1], e[2])`, ahol `e` a lista egy rendezett hármasa lehet. A `lambda` kifejezés körüli zárójelekre feltétlenül szükség van, amennyiben a kifejezés rendezett sorozat (tuple), és a lambdafüggvényt egy függvény paramétereként hozzuk létre. Hasonló eredményt érhetünk el szeleteléssel is:

```
elements.sort(key=lambda e: e[1:3])
```

Az alábbi, némiképp kifinomultabb változatban a „név, szám" párok alapján rendezünk, de a nevekben nem különböztetjük meg a kis- és nagybetűket:

```
elements.sort(key=lambda e: (e[2].lower(), e[1]))
```

Lássunk most két egyenértékű módszert egy függvény létrehozására, amely a háromszög területét a hagyományos ½ × alap × magasság képlettel számítja ki:

```
area = lambda b, h: 0.5 * b * h
```

```
def area(b, h):
return 0.5 * b * h
```

Az `area(6, 5)` hívást elvégezve megegyező eredményt kapunk, akár a `def` utasítással, akár a `lambda` kifejezéssel hoztuk létre a függvényt.

A lambdafüggvények további nagyszerű felhasználási területe az alapértelmezett szótárak létrehozása. Emlékezzünk vissza az előző fejezetben olvasottakra, miszerint ha egy alapértelmezett szótárt nem létező kulccsal próbálunk elérni, a rendszer létrehozza a megfelelő elemet a megadott kulccsal és egy alapértelmezett értékkel. Lássunk néhány példát:

```
minus_one_dict = collections.defaultdict(lambda: -1)
point_zero_dict = collections.defaultdict(lambda: (0, 0))
message_dict = collections.defaultdict(lambda: "No message available")
```

Ha a `minus_one_dict` szótárt egy nem létező kulccsal próbáljuk elérni, a rendszer új elemet hoz létre ezzel a kulccsal és a -1 értékkel. Hasonló a helyzet a `point_zero_dict` esetében is, ahol az érték a `(0, 0)` rendezett pár lesz, a `message_dict` esetében pedig a „No message available" karakterlánc.

Feltételezések

Mi történik, ha egy függvény érvénytelen adatokhoz jut a paraméterein keresztül? Mi a helyzet akkor, ha valamit elrontottunk az algoritmus megvalósításánál, és hibás számítást végzünk? Nos, a legrosszabb, ami bekövetkezhet, hogy a program minden (feltűnő) gond nélkül fut tovább, és észrevétlenül elsiklunk a hiba felett. Az egyik módszer az ilyen alattomos csapdák elkerülésére az, ha teszteket írunk – erről röviden szót ejtünk az 5. fejezetben. A másik járható út előfeltételezések és utófeltételezések meghatározása – amennyiben ezek valamelyike nem teljesül, tudhatjuk, hogy hibát találtunk. A legjobb megoldás persze az, ha az elő- és utófeltételezéseket a tesztekkel együtt alkalmazzuk.

Az elő- és utófeltételezéseket az `assert` („állítás") utasítással határozhatjuk meg, a következő alakban:

```
assert logikai_kifejezés, nem_kötelező_kifejezés
```

Amennyiben a *`logikai_kifejezés`* értéke `False`, az `AssertionError` kivételhez jutunk. Ha a *`nem_kötelező_kifejezés`*-t is megadjuk, ez az `AssertionError` kivétel paramétereként kap szerepet – nagyszerű módot adva a hibaüzenetek meghatározására. Fontos megjegyeznünk azonban, hogy a feltételezések kifejezetten a fejlesztők eszközei – semmiképpen se alkalmazzuk őket arra, hogy a végfelhasználóknak jelezzünk. A programok használata során felmerülő gondok – hiányzó fájlok vagy éppen hibás parancssori paraméterek – kezelésére más módot kell találnunk, például hibaüzenetet írhatunk ki, vagy naplóbejegyzést készíthetünk.

Az alábbiakban bemutatjuk a `product()` függvény két új változatát, amelyek egyaránt megkövetelik, hogy egyetlen paraméterük se legyen 0. Ha ilyen hívást észlelnek, egyértelműen kódolási hibának tekintik.

```
def product(*args): # pesszimista
assert all(args), "0 argument"
result = 1
for arg in args:
result *= arg
return result
```

```
def product(*args): # optimista
result = 1
for arg in args:
result *= arg
assert result, "0 argument"
return result
```

A „pesszimista" változat minden hívásnál minden paramétert megvizsgál (az első 0 értékig). Az „optimista" változat ugyanakkor csak az eredményt vizsgálja, hiszen ez kizárólag akkor 0, ha legalább egy paraméter értéke 0 volt.

Ha a fenti `product()` függvények valamelyikét egy 0 paraméterrel hívjuk meg, `AssertionError` kivételt kapunk, és az alábbihoz hasonló üzenet jelenik meg a hibafolyamon (`sys.stderr`; ez jobbára maga a konzol):

```
Traceback (most recent call last):
  File "program.py", line 456, in <module>
    x = product(1, 2, 0, 4, 8)
  File "program.py", line 452, in product
    assert result, "0 argument"
AssertionError: 0 argument
```

A Python automatikusan visszakövetést biztosít, amely tartalmazza a fájlnevet, a függvényt és a sor számát, továbbá az általunk megadott hibaüzenetet.

Mit tegyünk az `assert` utasításokkal, ha a program készen áll a kiadásra (természetesen megfelelt a teszteken, és nem sért egyetlen feltételezést sem)? Nos, ilyenkor utasíthatjuk a Pythont, hogy ne hajtsa végre az `assert` utasításokat – vagyis vesse el őket futásidőben. Erre szolgál a `-O` parancssori kapcsoló, amelyet valahogy így használhatunk: `python -O program.py`. Hasonló eredményt érhetünk el akkor is, ha a `PYTHONOPTIMIZE` környezeti változónak a 0 értéket adjuk. Ha a dokumentációs karakterláncok nem bírnak jelentőséggel a felhasználóink számára (jobbára így áll a helyzet), alkalmazhatjuk a `-OO` kapcsolót is, ami lényegében megszabadít ezektől a karakterláncoktól az `assert` utasításokkal egyetemben. Fontos megjegyeznünk, hogy ezúttal nem létezik olyan környezeti változó, amellyel hasonló eredményt érhetnénk el. Egyes fejlesztők még ennél is egyszerűbb megoldást választanak: másolatot készítenek a programról, amelyben minden `assert` utasítást megjegyzéssé alakítanak, és amennyiben ez átmegy a teszteken, végül egy feltételezésektől mentes változatot bocsátanak forgalomba.

Példa: make_html_skeleton.py

A következőkben egy példaprogramot készítünk, amely jól mutatja számos, a fejezetünkben tárgyalt módszer használatát.

Az igen kicsiny webhelyek esetében gyakori, hogy az elkészítésük és a fenntartásuk egyaránt közvetlenül, kézi munkával történik. Ezt az embert próbáló feladatot némiképp megkönnyíthetjük egy olyan programmal, amely képes elkészíteni a HTML fájlok vázát, amelyet ezután már csak tartalommal kell megtölteni. A `make_html_skeleton.py` éppen erre a feladatra alkalmas: elkéri a felhasználótól a megfelelő adatokat, majd elkészíti a HTML fájl vázát. A program `main()` függvénye egy ciklust tartalmaz, így egymás után több vázat is készíthetünk vele – ráadásul megőriz bizonyos adatokat (például a szerzői jogi kitételeket), így a felhasználóknak ezeket csak egyszer kell megadniuk. Íme egy jellemző munkamenet átirata:

```
make_html_skeleton.py
Make HTML Skeleton
Enter your name (for copyright): Harold Pinter
Enter copyright year [2008]: 2009
Enter filename: career-synopsis
```

```
Enter title: Career Synopsis
Enter description (optional): synopsis of the career of Harold Pinter
Enter a keyword (optional): playwright
Enter a keyword (optional): actor
Enter a keyword (optional): activist
Enter a keyword (optional):
Enter the stylesheet filename (optional): style
Saved skeleton career-synopsis.html

Create another (y/n)? [y]:
Make HTML Skeleton

Enter your name (for copyright) [Harold Pinter]:
Enter copyright year [2009]:
Enter filename:
Cancelled

Create another (y/n)? [y]: n
```

Vegyük észre, hogy a második váz létrehozásához a név és az évszám alapértelmezett értékeit már meghatároztuk a korábbiakban, így ezeket nem kellett újra beírni. A fájlnév esetében azonban nincs ilyen alapértelmezés, ezért amikor ezt nem adtuk meg, a program megszakította a váz elkészítését.

Miután megismerkedtünk a program használatával, ideje, hogy hozzáfogjunk a kód tanulmányozásához. A program két betöltési művelettel veszi kezdetét:

```
import datetime
import xml.sax.saxutils
```

A `datetime` modulban néhány egyszerű függvényt találhatunk, amelyek segítenek a `datetime.date` és a `datetime.time` objektumok létrehozásában. Az `xml.sax.saxutils` modul rendelkezésünkre bocsátja az `xml.sax.saxutils.escape()` függvényt, amely egy karakterláncot fogad, és olyan formátumban adja vissza, amely a HTML-karaktereket (&, < és >) a védett alakjukban (&, < és >) tartalmazza.

Három globális karakterláncot határozunk meg, amelyeket sablonként használunk majd:

```
COPYRIGHT_TEMPLATE = "Copyright (c) {0} {1}. All rights reserved."
STYLESHEET_TEMPLATE = ('<link rel="stylesheet" type="text/css" '
                       'media="all" href="{0}" />\n')
HTML_TEMPLATE = """<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" \
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<head>
<title>{title}</title>
```

```
<!-- {copyright} -->
<meta name="Description" content="{description}" />
<meta name="Keywords" content="{keywords}" />
<meta equiv="content-type" content="text/html; charset=utf-8" />
{stylesheet}\
</head>
<body>

</body>
</html>
"""
```

str.format()
75

Ezeket a karakterláncokat az `str.format()` függvény hívásakor alkalmazzuk majd sablonként. A `HTML_TEMPLATE` esetében a mezőket inkább a nevük, mint a sorszámuk alapján érjük el, mint például a `{title}` esetében. Hamarosan láthatjuk majd, hogy az értékadásukhoz kulcsszó-paramétereket kell használnunk.

```
class CancelledError(Exception): pass
```

Meghatározunk egy egyedi kivételt is – hamarosan használatban is találkozhatunk vele, amint belekezdünk a program függvényeinek vizsgálatába.

Programunk `main()` függvényének szerepe abban áll, hogy meghatároz néhány kezdőértéket, továbbá végrehajt egy ciklust. A felhasználó minden ciklusban lehetőséget kap néhány adat beírására a HTML-oldal elkészítéséhez, ha pedig ezzel elkészült, kiléphet, vagy folytathatja a következő ciklusismétléssel.

```
def main():
    information = dict(name=None, year=datetime.date.today().year,
                       filename=None, title=None, description=None,
                       keywords=None, stylesheet=None)
    while True:
        try:
            print("\nMake HTML Skeleton\n")
            populate_information(information)
            make_html_skeleton(**information)
        except CancelledError:
            print("Cancelled")
        if (get_string("\nCreate another (y/n)?", default="y").lower()
        not in {"y", "yes"}):
            break
```

A `datetime.date.today()` függvény egy olyan `datetime.date` objektumot ad vissza, amely az aktuális nap dátumát tartalmazza. Itt mindössze a `year` (év) tulajdonságra van szükségünk – a többi adat esetében a `None` értéket állítjuk be, hiszen ezeknek nem tudunk ésszerű alapértelmezést adni.

A `while` ciklusban a program először kiír egy címfeliratot, majd meghívja a `populate_information()` függvényt, átadva neki az `information` szótárat, amelyet frissítünk a függvény belsejében. Ezt követi a `make_html_skeleton()` függvény hívása, amely számos paramétert vár. Ahelyett azonban, hogy ezeket egyenként megadnánk, az `information` szótár kicsomagolt tartalmát adjuk át neki.

Amennyiben a felhasználó megszakítja a program futását, például úgy, hogy nem ad meg valamilyen kötelező adatot, a program a „Cancelled" üzenetet írja ki. Az egyes ciklusok végén (akár megszakította a folyamatot a felhasználó, akár nem), a program rákérdez, hogy szeretnénk-e újabb vázat készíteni – amennyiben nem, kilép a ciklusból, és a futása véget ér.

```
def populate_information(information):
    name = get_string("Enter your name (for copyright)", "name",
                      information["name"])
    if not name:
        raise CancelledError()
    year = get_integer("Enter copyright year", "year",
                       information["year"], 2000,
                       datetime.date.today().year + 1, True)
    if year == 0:
        raise CancelledError()
    filename = get_string("Enter filename", "filename")
    if not filename:
        raise CancelledError()
    if not filename.endswith((".htm", ".html")):
        filename += ".html"
    ...
    information.update(name=name, year=year, filename=filename,
                       title=title, description=description,
                       keywords=keywords, stylesheet=stylesheet)
```

A címfelirat és a leíró szövegek, a HTML-kulcsszavak, valamint a stíluslap beolvasásának kódját nem tüntettük fel – azt azonban elárulhatjuk, hogy mindegyikük a `get_string()` függvényt alkalmazza, amelyről hamarosan szót ejtünk. Elegendő annyit tudnunk, hogy a függvény a kiírandó üzenetet, a megfelelő változó „nevét" (ezt a hibaüzenetekben használjuk), valamint az esetleges alapértelmezett értéket várja. Hasonlóan, a `get_integer()` függvény is egy kiírandó üzenetet, egy változónevet, egy alapértelmezett értéket, illetve egy minimum- és maximumértéket vár, valamint azt is megadhatjuk, hogy a 0 érték megengedett-e.

Végezetül, kulcsszó-paraméterek segítségével frissítjük az `information` szótárat az új értékekkel. Az egyes *`kulcs=érték`* párok esetében a *`kulcs`* a szótár azon kulcsának neve, amelyet az adott *`érték`*-kel helyettesítünk – az *`érték`* ugyanakkor ez esetben egy olyan változót jelent, amelynek a neve megegyezik a szótárban található, neki megfelelő kulcséval.

A függvénynek nem adtunk meg visszatérési értéket (így a `None` értéket adja vissza). A futása megszakadhat, amennyiben a `CancelledError` kivétel bekövetkezik – ezt a rendszer továbbküldi felfelé a hívásveremben, és végül a `main()` függvény kezeli.

A `make_html_skeleton()` függvényt két részre bontva mutatjuk be:

```
def make_html_skeleton(year, name, title, description, keywords,
                       stylesheet, filename):
    copyright = COPYRIGHT_TEMPLATE.format(year,
                                          xml.sax.saxutils.escape(name))
    title = xml.sax.saxutils.escape(title)
    description = xml.sax.saxutils.escape(description)
    keywords = ",".join([xml.sax.saxutils.escape(k)
                         for k in keywords]) if keywords else ""
    stylesheet = (STYLESHEET_TEMPLATE.format(stylesheet)
                  if stylesheet else "")
    html = HTML_TEMPLATE.format(title=title, copyright=copyright,
                                description=description,
                                keywords=keywords,
                                stylesheet=stylesheet)
```

A szerzői jogi szöveg elkészítéséhez a `COPYRIGHT_TEMPLATE` sablonra meghívjuk az `str.format()` függvényt, helyzeti paraméterként átadva neki az aktuális évszámot, valamint (a megfelelően levédett) nevet, amelyek a `{0}` és az `{1}` mezők helyére kerülnek. A cím és a leírás eredeti szövegei helyett a levédett változataikat alkalmazzuk.

A HTML-kulcsszavak esetében két esetet különíthetünk el – erre egy feltételes kifejezést alkalmazunk. Amennyiben a felhasználó nem írt be kulcsszót, a `keywords` karakterláncot üressé tesszük a `""` értéket rendelve hozzá. Egyébként a kulcsszavak listáján végighaladva létrehozzuk a levédett változataikat, majd az `str.join()` függvénnyel egyetlen karakterláncban, vesszővel elválasztva soroljuk fel az így kapott kulcsszavakat.

A stíluslapot a szerzői jogi szöveghez hasonlóan készítjük el, ezúttal azonban mindezt egy feltételes kifejezés belsejében tesszük meg, így ha üres karakterláncot kapunk, egyáltalán nem határozunk meg stíluslapot.

`str.format()`
75 ☜

A `html` szöveget a `HTML_TEMPLATE` sablonból készítjük el, most azonban – eltérően más sablonoktól – helyzeti paraméterek helyett kulcsszó-paraméterekkel érjük el a cseremezőket.

```
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        fh.write(html)
    except EnvironmentError as err:
        print("ERROR", err)
```

```
else:
    print("Saved skeleton", filename)
finally:
    if fh is not None:
        fh.close()
```

Miután elkészítettük a HTML-kódot, nem marad más hátra, mint tárolni a megadott nevű fájlban. Tájékoztatjuk a felhasználót, hogy az oldal vázát lemezre mentettük, illetve kiírjuk a megfelelő hibaüzenetet, ha időközben valami rosszul sült el. Végezetül egy `finally` blokkban biztosítjuk, hogy a korábban megnyitott fájlokat be is zárjuk.

```
def get_string(message, name="string", default=None,
               minimum_length=0, maximum_length=80):
    message += ": " if default is None else " [{0}]: ".format(default)
    while True:
        try:
            line = input(message)
            if not line:
                if default is not None:
                    return default
                if minimum_length == 0:
                    return ""
                else:
                    raise ValueError("{0} may not be empty".format(
                                                                name))
            if not (minimum_length <= len(line) <= maximum_length):
                raise ValueError("{0} must have at least {1} and "
                                 "at most {2} characters".format(
                                 name, minimum_length, maximum_length))
            return line
        except ValueError as err:
            print("ERROR", err)
```

A függvénynek egyetlen paraméterét (`message`) kell feltétlenül meghatároznunk, a többi négy megadása nem kötelező. Ha valahol megadtunk egy alapértelmezett értéket, azt elhelyezzük a `message` karakterláncban is, így a felhasználó látja, mi is kerül a kódba, ha szöveg beírása nélkül lenyomja az ENTER billentyűt. A függvény további része egy végtelen ciklus belsejében található, amelyből a felhasználó úgy léphet ki, ha beír egy érvényes karakterláncot, illetve az ENTER lenyomásával elfogadja az alapértelmezést (amennyiben létezik ilyen).

A felhasználó számára kínálkozik még egy mód a ciklus (és ezzel együtt a teljes program) elhagyására – a CTRL+C billentyűk lenyomása, ami a `KeyboardInterrupt` kivételhez vezet. Mivel pedig ezt a program egyetlen kivételkezelője sem kezeli, megjelenése a program leállásához vezet, és a rendszer egy visszakövetési üzenetet ír ki. Érdemes ilyen „kiskaput" hagynunk? Nos, ha nem tesszük, és a programban benn marad valamilyen szerencsétlen

hiba, meglehet, hogy a felhasználó végtelen ciklusba kerül, amelyből csak a folyamat megsemmisítésével léphet ki. Ezért – ha csak valamilyen lényeges szempont nem indokolja ennek ellenkezőjét – tegyük lehetővé a program megszakítását a Ctrl+C billentyűkombinációval – vagyis ne fogjuk el a megfelelő kivételt.

Érdemes észrevennünk, hogy ezt a függvényt nem csak a `make_html_skeleton.py` programban használhatjuk – a lehetőségei jól jöhetnek más esetekben is, amikor a felhasználóval „párbeszédet" kell folytatnunk. Újrahasznosítására alkalmas lehet a másolás és beillesztés módszere, de ez a programok karbantartása során sok fejfájást okozhat. A következő fejezetben ezekre a problémákra is megoldást lelhetünk, hiszen megismerkedünk az egyedi modulokkal, amelyek lehetővé teszik az ilyen és ehhez hasonló függvények megosztását tetszőleges számú program között.

```
def get_integer(message, name="integer", default=None, minimum=0,
                maximum=100, allow_zero=True):
    ...
```

Ez a függvény olyannyira hasonlít a `get_string()`-hez, hogy felesleges itt bemutatnunk. (A könyvhöz mellékelt forráskódban természetesen megtalálható.) Az `allow_zero` paraméter akkor lehet hasznos, ha a 0 nem érvényes érték, de egyetlen érvénytelen érték használatát szeretnénk lehetővé tenni, hogy ezzel jelezhessük, ha a felhasználó megszakította a műveletet. Hasonló megoldás az is, amikor egy érvénytelen alapértelmezett értéket adunk át, és ha ezt kapjuk vissza, azt úgy értelmezzük, hogy a felhasználó a művelet megszakítása mellett döntött.

Programunk utolsó utasítása a `main()` függvény meghívása. A kód összességében kicsit több mint 150 soros, és felidéz számos, ebben és az ezt megelőző fejezetben megtanult módszert.

Összefoglalás

Ebben a fejezetben bemutattuk a Python vezérlési szerkezeteinek használati alakját, emellett megtanultuk, hogy miként váltsunk ki és fogjunk el kivételeket, továbbá hogyan hozzunk létre egyedi kivételtípusokat.

A fejezet legnagyobb részében az egyedi függvényekkel foglalkoztunk. Megismerkedtünk a létrehozásuk módjával, továbbá néhány, a neveikre és paramétereikre vonatkozó elnevezési szabállyal. Megtudtuk azt is, miként adhatunk meg leírást a függvényeinkhez. A Python rugalmas paraméterhasználatáról és a paraméterek átadásáról részletesen beszámoltunk – megtanultuk, mit kell tennünk, ha rögzített, illetve tetszőleges számú helyzeti vagy kulcsszó-paramétert szeretnénk alkalmazni. Szót ejtettünk emellett mind az állandó értékű, mind a változtatható adattípusok alapértelmezett értékeiről is. Röviden emlékeztettünk a * műveletre, amellyel sorozatokat csomagolhatunk ki, és megismerkedtünk a ** művelettel, amely a megfeleltetések kicsomagolására ad módot.

Amennyiben egy függvény belsejében új értéket szeretnénk adni egy globális változónak, ezt úgy tehetjük meg, ha a `global` kulcsszó feltüntetése mellett hivatkozunk rá, egyébként a Python egy azonos nevű helyi változót hoz létre, és ehhez rendeli az értéket. Általánosságban elmondható azonban, hogy a globális változókat jobb, ha kizárólag állandók tárolására használjuk.

A lambdafüggvényeket gyakran használják kulcsfüggvényként, illetve olyan helyeken, ahol paraméterként függvényt kell átadni. Ebben a fejezetben bemutattuk, hogy miként hozhatunk létre lambdafüggvényeket, akár névtelen függvényként alkalmazva őket, akár változókhoz rendelve egysoros, névvel ellátott függvényként hivatkozva rájuk.

A fejezetben megismerkedtünk az `assert` utasítással is, amellyel olyan elő- és utófeltételezéseket rögzíthetünk, amelyeknek az adott függvény minden használatánál teljesülniük kell. Ezek sokat segíthetnek a hibakeresésben, illetve abban, hogy kevésbé sérülékeny programokot készítsünk.

Megismerkedtünk hát a függvények írásának alapjaival, de ezen felül is számos módszer áll a rendelkezésünkre. Így alkalmazhatunk dinamikus függvényeket, vagyis a függvényeket futásidőben hozhatjuk létre, a megvalósításukat az aktuális körülményekhez igazítva (róluk az 5. fejezetben szólunk), helyi (beágyazott) függvényeket (ezek tárgyalására a 7. fejezetben kerül sor), továbbá önhivatkozó és generátorfüggvényeket, valamint más egyebeket is (ezekről a lehetőségekről a 8. fejezetben olvashatunk bővebben).

Jóllehet a Python számos beépített lehetőséggel rendelkezik, és az alapkönyvtára is kiterjedt, mégis jó eséllyel írunk majd olyan függvényeket, amelyeket több programban is fel tudunk használni. Ezeknek a függvényeknek az ide-oda másolgatása komoly gondokat okozhat a programok fenntartásában, de a Python szerencsére egy tiszta és egyszerű megoldással siet a segítségünkre: lehetővé teszi egyedi modulok létrehozását. A következő fejezetben ezekről esik szó, valamint arról, hogy miként helyezhetjük el bennük a saját függvényeinket. Megtanuljuk továbbá, hogyan töltsük be az alapkönyvtár és saját moduljaink lehetőségeit, és röviden áttekintjük, hogy mit kapunk készen az alapkönyvtártól, hogy kreativitásunkat a valódi feladatokra tartogathassuk.

Gyakorlat

Írjunk interaktív programot, amely képes karakterláncok fájlokban tárolt listáinak kezelésére!

A program készítsen listát azokról a fájlokról, amelyek az aktuális könyvtárban találhatók, és a kiterjesztésük `.lst`. Alkalmazzuk az `os.listdir(".")` parancsot az összes fájl nevének beolvasására, és szűrjük ki azokat, amelyek nem rendelkeznek `.lst` kiterjesztéssel. Ha nincs egyetlen találat sem, kérjük meg a felhasználót, hogy írjon be egy fájlnevet – ezt szükség esetén egészítsük ki az `.lst` kiterjesztéssel. Ha léteznek `.lst` fájlok a könyvtár-

ban, egy számozott lista alakjában írjuk a neveiket a képernyőre, az 1-es sorszámmal kezdve. Kérjük meg a felhasználót, hogy válasszon a sorszámok közül, illetve írjon be 0-t, ha új fájlt szeretne meghatározni.

Amennyiben a felhasználó már létező fájlt adott meg, olvassuk be ennek elemeit. Ha a fájl üres, illetve új fájlt hoztunk létre, egy üzenetet kell kiírnunk, ami szerint „nincsenek elemek a listában".

Amennyiben nincsenek listaelemek, az Add (Hozzáadás) és a Quit (Kilépés) lehetőségeket kell elérhetővé tennünk. Ha a lista nem üres, írjuk ki az elemeit egy 1-től kezdve sorszámozott listában, és ajánljuk fel az Add, Delete (Törlés), Save (Mentés; amennyiben vannak még nem mentett beállítások) és Quit lehetőségeket. Ha a felhasználó ez utóbbit választja, és léteznek még nem mentett módosítások, adjunk lehetőséget a mentésre. Íme egy a programmal folytatott munkamenet átirata (a legtöbb üres sort, valamint az egyes listák felett álló List Keeper feliratot elhagytuk).

```
Choose filename: movies
-- no items are in the list --
[A]dd [Q]uit [a]: a
Add item: Love Actually
1: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: a
Add item: About a Boy
1: About a Boy
2: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]:
Add item: Alien
1: About a Boy
2: Alien
3: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: k
ERROR: invalid choice--enter one of 'AaDdSsQq'
Press Enter to continue...
[A]dd [D]elete [S]ave [Q]uit [a]: d
Delete item number (or 0 to cancel): 2
1: About a Boy
2: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: s
Saved 2 items to movies.lst
Press Enter to continue...
1: About a Boy
2: Love Actually
[A]dd [D]elete [Q]uit [a]:
Add item: Four Weddings and a Funeral
```

```
1: About a Boy
2: Four Weddings and a Funeral
3: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: q
Save unsaved changes (y/n) [y]:
Saved 3 items to movies.lst
```

Gondoskodjunk róla, hogy a `main()` függvény elég kicsi legyen (30 sornál rövidebb), és helyezzük el itt a program központi ciklusát. Írjunk egy függvényt, amely beolvassa az új vagy már létező fájlnevet (az utóbbi esetben pedig be is tölti az elemeket), továbbá egy másikat, amely a felhasználó elé tárja a lehetőségeket, és kezeli a választást. Szükségünk lesz függvényekre, amelyek gondoskodnak az egyes elemek hozzáadásáról, törléséről, a lista kiírásáról (legyen szó elemekről vagy fájlnevekről), betöltéséről és mentéséről. Lemásolhatjuk a `make_html_skeleton.py` programban található `get_string()` és `get_integer()` függvényeket, de elkészíthetjük a saját változatainkat is.

A fájlnevek listájának kiírásakor tíznél kevesebb elem esetén alkalmazzunk 1 karakter szélességű mezőt, 100-nál kevesebb elemnél 2, egyébként pedig 3 szélességűt. Tároljuk az elemeket a betűméretre érzéketlen, ábécésorrendbe állított listában, és tartsuk számon, hogy a lista „piszkos-e”, azaz vannak-e még nem tárolt módosítások. A Save lehetőséget csak akkor ajánljuk fel, ha a lista piszkos, továbbá csak ilyen esetben kérdezzünk rá a módosítások mentésére, miután a felhasználó a Quit lehetőséget választotta. Az elemek beillesztése és törlése piszkossá teszi a listát, míg a mentés újra „kitisztítja”.

Egy lehetséges megoldást megtalálhatunk a `listkeeper.py` fájlban – ez a program 200-nál kevesebb sort tartalmaz.

5

Modulok

- Modulok és csomagok
- A Python alapkönyvtárának áttekintése

Míg a függvényekkel lehetővé válik a kód feldarabolása, illetve az egyes darabok többszöri felhasználása a programon belül, a modulokkal úgy tudjuk a függvényeket (és – ahogy a következő fejezetben látni fogjuk – az egyedi adattípusokat) csoportokba rendezni, hogy akár valamennyi programunkban használhatók lesznek. A Python rendelkezik olyan eszközökkel is, amelyekkel *csomagokat*, vagyis csoportokba szervezett modulokat hozhatunk létre. A csomagokban lévő modulok általában vagy egymáshoz szorosabban kapcsolódó részterületeket fednek le, vagy egymásra támaszkodnak a működésük során.

A fejezet első része a modulokban és csomagokban tárolt függvények használatba vételéről szól, az alapkönyvtárban lévő és a saját magunk írta modulokra és csomagokra egyaránt vonatkozóan. Ezt követően bemutatjuk a saját csomagok és saját modulok létrehozásának mikéntjét. Két saját készítésű modult mutatunk be; az első példa inkább bevezető jellegű, a második pedig a gyakorlatban felmerülő problémák sokaságának kezelését mutatja be – például a rendszerfüggetlenség és a tesztelés kérdését.

Elektronikus dokumentáció
167

A fejezet második része a Python alapkönyvtáráról (standard library) nyújt rövid áttekintést. Fontos tisztában lennünk azzal, hogy milyen lehetőségeket nyújt az alapkönyvtár, hiszen a programozás lényegesen gyorsabb a már megírt függvények használatával, mintha mindent a nulláról indulva írnánk meg. Ezen felül az alapkönyvtár moduljait általában sokan használják – így széleskörű a tesztelésük, és kellően megbízhatók. Az áttekintésen túl néhány rövid példával szemléltetjük az alapkönyvtár mindennapi használatát, valamint utalunk a többi fejezetben tárgyalt modulokra.

Modulok és csomagok

Ha egyszerűen fogalmazunk, a Python-modulok nem egyebek `.py` kiterjesztésű fájloknál. A modulokban olyan Python nyelvű kódot helyezünk el, amilyet csak akarunk. Az eddig megírt programjaink mind egy-egy `.py` kiterjesztésű fájlt jelentenek, tehát egyszerre modul és program valamennyi. A legfőbb különbség abban rejlik, hogy amíg a programokat futtatásra szánjuk, addig a modulok célja, hogy a programjainkba betöltve használhassuk őket.

Nem minden modul esetében találjuk meg a modulnak megfelelő `.py` kiterjesztésű fájlt – példának okáért a `sys` modul a Pythonba beépítve létezik, és van néhány olyan modul is, amelyet más nyelven, főleg C-ben írtak. Ugyanakkor a Python alapkönyvtárának legnagyobb része Python nyelven készült, így például amikor az `import collections` utasítást szerepeltetjük a programunkban, a `collections.namedtuple()` tagfüggvény hívásával nevesített rendezett sorozatokat hozhatunk létre, a tagfüggvény maga pedig a `collections.py` fájlban található. A programjaink szempontjából nem érdekes, hogy egy modult melyik programozási nyelven írtak, ugyanis minden modul betöltése és használata azonos módon történik. A modulok betöltésekor használt utasításforma többféle lehet, például:

```
import valami
import valami1, valami2, …, valamiN
import valami as nekünk_tetsző_név
```

A feni példában a *valami* általában egy modult jelöl – például a `collections` modult. Lehet azonban csomag is vagy egy csomagon belüli modul – ez esetben a név részeit pont (`.`) választja el, mint az `os.path` esetében is. A könyvben az első két utasításformát használjuk, mert ezek a legegyszerűbbek és a legbiztonságosabbak, és minthogy így rákényszerülünk a teljes nevek használatára, biztosan nem lesz a nevek keveredéséből származó problémánk.

A harmadik forma lehetővé teszi, hogy a betöltött modult, illetve csomagot az általunk választott névvel lássuk el – ez elvileg névütközéshez is vezethetne, de a gyakorlatban ez elkerülhető az `as` kulcsszó használatával. Az átnevezés különösen akkor jöhet jól, amikor egy modul különféle megvalósításaival kísérletezgetünk. Példának okáért olyankor, amikor két, egyforma API-val (Application Programming Interface, alkalmazásprogramozási felület) rendelkező modulunk van – `MyModuleA` és `MyModuleB` – a programba elegendő az `import MyModuleA as MyModule` sort írnunk, később pedig minden különösebb nehézség nélkül átválthatunk a `MyModuleB` használatára.

Hol érdemes az `import` utasítást tartalmazó kifejezéseket elhelyeznünk? Általánosan elterjedt gyakorlat, hogy az `import` utasításokat a `.py` kiterjesztésű állományok elején, a shebang-sor és a modul dokumentációja után szerepeltetjük. Ahogy azt régebben, az első fejezetben már említettük, azt tanácsoljuk, hogy az alapkönyvtár moduljait töltsük be elsőként, ezt kövessék a máshonnan származó modulok, és a sort a saját moduljainkkal zárjuk.

Íme néhány egyéb forma az `import` utasítás használatára:

```
from valami import objektum as nekünk_tetsző_név
from valami import objektum1, objektum2, ..., objektumN
from valami import (objektum1, objektum2, objektum3, objektum4,
objektum5,
    objektum6, ..., objektumN)
from valami import *
```

A fenti utasításformák vezethetnek névütközéshez, ugyanis a használatukkal a betöltött objektumok (változók, függvények, adattípusok és modulok) közvetlenül elérhetőkké válnak. Ha a `from...import` utasítással sok objektum betöltését szeretnénk elvégezni, többsoros utasításokat is írhatunk: az utolsó kivételével levédjük az új sorokat, vagy a harmadik utasításformának megfelelően zárójelbe tesszük az objektumok neveit.

Az utolsó példában előkerülő csillag azt jelenti, hogy „tölts be mindent, ami nem privát". Ez a gyakorlatban az aláhúzással kezdődő nevű objektumok kivételével minden objektum betöltését jelenti, illetve amennyiben a modulban található egy névlistát tartalmazó globális `__all__` változó, akkor az ebben a változóban megnevezett objektumokat töltjük be.

`__all__` ☞ 197

Íme néhány, az `import` utasítás használatát bemutató példa:

```
import os
print(os.path.basename(filename))  # biztonságos,
                                   # teljes nevet használó elérés

import os.path as path
print(path.basename(filename))  # a path objektummal való
                                # névütközést kockáztatjuk

from os import path
print(path.basename(filename))  # a path objektummal való
                                # névütközést kockáztatjuk

from os.path import basename
print(basename(filename))  # a basename objektummal való
                           # névütközést kockáztatjuk

from os.path import *
print(basename(filename))  # sok névütközést kockáztatunk
```

A `from valami import *` formával a modul valamennyi objektumát (vagy a csomag valamennyi modulját) betöltjük, és ez nevek százait jelentheti. A `from os.path import *` utasítás esetében majdnem negyven nevet töltünk be – ezek között megtalálható a `dirname`, az `exists` és a `split`, csupa olyan, amelyet akár saját változóinak, illetve függvényeinek is névül választ az ember.

Például, ha kiadjuk az `from os.path import dirname` utasítást, kényelmesen meghívhatjuk a `dirname()` függvényt, anélkül, hogy a teljes nevét megadnánk. Ha azonban a programunkban később kiadjuk a `dirname = "."` utasítást, a `dirname` nevű objektumhivatkozás

a `dirname()` függvény helyett immár a „.“ karakterláncra fog hivatkozni. Ha ekkor a `dirname()` függvény hívásával próbálkozunk, `TypeError` kivétel lép fel, hiszen a `dirname` ekkorra egy karakterláncot jelöl, és a karakterláncokat nem lehet meghívni.

Az `import *` utasításforma által okozható névütközésekre való tekintettel néhány programozói csapatban alapelv, hogy csak az `import valami` utasításformát használják. Mindazonáltal néhány terjedelmes csomag esetében – különös tekintettel a GUI (Graphical User Interface – grafikus felhasználói felület) programozására szolgáló könyvtárakra – gyakran használják az `import *` utasításformát, ugyanis ezekben olyan rengeteg függvény és osztály (saját adattípus) fordul elő, hogy a kézzel történő begépelésük könnyen hosszadalmassá válhat.

Azonnal adódik a kérdés: honnan tudja a Python, hogy hol kell keresnie a betöltendő modulokat, illetve csomagokat? A beépített `sys` modulban található egy `sys.path` nevű lista: az itt szereplő könyvtárak együttese a Python elérési útvonalainak jegyzéke, a *Python path*. Az első mappa az, amelyik magát a programot tartalmazza – abban az esetben is, ha a programot egy másik mappából indítottuk el. Ha a `PYTHONPATH` környezeti változónak van beállított értéke, akkor az itt megadott mappák következnek a listában. A lista végén lévő mappák a Python alapkönyvtárának eléréséhez szükségesek – ezek a Python telepítése során kapnak értéket.

Amikor először töltünk be egy nem beépített modult, a Python egymás után átvizsgálja a `sys.path` listában feltüntetett könyvtárakat. Ennek a műveletnek az egyik eredménye többek között az, hogy amennyiben a Python alapkönyvtárának valamelyik moduljával megegyező nevű modult készítünk, a saját modulunkat találja meg először a Python, ami nyilvánvalóan problémák forrása lesz. Ez úgy kerülhető el, ha soha nem készítünk a Python alapkönyvtárának felső szintű moduljaival vagy mappáival megegyező nevű programot és modult – kivéve, ha az adott modul saját megvalósítását készítjük el, és szándékosan bíráljuk felül az eredetit. (A felső szintű modulok olyan modulok, amelyek `.py` kiterjesztésű állománya valamelyik, a `PYTHONPATH`-ban lévő mappában, és nem ezek egyik almappájában található.) Windowson dolgozva a `PYTHONPATH` például rendszerint tartalmaz egy `C:\Python30\Lib` nevű mappát, vagyis ezen az operációs rendszeren ne készítsünk sem `Lib.py` nevű modult, sem olyat, amelynek a neve megegyezik valamelyik, a `C:\Python30\Lib` nevű mappában lévő modulokéval.

Figyelem! *Az egyik gyors módszer annak ellenőrzésére, hogy egy modulnév már foglalt-e, ha megpróbálkozunk a modul betöltésével. Ezt a konzolban úgy tehetjük meg, hogy a parancsértelmezőt a* `-c` *(„kódvégrehajtás”) parancssori paraméterrel indítjuk el, amelyet a betöltési utasítás követ. Ha például azt akarjuk kideríteni, hogy létezik-e modul* `Music.py` *néven (vagy egy felső szintű,* `Music` *nevű mappa a Python elérési útvonalain), a konzolban a következő utasítást kell kiadnunk.*

```
python -c "import Music"
```

Ha `ImportError` kivételt kapunk, tudhatjuk, hogy ilyen nevű modul, illetve felső szintű mappa nincs használatban. Bármilyen más kimenet – vagy a kimenet hiánya – annyit jelent, hogy a név már foglalt. Sajnos az, hogy a név jelenleg még használható, semmiféle garanciát nem jelent arra nézve, hogy ez mindig így lesz. Később ugyanis elképzelhető, hogy olyan, más fejlesztőtől származó Python-csomagot vagy -modult telepítünk, amelyben a mienkkel ütköző név található – mindenesetre a gyakorlatban ez igen ritkán okoz gondot.

Ha például egy `os.py` nevű modult készítenénk, névütközés lépne fel az alapkönyvtár `os` nevű modulja miatt. Ha azonban `path.py` nevű modult készítünk, akkor minden rendben, ugyanis az alapkönyvtár modulját `os.path` néven tölthetjük be. Ebben a könyvben a saját moduljaink fájlneveit nagy kezdőbetűvel írjuk, ezzel a névütközés – legalábbis Unixon – elkerülhető; az alapkönyvtár fájlnevei ugyanis kis kezdőbetűsek.

Egy program betölthet néhány modult, amelyek aztán további modulokat tölthetnek be, akár olyanokat is, amelyeket már előzőleg betöltöttünk – ez nem okoz problémát.
A Python minden betöltési utasítás végrehajtása előtt ellenőrzi, hogy a modult korábban nem töltötte-e be. Ha nem, akkor a Python végrehajtja a bájtkódúra fordított kódot, létrehozva ezzel a változókat, függvényeket és a kódban lévő egyéb objektumokat, és emellett feljegyzi magának, hogy ezt a modult már betöltötte. A Python minden további betöltés alkalmával felismeri, hogy a modult már betöltötte, és tétlen marad.

Amikor a Pythonnak a bájtkódúra fordított kódra van szüksége, automatikusan létrehozza azt. Ennyiben eltér például a Javától, ahol a bájtkódú fordítást külön kérnünk kell.
A Python először a modul `.py` kiterjesztésű állományát keresi, de `.pyo` kiterjesztéssel, vagyis az optimalizált, bájtkódúra fordított változatot. Ha a Python `.pyo` kiterjesztésű fájlt nem talál – vagy talál ugyan, de a `.py` kiterjesztésűnél régebbit, tehát nem a legfrissebb változatról készültet –, akkor `.pyc` kiterjesztésűt keres, ami a nem optimalizált bájtkódú változatot tartalmazza. Ha ez eléggé friss, akkor betölti, egyébként pedig betölti, és bájtkódúvá fordítja a `.py` kiterjesztésű változatot. A végeredmény minden esetben az, hogy a Python a memóriába tölti a bájtkódú modulváltozatot.

Ha a Pythonnak kellett elvégeznie a `.py` kiterjesztésű változat bájtkódra fordítását, a létrejött `.pyc` (ha a Python parancssorában megadtuk a `-O` kapcsolót, illetve a `PYTHONOPTIMIZE` környezeti változót úgy állítottuk be, akkor `.pyo`) kiterjesztésű fájlt menti is, már amennyiben a mappa írható. A bájtkódú változat mentése a `-B` kapcsolóval, illetve a `PYTHONDONTWRITEBYTECODE` környezeti változó beállításával letiltható.

A bájtkódú változatok használata gyorsabban induló programokat eredményez, lévén a parancsértelmezőnek a kód betöltése, fordítása, lehetőség szerinti mentése és futtatása helyett csak a kód betöltése és futtatása a dolga. A program maga természetesen nem fut gyorsabban. Az alapkönyvtár moduljait rendszerint a telepítési folyamat részeként fordítja bájtkódúra a Python.

Csomagok

A csomag (package) nem egyéb, mint egy olyan mappa, amely néhány modult, valamint egy `__init__.py` nevű fájlt tartalmaz. Tegyük fel például, hogy van egy képzeletbeli modulkészletünk, amellyel különféle grafikus formátumok írása és olvasása valósítható meg. A modulok: a `Bmp.py`, a `Jpeg.py`, a `Png.py`, a `Tiff.py`, és az `Xpm.py` mindegyike rendelkezik `load()`, `save()` és egyéb függvényekkel.* A modulok lehetnének a programunkkal egy mappában, de egy nagy program esetében, amely saját készítésű modulok sokaságára támaszkodik, a grafikai modulokat nehéz lenne így kezelnünk. Ha azonban egy külön alkönyvtárba, mondjuk a `Graphics` nevűbe kerülnek, akkor könnyebb az egységes kezelés. Ha még egy üres `__init__.py` fájlt is elhelyezünk közöttük, a mappa csomaggá lép elő:

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
    Tiff.py
    Xpm.py
```

Amennyiben a `Graphics` mappa a programunk mappájának egyik alkönyvtára, vagy megtalálható a Python elérési útvonalait rögzítő listában, a fenti modulok bármelyike betölthető és használható. Óvatosnak kell lennünk, nehogy a felső szintű modulnév (a `Graphics`) névütközésbe kerüljön az alapkönyvtár valamelyik felső szintű nevével. (Unix rendszereken ez könnyen kivédhető, ha a modul neve nagybetűvel kezdődik, lévén az alapkönyvtár nevei kisbetűvel kezdődnek. A modul betöltésének, illetve használatba vételének a módja a következő:

```
import Graphics.Bmp
image = Graphics.Bmp.load("bashful.bmp")
```

Rövid programok esetében néhány programozó előnyben részesíti a rövidebb nevek használatát. A Python kétféle módszert kínál erre.

```
import Graphics.Jpeg as Jpeg
image = Jpeg.load("doc.jpeg")
```

A fenti példában a `Graphics` csomag `Jpeg` moduljának betöltésekor szóltunk a Pythonnak, hogy a teljes név (`Graphics.Jpeg`) használata szerint jobb szeretnénk az egyszerűbb, `Jpeg` elnevezést használni.

```
from Graphics import Png
image = Png.load("dopey.png")
```

Ez a kódrészlet a `Graphics` csomag `Png` modulját közvetlenül tölti be a programba. A `from...import` utasításforma közvetlenül elérhetővé teszi a `Png` modult.

---

* A grafikus állományok kezelését a külső fejlesztők által készített modulok sokasága támogatja. A legfigyelemre-

méltóbb alighanem a Python Imaging Library (www.pythonware.com/products/pil).

Semmi sem kötelez bennünket arra, hogy a programunkban a csomag eredeti nevét használjuk. Például:

```
from Graphics import Tiff as picture
image = picture.load("grumpy.tiff")
```

Ebben a példában a `Tiff` modult használjuk, de más néven: a program `picture` néven hivatkozik rá.

Vannak olyan helyzetek, amikor kényelmes egy csomag összes modulját egyetlen utasítással betöltenünk. Ha ezt szeretnénk tenni, a csomag `__init__.py` állományát úgy kell átszerkesztenünk, hogy tartalmazza a betölteni kívánt modulok neveit megadó utasítást. Az utasításnak a modulokat az `__all__` különleges szerepű változónak kell értékül adnia. Például a `Graphics/__init__.py` fájlban az alábbi sort kell elhelyeznünk:

```
__all__ = ["Bmp", "Jpeg", "Png", "Tiff", "Xpm"]
```

Mindössze ennek a sornak kell mindenképpen benne lennie az `__init__.py` fájlban, de igazából azt írunk még bele, ami nekünk tetszik. Most már használható az `import` utasítás egy újabb formája:

```
from Graphics import *
image = Xpm.load("sleepy.xpm")
```

A `from` *csomag* `import *` utasításformával az `__all__` listában megnevezett valamennyi modult közvetlenül betöltjük a programunkba, vagyis az utasítás végrehajtása után az `Xpm` modulon felül az összes többi modul is közvetlenül elérhető lesz.

Ahogy korábban jeleztük, ez az utasításforma modulok esetében is alkalmazható, vagyis a `from` *modul* `import *` utasítással a modulban megadott összes függvény, változó és egyéb objektum (kivéve azokat, amelyeknek aláhúzással kezdődik a neve) betöltése megvalósul. Ha pontosan szabályozni szeretnénk, hogy minek a betöltése történik meg, amikor a `from` *modul* `import *` utasításformát használjuk, a modulban magában is megadhatunk egy `__all__` nevű listát. Ebben az esetben a `from` *modul* `import *` utasítás csak az `__all__` listában megnevezett objektumok betöltését végzi el.

Eddig az egymásba ágyazhatóságnak csak egyetlen szintjét mutattuk be, de a Python nyelv lehetővé teszi, hogy olyan mélységig ágyazzunk egymásba csomagokat, ameddig csak szeretnénk. Így a `Graphics` könyvtárban lehet mondjuk egy `Vector` nevű alkönyvtárunk, ebben pedig további modulok, például az `Eps.py` és az `Svg.py`:

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
```

```
Tiff.py

Vector/
    __init__.py
    Eps.py
    Svg.py
Xpm.py
```

Hogy a `Vector` könyvtár is csomagnak minősüljön, kell benne lennie egy `__init__.py` állománynak, amely, ahogy már jeleztük, lehet üres, illetve tartalmazhat egy `__all__` nevű listát, amely azoknak a programozóknak teszi kényelmesebbé az életét, akik a betöltést a `from Graphics.Vector import *` utasítással szeretnék elvégezni. A beágyazott csomagok eléréséhez a már használt utasításformából indulunk ki:

```
import Graphics.Vector.Eps
image = Graphics.Vector.Eps.load("sneezy.eps")
```

A teljes név meglehetősen hosszú, emiatt sok programozó igyekszik a modulok hierarchiáját minél kevesebb szinttel megvalósítani.

```
import Graphics.Vector.Svg as Svg
image = Svg.load("snow.svg")
```

Egy modulra mindig hivatkozhatunk az általunk megadott rövid néven, ahogy a fenti példában is tettük, ez a módszer azonban növeli a névütközés veszélyét.

Egyedi modulok

Minthogy a modulok tulajdonképpen egyszerű, `.py` kiterjesztésű fájlok, a létrehozásuk sem jelent különösebb gondot. Ebben a részben két saját készítésű modult tekintünk át. Az első, a `TextUtil.py` fájlban található `TextUtil` modul, mindössze három függvényt tartalmaz. Az `is_balanced()` függvény `True` értékkel tér vissza, ha a neki átadott függvényben a nyitó és záró zárójelek típusban és számban megfelelnek egymásnak. A `shorten()` függvényt korábban már bemutattuk, a `simplify()` pedig szóközöket és egyéb karaktereket távolít el egy karakterláncból. A modul tárgyalása során azt is megtudjuk, hogy miként lehetséges a dokumentációs karakterláncban lévő kód végrehajtása egységtesztek elvégzésekor.

A második, a `CharGrid.py` fájlban lévő `CharGrid` modul egy karakterekből álló mintát (rácsot) tárol, és lehetővé teszi, hogy vonalakat, négyszögeket és szöveget „rajzoljunk" ebbe a rácsba, majd mindezt a konzolon is megjelenítsük. A modul pár olyan módszert is bemutat, amellyel korábban még nem találkoztunk, és egészében véve is jellemzőbb példáját nyújtja a nagyszabású, összetett moduloknak.

A TextUtil modul

Ennek a modulnak – és a legtöbb más modulnak is – különbözik a felépítése a programoknál megszokottól. Az első sor tartalmazza a shebang-et, majd pár megjegyzés következik (rendszerint a szerzői jogi és a felhasználási engedélyekkel kapcsolatos információ). Ezeket

általában egy három idézőjelbe zárt karakterlánc követi, amely áttekintést nyújt a modul tartalmáról, gyakran a felhasználás módját bemutató példákkal – ez a rész a modulban lévő engedélyhez tartozó megjegyzéseket elhagyva:

```
#!/usr/bin/env python3
# Copyright (c) 2008 Qtrac Ltd. All rights reserved.
"""
This module provides a few string manipulation functions.

>>> is_balanced("(Python (is (not (lisp))))")
True
>>> shorten("The Crossing", 10)
'The Cro...'
>>> simplify(" some   text   with spurious whitespace ")
'some text with spurious whitespace'
"""

import string
```

A modulban lévő dokumentációs karakterlánc akkor válik hozzáférhetővé a programok, illetve más modulok számára, ha azok a betöltést a `TextUtil.__doc__` formában végzik el. A modulban a docstring után következnek a betöltést végző sorok – esetünkben egyetlen egy –, és ezt a modul többi része követi.

A `shorten()` függvény már szerepelt a könyvben, így most nem ismételjük meg. És minthogy a függvények helyett ezúttal a modulokra összpontosítunk, a `simplify()` függvényt teljes egészében, a dokumentációs karakterlánccal együtt mutatjuk be, de az `is_balanced()` függvénynek már csak a kódot tartalmazó része fog szerepelni. Íme a `simplify()` függvény, két részletben:

```
def simplify(text, whitespace=string.whitespace, delete=""):
r"""Returns the text with multiple spaces reduced to single spaces

The whitespace parameter is a string of characters, each of which
is considered to be a space.
If delete is not empty it should be a string, in which case any
characters in the delete string are excluded from the resultant
string.

>>> simplify("   this and\n that\t too")
'this and that too'
>>> simplify(" Washington  D.C.\n")
'Washington D.C.'
>>> simplify(" Washington  D.C.\n", delete=",;:.")
'Washington DC'
>>> simplify(" disemvoweled ", delete="aeiou")
'dsmvwld'
"""
```

Nyers karakterláncok
☜ 61

A `def` utasítást tartalmazó sor után következik a függvény dokumentációs karakterlánca, amely a hagyomány szerint egy egysoros leírásból, egy ezt követő üres sorból és további leírásból, majd néhány példából áll. A példákat úgy adjuk meg, mintha interaktívan írtuk volna be őket. Minthogy az idézőjelben lévő karakterláncok a dokumentációs karakterláncon belül vannak, a bennük lévő fordított perjeleket vagy le kell védenünk, vagy ahogy mi is tettük, nyers, háromszoros idézőjelbe tett karakterláncot kell használnunk.

```
    result = []
    word = ""
    for char in text:
        if char in delete:
            continue
        elif char in whitespace:
            if word:
                result.append(word)
                word = ""
        else:
            word += char
    if word:
        result.append(word)
    return " ".join(result)
```

A `result` nevű listában tároljuk a „szavakat”, vagyis az olyan karakterláncokat, amelyek nem tartalmaznak szóközöket, illetve törlésre kijelölt karaktereket. Az átadott szöveget karakterenként járjuk be, a törlésre jelölt karaktereket átlépve. Ha szóközként megadott karakterbe botlunk, és éppen egy szó belsejében vagyunk, a szót hozzáadjuk az eredményeket tároló listához, majd a szót eddig tároló változót kiürítjük. Minden más karakter fellelésekor a karakter az éppen kiépülőben lévő szó végére kerül. A függvény végül egyetlen karakterláncot ad vissza, amelyben az eredményeket tartalmazó lista szavai szerepelnek, egy-egy szóközzel elválasztva.

Az `is_balanced()` függvény hasonló mintát követ: a `def` utasítást tartalmazó sort követi a docstring, benne az egysoros, illetve az üres sort követő bővebb leírás, néhány példa, majd maga a kód. Íme a dokumentációs karakterlánc nélküli kód:

```
def is_balanced(text, brackets="()[]{}<>"):
    counts = {}
    left_for_right = {}
    for left, right in zip(brackets[::2], brackets[1::2]):
        assert left != right, "the bracket characters must differ"
        counts[left] = 0
        left_for_right[right] = left
    for c in text:
        if c in counts:
            counts[c] += 1
        elif c in left_for_right:
            left = left_for_right[c]
```

```
            if counts[left] == 0:
                return False
            counts[left] -= 1
    return not any(counts.values())
```

A függvény két szótárat épít fel. A `counts` szótár kulcsai a nyitó zárójelek („(”, „[”, „{” és „<”), az értékei pedig egész számok. A `left_for_right` szótár kulcsai a záró zárójelek („)”, „]”, „}” és „>”), az értékei pedig a megfelelő nyitó zárójelek. Ha a szótárak elkészültek, a függvény karakterenként bejárja a szöveget. Amikor nyitó zárójelet találunk, a megfelelő mérőszámot eggyel növeljük. Ehhez hasonlóan, amikor záró zárójelhez érünk, a függvény megkeresi a megfelelő nyitó zárójelet. Ha pedig annak nulla a száma, az annyit tesz, hogy fölös számú záró zárójelre akadtunk, vagyis azonnal `False` értéket kapunk vissza. Ha az érték nullától különböző, akkor eggyel csökkentjük. A szöveg bejárásának végeztével elvileg minden mérőszám nulla értékű – már amennyiben ugyanannyi nyitó és záró zárójel van. Ha tehát bármelyik szám nullától eltérő értékű, a függvény `False` értéket ad vissza, és `True` a visszaadott érték, ha mindegyik nulla.

Eddig a pontig minden majdnem olyan volt, mint bármely Python-programban. Ha a `TextUtil.py` fájl programfájl lenne, alighanem lenne benne néhány további függvény, a fájl vége felé pedig egy önmagában álló függvényhívás, ami a feldolgozás megkezdését végezné. Ez a fájl azonban modul, vagyis azért készítettük, hogy más fájlokba lehessen betölteni, és ekképpen elég, ha csak megadjuk benne a függvényeket. Immár bármely program és bármely modul betöltheti és használhatja a `TextUtil` modult:

```
import TextUtil

text = "  a     puzzling conundrum "
text = TextUtil.simplify(text) # text == 'a puzzling conundrum'
```

Ha azt szeretnénk, hogy egy bizonyos program használhassa a `TextUtil` modult, elég, ha a `TextUtil.py` fájlt a programmal azonos könyvtárban helyezzük el. Ha azt szeretnénk, hogy a `TextUtil.py` fájlban lévő modul minden programunk számára elérhető legyen, többféleképpen is megközelíthető a probléma. Az egyik lehetőség, hogy a modult elhelyezzük a Python `site-packages` alkönyvtárában – ez Windows rendszereken rendszerint a `C:\Python30\Lib\site-packages` könyvtár, Mac OS X és Unix rendszereken azonban különféle helyekre szokott kerülni. A könyvtár megtalálható a Python elérési útvonalán, vagyis az ide helyezett modulok mindig elérhetők. A második lehetőség, ha az olyan modulok számára, amelyeket minden programunkban használni szeretnénk, külön mappát hozunk létre, és a mappát beállítjuk a `PYTHONPATH` környezeti változóban. A harmadik lehetőség, ha a modul a *helyi* `site-packages` alkönyvtárba kerül; ez Windowson az `%APPDATA%/Python/Python30/site-packages` mappa, Unixon (a Mac OS X operációs rendszert is ideértve) pedig a `~/.local/lib/python3.0/site-packages` mappa. Ez a mappa is benne van a Python elérési útvonalait tartalmazó listában. A második és harmadik módszernek megvan az az előnye, hogy a saját kódunkat a hivatalos telepítéstől elkülönítve tároljuk.

Az, hogy elkészült a `TextUtil` modul, igazán remek dolog, de ha sok programban szeretnénk használni a modult, talán pontosabban meg kellene bizonyosodnunk a helyes működéséről. Ennek egy igazán egyszerű módja, ha a dokumentációs karakterláncban lévő példákat végrehajtjuk, és megbizonyosodunk róla, hogy a várt eredményt kapjuk. Mindez elvégezhető, ha a modult tartalmazó, `.py` kiterjesztésű fájl végén elhelyezünk három sort:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Ha egy Python-programba betöltünk egy modult, a Python a modul számára létrehoz egy `__name__` nevű változót, és a modul nevét elhelyezi ebben a változóban. A modul neve megegyezik az őt tartalmazó fájl nevével, mindössze a `.py` kiterjesztést hagyjuk el. Így a példánkban a modul betöltésének megtörténtével a `__name__` változó értéke `TextUtil` lesz. Így az `if` utasításnál megadott feltétel nem teljesül, vagyis a két utolsó sor végrehajtása elmarad. Ez annyit tesz, hogy a modul betöltésekor az utolsó három sor gyakorlatilag nem jelent költséget.

Amikor egy `.py` kiterjesztésű fájlt futtatunk, a Python a program számára létrehoz egy `__name__` nevű változót, amely a `"__main__"` karakterláncot kapja értékül. Vagyis, ha a `TextUtil.py` állományt *futtatjuk*, mintha program lenne, a Python a `__name__` nevű változó értékéül beállítja a `"__main__"` karakterláncot, az `if` utasítás feltételének kiértékelésekor pedig `True` értéket kapunk, és az utolsó két sort végrehajtjuk.

A `doctest.testmod()` függvény a Python önellenőrző képességeit kihasználva megkeresi a modulban lévő valamennyi függvényt, illetve a függvényekhez tartozó dokumentációs karakterláncokat, és megpróbálja végrehajtani az összes olyan kódrészletet, amit azokon belül talál. Ha ezen a módon futtatjuk a modult, csak akkor kapunk kimenetet, ha a futás során hiba merül fel. Ez első közelítésben zavaró lehet, mivel úgy tűnik, mintha mi sem történne, de ha a `-v` parancssori kapcsolót is használjuk, az alábbihoz hasonló kimenetet kapunk:

```
Trying:
    is_balanced("(Python (is (not (lisp))))")
Expecting:
    True
ok
...
Trying:
    simplify(" disemvoweled ", delete="aeiou")
Expecting:
    'dsmvwld'
ok
4 items passed all tests:
    3 tests in __main__
    5 tests in __main__.is_balanced
    3 tests in __main__.shorten
```

```
    4 tests in __main__.simplify
15 tests in 4 items.
15 passed and 0 failed.
Test passed.
```

Ahol kihagytunk sorokat, ott ezt három ponttal jeleztük. Ha a `-v` kapcsolót használjuk, azoknak a függvényeknek (osztályoknak, tagfüggvényeknek) a listáját is megkapjuk, amelyekhez nem adtunk meg tesztet. Figyeljük meg, hogy a `doctest` modul a modul és az egyes függvények dokumentációs karakterláncaiban lévő teszteket egyaránt megtalálja.

A dokumentációs karakterláncokban lévő végrehajtható példák neve *doctest*. Vegyük észre, hogy ha doctest-et írunk, sem a `simplify()` függvényt, sem az egyéb függvényeket nem kell a teljes nevükkel hívnunk, hiszen a tesztek magában a modul belsejében kapnak helyet. A modulon kívül – feltételezve, hogy a modult betöltöttük – a függvények teljes nevét kell használnunk, például: `TextUtil.is_balanced()`.

A következő részben megmutatjuk, hogy miként végezhetünk alaposabb tesztelést, különös tekintettel az olyan esetekre, amikor hibákra számítunk – például érvénytelen adatok okozta kivételekre. Emellett áttekintünk pár egyéb, a modulok létrehozása során felmerülő kérdést. Ilyen például a modulok előkészítése, az operációs rendszerek okozta különbségek figyelembe vétele, és annak kezelése, hogy a `from` *`modul`* `import *` utasításforma használata esetén csak azok az objektumok kerüljenek át a betöltő programba, amelyeket kívülről elérhetővé akarunk tenni.

A CharGrid modul

A `CharGrid` modul a memóriában karakterekből álló rácsot alakít ki. Arra való, hogy a rácsra karakterekből álló vonalakat, négyszögeket és szöveget „rajzoljunk”, és ezt a konzolon megjelenítsük. Íme a modul dokumentációs karakterláncában lévő dokumentációs tesztek (doctest):

```
>>> resize(14, 50)
>>> add_rectangle(0, 0, *get_size())
>>> add_vertical_line(5, 10, 13)
>>> add_vertical_line(2, 9, 12, "!")
>>> add_horizontal_line(3, 10, 20, "+")
>>> add_rectangle(0, 0, 5, 5, "%")
>>> add_rectangle(5, 7, 12, 40, "#", True)
>>> add_rectangle(7, 9, 10, 38, " ")
>>> add_text(8, 10, "This is the CharGrid module")
>>> add_text(1, 32, "Pleasantville", "@")
>>> add_rectangle(6, 42, 11, 46, fill=True)
>>> render(False)
```

A `CharGrid.add_rectangle()` függvény legalább négy paramétert vár: a bal felső sarok és a jobb alsó sarok sorának és oszlopának számait. A körvonal megrajzolásához használni kívánt karakter lehet az ötödik paraméter, a hatodikként átadott logikai érték pedig azt szabályozza, hogy a körvonalat ki kell-e tölteni (a körvonalhoz használt karakterrel).

Amikor a függvényt először hívjuk, a harmadik és a negyedik paramétert egy kételemű rendezett sorozat (szélesség, magasság) kicsomagolásával adjuk át. A rendezett sorozatot a `CharGrid.get_size()` függvény visszatérési értékeként kapjuk meg.

A `CharGrid.render()` függvény alapértelmezés szerint a rács kiírása előtt kiüríti a képernyőt, de ez meggátolható a `False` paraméter átadásával – mi is ezt tettük. Íme az előző dokumentációs tesztek eredményéből származó minta:

```
%%%%*************************************************
%   %                                @@@@@@@@@@@@@@@@  *
%   %                                @Pleasantville@  *
%   %                     ++++++++++ @@@@@@@@@@@@@@@@ *
%%%%                                                  *
*      ##################################           *
*      ##################################    ****   *
*      ##                              ##    ****   *
*      ## This is the CharGrid module ##    ****   *
* !    ##                              ##    ****   *
* ! |  ##################################    ****   *
* ! |  ##################################           *
*   |                                              *
************************************************************
```

A modul a `TextUtil` modulnál megismert módon kezdődik: shebang-sor, szerzői jogi és a felhasználási engedélyekkel kapcsolatos megjegyzések, majd a modulszintű dokumentációs karakterlánc, amely leírja a modul működését, és tartalmazza a korábban bemutatott dokumentációs teszteket. Ezt követi maga a kód, két, `import` utasítást tartalmazó sorral, amelyek közül az egyik a `sys`, a másik a `subprocess` modult tölti be. A `subprocess` modulról a 9. fejezetben szólunk részletesebben.

A modulon belül két hibakezelési házirend érvényesül. Számos függvénynek van `char` típusú paramétere, aminek valójában mindig egy olyan karakterláncnak kell lennie, amely pontosan egy karaktert tartalmaz. Ha ez a kívánalom sérül, azt végzetes programhibának tekintjük, így `assert` utasítást tartalmazó kifejezésekkel ellenőrizzük a karakterlánc hosszát. Ha túl magas értékeket kapunk sor- és oszlopszámokként, azt olyan hibának tekintjük, ami előfordulhat, vagyis ha ilyen esemény történik, egyedi kivételeket hozunk létre.

Az alábbiakban a kód néhány jellemző, kulcsszerepű részét tekintjük át, kezdve az egyedi kivételeknél:

```
class RangeError(Exception): pass
class RowRangeError(RangeError): pass
class ColumnRangeError(RangeError): pass
```

A modulban lévő, kivételeket létrehozni képes függvények egyike sem fog `RangeError` kivételt adni, mindig `RowRangeError` vagy `ColumnRangeError` jön létre, attól függően, hogy a sor vagy az oszlop száma esik kívül a megadható tartományon. Így azonban, hogy

a kivételeket hierarchiába szerveztük, a modul felhasználójának a kezébe adjuk a választás lehetőségét: vagy a pontosan a hibának megfelelő kivételt fogja el, vagy mindkettőt egyszerre, ha a `RangeError` alaposztályba tartozó kivételeket figyeli. Figyeljük meg, hogy bár a dokumentációs tesztekben a kivételek nevét úgy használjuk, ahogy az előző sorokban is írtuk, a `CharGrid` modult betöltő programban a kivételek neve természetesen módosul: a `CharGrid.RangeError`, a `CharGrid.RowRangeError`, és a `CharGrid.ColumnRangeError` neveket kell használnunk.

```
_CHAR_ASSERT_TEMPLATE = ("char must be a single character: '{0}' "
                         "is too long")
_max_rows = 25
_max_columns = 80
_grid = []
_background_char = " "
```

A fenti sorokban néhány privát változónak adtunk értéket; ezeket csak a modul használja. A változók nevei aláhúzással kezdődnek, így amennyiben a modul betöltését a `from CharGrid import *` utasítással végezzük, ezek a változók kimaradnak a betöltésből. (A probléma másik megközelítési módja az lenne, ha értéket adnánk az `__all__` nevű listának.) A `_CHAR_ASSERT_TEMPLATE` egy karakterlánc, amelyet a `str.format()` függvény használ. A modulban az a szerepe, hogy az `assert` utasítások hibaüzenetét tárolja. A többi változót az első előfordulásuk helyén tárgyaljuk.

```
if sys.platform.startswith("win"):
    def clear_screen():
        subprocess.call(["cmd.exe", "/C", "cls"])
else:
    def clear_screen():
        subprocess.call(["clear"])
clear_screen.__doc__ = """Clears the screen using the underlying \
window system's clear screen command"""
```

A konzolképernyő kiürítésének módja az operációs rendszertől függ. Windowson a `cmd.exe` programot kell a megfelelő paraméterekkel futtatnunk, a legtöbb Unix rendszeren pedig a `clear` programot futtatjuk. A subprocess modul `subprocess.call()` függvénye lehetővé teszi egy külső program futtatását, így használható arra, hogy az operációs rendszer szempontjából megfelelő módszerrel ürítsük ki a képernyőt.
A `sys.platform` karakterlánc tárolja a programot futtató operációs rendszer nevét – például: win32 vagy linux2. Így az operációs rendszerek eltéréséből eredő különbségek kezelésének egyik módja az lenne, ha megadnánk egy az alábbihoz hasonló `clear_screen()` függvényt:

```
def clear_screen():
    command = (["clear"] if not sys.platform.startswith("win") else
               ["cmd.exe", "/C", "cls"])
    subprocess.call(command)
```

A megközelítés hátránya, hogy bár tisztában vagyunk azzal, hogy a program futása alatt nem változhat az operációs rendszer, mégis minden függvényhívás alkalmával lefut az ellenőrzés.

Ha el akarjuk kerülni azt, hogy a `clear_screen()` függvény minden egyes hívásakor megtörténjen a programot futtató operációs rendszer ellenőrzése, a modul betöltésekor létrehozunk egy az operációs rendszernek megfelelő `clear_screen()` függvényt, és onnantól fogva mindig azt használjuk. Mindez azért lehetséges, mert a `def` utasítás is olyan, mint a Python nyelv többi utasítása: amikor a parancsértelmező eléri az `if` utasítást, vagy az első, vagy a második `def` utasítást hajtja végre, *dinamikusan* hozva létre vagy az egyik, vagy a másik `clear_screen()` függvényt. Minthogy a függvényt nem egy másik függvény belsejében (vagy másik osztály belsejében, ahogy a következő fejezetben látni fogjuk) hozzuk létre, a függvényünk globális, vagyis éppen úgy érhető el, mint a modulban lévő többi függvény.

A függvény létrehozása után pontos utasítással adjuk meg a hozzá tartozó dokumentációs karakterláncot, ezzel elkerüljük ugyanannak a docstring-nek a két helyen való megadását, és megmutatjuk azt is, hogy a dokumentációs karakterlánc nem egyéb, mint a függvény jellemzőinek egyike. A jellemzők között megtalálható a függvény modulja és a neve.

```
def resize(max_rows, max_columns, char=None):
    """Changes the size of the grid, wiping out the contents and
    changing the background if the background char is not None
    """
    assert max_rows > 0 and max_columns > 0, "too small"
    global _grid, _max_rows, _max_columns, _background_char
    if char is not None:
        assert len(char) == 1, _CHAR_ASSERT_TEMPLATE.format(char)
        _background_char = char
    _max_rows = max_rows
    _max_columns = max_columns
    _grid = [[_background_char for column in range(_max_columns)]
             for row in range(_max_rows)]
```

Ez a függvény egy `assert` utasítással juttatja érvényre a kódolási hibává nyilvánítást, amikor a rácsot 1x1-es méretnél kisebbre próbáljuk állítani. Ha a háttérül szolgáló karakter is meg van adva, szintén egy `assert` utasítással ellenőrizzük, hogy pontosan egy karakterről van-e szó. Ha nem, akkor a művelet hibaüzenete a `_CHAR_ASSERT_TEMPLATE` karakterláncban tárolt szöveg lesz, a `{0}` helyére az átadott karakterlánc kerül.

Sajnos szükség van a `global` utasítás használatára, hiszen a függvény belsejében számos globális változót kell frissítenünk. Ez olyasmi, amit objektumközpontú megközelítéssel elkerülhetünk, ahogy azt a 6. fejezetben látni fogjuk.

Listaértelmezések 113 ☜

A _grid lista egy listaértelmezés belsejében lévő listaértelmezéssel jön létre. A [[char] * columns] * rows alakú listamásolási művelet nem vezet eredményre, hiszen ez csak a belső lista megosztását (sekély másolását) végzi el. Használhattunk volna helyette egymásba ágyazott for...in ciklusokat:

```
_grid = []
for row in range(_max_rows):
    _grid.append([])
    for column in range(_max_columns):
        _grid[-1].append(_background_char)
```

A fenti kódot viszonylag nehezebb megérteni, mint a listaértelmezést, ugyanakkor lényegesen hosszabb is nála.

Elsődleges célunk a modulok megvalósításának megértése, ezért a rajzolást végző függvények közül csak egyet tekintünk át, és éppen csak belekóstolunk a rajzolás mikéntjébe. Íme az add_horizontal_line() függvény, két részletben közölve:

```
def add_horizontal_line(row, column0, column1, char="-"):
    """Adds a horizontal line to the grid using the given char

    >>> add_horizontal_line(8, 20, 25, "=")
    >>> char_at(8, 20) == char_at(8, 24) == "="
    True
    >>> add_horizontal_line(31, 11, 12)
    Traceback (most recent call last):
    ...
    RowRangeError
    """
```

A dokumentációs karakterláncokban két teszt található: az egyiktől azt várjuk, hogy működjön, a másiktól pedig azt, hogy hozzon létre kivételt. Ha a dokumentációs tesztekben lévő kivételekkel bajlódunk, a módszer az, hogy a „Traceback" kezdetű sort adjuk meg – ez ugyanis mindig ugyanaz. Így hozhatjuk tudomására a doctest modulnak, hogy kivételre számítunk. Ezek után három ponttal jelezzük a közbeeső sorokat, amelyek mindig mások lehetnek, végül pedig a várt kivétel nevét tartalmazó sor következik. A char_at() függvény egyike a modulban szereplő függvényeknek, amely a rács adott sorában és oszlopában álló karaktert adja vissza.

```
    assert len(char) == 1, _CHAR_ASSERT_TEMPLATE.format(char)
    try:
        for column in range(column0, column1):
            _grid[row][column] = char
    except IndexError:
        if not 0 <= row <= _max_rows:
            raise RowRangeError()
        raise ColumnRangeError()
```

A kód ugyanazzal a karakterlánc hosszúságát ellenőrző résszel kezdődik, amelyet a `resize()` függvényben is használtunk. A sor- és az oszlopszámot hordozó paraméter külön ellenőrzése helyett a függvény feltételezi, hogy a paraméterek érvényesek. Ha nem létező sort vagy oszlopot próbálunk elérni, `IndexError` kivétel lép fel, amit elkapunk, és helyette egy a modulra jellemző kivételt hozunk létre. Az efféle programozási stílust szokás „könnyebb elnézést kérni, mint engedélyt" stílusnak nevezni, és általában *„pythonosabb"* módszernek számít, mint a „lassan járj, tovább érsz" módszer, azaz az ellenőrzések előzetes elvégzése. Ha kivételes esetek ritkán fordulnak elő, hatékonyabb a létrejövő kivételekre támaszkodni, mint előre ellenőrizni a dolgokat. (Az `assert` utasításokat tartalmazó kifejezések használata másféle elbírálás alá esik, mivel jó esetben velük soha nem történik meg a baj, és a kész kódban gyakran megjegyzéssé is alakítják őket.)

A modulnak majdnem a legvégén – miután valamennyi függvény megadása megtörtént –, egyetlen egyszer hívjuk meg a `resize()` függvényt:

```
resize(_max_rows, _max_columns)
```

Ez a hívás előkészíti az alapértelmezett (25x80) méretű rácsot, és figyel arra is, hogy a kód, amely a modult betölti, azonnal használatba is vehesse. A hívás nélkül a betöltő programnak, illetve modulnak mindig meg kellene hívnia a `resize()` függvényt a rács előkészítéséhez, valahányszor a modult betöltjük. Erre a programozóknak mindig emlékezniük kellene, és a dolog többszörös előkészítéshez is vezethet.

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

A modul három utolsó sora a szokásos három sor azokban az esetekben, amikor a `doctest` modullal ellenőrizzük a modul dokumentációs tesztjeit.

A `CharGrid` modulnak van egy fontos képességbeli hiányossága: csak egyetlen karakteres rács alakítható ki vele. A probléma egyik megoldását az jelentené, ha a modulban rácsok gyűjteményét tárolnánk, de ez azzal járna, hogy a modul használóinak a függvény minden hívásakor meg kellene adniuk a rács azonosítására szolgáló sorszámot vagy kulcsot. Olyan esetekben, amikor egy objektumból több példányra is szükség van, jobb megoldást jelent olyan modult készíteni, amely megad egy objektumosztályt (egy egyedi adattípust), ugyanis az objektumosztályt annyiszor példányosítjuk (annyi, az adott típusba tartozó objektumot hozunk létre), ahányszor csak kedvünk tartja. Az osztály létrehozásának további előnye lenne, hogy elkerülhetnénk a `global` utasítás használatát az osztályszintű (statikus) adatok tárolásakor. A következő fejezetben meg is ismerkedünk az osztályok létrehozásának módjával.

A Python alapkönyvtárának áttekintése

A Python alapkönyvtárárárról szólva gyakran mondják, hogy minden benne van. És tényleg: a majdnem kétszáz csomag és modul a felmerülő problémák széles körére kínál kész megoldásokat. Ebben a részben tematikus megközelítésben vesszük sorra, hogy mi mindenre találunk megoldást, kihagyva a túlzottan speciális problémák kezelésére való, illetve a csak egyetlen operációs rendszeren futó csomagokat és modulokat. Sok esetben rövid példával adunk ízelítőt az adott csomag vagy modul használatából, a könyv más részein tárgyalt csomagokra és modulokra pedig a megfelelő helyen hivatkozunk.

Karakterláncok kezelése

A `string` modulban megtalálható néhány olyan hasznos állandó, mint a `string.ascii_letters` (ASCII-betűk) és a `string.hexdigits` (a tizenhatos – hexadecimális – számrendszerben használt számok). Itt találjuk a `string.Formatter` osztályt, amelyből alosztályokat képezve egyedi formázókat hozhatunk létre.* A `textwrap` modul arra használható, hogy egy szöveg sorait adott szélességűvé alakítsa, és a behúzásokat csökkentse.

A `struct` modulban számok, logikai értékek és karakterláncok `bytes` osztályú objektumokba való csomagolására és a belőlük való kicsomagolására szolgáló, a kettes számrendszerű (bináris) ábrázolásukat használó függvényeket találunk. Ez akkor lehet hasznos, amikor alacsony szintű, C nyelven írt könyvtárakkal cserélünk adatokat. A 7. fejezetben tárgyalt `convert-incidents.py` program a `struct` és a `textwrap` modult egyaránt használja.

`bytes` típus ☞ 297

`struct` modul ☞ 296

A `difflib` modulban olyan osztályokat és tagfüggvényeket találunk, amelyekkel sorozatok – például karakterláncok – összehasonlítása végezhető el, a kimenetet pedig a `diff` parancs megszokott formájában vagy akár HTML-ben is kérhetjük. A Python legügyesebb karakterlánc-kezelő modulja az `re` (regular expression) modul. Erről a modulról a 12. fejezet szól.

Az `io.StringIO` osztályból olyan, karakterlánchoz hasonló objektumot példányosíthatunk, amely a memóriában tárolt szövegfájlként viselkedik. Ez igen kényelmes lehet, ha a fájlok írására használatos kódot szeretnénk karakterlánc írására használni.

Példa: Az io.StringIO osztály

A Python nyelv két különböző módon teszi lehetővé szöveg fájlba írását. Az egyik módszer a fájlt ábrázoló objektum `write()` tagfüggvényének használata, a másik pedig az, ha a `print()` függvényt használjuk, és a `file` kulcsszó-paraméter értékéül az írásra megnyitott fájlt adjuk meg. Például:

---

* Az *alosztályképzés* vagy a *specializáció* szakkifejezést arra használjuk, amikor egy egyedi adattípust – vagyis egy osztályt – egy másik objektumosztály alapján hozunk létre. A 6. fejezet teljes körűen tárgyalja a témát.

```
print("An error message", file=sys.stdout)
sys.stdout.write("Another error message\n")
```

Mindkét kiírt szövegsor a `sys.stdout` objektumra kerül, ami egy fájlobjektum, és a „szabványos kimeneti csatornának" felel meg. Ez rendes esetben a konzol, és annyiban különbözik a „szabványos hibacsatornától", hogy ez utóbbi nem rendelkezik átmeneti tárral. (A Python a programok elindulásakor automatikusan létrehozza a `sys.stdin` (szabványos bemenet), a `sys.stdout`, és a `sys.stderr` csatornát.) A `print()` függvény alapértelmezés szerint a sor végén sortörést is elhelyez, bár ez meggátolható, ha átadjuk neki az üres karakterlánc értékűre állított `end` kulcsszó-paramétert.

Vannak helyzetek, amikor hasznos lehet, ha a fájlba irányított kimenetet egy karakterlánccal elkapjuk. Ez a `io.StringIO` osztály segítségével tehető meg, amelyből a fájlobjektumokkal azonos módon használható, de a beleírt adatot karakterláncban tároló objektum példányosítható. Ha a `io.StringIO` osztályú objektumnak átadunk egy kezdeti karakterláncot, akkor az is úgy olvasható vissza, mintha egy fájlba került volna. A `io.StringIO` osztály az `import io` parancs kiadása után érhető el, és használható például a `sys.stdout` fájlobjektumnak átadott kimenet elfogására:

```
sys.stdout = io.StringIO()
```

Ha a fenti sort elhelyezzük egy program elején, a betöltések után, de még mielőtt a `sys.stdout` objektumot használnánk, minden, a `sys.stdout` objektumba küldött szöveg valójában a fenti sorral létrehozott és az alapértelmezett `sys.stdout` objektumot lecserélő `io.StringIO` osztályú, fájlként viselkedő objektumba kerül. Ha a korábban bemutatott, a `print()` és a `sys.stdout.write()` utasítást tartalmazó sorokat ekkor hajtjuk végre, a kimenetük a konzol helyett az `io.StringIO` osztályú objektumba kerül. (Az eredeti `sys.stdout` objektum a `sys.stdout = sys.__stdout__` utasítással visszaállítható.)

Az `io.StringIO` osztályú objektumba került valamennyi karakterlánchoz hozzáférhetünk, ha az `io.StringIO.getvalue()` tagfüggvényt hívjuk, esetünkben a `sys.stdout.getvalue()` formában. A tagfüggvény visszatérési értéke a beírt összes sort tartalmazó karakterlánc. Ezt a karakterláncot kiírhatjuk a képernyőre, menthetjük egy naplófájlba, vagy továbbíthatjuk a hálózaton – bármely más karakterlánchoz hasonlóan. Az `io.StringIO` osztály használatára a fejezetben kicsit később újabb példát is látunk majd.

Parancssori programozás

Ha olyan programra van szükségünk, amely a konzolon átirányított vagy a parancssorban felsorolt fájlokban lévő szöveg feldolgozására képes, érdemes lehet a `fileinput` modul `fileinput.input()` függvényét használnunk. Ez a függvény bejárja a konzolról átirányított sorokat – ha vannak ilyenek –, illetve a parancssorban felsorolt fájlokat, méghozzá sorok folyamatos sorozataként. A modul a `fileinput.filename()` és a `fileinput.lineno()` függvényekkel képes bármely pillanatban megadni az aktuális fájlnevet és sort, és képes néhány tömörített fájlformátum kezelésére is.

A parancssori paraméterek kezelésére két külön modul áll rendelkezésre: az `optparse` és a `getopt`. A `getopt` modul azért népszerű, mert egyszerű, és már régóta része az alapkönyvtárnak. Az `optparse` modul újabb és sokoldalúbb.

Példa: Az optparse modul

A `csv2html.py` példaprogram
91

Még a második fejezetben tárgyaltuk a `csv2html.py` programot. A fejezet gyakorlatai között ajánlottuk, hogy a programot tegyük képessé a `maxwidth` (maximális szélesség) és a `format` (formátum) paraméterek kezelésére; az előbbi egy egész szám, az utóbbi egy karakterlánc. A megadott megoldás (`csv2html2_ans.py`) 26 sorban dolgozza fel a paramétereket. Íme a `csv2html2_opt.py` program `main()` függvényének eleje, amelyben a parancssori paraméterek feldolgozására saját függvény helyett az `optparse` modult használjuk:

```
def main():
    parser = optparse.OptionParser()
    parser.add_option("-w", "--maxwidth", dest="maxwidth", type="int",
        help=("the maximum number of characters that can be "
              "output to string fields [default: %default]"))
    parser.add_option("-f", "--format", dest="format",
            help=("the format used for outputting numbers "
                  "[default: %default]"))
    parser.set_defaults(maxwidth=100, format=".0f")
    opts, args = parser.parse_args()
```

Mindössze kilenc sornyi kódra volt szükség, meg persze az `import optparse` kifejezésre. Mi több, a továbbiakban nincs szükség a `-h` és a `-help` kapcsoló külön megadására. Ezeket mostantól az `optparse` modul kezeli, megfelelő használati utasítást készítve a `help` kulcsszó-paraméterekből, a `%default` szövegeket a kapcsoló alapértelmezett értékével helyettesítve.

Figyeljük meg azt is, hogy a kapcsolók immár a hagyományos Unix-stílust követik: rövid és hosszú, kötőjelekkel kezdődő nevekkel rendelkeznek. A rövid nevek a konzolban végrehajtott interaktív műveletek során kényelmesek, a hosszú nevek pedig érthetőbbek, ha héjprogramokban használjuk őket. Például a maximális szélesség értékének beállításához egyaránt használható a `-w80`, a `-w 80`, a `–width=80` és a `–width 80` forma. Miután a parancssort feldolgoztuk, a paraméterek a `dest` paranccsal megadott neveken érhetők el, például `opts.maxwidth` és `opts.format` néven. Azok a parancssori paraméterek, amelyeknek a feldolgozása nem történt meg (ezek rendszerint fájlnevek), az `args` listába kerülnek.

Ha a parancssor feldolgozása során valami nem sikerül, az `optparse` modul elvégzi a `sys.exit(2)` függvényhívást. Ez a program tiszta megszakításához vezet, az operációs rendszer pedig a programtól a 2-es visszatérési értéket kapja. A 2-es visszatérési érték hagyományosan felhasználói hibát jelez, az 1 egyszerűen csak hibát, a 0 pedig sikeres futást. Ha a `sys.exit()` függvényt paraméterek nélkül hívjuk meg, 0-át ad vissza az operációs rendszernek.

Matematika és számok

A beépített `int` (egész), `float` (lebegőpontos) és `complex` (komplex) számokat tároló típusokon felül az alapkönyvtár lehetőséget nyújt a tizedestörtek és az emeletes törtek kezelésére is (`decimal.Decimal`, `fractions.Fraction`). Három számfeldolgozó könyvtár áll a rendelkezésünkre: a `math` a racionális számokkal dolgozó matematikai függvényeket, a `cmath` a komplex számokat használó függvényeket tartalmazza, a `random` pedig a véletlenszámok előállítására használható függvények sokaságával szolgál. Ezeket a modulokat a második fejezetben ismertük meg.

A Python nyelv számfeldolgozó elvont alaposztályai (olyan objektumosztályok, amelyekből származtatható osztály, de a közvetlen példányosításuk nem lehetséges) a `numbers` modulban találhatók. Ezek akkor hasznosak, ha egy objektumról, mondjuk *x*-ről, el akarjuk dönteni, hogy valamilyen szám-e: `isinstance(x, numbers.Number)`, vagy valamilyen adott típusú szám-e: `isinstance(x, numbers.Rational)`, `sinstance(x, numbers.Integral)`.

Azok, akik a tudományos vagy mérnöki számítások elvégzésében érdekeltek, hasznosnak fogják találni a `NumPy` nevű, külső fejlesztőktől származó csomagot. Ebben igen hatékony n-dimenziós tömböket, alapvető lineáris algebrai függvényeket, Fourier-transzformációkat, illetve a C, C++ és Fortran nyelvekkel való együttműködést segítő eszközöket találunk. A `SciPy` csomagnak is része a `NumPy` csomag, és ezen felül találhatók benne statisztikai számításokat, jel- és képfeldolgozást segítő eszközök, genetikai algoritmusok, és sok minden más. Mindkét csomag ingyenesen elérhető a `www.scipy.org` webhelyen.

Dátumok és időpontok

A `calendar` és a `datetime` modul az idő- és a dátumkezelés megkönnyítésére szolgáló osztályokat és függvényeket tartalmaz. Mindkét modul a javított Gergely-naptáron alapul, a korábbi dátumok kezelésére alkalmatlanok. Az idő- és a dátumkezelés meglehetősen összetett probléma: a használatban lévő naptárak eltérő helyeken, eltérő időpontokban mások és mások, egy nap nem pontosan 24 óra, egy év nem pontosan 365 nap, és még akkor nem is beszéltünk a téli-nyári időszámításból és az időzónák létéből eredő problémákról. A `datetime.datetime` osztályt felkészítették az időzónák kezelésére (nem így a `datetime.date` osztályt), de ezt nem végzi el automatikusan. Külső fejlesztőktől származó modulok próbálják orvosolni a problémát, például a `www.labix.org/python-dateutil` webhelyről származó `dateutil`, illetve a `www.egenix.com/products/python/mxBase/mxDateTime` webhelyről származó `mxDateTime` modul.

A `time` modul az időbélyegeket kezeli. Az időbélyegek egyszerű számok, amelyek a Unix rendszerek időszámításának kezdetétől, vagyis 1970-01-01 00:00:00-tól kezdve eltelt másodperceket jelentik. Ez a modul használható arra, hogy megtudjuk a számítógép aktuális, UTC (koordinált világidő) szerinti időbélyegét, illetve a helyi, téli-nyári időszámítást figyelembe vevő időt. A modul képes továbbá különbözőképpen formázott dátumok és időpontok megjelenítésére, valamint olyan karakterláncok feldolgozására, amelyek dátumokat és időpontokat tárolnak.

Példa: A calendar, a datetime és a time modul

A `datetime.datetime` osztály objektumait rendszerint programból állítjuk elő, ugyanakkor azok az objektumok, amelyek UTC szerinti dátumokat és időpontokat tárolnak, rendszerint külső forrásból, például fájlok időbélyegéből érkeznek. Íme néhány példa:

```
import calendar, datetime, time
moon_datetime_a = datetime.datetime(1969, 7, 20, 20, 17, 40)
moon_time = calendar.timegm(moon_datetime_a.utctimetuple())
moon_datetime_b = datetime.datetime.utcfromtimestamp(moon_time)
moon_datetime_a.isoformat()     # visszaadott időpont:
                                # '1969-07-20T20:17:40'
moon_datetime_b.isoformat()     # visszaadott időpont:
                                # '1969-07-20T20:17:40'
time.strftime("%Y-%m-%dT%H:%M:%S", time.gmtime(moon_time))
```

A `moon_datetime_a` változó a `datetime.datetime` típusba tartozik, és azt az időpontot tárolja, amikor az Apollo 11 leszállt a Holdon. A `moon_time` változó `int` típusú, és a holdraszállástól számított másodperceket tárolja – a számot a `calendar.timegm()` függvénytől nyeri, amely a `datetime.datetime.utctimetuple()` függvény által visszaadott `time_struct` objektumot kapja paraméterül, és a benne ábrázolt másodpercek számát adja vissza. (Minthogy a holdraszállás a Unix időszámítás kezdete előtt történt, a szám negatív.) A `moon_datetime_b` változó is a `datetime.datetime` típusba tartozik, és a `moon_time` egész típusú változóból hozzuk létre, bemutatva az időszámítás kezdőpontja óta eltelt másodpercek átalakítását `datetime.datetime` osztályú objektummá.* Az utolsó három sor ugyanazt az ISO 8601 szabványban leírt formátumú dátumot, illetve időt adja vissza.

Az aktuális, UTC szerinti dátum és idő megkapható `datetime.datetime` osztályú objektumként, ha a `datetime.datetime.utcnow()` függvényt, illetve az időszámítás kezdőpontjától eltelt másodpercekben kifejezve, ha a `time.time()` függvényt hívjuk. A helyi idő kinyeréséhez a `datetime.datetime.now()`, illetve a `time.mktime(time.localtime())` függvények valamelyikét érdemes használnunk.

Algoritmusok és gyűjteményes adattípusok

A `bisect` modulban olyan függvények találhatók, amelyek rendezett sorozatokban, például rendezett listákban tudnak keresni, illetve képesek a rendezettség megtartása mellett új elemek beszúrására. A `modul` függvényei a bináris keresés algoritmusát használják, így nagyon gyorsak. A `heapq` modulban sorozatok, például listák kupacokká alakításához találunk függvényeket (a kupac – `heap` – olyan gyűjteményes adattípus, ahol az első, vagyis 0. sorszámú elem mindig a legkisebb), valamint a kupacforma megtartása mellett történő elembeszúráshoz és törléshez.

---

* A Windows-felhasználókat érinti, hogy a `datetime.datetime.utcfromtimestamp()` függvény képtelen a negatív, vagyis az 1970. január 1. előtti időbélyegek kezelésére.

Alapértelmező szótár 179 ☜

Nevesített rendezett sorozat 105 ☜

A `collections` csomagban kapott helyet a korábban már tárgyalt `collections.defaultdict` (alapértelmező szótár) és a `collections.namedtuple` (nevesített rendezett sorozat) gyűjteményes adattípus. A csomagban ezen felül megtaláljuk a `collections.UserList` és a `collections.UserDict` nevű típusokat, bár talán gyakrabban használják a beépített `list` és `dict` osztályokból származtatott osztályokat, mint a fenti típusokat. Szintén itt találjuk a listához hasonló `collections.deque` (kétvégű sor) típust, amely a listához hasonló, de míg a listának csak a végén tudunk gyorsan új elemet beszúrni és eltávolítani, a kétvégű sornál ezek a műveletek a sor elején és végén is nagyon gyorsan működnek.

A Python nem számokkal foglalkozó elvont alaposztályai (olyan osztályok, amelyekből származtatható osztály, de a közvetlen példányosításuk nem lehetséges) szintén a `collections` csomagban találhatók. Ezeket a 8. fejezetben tárgyaljuk.

Az `array` (tömb) modulba került az `array.array` sorozattípus, amely a számokat és a karaktereket igen helyhatékony módon tárolja. A `list` típushoz hasonló a viselkedése, azzal a kivétellel, hogy a létrehozásakor eldől, hogy mely osztályba tartozó objektumok tárolására képes, vagyis a listáktól eltérően csak egyetlen típusba tartozó objektumokat tárolhat. A már korábban említett, külső fejlesztésű `NumPy` csomag szintén hatékony tömböket tartalmaz.

Programunkat a `weakref` modul tudja felruházni a gyenge hivatkozások létrehozásának képességével. A gyenge hivatkozások a rendes objektumhivatkozásokhoz hasonlóan működnek, kivéve, hogy amennyiben az objektumra már csak gyenge hivatkozás mutat, akkor a szemétgyűjtő felszabadíthatja az objektum által elfoglalt memóriát. Ezen a módon megakadályozható, hogy egy objektum csak azért maradjon a memóriában, mert mutat rá objektumhivatkozás. Természetesen mielőtt megpróbálnánk elérni, ellenőrizhető, hogy a gyenge hivatkozás által hivatkozott objektum létezik-e még.

Példa: A heapq modul

A `heapq` modulban találjuk azokat a függvényeket, amelyekkel egy listát kupaccá alakíthatunk, illetve a kupachoz új elemet adhatunk, vagy kivehetjük belőle a már ott lévőket – mindezt a kupactulajdonság megőrzése mellett. A kupac tulajdonképp olyan bináris fa, amelyben igaz, hogy az első (0. sorszámú) elem mindig a legkisebb.* A kupac részfái is kupacok, vagyis ezek is megőrzik a kupactulajdonságot. Íme egy kupac létrehozása a semmiből:

```
import heapq
heap = []
heapq.heappush(heap, (5, "rest"))
heapq.heappush(heap, (2, "work"))
heapq.heappush(heap, (4, "study"))
```

* Szigorú értelemben a `heapq` modul *minimumkupacok* kezelésére képes. Azokat a kupacokat, amelyeknek az első eleme mindig a legnagyobb, *maximumkupacoknak* nevezzük.

Ha már van egy rendelkezésre álló listánk, akkor azt a `heapq.heapify(lista)` hívással alakíthatjuk kupaccá. A tagfüggvény a szükséges újrarendezést helyben végzi el. A kupacból a legkisebb elemet a `heapq.heappop` (*kupac*) hívással távolíthatjuk el.

```
for x in heapq.merge([1, 3, 5, 8], [2, 4, 7], [0, 1, 6, 8, 9]):
    print(x, end=" ") # prints: 0 1 1 2 3 4 5 6 7 8 8 9
```

A `heapq.merge()` függvénynek akárhány bejárható adatot átadhatunk paraméterül, a visszatérési értéke pedig az összes átadott adat elemeit a megfelelő sorrendben tartalmazó bejárható objektum lesz.

Fájlformátumok, kódolások és az adatok megőrzése

Karakterkódolások 86 ☜

Az alapkönyvtár kiterjedt támogatással rendelkezik a szabványos fájlformátumok és kódolások terén. A `base64` modul az RFC 3548 dokumentumban* megadott Base16, Base32 és Base64 kódolású adatok olvasására és írására való függvények helye. A `quopri` modulban találjuk a „quoted-printable" formátum írására és olvasására való függvényeket. A formátumot az RFC 1521 dokumentum adja meg, és a MIME- (Multipurpose Internet Mail Extensions) adatok kapcsán használatos. Az `uu` modulban találjuk az `uu`, vagyis Unix-Unix kódolású adatok írására és olvasására való függvényeket. Az RFC 1832 dokumentum adja meg az External Data Representation Standard (külső adatok megjelenítésének szabványa) követelményeit; az ilyen formátumú adatok írására és olvasására szolgáló függvényeket az `xdrlib` modulban találjuk.

Az alapkönyvtárban megtalálhatók a legnépszerűbb archívum-formátumok kezelését végző modulok is. A `bz2` modul a `.bz2` kiterjesztésű fájlok kezelését végzi, a `gzip` modul a `.gz` kiterjesztésűekét, a `tarfile` modul a `.tar`, a `.tar.gz`, illetve `.tgz`, valamint a `.tar.bz2` formátumúakét, a `zipfile` modul pedig a `.zip` kiterjesztésűekét. A `tarfile` modul használatára még ebben a részben látunk példát, később pedig egy rövid példa mutatja be a `gzip` modul néhány lehetőségét. A `gzip` modul működését a 7. fejezetben is tanulmányozhatjuk.

Az alapkönyvtár néhány hangfájlformátum kezelését is támogatja: az `aifc` modul az AIFF (Audio Interchange File Format) formátumot, a `wave` modul pedig a tömörítetlen, `.wav` kiterjesztésű fájlok használatát segíti. Egyes hangformátumok az `audioop` modullal kezelhetők, az `sndhdr` modul pedig néhány, a fájlban tárolt hangadatok jellemzőit (például mintavételezési gyakoriság), és egyéb tulajdonságait kideríteni képes függvénnyel szolgál.

A beállítófájlok – a Windows operációs rendszer régi `.ini` fájljaihoz hasonló – egyik formátumát adja meg az RFC 822 dokumentum; az ilyen fájlok írására és olvasására való függvényeket a `configparser` modulban találjuk.

---

* Az RFC, vagyis Request for Comments (véleménynyilvánításra beterjesztett) dokumentumok írják le az Interneten használatos különféle technológiákat. Mindegyik dokumentum saját azonosítószámmal bír, és sokból hivatalosan is elfogadott szabvány lett.

Sok alkalmazás – például az Excel – képes olvasni a CSV (Comma Separated Values, vesszővel tagolt értékek) formátumú fájlokat, illetve ezek változatait – például a tabulátorokkal tagolt adatokat tartalmazókat. A `csv` modul képes az ilyen formátumú fájlok olvasására és írására, és kezeli az olyan egyéni hóbortokat, amelyek megnehezítik a CSV formátumú fájlok egyszerű, azonnali kezelését.

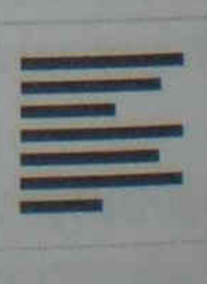
Azon felül, hogy az alapkönyvtár különféle fájlformátumok kezelését támogatja, találunk benne az adatok megőrzését szolgáló modulokat is. A `pickle` modul használható Python-objektumok, akár teljes gyűjtemények lemezre írására és onnan történő visszaolvasására – a modult a 7. fejezetben tárgyaljuk.

Az alapkönyvtár támogatja a különféle adatbázis-formátumokat (DBM) is – ezek a szótár típushoz hasonlóak, de az elemeiket nem a memóriában, hanem lemezen tárolják, a kulcsaik és értékeik pedig egyaránt vagy `bytes` típusú objektumok, vagy karakterláncok. A `shelve` modul, amelyet a 11. fejezetben tárgyalunk, olyan DBM-fájlok kialakítására használható, amelyeknek a kulcsa karakterlánc, az értéke pedig bármely nekünk tetsző, a Pythonban megvalósított osztályba tartozó objektum lehet – a modul a színfalak mögött gond nélkül elvégzi a Python objektumai és a `bytes` formátum közötti átalakításokat. A DBM-modulokról, a Python adatbázis-kezelési alkalmazásprogramozási felületéről és a beépített SQLite adatbázis-kezelőről a 11. fejezet szól.

Példa: A base64 modul

A `base64` modult leginkább az elektronikus levelekbe ASCII formátumú szövegként beágyazott bináris adatok kezelésére használják, de használható bináris adatok Python-programállományokban való tárolására is. Az alábbi kódrészlet feltételezi, hogy a `base64` modult előzőleg betöltöttük, illetve hogy a `left_align_png` változó egy `.png` formátumú fájl elérési útját és nevét tárolja.

```
binary = open(left_align_png, "rb").read()
ascii_text = ""
for i, c in enumerate(base64.b64encode(binary)):
if i and i % 68 == 0:
ascii_text += "\\\n"
ascii_text += chr(c)
```



left_align.png

`bytes` típus
☞ 297

A fenti kódrészlet bináris fájlként olvassa be az állományt, majd ASCII karakterekből álló, Base64 kódolásúvá alakítja. Minden hatvannyolcadik karakternél elhelyez egy fordított perjelből és egy újsor karakterből álló kombinációt. Ezzel az ASCII-karaktereket tartalmazó sorok hosszát 68 karakterben korlátozza, és vigyáz arra is, hogy az adat visszaolvasásakor az újsor karakterek ne legyenek figyelembe véve (a fordított perjel ugyanis levédi azokat). Az ekképpen nyert, ASCII-kódolású szöveget a Python-programjainkban `bytes` típusú adatként közvetlenül is tárolhatjuk, például így:

```
LEFT_ALIGN_PNG = b"""\
iVBORw0KGgoAAAANSUhEUgAAACAAAAAgCAYAAABzenr0AAAABGdBTUEAALGPC/xhBQAA\
...
bmquu8PAmVT2+CwVV6rCyA9UfFMCkI+bN6p18tCWqcUzrDOwBh2zVCR+JZVeAAAAAE1F\
TkSuQmCC"""
```

Ahogy a három pont jelzi, a sorok nagy részét a fenti részletben elhagytuk.

Az adatok az alábbi módszerrel alakíthatók vissza az eredeti formájukra:

```
binary = base64.b64decode(LEFT_ALIGN_PNG)
```

Az `open(`*fájlnév*`, "wb").write(binary)` utasítással az adatok fájlba írhatók. A bináris adatok Python-programfájlokban való tárolása lényegesen több helyet igényel, mintha az eredeti formájukban tárolnánk őket, de hasznos lehet, ha olyan programot akarunk írni, amelynél a programfájlban ilyen adatokat kell elhelyeznünk.

Példa: A tarfile modul

A Windows legtöbb változata nem támogatja a Unix rendszereken olyannyira elterjedt `.tar` formátumot. Az ebből eredő kényelmetlenség gyorsan orvosolható a Python `tarfile` moduljával, amely képes `.tar` és `.tar.gz` formátumú archívumok (*tarball* néven is hivatkoznak rájuk) létrehozására és kicsomagolására. Ha a megfelelő könyvtárak telepítve vannak, a `.tar.bz2` formátum is kezelhető a modullal. Az `untar.py` program a `tarfile` modul használatával képes a fenti formátumú archívumok kicsomagolására. Az alábbiakban csak a program kulcsfontosságú részeit mutatjuk be, kezdve az első `import` utasítást tartalmazó kifejezéssel:

```
BZ2_AVAILABLE = True
try:
    import bz2
except ImportError:
    BZ2_AVAILABLE = False
```

A `bz2` modult használjuk a `bzip2` tömörített formátum kezelésére, de ha a Python fordítása úgy történt, hogy a bzip2 programkönyvtár nem volt hozzáférhető, a modul sem lesz betölthető. (A Python Windowsra telepíthető változata mindig tartalmazza a bzip2 tömörítést, legfeljebb néhány Unix-telepítés esetében hiányozhat a tömörítési módszer támogatása.) Annak lehetőségét, hogy a modul esetleg elérhetetlen, a `try...except` utasítások közötti kódrésszel vesszük figyelembe, a vizsgálat eredményét egy később is használható logikai változóban tárolva (bár a változót használó kódrészlet a fentiekben nem szerepel).

```
UNTRUSTED_PREFIXES = tuple(["/", "\\"] +
    [c + ":" for c in string.ascii_letters])
```

A fenti kifejezés a `('/', '\', 'A:', 'B:', ..., 'Z:', 'a:', 'b:', ..., 'z:')` rendezett sorozatot hozza létre. A kicsomagolás alatt lévő, `.tar` formátumú fájlban lévő minden olyan állomány, amely a rendezett sorozatban lévő karakterek valamelyikével kezdődik,

gyanús. A .tar formátumú fájloknak nem lenne szabad abszolút elérési utakat tartalmazniuk, hiszen így azt kockáztatjuk, hogy felülírjuk valamelyik rendszerfájlt. Így elővigyázatosságból kicsomagolatlanul hagyjuk azokat a fájlokat, amelyeknek a neve a rendezett sorozatban lévő karakterek valamelyikével kezdődik.

```
def untar(archive):
    tar = None
    try:
        tar = tarfile.open(archive)
        for member in tar.getmembers():
            if member.name.startswith(UNTRUSTED_PREFIXES):
                print("untrusted prefix, ignoring", member.name)
            elif ".." in member.name:
                print("suspect path, ignoring", member.name)
            else:
                tar.extract(member)
                print("unpacked", member.name)
    except (tarfile.TarError, EnvironmentError) as err:
        error(err)
    finally:
        if tar is not None:
            tar.close()
```

A .tar formátumú fájlban lévő valamennyi állomány az archívum *tagja* (member). A tarfile.getmembers() függvény tarfile.TarInfo osztályú objektumokból álló listát ad vissza – tagonként egy objektumot. A taghoz tartozó fájlnév, beleértve az elérési utat, a tarfile.TarInfo.name jellemzőből olvasható ki. Ha a név valamilyen gyanús karakterrel kezdődik, vagy .. található az elérési útban, hibaüzenetet írunk ki, egyéb esetben a tarfile.extract() függvény hívásával lemezre mentjük a tagot. A tarfile modul saját kivételek teljes készletével bír, mi azonban azt az egyszerű megközelítési módot választottuk, hogy akármilyen kivételes esemény bekövetkeztekor kiírjuk a hibaüzenetet, és kilépünk.

```
def error(message, exit_status=1):
    print(message)
    sys.exit(exit_status)
```

Az error() függvény csak a teljesség kedvéért került ide. A fentiekben nem szereplő main() függvény felhasználási utasítást ír ki, ha a -h, vagy a –help kapcsolóval indítják a programot, egyéb esetekben pedig végrehajt néhány alapvető ellenőrzést, és meghívja az untar() függvényt.

Fájlok, könyvtárak és folyamatok kezelése

Az shutil modul magas szintű fájl- és könyvtárkezelő függvényeket bocsát a rendelkezésünkre, többek között az shutil.copy() és az shutil.copytree() nevűt, amelyekkel fájlok, illetve teljes könyvtárfák másolása lehetséges. Az shutil.move() függvény könyvtárfák mozgatását végzi, az shutil.rmtree() függvénnyel pedig teljes könyvtárfák eltávolítása lehetséges – olyanoké is, amelyek nem üresek.

Ideiglenes fájlok létrehozására a `tempfile` modult érdemes használni, amelyben megtalálhatók a szükséges függvények – például a `tempfile.mkstemp()` –, és a fájlok létrehozásánál a lehető legnagyobb biztonsági követelményeknek igyekszik megfelelni.

A `filecmp` modul `filecmp.cmp()` függvényével fájlokat, a `filecmp.cmpfiles()` függvényével pedig teljes könyvtárakat hasonlíthatunk össze.

A Python nyelvű programok igen hatékonyak és sokoldalúak lehetnek egyéb programok futásának vezénylőjeként. Ilyenkor használhatjuk a `subprocess` modult, amely képes egyéb folyamatok elindítására, a velük csővezetéken keresztül való kommunikációra, és a tőlük kapott eredmények fogadására. A modullal a 9. fejezetben foglalkozunk. Még ennél a modulnál is többre képes a `multiprocessing` modul, amely kiterjedt eszköztárával képes egy feladatot több folyamat között elosztani, és az eredményeket összesíteni – eképpen többszálú program készítése helyett gyakran lehet megfontolandó választás.

Az `os` modul operációs rendszertől független hozzáférést biztosít az operációs rendszer bizonyos részeihez. Az `os.environ` változó egy megfeleltetési típusú objektumot tárol, amelynek elemei a környezeti változók nevei és értékei. A program munkakönyvtárát az `os.getcwd()` függvény mutatja meg, a megváltoztatása pedig az `os.chdir()` függvénnyel lehetséges. A modulban találhatók olyan függvények is, amelyek az alacsonyszintű, fájlleíró alapú fájlkezelést végzik. Az `os.access()` függvény használható egy fájl létének, olvashatóságának, írhatóságának ellenőrzésére, az `os.listdir()` a neki átadott könyvtár tartalmát adja vissza (vagyis a fájljait és könyvtárait, de a `.` és a `..` könyvtárbejegyzéseket nem), az `os.stat()` függvény pedig különféle információkat ad vissza az aktuális fájlról, illetve könyvtárról, például a módját (ki olvashatja, ki írhatja, ki hajthatja végre), az utolsó elérése idejét és a méretét.

Könyvtárak létrehozására az `os.mkdir()` függvény használható, ha pedig köztes könyvtárakat is létre kell hozni, használjuk az `os.makedirs()` függvényt. Az üres könyvtárak eltávolítása az `os.rmdir()` függvénnyel, a csak üres könyvtárakat tartalmazó könyvtárak eltávolítása az `os.removedirs()` függvénnyel lehetséges. Fájlok és könyvtárak eltávolítására az `os.remove()`, átnevezésére az `os.rename()` függvény használható.

Az `os.walk()` függvény teljes könyvtárfák bejárására alkalmas, és az egyes fájlok, illetve könyvtárak nevét adja vissza.

Az `os` modul tartalmaz néhány alacsonyszintű, csak egy-egy operációs rendszerre jellemző függvényt is, például a fájlleírókkal való munkára, folyamatok elágaztatására (csak Unix-rendszereken), más folyamat indítására, illetve leszármaztatására.

Míg az `os` modul függvényei főleg az operációs rendszerrel való érintkezésre valók, különös tekintettel a fájlrendszerre, az `os.path` modul egyedi keverékét tartalmazza az elérési utak karakterláncokként való kezelésének, illetve a fájlrendszerrel kapcsolatos, a kényelmünket

szolgáló függvényeknek. Az `os.path.abspath()` függvény a neki átadott paraméter abszolút elérési útját adja vissza, a többször szereplő elválasztók, illetve a .. elemek nélkül. Az `os.path.split()` függvény visszatérési értéke egy kételemű rendezett sorozat, amelynek az első eleme az elérési utat, a második a fájlnevet tartalmazza – ez utóbbi üres, ha fájlnév nélküli elérési utat adtunk meg. A két rész közvetlenül is hozzáférhető az `os.path.basename()` és az `os.path.dirname()` függvénnyel. Az `os.path.splitext()` függvény két részre osztja a fájlneveket: névre és kiterjesztésre. Az `os.path.join()` függvény paraméterei olyan karakterláncok, amelyek elérési utakban szerepelhetnek, a visszatérési értéke pedig egyetlen elérési út, amelynek a részei között az adott operációs rendszerre jellemző elválasztó karakter található.

Ha sok különféle információra vagyunk kíváncsiak egy adott fájlt, illetőleg könyvtárat illetőleg, az `os.stat()` függvényt, ha azonban csak egy adott információt szeretnénk kideríteni, az `os.path` modul megfelelő függvényét használhatjuk. A fájl, illetve könyvtár létéről az `os.path.exists()`, a méretéről az `os.path.getsize()` függvény nyújt felvilágosítást, az `os.path.isfile()` és az `os.path.isdir()` függvénnyel pedig eldönthető, hogy az éppen vizsgált objektum fájl vagy könyvtár-e.

A `mimetypes` modulba került a `mimetypes.guess_type()` függvény, amely az átadott fájl MIME-típusát próbálja kitalálni.

Példa: Az os és az os.path modul

Az alábbiakban bemutatjuk, hogy miként hozhatunk létre az `os`, illetve az `os.path` modullal olyan szótárat, amelynek a kulcsai az elérési utat is tartalmazó fájlnevek, az értékei pedig az elérési útban megadott fájlok utolsó módosítási idejét megmutató időbélyegek (a Unix-időszámítás kezdete óta eltelt másodpercek száma):

```
date_from_name = {}
for name in os.listdir(path):
    fullname = os.path.join(path, name)
    if os.path.isfile(fullname):
        date_from_name[fullname] = os.path.getmtime(fullname)
```

A kód meglehetősen magától értetődő, és csak egyetlen könyvtárban lévő fájlok dolgozhatók fel vele. Ha egy egész könyvtárfát kell végigjárnunk, az `os.walk()` függvény lehet a segítségünkre.

Egy kódrészlet következik a többször előforduló fájlok keresését végző `finddup.py` programból.* A kód egy olyan szótárat alakít ki, amelynek minden kulcsa egy-egy kételemű, egy fájl méretét és elérési út nélküli nevét tároló rendezett sorozat (tuple), az egyes értékek pedig olyan listák, amelyek a kulcsban tárolt fájllal megegyező méretű és nevű fájlok elérési utat is tartalmazó neveit tárolják:

---

* Egy szintén a többször előforduló fájlok keresését végző, ám lényegesen kifinomultabb programot tárgyalunk a 9. fejezetben. A `findduplicates-t.py` többszálú, és a működése MD5 ellenőrzőösszegek képezésén alapul.

```
data = collections.defaultdict(list)
for root, dirs, files in os.walk(path):
for filename in files:
fullname = os.path.join(root, filename)
key = (os.path.getsize(fullname), filename)
data[key].append(fullname)
```

Az egyes könyvtárak tekintetében az `os.walk()` függvény visszaadja a gyökérkönyvtárat, valamint két listát is: az egyik az adott könyvtár alkönyvtárait, a másik a benne lévő fájlokat tartalmazza. A teljes elérési út képzése a gyökérkönyvtár és a tárolt fájl nevének kombinációjával oldható meg. Figyeljük meg, hogy nem a mi feladatunk a könyvtárak alkönyvtárainak bejárása – az `os.walk()` függvény megteszi ezt helyettünk. Ha az adatgyűjtéssel végeztünk, a végigjárás következik, amelynek az eredménye az a jelentés, amely az esetleg többször előforduló fájlok neveit tartalmazza:

```
for size, filename in sorted(data):
    names = data[(size, filename)]
    if len(names) > 1:
        print("{0} ({1} bytes) may be duplicated "
              "({2} files):".format(filename, size, len(names)))
        for name in names:
            print("\t{0}".format(name))
```

Minthogy a szótár kulcsai (*méret*, *fájlnév*) alakú rendezett sorozatok, nem kell a `key` függvénnyel méret szerinti sorrendbe állítani az adatokat. Ha bármely (*méret*, *fájlnév*) rendezett sorozat listájában egynél több fájlnév szerepel, valószínűleg többszöri előfordulással állunk szemben.

```
...
shell32.dll (8460288 bytes) may be duplicated (2 files):
        \windows\system32\shell32.dll
        \windows\system32\dllcache\shell32.dll
```

Az előző sorok az utolsók abból a 3282 sornyi kimenetből, amely egy Windows XP rendszeren futtatott `finddup.py \windows` parancs kimenetéből származik.

Hálózatkezelés és internetes programozás

A hálózatkezeléssel és az Internettel kapcsolatos programozási feladatok ellátásában szerepet játszó csomagok és modulok a Python alapkönyvtárának jelentős részét képezik. A `socket` modulban található függvények a legalacsonyabb szinten valósítják meg az alapvető hálózati műveleteket. Találunk csatolók (socket) létrehozására, DNS-keresések (Domain Name System) végrehajtására, és IP-címek (Internet Protocol) kezelésére alkalmas függvényeket. Az `ssl` modullal titkosított és azonosítást kívánó csatolók alakíthatók ki. A `socketserver` modul TCP- (Transmission Control Protocol) és UDP- (User Datagram Protocol) kiszolgálókkal lát el bennünket. A kiszolgálók alkalmasak a kérelmek közvetlen kezelésére, de az egyes kérelmek kezelésére létre tudnak hozni önálló folyamatot (elágaztatással), sőt akár önálló szálat is. Az aszinkron ügyfél-kiszolgáló csatolókezelés az `asyncore`, illetve a rá épülő, magasabb szintű `asynchat` modul használatával valósítható meg.

A Python a WSGI (Web Server Gateway Interface, webkiszolgálói átjárófelület) felületet azért alakította ki, hogy a webkiszolgálók és a Python nyelven írt alkalmazások közötti kommunikációnak szabványos felületet biztosítson. A szabványt támogatandó, a `wsgiref` csomag tartalmazza a WSGI egy példaként szolgáló megvalósítását, modulokkal a WSGI felülettel együttműködni képes HTTP-kiszolgálók kialakításához, illetve a válaszfejlécek és a CGI-parancsfájlok (Common Gateway Interface) kezeléséhez. Ezen felül a `http.server` modul olyan HTTP-kiszolgálót ad a kezünkbe, amelynek átadható egy CGI-parancsfájlok futtatásához szükséges kérelemkezelő. A `http.cookies` és a `http.cookiejar` modul tartalmazza a sütik kezelését végző függvényeket, a CGI-parancsfájlok támogatását pedig a `cgi` és a `cgitb` modul végzi.

A HTTP-kérelmekhez ügyféloldalról a `http.client` modullal férhetünk hozzá, bár a magasabb szintű `urllib` csomag `urllib.parse`, `urllib.request`, `urllib.response`, `urllib.error` és `urllib.robotparser` moduljai könnyebb és kényelmesebb URL-kezelést tesznek lehetővé. Egy fájl letöltése az Internetről ilyen egyszerű:

```
fh = urllib.request.urlopen("http://www.python.org/index.html")
html = fh.read().decode("utf8")
```

Az `urllib.request.urlopen()` függvény olyan objektumot ad vissza, amelynek a viselkedése leginkább egy bináris módban olvasásra megnyitott fájléra hasonlít. A fenti példában a Python webhelyének `index.html` fájlját `bytes` osztályú objektumként töltjük le, majd a tartalmát a `html` változóban tároljuk, immár karakterláncként. Fájlok letöltése és helyi fájlként való tárolása is lehetséges; ilyenkor a `urllib.request.urlretrieve()` függvényt érdemes használnunk.

A HTML és az XHTML dokumentumok feldolgozását végezhetjük a `html.parser` modullal, az URL-ek feldolgozását és létrehozását az `urllib.parser` modullal, a `robots.txt` állományok feldolgozását pedig az `urllib.robotparser` modullal. A JSON (JavaScript Object Notation) formátum használatával ábrázolt adatok olvasását és írását a `json` modullal oldhatjuk meg.

A HTTP-kiszolgálók és -ügyfelek támogatásán túl az alapkönyvtár az XML-RPC (Remote Procedure Call, távoli eljáráshívás) technológiát is támogatja, mégpedig a `xmlrpc.client` és az `xmlrpc.server` modulokkal. További ügyféloldali támogatás található az FTP (File Transfer Protocol, fájlátviteli protokoll) protokollhoz az `ftplib` modulban, az NNTP (Network News Transport Protocol, hálózati hírcsatornák kezelésére szolgáló protokoll) protokollhoz az `nntplib` modulban és a TELNET protokollhoz a `telnetlib` modulban.

Az `smtpd` modul egy SMTP-kiszolgálót (Simple Mail Transport Protocol, egyszerű levéltovábbítási protokoll) bocsát a rendelkezésünkre, a levelezés ügyféloldali moduljai pedig az `smtplib` az SMTP esetében, az `imaplib` az IMAP4 (Internet Message Access Protocol, internetes üzenethozzáférési protokoll) esetében, és a `poplib` a POP3 (Post Office

Protocol, postahivatal-protokoll) esetében. A különböző formátumú postafiókok elérését a `mailbox` modul biztosítja. Egy-egy üzenet (akár több részes is lehet) létrehozására és kezelésére az `email` modul szolgál.

Ha az alapkönyvtár csomagjai és moduljai elégtelennek bizonyulnak a területen, a Twisted nevű, átfogó, külső fejlesztésű hálózatkezelő könyvtár (`www.twistedmatrix.com`) lehet a megoldás. Sok további külső fejlesztésű könyvtár található az Interneten, ilyen például a webalkalmazások készítésére való Django (`www.djangoproject.com`) és Turbogears (`www.turbogears.org`), valamint a teljes webes keretrendszert és tartalomkezelő rendszert kínáló Plone (`www.plone.org`) és Zope (`www.zope.org`). Az előzőekben felsorolt könyvtárakat mind Python nyelven írták.

XML

Az XML nyelvű dokumentumok feldolgozására két széles körben elterjedt módszer áll a rendelkezésünkre. Az egyik a DOM (Document Object Model, dokumentum-objektum-modell), a másik pedig a SAX (Simple API for XML) használata. Az alapkönyvtárral két DOM-feldolgozót kapunk; az egyik az `xml.dom` modulban, a másik az `xml.dom.minidom` modulban található. A SAX-feldolgozást az `xml.sax` modul biztosítja. A benne lévő `xml.sax.saxutils.escape()` tagfüggvény miatt már használtuk az `xml.sax.saxutils` modult (az „&”, a „>” és a „<” jelek XML-ben szokásos levédésére). Létezik az `xml.sax.saxutils.quoteattr()` függvény is, amely ugyanezt teszi, de ezen felül levédi az idézőjeleket is (alkalmassá téve a szöveget arra, hogy egy címke jellemzőjeként használjuk). Az ellenkező irányú átalakítást a `xml.sax.saxutils.unescape()` függvény végzi.

Két további feldolgozót is használhatunk. Amennyiben az `expat` könyvtár hozzáférhető, az `xml.parsers.expat` modul használható az XML-dokumentumok `expat`-tal való feldolgozására. Az `xml.etree.ElementTree` modul segítségével pedig egyfajta szótár/lista-felületet használva végezhetjük el az XML-dokumentumok feldolgozását. (Alapértelmezés szerint a DOM és az elemfa-feldolgozók maguk is az `expat` feldolgozót használják a háttérben.)

Az XML-dokumentumok saját kezű, a DOM-mal, illetve elemfák használatával való megírása, valamint az XML-dokumentumok DOM-, SAX- vagy elemfa-feldolgozóval történő értelmezése a 7. fejezet témája.

Példa: Az xml.etree.ElementTree modul

A Python DOM- és SAX-feldolgozói szolgáltatják azt az alkalmazásprogramozási felületet, amihez a tapasztalt XML-programozók szoktak hozzá, az `xml.etree.ElementTree` modul pedig egy „pythonosabb” megközelítését kínálja az XML-dokumentumok olvasásának és írásának. Az elemfa-modul csak mostanában került be az alapkönyvtárba*, így néhány olvasónk esetleg nem ismeri. Ennek tükrében az alábbiakban ízelítőt nyújtunk belőle,

* A `xml.etree.ElementTree` modul a 2.5-ös Python-változatban szerepelt először.

mégpedig egy igen rövid példa segítségével. A 7. fejezetben találunk egy alaposabb példát, és az a fejezet mutat majd be egy összehasonlításra is alkalmas kódot, a DOM és a SAX technológiák használatával.

Az Egyesült Államok kormányzatának NOAA (National Oceanic and Atmospheric Administration) webhelye meglehetősen sokféle adatot szolgáltat, többek között egy olyan XML-állományt, amelyben az Egyesült Államok időjárás-megfigyelő állomásainak listája található. A fájl 20 000 sornál is hosszabb, és nagyjából kétezer állomás részletes adatait tartalmazza. Íme egy átlagos állomás:

```
<station>
    <station_id>KBOS</station_id>
    <state>MA</state>
    <station_name>Boston, Logan International Airport</station_name>
    ...
    <xml_url>http://weather.gov/data/current_obs/KBOS.xml</xml_url>
</station>
```

Pár sort kihagytunk, és a fájlban használt behúzást is csökkentettük. A fájl mérete körülbelül 840 kilobájt, így a gzip használatával 72 kilobájtos, használható méretűre tömörítettük. Sajnos az elemfa-feldolgozó fájlnevet vagy fájlobjektumot vár olvasásra, így tömörített fájlt nem adhatunk át neki, az ugyanis a feldolgozó szempontjából nem lenne egyéb, mint véletlenszerű bináris adatok halmaza. A probléma megoldását szolgálja a két kezdeti lépés:

```
binary = gzip.open(filename).read()
fh = io.StringIO(binary.decode("utf8"))
```

io.StringIO 74, 209 ☞

A `gzip` modul `gzip.open()` függvénye hasonló a beépített `open()` függvényhez, azzal a különbséggel, hogy a gzip tömörítésű (`.gz` kiterjesztésű) állományokat nyers bináris adatként képes olvasni. Az adatokra olyan fájl formájában van szükségünk, amellyel az elemfa-feldolgozó dolgozni tud. Így a `bytes.decode()` tagfüggvénnyel a bináris adatot karakterlánccá alakítjuk, ugyanannak az UTF-8 karakterkódolásnak a használatával, amit az XML-fájl is használ, majd létrehozunk egy fájlként viselkedő `io.StringIO` osztályú objektumot, benne az XML-fájlban lévő összes adatot tartalmazó karakterlánccal.

bytes típus ☞ 297

```
tree = xml.etree.ElementTree.ElementTree()
root = tree.parse(fh)
stations = []
for element in tree.getiterator("station_name"):
    stations.append(element.text)
```

A fenti kódrészletben új `xml.etree.ElementTree.ElementTree` osztályú objektumot hozunk létre, majd a feldolgozni kívánt XML-adatokat fájlobjektum formájában átadjuk neki. Az elemfa-feldolgozó szempontjából az átadott adat egy olvasásra megnyitott fájlt tartalmazó objektum, még akkor is, ha a valóságban egy `io.StringIO` osztályú objektumban tá-

rolt karakterláncot kapott. Szeretnénk kinyerni az egyes időjárásjelző állomások neveit, és ennél mi sem egyszerűbb, ha a `xml.etree.ElementTree.ElementTree.getiterator()` tagfüggvényt használjuk, amelynek a visszatérési értéke egy olyan bejárható objektum, amely viszont az átadott címkenévvel bíró `xml.etree.ElementTree.Element` osztályú objektumokat adja vissza. A szöveg kinyeréséhez egyszerűen az elem `text` jellemzőjéhez fordulunk. Csakúgy, mint az `os.walk()` függvény esetében, itt sem magunknak kell elvégeznünk a bejárást: a bejárható objektum megteszi ezt helyettünk. Címkét sem kell megadnunk – a bejárható objektum így ugyanis a teljes XML-dokumentum minden egyes elemét visszaadja.

Egyéb modulok

Nem áll rendelkezésünkre elegendő hely az alapkönyvtár majd' kétszáz csomagjának és moduljának tárgyalására. Mindazonáltal ez az általános áttekintés elvileg elég volt arra, hogy belekóstoljunk az alapkönyvtár nyújtotta szolgáltatásokba, és megismerjük az általa lefedett főbb területek legfontosabb csomagjait. A fejezet utolsó részéhez érve néhány további, érdeklődésünkre számot tartó témakör megismerésével folytatjuk.

Az előzőkben láthattuk, hogy mennyire egyszerűen tudunk dokumentációs karakterláncokat létrehozni, és milyen egyszerű őket a `doctest` modullal futtatni. Az alapkönyvtárnak is van egy egységtesztek lebonyolítására szolgáló keretrendszere: a `unittest` modul, amely a Java nyelv JUnit teszt-keretrendszerének Python nyelvű változata. A `doctest` modul képes alapszinten együttműködni a `unittest` modullal. Számos külső fejlesztésű tesztelési keretrendszer is a rendelkezésünkre áll, például a `codespeak.net/py/dist/` webhelyről letölthető `py.test` és a `www.somethingaboutorange.com/mrl/projects/nose/` webhelyről letölthető `nose`.

A közvetlen felhasználói beavatkozás nélkül futó alkalmazások, például a kiszolgálók, gyakran használnak naplófájlokat a felmerülő problémák jelzésére. A `logging` modul egységes, a naplózás feladatának ellátását segítő felületet nyújt. Azon felül, hogy fájlokba képes naplózni, a GET és a POST HTTP-módszerrel is el tudja végezni a naplózást, sőt elektronikus levelek vagy csatolók használatával is.

Az alapkönyvtárnak vannak olyan moduljai is, amelyek az önelemzést, illetve a kód módosítását szolgálják. A legtöbbjük tárgyalása túlmutat könyvünk keretein, mégis megemlítünk egyet: a `pprint` nevűt, amely a Python-objektumok és -adattípusok kódjának formázására (angol nyelvű szakzsargonban „pretty printing" – innen a modul neve) való, és mint ilyen, gyakran van a segítségünkre a hibakeresés során is. Az `inspect` modul használatát a 8. fejezetben mutatja be egy egyszerű példa; a modul az élő objektumok (live objects) önelemzését segíti.

A `threading` modul a többszálú alkalmazások készítését támogatja, a `queue` pedig három különböző szálbiztos sortípus-megvalósítást kínál. A többszálúsággal a 9. fejezet foglalkozik.

A Python nem támogatja beépítetten a grafikus felhasználói felületek készítését, de Python nyelvű programjainkban számos ilyen feladatot ellátó programkönyvtárat használhatunk. A Tk könyvtárat a rendszerint az alaptelepítés részeként a gépünkre kerülő `tkinter` modul támogatja. A grafikus felhasználói felületek programozásába a 13. fejezet nyújt betekintést.

Az `abc` (Abstract Base Class, elvont alaposztály) modul az elvont alaposztályok létrehozásához szükséges függvények helye. A modullal a 8. fejezetben foglalkozunk.

A `copy` modulban találjuk a 3. fejezetben megismert `copy.copy` és `copy.deepcopy` függvényeket.

Sekély és mély másolás 143 ☜

A *külső függvények*, vagyis a megosztott könyvtárakban (a Windows `.dll`, a Mac OS X `.dylib` és a Linux `.so` állományaiban) lévő függvények a `ctypes` modul használatával válnak hozzáférhetővé.
A Pythonnak része a C nyelvhez való alkalmazásprogramozási felület, vagyis lehetőségünk van arra, hogy a C nyelven létrehozott egyedi adattípusokat és függvényeket a Python nyelvű programjainkban is használjuk. A `ctypes` modul és a C-alkalmazásprogramozási felület tárgyalása egyaránt kívül esik könyvünk keretein.

Ha az előzőekben említett csomagok, illetve modulok egyikében sem találjuk meg, amit keresünk, ne fogjunk hozzá azonnal a megírásához. Előbb érdemes körülnéznünk a Python dokumentáció *Global Module Index*-ében, hátha létezik az a modul, amelyet éppen megírni készülünk – a fejezetben nem volt lehetőség minden modul megemlítésére. Ha itt sem találnánk, amit keresünk, keresgéljünk a *Python Package Index* webhelyen (`pypi.python.org/pypi`), ahol több ezer Python-bővítményt lelhetünk, a rövid, egyetlen fájlból álló moduloktól a hatalmas programkönyvtárakig, illetve a pár sorból álló csomagocskáktól a több száz modulból álló keretrendszerekig.

Összefoglalás

A fejezet elején bemutattuk a csomagok, modulok, illetve a modulok belsejében lévő objektumok betöltésekor használatos utasításformákat. Rámutattunk, hogy a névütközések elkerülése végett sok programozó az `import valami` utasításformát használja. Ügyelnünk kell arra, hogy a programjaink, illetve moduljaink a felső szintű Python-moduloktól, illetve -mappáktól eltérő nevet kapjanak.

A Python csomagjairól szintén esett szó. Egyszerűen olyan mappákról beszélünk, amelyekben egy `__init__.py` nevű fájlt és egy vagy több `.py` kiterjesztésű modulfájlt találunk. Az `__init__.py` nevű fájl üres is lehet, de ha szeretnénk támogatni a `from valami import *` utasításformát, akkor az `__init__.py` nevű fájlban létre kell hoznunk egy `__all__` nevű, különleges szerepű változót, amely értékül a modulok neveiből álló listát kapja.

Az `__init__.py` fájlban emellett elhelyezhetünk általánosan használt előkészítő kódot is. Megemlítettük, hogy a csomagok egymásba ágyazása is járható út: almappák létrehozásával és az egyes almappák saját `__init__.py` fájljainak létrehozásával történik.

Két egyedi modult tárgyaltunk részletesebben. Az első mindössze néhány függvényt tartalmaz, `doctest`-jei pedig igen egyszerűek. A második kidolgozása alaposabb: saját kivételek, futásidő alatti függvénylétrehozás az operációs rendszernek megfelelő egyedi függvény kialakítására, privát globális adatok, egy előkészítő függvény meghívása, valamint alaposabban átgondolt `doctest`-ek jellemzik.

A fejezetnek nagyjából a felét a Python alapkönyvtárának madártávlatú áttekintésére áldoztuk. Több karakterlánc-kezelő modult említettünk, és bemutattunk néhány, az `io.StringIO` osztállyal kapcsolatos példát. Az egyik példa bemutatta, hogy miként lehetséges szöveg fájlba írása, akár a beépített `print()` függvény, akár a fájlobjektumokra jellemző `write()` tagfüggvény használatával. Megtudtuk azt is, hogy miként használható igazi fájl helyett egy `io.StringIO` osztályú objektum. Az előző fejezetekben a `sys.argv` objektum feldolgozását saját magunk végeztük, de amikor az alapkönyvtár parancssori programozást segítő részéről volt szó, megismertük a parancssori paraméterek kezelését nagyban leegyszerűsítő `optparse` modult – amelyet mostantól kiterjedten használunk.

Szóba került az is, hogy a Python nyelv remekül alkalmas számokkal való munkára. Átfutottuk az alapkönyvtár számok tárolására való adattípusait és a matematikai függvényeknek otthont adó három modult csakúgy, mint a `SciPy` projekt által kínált, tudományos és mérnöki számítások elvégzését segítő lehetőségeket. Futólag említettük az alapkönyvtár és a külső fejlesztőktől származó, dátum- és időkezeléssel foglalkozó osztályokat, láttunk példát az aktuális dátum és idő kinyerésére, illetve a Unix-időszámítás kezdete óta eltelt másodpercek, valamint a `datetime.datetime` tagfüggvény kimenete közötti átváltásra. Megtárgyaltuk az alapkönyvtárban lévő gyűjteményes adattípusokat és a rendezett sorozatokkal való munkát támogató algoritmusokat, illetve áttekintettünk néhány, a `heapq` modul függvényeit bemutató példát.

Előkerültek azok a modulok, amelyek – a karakterkódolásokon felül – a különféle fájlkódolásokat támogatják, valamint azok is, amelyek a legnépszerűbb archívumformátumok kezelését biztosítják, sőt a hangadatokat kezelő modulokról is esett szó. Láttunk példát arra, hogy miként tárolható Python nyelvű állományban Base64 kódolással átalakított bináris adat, és megvizsgáltunk egy `.tar` kiterjesztésű fájlok kicsomagolására szolgáló programot is. A fájlok és mappák kezelésének támogatottsága jelentős – ráadásul mindez operációs rendszertől független függvényekkel valósul meg. Mutattunk példát olyan szótár létrehozására, amelynek a kulcsai fájlnevek, az értékei pedig az utolsó módosítást jelző időbélyegek, és értelmeztük egy könyvtárak rekurzív (alkönyvtárakat is bejáró) átvizsgálását végző, fájlnév és -méret alapján többször előforduló fájlokat kereső program működését.

Az alapkönyvtár nagy részét a hálózatkezelésnek és az Internettel kapcsolatos programok írásának szentelték. Éppen csak érintettük a rendelkezésünkre álló lehetőségeket, amelyek között a nyers csatolók (beleértve a kódolt csatolókat) éppúgy megtalálhatók, mint a TCP- és UDP-kiszolgálók vagy a HTTP-kiszolgálók és a WSGI támogatása. Megemlítettük a sütik, a CGI-parancsfájlok, a HTTP-adatok kezelésére, a HTML-, az XHTML- és az URL-értelmezésre szolgáló modulokat is. A további megemlített modulok között szerepeltek az XML-RPC kezelését, illetve a magasabb szintű protokollok – az FTP, az NNTP, az SMTP protokoll ügyfél- és kiszolgálóoldala, illetve az IMAP4 és a POP3 ügyféloldala – támogatását végző modulok.

Szintén megemlítettük az alapkönyvtár kiterjedt XML-író és -értelmező támogatását, beleértve a DOM-, a SAX- és az elemfa-értelmezőket, illetve az `expat` modult, és példát is láttunk az elemfa-értelmező modul működésére. Emellett megemlítettünk néhány további, az alapkönyvtárban található csomagot és modult.

A Python alapkönyvtára hihetetlenül hasznos forrást jelent, elképesztő mennyiségű idő és erőfeszítés spórolható meg a használatával, sok esetben pedig lényegesen rövidebb programok írhatók az alapkönyvtár nyújtotta lehetőségek kihasználásával. Azoknak a területeknek a lefedésére, amelyeken az alapkönyvtár esetleg nem kielégítő, szó szerint sok ezer külső fejlesztésű csomag létezik. Ennek a rengeteg dolognak a segítségével úgy összpontosíthatunk a programjaink működésére, hogy a részletekkel való törődést a programkönyvtárakban lévő modulokra hagyhatjuk.

A fejezettel befejeztük az eljárásközpontú programozás alapjainak áttekintését. A következő fejezetek, különösen a 8., az eljárásközpontú programozás mélyebb, kevésbé ismert területeire kalauzolnak, a soron következő fejezetben pedig bemutatjuk az objektumközpontú programozást. A Python nyelv tisztán eljárásközpontú nyelvként való használata lehetséges, és – főként kis programok esetében – tanácsos is, de a közepes és nagy programoknál, az egyedi csomagoknál és moduloknál, és azokban az esetekben, ahol a hosszú időn át karbantartható kód fontos szempont, az objektumközpontú megközelítés a befutó. Szerencsére minden, amit eddig tárgyaltunk, egyaránt hasznos és helytálló az objektumközpontú programozás világában is, így a következő fejezetek az eddig megalapozott Python-tudásunkra építve fejleszthetik tovább programozói képességeinket.

Gyakorlat

Írjunk olyan programot, amely könyvtárak tartalmát írja ki, vagyis a Windows `dir`, illetve a Unix `ls` parancsához hasonló feladatot végez. A saját könyvtártartalom-kiíró program előnye, hogy megadhatjuk a saját alapértelmezéseinket, és a programot minden operációs rendszeren használhatjuk, tehát nem kell emlékeznünk a `dir` és az `ls` közötti különbségekre. Készítsük el az alábbiaknak megfelelő felhasználói felülettel rendelkező programot:

```
Usage: ls.py [options] [path1 [path2 [... pathN]]]

The paths are optional; if not given . is used.

Options:
-h, --help          show this help message and exit
-H, --hidden        show hidden files [default: off]
-m, --modified      show last modified date/time [default: off]
-o ORDER, --order=ORDER
                    order by ('name', 'n', 'modified', 'm', 'size', 's')
                    [default: name]
-r, --recursive recurse into subdirectories [default: off]
-s, --sizes         show sizes [default: off]
```

(A kimenetet a könyv oldalszélességéhez igazítottuk.)

Íme egy kis könyvtár tartalma, amelyet az `ls.py -ms -os misc/` paranccsal írattunk ki:

```
2007-04-10 15:49:01          322 misc/chars.pyw
2007-08-01 11:24:57        1,039 misc/pfa-bug.pyw
2007-10-12 09:00:27        2,445 misc/test.lout
2007-04-10 15:50:31        2,848 misc/chars.png
2008-02-11 14:17:03       12,184 misc/abstract.pdf
2008-02-05 14:22:38      109,788 misc/klmqtintro.lyx
2007-12-13 12:01:14    1,359,950 misc/tracking.pdf
                                 misc/phonelog/

7 files, 1 directory
```

A parancssorban a paramétereket csoportokba foglaltuk (az `optparse` modul önműködően kezeli mindezt), és ugyanezt az eredményt értük volna el a paraméterek szétválasztásával (`ls.py -m -s -os misc/`), egyetlen csoportba foglalásával (`ls.py -msos misc/`), a hosszú paraméterformák használatával (`ls.py –modified –sizes –order=size misc/`), illetve ezek bármely kombinációjával. Megjegyeznénk, hogy „rejtettnek” azokat a mappákat és fájlokat tekintjük, amelyeknek ponttal (.) kezdődik a nevük.

A gyakorlat sok kihívást rejt magában. Át kell olvasnunk az `optparse` modul dokumentációját, és meg kell értenünk, hogy milyen paraméterekkel állítható be `True` érték, és hogy miként ajánlható fel lehetőségek egy adott listája. Ha a felhasználó rekurzív bejárást kér, a fájlok feldolgozását (a könyvtárakét nem) az `os.walk()` függvénnyel kell elvégeznünk, egyébként az `os.listdir()` függvény használatával magunknak kell gondoskodnunk mind a fájlok, mind a könyvtárak feldolgozásáról.

A rejtett könyvtárak elkerülése egy kis ügyességet igényel a rekurzív feldolgozás során. A lista módosításával eltávolíthatók az `os.walk()` tagfüggvény `dirs` listájából – így az `os.walk()` majd kihagyja őket. De vigyázzunk: ne adjuk értéket közvetlenül a `dirs` változónak – hiszen ezzel nem a hivatkozott listát fogjuk megváltoztatni, csak egyszerűen

(és értelmetlenül) lecseréljük egy másikra. Érdemes a receptkönyvi megoldást használnunk: adjunk a teljes lista egy részlistájának értéket, vagyis gondoljunk a következő alakú utasításra: `dirs[:] = [dir for dir in dirs if not dir.startswith("."):]`.

`locale.setlocale()` 81 ☜

A fájlméretek kiírásánál a megfelelő számcsoportképző karakter kinyerésének legjobb módja, ha betöltjük a `locale` modult; a `locale.setlocale()` függvény hívásával megkapjuk a felhasználó alapértelmezett beállításait, majd az `n` formázókaraktert használjuk. Az `ls.py` nagyjából 130 sor hosszú, a 130 sor négy függvény között oszlik meg.

6

Objektumközpontú programozás

- Az objektumközpontú megközelítés
- Egyedi osztályok
- Egyedi gyűjteményosztályok

A korábbi fejezetekben rengeteg objektumot használtunk, a programozási stílusunk azonban szigorúan eljárásközpontú volt. A Python azonban többféle megközelítést alkalmazó nyelv, vagyis egyaránt lehetővé teszi az eljárásközpontú, az objektumközpontú és a függvényközpontú programozási stílus alkalmazását, sőt ezek keverékét is – hiszen semmiféle módon nem korlátozza a programozási lehetőségeinket.

Tetszőleges programot megírhatunk eljárásközpontú alakban, és ez a kisebb programoknál (mondjuk úgy 500 sorig) egyáltalán nem jár semmiféle nehézséggel. A legtöbb program esetében azonban – különösképpen a közepes, illetve nagy méretűeknél – az objektumközpontú programozás számos előnnyel jár.

Ebben a fejezetben bemutatjuk mindazokat az alapelveket és módszereket, amelyek lehetővé teszik az objektumközpontú programozást a Pythonban. Az első rész leginkább azoknak szól, akik kevesebb tapasztalattal rendelkeznek ezen a téren, illetve eljárásközpontú nyelveket (mint a C vagy a Fortran) használtak a korábbiakban. Először olyan kérdésekre világítunk rá, amelyek gondot okozhatnak az eljárásközpontú programozás világában, de az objektumközpontú módszerek választ adnak rájuk. Ezt követően röviden bemutatjuk az objektumközpontú programozás alapjait a Pythonban, valamint megismerkedünk néhány alapvető kifejezéssel. Ha mindezeken túlestünk, rátérünk a fejezet két fő részére.

A második részben megtudjuk, hogyan hozhatunk létre egyedi adattípusokat, amelyek egy-egy elemet tárolnak (jóllehet ezek az elemek több jellemzővel is rendelkezhetnek), majd megismerkedünk az egyedi gyűjteményes adattípusokkal, amelyek tetszőleges számú és típusú objektumot tartalmazhatnak. Ezeken az oldalakon szinte mindent megtalálunk, amit az objektumközpontú programozásról a Pythonban tudnunk kell, bár néhány komolyabb felkészültséget igénylő témakört csak a 8. fejezetben fejtünk ki bővebben.

Az objektumközpontú megközelítés

A következőkben egy példa kapcsán világítunk rá a tisztán eljárásközpontú megközelítés gyenge pontjaira. A példában köröket kell ábrázolnunk, amelyekből esetleg igen sok is lehet. Egy kör meghatározásához legalább három adatra van szükség – a középpontjának (x, y) a koordinátája és a sugara (radius) éppen megteszi. Az adatok tárolására alkalmazhatunk rendezett hármasokat, például így:

```
circle = (11, 60, 8)
```

Ennek a módszernek az az egyik hátulütője, hogy nem tudhatjuk pontosan, hogy a hármas melyik tagja mit jelent. Gondolhattunk az `(x, y, radius)` hármasra, de éppúgy tárolhattuk az adatokat `(radius, x, y)` alakban is. További gondokat okozhat, hogy az elemeket csak a sorszámuk alapján érhetjük el. Ha ugyanis van két függvényünk – a `distance_from_origin(x, y)` és az `edge_distance_from_origin(x, y, radius)` –, amelyek körök adatait várják paraméterként, a sorozatkicsomagoló műveletet kell használnunk az elérésükhöz, valahogy így:

```
distance = distance_from_origin(*circle[:2])
distance = edge_distance_from_origin(*circle)
```

Itt mindkét esetben feltételeztük, hogy a rendezett hármasok `(x, y, radius)` alakban tartalmazzák a körök adatait. A sorrend ismeretének igényén és a kicsomagolás bonyodalmain túllendülhetünk, ha nevesített rendezett sorozatot használunk:

```
import collections
Circle = collections.namedtuple("Circle", "x y radius")
circle = Circle(13, 84, 9)
distance = distance_from_origin(circle.x, circle.y)
```

Így most már létrehozhatunk `Circle` rendezett hármasokat, amelyek jellemzőire a nevükkel hivatkozhatunk – így a függvényhívások érthetőbbé válnak. Sajnos a gondjainknak ezzel még nem szakad vége. Semmi sem tiltja meg például, hogy érvénytelen adatokkal hozzunk létre köröket, mint alább:

```
circle = Circle(33, 56, -5)
```

Természetesen semmi értelme egy negatív sugarú körnek, de a `circle` hármas mindenféle gond (kivétel kiváltása) nélkül létrejön – éppúgy, mintha a sugarat egy negatív értéket tartalmazó változóban adtuk volna át. A hiba csak abban az esetben válik nyilvánvalóvá, ha meghívjuk az `edge_distance_from_origin()` függvényt, sőt ez esetben is csak akkor, ha a kódban kiszűrjük a negatív sugárértékeket. Az objektumok létrehozásánál így nem tudjuk ellenőrizni az érvényességüket – talán ez a tisztán eljárásközpontú megközelítés igazi Achilles-sarka.

Ha azt szeretnénk, hogy a körök módosulhassanak, vagyis elmozdíthassuk őket a középpontjuk áthelyezésével, illetve átméretezhessük a sugaruk módosításával, a `collections.namedtuple._replace()` tagfüggvényt alkalmazhatjuk:

```
circle = circle._replace(radius=12)
```

A `Circle` objektumok létrehozásánál látottakhoz hasonlóan itt sem tiltja semmi az érvénytelen adatok használatát (és figyelmeztetést sem kapunk).

Ha a körök adatai előreláthatóan gyakran módosulnak, a kényelmesebb munka kedvéért választhatunk valamilyen változtatható adattípust, például listát az adatok tárolásához:

```
circle = [36, 77, 8]
```

Ez továbbra sem véd meg az érvénytelen adatok használatától, ha pedig név szerint kívánjuk elérni az elemeket, a legtöbb, amit tehetünk, hogy meghatározunk néhány alkalmas állandót, mint a `circle[RADIUS] = 5` kifejezésben. A listák használatánál azonban újabb gondok merülnek fel – nyugodtan használhatjuk például a `circle.sort()` hívást! Szótárakat is alkalmazhatunk (például `circle = dict(x=36, y=77, radius=8)`), de itt sem kínálkozik semmilyen mód a sugár érvényes értékének biztosítására, és a nem megfelelő tagfüggvények hívásának megakadályozására.

Az objektumközpontú programozás alapelvei és alapfogalmai

Az előzőekből leszűrhető, hogy valójában a kört leíró adatok egységbe zárására van szükségünk, emellett biztosítanunk kell, hogy csak olyan tagfüggvényeket alkalmazhassunk rájuk, amelyek érvényes műveleteket végeznek rajtuk. Mindkét célunk elérhető egy egyedi `Circle` adattípussal. Létrehozásának módját bemutatjuk a későbbiekben, előbb azonban tisztáznunk kell néhány alapfogalmat. Ha a következőkben leírtak első pillantásra nehezen érthetőnek is tűnnek, ne aggódjunk, a példák majd mindent megvilágítanak.

Az *osztály*, a *típus* és az *adattípus* fogalmakat szinonimáknak tekintjük. A Pythonban létrehozott egyedi osztályok teljes mértékben beépülnek a nyelv szerkezetébe, vagyis ugyanúgy használhatjuk őket, mint a beépített adattípusokat. Osztályokkal már többször is találkozhattunk – ilyen volt például a `dict`, az `int` és az `str` is. Egy adott osztály példányára az *objektum*, illetve esetenként a *példány* elnevezéssel hivatkozunk. Így az `5` egy `int`, az `"oblong"` pedig egy `str` objektum.

A legtöbb osztály egyaránt tartalmaz adatokat, valamint tagfüggvényeket, amelyek ezekre az adatokra alkalmazhatók. Így például az `str` osztály egy Unicode-karakterláncot képes adatként tárolni, amelyre tagfüggvényeket – mint mondjuk az `str.upper` – alkalmazhatunk. Számos osztály további lehetőségeket is elérhetővé tesz – így például a `+` művelettel egymáshoz fűzhetünk két karakterláncot (vagy bármely más sorozatot), a beépített `len()` függvénnyel pedig megkaphatjuk egy sorozat hosszát. Ezeket a lehetőségeket *különleges tagfüggvények* biztosítják, amelyek annyiban térnek el a „rendes” tagfüggvényektől, hogy előre meghatározottak, emellett a nevük minden esetben két aláhúzással kezdődik és

végződik. Így ha olyan osztályt szeretnénk létrehozni, amely lehetővé teszi az összefűzést a + művelettel és a hossz megállapítását a `len()` függvénnyel, meg kell valósítanunk benne az `__add__()` és a `__len__()` különleges tagfüggvényeket. Mindebből ugyanakkor az is következik, hogy magunk nem írhatunk olyan tagfüggvényt, amelynek a neve két aláhúzással kezdődik és végződik – hacsak nem valamilyen előre meghatározott tagfüggvényről van szó, amely az adott osztályban alkalmazható. Ez elejét veszi annak, hogy a tagfüggvényeink nevei ütközzenek a Python későbbi változataiban esetleg megjelenő új különleges tagfüggvényekével.

Az objektumok többnyire rendelkeznek jellemzőkkel (attribute) – a tagfüggvények meghívható jellemzők, a többiek pedig az adatokat tárolják. Egy `complex` objektum például rendelkezik a `real` és az `imag` jellemzővel, valamint rengeteg tagfüggvénnyel – találhatunk köztük különleges tagfüggvényeket, mint az `__add__()` vagy a `__sub__()` (ezek teszik lehetővé a + és a - műveletek használatát), továbbá egyszerű tagfüggvényeket, mint a `conjugate()`. Az adatjellemzőket (amelyekre gyakran egyszerűen jellemzőként hivatkozunk) többnyire *példányváltozókkal* valósítjuk meg, amelyeknek a tartalma objektumonként eltérő lehet. A használatukra számos példát láthatunk majd, mint ahogy megtanuljuk azt is, hogy miként tegyük elérhetővé az adatjellemzőket tulajdonságként (property), amelynek az értékét a példányváltozókhoz hasonlóan olvashatjuk ki és állíthatjuk be, de az elérést a háttérben tagfüggvények biztosítják. Amint azt a későbbiekben láthatjuk majd, a tulajdonságok nagy szolgálatot tesznek az adatok érvényességének biztosításában.

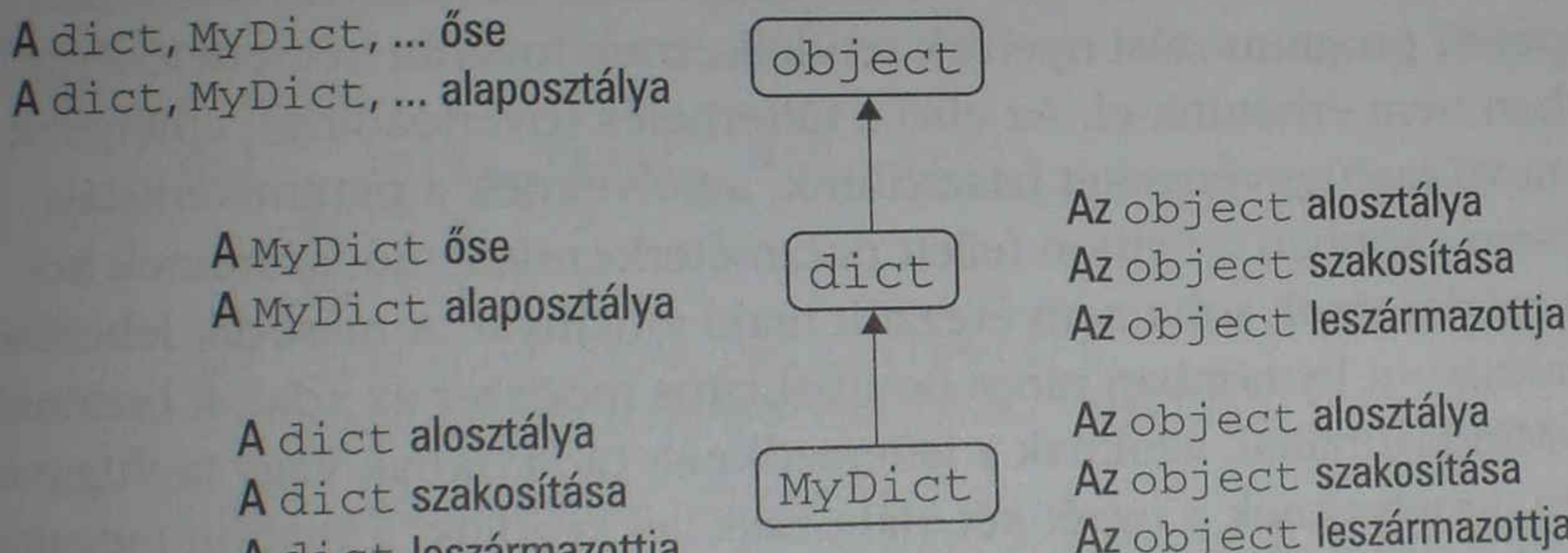
A tagfüggvényen (ami voltaképpen egy egyszerű függvény, azzal a kitétellel, hogy első paraméterként azt az objektumot várja, amelyen műveletet kell végeznie) belül számos, különböző típusú változót érhetünk el. Az objektum példányváltozóihoz úgy juthatunk hozzá, ha a nevüket magával a példánnyal minősítjük. A helyi változókat a tagfüggvényen belül hozhatjuk létre – ezeknél nincs szükség minősítésre. Az osztályváltozókat (amelyeket olykor statikus változóknak is neveznek) az osztálynévvel minősítve érhetjük el, míg a globális, vagyis modulszintű változókhoz minősítés nélkül juthatunk hozzá.

A Pythonnal foglalkozó irodalomban ismert a *névtér* fogalma is, amely a neveket objektumoknak feleltheti meg. A modulok voltaképpen névterek – így az `import math` utasítást követően elérhetjük a `math` modul objektumait, ha a névterük nevével minősítjük őket (például: `math.pi` vagy `math.sin()`). Az osztályok és az objektumok szintén névterek: így a `z = complex(1, 2)` utasítással meghatározott objektum névterében két jellemzőt érhetünk el (ezek a `z.real` és a `z.imag`).

Az objektumközpontú megközelítés egyik legvonzóbb lehetősége, hogy ha rendelkezünk egy osztállyal (class), létrehozhatjuk egy *alosztályát* (subclass) – ezt *szakosításnak* vagy *specializációnak* is hívják –, vagyis egy új osztályt, amely örökli az eredeti osztály összes jellemzőjét (adatait és tagfüggvényeit), az eredményben pedig új tagfüggvényeket helyezhetünk el, egyesekkel esetleg régebbieket felváltva, illetve újabb példányváltozókat vehe-

tünk használatba. Ez a lehetőség rendelkezésre áll minden Python-osztályban – legyen az beépített, az alapkönyvtárból származó vagy egyedi osztály*. Az alosztályok használatának lehetősége az objektumközpontú megközelítés egyik legnagyobb vívmánya, hiszen így egy jól bevált régi osztályra építve új adatjellemzők és tagfüggvények hozzáadásával egyszerűen és gyorsan létrehozhatjuk a saját változatunkat. Ráadásul az új osztályunk objektumait nyugodt lélekkel átadhatjuk olyan függvényeknek és tagfüggvényeknek, amelyek az eredeti osztály példányait igénylik.

Azt az osztályt, amelyből az új osztályunkat – közvetlenül vagy közvetve – származtatjuk, az új osztály *ősének* vagy *alaposztályának* nevezzük, maga az új osztály pedig a *származtatott osztály* vagy más néven *alosztály*. A Pythonban minden beépített, illetve könyvtári osztály, továbbá minden egyedi osztály közvetve vagy közvetlenül az `object` nevű ősosztálytól származik. Az öröklés fogalmait áttekinthetjük a 6.1. ábrán.



6.1. ábra
Az objektumok öröklődésének fogalomrendszere

Bármely tagfüggvény felülírható egy adott alosztályban, vagyis elkészíthetjük egy új megvalósítását – ez hasonló a Javában látottakhoz (eltekintve a „final" tagfüggvényektől).** Ha rendelkezünk egy a `MyDict` osztályba tartozó objektummal (amely a `dict` leszármazottja), majd meghívunk egy olyan tagfüggvényt, amelyet meghatároztak mind a `dict`, mind a `MyDict` osztályban, a Python okosan a `MyDict`-változat használata mellett dönt – ez a *dinamikus tagfüggvénykötés* vagy más néven *többalakúság*. Amennyiben az ős tagfüggvényére van szükségünk az átírt tagfüggvény belsejében, a beépített `super()` függvény segítségével ezt is elérhetjük.

A Python a típusokat *fekete dobozként* kezeli – vagyis ha bizonyos tagfüggvényeket meg szeretnénk hívni egy objektumra, a rendszert nem érdekli, hogy milyen osztályba tartozik, csak azt vizsgálja, hogy rendelkezik-e a kívánt tagfüggvényekkel. Az előző fejezetben láthattuk, hogy amikor egy fájlobjektumra volt szükségünk, hozzájuthattunk egyhez

---

* Egyes C nyelven megvalósított osztályokból nem származtathatunk alosztályt – erre a korlátozásra nézve az adott osztály leírása ad tájékoztatást.

** A C++ szóhasználata szerint a Python összes tagfüggvénye virtuális.

a beépített `open()` függvénnyel – de létrehozhattunk helyette egy `io.StringIO` objektumot is, hiszen ezek ugyanazzal az API-val (alkalmazásprogramozási felülettel), vagyis ugyanolyan tagfüggvényekkel rendelkeznek, mint azok a fájlobjektumok, amelyeket az `open()` szöveges módban visszaad.

Az öröklléssel „megfelel" („is-a") típusú kapcsolatokat ábrázolhatunk, ahol egy adott osztály objektumai lényegében megfelelnek egy másik osztály példányainak, mindössze néhány további adattal és tagfüggvénnyel rendelkeznek. Az objektumok között „birtokol" típusú kapcsolatok is megjelenhetnek – erre szolgál az *összetétel* (aggregation vagy composition), amelyben az egyik osztály példányváltozóként tartalmazza más osztályok objektumait. A Pythonban minden osztály részt vesz az öröklésben, lévén az `object` leszármazottja, és a legtöbbjük az összetételt is alkalmazza, hiszen az osztályok általában különféle típusú példányváltozókkal rendelkeznek.

Egyes objektumközpontú programozási nyelvek rendelkeznek további két lehetőséggel, amelyeket a Pythonban nem érhetünk el. Az első a túlterhelés (overloading), amelynek során olyan azonos nevű tagfüggvényeket használunk, amelyeknek a paraméterlistája különbözik. Szerencsére azonban a Python fejlett paraméterkezelési módszereinek köszönhetően ennek a módszernek soha nem érezzük majd a hiányát. A második lehetőség a *hozzáférés-szabályozás* – a Pythonban nincs bombabiztos módszer az adatok biztonságának megőrzésére. Mindazonáltal, azoknak a jellemzőknek (adatoknak vagy tagfüggvényeknek) az esetében, amelyeknek a nevét két aláhúzásjellel kezdjük, a Python megakadályozza az akaratlan hozzáférést, így voltaképpen privátnak tekinthetők. (Mindezt a névkeverés segítségével éri el, amelyre majd a 8. fejezetben láthatunk példát.)

Az egyedi osztályok elnevezésénél az egyedi modulokhoz hasonlóan járunk el: nagybetűvel kezdjük a nevüket. Annyi osztályt hozhatunk létre, amennyit csak akarunk, akár közvetlenül a programban, akár modulokban – az osztályok nevei nem kell, hogy egyezzenek a modulokéival, emellett egy modulban tetszőleges számú osztálymeghatározást elhelyezhetünk.

Láthattunk néhány problémát, amelyekre az osztályok megoldást adhatnak, megismerkedtünk az ebben a témakörben használatos elnevezésekkel, sőt némi háttértudásra is szert tettünk – nincs más hátra, mint hogy létrehozzunk néhány egyedi osztályt!

Egyedi osztályok

Egyedi osztályokat már a korábbi fejezetekben is hoztunk létre: ezek voltak az egyedi kivételek. Lássuk most az osztályok létrehozásának két új alakját:

```
class osztályNév:
    blokk

class osztályNév(alaposztályok):
    blokk
```

Mivel a kivételek alosztályainak létrehozásakor nem adtunk hozzájuk új jellemzőket (sem adatokat, sem tagfüggvényeket), a *blokk* helyére a `pass` kulcsszót írtuk (ami azt jelenti, hogy nincs semmi újdonság), és mivel a blokk egyetlen utasításból állt, a `class` utasítással egy sorba írhattuk. Érdemes megjegyezni, hogy a `def`-hez hasonlóan a `class` is utasítás, így osztályokat szükség esetén dinamikusan is létrehozhatunk. Az osztály tagfüggvényeit az osztály blokkján belül hozzuk létre `def` utasítások segítségével. Ahhoz pedig, hogy példányokat hozzunk létre az osztályból, magát az osztályt kell meghívnunk a megfelelő paraméterekkel. Így az `x = complex(4, 8)` egy komplex számot hoz létre, és így `x` egy erre a számra hivatkozó objektum lesz.

Jellemzők és tagfüggvények

Kezdjük egy igen egyszerű osztállyal – ez a `Point`, amely az `(x, y)` koordinátapárt tárolja. Az osztály a `Shape.py` fájlban található, teljes megvalósítása pedig (a dokumentációs karakterlánc kivételével) alább látható:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

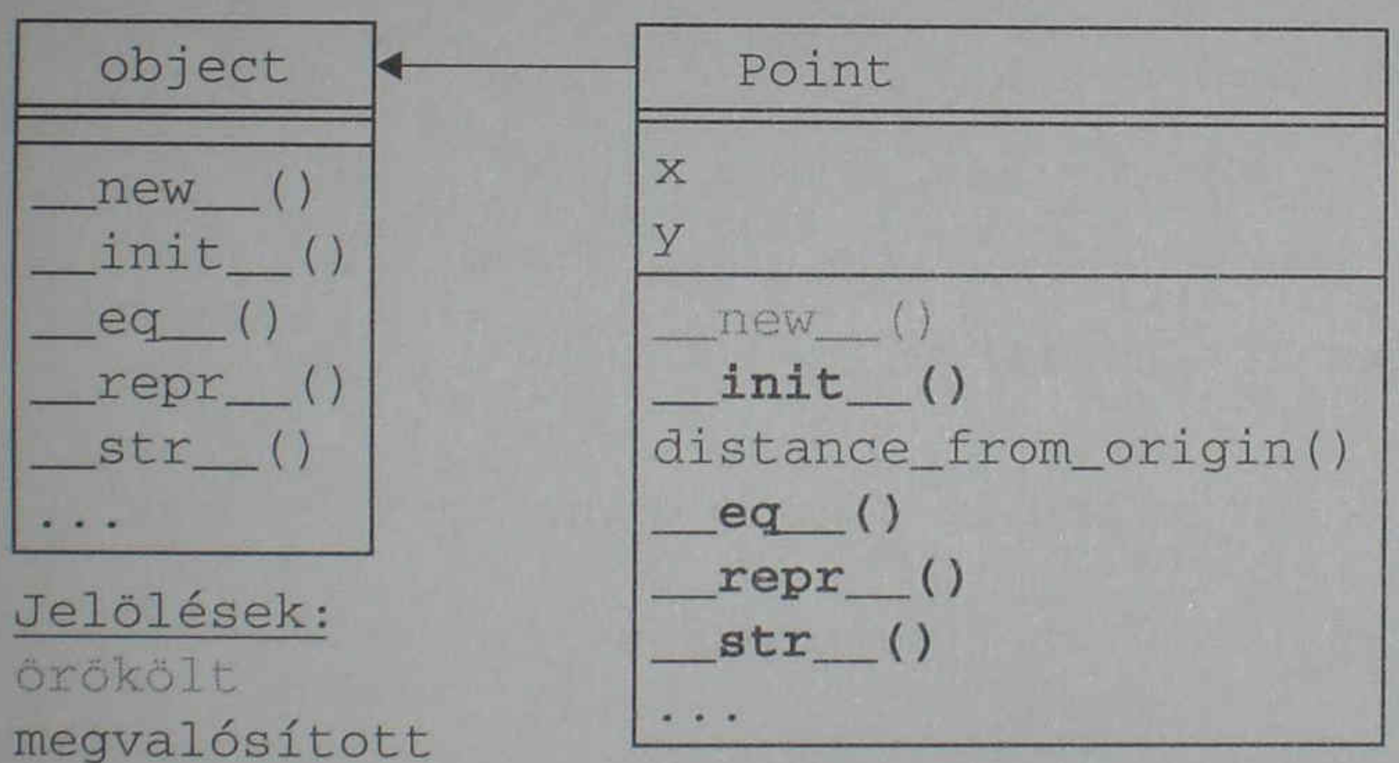
Mivel nem adtunk meg alaposztályt, a `Point` az `object` közvetlen leszármazottja, vagyis meghatározhattuk volna `class Point(object)` alakban is. Mielőtt rátérnénk a tagfüggvényei bemutatására, lássunk néhány példát a használatára:

```
import Shape
a = Shape.Point()
repr(a)                          # az eredmény: 'Point(0, 0)'
b = Shape.Point(3, 4)
str(b)                           # az eredmény: '(3, 4)'
b.distance_from_origin()         # az eredmény: 5.0
b.x = -19
str(b)                           # az eredmény: '(-19, 4)'
a == b, a != b                   # az eredmény: (False, True)
```

A `Point` osztály két adatjellemzővel – `self.x` és `self.y` –, valamint öt tagfüggvénnyel rendelkezik (nem számítva az örökölteket), amelyek közül négy különleges tagfüggvény (lásd a 6.2. ábrát). Miután beemeltük a `Shape` modult, a `Point` osztály más társaihoz hasonlóan használatba vehető. Az adatjellemzőket közvetlenül elérhetjük (vagyis `y = a.y` alakban), és

az osztály nagyszerűen igazodik a nyelv más osztályaihoz, hiszen támogatja az egyenlőségvizsgálatot (==), továbbá ábrázoló és egyszerű karakterláncot is biztosít az objektumok leírására. A nem egyenlő (!=) műveletet már maga a Python adja, az egyenlőség műveletre építve. (Lehetőségünk van arra is, hogy mindkét műveletet magunk határozzuk meg, amennyiben nem szeretnénk, ha pontosan egymás ellentettjei lennének.)

A Python automatikusan megadja a tagfüggvények hívásának első paraméterét – ez egy objektumhivatkozás magára az objektumra (a C++-ban és a Javában ez a *this*). Mindazonáltal a tagfüggvények meghatározásakor ezt a paramétert is fel kell tüntetnünk a paraméterlistában – a neve közmegegyezés szerint `self`. Az objektum minden jellemzőjét (legyen szó adatról vagy tagfüggvényről) a `self`-fel kell minősítenünk. Ez némiképp több gépelést jelent, mint egyéb nyelvekben, de annyi előnye mindenképpen van, hogy így tökéletesen tisztán láthatunk: amennyiben egy jellemzőt a `self` kulcsszóval minősítünk, mindig tudhatjuk, hogy az aktuális objektum jellemzőjéről van szó.



6.2. ábra

A Point osztály öröklési hierarchiája

Ahhoz, hogy objektumot hozzunk létre, két lépésre van szükség. Először létre kell hoznunk egy „nyers”, vagyis kezdőértékekkel nem rendelkező objektumot, majd biztosítanunk kell a számára a kezdőértékeket. Egyes programozási nyelvek (mint a C++ és a Java) egyesítik ezeket a lépéseket, a Python azonban elkülöníti őket. Amikor létrehozunk egy objektumot (például a `p = Shape.Point()` értékadással), a rendszer először a `__new__()` nevű különleges tagfüggvényt hívja meg az objektum létrehozásához, majd az `__init__()` segítségével ellátja kezdőértékekkel.

Szinte akármilyen Python-osztályt hozunk is létre, kizárólag az `__init__()` tagfüggvényt kell újra megvalósítanunk, ugyanis az `object.__new__()` az esetek túlnyomó többségében tökéletesen megteszi, és a rendszer automatikusan ezt hívja meg, amennyiben nem adunk meg mást helyette.

Alternatíva: FuzzyBool ☞ 247

(A fejezet egy későbbi részében bemutatunk egy igen ritka esetet, amikor valóban szükség van arra, hogy újra megvalósítsuk a `__new__()` tagfüggvényt.) Az objektumközpontú programozás egyik fontos vívmánya ez: ha az alaposztály tagfüggvénye alkalmas egy feladat ellátására, a leszármazottban nem kell újra megvalósítanunk. Mindezt az teszi lehetővé, hogy ha meghívunk egy objektumra egy tagfüggvényt, amelynek a megvalósítását a rendszer nem találja az objektum osztályában, automatikusan sorra veszi ennek őseit, majd az ősök őseit, és így tovább, amíg rá nem akad a kívánt tagfüggvényre – ha pedig végképp nem jár sikerrel, `AttributeError` kivételt vált ki.

Ha például elkezdjük végrehajtani a `p = Shape.Point()` utasítást, a Python először a `Point.__new__()` tagfüggvényt keresi meg. Mivel ezt nem valósítottuk meg újra, a `Point` alaposztályaiban folytatja a keresést. Esetünkben a `Point` egyetlen őse az `object`, így a rendszer meghívja az `object.__new__()` tagfüggvényt, amely egy nyers, kezdőértékek nélküli objektumot hoz létre. A Python ez után az `__init__()` tagfüggvény keresésébe kezd, amelyet magunk valósítottunk meg, így a `Point.__init__()` hívással célt ér. Végezetül már csak arra van szükség, hogy a `p` változóban az újonnan létrehozott és kezdőértékekkel ellátott `Point` objektum hivatkozását tárolja.

Rövidségük és a felejtéshez éppen elegendő pár oldalnyi távolság miatt az egyes tagfüggvényeket újfent bemutatjuk majd a részletes tárgyalásuk előtt.

```
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
```

A `self.x` és a `self.y` példányváltozókat az `__init__()` tagfüggvényben hozzuk létre, az `x` és az `y` paramétereket rendelve hozzájuk. Mivel a Python ezt a tagfüggvényt találja meg az új `Point` objektumok létrehozásakor, az `object.__init__()` hívására nem kerül sor – ha a rendszer megtalálta, amit keresett, nem folytatja tovább a kutakodást.

Az objektumközpontú programozás igazi megszállottai a tagfüggvény kezdetén minden bizonnyal meghívnák az alaposztály `__init__()` tagfüggvényét a `super().__init__()` utasítással. Azoknak az osztályoknak az esetében, amelyek az `object` közvetlen leszármazottai, erre semmi szükség, és könyvünkben is csak akkor hivatkozunk az alaposztály tagfüggvényeire, ha erre feltétlenül szükség van – például olyan osztályok létrehozásánál, amelyeket kifejezetten alosztályok szülőinek szánunk, illetve olyanoknál, amelyek nem közvetlen leszármazottai az `object` osztálynak. Bizonyos fokig persze mindez csak programozási stílus kérdése – tökéletesen megérthető az is, ha valaki az egyedi osztályai `__init__()` tagfüggvényeinek elején mindig elhelyezi a `super().__init__()` hívást.

```
def distance_from_origin(self):
    return math.hypot(self.x, self.y)
```

Ez egy hagyományos tagfüggvény, amely némi számítást végez az objektum példányváltozóiból kiindulva. A tagfüggvények esetében gyakori, hogy igen rövidek, és csak azt az objektumot várják paraméterként, amelyre meghívták őket, hiszen a működésükhöz szükséges adatok többnyire elérhetők az objektumon belül.

```
def __eq__(self, other):
    return self.x == other.x and self.y == other.y
```

A tagfüggvények neve általában nem kezdődhet és végződhet két aláhúzással – hacsak nem előre meghatározott különleges tagfüggvényről van szó. A Python minden összehasonlító művelethez biztosít ilyen különleges tagfüggvényeket (lásd a 6.1. táblázatban).

Az egyedi osztályok példányai alapértelmezés szerint támogatják az == műveletet, és az összehasonlítás mindig `False` eredményt ad. Ezt a viselkedést felülírhatjuk, ha újra megvalósítjuk az `__eq__()` tagfüggvényt – ezt tettük a fentiekben. Ha megvalósítjuk az `__eq__()` tagfüggvényt, a Python automatikusan biztosítja számunkra a `__ne__()` nevezetű ellentettpárját (vagyis a „nem egyenlő” műveletet), amennyiben mi magunk nem gondoskodunk a megvalósításáról.

Az egyedi osztályok példányai alapértelmezés szerint kivonatolhatók (hashable), így meghívhatjuk rájuk a `hash()` függvényt, továbbá szerepelhetnek szótárakban és halmazokban. A kivonatolhatóság végét jelenti azonban, ha újra megvalósítjuk az `__eq__()` tagfüggvényt – erre a helyzetre ad megoldást a később tárgyalt `FuzzyBool` osztály.

`FuzzyBool` ☞ 247

Ennek a különleges tagfüggvénynek a megvalósításával immár összehasonlíthatunk két `Point` objektumot, ha azonban egy `Point` objektumot egy másik típusú – például `int` – objektummal próbálunk összevetni, `AttributeError` kivételhez jutunk (hiszen az `int` értékek nem rendelkeznek `x` jellemzővel). Másrészről – köszönhetően a Python feketedoboz-szemléletének – a `Point` objektumokat összehasonlíthatjuk egyéb objektumokkal, amelyek rendelkeznek `x` jellemzővel, jóllehet az esetek többségében meglepő eredményt kapunk.

6.1. táblázat *Az összehasonlítás különleges tagfüggvényei*

Különleges tagfüggvény	Használat	Leírás
`__lt__(self, other)`	`x < y`	Az eredmény `True`, ha `x` kisebb, mint `y`.
`__le__(self, other)`	`x <= y`	Az eredmény `True`, ha `x` kisebb, mint `y`, vagy egyenlő azzal.
`__eq__(self, other)`	`x == y`	Az eredmény `True`, ha `x` egyenlő `y`-nal.
`__ne__(self, other)`	`x != y`	Az eredmény `True`, ha `x` nem egyenlő `y`-nal.
`__ge__(self, other)`	`x >= y`	Az eredmény `True`, ha `x` nagyobb, mint `y`, vagy egyenlő azzal.
`__gt__(self, other)`	`x > y`	Az eredmény `True`, ha `x` nagyobb, mint `y`.

Az értelmetlen összehasonlítások elkerülésére tehetünk néhány óvintézkedést. Alkalmazhatunk egy feltételezést – az `assert isinstance(other, Point)` éppen megteszi. Kiválthatunk egy `TypeError` kivételt is, jelezve, hogy az adott két típus összehasonlítását nem támogatjuk: `if not isinstance(other, Point): raise TypeError()`. Végezetül, a harmadik (és a leginkább „pythonos") módszer a következő kódrészlet használata: `if not isinstance(other, Point): return NotImplemented`. Ez esetben a `NotImplemented` visszatérési érték megjelenése esetén a Python megkísérli az `other.__eq__(self)` hívását – hátha a másik típus támogatja az összehasonlítást. Ha pedig nincs ilyen tagfüggvény, illetve itt is a `NotImplemented` értéket kapjuk vissza, a rendszer feladja, és kiváltja a `TypeError` kivételt. (Fontos megjegyeznünk, hogy a 6.1. táblázatban felsorolt különleges tagfüggvények esetében kizárólag az újra megvalósított változatok adhatják vissza a `NotImplemented` értéket.)

A beépített `isinstance()` függvény egy objektumot és egy osztályt (vagy osztályok rendezett sorozatát) várja, és akkor ad vissza `True` értéket, ha ez az objektum adott osztályból (illetve a sorozat egyik osztályából) való, illetve az osztály (vagy a sorozat egyik osztálya) ősének példánya.

```
def __repr__(self):
    return "Point({0.x!r}, {0.y!r})".format(self)
```

`str.format()` 75 ☞

A beépített `repr()` függvény meghívja az adott objektum `__repr__()` nevű különleges tagfüggvényét, és visszaadja az eredményt. A kapott karakterlánc kétféle típusba eshet. Az első esetén az eredményre a beépített `eval()` függvényt alkalmazva visszakaphatjuk a `repr()` számára megadott objektumot, míg a második típusnál ez nem járható út (a későbbiekben példát is láthatunk erre). Lássuk, hogyan kaphatunk egy `Point` objektumból karakterláncot, majd ebből ismét `Point` objektumot:

```
p = Shape.Point(3, 9)
repr(p)                                    # az eredmény: 'Point(3, 9)'
q = eval(p.__module__ + "." + repr(p))
repr(q)                                    # az eredmény: 'Point(3, 9)'
```

`import` ☞ 32, 192

Az `eval()` használata során meg kell adnunk a modul nevét, amennyiben a korábbiakban az `import Shape` utasítást alkalmaztuk. (Minderre nincs szükség, ha a betöltést másképp, például a `from Shape import Point` alakban végezzük el.) A Python minden objektumhoz biztosít néhány privát jellemzőt – ezek egyike a `__module__`, amely egy karakterlánccal megadja az objektumhoz tartozó modul nevét. Esetünkben ez a `"Shape"`.

A kódrészlet végén két `Point` objektum – a `p` és a `q` – áll előttünk, amelyek jellemzői megegyeznek, így az összehasonlítás során egyenlőnek bizonyulnak. Az `eval()` a paraméterként megadott karakterlánc végrehajtásának eredményét adja vissza – fontos tehát, hogy ez a karakterlánc érvényes Python-utasítás legyen.

Dinamikus kódvégrehajtás ☞ 349

```
    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

A beépített `str()` függvény működése hasonló a `repr()` függvénynél látottakhoz, azzal a különbséggel, hogy az objektum `__str__()` különleges tagfüggvényét hívja meg. Az eredmény a fejlesztők szándékai szerint emberi értelmezésre alkalmas alak – ugyanakkor éppen ezért nincs értelme az `eval()` függvénynek átadnunk. Előző példánk változóit véve ismét alapul, az `str(p)` (és hasonlóképpen az `str(q)`) eredményként a `'(3, 9)'` karakterláncot adja vissza.

Részletesen bemutattuk az aprócska `Point` osztályt – emellett pedig megismerkedtünk számos olyan háttérfolyamattal, amelyekről jó, ha tudunk, de egyébiránt ritkán kerülnek a szemünk elé. A `Point` osztály egy `(x, y)` koordinátapár tárolására alkalmas. Ez az adategység fontos részét képezi a körök leírásának, amint azt a fejezet elején láthattuk. A következőkben megfigyelhetjük, hogyan hozhatjuk létre egyedi `Circle` osztályunkat a `Point` leszármazottjaként, így nem újra megírnunk az `x` és `y` jellemzők, valamint a `distance_from_origin()` tagfüggvény kódját.

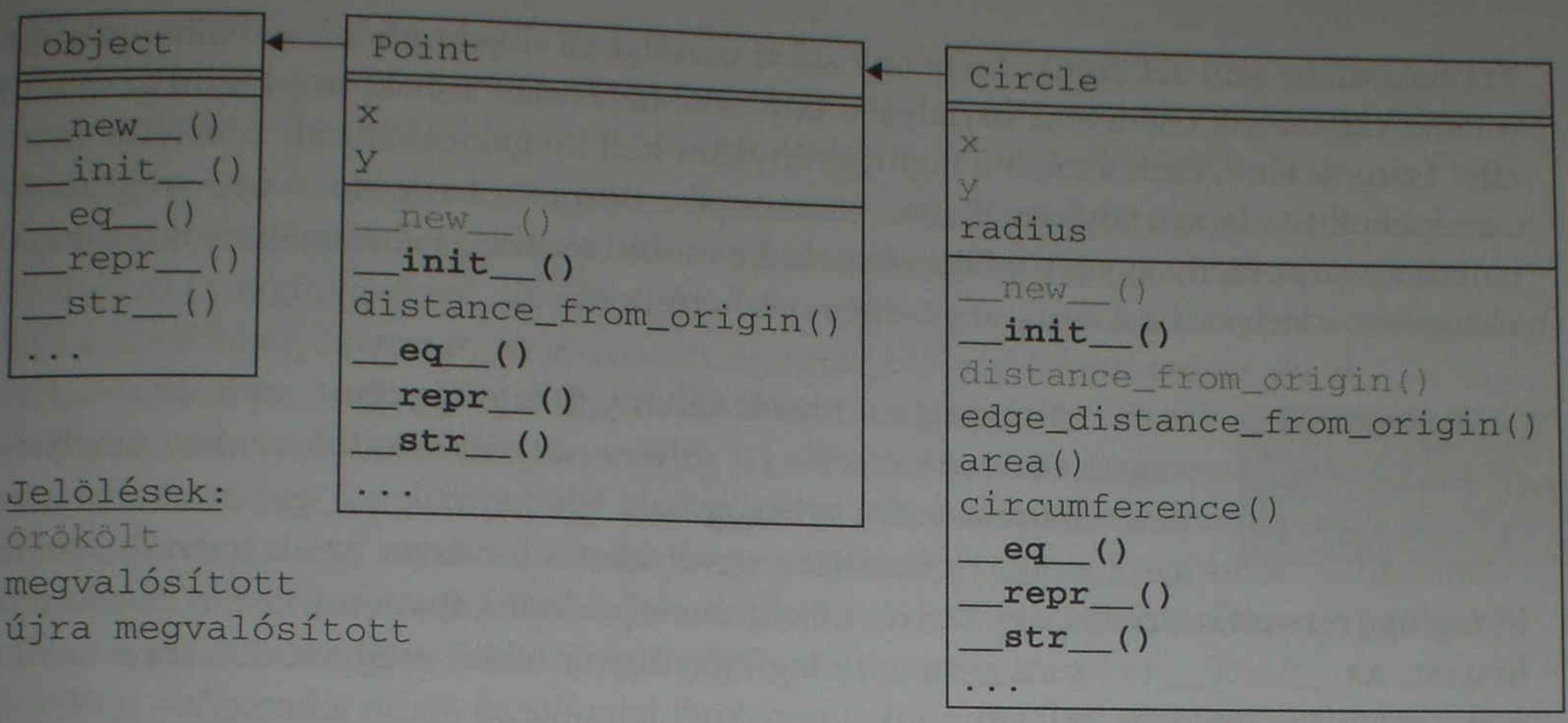
Öröklés és többalakúság

A `Circle` a `Point` osztályra épül, méghozzá az öröklés révén. Ősét egyetlen adatjellemzővel (`radius`) és három tagfüggvénnyel egészíti ki, emellett újra megvalósítja a `Point` néhány tagfüggvényét. Íme az osztály teljes meghatározása:

```
class Circle(Point):
def __init__(self, radius, x=0, y=0):
    super().__init__(x, y)
    self.radius = radius
def edge_distance_from_origin(self):
    return abs(self.distance_from_origin() - self.radius)
def area(self):
    return math.pi * (self.radius ** 2)
def circumference(self):
    return 2 * math.pi * self.radius
def __eq__(self, other):
    return self.radius == other.radius and super().__eq__(other)
def __repr__(self):
    return "Circle({0.radius!r}, {0.x!r}, {0.y!r})".format(self)
def __str__(self):
    return repr(self)
```

Az öröklléshez nem kell mást tennünk, mint felsorolni a kívánt őst (vagy ősöket) a `class` utasítás sorában.* Esetünkben a `Point` volt az alaposztály – a `Circle` öröklési hierarchiáját a 6.3. ábrán figyelhetjük meg.

* A többszörös öröklés, az elvont alaptípusok és más haladó objektumközpontú módszerek használatának tárgyalására a 8. fejezetben kerül sor.



6.3. ábra

A Circle osztály öröklési hierarchiája

Az `__init__()` tagfüggvény törzsében a `super()` segítségével meghívjuk az alaposztály `__init__()` tagfüggvényét – ez létrehozza és kezdőértékkel látja el a `self.x` és a `self.y` jellemzőket. Az osztály felhasználói nyugodtan megadhatnak érvénytelen sugárértékeket (mint például a `-2`) – a későbbiekben megmutatjuk, hogyan lehetünk úrrá ezen a helyzeten, ha a jellemzők sérülékenységét csökkentjük a tulajdonságok révén.

Az `area()` és a `circumference()` tagfüggvények kódja magától értetődő.
Az `edge_distance_from_origin()` számítása részeként meghívja a `distance_from_origin()` tagfüggvényt. Mivel ehhez a `Circle` osztály nem szolgál megvalósítással, a Python megtalálja a `Point` osztályban megadott alakot, és ezt használja fel. Az `__eq__()` tagfüggvényt ugyanakkor újra megvalósítjuk a `Circle` osztályban. A kódja összehasonlítja a két kör sugarát, és amennyiben ezek megegyeznek, a `super()` segítségével meghívja a szülőosztály `__eq__()` tagfüggvényét. Ha nem használtuk volna a `super()` tagfüggvényt, végtelen önhivatkozásba kerültünk volna, hiszen a `Circle.__eq__()` saját magát hívná meg. Érdemes észrevennünk, hogy a `super()` hívásánál nem kellett megadnunk a `self` paramétert, ugyanis a Python automatikusan megteszi ezt helyettünk.

Lássunk pár példát az osztály használatára:

```
p = Shape.Point(28, 45)
c = Shape.Circle(5, 28, 45)
p.distance_from_origin()          # az eredmény: 53.0
c.distance_from_origin()          # az eredmény: 53.0
```

A `distance_from_origin()` tagfüggvényt egyaránt meghívhatjuk `Point` és `Circle` objektumokra, hiszen az utóbbiak szerepelhetnek az előbbiek helyén.

Többalakúság alatt azt értjük, hogy egy adott osztálybeli objektumot használhatunk úgy, mintha valamelyik ősének az osztályába tartozna. Ez az oka annak, hogy amikor új alosztályt hozunk létre, csak azokat a tagfüggvényeket kell megvalósítanunk, amelyek nem szerepeltek az alaposztályban, illetve – amennyiben erre szükség van – újra megvalósíthatjuk az alaposztály egyes tagfüggvényeit. Ez utóbbi esetben is használhatjuk az eredeti változatot, amelyet a `super()` segítségével érhetünk el.

Sekély és mély másolás 143 ☜

A `Circle` esetében írtunk néhány új tagfüggvényt, mint az `area()` vagy a `circumference()`, és újra megvalósítottuk azokat, amelyeknek a módosítására szükség volt. Ide tartozik a `__repr__()` és az `__str__()` is, hiszen egyébként a rendszer az alaposztály megfelelő tagfüggvényeit használja, így `Point` objektumok leírását kapjuk a `Circle` objektumok helyett. Az `__init__()` és az `__eq__()` tagfüggvények újbóli megvalósítására is szükség volt, ugyanis számításba kell vennünk a köröknél jelentkező újabb jellemzőt – erőfeszítéseink mérséklésére azonban segítségül hívtuk az alaposztály megfelelő tagfüggvényeit is.

A `Point` és a `Circle` osztályok lehetőségei kedvünk szerint bővíthetők. Írhatunk például különleges tagfüggvényeket az összehasonlításhoz, hogy rendezni tudjuk a `Point` és `Circle` objektumokat. Érdemes lehet a megfelelő tagfüggvénnyel lehetővé tennünk a `Point`, illetve `Circle` objektumok másolását, ugyanis a legtöbb Python-osztály nem tesz elérhetővé `copy()` tagfüggvényt (kivételként megemlítenénk a `dict.copy()` és a `set.copy()` példáját). Ha másolási lehetőségekre vágyunk, könnyen célt érhetünk, csak töltsük be a `copy` modult, és máris hozzájutunk a `copy.copy()` függvényhez. (A `copy.deepcopy()` használatára ez esetben nincs szükség, ugyanis a `Point` és a `Circle` objektumok kizárólag állandó értékű példányváltozókat tartalmaznak.)

Jellemzők elérésének szabályozása tulajdonságokkal

A korábbiakban láthattuk, hogy a `Point` osztály tartalmaz egy `distance_from_origin()` nevű tagfüggvényt, a `Circle` osztályban pedig elérhető az `area()`, a `circumference()`, valamint az `edge_distance_from_origin()` tagfüggvény is. Mindegyikük egy-egy `float` értéket ad vissza, így a felhasználók szempontjából lehetnének akár (csak olvasható) adatjellemzők is. A `ShapeAlt.py` programban létrehoztuk az osztályok eredetitől eltérő változatait, amelyekben a fenti tagfüggvényeket tulajdonságok formájában valósítottuk meg. Mindez lehetőséget ad az alábbi kód végrehajtására:

```
circle = Shape.Circle(5, 28, 45) # feltételezi: import ShapeAlt as Shape
circle.radius                    # az eredmény: 5
circle.edge_distance_from_origin # az eredmény: 48.0
```

Íme a `ShapeAlt.Circle` osztály area és `edge_distance_from_origin` tulajdonságainak kiolvasó tagfüggvényei:

```
@property
def area(self):
    return math.pi * (self.radius ** 2)
```

```
    @property
    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin - self.radius)
```

Amennyiben csak a kiolvasó függvényeket adjuk meg, mint ahogy itt is tettük, a tulajdonságok csak olvashatók lesznek. Az `area` tulajdonság kódja ugyanaz, mint a korábban látott `area()` tagfüggvényé, az `edge_distance_from_origin` kódja azonban némiképp eltér a korábbitól, hiszen most a `distance_from_origin` tulajdonságot éri el a `distance_from_origin()` függvény helyett. A legfeltűnőbb eltérés mindkét esetben a *tulajdonságdíszítő* használatában rejlik – a díszítő nem más, mint egy függvény, amely paraméterként egy függvényt vagy tagfüggvényt vár, és visszaadja annak „díszített" változatát, vagyis egy olyan változatot, amely némiképp eltér a korábbitól. A díszítőket a nevük előtt feltüntetett @ jel jelzi. Jelenleg tekintsük őket a kódolás formai részének – a létrehozásuk lehetőségeiről a 8. fejezetben tanulunk majd.

A `property()` beépített díszítőfüggvény, amely négy paramétert vár: egy kiolvasó függvényt, egy beállító függvényt, egy törlő függvényt, valamint egy dokumentációs karakterláncot. Az `@property` használatával ugyanazt a hatást érjük el, mintha a `property()` függvényt egyetlen paraméterrel, a beolvasó függvénnyel hívtuk volna meg. Az `area` tulajdonságot így is létrehozhattuk volna:

```
def area(self):
    return math.pi * (self.radius ** 2)
area = property(area)
```

Ezt az alakot azonban ritkán használjuk, ugyanis a díszítővel rövidebb és tisztább kódot kapunk.

Az előzőekben megjegyeztük, hogy a `Circle` objektum `radius` jellemzőjén semmiféle érvényességvizsgálatot nem végzünk – ezen a helyzeten most változtathatunk, ha a jellemzőből tulajdonságot hozunk létre. Mindehhez nem kell megváltoztatnunk a `Circle.__init__()` tagfüggvényt, illetve semmilyen kódot, amely eléri a `Circle.radius` jellemzőt – minden változatlanul működik, azzal a különbséggel, hogy a rendszer ellenőrzi a `radius` értékének érvényességét, mihelyt beállítottuk.

A Python-programozók rendszerint tulajdonságokat alkalmaznak, nem pedig kifejezett kiolvasó és beállító függvényeket – mint például `getRadius()` vagy `setRadius()` –, mint más objektumközpontú nyelvekben. Mindennek az az oka, hogy itt igen egyszerűen tulajdonsággá alakíthatunk egy adatjellemzőt anélkül, hogy az osztály használatát bármilyen módon befolyásolnánk.

Ahhoz, hogy egy jellemzőt írható/olvasható tulajdonsággá alakítsunk, létre kell hoznunk egy privát jellemzőt, amelyben helyet kap maga az adat, valamint meg kell írnunk a megfelelő beállító és kiolvasó tagfüggvényeket. Íme a sugár kiolvasó és beállító tagfüggvényei a maguk teljes valójában, dokumentációs karakterláncokkal együtt:

```
    @property
    def radius(self):
        """The circle's radius

        >>> circle = Circle(-2)
        Traceback (most recent call last):
        ...
        AssertionError: radius must be nonzero and non-negative
        >>> circle = Circle(4)
        >>> circle.radius = -1
        Traceback (most recent call last):
        ...
        AssertionError: radius must be nonzero and non-negative
        >>> circle.radius = 6
        """
        return self.__radius

    @radius.setter
    def radius(self, radius):
        assert radius > 0, "radius must be nonzero and non-negative"
        self.__radius = radius
```

Az `assert` utasítás segítségével biztosítjuk, hogy a sugár ne lehessen nulla vagy negatív, az értékét pedig a `self.__radius` privát jellemzőben tároljuk. Figyeljük meg, hogy a kiolvasó és a beállító függvény (valamint a törlő függvény, amennyiben ilyenre is szükség van) neve megegyezik – a díszítők tesznek különbséget közöttük: ezek adnak nekik új neveket, hogy ne ütközzenek egymással.

A beállító függvény díszítője első pillantásra furcsának tűnhet. Minden általunk létrehozott tulajdonság rendelkezik `getter`, `setter` és `deleter` jellemzővel, így, mihelyt létrehoztuk a `radius` tulajdonságot az `@property` utasítással, a `radius.getter`, a `radius.setter` és a `radius.deleter` egy csapásra elérhetővé válik. A `radius.getter` jellemzőhöz a `@property` díszítő rendeli hozzá a kiolvasó függvényt, míg a másik kettőt a Python határozza meg. Az utóbbiak így nem tesznek semmit (vagyis nem írhatjuk vagy törölhetjük a tulajdonság tartalmát), hacsak nem alkalmazzuk őket díszítőként – ilyenkor önmagukat helyettesítik azzal a tagfüggvénnyel, amelyet díszítünk velük.

A `Circle` kezdeti értékadó tagfüggvénye, a `Circle.__init__()` tartalmazza a `self.radius = radius;` utasítást, amely meghívja a `radius` tulajdonság beállító tagfüggvényét, így ha az objektum létrehozásakor érvénytelen sugárértéket adunk meg, `AssertionError` kivételt kapunk. Hasonlóképpen, ha egy `Circle` objektum `radius` tulajdonságának érvénytelen értéket adunk meg, ismét a beállító függvény állja az utunkat, így kivételt kapunk. A dokumentációs karakterláncban megtalálhatjuk a megfelelő teszteket annak ellenőrzésére, hogy a kivétel az elvárt esetekben keletkezik-e.

Mind a `Point`, mind a `Circle` olyan adattípusok, amelyek elegendő lehetőséggel rendelkeznek ahhoz, hogy jó hasznukat vegyük. A legtöbb adattípus, amelyet a későbbiekben magunk hozunk létre, ehhez hasonló elvek mentén épül fel, de adódhatnak olyan esetek, amikor valóban mindenféle értelemben teljes adattípust kell készítenünk. Erre láthatunk példákat a következőkben.

Teljes és tökéletesen beágyazott adattípusok létrehozása

Teljes adattípus létrehozásánál alapvetően két lehetőség áll előttünk. Létrehozhatjuk a nulláról indulva – ilyenkor – jóllehet az osztály (hasonlóan a Python összes osztályához) az `object` leszármazottja – minden adatjellemzőt és tagfüggvényt nekünk kell meghatároznunk az egyetlen `__new__()` kivételével. A másik lehetőség, hogy az osztályt egy már létező adattípusból származtatjuk, amely hasonlít az elvárt típushoz. Ilyenkor a munkánk jobbára abból áll, hogy újra megvalósítjuk azokat a tagfüggvényeket, amelyeknek a működését meg szeretnénk változtatni, továbbá megszabadulunk azoktól, amelyekre nincs szükségünk a továbbiakban.

A következőkben először a nulláról indulva létrehozzuk a `FuzzyBool` adattípust, majd megfigyeljük azt is, hogy miként valósíthatjuk meg úgy, hogy az öröklés lehetőségét alkalmazzuk az erőfeszítéseink mérséklésére. A beépített `bool` típus két értéket (`True` és `False`) vehet fel, de a mesterséges intelligencia alkalmazásának bizonyos területein többállapotú logikai értékekre („fuzzy boolean”) van szükség, vagyis az „igaz” és „hamis” között is találunk igazságállapotokat. A mi megvalósításainkban az igazságértékek ábrázolására lebegőpontos számokat használunk, ahol a 0.0 a `False`, az 1.0 pedig a `True` értéket jelöli. Ebben a rendszerben a 0.5 azt jelenti, hogy a logikai érték 50%-ban igaz, a 0.25 pedig 25%-os igazságtartalmat takar. Az alábbiakban bemutatunk néhány példát a típus használatára (ezek ugyanazt az eredményt adják mindkét megvalósítás esetében):

```
a = FuzzyBool.FuzzyBool(.875)
b = FuzzyBool.FuzzyBool(.25)
a >= b                                  # az eredmény: True
bool(a), bool(b)                        # az eredmény: (True, False)
~a                                      # az eredmény: FuzzyBool(0.125)
a & b                                   # az eredmény: FuzzyBool(0.25)
b |= FuzzyBool.FuzzyBool(.5)            # a b értéke most: FuzzyBool(0.5)
"a={0:.1%} b={1:.0%}".format(a, b)      # az eredmény: 'a=87.5% b=50%'
```

Szeretnénk, ha a `FuzzyBool` típus támogatna minden összehasonlító műveletet (`<`, `<=`, `==`, `!=`, `>=`, `>`), valamint a három alapvető logikai műveletet, vagyis a `not` (`~`) az `and` (`&`) és az `or` (`|`) operátorokat. Mindemellett más logikai tagfüggvényeket is biztosítani szeretnénk, mint a `conjunction()` és a `disjunction()`, amelyek képesek tetszőleges számú `FuzzyBool` paraméter fogadására, és az eredményt egy `FuzzyBool` objektum alakjában adják vissza. Mindemellett az adattípus teljessé tételéhez lehetővé tesszük az átalakítást `bool`, `int`, `str` és `float` típusokká, és készítünk egy ábrázolási alakot is, amely az `eval()` segítségével kiértékelhető. Végezetül pedig elérjük azt is, hogy a `FuzzyBool` támogassa az `str.format()` formátumleírásait, a `FuzzyBool` objektumok használhatók legyenek

szótárak kulcsaiként és halmazok elemeiként, és állandó értékű típussá tesszük a `FuzzyBool`-t – ugyanakkor gondoskodunk róla, hogy az egyszerűbb használat érdekében a bővített értékadó műveletek (`&=` és `|=`) rendelkezésre álljanak.

A 6.1. táblázatban láthattuk az összehasonlítás különleges tagfüggvényeit, a 6.2. táblázat az alapvető különleges tagfüggvényeket sorolja fel, a 6.3. táblázat pedig a számműveletek különleges tagfüggvényeit mutatja be – ez utóbbiak közé tartoznak a bitenkénti műveletek (`~`, `&` és `|`), amelyeket a `FuzzyBool` objektumok logikai műveletként alkalmaznak, valamint a számtani műveletek, mint a `+` és a `-`, amelyeket a `FuzzyBool` osztály nem valósít meg, hiszen a használatuk értelmetlen lenne.

Adattípusok létrehozása nulláról

Ahhoz, hogy a nulláról indulva hozzuk létre a `FuzzyBool` típust, létre kell hoznunk egy adatjellemzőt, amely tárolja az objektum értékét, valamint az ennek kezeléséhez szükséges tagfüggvényeket. Íme a `class` utasítás sora, valamint az osztály kezdeti értékadó tagfüggvénye, ahogy a `FuzzyBool.py` fájlban látható:

```
class FuzzyBool:

    def __init__(self, value=0.0):
        self.__value = value if 0.0 <= value <= 1.0 else 0.0
```

A tulajdonságjellemzőt privát értékként határoztuk meg, hiszen állandó értékű `FuzzyBool` objektumokra van szükségünk, vagyis nem lenne jó, ha megengednénk a hozzáférést ehhez a jellemzőhöz. Ha pedig valamilyen, a megengedett tartományon kívül eső értéket adnánk meg, az objektum egy biztosan érvényes értéket (0.0, vagyis hamis) alkalmaz helyette. Ha visszaemlékszünk, az előzőekben a `ShapeAlt.Circle` esetében szigorúbb eljárást választottunk, és ha a `Circle` objektum a kezdeti értékadásnál érvénytelen sugárértékkel találkozott, kivételt váltott ki. A `FuzzyBool` családfáját a 6.4. ábrán tekinthetjük meg.

A legegyszerűbb logikai művelet a `not`, amelyet a bitenkénti ellentettművelettel (`~`) jelenítünk meg:

```
def __invert__(self):
    return FuzzyBool(1.0 - self.__value)
```

6.2. táblázat *Alapvető különleges tagfüggvények*

Különleges tagfüggvény	Használat	Leírás
`__bool__(self)`	`bool(x)`	Ha megadjuk, az `x` igazságértékét adja vissza; hasznos az `if x: ...` kifejezésekben.
`__format__(self, format_spec)`	`"{0}".format(x)`	Az `str.format()` támogatását biztosítja egyedi osztályoknál.
`__hash__(self)`	`hash(x)`	Ha megadjuk, az `x` alkalmazható szótárkulcsként vagy halmazok elemeként.

6.2. táblázat *Alapvető különleges tagfüggvények* *(folytatás)*

Különleges tagfüggvény	Használat	Leírás
`__init__(self, paraméterek)`	`x = X(paraméterek)`	A rendszer az objektum kezdeti értékadásánál hívja meg.
`__new__(cls, paraméterek)`	`x = X(paraméterek)`	A rendszer az objektum létrehozásánál hívja meg.
`__repr__(self)`	`repr(x)`	Az `x` karakterlánc-ábrázolását adja vissza; ahol ez lehetséges, `eval(repr(x)) == x`
`__repr ascii(x) __(self)`		Az `x` csak ASCII-karakterekből álló ábrázolását adja vissza.
`__str__(self)`	`str(x)`	Az `x` emberi értelmezésre alkalmas karakterlánc-ábrázolását adja vissza.

`__new__()` ☞ 256, 399

`str.format()` 75 ☜

A `__del__(self)` különleges tagfüggvény

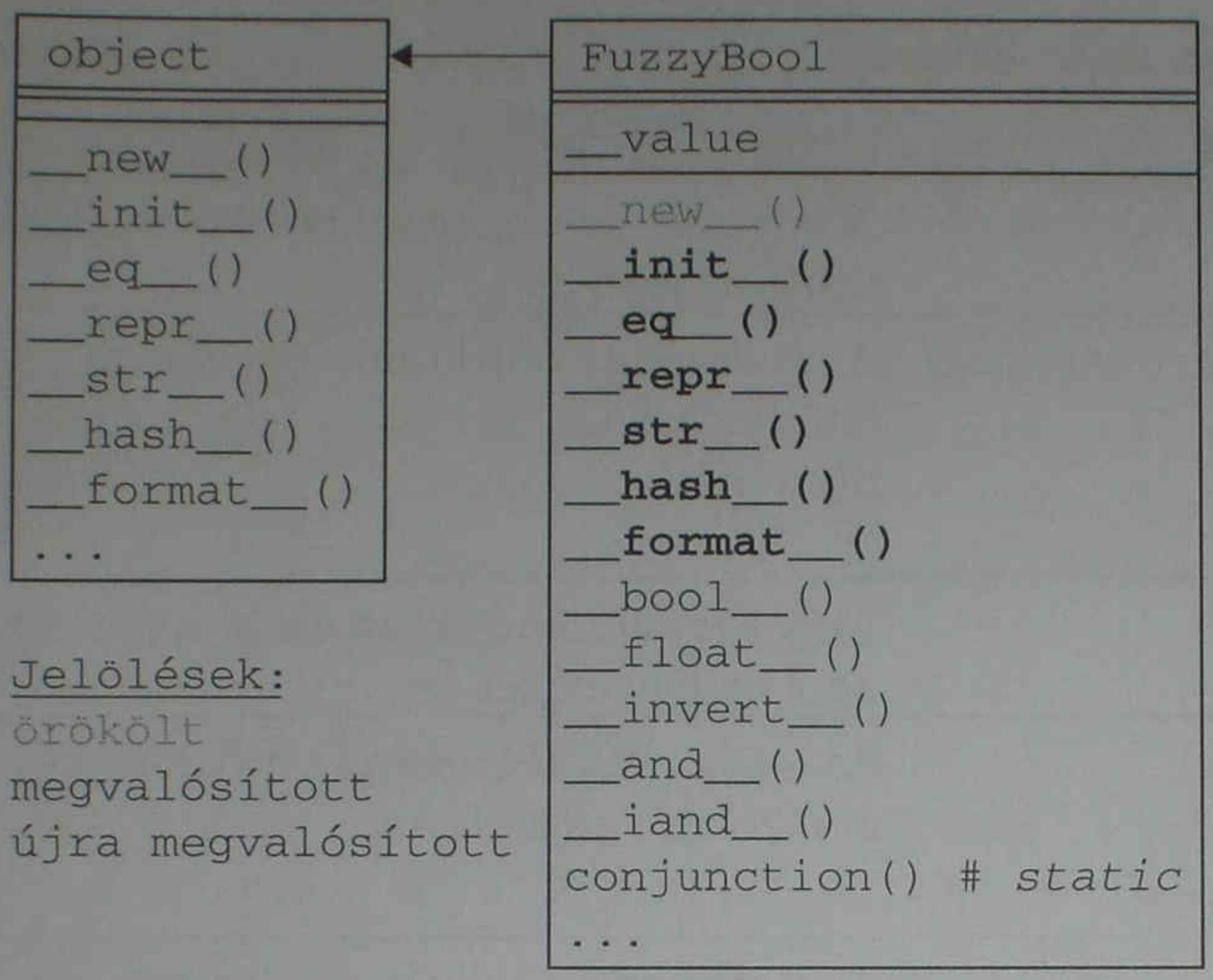
Figyelem!

A `__del__(self)` *különleges tagfüggvényt akkor hívja meg a rendszer, amikor egy objektumot megsemmisít – legalábbis elméletben. A gyakorlatban azonban előfordulhat, hogy a tagfüggvény hívása soha nem kerül sorra, még a program befejezésekor sem. Ráadásul, ha a* `del x` *utasítással élünk, a rendszer mindössze az* `x` *nevű objektumhivatkozást törli, és eggyel csökkenti annak a számlálónak az értékét, amelyik a hivatkozott objektum hivatkozásainak számát tartja nyilván.*
A `__del__()` *hívására csak akkor számíthatunk, ha ez a számláló eléri a 0 értéket – jóllehet a Python még ezt sem garantálja. Mindennek ismeretében érthető, hogy ritkán találkozunk a* `__del__()` *újra megvalósított változatával (könyvünk példáiban egyszer sem), továbbá éppen ezért nem érdemes az erőforrások felszabadítására használnunk, vagyis nem alkalmas fájlok bezárására, illetve hálózati vagy adatbázis-kapcsolatok bontására.*

A Python kétféle módot ad arra, hogy biztonságosan felszabadítsuk a lefoglalt erőforrásokat. Az egyik a már megismert `try...finally` *blokkok használata, amelyekkel ismét foglalkozunk a 7. fejezetben, a másik pedig egy környezeti objektum és a* `with` *utasítás együttes alkalmazása – róluk a 8. fejezetben ejtünk szót.*

A bitenkénti logikai `and` művelet (`&`) megvalósításáról az `__and__()` különleges tagfüggvény gondoskodik, az értékadó változatot (`&=`) pedig az `__iand__()` biztosítja:

```
def __and__(self, other):
    return FuzzyBool(min(self.__value, other.__value))
```



6.4. ábra
A FuzzyBool osztály öröklési hierarchiája

```
def __iand__(self, other):
    self.__value = min(self.__value, other.__value)
    return self
```

A bitenkénti `and` művelet egy új `FuzzyBool` objektumot ad vissza az aktuális, valamint egy másik objektumot alapul véve, míg az értékadó változat a privát értéket módosítja. Szigorúan véve ez nem felel meg az állandó értékű típusok viselkedésének, de mégis hasonlít több állandó értékű Python-típushoz – ilyen például az `int`, ahol a `+=` használatakor úgy tetszik, mintha a bal oldali tényezőt módosítanánk, holott a rendszer egy új `int` objektumhoz rendeli hozzá az összeadás eredményét. Esetünkben azonban erre az újrakötésre nincs is szükség, hiszen magát a `FuzzyBool` objektumot változtatjuk meg. Végül pedig, azért a `self` értéket adjuk vissza, hogy lehetővé tegyük a műveletek láncba rendezését.

Megvalósíthatjuk továbbá a `__rand__()` különleges tagfüggvényt is, amelyet a rendszer akkor hív meg, ha a `self` és az `other` különböző típusúak, és erre a párra nem valósítottuk meg az `__and__()` tagfüggvényt. `FuzzyBool` osztályunkhoz erre nincs szükség. A bináris műveletek különleges tagfüggvényei többnyire egyaránt rendelkeznek „i” és „r” változattal – az előbbi az értékadó alakhoz tartozik, míg az utóbbi megcseréli a tényezőket.

Nem mutatjuk be a bitenkénti `or` művelethez (`|`) tartozó `__or__()`, illetve értékadó változatához tartozó `__ior__()` különleges tagfüggvényeket, hiszen a kódjuk pontosan úgy fest, mint az `and` esetében, azzal a különbséggel, hogy ezúttal a `self` és az `other` értékeinek maximumát vesszük a minimum helyett.

```
def __repr__(self):
    return ("{0}({1})".format(self.__class__.__name__,
```

Létrehoztuk az objektum ábrázolási alakját, amelyet az `eval()` segítségével kiértékelhetünk. Így az `f = FuzzyBool.FuzzyBool(.75);` utasítást követően a `repr(f)` eredményeként a `'FuzzyBool(0.75)'` karakterláncot kapjuk.

Minden objektum rendelkezik néhány különleges jellemzővel, amelyeket a Python biztosít a számára – egyikük a `__class__`, amely az objektum osztályát adja meg. Minden osztálynál elérhető a privát `__name__` jellemző, amelyhez szintén automatikusan hozzájutunk. Ezeket a jellemzőket az ábrázolási alaknál az osztály nevének előállítására használtuk. Mindez azt jelenti, hogy ha a `FuzzyBool` osztályból alosztályokat származtatunk, hogy újabb tagfüggvényeket vehessünk használatba, az örökölt `__repr__()` tagfüggvényt nem kell újra megvalósítanunk, hiszen automatikusan az alosztály nevét olvassa ki.

```
def __str__(self):
    return str(self.__value)
```

Az egyszerű karakterlánc alakban az objektum lebegőpontos értékét adjuk vissza kartakterláncként formázva. A végtelen ciklus elkerülésére nem szükséges a `super()` hívást alkalmaznunk, mivel az `str()` tagfüggvényt a `self.__value` jellemzőre, nem pedig magára az objektumra hívjuk meg.

```
def __bool__(self):
    return self.__value > 0.5
def __int__(self):
    return round(self.__value)
def __float__(self):
    return self.__value
```

A `__bool__()` különleges tagfüggvény az objektumot logikai értékké alakítja, vagyis minden esetben `True` vagy `False` értéket ad vissza. Hasonlóképpen, az `__int__()` különleges tagfüggvény egésszé alakítja az objektumot. Ez utóbbi megvalósításához a beépített `round()` függvényt alkalmaztuk, mivel az `int()` egyszerűen csonkolja a megadott értéket (vagyis az 1.0 kivételével minden `FuzzyBool` objektum esetében 0-t adna vissza). A lebegőpontos számmá alakítás egyszerű feladat, hiszen az objektumban tárolt érték eleve lebegőpontos.

```
def __lt__(self, other):
    return self.__value < other.__value
def __eq__(self, other):
    return self.__value == other.__value
```

Ahhoz, hogy biztosítsuk az összehasonlító műveletek (<, <=, ==, !=, >=, >) mindegyikét, legalább hármat meg kell valósítanunk közülük – a <, a <= és a == alapján azután a Python képes létrehozni a párjaikat, vagyis a >, a >= és a != műveleteket. Közülük is csak kettőt mutattunk be, hiszen általában igencsak hasonlóak.*

* Valójában csak a fent bemutatott `__lt__()` és `__eq__()` tagfüggvényeket valósítottuk meg – a többiről a rendszer automatikusan gondoskodott. Ennek mikéntjéről a 8. fejezetben ejtünk szót.

6.3. táblázat *Számműveleti és bitenkénti különleges tagfüggvények*

Különleges tagfüggvény	Használat	Különleges tagfüggvény	Használat
`__abs__(self)`	`abs(x)`	`__complex__(self)`	`complex(x)`
`__float__(self)`	`float(x)`	`__int__(self)`	`int(x)`
`__index__(self)`	`bin(x)` `oct(x)` `hex(x)`	`__round__(self, számjegyek)`	`round(x, számjegyek)`
`__pos__(self)`	`+x`	`__neg__(self)`	`-x`
`__add__(self, other)`	`x + y`	`__sub__(self, other)`	`x - y`
`__iadd__(self, other)`	`x += y`	`__isub__(self, other)`	`x -= y`
`__radd__(self, other)`	`y + x`	`__rsub__(self, other)`	`y - x`
`__mul__(self, other)`	`x * y`	`__mod__(self, other)`	`x % y`
`__imul__(self, other)`	`x *= y`	`__imod__(self, other)`	`x %= y`
`__rmul__(self, other)`	`y * x`	`__rmod__(self, other)`	`y % x`
`__floordiv__(self, other)`	`x // y`	`__truediv__(self, other)`	`x / y`
`__ifloordiv__(self, other)`	`x //= y`	`__itruediv__(self, other)`	`x /= y`
`__rfloordiv__(self, other)`	`y // x`	`__rtruediv__(self, other)`	`y / x`
`__divmod__(self, other)`	`divmod(x, y)`	`__rdivmod__(self, other)`	`divmod(y, x)`
`__pow__(self, other)`	`x ** y`	`__and__(self, other)`	`x & y`
`__ipow__(self, other)`	`x **= y`	`__iand__(self, other)`	`x &= y`
`__rpow__(self, other)`	`y ** x`	`__rand__(self, other)`	`y & x`
`__xor__(self, other)`	`x ^ y`	`__or__(self, other)`	`x | y`
`__ixor__(self, other)`	`x ^= y`	`__ior__(self, other)`	`x |= y`
`__rxor__(self, other)`	`y ^ x`	`__ror__(self, other)`	`y | x`
`__lshift__(self, other)`	`x << y`	`__rshift__(self, other)`	`x >> y`
`__ilshift__(self, other)`	`x <<= y`	`__irshift__(self, other)`	`x >>= y`
`__rlshift__(self, other)`	`y << x`	`__rrshift__(self, other)`	`y >> x`
		`__invert__(self)`	`~x`

```
def __hash__(self):
    return hash(id(self))
```

Alapértelmezés szeint az egyedi osztályok példányai támogatják az == műveletet (amely minden esetben a `False` értékkel tér vissza), továbbá kivonatolhatók (vagyis alkalmazhatók szótárak kulcsaiként, és elhelyezhetők halmazokban). Ha azonban újra megvalósítjuk az `__eq__()` különleges tagfüggvényt az egyenlőség valóban megfelelő vizsgálatához, az objektumok a továbbiakban nem lesznek kivonatolhatók. A gondokat orvosolhatjuk egy `__hash__()` különleges tagfüggvénnyel – éppen ezt tettük itt is.

A Python kivonatoló függvényeket biztosít karakterláncok, számok, fagyasztott halmazok és más osztályok számára. Esetünkben egyszerűen a beépített `hash()` függvényt használtuk (amely képes kezelni bármilyen típust, amely rendelkezik a `__hash__()` különleges tagfüggvénnyel), átadva neki az objektum egyedi azonosítóját a kivonat (hash) kiszámításához. (Nem használhattuk a `self.__value` privát változót, hiszen ennek értéke módosulhat egy bővített értékadó műveletben, míg az objektum kivonatértéke sosem változhat.)

A beépített `id()` függvény visszaadja a számára paraméterként megadott objektumhoz tartozó egész értékű azonosítót. Ez leggyakrabban az objektum memóriabeli címe, de biztosak csak abban lehetünk, hogy minden objektum azonosítója különböző. A színfalak mögött az `is` művelet is az `id()` függvényt alkalmazza annak megállapítására, hogy két objektumhivatkozás vajon ugyanarra az objektumra utal-e.

```
def __format__(self, format_spec):
    return format(self.__value, format_spec)
```

A beépített `format()` függvényre igazán csak az osztálymeghatározásoknál van szükség., A függvény egyetlen objektumot, valamint esetleg egy formátumleírást vár, és az objektum megfelelően formázott alakjával tér vissza.

Minden beépített osztály rendelkezik megfelelő `__format__()` tagfüggvénnyel – ez esetben a `float.__format__()` tagfüggvényt alkalmazzuk, átadva neki az objektum lebegőpontos értékét és a formátumleírást. Ugyanezt az eredményt elérhettük volna az alábbi kóddal is:

```
def __format__(self, format_spec):
    return self.__value.__format__(format_spec)
```

A `format()` függvény használata egy kicsit kevesebb gépelést igényel, ráadásul az eredmény is átláthatóbb. Mindazonáltal semmi sem kötelez arra, hogy ezt a függvényt használjuk – létrehozhatjuk a saját formátumleíró nyelvünket is, és megvalósíthatjuk a `__format__()` tagfüggvényben. Mindössze arra kell ügyelnünk, hogy végül karakterláncot adjunk vissza.

```
@staticmethod
def conjunction(*fuzzies):
    return FuzzyBool(min([float(x) for x in fuzzies]))
```

A beépített `staticmethod()` függvény rendeltetése, hogy díszítőként használjuk – éppen ezt tettük a fentiekben. A statikus tagfüggvények mindössze annyiban térnek el korábban megismert társaiktól, hogy a Python *nem* adja át nekik automatikusan a `self` értéket vagy bármilyen más paramétert.

Az `&` műveletet láncban is használhatjuk, így ha adottak az `f`, `g` és `h` `FuzzyBool`-objektumok, konjunkciójukat megkaphatjuk az `f & g & h` alakban. Ez nagyszerűen működik, ha egy maroknyi objektumról van szó, ha azonban már egy tucatnyival dolgozunk, kérdések merülnek fel a művelet hatékonyságával kapcsolatban, hiszen minden `&` jel egy függvényhívást jelent. A fenti tagfüggvényt alkalmazva ugyanezt egyetlen függvényhívással elérhetjük, méghozzá így: `FuzzyBool.FuzzyBool.conjunction(f, g, h)`. Mindezt kicsit rövidebb alakban is elvégezhetjük egy `FuzzyBool`-példány megadásával, de mivel a statikus tagfüggvények nem jutnak hozzá a `self` paraméterhez, ezt magunknak kell átadnunk, mint az `f.conjunction(f, g, h)` esetében.

A megfelelő `disjunction()` függvényt nem mutatjuk be, hiszen társától csak nevében, illetve abban tér el, hogy a `min()` helyett a `max()` függvényt alkalmazza.

Egyes Python-programozók úgy tartják, hogy a statikus tagfüggvények használata nem elég „pythonos”, és csak akkor folyamodnak hozzájuk, ha más nyelvekből (mint a C++ vagy a Java) kapott kódokat kell átalakítaniuk, illetve ha olyan tagfüggvényt írnak, amely nem használja a `self` paramétert. A Pythonban a statikus tagfüggvények helyett jobb, ha modulfüggvényeket írunk (lásd a következőkben), vagy osztálytagfüggvényeket hozunk létre (róluk a fejezet vége felé esik szó).

Hasonlóan, ha létrehozunk egy változót az osztály meghatározásán belül, de a tagfüggvényeken kívül, statikus változóhoz (osztályváltozóhoz) jutunk. Az állandók esetében kényelmesebb modulszintű globális változókat alkalmaznunk, de az osztályváltozók jó szolgálatot tehetnek, ha az osztály példányai között szeretnénk adatokat megosztani.

Végére értünk hát a `FuzzyBool` osztály megvalósításának, amelyet valóban a „nulláról” kezdtünk. Összesen 15 tagfüggvényt kellett újra megvalósítanunk (illetve 17-et, ha a mind a négy alapvető összehasonlító műveletet beleszámoljuk), továbbá két statikus tagfüggvényt. A következőkben az osztály egy újabb változatát hozzuk létre, ezúttal a `float` típusból származtatva. Itt mindössze nyolc tagfüggvény újbóli megvalósítására lesz szükség, továbbá két modulfüggvény megírására – emellett pedig 32 tagfüggvényt ki kell iktatnunk.

A legtöbb objektumközpontú nyelvben az öröklés során olyan új osztályokat hozunk létre az ősökből, amelyek rendelkeznek a szülő minden tagfüggvényével és jellemzőjével, és ezek mellé jönnek azok az új tagfüggvények és jellemzők, amelyeket újdonságként kívánunk bevezetni. A Python mindezt teljes mértékben támogatja, így írhatunk új tagfüggvényeket, és újra megvalósíthatjuk a régieket, módosítva a viselkedésüket. Mindemellett

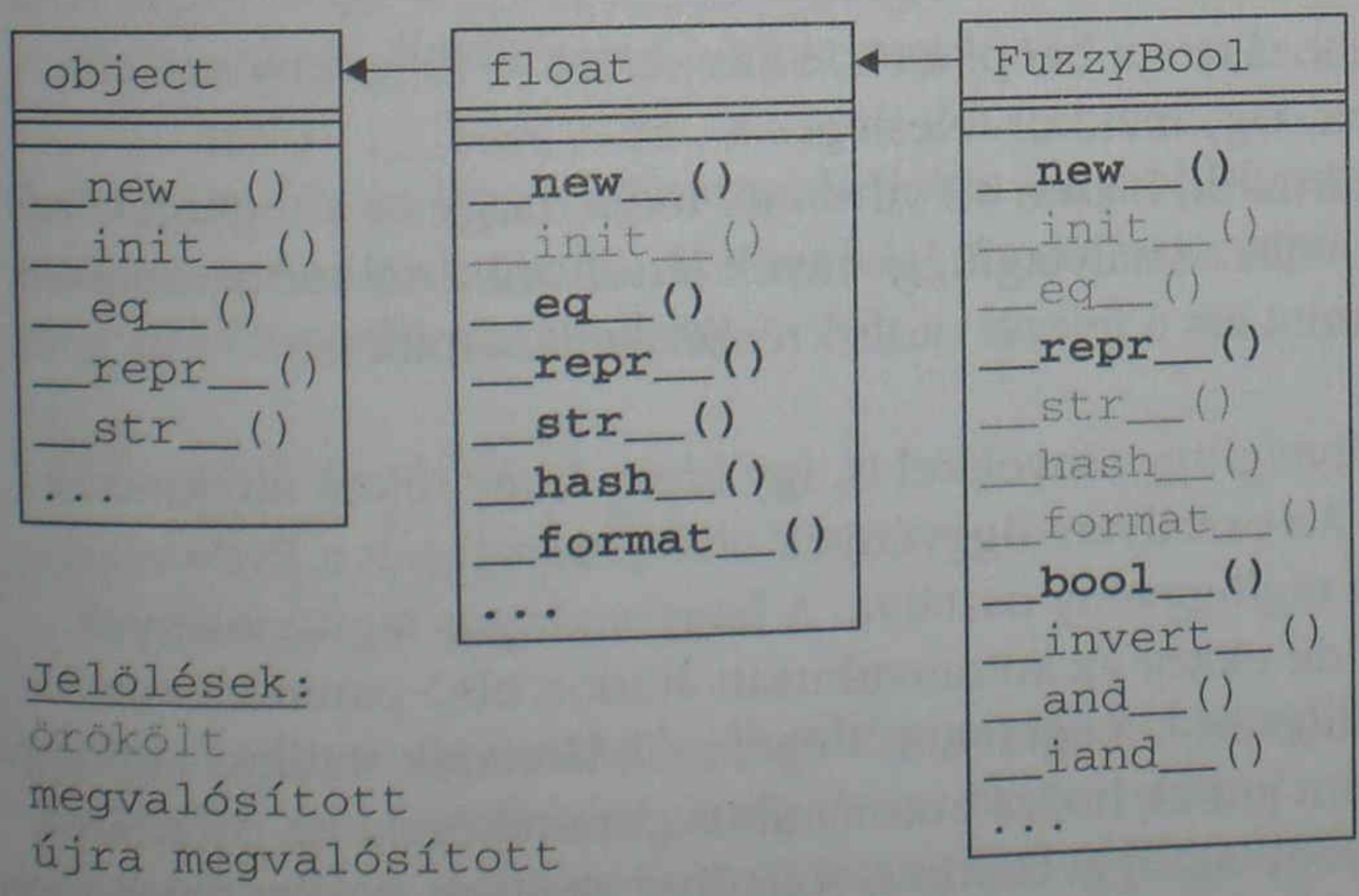
azonban lehetőségünk van egyes tagfüggvények kiiktatására is, így elérhetjük, hogy az új osztály úgy viselkedjen, mintha nem rendelkezne bizonyos örökölt tagfüggvényekkel. Az objektumközpontú programozás igazi elkötelezettjei persze idegenkedhetnek ettől, hiszen így elvész a többalakúság – mindazonáltal a Pythonban néha igen jól jöhet ez a lehetőség.

Adattípusok létrehozása más típusokból kiindulva

A `FuzzyBool` osztály következőkben bemutatott megvalósítását a `FuzzyBoolAlt.py` fájlban találhatjuk meg. Nyomban láthatjuk, hogy különbözik az előző változattól, hiszen a `conjunction()` és a `disjunction()` korábban statikus tagfüggvények voltak, most azonban modulfüggvények alakjában tesszük őket elérhetővé, valahogy így:

```
def conjunction(*fuzzies):
    return FuzzyBool(min(fuzzies))
```

A mindezt megvalósító kód számottevően egyszerűbb, mint a korábbiakban látott, ugyanis a `FuzzyBoolAlt.FuzzyBool` objektumok a `float` alosztályai, így közvetlenül használhatók `float` értékek helyett mindenféle átalakítás nélkül. (Az öröklési fát a 6.5. ábrán láthatjuk). A függvények elérése is egyszerűbb, mint előzőleg – korábban a modult és az osztályt kellett megadnunk (vagy egy példányt használnunk), az `import FuzzyBoolAlt` utasítást követően viszont ezúttal az egyszerűbb `FuzzyBoolAlt.conjunction()` alakot használhatjuk.



6.5. ábra

A FuzzyBool osztály újabb változatának öröklési hierarchiája

Íme a `FuzzyBool` osztály `class` sora és `__new__()` tagfüggvénye:

```
class FuzzyBool(float):
    def __new__(cls, value=0.0):
        return super().__new__(cls,
            value if 0.0 <= value <= 1.0 else 0.0)
```

Ha új osztályt hozunk létre, az többnyire változtatható, és a nyers, kezdőértékek nélküli objektum létrehozásához az `object.__new__()` tagfüggvényre támaszkodik. Az állandó értékű objektumok esetében azonban a létrehozást és a kezdeti értékadást egy lépésben kell elvégeznünk, hiszen ezek az objektumok a születésük után nem módosulhatnak.

A `__new__()` tagfüggvényt a rendszer még azelőtt meghívja, hogy egyetlen objektumot is létrehozott volna (hiszen éppen az objektumok létrehozása a `__new__()` feladata), így a `self` objektumot nem adhatjuk át neki, mivel az még nem is létezik. A `__new__()` valójában az osztálytagfüggvények közé tartozik, amelyek annyiban térnek el „rendes" társaiktól, hogy magára az osztályra hívjuk meg őket, nem pedig annak egy példányára, és a Python első paraméterként ezt az osztályt adja át a számukra. Az osztályt azonosító `cls` változónév mindössze a közmegegyezés miatt alakult így, hasonlóan az objektumokat azonosító `self` névhez.

Így ha az `f = FuzzyBool(0.7)` utasítást alkalmazzuk, a háttérben a Python a `FuzzyBool.__new__(FuzzyBool, 0.7)` hívással létrehoz egy új objektumot (nevezzük mondjuk `fuzzy`-nak), majd a `fuzzy.__init__()` hívásával elvégzi a további kezdeti értékadásokat, végül pedig a `fuzzy` egy objektumhivatkozását adja vissza – ez lesz az `f` értéke. A `__new__()` tagfüggvény feladatainak nagy részét az alaposztálybeli megvalósítása, vagyis az `object.__new__()` végzi el, nekünk csak azt kell biztosítanunk, hogy az érték a megfelelő tartományba essen.

Az osztálytagfüggvények létrehozására a beépített `classmethod()` függvény szolgál, amelyet díszítőként használunk. Ugyanakkor felesleges a `def __new__()` előtt az `@classmethod` kulcsszót kiírnunk, hiszen a Python jól tudja, hogy ez a tagfüggvény csak osztálytagfüggvény lehet. Más osztálytagfüggvények létrehozásánál természetesen szükségünk van a díszítőre, amint azt a fejezet utolsó részében láthatjuk is.

Megismerkedtünk hát az osztálytagfüggvényekkel is, így ideje, hogy újfent áttekintsük a Python tagfüggvénytípusait. Az osztálytagfüggvények első paraméterét a Python adja, ennek értéke nem más, mint a tagfüggvény osztálya. A hagyományos tagfüggvények esetében is hasonló a helyzet, de ekkor az automatikusan átadott első paraméter az az objektum, amelyikre a tagfüggvényt meghívtuk. Végezetül, léteznek statikus tagfüggvények is, amelyek egyáltalán nem jutnak hozzá automatikus paraméterekhez. Mindezek mellett természetesen mindhárom tagfüggvénytípus fogadhat általunk átadott paramétereket is (az előbbi két esetben a második, az utóbbiban pedig az első paramétertől kezdve).

```
def __invert__(self):
    return FuzzyBool(1.0 - float(self))
```

Ezzel a tagfüggvénnyel támogatjuk a bitenkénti `not` műveletet (`~`), éppúgy, mint a korábbiakban. Vegyük észre, hogy a `FuzzyBool` értékét tartalmazó privát változó helyett közvetlenül a `self` objektumot érjük el. Mindez annak köszönhető, hogy az osztályunkat

a `float` típusból származtattuk, így a `FuzzyBool` objektumok bárhol használhatók, ahol a `float` értékek – már amennyiben nem próbáljuk meghívni a `FuzzyBool` kiiktatott tagfüggvényeit.

```
def __and__(self, other):
    return FuzzyBool(min(self, other))
def __iand__(self, other):
    return FuzzyBool(min(self, other))
```

A fenti függvények szerkezete megegyezik a korábbiakban látottakkal (jóllehet a kód kissé eltér), és az `__invert__()` tagfüggvényhez hasonlóan a `self` és az `other` objektumokat úgy használhatjuk, mintha csak `float` típusúak lennének. Az `or` változatait nem mutatjuk be, hiszen csak a nevükben (`__or__()` és `__ior__()`), valamint abban térnek el, hogy a `max()`, nem pedig a `min()` függvényt alkalmazzák.

```
def __repr__(self):
    return ("{0}({1})".format(self.__class__.__name__,
                              super().__repr__()))
```

A `__repr__()` tagfüggvényt újra meg kell valósítanunk, hiszen alaposztálybeli változata, a `float.__repr__()` egyszerűen karakterláncként adja vissza az objektumban tárolt számot, míg nekünk az osztály nevére is szükségünk van, hogy kiértékelhessük az `eval()` segítségével. Az `str.format()` második paramétereként nem adhatjuk át a `self` objektumot, ugyanis így a `__repr__()` tagfüggvény hívásának végtelen ciklusába kerülnénk, ezért az alaposztálybeli változatot hívjuk meg.

Az `__str__()` tagfüggvényt nem kell újra megvalósítanunk, mivel az alaposztálybeli változat, a `float.__str__()` megfelel, és ezt használjuk a `FuzzyBool.__str__()` újra megvalósított változata helyett.

```
def __bool__(self):
    return self > 0.5
def __int__(self):
    return round(self)
```

Ha egy `float` objektumot logikai környezetben használunk, az értéke `False` lesz, ha eredetileg 0.0 volt, egyébként pedig `True`. Ez `FuzzyBool` objektumok esetén nem alkalmas viselkedés, így a tagfüggvényt újra meg kell valósítanunk. Hasonló a helyzet az `int()` függvénnyel is, hiszen az `int(self)` egyszerűen csonkolja a megadott értéket, így az 1.0 kivételével 0 eredményt kapunk. Ahhoz, hogy a megoldás megfeleljen az elvárásainknak, a `round()` függvényt vesszük használatba, amely 0.5 alatt a 0, felette pedig az 1 értéket adja vissza.

A `__hash__()`, a `__format__()`, valamint az összehasonlító műveleteket támogató tagfüggvényeket nem kellett újra megvalósítanunk, hiszen a `float` alaposztály által biztosított viselkedés tökéletesen megfelel a `FuzzyBool` objektumok esetében is.

Az újra megvalósított tagfüggvényekből összeáll a `FuzzyBool` osztály – és így sokkal kevesebb kódra van szükség, mint az előzőekben bemutatott változatnál. Van itt azonban egy aprócska gond: a `FuzzyBool` több mint 30 olyan tagfüggvényt örökölt, amelyeknek a használata nem egyeztethető össze a `FuzzyBool` objektum elvárt viselkedésével. Így például az egyszerű szám- vagy bitenkénti műveletek (+, -, *, /, <<, >> stb.) egyike sem alkalmazható értelmesen ebben a környezetben. Az alábbiakban láthatjuk, hogyan kezdhetünk hozzá az összeadás kiiktatásához:

```
def __add__(self, other):
    raise NotImplementedError()
```

Hasonló kódot kell írnunk az `__iadd__()` és az `__radd__()` tagfüggvények számára is, hogy az összeadás minden formáját megakadályozzuk. (Fontos tudnunk, hogy a `NotImplementedError` egy szabványos kivétel, amely nem tévesztendő össze a `NotImplemented` objektummal.) A `NotImplementedError` kivétel kiváltása helyett – különösen, ha utánozni szeretnénk a Python beépített osztályainak viselkedését – választhatjuk a `TypeError`-t is. Az alábbiakban bemutatjuk, miként érhetjük el, hogy a `FuzzyBool.__add__()` pontosan úgy viselkedjen, mint a beépített osztályok, amikor érvénytelen művelettel találkoznak:

```
def __add__(self, other):
    raise TypeError("unsupported operand type(s) for +: "
                    "'{0}' and '{1}'".format(
                self.__class__.__name__, other.__class__.__name__))
```

Az egytényezős műveletek esetében, amennyiben a kiiktatást a beépített típusok viselkedését utánozva szeretnénk megvalósítani, kicsit egyszerűbb dolgunk van:

```
def __neg__(self):
    raise TypeError("bad operand type for unary -: '{0}'".format(
                    self.__class__.__name__))
```

Az összehasonlító műveletek esetében a helyzet jelentősen egyszerűsödik. Az == kiiktatására például az alábbiakat írhatjuk:

```
def __eq__(self, other):
    return NotImplemented
```

Ha egy tagfüggvény, amely összehasonlító műveletet (<, <=, ==, !=, >=, >) valósít meg, a beépített `NotImplemented` objektumot adja vissza, amikor kísérletet teszünk a használatára, a Python először megfordítja az összehasonlítást, felcserélve a tényezőket (hátha a másik objektum rendelkezik a megfelelő összehasonlító tagfüggvénnyel, ha már a `self` nem), ha pedig ez sem vezet eredményre, a rendszer `TypeError` kivételt vált ki egy üzenettel, amely arról tájékoztat, hogy a műveletet az adott típusú tényezők nem támogatják. Az olyan nem összehasonlító tagfüggvények esetében viszont, amelyeket ki szeretnénk iktatni, `NotImplementedError`, illetve `TypeError` kivételt kell kiváltanunk, ahogy azt korábban az `__add__()` és a `__neg__()` tagfüggvényekkel tettük.

Minden olyan tagfüggvényt egyenként kiiktatni, amelyre nincs szükségünk, igencsak fáradságos munka, noha végül a kívánt eredményre vezet. Az alábbiakban egy ennél hatékonyabb módszert mutatunk be a tagfüggvények kiiktatására (ezt alkalmazzuk a `FuzzyBoolAlt` modulban is), de jobb, ha a következő részre ugrunk, és csak akkor térünk vissza ide, amikor a kiiktatásra a gyakorlatban is szükség lesz.

```
for name, operator in (("__neg__", "-"),
                       ("__index__", "index()")):
    message = ("bad operand type for unary {0}: '{{self}}'"
               .format(operator))
    exec("def {0}(self): raise TypeError(\"{1}\".format("
         "self=self.__class__.__name__))".format(name, message))
```

A beépített `exec()` függvény dinamikusan végrehajtja a paraméterként megadott objektumtól kapott kódot. Esetünkben egy karakterláncot adtunk át a számára, jóllehet más típusú objektumok is szerepelhetnek paraméterként. A rendszer alapértelmezés szerint az utasítást tartalmazó hatókör környezetében futtatja a kódot – esetünkben a `FuzzyBool` osztály meghatározásában –, így a végrehajtott `def` utasítások `FuzzyBool` tagfüggvényeket hoznak létre, ami éppen megfelel az elvárásainknak. A Python csak egyszer hajtja végre a kódot, mégpedig a `FuzzyBoolAlt` modul betöltésekor. Íme az első értékpár, vagyis a `("__neg__", "-")` esetében előállított kód:

```
def __neg__(self):
    raise TypeError("bad operand type for unary -: '{self}'"
                    .format(self=self.__class__.__name__))
```

A kivételt és a hibaüzenetet úgy választottuk meg, hogy megfeleljenek a Python saját típusainál kapottaknak. A kéttényezős tagfüggvények és az n-tényezős függvények (mint a `pow()`) hasonló kódot kapnak, mindössze a hibaüzenetben jelentkezik eltérés. A teljesség kedvéért bemutatjuk az itt alkalmazott kódot:

```
for name, operator in (("__xor__", "^"), ("__ixor__", "^="),
        ("__add__", "+"), ("__iadd__", "+="), ("__radd__", "+"),
        ("__sub__", "-"), ("__isub__", "-="), ("__rsub__", "-"),
        ("__mul__", "*"), ("__imul__", "*="), ("__rmul__", "*"),
        ("__pow__", "**"), ("__ipow__", "**="),
        ("__rpow__", "**"), ("__floordiv__", "//"),
        ("__ifloordiv__", "//="), ("__rfloordiv__", "//"),
        ("__truediv__", "/"), ("__itruediv__", "/="),
        ("__rtruediv__", "/"), ("__divmod__", "divmod()"),
        ("__rdivmod__", "divmod()"), ("__mod__", "%"),
        ("__imod__", "%="), ("__rmod__", "%"),
        ("__lshift__", "<<"), ("__ilshift__", "<<="),
        ("__rlshift__", "<<"), ("__rshift__", ">>"),
        ("__irshift__", ">>="), ("__rrshift__", ">>")):
```

```
        message = ("unsupported operand type(s) for {0}: "
                   "'{{self}}'{{join}} {{args}}".format(operator))
        exec("def {0}(self, *args):\n"
             "    types = [\"'\" + arg.__class__.__name__ + \"'\" "
             "for arg in args]\n"
             "    raise TypeError(\"{1}\".format("
             "self=self.__class__.__name__, "
             "join=(\" and\" if len(args) == 1 else \",\"),"
             "args=\", \".join(types)))".format(name, message))
```

A kód némiképp összetettebb az előzőekben látottaknál, ugyanis a kétténye-zős műveleteknél jelentkező hibaüzenetekben a két típus felsorolásánál a `type1` and `type2` alakot kell használnunk, míg ha három vagy több típus szerepel, a `type1`, `type2`, `type3` felsorolt alakot kell választanunk, ha utánozni szeretnénk a beépített típusok viselkedését. Íme az első értékpár, a `("__xor__", "^")` esetében előállított kód:

```
def __xor__(self, *args):
    types = ["'" + arg.__class__.__name__ + "'" for arg in args]
    raise TypeError("unsupported operand type(s) for ^: "
                    "'{self}'{join} {args}".format(
                    self=self.__class__.__name__,
                    join=(" and" if len(args) == 1 else ","),
                    args=", ".join(types)))
```

A két `for...in` ciklust egyszerűen kivághatjuk és átmásolhatjuk, majd eldönthetjük, hogy mely egy-, két-, illetve n-tényezős műveleteket szeretnénk meghagyni, illetve kiiktatni – eszerint kell eltávolítanunk vagy hozzáadnunk az elemeket az egyes ciklusokhoz.

Ha ezek a kódrészletek is a helyükre kerültek, az `f` és g `FuzzyBool` objektumokkal az `f + g` műveletet végezve egy `TypeError` kivételt kapunk az „unsupported operand type(s) for +: 'FuzzyBool' and 'FuzzyBool'" hibaüzenettel – ez éppen a kívánt viselkedés.

Az osztályok létrehozásánál a leggyakrabban úgy járunk el, mint a `FuzzyBool` első megvalósításánál – ez a módszer az esetek túlnyomó többségében tökéletesen megfelel. Ha azonban állandó értékű osztályra van szükségünk, akkor valamelyik beépített állandó értékű típusból érdemes kiindulnunk (`float`, `int`, `str` vagy `tuple`), és újra meg kell valósítanunk az `object.__new__()` tagfüggvényt, valamint meg kell írnunk a kívánt viselkedéshez szükséges további tagfüggényeket. Ennek a módszernek az a hátránya, hogy egyes tagfüggvényeket ki kell iktatnunk – ez tönkreteszi a többalakúságot, így a legtöbb esetben az összetétel módszere, amelyet a `FuzzyBool` első megvalósításánál használtunk, kedvezőbb megoldást ad.

Egyedi gyűjteményosztályok

A következőkben olyan osztályokkal foglalkozunk, amelyek nagy adatmennyiségek tárolására alkalmasak. Az elsőként bemutatott `Image` nevű osztály képek adatait képes megőrizni, és jól példázza az egyedi adattároló osztályok működését, hiszen amellett, hogy az adatait elérhetővé teszi a memóriában, rendelkezik olyan tagfüggvényekkel is, amelyek biztosítják a lemezre mentés és a betöltés lehetőségét. Ezt követően megismerkedünk a `SortedList` és a `SortedDict` osztályokkal, amelyekkel a Python alapkönyvtárának meglepő hiányosságát pótoljuk – lehetővé tesszük ugyanis beépített rendezéssel bíró gyűjteményes adattípusok használatát.

Gyűjteményeket egyesítő osztályok létrehozása

A síkbeli színes képek megjelenítésére kínálkozik egy egyszerű módszer, ahol egy kétdimenziós tömböt használunk, amelynek minden eleme egy szín. Így ha egy 100x100 képpontos képet szeretnénk ábrázolni, 10 000 szín tárolására van szükség. Az `Image` osztályban (amelyet az `Image.py` fájlban találhatunk meg) ennél hatékonyabb módszert alkalmazunk – az osztály ugyanis tárol egy háttérszínt, valamint azokat a pontokat, amelyeknek a színe eltér ettől a háttérszíntől. Mindehhez egy szótárt használunk, egyfajta ritka tömbként, amelyben a kulcsok az `(x, y)` koordinátapárok, a hozzájuk tartozó értékek pedig a megfelelő színek. Így ha egy 100x100-as képet szeretnénk tárolni, amelyben a pontok felerészben háttérszínűek, mindössze 5000+1 színt kell tárolnunk, ami jelentős memóriamegtakarítást jelent.

Az `Image.py` modul szerkezete ismerős módon épül fel: az elején egy shebang-sor áll, ezt követik a szerzői jogi adatok megjegyzések formájában, majd a modul dokumentációs karakterlánca néhány teszttel, ezután pedig betöltjük az `os` és a `pickle` modulokat. A képek betöltésének és mentésének tárgyalásakor ez utóbbiról ejtünk majd pár szót. A betöltés végeztével létrehozunk néhány egyedi kivételosztályt:

Karakterkódolások
☞ 297, 317

```
class ImageError(Exception): pass
class CoordinateError(ImageError): pass
```

Mindössze az első két kivételosztályt mutattuk be; a többi (`LoadError`, `SaveError`, `ExportError` és `NoFilenameError`) létrehozása hasonló módon történik, és mindegyikük az `ImageError` leszármazottja. Az `Image` osztály felhasználói e kivételek bármelyikét vizsgálhatják, de megelégedhetnek az `ImageError` alaposztály ellenőrzésével is.

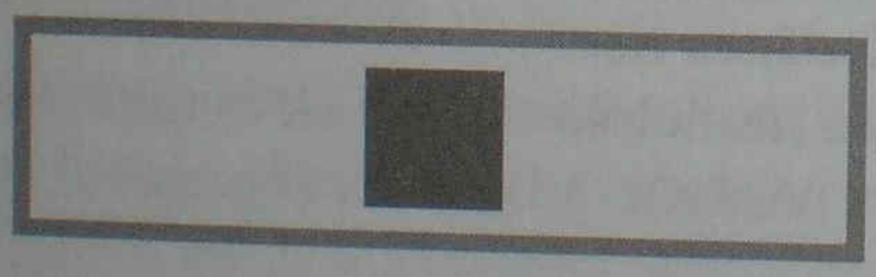
A modul további részében az `Image` osztály kódja található, valamint három szabványos sor a fájl végén a modul tesztjeinek futtatására. Mielőtt az osztály szerkezetének boncolgatásába fognánk, lássuk, miként vehetjük hasznát:

```
border_color = "#FF0000"     # vörös
square_color = "#0000FF"     # kék
width, height = 240, 60
midx, midy = width // 2, height // 2
```

```
image = Image.Image(width, height, "square_eye.img")
for x in range(width):
    for y in range(height):
        if x < 5 or x >= width - 5 or y < 5 or y >= height - 5:
            image[x, y] = border_color
        elif midx - 20 < x < midx + 20 and midy - 20 < y < midy + 20:
            image[x, y] = square_color
image.save()
image.export("square_eye.xpm")
```

Vegyük észre, hogy a kép színeinek beállítására használhatjuk az elemelérési műveletet (`[]`). A szögletes zárójeleket alkalmazhatjuk a pontok kiolvasására és törlésére is (ez utóbbi esetben háttérszínűre változtatjuk a kívánt pontot). A koordinátákat egyetlen rendezett sorozat objektumként adhatjuk át (hála a vesszőnek) – az eredmény ugyanaz, mintha az `image[(x, y)]` alakot használnánk. A használati alakok ilyen egységét a Pythonban könnyen elérhetjük – mindössze a megfelelő különleges tagfüggvényeket kell megvalósítanunk. Az elemelérési művelet esetében itt a `__getitem__()`, a `__setitem__()` és a `__delitem__()` tagfüggvényekről van szó.

Az `Image` osztály HTML-stílusú hexadecimális karakterláncokkal írja le a színeket. A háttérszínt az objektum létrehozásánál kell megadnunk, egyébként az alapértelmezett fehér színt veszi fel. Az osztály saját formátumában menti és tölti be a képeket, de lehetőséget kapunk a képek `.xpm` formátumú kivitelére is, amelyet számos képfeldolgozó alkalmazás ismer. A fenti kódrészletben előállított `.xpm` képet a 6.6. ábrán láthatjuk.



6.6. ábra

A square_eye.xpm kép

A következőkben megismerkedünk az `Image` osztály tagfüggvényeivel – elsőként nézzük a `class` sort és az előkészítést:

```
class Image:
    def __init__(self, width, height, filename="",
                 background="#FFFFFF"):
        self.filename = filename
        self.__background = background
        self.__data = {}
        self.__width = width
        self.__height = height
        self.__colors = {self.__background}
```

Ha létrehozunk egy `Image` objektumot, meg kell határoznunk a szélességét és magasságát, a fájlnevet és a háttérszínt azonban nem feltétlenül, hiszen az osztály itt alapértelmezett értékeket biztosít. A `self.__data` osztály kulcsai `(x, y)` koordináták, értékei pedig színkarakterláncok. A `self.__colors` halmazba kezdetben a háttérszín kerül, a továbbiakban pedig ez tartja nyilván, hogy mely színek szerepelnek a képen.

A fájlnév kivételével az összes adatjellemző privát, így valamilyen formában elérhetővé kell tennünk őket a felhasználók számára. Éppen erre szolgálnak a tulajdonságok.*

```
@property
def background(self):
    return self.__background

@property
def width(self):
    return self.__width

@property
def height(self):
    return self.__height

@property
def colors(self):
    return set(self.__colors)
```

Gyűjtemények másolása 132, 142 ☜

Ha egy adatjellemzőt adunk vissza egy objektumból, fontos tisztában lennünk azzal, hogy változtatható vagy állandó értékű jellemzőről van-e szó. Az utóbbi esetében az érték visszaadása mindig biztonságos, hiszen nem változtatható meg, a változtatható jellemzőknél azonban el kell gondolkodnunk, hogy mit is tegyünk. Ha közvetlenül a változtatható jellemző egy hivatkozását adjuk vissza, igen gyors és hatékony eredményt érhetünk el, hiszen nincs szükség másolásra, de ezzel a hívó hozzáfér az objektum belső állapotához, és ennek módosításával akár érvénytelenné is teheti az objektumot. Érdemesebb a változtatható adattípusok esetében csak egy másolatot visszaadnunk, hacsak a vizsgálatok ki nem mutatják, hogy ez rossz hatással van a program teljesítményére. (Esetünkben az egyedi színek tárolása helyett egy szín kiolvasásakor visszaadhatnánk a `set(self.__data.values()) | {self.__background}` értéket is.)

```
def __getitem__(self, coordinate):
    assert len(coordinate) == 2, "coordinate should be a 2-tuple"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    return self.__data.get(tuple(coordinate), self.__background)
```

* A 8. fejezetben a jellemzők elérésének egy teljesen eltérő módszerét mutatjuk majd be, amelyben a különleges tagfüggvények, mint a `__getattr__()` és a `__setattr__()` kapnak szerepet – bizonyos körülmények között ez a megoldás a célravezető.

Ez a tagfüggvény az elemelérési művelet ([]) segítségével olvassa ki az adott koordinátákkal meghatározott pont színét. Az elemelérési művelet, valamint néhány, a gyűjteményeknél használatos különleges tagfüggvény leírását megtalálhatjuk a 6.4. táblázatban.

Az elemek elérésénél két szabályt alkalmaztunk. Az első egy előfeltétel, amely szerint a koordináták megadásánál kételemű sorozatokat (jobbára rendezett párokat) kell használnunk – ezt egy feltételezéssel biztosítjuk. A második szabály alapján tetszőleges koordinátaértéket elfogadunk, de ha bármelyik koordináta kívül esik a kép tartományán, egyedi kivételt váltunk ki.

Egy adott koordinátával meghatározott képpont színének kiolvasásához a `dict.get()` tagfüggvényt használjuk, amelynek a háttérszínt adjuk meg alapértelmezett értékként. Ezzel biztosítjuk, hogy ha nem állítottuk be a pont színét, akkor visszakapjuk a háttérszínt – az egyébként várható `KeyError` kivétel helyett.

```
def __setitem__(self, coordinate, color):
assert len(coordinate) == 2, "coordinate should be a 2-tuple"
```

6.4. táblázat *A gyűjtemények különleges tagfüggvényei*

Különleges tagfüggvény	Használat	Leírás
`__contains__(self, x)`	`x in y`	Az eredmény `True`, ha az `x` szerepel az `y` sorozatban, vagy kulcs az `y` megfeleltetésben.
`__delitem__(self, k)`	`del y[k]`	Törli az `y` sorozat `k`-adik elemét, illetve az `y` megfeleltetés `k` kulcsához tartozó elemét.
`__getitem__(self, k)`	`y[k]`	Visszaadja az `y` sorozat `k`-adik elemét, illetve az `y` megfeleltetés `k` kulcsához tartozó elemét.
`__iter__(self)`	`for x in y: pass`	Visszaad egy bejárót az `y` sorozat elemeihez, illetve az `y` megfeleltetés kulcsaihoz.
`__len__(self)`	`len(y)`	Visszaadja az `y` elemeinek számát.
`__reversed__(self)`	`reversed(y)`	Visszaad egy hátrafelé haladó bejárót az `y` sorozat elemeihez, illetve az `y` megfeleltetés kulcsaihoz.
`__setitem__(self, k, v)`	`y[k] = v`	Az `y` sorozat `k`-adik elemét, illetve az `y` megfeleltetés `k` kulcsához tartozó elemét `v` értékre állítja.

```
if (not (0 <= coordinate[0] < self.width) or
    not (0 <= coordinate[1] < self.height)):
    raise CoordinateError(str(coordinate))
```

```
        if color == self.__background:
            self.__data.pop(tuple(coordinate), None)
        else:
            self.__data[tuple(coordinate)] = color
            self.__colors.add(color)
```

Ha a felhasználó egy koordináta értékét a háttérszínre állítja, egyszerűen törölhetjük a szótár megfelelő elemét, hiszen ha egy koordináta nincs a szótárban, az osztály a megfelelő pontot automatikusan háttérszínűnek tekinti. A `dict.pop()` tagfüggvényt alkalmaztuk egy szabadon választott második paraméterrel, ha ugyanis a `del`-t használnánk, a rendszer `KeyError` kivételt váltana ki, ha a kulcs (vagyis a koordináta) nem szerepel a szótárban.

Ha a szín eltér a háttérszíntől, tároljuk a megfelelő koordinátán, továbbá elhelyezzük a felhasznált színek halmazában.

```
    def __delitem__(self, coordinate):
        assert len(coordinate) == 2, "coordinate should be a 2-tuple"
        if (not (0 <= coordinate[0] < self.width) or
            not (0 <= coordinate[1] < self.height)):
            raise CoordinateError(str(coordinate))
        self.__data.pop(tuple(coordinate), None)
```

Amennyiben törölni szeretnénk egy koordináta színét, valójában háttérszínűvé kell alakítanunk. Ezúttal is a `dict.pop()` tagfüggvényt használjuk, hiszen ez mindig a megfelelő eredményt adja, akár a szótárban van a megadott koordináta, akár nem.

A `__len__()` számára nem készítettünk megvalósítást, hiszen e tagfüggvény használatának nem lenne sok értelme egy kétdimenziós objektumnál. Emellett ábrázolási alakot sem adhatunk meg, hiszen egy teljes `Image` objektumot nem lehet létrehozni az `Image()` hívással, így hát nincs szükség a `__repr__()` (vagy az `__str__()`) tagfüggvényre. Ha egy felhasználó mégiscsak meghívja a `repr()` vagy az `str()` függvényt egy `Image` objektumra, az alapértelmezett osztály `object.__repr__()` tagfüggvénye visszaadja a megfelelő karakterláncot, a következő alakban: `'<Image.Image object at 0x9c794ac>'`. Ez a formátum szabványos az `eval()` függvénnyel nem kiértékelhető objektumok körében. A megadott hexadecimális szám az objektum azonosítója – ez egyedi (többnyire az objektum címe a memóriában), de változhat.

Szeretnénk az `Image` osztály felhasználóinak lehetővé tenni, hogy lemezre menthessék, valamint betölthessék az adataikat – ennek érdekében készítettük el a `save()` és a `load()` tagfüggvényeket.

A tárolás megvalósítására az adatok *befőzése* (pickling) mellett döntöttünk. A Python nyelvezetében a befőzés a sorosítás (serialization) bizonyos fajtájára utal (amelynek során az adatokat egyszerű bájtsorozattá vagy karakterlánccá alakítjuk). A módszer jelentősége

abban rejlik, hogy a befőzött objektum származhat gyűjteményes adattípusból, így lehet lista vagy szótár, és tartalmazhat más objektumokat is (más gyűjteményeket, gyűjtemények gyűjteményeit, és így tovább) – ráadásul a rendszer arra is ügyel, hogy a többször előforduló objektumokat csak egyszer tárolja.

A befőttet (pickle) egyszerűen beolvashatjuk egy Python-változóba – nem kell értelmeznünk vagy átalakítanunk a kapott adatokat. Így a módszer különösen jó szolgálatot tesz ötletszerűen létrehozott adatgyűjtemények tárolására, különösen kisebb, illetve személyes használatra szánt programokban.

Figyelem! *Mindazonáltal a befőttek nem rendelkeznek semmiféle biztonsági eljárással (titkosítással vagy digitális aláírással), így a megbízhatatlan forrásból származó befőttek felbontása bizony veszélyekkel járhat. Ennek ismeretében a nem kifejezetten személyes használatra szánt programok esetében jobb, ha kifejezetten a programhoz készített egyedi fájlformátumot alkalmazunk. A 7. fejezetben megtanuljuk, miként hozhatunk létre saját bináris, szöveges és XML-fájlformátumokat.*

```
def save(self, filename=None):
    if filename is not None:
        self.filename = filename
    if not self.filename:
        raise NoFilenameError()
    fh = None
    try:
        data = [self.width, self.height, self.__background,
                self.__data]
        fh = open(self.filename, "wb")
        pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)
    except (EnvironmentError, pickle.PicklingError) as err:
        raise SaveError(str(err))
    finally:
        if fh is not None:
            fh.close()
```

A függvény első része teljes egészében a fájlnévvel foglalkozik. Amennyiben az `Image` objektumot fájlnév megadása nélkül hoztuk létre, és azóta sem határoztunk meg ilyet, a `save()` tagfüggvénynek mindenképpen át kell adnunk egy fájlnevet (ez esetben a tagfüggvény a „mentés másként” lehetőséget adja, ahol meg kell adnunk a programban a továbbiakban használt fájlnevet). Ha nem adunk át fájlnevet, a program az aktuális fájlnevet használja – ha nincs ilyen, és nem is adjuk meg, kivételt kapunk.

Létrehozunk egy `data` nevű listát – ez tárolja majd azokat az objektumokat, amelyeket menteni szeretnénk. Ide tartozik a koordinátákhoz tartozó színeket tartalmazó `self.__data` szótár, a felhasznált színek halmaza azonban kimarad, hiszen ez előállítható a szótár alapján. Ezek után már csak meg kell nyitnunk a fájlt a bináris módú íráshoz, és a `pickle.dump()` függvénnyel tárolnunk az adatobjektumot. Ennyi az egész!

A `pickle` modul különböző formátumokban képes sorosítani az adatainkat (ezeket a formátumokat a Python leírása *protokolloknak* nevezi) – közülük a `pickle.dump()` harmadik paraméterének megadásával választhatunk. A 0-s protokoll ASCII-adatokat tárol, és a hibakeresésben vehetjük nagy hasznát. Esetünkben a 3-as protokoll (`pickle.HIGHEST_PROTOCOL`) mellett döntöttünk, amely tömör, bináris formátumot biztosít – ez indokolja, hogy miért kellett bináris módban megnyitnunk a fájlt. A befőttek olvasásánál nem kell nyilatkoznunk a protokollról, ugyanis a `pickle.load()` függvény képes kikövetkeztetni, hogy melyiket használtuk.

```
def load(self, filename=None):
    if filename is not None:
        self.filename = filename
    if not self.filename:
        raise NoFilenameError()
    fh = None
    try:
        fh = open(self.filename, "rb")
        data = pickle.load(fh)
        (self.__width, self.__height, self.__background,
         self.__data) = data
        self.__colors = (set(self.__data.values()) |
                         {self.__background})
    except (EnvironmentError, pickle.UnpicklingError) as err:
        raise LoadError(str(err))
    finally:
        if fh is not None:
            fh.close()
```

A fenti függvény ugyanúgy kezdődik, mint a `save()`, így hozzájut a betöltendő fájl nevéhez. A fájlt bináris olvasási módban kell megnyitnunk, az adatokat pedig egyetlen utasítással olvassuk be, méghozzá ilyen egyszerűen: `data = pickle.load(fh)`. Az adatobjektum a mentett adatok pontos mását adja vissza, így egy listát kapunk, amelyben szerepel a szélesség és a magasság egész értékek formájában, a háttérszín karakterlánca, valamint a szótár a koordináta–szín elemekkel. A rendezett sorozatok kicsomagoló műveletével a lista elemeit hozzárendeljük a megfelelő változókhoz, így a bennük tárolt korábbi képadatok (érthető módon) elvesznek.

A felhasznált színek halmazát visszaállítjuk a szótár tartalma alapján, kiegészítve a háttérszínnel.

```
def export(self, filename):
    if filename.lower().endswith(".xpm"):
        self.__export_xpm(filename)
    else:
        raise ExportError("unsupported export format: " +
                          os.path.splitext(filename)[1])
```

Egyetlen általános kiviteli tagfüggvényt készítettünk, amely a fájl kiterjesztése alapján választja ki, hogy melyik privát tagfüggvény lehetőségeit vegye igénybe – ha pedig az adott fájlformátum esetében nem képes a kivitelre, kivételt vált ki. Esetünkben csak az `.xpm` fájlok kivitelét támogatjuk (azt is csak akkor, ha a kép 8390-nél kevesebb színt használ). Az `__export_xpm()` függvény kódját nem közöltük, mivel nem kapcsolódik szorosan fejezetünk anyagához, de a mellékelt forráskódban természetesen megtalálhatjuk.

Ezzel végére értünk az `Image` osztály megvalósításának, amellyel számos jellemzően előforduló feladatot oldottunk meg – így tároltunk a programra jellemző adatokat, elérhetővé tettük annak egyes elemeit, gondoskodtunk a lemezre mentés és a betöltés lehetőségéről, mindemellett a kódban kizárólag a feltétlenül szükséges tagfüggvények megvalósítására szorítkoztunk. A következőkben két általános egyedi gyűjteményes adattípust mutatunk be, amelyek teljes API-t tesznek elérhetővé a felhasználóik számára.

Gyűjteményosztályok létrehozása összetétellel

A következőkben összeállítunk egy `SortedList` nevezetű teljes egyedi gyűjteményosztályt, amely elemek rendezett listájának tárolására képes. A rendezéshez a < műveletet használjuk, amelyet az `__lt__()` különleges tagfüggvény valósít meg, illetve egy kulcsfüggvény, amennyiben megadunk ilyet. Az osztállyal kísérletet teszünk arra, hogy megfeleljünk a `list` osztály API-jának a könnyebb megértés és használat érdekében, de egyes tagfüggvényeket kénytelenek leszünk kihagyni – így például az összefűző műveletet (+), amelynek a használata hibás sorrendet eredményezhetne.

Az egyedi osztályok létrehozásánál megszokott módon választhatunk, hogy az osztályt egy másik hasonlóból származtatjuk, vagy teljesen új osztályt hozunk létre, és a működéséhez szükséges további osztályok példányait a belsejében helyezzük el. A most bemutatott `SortedList` osztály esetében tisztán ez utóbbi módszert alkalmazzuk, míg a `SortedDict` felépítésében egyszerre vesz részt az öröklés és az összetétel.

A 8. fejezetben láthatjuk majd, hogy az osztályok ígéreteket tesznek az általuk biztosított API-kkal kapcsolatban. Így például a `list` osztály a `MutableSequence` API-t biztosítja, így tudhatjuk, hogy támogatja az `in` műveletet, az `iter()` és a `len()` beépített függvényeket, az elemelérési műveletet (`[]`) az elemek kiolvasásához, beállításához és törléséhez, valamint az `insert()` tagfüggvényt. `SortedList` osztályunk nem teszi lehetővé az elemek beállítását, és nem rendelkezik `insert()` tagfüggvénnyel, így nem támogatja a `MutableSequence` API-t. Ha a `SortedList` osztályt a `list` leszármazottjaként hoznánk létre, az eredményként kapott osztály változtatható sorozatként mutatná magát, mindemellett azonban nem valósítaná meg a teljes API-t. Esetünkben a `SortedList` nem a `list` leszármazottja, és így semmiféle ígéretet nem tesz az API-jával kapcsolatban. Másrészről azonban, a későbbiekben bemutatott `SortedDict` osztály a `dict` osztályban elérhető teljes `MutableMapping` API-t megvalósítja, így létrehozhatjuk a `dict` alosztályaként.

Lássunk most néhány példát a `SortedList` használatára:

```
letters = SortedList.SortedList(("H", "c", "B", "G", "e"), str.lower)
# str(letters) == "['B', 'c', 'e', 'G', 'H']"
letters.add("G")
letters.add("f")
letters.add("A")
# str(letters) == "['A', 'B', 'c', 'e', 'f', 'G', 'G', 'H']"
letters[2] # returns: 'c'
```

A `SortedList` *összetétel* (aggregate) két privát jellemzőt tartalmaz (vagyis ezekből épül fel) – egy `self.__key()` nevezetű függvényt (amelyet a `self.__key` objektumhivatkozás alakjában tárol), valamint egy `self.__list` nevű listát.

Lambdafüggvények 167, 178 ☜

A kulcsfüggvényt a második paraméterben adjuk át (illetve a `key` kulcsszó-paraméterrel, ha nem adtunk meg kezdeti sorozatot). Amennyiben nem határozunk meg kulcsfüggvényt, az alábbi privát modulfüggvényt használjuk:

```
_identity = lambda x: x
```

Ez maga az identitásfüggvény: Egyszerűen változatlanul visszaadja a paraméterét, így ha a `SortedList` kulcsfüggvényeként alkalmazzuk, a lista objektumainak rendezési kulcsai önmaguk lesznek.

A `SortedList` típus nem teszi lehetővé az elemelérési műveletnek (`[]`), hogy megváltoztassa az elemeket (tehát nem valósítja meg a `__setitem__()` különleges tagfüggvényt), továbbá nem támogatja az `append()` és az `extend()` tagfüggvényeket, hiszen ezek érvényteleníthetnék a sorrendet. Elemek elhelyezésére kizárólag a `SortedList` objektum létrehozásakor átadott sorozattal, valamint a `SortedList.add()` tagfüggvénnyel van lehetőségünk. Másrészről az elemelérési műveletet nyugodtan használhatjuk elemek kiolvasására és törlésére, hiszen ez nem befolyásolja a sorrendet – megvalósítjuk tehát a `__getitem__()` és a `__delitem__()` különleges tagfüggvényeket.

A következőkben tagfüggvényenként elemezzük az osztály szerkezetét; elsőként szokás szerint a `class` sort és a kezdeti értékadást mutatjuk be:

```
class SortedList:
    def __init__(self, sequence=None, key=None):
        self.__key = key or _identity
        assert hasattr(self.__key, "__call__")
        if sequence is None:
            self.__list = []
        elif (isinstance(sequence, SortedList) and
              sequence.key == self.__key):
            self.__list = sequence.__list[:]
        else:
            self.__list = sorted(list(sequence), key=self.__key)
```

Mivel a függvény neve valójában egy objektumhivatkozás (amely magára a függvényre mutat), így a függvényeket ugyanúgy tárolhatjuk változókban, mint bármely más objektumhivatkozást. Esetünkben a `self.__key` változó egy hivatkozást tárol az átadott kulcsfüggvényre, illetve az identitásfüggvényre. A tagfüggvény első utasítása arra épít, hogy az `or` művelet első tényezőjét adja vissza, amennyiben annak értéke logikai környezetben `True` (márpedig, ha a kulcsfüggvény nem `None`, ez lesz az eredmény), egyébként pedig a második tényezővel tér vissza. Ugyanezt az eredményt kapjuk a kissé hosszabb, ám könnyebben érthető `self.__key = key if key is not None else _identity` utasítással is.

Ha rendelkezünk kulcsfüggvénnyel, egy feltételezéssel biztosítjuk, hogy hívható legyen. A beépített `hasattr()` függvény a `True` értéket adja vissza, ha az első paraméterében átadott objektum rendelkezik a második paraméterben meghatározott nevű jellemzővel. Érdemes tudnunk az ide tartozó `setattr()` és `delattr()` függvényekről is, amelyek tárgyalására a 8. fejezetben kerül sor. Minden hívható objektum – így a függvények és a tagfüggvények is – rendelkezik a `__call__` jellemzővel.

Ahhoz, hogy a `SortedList` objektumok létrehozását a `list` objektumokéhoz hasonlóvá tegyük, bevezetünk egy nem kötelezően megadandó `sequence` nevű paramétert, amely megfelel a `list()` egyetlen, nem kötelező paraméterének. A `SortedList` egy `list` gyűjteményt tárol a `self.__list` privát változóban, és ennek rendezését a megadott kulcsfüggvénnyel tartja fenn.

Az `elif` utasításban típusvizsgálattal állapítjuk meg, hogy az adott sorozat a `SortedList` egy példánya-e, és ha igen, vajon ugyanazzal a kulcsfüggvénnyel rendelkezik-e, mint az aktuális rendezett lista. Ha mindkét feltétel teljesül, egyszerűen sekély másolatot készítünk a sorozat listájából – rendezésre nincs szükség. Amennyiben a legtöbb kulcsfüggvényt helyben, a `lambda` utasítással hozzuk létre, még akkor is eltérőnek mutatkoznak, ha a kódjuk megegyezik, így a várt hatékonyságnövekedés elmaradhat.

```
@property
def key(self):
    return self.__key
```

Ha elkészült a rendezett lista, a kulcsfüggvényét rögzítjük, így privát változóként tároljuk, hogy megakadályozzuk a módosítását a felhasználók számára. Egyeseknek azonban szükségük lehet a kulcsfüggvény hivatkozására (amint azt hamarosan láthatjuk), így elérhetővé tettük a csak olvasható `key` tulajdonság révén.

```
def add(self, value):
    index = self.__bisect_left(value)
    if index == len(self.__list):
        self.__list.append(value)
    else:
        self.__list.insert(index, value)
```

Amennyiben a felhasználó ezt a tagfüggvényt hívja meg, a megadott értéket a `self.__list` privát lista megfelelő helyére kell beillesztenünk, hogy megtartsuk a sorrendet. A kívánt sorszámot a `SortedList.__bisect_left()` tagfüggvény adja meg, amint azt hamarosan láthatjuk. Ha az új érték nagyobb, mint a lista összes eleme, a lista végére kerül, így a sorszám megegyezik a lista hosszával (a sorszámok `0`-tól `len(L) - 1`-ig terjednek) – ha így áll a helyzet, egyszerűen hozzáfűzzük a listához az új értéket. Egyébként pedig a megadott sorszámon illesztjük be az értéket – ami 0 lesz, ha az új érték kisebb a lista összes eleménél.

```
def __bisect_left(self, value):
    key = self.__key(value)
    left, right = 0, len(self.__list)
    while left < right:
        middle = (left + right) // 2
        if self.__key(self.__list[middle]) < key:
            left = middle + 1
        else:
            right = middle
    return left
```

A fenti privát tagfüggvény megadja azt a sorszámot, amely a listában az adott értékhez tartozik, kiszámít egy összehasonlító kulcsot az adott érték számára a rendezett lista kulcsfüggvényének használatával, és összehasonlítja ezt a tagfüggvény által vizsgált elemek számított összehasonlítási kulcsaival. Az algoritmus *bináris keresést* alkalmaz, amely igen nagy listák esetében is hatékonyan működik – olyannyira, hogy ha például egy 1 000 000 elemből álló listában keressük egy érték helyét, mindössze 21 összehasonlítást kell végeznünk.* Érdemes mindezt összevetnünk az egyszerű, rendezetlen `list` objektum esetével, amely lineáris keresést alkalmaz, és ugyanehhez a feladathoz átlagosan 500 000, legrosszabb esetben pedig 1 000 000 összehasonlításra van szüksége.

```
def remove(self, value):
    index = self.__bisect_left(value)
    if index < len(self.__list) and self.__list[index] == value:
        del self.__list[index]
    else:
        raise ValueError("{0}.remove(x): x not in list".format(
                         self.__class__.__name__))
```

Ezzel a tagfüggvénnyel az adott érték első előfordulását távolítjuk el.
A `SortedList.__bisect_left()` tagfüggvénnyel megkeressük azt a sorszámot, ahová az érték tartozik, majd megvizsgáljuk, hogy ez a sorszám létezik-e a listában, és az e sorszámon álló érték megegyezik-e a megadottal. Ha az elem megfelel a feltételeknek, eltávolítjuk, egyébként pedig `ValueError` kivételt váltunk ki (a `list.remove()` viselkedése ilyen körülmények között pontosan így alakul).

---

* A Python `bisect` moduljában elérhetjük a `bisect.bisect_left()` függvényt és egyéb társait, de könyvünk írásának idején a modul egyetlen függvénye sem volt képes kulcsfüggvényeket kezelni.

```
    def remove_every(self, value):
        count = 0
        index = self.__bisect_left(value)
        while (index < len(self.__list) and
               self.__list[index] == value):
            del self.__list[index]
            count += 1
        return count
```

A fenti tagfüggvény hasonlít a `SortedList.remove()`-hoz – valójában a listák API-jának bővítése. Először is megkeresi azt a sorszámot, ahol az érték először megjelenik a listában, majd addig halad a lista elemein, amíg azok értéke megegyezik a megadottal. A kódban van egy apró csavar, ugyanis a ciklus minden végrehajtásakor törli a megfelelő elemet, így az aktuális sorszám mindig a következő elemre utal.

```
    def count(self, value):
        count = 0
        index = self.__bisect_left(value)
        while (index < len(self.__list) and
               self.__list[index] == value):
            index += 1
            count += 1
        return count
```

A tagfüggvény visszatérési értéke az átadott érték listabeli előfordulásainak száma (ez lehet 0 is). Működése a `SortedList.remove_every()` tagfüggvényben látott cikluson alapul, de ezúttal nem végzünk törlést, és minden ciklusban növeljük az aktuális sorszámot.

```
    def index(self, value):
        index = self.__bisect_left(value)
        if index < len(self.__list) and self.__list[index] == value:
            return index
        raise ValueError("{0}.index(x): x not in list".format(
                         self.__class__.__name__))
```

Mivel a `SortedList` rendezett listát tárol, az érték után gyors bináris kereséssel kutathatunk.

```
    def __delitem__(self, index):
        del self.__list[index]
```

A `__delitem__()` különleges tagfüggvény lehetővé teszi a `del L[n]` alak használatát, ahol *L* egy rendezett lista, *n* pedig egy egész értékű sorszám. A tartományon kívül eső sorszámok esetét nem vizsgáljuk, hiszen ilyen esetben a `self.__list[index]` hívás `IndexError` kivételt vált ki, és éppen ez az, amit vártunk.

```
    def __getitem__(self, index):
        return self.__list[index]
```

Ez a tagfüggvény támogatja az `x = L[n]` alak használatát, ahol *L* egy rendezett lista, *n* pedig egy egész értékű sorszám.

```
def __setitem__(self, index, value):
    raise TypeError("use add() to insert a value and rely on "
                    "the list to put it in the right place")
```

Nem szeretnénk, ha a felhasználó megváltoztathatná egy adott sorszámú elem értékét (vagyis kizárjuk az `L[n] = x` alakot), hiszen így a lista rendezettsége megszűnhetne. A nem támogatott műveleteket a `TypeError` kivétellel jelezhetjük.

```
def __iter__(self):
    return iter(self.__list)
```

A tagfüggvény megvalósítása egyszerű, hiszen a privát lista számára könnyedén visszaadhatunk egy bejárót a beépített `iter()` függvénnyel. Ez a tagfüggvény támogatja az *érték* `in` *bejárható_objektum* alak használatát.

Fontos tudnunk, hogy ha egy sorozatra van szükség, az osztály ezt a tagfüggvényt alkalmazza. Ha tehát egy `L` nevű `SortedList` objektumot listává szeretnénk alakítani, a `list(L)` hívást használjuk, amelynek során a Python a háttérben a `SortedList.__iter__(L)` hívással állítja elő azt a sorozatot, amelyre a `list()` függvénynek szüksége van.

```
def __reversed__(self):
    return reversed(self.__list)
```

Mindezzel a beépített `reversed()` függvényt támogatjuk, így alkalmazhatjuk az *érték* `in` `reversed (`*bejárható_objektum*`)` alakot.

```
def __contains__(self, value):
    index = self.__bisect_left(value)
    return (index < len(self.__list) and
            self.__list[index] == value)
```

A `__contains__()` tagfüggvény az `in` művelet működését teszi lehetővé. Itt ismét alkalmazhatjuk a gyors bináris keresést, szemben az egyszerű `list` típusnál használt, jóval lassabb lineáris változattal.

```
def clear(self):
    self.__list = []
def pop(self, index=-1):
    return self.__list.pop(index)
def __len__(self):
    return len(self.__list)
def __str__(self):
    return str(self.__list)
```

A `SortedList.clear()` tagfüggvény elveti a már meglevő listát, és egy új, üres példánnyal helyettesíti. A `SortedList.pop()` tagfüggvény eltávolítja és visszaadja az adott sorszámon található elemet, illetve, ha a sorszám a tartományon kívül esik, kiváltja az `IndexError` kivételt. A `pop()`, a `__len__()`, valamint az `__str__()` tagfüggvények esetében a munkát egyszerűen átháríjuk az összetétellel létrehozott `self.__list` objektumra.

A `__repr__()` különleges tagfüggvényt nem valósítjuk meg újra, így az alaposztály `object.__repr__()` tagfüggvénye lép működésbe, ha a felhasználó a `repr(L)` utasítást alkalmazza egy `L` nevű `SortedList` objektumra. Az eredmény így egy `'<SortedList.SortedList object at 0x97e7cec>'` alakú karakterlánc lesz, amelyben a hexadecimális azonosító természetesen változhat. A `__repr__()` számára képtelenek vagyunk értelmes megvalósítást készíteni, hiszen ehhez szükség lenne a kulcsfüggvényre, és egy függvényobjektum hivatkozását nem tudjuk az `eval()`-lal kiértékelhető karakterlánc alakjában ábrázolni.

Az `insert()`, a `reverse()` és a `sort()` tagfüggvényeket nem valósítottuk meg, mivel a használatuknak nem lenne értelme e típus esetén. Bármelyiküket is hívjuk meg, `AttributeError` kivételhez jutunk.

Ha egy rendezett listát az `L[:]` alakkal másolunk, egyszerű `list` objektumhoz jutunk a `SortedList` helyett. Ha valóban másolni szeretnénk, a legegyszerűbb, ha betöltjük a `copy` modult, és a `copy.copy()` függvényt alkalmazzuk – ez képes a rendezett listánkat (és általában a legtöbb egyedi osztályt) bármiféle külső segítség nélkül lemásolni. Mindennek ellenére úgy döntöttünk, hogy saját `copy()` tagfüggvényt írunk:

```
def copy(self):
    return SortedList(self, self.__key)
```

Az első paraméterként átadott `self` objektummal biztosítjuk, hogy a rendszer sekély másolatot készítsen a `self.__list` objektumból, és ne rendezze a másolás után. (Mindez az `__init__()` tagfüggvény `elif` utasításánál található típusvizsgálatnak köszönhető.) Ez a módszer elméletben hatékonyságnövekedést ígér, de ez jelen pillanatban nem érhető el a `copy.copy()` függvény számára – szerencsére a helyzeten egyetlen sorral változtathatunk:

```
__copy__ = copy
```

A `copy.copy()` hívásakor a függvény megpróbálja használatba venni az objektum `__copy__()` különleges tagfüggvényét, ha pedig nem talál ilyet, a saját kódját alkalmazza. A fenti sor használata esetén a függvény a `SortedList.copy()` tagfüggvényt veszi igénybe a rendezett listák másolásánál. (Készíthetünk egy `__deepcopy__()` nevű különleges tagfüggvényt is, de ez már több erőfeszítést igényel – a részletekről a `copy` modul leírásából tájékozódhatunk.)

Ezzel a `SortedList` osztály megvalósításának végére értünk. A következőkben az itt elért eket arra használjuk, hogy a `SortedDict` osztály kulcsait egy rendezett listában tároljuk.

Gyűjteményosztályok létrehozása öröklléssel

A következőkben bemutatott `SortedDict` osztály a `dict` viselkedését kísérli meg minél élethűbben utánozni. A lényegi különbség a két típus között abban áll, hogy a `SortedDict` objektumok kulcsai mindig rendezettek egy megadott kulcsfüggvény vagy az identitásfüggvény szerint. A `SortedDict` ugyanazt az API-t teszi elérhetővé, mint a `dict` (eltekintve a `repr()` függvénytől, amely az `eval()`-lal kiértékelve nem adja vissza az objektumot), két tagfüggvénnyel kiegészítve, amelyek használatának csak rendezett gyűjtemények esetében van értelme.*

Lássunk most néhány példát, hogy megismerkedjünk a `SortedDict` felhasználási lehetőségeivel:

```
d = SortedDict.SortedDict(dict(s=1, A=2, y=6), str.lower)
d["z"] = 4
d["T"] = 5
del d["y"]
d["n"] = 3
d["A"] = 17
str(d) # returns: "{'A': 17, 'n': 3, 's': 1, 'T': 5, 'z': 4}"
```

`SortedDict` osztályunk megvalósításánál egyaránt felhasználjuk az összetétel és az öröklés lehetőségeit. A kulcsok rendezett listáját egy példányváltozó formájában ágyazzuk be, ugyanakkor a `SortedDict` osztály a `dict` leszármazottja. A kód áttekintését a `class` sorral és a kezdeti értékadással nyitjuk, majd sorra vesszük az osztály tagfüggvényeit.

```
class SortedDict(dict):
    def __init__(self, dictionary=None, key=None, **kwargs):
        dictionary = dictionary or {}
        super().__init__(dictionary)
        if kwargs:
            super().update(kwargs)
        self.__keys = SortedList.SortedList(super().keys(), key)
```

A `class` sorban megadjuk a `dict` alaposztályt, majd a kezdeti értékadásnál a `dict()` függvényhez hasonlóan járunk el, de a kezdeti értékadó tagfüggvény egy további paramétert is vár a kulcsfüggvény beolvasására. `SortedDict` objektumunk kezdeti beállításait az alaposztály megfelelő függvényével végezzük el a `super().__init__()` hívással. Hasonlóan, ha kulcsszó-paramétereket alkalmazunk, az alaposztály `dict.update()` tagfüggvényével helyezzük el őket a szótárban. (Fontos megjegyeznünk, hogy a kulcsszó-paramétereknek csak egy előfordulása lehetséges, így a `kwargs` kulcsszó-paramétereiben nem fordulhat elő a `"dictionary"` vagy a `"key"` kulcsszó.)

* Az itt bemutatott `SortedDict` osztály eltér a szerző *Rapid GUI Programming with Python and Qt* című könyvében (ISBN 0132354187) találhatótól, valamint a Python Package Index hasonló nevű osztályától.

A szótár kulcsaiból másolatokat tárolunk egy rendezett listában, amely a `self.__keys` változóban kap helyet. A rendezett lista kezdeti beállításához átadjuk a szótár kulcsait az alaposztály `dict.keys()` tagfüggvényének – a `SortedList.keys()` tagfüggvényt nem használhatjuk, hiszen ez a `self.__keys` változóra épít, ami csak az után jön létre, hogy létrehoztuk ezt a rendezett listát.

```
def update(self, dictionary=None, **kwargs):
    if dictionary is None:
        pass
    elif isinstance(dictionary, dict):
        super().update(dictionary)
    else:
        for key, value in dictionary.items():
            super().__setitem__(key, value)
    if kwargs:
        super().update(kwargs)
    self.__keys = SortedList.SortedList(super().keys(),
                                        self.__keys.key)
```

Ez a tagfüggvény teszi lehetővé, hogy egy szótár elemeit egy másikéival, kulcsszó-paraméterekkel, illetve mindkét módon frissítsük. Azok az elemek, amelyek csak a másik szótárban léteznek, bekerülnek ebbe, míg ott, ahol a kulcs megegyezik, az új érték felülírja a régit. Az eredeti viselkedést némiképp bővítenünk kellett, így megtartjuk az eredeti szótár kulcsfüggvényét, még akkor is, ha a másik szótár a `SortedDict` egy példánya.

A frissítést két lépésben végezzük el. Először a szótár elemeit frissítjük. Amennyiben az adott szótár a `dict` alosztálya (ide tartozik a `SortedDict` is), az alaposztály `dict.update()` tagfüggvényét alkalmazzuk – így elkerüljük, hogy a `SortedDict.update()` tagfüggvény hívásával végtelen önhivatkozáshoz jussunk. Ha a szótár nem `dict` objektum, sorra vesszük az elemeit, és egyenként állítjuk be a kulcs-érték párokat. (Amennyiben a szótár nem `dict` objektum, ráadásul nem rendelkezik az `items()` tagfüggvénnyel, `AttributeError` kivételt váltunk ki.) Ha kulcsszó-paramétereket alkalmaztunk, a beágyazásukhoz ismét az alaposztály `update()` tagfüggvényét hívjuk meg.

A frissítés következtében a `self.__keys` lista elavulttá válik, így helyettesítjük egy új `SortedList` objektummal, amely tartalmazza a szótár kulcsait (ezeket az alaposztálytól kell beszereznünk, ugyanis a `SortedDict.keys()` tagfüggvény éppen azt a `self.__keys` listát használja, amelyet frissítünk), valamint az eredeti rendezett lista kulcsfüggvényét.

```
@classmethod
def fromkeys(cls, iterable, value=None, key=None):
    return cls({k: value for k in iterable}, key)
```

A `dict` osztály API-ja tartalmazza a `dict.fromkeys()` tagfüggvényt, amellyel új szótárt hozhatunk létre egy bejárható objektumból kiindulva. A bejárható objektum elemeiből lesznek a kulcsok, amelyek a `None`, illetve a megadott értéket kapják.

Osztálytagfüggvényről lévén szó, az első paramétert a Python szolgáltatja, értéke pedig maga az osztály. A `dict` osztály esetében ez a paraméter a `dict`, a `SortedDict` esetében pedig a `SortedDict`. Visszatérési értékként megkapjuk az osztályhoz tartozó szótárt, valahogy így:

```
class MyDict(SortedDict.SortedDict): pass
d = MyDict.fromkeys("VEINS", 3)
str(d)        # az eredmény: "{'E': 3, 'I': 3, 'N': 3, 'S': 3, 'V': 3}"
d.__class__.__name__          # az eredmény: 'MyDict'
```

Ha tehát az örökölt osztálytagfüggvényeket hívjuk meg, a `cls` változójukba a megfelelő osztály kerül – éppúgy, mint az egyszerű tagfüggvényeknél, ahol a `self` változó mindig az aktuális objektumot tartalmazza. Ezért ez jobb megoldás, mint a statikus tagfüggvények használata, hiszen azok adott osztályhoz kötődnek, függetlenül attól, hogy az eredeti osztály vagy egy alosztályának környezetében futtatjuk.

```
def __setitem__(self, key, value):
    if key not in self:
        self.__keys.add(key)
    return super().__setitem__(key, value)
```

Ezzel a tagfüggvénnyel valósítjuk meg a `d[key] = value` alakot. Ha a `key` nem szerepel a szótárban, hozzáadjuk a kulcsok listájához, a `SortedList` segítségével megtalálva a helyét a sorban. Ez után meghívjuk az alaposztály megfelelő tagfüggvényét, az eredményt pedig visszaadjuk a hívónak, hogy lehetővé tegyük a műveletek láncba rendezését, valahogy így: `x = d[key] = value`.

Vegyük észre, hogy az `if` utasításban a `not in self` kóddal vizsgáljuk meg, hogy a kulcs szerepel-e már a `SortedDict` objektumban. Mivel a `SortedDict` a `dict` leszármazottja, a példányai bárhol szerepelhetnek, ahol a `dict` objektumok, már pedig ez esetben a `self` `SortedDict` típusú. Amikor a `dict` tagfüggvényeit újra megvalósítjuk a `SortedDict` osztályban, ügyeljünk arra, hogy a szülőosztálybeli megvalósításokat a `super()` segítségével hívjuk meg – így tettünk a fenti tagfüggvény utolsó sorában is. Ezzel meggátoljuk, hogy az újra megvalósított tagfüggvény saját magát hívja meg, és így végtelen önhivatkozásba kerüljön.

Generátorfüggvények

A *generátorfüggvények*, illetve *generátor-tagfüggvények* jellemzője, hogy tartalmazzák a `yield` kifejezést, eredményként pedig bejárókat adnak vissza, amelyeknek az értékeit egyenként olvashatjuk ki a `__next__()` tagfüggvényükkel. A `__next__()` tagfüggvény minden hívásakor hozzájutunk a generátorfüggvény `yield` kifejezésének értékéhez (ha nincs ilyen, akkor a kapott érték a `None`). Ha a generátorfüggvény futása befejeződik, illetve végrehajt egy `return` utasítást, `StopIteration` kivétel lép fel.

A gyakorlatban ritkán hívjuk meg a `__next__()` tagfüggvényt, illetve fogjuk el a `StopIteration` kivételt – ehelyett úgy használjuk a generátorfüggvényeket, mint bármely más bejárható objektumot. Az alábbiakban két, majdhogynem egyenértékű függvényt mutatunk be – a bal oldali egy listát ad vissza, a jobb oldali pedig egy generátort.

```
# Létrehoz egy listát és visszaadja
def letter_range(a, z):
    result = []
    while ord(a) < ord(z):
        result.append(a)
        a = chr(ord(a) + 1)
    return result
```

```
# Kérésre szolgáltatja az egyes
# értékeket
def letter_range(a, z):
    while ord(a) < ord(z):
        yield a
        a = chr(ord(a) + 1)
```

Az eredmény elemeit mindkét esetben sorra vehetjük egy `for` ciklussal, valahogy így: `for letter in letter_range("m", "v"):`. Ha azonban a kapott betűk listájára vagyunk kíváncsiak, a `letter_range("m", "v")` csak a jobb oldali esetben elegendő, míg a bal oldali generátorfüggvénynél a `list(letter_range("m", "v"))` alakot kell használnunk.

A generátorfüggvényeket és -tagfüggvényeket (valamint a generátorkifejezéseket) bővebben a 8. fejezetben tárgyaljuk.

A `__getitem__()` tagfüggvényt nem kell újra megvalósítanunk, hiszen az alaposztálybeli változat nagyszerűen működik, és nincs hatással a kulcsok sorrendjére.

```
def __delitem__(self, key):
    try:
        self.__keys.remove(key)
    except ValueError:
        raise KeyError(key)
    return super().__delitem__(key)
```

Ez a tagfüggvény teszi lehetővé a `del d[key]` alak használatát. Ha a `key` nincs jelen, a `SortedList.remove()` hívás eredményeképpen `ValueError` kivételt kapunk. Ha ez bekövetkezik, elfogjuk, és `KeyError` kivételt váltunk ki helyette, hogy megfeleljünk a `dict` osztály API-jának. Egyébként pedig azt az eredményt adjuk vissza, amit a tagfüggvény alaposztálybeli változatától a kulcshoz tartozó elem törlésével kapunk.

```
def setdefault(self, key, value=None):
    if key not in self:
        self.__keys.add(key)
    return super().setdefault(key, value)
```

A fenti tagfüggvény visszaadja az adott kulcshoz tartozó értéket, amennyiben ez a kulcs megtalálható a szótárban, egyébként pedig létrehoz egy új elemet a meghatározott kulccsal és értékkel, majd visszaadja ez utóbbit. A `SortedDict` esetében biztosítanunk kell, hogy a kulcsot elhelyezzük a kulcsok listájában, amennyiben még nem szerepel a szótárban.

Ha a megadott kulcs jelen van a szótárban, ez a tagfüggvény visszaadja a hozzá tartozó értéket, majd törli a szótárból ezt a kulcs-érték párt. A kulcsot szintén el kell távolítanunk a kulcsok listájából.

A megvalósítás meglehetősen körülményes, mivel a `pop()` tagfüggvénynek két eltérő viselkedést kell mutatnia, hogy megfeleljen a `dict.pop()` működésének. Az első a `d.pop(k);` alakban érhető el – ilyenkor visszakapjuk a `k` kulcshoz tartozó értéket, ha pedig a `k` kulcs nem található, `KeyError` kivételt kapunk. A második használati alak a `d.pop(k, value);`, amelynél szintén a `k` kulcs értékéhez jutunk, ha viszont a kulcs nem elérhető, a `value` értéket kapjuk vissza (ami lehet `None` is). Bárhogy is legyen, ha a `k` kulcs létezik, a tagfüggvény eltávolítja a hozzá tartozó elemet.

```
def popitem(self):
    item = super().popitem()
    self.__keys.remove(item[0])
    return item
```

A `dict.popitem()` tagfüggvény eltávolít egy véletlenszerűen kiválasztott kulcs-érték párt az osztályból, amelyet eredményként visszaad. Először az alaposztálybeli változatot kell meghívnunk, hiszen nem tudjuk előre, hogy melyik elemet fogjuk eltávolítani. Töröljük az elemhez tartozó kulcsot a kulcsok listájából, és visszaadjuk az elemet.

```
def clear(self):
    super().clear()
    self.__keys.clear()
```

Itt töröljük a szótár, valamint a kulcsok listájának összes elemét.

```
def values(self):
    for key in self.__keys:
        yield self[key]
def items(self):
    for key in self.__keys:
        yield (key, self[key])
def __iter__(self):
    return iter(self.__keys)
keys = __iter__
```

A szótárak négy olyan tagfüggvénnyel rendelkeznek, amelyek bejárókat adnak vissza – a `dict.values()` a szótár értékeit, a `dict.items()` a kulcs-érték elemeket, a `dict.keys()` a kulcsokat, az `__iter__()` különleges tagfüggvény pedig kulcsokat vár, és az `iter(d)` alakot támogatja. (Valójában a tagfüggvények alaposztálybeli változatai szótárnézeteket adnak vissza, de a legtöbb esetben az itt megadott bejárók viselkedése azonos.)

Mivel az `__iter__()` és a `keys()` tagfüggvények viselkedése megegyezik, a `keys()` megvalósítása helyett egyszerűen létrehozhatunk egy `keys` nevű objektumhivatkozást, amellyel az `__iter__()` tagfüggvényre hivatkozunk. Így a `SortedDict` felhasználói a `d.keys()` vagy az `iter(d)` hívásával a szótár kulcsait felsoroló bejárót kaphatnak – hasonlóan ahhoz, ahogy a `d.values()` hívással kapott bejáró az értékeket veszi sorra.

A `values()` és az `items()` generátor-tagfüggvények – működésükről röviden az előző keretes írásban szóltunk. Mindkét esetben a kulcsok rendezett listáján haladnak végig, ami azt jelenti, hogy a kulcsok sorrendjét követik (a kulcsok sorrendje a kezdeti értékadásnál meghatározott kulcsfüggvénytől függ). Az `items()` és a `values()` tagfüggvények esetében az értékeket `d[k]` alakban olvassuk ki (ilyenkor a háttérben a `dict.__getitem__()` működik), hiszen a `self` objektumokat kezelhetjük `dict` típusúként.

Generátorok ☞ 346

```
def __repr__(self):
    return object.__repr__(self)
def __str__(self):
    return ("{" + ", ".join(["{0!r}: {1!r}".format(k, v)
                             for k, v in self.items()]) + "}")
```

A `SortedDict` számára nem adhatunk meg az `eval()` segítségével kiértékelhető ábrázolást, ugyanis ilyen a kulcsfüggvényhez sem áll rendelkezésre. A `__repr__()` megvalósításánál ezért átugorjuk a `dict.__repr__()` tagfüggvényt, és inkább az ősváltozatot (`object.__repr__()`) használjuk. Így olyasforma karakterláncot kapunk, mint általában az `eval()` révén ki nem értékelhető objektumoknál, vagyis valami ilyesmit: `'<SortedDict.SortedDict object at 0xb71fff5c>'`.

A `SortedDict.__str__()` tagfüggvényt magunk valósítottuk meg, mivel az elemeket a kulcsok szerint rendezett alakban szeretnénk kiírni. A tagfüggvényt így is megírhattuk volna:

```
items = []
for key, value in self.items():
    items.append("{0!r}: {1!r}".format(key, value))
return "{" + ", ".join(items) + "}"
```

Listaértelmezés használatával rövidebb alakot kapunk, és az `items` átmeneti változóra sincs szükség.

Az alaposztály `dict.get()`, `dict.__getitem__()` (a `v = d[k]` alakhoz), `dict.__len__()` (a `len(d)` használatához), valamint `dict.__contains__()` (az `x in d` alakhoz) tagfüggvényei tökéletesen megfelelnek a leszármazottban is, hiszen nem befolyásolják a kulcsok sorrendjét – ezért nem is kell őket újra megvalósítanunk.

A `dict` osztály utolsó tagfüggvénye, amelynek a megvalósítására szükségünk van, a `copy()`.

```
def copy(self):
    d = SortedDict()
    super(SortedDict, d).update(self)
    d.__keys = self.__keys.copy()
    return d
```

Választhattuk volna az egyszerűbb, `def copy(self): return SortedDict(self)` alakot, de mégis a fenti mellett döntöttünk, ugyanis így elejét vehetjük a már rendezett kulcsok újrarendezésének. Létrehozunk egy üres rendezett szótárat, majd az alaposztály `dict.update()` tagfüggvénye révén frissítjük az eredeti rendezett szótár elemeivel, így elkerülhetjük a `SortedDict.update()` újbóli megvalósítását, és a szótár `self.__keys` `SortedList` objektumát az eredeti sekély másolatával helyettesítjük.

Ha a `super()` tagfüggvényt paraméterek nélkül hívjuk meg, az alaposztályon, illetve a `self` objektumon fejti ki a hatását. Ugyanakkor tetszőleges objektumra vagy osztályra is alkalmazhatjuk, ha átadjuk számára azt. Ilyenkor a tagfüggvény a megadott osztály közvetlen ősére hat, így esetünkben a kód egyenértékű ezzel: `dict.update(d, self)`.

Mivel a Python rendezési algoritmusa igen gyors, és a működése különösen alkalmas részlegesen rendezett listák kezelésére, az így nyert hatékonyság kicsi vagy elhanyagolható mértékű a hatalmas szótárak esetét kivéve. Mindazonáltal a fenti kód legalább elviekben mutatja, hogy az egyedi `copy()` tagfüggvények használata hatékonyabb lehet, mint a Python beépített típusai által támogatott `copy_of_x = ClassOfX(x)` alak. A `SortedList` esetéhez hasonlóan a `__copy__ = copy` beállítással éltünk, így a `copy.copy()` függvény a saját kódja helyett a mi egyedi másoló tagfüggvényünket használja:

```
def value_at(self, index):
    return self[self.__keys[index]]
def set_value_at(self, index, value):
    self[self.__keys[index]] = value
```

Ez a két utolsó tagfüggvény a `dict` API bővítése, amelyekre azért volt szükség, mert a `SortedDict` a `dict` osztálytól eltérően rendezett, vagyis a kulcsok sorszámai is értelmet nyernek. Így például a szótár első eleme a `0` sorszámmal határozható meg, az utolsó pedig `len(d)-1`-gyel. Mindkét tagfüggvény olyan szótárelemet ér el, amelynek a kulcsa a kulcslistában az `index`-edik elem. Az öröklésnek köszönhetően a `SortedDict` objektum értékeit elérhetjük az elemelérési művelettel (`[]`) közvetlenül a `self`-en keresztül, hiszen ez egyben `dict` típusú is. Ha tartományon kívüli sorszámot adunk meg, `IndexError` kivételt kapunk.

Ezzel a `SortedDict` osztály megvalósításának végére értünk. Ritka, hogy ehhez hasonló teljes általános gyűjteményosztályokat kell létrehoznunk, de ha mégis erre van szükség, a Python különleges tagfüggvényei lehetővé teszik, hogy az osztályt teljesen beépítsük a nyelv szerkezetébe, így a felhasználók úgy kezelhetik, mintha a beépített típusok közül származna.

Összefoglalás

Ebben a fejezetben körbejártuk a Python objektumközpontú programozási lehetőségeinek alapjait. Először megismerkedtünk a tisztán eljárásközpontú megközelítés bizonyos hátrányaival, amelyek az objektumközpontú módszerekkel elkerülhetők. Ezt követően betekintést nyerhettünk az objektumközpontú világ fogalmaiba – így találkozhattunk több szinonimával is, mint az *alaposztály* és az *ős*.

Megtanultuk, hogyan hozhatunk létre egyszerű osztályokat adatjellemzőkkel és egyedi tagfüggvényekkel, és szó esett az öröklésről is, amelynek során már létező osztályokat egészíthetünk ki újabb adatjellemzőkkel és tagfüggvényekkel, továbbá bizonyos tagfüggvényeket kiiktathatunk. Ez utóbbira akkor lehet szükség, ha az osztályunkat egy másikból származtatjuk, de korlátozni szeretnénk az onnan kapott egyes tagfüggvények használatát. Ezt a módszert körültekintően kell alkalmaznunk, ugyanis ezzel elveszítjük a többalakúságot – vagyis az új osztályt nem mindenütt használhatjuk a régi helyében.

Az egyedi osztályokat tökéletesen beépíthetjük a Python szerkezetébe, így pontosan olyan alakban használhatók, mint a nyelv beépített és könyvtári osztályai. Mindezt a különleges tagfüggvények megvalósításával érhetjük el. Láthattuk, hogy a különleges tagfüggvényekkel miként támogathatjuk az összehasonlítást, hogyan biztosíthatunk karakterlánc- és ábrázolási alakot az objektumainknak, továbbá hogyan alakíthatjuk át az objektumokat más (például `int` vagy `float`) típusúvá, amennyiben ennek van értelme. Külön foglalkoztunk a `__hash__()` tagfüggvény megvalósításával, amely lehetőséget ad arra, hogy az egyedi osztályunk példányai szótárak kulcsaiként és halmazok elemeiként szerepelhessenek.

Az adatjellemzők önmagukban nem képesek biztosítani, hogy csak érvényes értékeket fogadjanak. Láthattuk, milyen egyszerűen helyettesíthetjük az adatjellemzőket tulajdonságokkal – így létrehozhatunk csak olvasható tulajdonságokat, az írhatóak esetében pedig könnyedén biztosíthatjuk az értékek ellenőrzését.

Az általunk létrehozott osztályok többsége „nem teljes", ugyanis jobbára csak azokat a tagfüggvényeket írjuk meg, amelyekre valóban szükség van. Ez persze minden további nélkül megtehető a Pythonban, de emellett teljes egyedi osztályokat is létrehozhatunk, amelyek minden elvárt tagfüggvényt elérhetővé tesznek. Először megtanultuk, hogyan hozzunk létre ilyen, egy értéket tároló osztályokat, mind a nulláról indulva, mind pedig az öröklés révén, majd pedig szó esett a gyűjteményosztályok megvalósításáról is. Az egyedi gyűjteményosztályok ugyanazokat a lehetőségeket képesek biztosítani, mint beépített társaik, így elérhetővé tehetik az `in` és az elemelérési (`[]`) műveletet, valamint a `len()`, az `iter()` és a `reversed()` függvényeket.

Megtudtuk, hogy az objektumok létrehozása és kezdeti értékadása két önálló lépés, amelyek szabályozását a Python lehetővé teszi, jóllehet jobbára csak a kezdeti értékadás testreszabásával foglalkozunk. Azt is megtanultuk, hogy az állandó értékű adatjellemzőket

nyugodtan visszaadhatjuk, míg a változtathatók esetében jobb, ha csak másolatot szolgáltatunk, ellenkező esetben ugyanis az objektum belső állapota nyilvánossá válhat, és esetleg véletlenül ezen keresztül érvényteleníthetjük az objektumot.

A Python egyszerű, statikus és osztálytagfüggvényeket, valamint modulfüggvényeket tesz elérhetővé a számunkra. Láttuk, hogy a tagfüggvényeink többsége az első csoportba tartozik, de olykor hasznát vesszük az osztálytagfüggvényeknek is. Ugyanakkor a statikus tagfüggvények lehetőségeit igen ritkán vesszük igénybe, mivel az osztálytagfüggvények és a modulfüggvények szinte minden esetben jobban használhatók helyettük.

A beépített `repr()` függvény egy objektum `__repr__()` különleges tagfüggvényét hívja meg. Ahol csak lehetséges, törekszünk arra, hogy az `eval(repr(x)) == x` azonosság fennálljon – láttuk, miként gondoskodhatunk erről. Ha mégsem tudunk ilyen ábrázoló karakterláncot létrehozni, az alaposztály `object.__repr__()` tagfüggvényéhez folyamodunk, ami szabványos eredményt ad – így azonban nem kapjuk vissza a kiértékeléssel az eredeti objektumot.

A beépített `isinstance()` függvénnyel végzett típusvizsgálat jelenthet némi hatékonyságnövekedést, de akik igazán komolyan veszik az objektumközpontú programozást, tartózkodnak a használatától. Az alaposztály tagfüggvényeit a beépített `super()` függvénnyel érhetjük el, amelynek a használata alapvető fontosságú, amikor egy tagfüggvény alosztálybeli újramegvalósításában az alaposztály megfelelő tagfüggvényét kell használnunk.

A generátorfüggvények és -tagfüggvények lépésről lépésre végzik a kiértékelést, vagyis az eredményt (a `yield` utasítás révén) értékenként, kérésre adják vissza, ha pedig ez nem lehetséges, `StopIteration` kivételt váltanak ki. A generátorokat mindenütt használhatjuk, ahol a kódban bejárókra (iterátorokra) van szükség, véges generátorok esetén pedig a lehetséges értékeiket listákban, illetve rendezett sorozatokban tárolhatjuk, ha a visszakapott bejárót átadjuk a `list()` vagy a `tuple()` függvénynek.

Az objektumközpontú programozás szinte minden esetben egyszerűbbé teszi a kódot a tisztán eljárásközpontú megközelítéshez képest. Az egyedi osztályokkal biztosíthatjuk, hogy csak a megfelelő műveletek legyenek elérhetők (hiszen csak ezeket a tagfüggvényeket valósítjuk meg), továbbá azt, hogy egyetlen művelet se tegye az objektum adatait érvénytelenné (erre a tulajdonságok adnak lehetőséget). Ahogy ráérzünk az objektumközpontú megközelítés ízére, úgy távolodik el a programozási stílusunk a globális adatszerkezetektől és az ezeket elérő globális függvényektől, és úgy használunk egyre inkább osztályokat és hozzájuk tartozó tagfüggvényeket. Az objektumközpontú programozás lehetővé teszi, hogy egységbe zárjuk az adatokat és azokat a tagfüggvényeket, amelyek alkalmasak ezen adatok kezelésére. Ezzel megelőzhetjük az adatok és a függvények keveredését, és programjaink könnyebben fenntarthatóvá válnak, hiszen az egyes működési egységek önálló osztályokban kapnak helyet.

Gyakorlatok

Az első két gyakorlatban a fejezetben bemutatott osztályok módosításával foglalkozunk, míg a másik kettőben teljesen új egyedi osztályokat kell létrehoznunk.

1. Módosítsuk a `Point` osztályt (lásd a `Shape.py` vagy a `ShapeAlt.py` fájlban) úgy, hogy támogassa az alábbi műveleteket, ahol `p`, `q` és `r` `Point` objektumok, `n` pedig egy szám:

```
p = q + r    # Point.__add__()
p += q       # Point.__iadd__()
p = q - r    # Point.__sub__()
p -= q       # Point.__isub__()
p = q * n    # Point.__mul__()
p *= n       # Point.__imul__()
p = q / n    # Point.__truediv__()
p /= n       # Point.__itruediv__()
p = q // n   # Point.__floordiv__()
p //= n      # Point.__ifloordiv__()
```

A hozzárendelési tagfüggvények megvalósítása négy-négy sor hosszú, ha a `def` sorát is beleszámítjuk, míg társaik ugyanígy számítva két sorból állnak – emellett természetesen igen egyszerűek és hasonlítanak egymásra. Némi leírást és pár tesztet is mellékelve összesen nagyjából százharminc új sorra lesz szükség. Egy lehetséges megoldást mutat a `Shape_ans.py`, valamint a `ShapeAlt_ans.py` fájl.

2. Módosítsuk az `Image.py` osztályt úgy, hogy elérhetővé tegye a `resize(width, height)` tagfüggvényt. Ha az új szélesség vagy magasság kisebb, mint az aktuális érték, az új határokon kívül eső pontokat törölnünk kell. Amennyiben akár a szélesség, akár a magasság paramétereként a `None` értéket adjuk meg, használjuk az eredeti beállítást. Végül pedig gondoskodjunk a `self.__colors` halmaz frissítéséről, és egy logikai értéket visszaadva jelezzük, hogy történt-e módosítás. A tagfüggvényt kevesebb mint 20 sorban elkészíthetjük (illetve 35-ben, amennyiben egyszerű leírást és egy tesztet is mellékelünk), és az `Image_ans.py` fájlban megtekinthetünk egy lehetséges megoldást.
3. Készítsünk el egy `Transaction` nevezetű osztályt, amely egy összeget, egy dátumot, egy pénznemet (utóbbi esetben az alapértelmezés az amerikai dollár, `"USD"`), egy dollárátváltási arányt (alapértelmezés szerint `1`), valamint egy leírást (alapértelmezés: `None`) vár. Minden esetben alkalmazzunk privát adatjellemzőket. Hozzuk létre a következő csak olvasható tulajdonságokat: `amount`, `date`, `currency`, `usd_conversion_rate`, `description` és `usd` (amelyet az `amount * usd_conversion_rate` képlettel számítunk ki). Az osztály nagyjából hatvan sorban megvalósítható – beleértve ebbe néhány tesztet is. Az `Account.py` egy lehetséges megoldást ad erre a gyakorlatra (és a következőre).
4. Valósítsuk meg az `Account` osztályt, amely egy számla számát, nevét, valamint egy `Transaction` objektumokból álló listát tartalmaz. A számlaszám csak olvasható legyen, a név viszont írható és olvasható, de egy feltételezéssel biztosítsuk, hogy

a hossza elérje a négy karaktert. Az osztály támogassa a beépített `len()` függvényt (ez adja vissza a tranzakciók számát), továbbá hozzunk létre két csak olvasható számított tulajdonságot: a `balance` adja meg a számlaegyenleget amerikai dollárban, az `all_usd` értéke pedig legyen `True`, ha minden tranzakció dollárban megy végbe, egyébként pedig `False`. Szükség van három további tagfüggvényre: az `apply()` segítségével elvégezhetünk egy tranzakciót, a `save()` és a `load()` tagfüggvényekkel pedig adatokat tárolunk. Ehhez bináris befőttet készítünk, amelynek a fájlneve a számlaszám, a kiterjesztése pedig `.acc`. A `save()` és a `load()` feladata, hogy biztosítsa a számla számának, nevének és tranzakciós adatainak tárolását és betöltését. Az osztály mindösszesen – egyszerű tesztekkel, valamint a mentés és betöltés lehetőségeivel együtt – nagyjából 90 sorban elkészíthető. Használjunk a `name = os.path.join(tempfile.gettempdir(), account_name)` sorhoz hasonló kódrészletet egy ideiglenes fájlnév létrehozására, ha pedig elkészültünk a tesztekkel, töröljük ezt a fájlt. Egy lehetséges megoldást találhatunk az `Account.py` fájlban.

7

Fájlkezelés

- Bináris adatok írása és olvasása
- Szövegfájlok írása és feldolgozása
- XML-fájlok írása és olvasása
- Véletlen elérésű bináris fájlok

A legtöbb program a működése során fájlokból olvas és fájlokba ír: például adatot, állapotinformációt. A Python többféle módon támogatja a fenti műveletek megvalósítását.

A szövegfájlok kezeléséről esett már néhány szó a 3. fejezetben, és érintőlegesen az előző fejezetben is előkerült a téma. Ebben a fejezetben lényegesen mélyebben tárgyaljuk a fájlkezelés témakörét.

A fejezetben bemutatott valamennyi megoldás független a használt rendszertől. Ez annyit tesz, hogy a példaprogramok által az egyik gép processzorával és operációs rendszerével mentett fájlok a másik gép processzorával és operációs rendszerével is beolvashatók. Mindez az Olvasó által készített programok esetében is igaz lehet, amennyiben a példaprogramokkal megegyező módszereket használja bennük.

A fejezet első három szakasza azt a gyakori esetet tárgyalja, amikor teljes adatgyűjtemények lemezre mentését, illetve lemezről történő beolvasását végezzük el. Az első szakasz azt mutatja be, hogy miként tehető ez meg bináris fájlformátumok használatával. Először „befőttet" fogunk használni – akár tömörítetteket is –, utána pedig befőttek nélkül végezzük el a munkát. A második rész a szövegfájlok kezelését mutatja be. A szövegek írása egyszerű, a visszaolvasásuk azonban nem feltétlenül az, főleg ha nem csak szöveget, hanem dátumokat vagy számokat is vissza kell olvasnunk. A szövegfeldolgozásnak két megközelítését mutatjuk be: a kézi feldolgozás mellett megismerjük a szabályos kifejezéseken alapuló módszert is. A harmadik szakasz az XML-fájlok olvasását és írását mutatja be. Ebben a részben az írást és a feldolgozást elvégezzük elemfákkal, illetve a DOM (Document Object Model, dokumentum-objektummodell) használatával is. Az írást elvégezzük kézi módszerrel is, feldolgozáskor pedig a SAX (Simple API for XML) használatát is kipróbáljuk.

A negyedik szakasz a véletlen vagy tetszőleges elérésű bináris fájlok kezelését mutatja be. Ennek akkor vesszük hasznát, ha az egyes adatok azonos méretűek, és több adatelemünk van, mint amennyit el akarunk, illetve el tudunk helyezni a gép memóriájában.

Teljes gyűjtemények tárolására melyik fájlformátum a legmegfelelőbb? A bináris fájlok, a szövegfájlok, vagy az XML? Melyik az egyes formátumok kezelésére a legmegfelelőbb módszer? Ezek a kérdések túlzottan függnek az adott helyzettől ahhoz, hogy egyetlen helyes válasz adható legyen rájuk, már csak azért is, mert nem sorolhatók fel az egyes formátumok és kezelési módok előnyei, illetve hátrányai. Bemutatjuk az egyes kombinációkat, így segítve az Olvasót, hogy megalapozott döntést hozhasson a konkrét esetekben.

7.1. ábra *Repülőgép-balesetek tárolására szolgáló adatbázisrekordok*

Név	Adattípus	Megjegyzés
`report_id`	`str`	Legalább nyolc karakter, nincs benne szóköz
`date`	`datetime.date`	
`airport`	`str`	Nem lehet üres, nincs benne újsor karakter
`aircraft_id`	`str`	Nem lehet üres, nincs benne újsor karakter
`aircraft_type`	`str`	Nem lehet üres, nincs benne újsor karakter
`pilot_percent_hours_on_type`	`float`	Tartomány 0.0-tól 100.0-ig
`pilot_total_hours`	`int`	Pozitív egész szám
`midair`	`bool`	
`narrative`	`str`	Többsoros

A bináris formátumok mentése és betöltése rendszerint nagyon gyors, a fájlok mindemellett igen tömörek lehetnek. A bináris adatok esetében nincs szükség feldolgozásra, ugyanis ilyenkor az egyes adattípusokba tartozó változókat a memóriabeli ábrázolásuknak megfelelően tároljuk. A binárisan tárolt adatokat mi emberek nem tudjuk átalakítás nélkül elolvasni, illetve írni, a pontos formátum ismerete nélkül pedig lehetetlen a programunktól különálló, a bináris adatokat kezelni képes eszközt kialakítani.

A szövegfájlokat az ember azonnal képes olvasni és szerkeszteni, ezáltal könnyebbé válik a szövegfájlok programunktól különálló eszközzel való feldolgozása, szövegszerkesztővel való szerkesztése. A szöveges formátumú fájlok feldolgozása nehézségeket rejthet magában, ha pedig a formátum sérül – mondjuk a kellő odafigyelés nélkül végzett szerkesztés miatt –, sokszor gondot jelent a megfelelő hibaüzenet megjelenítése.

Az XML formátumú fájlok ember által olvashatók és szerkeszthetők, ugyanakkor gyakran válnak túlontúl részletessé, és gyakran eredményeznek túlméretezett fájlokat. A szöveges formátumokhoz hasonlóan az XML formátumok feldolgozása is megoldható a programunktól különálló eszközökkel. Az XML formátumú fájlok feldolgozása egyszerű – már amennyiben nem saját kezűleg óhajtjuk elvégezni a feladatot, hanem valamilyen XML-feldolgozót használunk –, és néhány XML-feldolgozó igen ügyes a hibajelzéseket illetőleg. Az XML-feldolgozók igen lassúak tudnak lenni, azaz a nagyon nagy XML-fájlok beolvasása lényegesen hosszabb időt vehet igénybe a megfelelő bináris, illetve szöveges fájlok beolvasásánál. Az XML formátumú fájlok metaadatokat is tartalmazhatnak – például a karakterkódolás is megadható –, így a hordozásuk egyszerűbb a szöveges fájlokénál.

A végfelhasználók számára rendszerint a szöveges állományok használata a legkényelmesebb, de néha a teljesítménnyel kapcsolatos megfontolásokból kifolyólag a bináris formátum az egyetlen szóba jöhető választás. Mindazonáltal minden esetben érdemes biztosítani az XML-fájlok betöltésének és mentésének lehetőségét, hiszen az ilyen formátumú fájlok feldolgozása úgy lehetséges külső eszközök felhasználásával, hogy közben nem kell lemondanunk az optimális szöveges, illetve bináris formátum használatáról a program normális működése közben.

7.2. ábra

A repülőgép-baleseteket tároló különböző formátumú fájlok írásának és olvasásának összehasonlítása

Formátum	Olvasó/Író	Olvasó + író kód sorainak száma		Kód sorainak száma összesen	Kimeneti állomány mérete (~KB)
Bináris	Befőtt (gzip tömörítés)	20 + 16	=	36	160
Bináris	Befőtt	20 + 16	=	36	416
Bináris	Kézi (gzip tömörítés)	60 + 34	=	94	132
Bináris	Kézi	60 + 34	=	94	356
Szöveges	Szabályos kifejezés olvasó/kézi író	39 + 28	=	67	436
Szöveges	Kézi	53 + 28	=	81	436
XML	Elemfa	37 + 27	=	64	460
XML	DOM	44 + 36	=	80	460
XML	SAX olvasó/kézi író	55 + 37	=	92	464

A fejezet első három szakaszában ugyanazt az adatgyűjteményt használjuk: repülőgép-baleseteket tároló adatbázisrekordok halmazát. A 7.1. ábra a repülőgép-baleseteket tároló adatbázisrekordok mezőinek nevét, adattípusát és érvényességi megszorításait mutatja be. A feldolgozott adat nem igazán számít, a fontos az, hogy megtanuljuk az alapvető adattípusok – karakterláncok, egész és lebegőpontos számok, logikai értékek és dátumok – feldolgozását, mégpedig azért, mert ha az előző típusokat fel tudjuk dolgozni, akkor bármilyen egyéb adattípussal is boldogulni fogunk.

Az, hogy a bináris, a szöveges és az XML formátum esetében is ugyanazt a repülőgép-balesetekről szóló adathalmazt dolgozzuk fel, lehetővé teszi a különféle formátumok és a kezelésükhöz szükséges kód összehasonlítását, illetve a közöttük lévő különbségek kiemelését. A 7.2. ábra bemutatja és összehasonlítja az egyes formátumok olvasásához és írásához szükséges kódsorok számát, illetve ezek összegét is.

A fájlméretek csak közelítőek, és az adott, 596 repülőgép-balesetet tartalmazó minta esetében érvényesek.* Ha ugyanazt az adatot más fájlnév használatával, valamilyen tömörített formátumban mentjük, az eredmény néhány bájtnyi eltérést mutathat, lévén a fájlnevek is bekerülnek a tömörített adatba, és a fájlnevek hossza eltérő. Ehhez hasonlóan az XML-fájlok mérete is változó lehet, hiszen néhány XML-író az idézőjeleket `&quot`, az aposztrófokat pedig `&apos` szöveggel helyettesíti a szöveges adatok belsejében, míg más programok ezt az átalakítást nem végzik el.

Az első három szakaszban lévő kódszövegek egyazon programból, a `convert-incidents.py` nevűből származnak. Ez a program képes a repülőgép-balesetek adatait az egyik formátumból beolvasni, majd egy másikba kiírni. Íme a program konzolon megjelenített súgószövege (a kimenetet kissé módosítottuk, hogy a könyv oldalszélességének megfeleljen):

```
Usage: convert-incidents.py [options] infile outfile
Reads aircraft incident data from infile and writes the data to
outfile. The data formats used depend on the file extensions:
.aix is XML, .ait is text (UTF-8 encoding), .aib is binary,
.aip is pickle, and .html is HTML (only allowed for the outfile).
All formats are platform-independent.

Options:
  -h, --help       show this help message and exit
  -f, --force      write the outfile even if it exists [default: off]
  -v, --verbose    report results [default: off]
  -r READER, --reader=READER
                   reader (XML): 'dom', 'd', 'etree', 'e', 'sax', 's'
                   reader (text): 'manual', 'm', 'regex', 'r'
                   [default: etree for XML, manual for text]
  -w WRITER, --writer=WRITER
                   writer (XML): 'dom', 'd', 'etree', 'e',
                   'manual', 'm' [default: manual]
  -z, --compress   compress .aib/.aip outfile [default: off]
  -t, --test       execute doctests and exit (use with -v for verbose)
```

A kapcsolók használata bonyolultabb annál, mint normális esetben lehetne, hiszen egy végfelhasználót vajmi kevéssé érdekel, hogy melyik olvasási és írási módszert használjuk egy adott formátum esetében. Ha a programot valós használatra szánnánk, az olvasási és

* A példában használt adatok valós repülőgép-balesetek adatain alapulnak. Az adatok forrása az FAA (az Egyesült Államok kormányának Federal Aviation Administration nevű intézménye, `www.faa.gov`).

az írási módszert beállító kapcsolók nem léteznének, és a módszerek közül formátumonként csak egyet-egyet valósítanánk meg. A `test` kapcsoló csak a kód tesztelésére szolgál, és ugyanígy nem volna jelen a program egy kiadott változatában.

A programban egyetlen saját kivételt adunk meg:

```
class IncidentError(Exception): pass
```

A repülőgép-baleseteket `Incident` osztályú objektumokként tartjuk nyilván. Íme a `class` utasítást tartalmazó sor és az osztály előkészítése:

```
class Incident:

def __init__(self, report_id, date, airport, aircraft_id,
             aircraft_type, pilot_percent_hours_on_type,
             pilot_total_hours, midair, narrative=""):
    assert len(report_id) >= 8 and len(report_id.split()) == 1, \
           "invalid report ID"
    self.__report_id = report_id
    self.date = date
    self.airport = airport
    self.aircraft_id = aircraft_id
    self.aircraft_type = aircraft_type
    self.pilot_percent_hours_on_type = pilot_percent_hours_on_type
    self.pilot_total_hours = pilot_total_hours
    self.midair = midair
    self.narrative = narrative
```

A jelentésazonosító érvényesítése akkor történik, amikor az osztályból származó objektumot példányosítjuk; elérése pedig a `report_id` nevű, csak olvasható tulajdonság olvasásával történik. Minden egyéb adattulajdonság írható és olvasható. Például íme a `date` (dátum) tulajdonság kódja:

```
@property
def date(self):
    return self.__date
@date.setter
def date(self, date):
    assert isinstance(date, datetime.date), "invalid date"
    self.__date = date
```

Minden további tulajdonság ezt a mintát követi, így az ismertetésüktől itt most eltekintünk, hiszen különbségeket mindössze az `assert` utasítást tartalmazó sorokban találunk.
Az `assert` utasítások használata miatt a program leáll, amennyiben érvénytelen adatok alapján szeretnénk létrehozni egy `Incident` osztályú objektumot, és olyankor is, ha egy már létező `Incident` osztályú objektum valamelyik tulajdonságának érvénytelen adatot próbálunk értékül adni. Azért döntöttünk ilyen szigorú elbírálás mellett az érvénytelen adatokat illetően, mert biztosak akarunk lenni benne, hogy a mentett és beolvasott adat mindig érvényes. Ha pedig mégsem volna az, jobb, ha a program futása némi panaszkodás mellett megszakad, mint ha a problémát elsunnyogva folytatná a futást.

A baleseteket tároló objektumokat egy `IncidentCollection` osztályú objektumban gyűjtjük. Az osztály a `dict` osztály leszármazottja, amelytől sok mindent örököl: például az elemelérési műveletet (`[]`), amelyet az egyes balesetek kiolvasására, beállítására és törlésére fogunk használni. Íme a `class` utasítást tartalmazó sor és az osztály néhány tagfüggvénye:

```
class IncidentCollection(dict):
    def values(self):
        for report_id in self.keys():
            yield self[report_id]

    def items(self):
        for report_id in self.keys():
            yield (report_id, self[report_id])

    def __iter__(self):
        for report_id in sorted(super().keys()):
            yield report_id

keys = __iter__
```

Az előkészítő újbóli megvalósítását nem kellett elvégeznünk, hiszen a `dict.__init__()` megfelelő. A kulcsok a jelentésazonosítók (`report_id`), az értékek pedig `Incident` osztályú objektumok. A `values()`, az `items()` és a `keys()` tagfüggvény újbóli megvalósítására azért volt szükség, hogy a bejárásuk a jelentésazonosítók sorrendjében történjen. A megoldás azért megfelelő, mert a `values()` és az `items()` tagfüggvény az `IncidentCollection.keys()` tagfüggvénytől visszakapott kulcsokat járja be, amely viszont a bejárást rendezett sorrendben, az ősosztály `dict.keys()` tagfüggvényétől visszakapott értékek alapján végzi. (Az `IncidentCollection.keys()` egyébiránt nem más, mint az `IncidentCollection.__iter__()` tagfüggvény másik neve.)

Az eddigieken felül az `IncidentCollection` osztály rendelkezik egy `export()` és egy `import_()` tagfüggvénnyel. (A név és a zárójel között álló aláhúzásjellel a beépített `import` utasítás nevétől szándékszunk eltérni.) Az `export()` tagfüggvénynek egy fájlnevet adunk át, elhagyható paraméterként pedig egy írási módszer megjelölését, illetve egy tömörítésre utasító jelzést. A tagfüggvény az írási módszer és a fájl kiterjesztése alapján a munkát egy pontosabban meghatározott tagfüggvénynek, például az `export_xml_dom()` vagy az `export_xml_etree()` nevűnek adja át. Az `import_()` tagfüggvény egy fájlnevet és elhagyhatóan egy olvasási módot kap paraméterül, egyebekben a párjához hasonlóan működik. Azok a betöltést végző tagfüggvények, amelyek bináris adatot olvasnak be, nem kapnak jelzést a bemeneti állomány tömörítettségére vonatkozólag – egyedül kell felmérniük a helyzetet, a munkát pedig ennek megfelelően kell elvégezniük.

Bináris adatok írása és olvasása

A bináris formátumban tárolt adatok rendszerint még tömörítés nélkül is a legkevesebb helyet foglalják; rendszerint az írásuk és az olvasásuk is a leggyorsabb. A legegyszerűbb, ha befőtteket használunk, bár a kézzel feldolgozott adatok elvileg kevesebb helyet foglalnak.

Befőttek tömörítéssel vagy anélkül

Figyelem!

Python nyelvű programjainkban az adatok mentésére és betöltésére a befőttek (pickle) kínálják a legegyszerűbb módszert, de ahogy azt már az előző fejezetben is jeleztük, a befőttek semmiféle titkosítással nem rendelkeznek: nincs titkosító algoritmus, nincs digitális aláírás. Így az ismeretlen eredetű befőttek veszélyeket rejthetnek magukban. A veszély forrását az jelenti, hogy a befőttek képesek modulok betöltésére és függvények meghívására, vagyis kaphatunk olyan módon kezelt befőttet, amely miatt a parancsértelmező valamilyen káros kódot hajt végre a befőtt betöltésekor. Mindemellett a befőttek igen alkalmasak ideiglenes, egyszer használatos adatok kezelésére, különösen a személyes használatra készült programokban.

Fájlformátumok kialakításakor rendszerint egyszerűbb a betöltést végző kód megírása előtt megírni a mentést végző kódot. Így először azt tekintjük át, hogy miként menthetők a repülőgép-balesetek egy befőttbe.

```
def export_pickle(self, filename, compress=False):
    fh = None
    try:
        if compress:
            fh = gzip.open(filename, "wb")
        else:
            fh = open(filename, "wb")
            pickle.dump(self, fh, pickle.HIGHEST_PROTOCOL)
            return True
        except (EnvironmentError, pickle.PicklingError) as err:
            print("{0}: export error: {1}".format(
                    os.path.basename(sys.argv[0]), err))
            return False
        finally:
            if fh is not None:
                fh.close()
```

Ha a felhasználó kérte a fájl tömörítését, a `gzip` modul `gzip.open()` függvényének használatával nyitjuk meg a fájlt, egyébként maradunk a beépített `open()` függvénynél. Ha az adatok befőzése bináris formátumba történik, a `wb` („write binary”, azaz bináris írás) módot kell választanunk. (A Python 3.0-ban a `pickle.HIGHEST_PROTOCOL` a 3-as protokoll, ami egy tömör bináris befőttformátum.*)

* A 3-as protokollt csak a Python 3-ban találjuk meg. Ha azt szeretnénk, hogy a befőttjeinket a Python 2 és a Python 3 egyaránt be tudja olvasni, a 2-es protokollt kell használnunk.

Hibakezelési stratégiául azt választottuk, hogy a hiba felmerülésekor azonnal jelzünk a felhasználónak, a hívónak pedig a sikert vagy hibát jelző logikai változót adunk át. A `finally` utasítással kezdődő blokkal bizonyosodunk meg róla, hogy a műveleteket követően a fájl lezárul – attól függetlenül, hogy volt-e hiba. A 8. fejezetben egy tömörebb formát ismerünk meg a fájlok lezárásának ellenőrzésére, olyat, ami szükségtelenné teszi a `finally` blokk használatát.

Környezetkezelők ☞ 375

A fenti kód igen hasonló az előző fejezetben megismerthez, de van egy aprócska, mégis figyelemreméltó különbség. A befőttben lévő adat szótár típusú. A szótár értékei azonban `Incident` osztályú objektumok, vagyis általunk létrehozott osztályból származnak. A befőttek kezelését végző `pickle` modul elég okos ahhoz, hogy a legtöbb saját készítésű osztály objektumait is anélkül tudja menteni, hogy nekünk bármit kellene tennünk az ügy érdekében.

A befőttek általában logikai, szám és karakterlánc típusú változókat tartalmazhatnak, valamint osztályokból példányosított objektumokat – akár egyedi osztályból származókat is –, amennyiben az adott osztály objektumainak privát `__dict__` változója megfelelő befőttek készítéséhez. Befőtt ezen felül bármely beépített gyűjteményes adattípus változóiból – rendezett sorozatokból, listákból, halmazokból, szótárakból – készíthető, amennyiben bennük csak befőzhető adattípusból származó objektumok vannak. Ez érvényes a bennük lévő gyűjteményes adattípusokra is, vagyis a rekurzív szerkezetek is lehetnek befőzhetők. Lehetőség van továbbá arra is, hogy olyan objektumokat (olyan egyedi osztályok példányait) is befőttbe tegyük, amelyek rendesen nem kerülhetnek befőttbe – például azért, mert valamelyik jellemzőjük megakadályozza. Ilyen esetekben vagy némi segítséget nyújtunk a `pickle` modulnak, vagy saját függvényeket valósítunk meg a befőttbe helyezésre és az abból való kivételre. A `pickle` modul leírásában minden ide vonatkozó részlet megtalálható.

__dict__ ☞ 369

Ha a befőttbe helyezett adatokat vissza szeretnénk olvasni, észre kell vennünk a különbséget a tömörített és tömörítetlen befőttek között. Minden olyan fájl, amelyet a `gzip` tömörítési eljárással tömörítettünk, egy bizonyos *mágikus számmal* kezdődik. A mágikus számok olyan, egy vagy több bájt hosszúságú jelsorozatok, amelyek az állományok elején találhatók, és az állomány típusát jelzik. A `gzip` tömörítésű fájlok esetében ez a mágikus szám a `0x1F 0x8B` bájtpár, amelyet egy `bytes` típusú változóban tárolunk:

```
GZIP_MAGIC = b"\x1F\x8B"
```

A `bytes` adattípusról részletesebb információkat kicsit később, *A bytes és a bytearray adattípus* című keretes szövegben, illetve a típusok tagfüggvényeit felsoroló 7.1., 7.2. és 7.3. táblázatokban találunk.

Íme a kód, amely az `Incident` objektumokból készült befőtt beolvasását végzi:

```
def import_pickle(self, filename):
    fh = None
    try:
        fh = open(filename, "rb")
        magic = fh.read(len(GZIP_MAGIC))
        if magic == GZIP_MAGIC:
            fh.close()
            fh = gzip.open(filename, "rb")
        else:
            fh.seek(0)
        self.clear()
        self.update(pickle.load(fh))
        return True
    except (EnvironmentError, pickle.UnpicklingError) as err:
        print("{0}: import error: {1}".format(
                os.path.basename(sys.argv[0]), err))
        return False
    finally:
        if fh is not None:
            fh.close()
```

Nem tudjuk, hogy az átadott fájl tömörített-e. Mindkét lehetséges esetben a fájlt `rb` („read binary", azaz bináris olvasás) módban nyitjuk meg először, és beolvassuk az első két bájtot. Ha a két bájt megegyezik a `gzip` tömörítést jelölő két bájttal, lezárjuk a fájlt, és a `gzip.open()` függvénnyel új objektumot hozunk létre. Ha tömörítetlen fájllal állunk szemben, az `open()` függvénytől visszakapott fájlobjektumot használjuk, amelynek a `seek()` tagfüggvényét hívva visszaállítjuk a fájlmutatót az állomány elejére. Így a következő olvasás – amit a `pickle.load()` függvény végez majd – jó helyről indulhat.

A `self` objektumnak nem adhatunk értéket, mert azzal a használatban lévő `IncidentCollection` osztályú objektumot megszüntetnénk. Így inkább az `Incident` objektumok eltakarításával kiürítjük a szótárt, majd a `dict.update()` tagfüggvény segítségével a befőttből betöltött, `IncidentCollection` osztályú szótárban tárolt `Incident` objektumokkal töltjük fel ismét.

Megjegyeznénk, hogy semmi jelentősége sincs, hogy a processzor milyen bájtsorrendet alkalmaz: a mágikus számot bájtonként olvassuk be, az adatok esetében a bájtsorrenddel a `pickle` modul törődik helyettünk.

Nyers bináris adatok tömörítéssel és anélkül

A nyers bináris adatok kezelését végző saját kód megírása a fájlformátumunkat illetőleg teljes irányítást ad a kezünkbe. Akár biztonságosabb is lehet a befőttek használatán alapuló módszernél, hiszen a rossz szándékkal megváltoztatott adatokat ezúttal a saját kódunk kezeli, és nem a parancsértelmező hajtja végre.

Amikor saját bináris fájlformátumot hozunk létre, a fájltípus azonosítására bölcs dolog mágikus számot is létrehozni, a használt fájlformátum-változat azonosítása pedig a fájlban elhelyezett változatszámmal oldható meg. Íme a `convert-incidents.py` programban használt számok:

```
MAGIC = b"AIB\x00"
FORMAT_VERSION = b"\x00\x01"
```

Négy bájton tároljuk a mágikus számot, kettőn a fájlformátum változatszámát. Az, hogy milyen bájtsorrendet használunk, nem érdekes, mert az adatot bájtonként fogjuk írni, nem pedig egész számok bájtábrázolásaként – így minden processzor esetében azonos lesz a leírt adat.

Ha nyers bináris adatot szeretnénk írni és olvasni, valamiféle átalakítást kell elvégeznünk a Python-objektum, illetve az objektum megfelelő bináris ábrázolása között. Amire szükségünk van, annak a legnagyobb része megtalálható a `struct` modulban (a modul rövid leírását a kicsit később következő, *A struct modul* című keretes szövegben találjuk), illetve a `bytes` és a `bytearray` adattípus biztosítja (ezeket röviden *A bytes és a bytearray adattípus* című keretes szöveg ismerteti).

Sajnos a `struct` modul csak egyforma hosszúságú karakterláncok kezelésére képes, nekünk pedig változó hosszúságú karakterláncokra van szükségünk, mind a jelentésazonosító, mind a repülőgép-azonosító esetében, nem különben a repülőtér, a repülőgéptípus és a kísérőszöveg tárolására. Ennek az igénynek a kielégítése végett létrehoztuk a `pack_string()` függvényt. A függvény paramétere egy karakterlánc, a visszatérési értéke pedig egy `bytes` típusú objektum, amely két összetevőt tartalmaz. Az első egy hosszúságot jelölő egész szám, a második pedig egy az előző számnak megfelelő hosszúságú, UTF-8 kódolású szöveg, amely az átadott karakterláncban lévő szöveget ábrázolja.

Helyi függvények
☞ 357

Minthogy az egyetlen olyan hely, ahol a `pack_string()` függvényre szükség van, az `export_binary()` függvény belsejében található, a `pack_string()` függvény megadása az `export_binary()` függvény belsejében történik. Ez annyit jelent, hogy a `pack_string()` függvény az `export_binary()` függvényen kívülről nem látható, és egyértelművé teszi, hogy csak egy helyi segédfüggvényről van szó. Íme az `export_binary()` függvény eleje, benne a teljes beágyazott `pack_string()` függvénnyel:

```
def export_binary(self, filename, compress=False):
    def pack_string(string):
        data = string.encode("utf8")
        format = "<H{0}s".format(len(data))
        return struct.pack(format, len(data), data)
```

A bytes és a bytearray adattípus

A Python két adattípussal segíti a nyers bájtok kezelését: az egyik az állandó értékű `bytes`, a másik a változtatható `bytearray` típus. Mindkét típus nulla vagy több, előjel nélküli, nyolc biten ábrázolható egész szám (azaz bájt) sorozatát tárolja. Minden bájt 0 és 255 közötti értéket vehet fel.

Mindkét típus nagyon hasonló a karakterláncokhoz, és sok a közös tagfüggvény, illetve lehetőség is – ilyen többek között a szeletelés. Ezen felül a `bytearray` típus rendelkezik néhány, a listákra jellemző tagfüggvénnyel. A típusok tagfüggvényeinek listáját a 7.1., 7.2. és 7.3. táblázatban találjuk.

Míg egy `bytes` vagy `bytearray` egy szelete a kiindulóval megegyező típusú objektumot ad vissza, az elemelérési művelet (`[]`) egy egész értéket, amely a megadott bájt értékét hordozza. Például:

```
word = b"Animal"
x = b"A"
word[0] == x      # returns: False  # word[0] == 65;     x == b"A"
word[:1] == x     # returns: True   # word[:1] == b"A"; x == b"A"
word[0] == x[0]   # returns: True   # word[0] == 65;     x[0] == 65
```

Íme néhány további `bytes`, illetve `bytearray` típusú adat:

```
data = b"5 Hills \x35\x20\x48\x69\x6C\x6C\x73"
data.upper() # az eredmény: b'5 HILLS 5 HILLS'
                                    # az eredmény: b'5 Hats 5 Hats'
data.replace(b"ill", b"at")
bytes.fromhex("35 20 48 69 6C 6C 73")  # az eredmény: b'5 Hills'
bytes.fromhex("352048696C6C73")        # az eredmény: b'5 Hills'
data = bytearray(data)                 # az adat immár bytearray
data.pop(10)                           # az eredmény: 72 (ord("H"))
data.insert(10, ord("B"))              # data == b'5 Hills 5 Bills'
```

Azok a tagfüggvények, amelyek csak karakterláncok esetében értelmezhetők – ilyen például a `bytes.upper()` –, feltételezik, hogy ASCII kódolású szöveget tárolunk a változóban. A `bytes.fromhex()` tagfüggvény figyelmen kívül hagyja a térközkaraktereket, és minden kétszámjegynyi karakterlánc-részletet tizenhatos számrendszerű számként értelmez. Vagyis a „35” egy `0x35` értékű bájt lesz, és így tovább.

Karakterkódolás
86 ☜

A `str.encode()` tagfüggvény egy `bytes` objektumot ad vissza, benne a megadott karakterkódolás szerint kódolt karakterlánccal. Az UTF-8 karakterkódolás igen kényelmes, ugyanis minden Unicode-karakter tárolására képes, az ASCII-karakterek tárolásakor pedig különösen tömör: mindegyiket egyetlen bájton ábrázolja. A `format` nevű változó értéke egy a `struct` modulban használ-

ható formátumot tárol, amelyet a karakterlánc hossza alapján határozunk meg. Például, ha az „en.wikipedia.org” karakterláncot tekintjük, a `format` változó értéke „`<H16s`” lesz („littleendian” bájtsorrend, kétbájtos előjel nélküli egész, 16 bájtos karakterlánc), a visszaadott objektum pedig a `b'\x10\x00en.wikipedia.org'` értéket fogja hordozni. Igen kényelmes dolog, hogy a Python a `bytes` típusú objektumokat tömör formában, ahol lehet, nyomtatható ASCII-karakterek használatával, ahol pedig nem, ott levédett tizenhatos számrendszerű értékekkel (és pár megkülönböztetett szerepű védőkarakterrel, mint a `\t` és a `\n`) mutatja meg.

A struct modul

A `struct` modulban találjuk többek között a `struct.pack()` és a `struct.unpack()` függvényt, és természetesen a `struct.Struct()` osztályt. A `struct.pack()` függvény paramétere egy a `struct` modulra jellemző, formátumleíró karakterlánc, illetve egy vagy több érték. Visszatérési értéke egy olyan `bytes` objektum, amely az átadott értékeket a megadott karakterlánc szerint formázva tartalmazza. A `struct.unpack()` függvény paramétere egy formátumleírás, és egy `bytes` vagy egy `bytearray` objektum, visszatérési értéke pedig egy olyan rendezett sorozat, amely azokat az értékeket tartalmazza, amelyeket a formátum használatával valamikor régebben becsomagoltunk.

Például:

```
data = struct.pack("<2h", 11, -9)  # data == b'\x0b\x00\xf7\xff'
items = struct.unpack("<2h", data) # items == (11, -9)
```

A formátumleíró karakterláncok egy vagy több karakterből állnak. A legtöbb karakter egy adott típusú értéket képvisel. Ha egy típusból több értékre is szükségünk van, akkor vagy annyiszor írjuk be a típusnak megfelelő karaktert, ahány értékünk van („`hh`”), vagy a karakter előtt álló számmal jelezzük a kívánt mennyiséget – ahogy az előző példában látható („`2h`”).

A `struct` modul leírásában sok formátumleíró karaktert találunk. A „b” a nyolc bites előjeles egész számokat jelöli, a „B” a nyolc bites előjel nélküli egész számok jele, a „h” a tizenhat bites előjeles egészek jele (a fenti példákban is ezt használtuk), a „H” a tizenhat bites előjel nélküli egészeket jelöli. Az „i” jelöli a harminckét bites előjeles egészeket, az „I” pedig a harminckét bites előjel nélküli egészeket, a „q” jelöli a hatvannégy bites előjeles egészeket, a „Q” pedig a hatvannégy bites előjel nélküli egészeket. A harminckét bites lebegőpontos számok jele az „f”, a hatvannégy biteseké – a Python nyelv `float` típusának megfelelően a „d”. A kérdőjel a logikai értékeket jelöli, az „s” a `bytes`, illetve a `bytearray` típus jele, és még igen sok jel létezik.

Néhány adattípus esetében – például a több bájtos egészeknél – a processzor által használt kódolás hatással van a bájtsorrendre. Kikényszeríthetjük egy bizonyos kódolás használatát, ha a formátumleírót egy ennek megfelelő karakterrel kezdjük. Könyvünkben mindig

a „kisebb mint” jel, a „little-endian” kódolás jele szerepel. Azért éppen ez, mert a széles körben használt Intel és AMD processzorok is ezt használják. A „big-endian” kódolás úgy tetszik: hálózati bájtsorrend – jele a „nagyobb mint” jel vagy a felkiáltójel. Ha a kódolást nem adjuk meg, a Python a számítógépnek megfelelő kódolást használja. Érdemes az endian-kódolást mindig megadnunk, hiszen így hordozható adatokat kapunk eredményül.

A `struct.calcsize()` függvény paramétere egy formátumleíró, visszatérési értéke pedig egy szám: ennyi bájtot foglal el az adott formátumleíró alapján formázott adatszerkezet.

A formátumleírók tárolása történhet `struct.Struct()` osztályú objektumokban, amelyeknek paraméterül a formátumleírót adjuk át. Az objektum méretét a `size` jellemző adja meg. Például:

```
TWO_SHORTS = struct.Struct("<2h")
data = TWO_SHORTS.pack(11, -9)      # data == b'\x0b\x00\xf7\xff'
items = TWO_SHORTS.unpack(data)     # items == (11, -9)
```

A `11` mindkét példában `0x000b`, de a little-endian bájtsorrendűvé alakítás miatt ez `0x0b 0x00` formában jelenik meg.

7.1. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (1)*

Utasításforma	Leírás
ba`.append(i)`	Az *i* 0 és 255 közötti egész számot a *ba* `bytearray` típusú változó végére fűzi.
b`.capitalize()`	A *b* `bytes/bytearray` típusú változó értékét adja vissza, az első karaktert nagybetűre cserélve (már amennyiben az egy ASCII kódolású betű).
b`.center(`*szélesség*`,` *bájt*`)`	A *b* változó szóközökkel vagy a *bájt* értékkel *szélesség* szélességűre kiegészített, középre igazított másolatát adja vissza.
b`.count(`*x*`,` *kezdet*`,` *vég*`)`	Az *x* `bytes/bytearray` típusú érték előfordulásainak számát adja vissza a *b* `bytes/bytearray` típusú változóban (vagy a *b* változó *kezdet*`:`*vég* részében).
b`.decode(`*kódolás*`,` *hiba*`)`	Olyan `str` osztályú objektumot ad vissza, amely a `bytes` típusú adat UTF-8, illetve a *kódolás* értékkel megadott kódolású ábrázolását tartalmazza. A hibák kezelése az elhagyható *hiba* paraméternek megfelelően történik. Karakterkódolás 86
b`.endswith(`*x*`,` *kezdet*`,` *vég*`)`	`True` értéket ad vissza, ha a *b* változó (vagy annak *kezdet*`:`*vég* része) az *x* `bytes/bytearray` típusú értékkel, vagy az *x* rendezett sorozatban lévő bármely `bytes/bytearray` típusú értékkel végződik; egyéb esetekben `False` a visszaadott érték.

7.1. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (1)* *(folytatás)*

Utasításforma	Leírás
b.expandtabs(*méret*)	A *b* bytes/bytearray típusú változó másolatát adja vissza, amelyben a tabulátorokat 8, illetve – ha megadtuk – a *méret* többszöröseinek megfelelő számú szóközre cseréli.
ba.extend(*sorozat*)	A *ba* bytearray típusú változót a *sorozat*-ban lévő egész számokkal egészíti ki; a számok 0 és 255 közöttiek lehetnek.
b.find(*x*, *kezdet*, *vég*)	A *b* változóban (vagy annak *kezdet*:*vég* részében) az *x* bytes/bytearray típusú érték bal szélső előfordulását adja vissza, illetve -1-et, ha *x* nem fordul elő *b*-ben. A jobb szélső előfordulás megtalálása az rfind() tagfüggvénnyel lehetséges.
b.fromhex(*h*)	A *h* karakterláncban lévő tizenhatos számrendszerű egész számoknak megfelelő bytes osztályú objektumot ad vissza.
b.index(*x*, *kezdet*, *vég*)	A *b* változóban (vagy annak *kezdet*:*vég* részében) az *x* érték bal szélső előfordulását adja vissza, illetve ValueError kivételt ad, ha *x* nem fordul elő *b*-ben. A jobb szélső előfordulás megtalálása az rindex() tagfüggvénnyel lehetséges.
ba.insert(*p*, *i*)	A *ba* változóba *p* helyen beszúrja az *i* egész számot, amelynek az értéke 0 és 255 közötti lehet.
b.isalnum()	True értéket ad vissza, ha a *b* bytes/bytearray típusú változó nem üres, és minden karaktere ASCII-szám vagy -betű.
b.isalpha()	True értéket ad vissza, ha a *b* bytes/bytearray típusú változó nem üres, és minden karaktere ASCII-betű.
b.isdigit()	True értéket ad vissza, ha a *b* bytes/bytearray típusú változó nem üres, és minden karaktere ASCII-számjegy.
b.islower()	True értéket ad vissza, ha a *b* bytes/bytearray típusú változó nem üres, van legalább egy ASCII-betű karaktere, és minden betűkaraktere kisbetű.
b.isspace()	True értéket ad vissza, ha a *b* bytes/bytearray típusú változó nem üres, és minden karaktere ASCII-szóközt jelölő karakter.

7.2. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (2)*

Utasításforma	Leírás
b.istitle()	True értéket ad vissza, ha a *b* változó nem üres, és az angol címeknek megfelelően nagy kezdőbetűs szavakból áll.
b.isupper()	True értéket ad vissza, ha a *b* változóban legalább egy ASCII-betű van, és minden betű karaktere nagybetű.
b.join(*sorozat*)	Összefűzve visszaadja a *sorozat* sorozatban lévő bytes/bytearray típusú változókat, az egyes változók értékei közé beékelve *b*-t, amely üres is lehet.

7.2. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (2)* *(folytatás)*

Utasításforma	Leírás
`b.ljust(szélesség, bájt)`	A `b` bytes/bytearray típusú változó szóközökkel vagy az elhagyható `bájt` értékkel `szélesség` szélességűre kiegészített, balra rendezett másolatát adja vissza. Jobbra rendezéshez az `rjust()` tagfüggvény használható.
`b.lower()`	A `b` bytes/bytearray típusú változó ASCII szerinti kisbetűs másolatát adja vissza.
`b.partition(elválasztó)`	Három bytes típusú változóból álló rendezett sorozatot ad vissza. Az első elem a `b` bytes/bytearray típusú változónak a változóban lévő bal szélső `elválasztó` előtti része, ezt követi maga az `elválasztó`, majd `b` `elválasztó` utáni része következik. Ha `b`-nek az `elválasztó` nem része, akkor `b`-t és két bytes típusú üres objektumot kapunk vissza. Ha az `elválasztó` jobb szélső előfordulásánál szeretnénk tagolni `b`-t, használjuk az `rpartition()` tagfüggvényt.
`ba.pop(p)`	A `p` helyen álló egész számot kiveszi a `ba` változóból, és visszaadja a hívónak.
`ba.remove(i)`	A `ba` bytearray típusú változóból eltávolítja az `i` egész szám első előfordulását.
`b.replace(x, y, n)`	A `b` változó másolatát adja vissza, benne az `x` bytes/bytearray típusú érték minden (ha `n` adott, akkor az első `n`) előfordulását `y`-ra cserélve.
`ba.reverse()`	A `ba` bytearray típusú változót helyben megfordítja.
`b.split(x, n)`	`x` előfordulásainál szeli `b`-t, legfeljebb `n` alkalommal. Ha `x` nem adott, akkor a szóközöknél szel, ha `n` nem adott, akkor minden lehetséges helyen. Visszatérési értéke egy bytes objektumokból álló lista. Jobbról kezdődő szeleteléshez az `rsplit()` tagfüggvényt kell használnunk.
`b.splitlines(f)`	A `b` változó sorvégeknél történt szeletelésével képzett sorokból álló listát adja vissza. A sor végének jelzését elhagyja, kivéve, ha `f` értéke `True`.
`b.startswith(x, kezdet, vég)`	`True` értékkel tér vissza, ha a `b` bytes/bytearray típusú változó (vagy annak `kezdet:vég` közötti része) az `x` bytes/bytearray típusú értékkel, vagy az `x` rendezett sorozatban lévő bármely bytes/bytearray típusú értékkel kezdődik, egyéb esetekben `False` a visszaadott érték.
`b.strip(x)`	`b` olyan másolatát adja vissza, amelynek az elejéről és végéről a szóközöket (illetve az `x` bytes/bytearray típusú értéket) eltávolította. Az `lstrip()` tagfüggvény csak a paraméter elejénél, az `rstrip()` tagfüggvény csak a paraméter végénél vág.

7.2. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (2)* *(folytatás)*

Utasításforma	Leírás
`b.swapcase()`	`b` olyan másolatát adja vissza, amelyben a kisbetűs ASCII-karakterek nagybetűsek lesznek, és fordítva.
`b.title()`	`b` olyan másolatát adja vissza, amelyben minden egyes szó első ASCII-betűje nagybetűs, az összes többi pedig kisbetűs.
`b.translate(bt, d)`	`b` olyan másolatát adja vissza, amelyben nincs `d`-ben előforduló bájt, és amelyben miden egyéb bájt a `bt` `bytes` típusú érték „bájtadik” bájtjára van cserélve.

7.3. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (3)*

Utasításforma	Leírás
`b.upper()`	A `b` `bytes/bytearray` típusú változó ASCII szerinti nagybetűs másolatát adja vissza.
`b.zfill(szélesség)`	`b` olyan másolatát adja vissza, amelynek az elejére annyi nullát (`0x30` kódú karaktert) írunk, hogy *`szélesség`* szélességűvé váljon.

A `pack_string()` tagfüggvény akár 65 535 UFT-8 kódolású karakter kezelésére képes. Ami a bájtok számát illeti, könnyedén választhatunk más egész típust – például a négybájtos előjeles egész számokkal („i” formátum) akár 2^{31}-1 (kétmilliárdnál is nagyobb) hosszúságú karakterláncokat is használhatunk.

A `struct` modulban találunk egy hasonló beépített formátumot, a „p”-t, amely egy bájtot karakterszámként tárol, és ezt 255 további karakter követheti. A becsomagolásnál a „p” formátumot használó kód lényegesen egyszerűbb megoldást jelent, mintha minden munkát magunknak kellene elvégeznünk. A „p” formátum azonban legfeljebb 255 UTF-8 kódolású karakter használatát teszi lehetővé, és a kicsomagolás során alig jelent valami előnyt. (Az összehasonlítás kedvéért a `convert-incidents.py` fájlban megtaláljuk a `pack_string()` és az `unpack_string()` függvény „p” formátumot használó változatát is.

Figyelmünket immáron a kód hátralévő részére, az `export_binary()` tagfüggvényre fordíthatjuk.

```
fh = None
try:
    if compress:
        fh = gzip.open(filename, "wb")
    else:
        fh = open(filename, "wb")
    fh.write(MAGIC)
    fh.write(FORMAT_VERSION)
```

```
        for incident in self.values():
            data = bytearray()
            data.extend(pack_string(incident.report_id))
            data.extend(pack_string(incident.airport))
            data.extend(pack_string(incident.aircraft_id))
            data.extend(pack_string(incident.aircraft_type))
            data.extend(pack_string(incident.narrative.strip()))
            data.extend(NumbersStruct.pack(
                                incident.date.toordinal(),
                                incident.pilot_percent_hours_on_type,
                                incident.pilot_total_hours,
                                incident.midair))
            fh.write(data)
        return True
```

Az `except` és a `finally` kódblokkokat elhagytuk, mert az `except` blokkban elkapott kivételektől eltekintve megegyeznek az előző szakaszban lévőkkel.

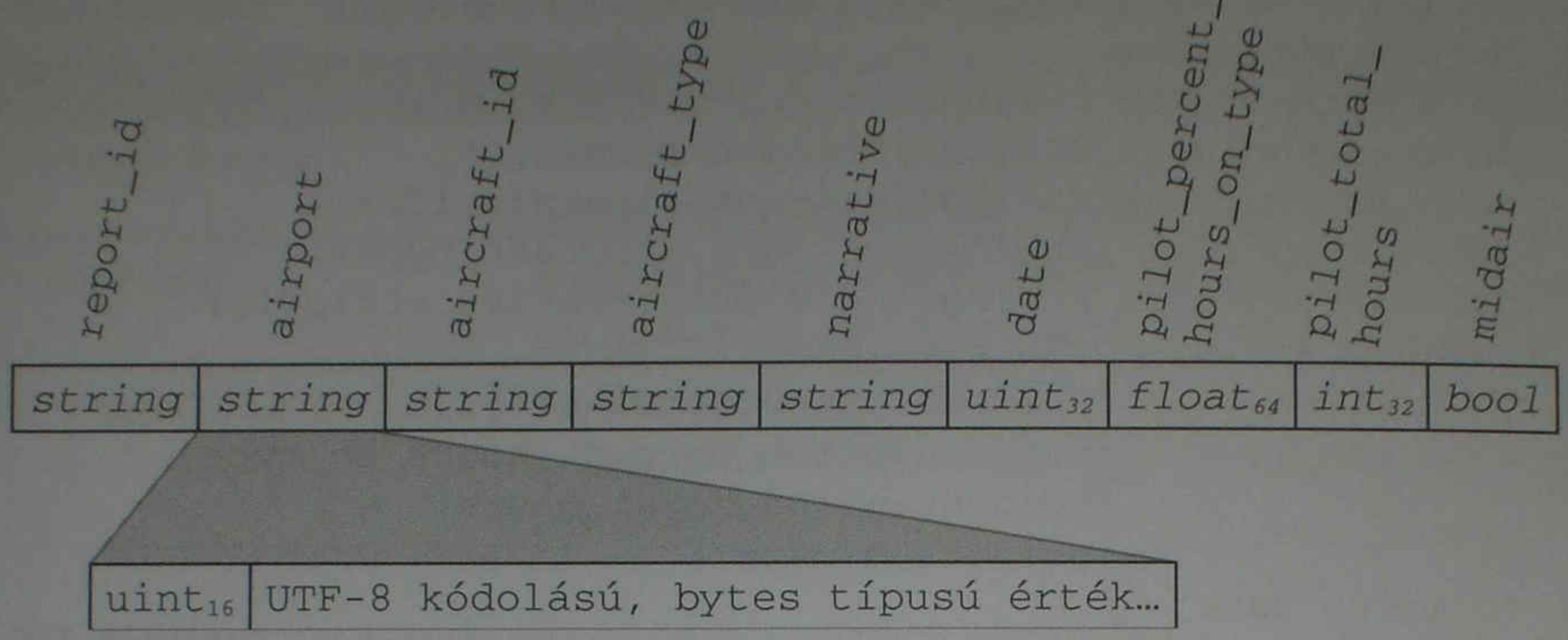
A fájlt a tagfüggvény elején `wb` („write binary", azaz bináris írás) módban nyitjuk meg, a `compress` változóban tárolt értéktől függően mint rendes vagy mint `gzip`-pel tömörített állományt. Ezek után leírjuk azt a négybájtos mágikus számot, amely – remélhetőleg – csak a mi programunkra jellemző, amit a kétbájtos változatszám követ.* A változatszám használata megkönnyíti a formátum megváltoztatását a jövőben – a változatszám beolvasásával meghatározható, hogy a beolvasáskor melyik kódot használjuk.

Az `Incident` objektumok bejárásával folytatjuk, és minden példány számára létrehozunk egy `bytearray` típusú változót. A változóban minden adatelemet változó hosszúságú karakterláncként helyezünk el. A `date.toordinal()` tagfüggvény egyetlen, a tárolt dátumot ábrázoló egész számot ad vissza, amelyet a `datetime.date.fromordinal()` tagfüggvény tud rendes dátummá visszaalakítani a számból. A `NumbersStruct` objektumot a program korábbi részében, az alábbi utasítással adtuk meg:

```
NumbersStruct = struct.Struct("<Idi?")
```

A formátumleíró little-endian bájtsorrendet ad meg, egy előjel nélküli harminckét bites egész számot (az egész számmá alakított dátum tárolására), egy hatvannégy bites lebegőpontos számot (a pilóta adott repülőgép-típuson repült idejére vonatkozó százalékos arány tárolására), egy harminckét bites egész számot (a teljes repült óraszám tárolására) és egy logikai értéket (ami azt mutatja, hogy a baleset a levegőben történt-e). A repülőgép-baleset tárolására szolgáló adatbázisrekord sematikus rajzát a 7.3. ábrán láthatjuk.

* A mágikus számoknak nincs olyan központi nyilvántartásuk, mint például az internetes tartományneveknek – így az egyediséget nem biztosítja semmi.



7.3. ábra

A repülőgép-baleset tárolására szolgáló bináris adatbázisrekord felépítése

Ha a `bytearray` típusú objektumba minden, a balesettel kapcsolatos adat bekerült, az objektumot lemezre írjuk. Amikor pedig az összes baleset a lemezen van, `True` értéket adunk vissza (már amennyiben nem volt hiba). A visszatérés előtt közvetlenül még a `finally` kódblokkal megbizonyosodunk arról, hogy a fájl le van-e zárva.

Az adatok visszaolvasása nem annyira egyszerű, mint a lemezre írásuk – legalábbis egy dolog miatt további hibaellenőrzést kell végeznünk. A változó hosszúságú karakterláncok visszaolvasása szintén rejt érdekességeket magában. Íme az `import_binary()` tagfüggvény eleje, benne a teljes beágyazott `unpack_string()` függvénnyel, amely a változó hosszúságú karakterláncok visszaolvasását végzi:

```
def import_binary(self, filename):
    def unpack_string(fh, eof_is_error=True):
        uint16 = struct.Struct("<H")
        length_data = fh.read(uint16.size)
        if not length_data:
            if eof_is_error:
                raise ValueError("missing or corrupt string size")
            return None
        length = uint16.unpack(length_data)[0]
        if length == 0:
            return ""
        data = fh.read(length)
        if not data or len(data) != length:
            raise ValueError("missing or corrupt string")
        format = "<{0}s".format(length)
        return struct.unpack(format, data)[0].decode("utf8")
```

Minthogy minden baleset rekordja a jelentésazonosítót tároló karakterlánccal kezdődik, amikor megpróbálkozunk a karakterlánc beolvasásával, és az sikerül, azonnal egy új rekord elejére kerülünk. Ha pedig problémába ütközünk, akkor elértük a fájl végét, és el is készül-

tünk. Amikor egy jelentésazonosító beolvasásával próbálkozunk, az `eof_is_error` (a fájlvége-jel hibát jelent) értékét `False`-ra állítjuk, hiszen ha nem kapunk adatot, az csak annyit tesz, hogy befejezhetjük a beolvasást. Minden egyéb karakterlánc esetében elfogadjuk az alapértelmezett `True` beállítást, hiszen ha egy másik karakterláncban nincs adat, az baj. (Még az üres karakterláncok előtt is ott van a tizenhat bites előjel nélküli, a karakterlánc hosszát jelző szám.)

Elsőként a karakterlánc hosszát jelző számot próbáljuk meg beolvasni. Ha új balesetet szeretnénk beolvasni, és a hosszúság beolvasása sikertelen, a `None` értéket adjuk vissza, ezzel jelezve a fájl végét. Egyéb esetekben `VauleError` kivételt váltunk ki, ami most a hibás vagy hiányzó adat jele. A `struct.unpack()` tagfüggvény mindig egy rendezett sorozatot ad vissza, akkor is, ha csak egyetlen érték van benne. Kicsomagoljuk a hosszúságra vonatkozó adatot, és a benne ábrázolt számot a `length` változóban tároljuk. Mostanra tudjuk, hogy mennyi bájt beolvasása szükséges a karakterlánc kinyeréséhez. Ha a `length` változó értéke nulla, egyszerűen üres karakterláncot adunk vissza. Egyéb esetekben megkíséreljük a megadott számú bájt beolvasását. Ha nem kapunk adatot, vagy nem annyit, amennyire számítunk, `VauleError` kivételt váltunk ki.

Ha megvan a kívánt számú bájt, a `struct.unpack()` tagfüggvény számára létrehozunk egy megfelelő formátumleíró karakterláncot, majd visszaadjuk az adat kicsomagolásából, és a bájtok UTF-8 kódolású karakterekként történt értelmezéséből származó karakterláncot. (Elméletben az utolsó két sort lecserélhetnénk a `return data.decode("utf8")` utasításra, de jobbnak tartjuk, ha végig figyelemmel kísérjük a kicsomagolás folyamatát. Lehetséges ugyanis – bár nem túl valószínű –, hogy az „s” formátumleíró úgy alakítja át az adatainkat, hogy beolvasáskor visszaalakítást kell végeznünk.)

Most pedig megnézzük az `import_binary()` tagfüggvény fennmaradó részét – a magyarázat megkönnyítése végett mindezt két részletben tesszük.

```
fh = None
try:
    fh = open(filename, "rb")
    magic = fh.read(len(GZIP_MAGIC))
    if magic == GZIP_MAGIC:
        fh.close()
        fh = gzip.open(filename, "rb")
    else:
        fh.seek(0)
    magic = fh.read(len(MAGIC))
    if magic != MAGIC:
        raise ValueError("invalid .aib file format")
    version = fh.read(len(FORMAT_VERSION))
    if version > FORMAT_VERSION:
        raise ValueError("unrecognized .aib file version")
    self.clear()
```

A fájl lehet tömörített és tömörítetlen is, így ugyanazzal a módszerrel döntjük el, hogy a `gzip.open()` vagy a beépített `open()` függvénnyel nyitjuk meg a fájlt, amit a befőttek beolvasásánál is használtunk.

Ha a fájl megnyitásával végeztünk, és az elején állunk, beolvassuk az első négy bájtot (`len(MAGIC)`). Ha ezek nem egyeznek meg a mágikus számunkkal, tudjuk, hogy nem bináris repülőgépbaleset-tároló adatfájllal van dolgunk, és `ValueError` kivételt váltunk ki. A műveletet a kétbájtos változatszám beolvasásával folytatjuk. Ezen a ponton kellene választanunk beolvasó kódot – a változatszám függvényében. Most azonban csak annyit ellenőrzünk, hogy a változat nem magasabb-e annál, amit a program beolvasni képes.

Ha a mágikus szám helyes, és a fájlváltozat is olyan, amit kezelni tudunk, akkor készen állunk az adatok beolvasására. A munkát a szótár kiürítésével, a már benne lévő balesetek törlésével kezdjük.

```
while True:
    report_id = unpack_string(fh, False)
    if report_id is None:
        break
    data = {}
    data["report_id"] = report_id
    for name in ("airport", "aircraft_id",
                 "aircraft_type", "narrative"):
        data[name] = unpack_string(fh)
    other_data = fh.read(NumbersStruct.size)
    numbers = NumbersStruct.unpack(other_data)
    data["date"] = datetime.date.fromordinal(numbers[0])
    data["pilot_percent_hours_on_type"] = numbers[1]
    data["pilot_total_hours"] = numbers[2]
    data["midair"] = numbers[3]
    incident = Incident(**data)
    self[incident.report_id] = incident
return True
```

A `while` ciklus addig ismétlődik, amíg ki nem fogyunk az adatokból. Először egy jelentés-azonosítót szeretnénk látni. Ha `None` értéket kapunk, az annyit tesz, hogy elértük a fájl végét – így kiléphetünk a ciklusból. Egyéb esetben létrehozunk egy `data` nevű szótárt, amely egy baleset adatainak tárolására fog szolgálni. Ezt követően megkíséreljük a baleset többi adatának beolvasását. A karakterlánc esetében az `unpack.string()` tagfüggvényt használjuk, a többi adatot pedig egy nekifutásra, a `NumbersStruct` nevű formátumleíró használatával olvassuk be. Minthogy az adatok tárolását átalakítás előzte meg, az adataink visszanyeréséhez most el kell végeznünk a visszaalakítást. A többi elem esetében azonnal használhatjuk a kicsomagolatlan adatokat – nincs szükség ellenőrzésre vagy átalakításra, mert annak idején a megfelelő adattípusok használatával vittük lemezre az adatainkat, most pedig ugyanazokat az adattípusokat olvassuk vissza a `NumbersStruct` nevű formátumleíróban tárolt formátum használatával.

Ha valamilyen hibába ütközünk, például nem sikerül az összes szám kicsomagolása, kivétel lép fel, amelyet majd az `except` kódblokkban kezelünk. (Az `except` és a `finally` kódblokk nem szerepel a könyvben, mivel a szerkezetüket tekintve megegyeznek az előző, `import_pickle()` tagfüggvényt bemutató szakaszban lévővel.)

Megfeleltetés-kicsomagoló művelet 175

A kód vége felé a kényelmes, megfeleltetés-kicsomagoló művelettel hozunk létre olyan `Incident` objektumokat, amelyek aztán bekerülnek a baleseteket tároló szótárba.

A változó hosszúságú karakterláncok kezelésétől eltekintve a `struct` modul igencsak megkönnyíti az adatok bináris formátumban való tárolását és visszatöltését. Ami pedig a változó hosszúságú karakterláncok kezelését illeti, az itt bemutatott `pack_string()` és `unpack_string()` tagfüggvény a legtöbb célra tökéletesen megfelelő.

Szövegfájlok írása és feldolgozása

A szövegek írása egyszerű, de a visszaolvasásuk problémás lehet, vagyis a felépítést úgy kell megválasztanunk, hogy ne legyen túlságosan bonyolult a szöveg feldolgozása. A 7.4. ábrán lévő példa a repülőgép-baleseteket tartalmazó adatbázisunk egy rekordját mutatja be abban a szöveges formátumban, amelyet használni fogunk. A baleseti adatbázis rekordjainak fájlba írásakor minden rekordot egy üres sor követ, bár a fájl feldolgozásakor azt is elfogadjuk, ha az egymás után álló baleseteket egyetlen sor sem választja el, és azt is, ha több elválasztó sor is van.

```
[20070927022009C]
date=2007-09-27
aircraft_id=1675B
aircraft_type=DHC-2-MK1
airport=MERLE K (MUDHOLE) SMITH
pilot_percent_hours_on_type=46.1538461538
pilot_total_hours=13000
midair=0
.NARRATIVE_START.
    ACCORDING TO THE PILOT, THE DRAG LINK FAILED DUE TO AN OVERSIZED
    TAIL WHEEL TIRE LANDING ON HARD SURFACE.
.NARRATIVE_END.
```

7.4. ábra

A repülőgép-baleseti adatbázis egy rekordjának szövege

Szövegírás

A baleseti adatbázis minden rekordja egy szögletes zárójelek (`[]`) közé zárt jelentésazonosítóval kezdődik. Ezt követik az adatelemek, amelyek mindegyike *kulcs=érték* alakú. A többsoros kísérőszöveg kezdetét a `.NARRATIVE_START.`, a végét pedig a `.NARRATIVE_END.` szöveg jelöli. A kezdő- és a végjel közötti szöveget behúzással is elkülönítjük, nehogy összekeverjük a jelzésekkel a szövegnek akár egyetlen sorát is.

Alább következik az `export_text()` függvény kódja, amelyből az `except` és a `finally` blokkot elhagytuk, ugyanis azok megegyeznek a korábban látott példákban lévőkkel, kivéve a kezelt kivételeket.

```
def export_text(self, filename):
    wrapper = textwrap.TextWrapper(initial_indent="  ",
                                   subsequent_indent="  ")
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        for incident in self.values():
            narrative = "\n".join(wrapper.wrap(
                                  incident.narrative.strip()))
            fh.write("[{0.report_id}]\n"
                     "date={0.date!s}\n"
                     "aircraft_id={0.aircraft_id}\n"
                     "aircraft_type={0.aircraft_type}\n"
                     "airport={airport}\n"
                     "pilot_percent_hours_on_type="
                     "{0.pilot_percent_hours_on_type}\n"
                     "pilot_total_hours={0.pilot_total_hours}\n"
                     "midair={0.midair:d}\n"
                     ".NARRATIVE_START.\n{narrative}\n"
                     ".NARRATIVE_END.\n\n".format(incident,
            airport=incident.airport.strip(),
            narrative=narrative))
    return True
```

A kísérőszövegben lévő sortörések a mi szempontunkból nem jelentősek, azaz a szöveget úgy tördeljük, ahogy kedvünk tartja. Rendes esetben a `textwrap` modul `textwrap.wrap()` függvényét használnánk, de ezúttal behúzásra és sortörésre is szükség van. Így először létrehozunk egy `textwrap.TextWrap` objektumot, mégpedig a használni kívánt behúzással: négy szóközzel minden sor előtt. Az objektum alapértelmezés szerint hetven karakter szélesség elérésénél töri a sorokat, ami azonban más kulcsszó-paraméter átadásával megváltoztatható.

`datetime` modul 212 ☜

Használhattunk volna háromszoros idézőjelben lévő szöveget, de jobb szeretjük saját kezűleg elhelyezni a sortöréseket. A `textwrap.TextWrap` objektumnak van egy `wrap()` nevű tagfüggvénye, amely egy karakterláncot –

str.format()
75

__format__()
253

esetünkben a kísérőszöveget – vár bemenetként, visszatérési értéke pedig egy olyan lista, amelynek az elemei a megfelelő behúzással ellátott, az adott szélességet túl nem lépő karakterláncok. Ezt követően a listában lévő sorokat egyetlen karakterlánccá egyesítjük, összekötőként újsor karaktert használva. A baleset dátumát `datetime.date` osztályú objektumként tároljuk; amikor az `str.format()` tagfüggvénnyel a dátumot leírtuk, kikényszerítettük a karakterláncként való ábrázolást – ezzel a dátumot az igen kényelmesen kezelhető, ISO 8601 szabvány szerinti ÉÉÉÉ-HH-NN formátumban kaptuk meg.

Az `str.format()` függvényt arra is megkértük, hogy a `midair` nevű logikai változót egész szám formájában adja meg – ez igaz értékek esetén 1-et, hamis értékek esetén pedig 0-át jelent. Általában véve az `str.format()` tagfüggvény használata igen kényelmessé teszi a szöveges adatok írását, ugyanis a tagfüggvény a Python nyelv összes adattípusát automatikusan kezeli – sőt az egyedi adattípusokat is, amennyiben megvalósítjuk az `__str__()` vagy a `__format__()` különleges szerepű tagfüggvények valamelyikét.

Szövegfeldolgozás

Az a tagfüggvény, amely a baleseti adatbázis rekordjainak beolvasását és feldolgozását végzi, hosszabb, és több ponton számíthat az érdeklődésünkre, mint az írásra szolgáló. A fájl olvasása alatt a programunk számos állapota különíthető el. Lehet, hogy a kísérőszöveg sorainak beolvasását végezzük, állhatunk egy *kulcs=érték* alakú sor előtt vagy éppen egy új baleset elejénél, a jelentésazonosítónál. Az `import_text_manual()` tagfüggvényt öt részletben tekintjük át.

```
def import_text_manual(self, filename):
    fh = None
    try:
        fh = open(filename, encoding="utf8")
        self.clear()
        data = {}
        narrative = None
```

A tagfüggvény azzal kezdődik, hogy a fájlt szövegolvasó módban megnyitjuk. Ez után kitakarítjuk a baleseteket tároló szótárat, majd létrehozzuk azt a `data` szótárat, amely egyetlen baleset adatait tudja tárolni – éppen úgy, ahogyan a bináris fájl beolvasásakor tettük. A `narrative` változónak két feladata van: állapotjelző, és az aktuális baleset kísérőszövegét is tartalmazza. Ha a `narrative` változó értéke `None`, az annyit jelent, hogy az adott pillanatban nem folyik semmilyen kísérőszöveg beolvasása. Ha az érték valamilyen karakterlánc – lehet üres karakterlánc is –, az viszont azt jelenti, hogy éppen a kísérőszöveg sorainak beolvasását végezzük.

```
for lino, line in enumerate(fh, start=1):
    line = line.rstrip()
    if not line and narrative is None:
        continue
    if narrative is not None:
        if line == ".NARRATIVE_END.":
```

```
                data["narrative"] = textwrap.dedent(narrative).strip()
                if len(data) != 9:
                    raise IncidentError("missing data on "
                                        "line {0}".format(lino))
                incident = Incident(**data)
                self[incident.report_id] = incident
                data = {}
                narrative = None
            else:
                narrative += line + "\n"
```

Minthogy a beolvasást soronként végezzük, mindig ismerjük az aktuális sor sorszámát, amelynek a segítségével a bináris állományok beolvasásakor lehetségesnél több információt nyújtó hibaüzenetet adhatunk. Először a sor végén lévő üreshely karaktereket vágjuk le, és ha ekkor üres sorunk marad – és nem éppen egy kísérőszöveg belsejében járunk –, egyszerűen továbblépünk a következő sorra. Ekként lehetséges az, hogy az egyes balesetek közötti üres sorok száma nem érdekes, ugyanakkor megőrizzük a kísérőszövegekben lévő üres sorokat.

Ha a `narrative` változó értéke nem `None`, hanem valami más, akkor tudjuk, hogy éppen egy kísérőszöveg belsejében járunk. Ha a sor a kísérőszöveg végét jelző karakterlánc, akkor tudjuk, hogy azon felül, hogy a kísérőszöveg feldolgozását befejeztük, befejeztük az aktuális baleset feldolgozását is. Ebben az esetben a kísérőszöveget elhelyezzük a `data` szótárban – persze csak miután a behúzásokat a `textwrap.dedent()` függvénnyel eltávolítottuk. Ha megvan mind a kilencféle szükséges adat, új objektumot példányosítunk az `Incidents` osztályból, és az elkészült objektumot elhelyezzük a szótárban. Ennek végeztével kiürítjük a `data` szótárat, és a `narrative` változó alaphelyzetbe állításával felkészülünk az újabb baleset fogadására. Ha a beolvasott sor nem a kísérőszöveg végét jelző karakterlánc, akkor hozzáfűzzük a `narrative` változó végéhez – azzal az újsor karakterrel együtt, amit az előbb levágtunk.

```
        elif (not data and line[0] == "["
                       and line[-1] == "]"):
            data["report_id"] = line[1:-1]
```

Ha a `narrative` változó értéke `None`, akkor vagy új jelentésazonosítóhoz érkeztünk, vagy valamilyen egyéb adat beolvasását végezzük. Új jelentésazonosítónál csak akkor lehettünk, ha a `data` szótár üres (üres, amikor a baleset beolvasását elkezdjük, és az egyes balesetek beolvasásának végeztével kiürítjük), a sor pedig nyitó szögletes zárójellel kezdődik, és záró szögletes zárójellel ér véget. Ilyenkor a jelentésazonosítót elhelyezzük a `data` szótárban. Mindez annyit tesz, hogy az `elif` ág feltétele nem válik igazzá egészen addig, amíg a `data` szótárat ismét ki nem ürítjük.

```
        elif "=" in line:
            key, value = line.split("=", 1)
            if key == "date":
                data[key] = datetime.datetime.strptime(value, "%Y-%m-%d").date()
```

```
        elif key == "pilot_percent_hours_on_type":
            data[key] = float(value)
        elif key == "pilot_total_hours":
            data[key] = int(value)
        elif key == "midair":
            data[key] = bool(int(value))
        else:
            data[key] = value
    elif line == ".NARRATIVE_START.":
        narrative = ""
    else:
        raise KeyError("parsing error on line {0}".format(lino))
```

Ha éppen nem járunk kísérőszöveg belsejében, és nem is egy jelentésazonosító beolvasását végezzük, akkor már csak három további helyzet lehetséges: vagy *kulcs=érték* párokat olvasunk be, vagy a kísérőszöveg kezdetét jelző karakterláncnál vagyunk, vagy valami nem egészen megfelelően alakult.

Ha *kulcs=érték* párokat olvasunk be, a sort az első egyenlőségjelnél vágjuk. Legfeljebb egy vágást engedélyezünk, hogy nyugodtan írhassunk egyenlőségjelet az *érték* belsejébe. Minden adatot Unicode-karakterláncként olvasunk be, vagyis dátumok, számok és logikai értékek esetén gondoskodnunk kell az érték megfelelő átalakításáról.

A dátumok esetében a `datetime.datetime.strptime()` függvényt használjuk (a `strptime` a „string parse time”, vagyis a „karakterlánccal értelmezett adat” rövidítése), amelynek paraméterül egy formátumleíró karakterláncot adunk át, a visszatérési értéke pedig egy `datetime.datetime` osztályú objektum. Olyan formátumleírót használtunk, ami az ISO 8601 dátumformátumnak tesz eleget; arra pedig, hogy a visszakapott `datetime.datetime` objektumból `datetime.date` objektumot tudjunk kinyerni, a `datetime.datetime.date()` tagfüggvényt használjuk. Ez azért szükséges, mert a dátum és az időpont közül nekünk csak a dátum kell. A számátalakítások elvégzésekor a Python nyelv beépített típusfüggvényeire, a `float()` és az `int()` függvényekre támaszkodunk. Érdemes megjegyeznünk, hogy az `int("4.0")` függvényhívás `ValueError` kivételt eredményez, és ha tágabb határokat szeretnénk engedni az egész számokat illetően, használhatjuk az `int(float("4.0"))` függvényhívást, ha pedig a csonkolás helyett inkább kerekíteni óhajtunk, a `round(float("4.0"))` hívás a megfelelő. A logikai értékek kinyerése hangyányi ügyeskedéssel jár, mert például a `bool("0")` hívás igaz értéket ad vissza (minden olyan karakterlánc, amely nem üres, igaz értéket ad), vagyis a karakterláncot előbb egész számmá kell alakítanunk.

A hibás, hiányos, vagy a megengedett tartományon kívül eső értékek minden esetben kivételhez vezetnek. Ha valamelyik átalakítás zárul sikertelenül, `ValueError` kivételt kapunk. Ha bármely érték kívül esik a megengedett tartományon, `IncidentError` kivétel lép fel abban a pillanatban, amikor az érték használatával megpróbáljuk példányosítani a megfelelő `Incident` objektumot.

Ha egy sor nem tartalmaz egyenlőségjelet, ellenőrizzük, hogy nem éppen a kísérőszöveg kezdetét jelölő karakterláncot olvastuk-e be. Ha igen, a `narrative` változónak üres karakterláncot adunk értékül. Így az elágazás első feltétele a következő sorok esetében egészen addig igaz marad, amíg be nem olvassuk a kísérőszöveg végét jelölő karakterláncot.

Ha sem az `if`, sem az `elif` ág feltétele nem vált igazzá, akkor valamilyen hibába ütköztünk, ezért az elágazás végén lévő `else` záradékkal `KeyError` kivételt hozunk létre.

```
        return True
    except (EnvironmentError, ValueError, KeyError,
            IncidentError) as err:
        print("{0}: import error: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
    finally:
        if fh is not None:
            fh.close()
```

Miután minden sort beolvastunk, `True` értéket adunk vissza a hívónak – hacsak nem keletkezett kivétel, mely esetben az `except` kódblokk elkapja a kivételt, hibaüzenetet ír a képernyőre a felhasználónak, és `False` értéket ad vissza a hívónak. Akármi történt is, ha a fájt megnyitottuk, be is zárjuk mindezek végeztével.

Szövegfeldolgozás szabályos kifejezések használatával

Azoknak az olvasóknak, akik nem mozognak otthonosan a szabályos kifejezések (regular expression vagy „regex"), világában, azt tanácsoljuk, hogy mielőtt a következő részbe belefognának, vagy olvassák el a 12. fejezetet, vagy folytassák az olvasást a következő szakasszal, ide pedig, ha úgy tartja kedvük, térjenek vissza később.

A szövegfájlok feldolgozásának szabályos kifejezések használatával történő megoldása gyakran rövidebb kódot eredményez, mint ha mindent saját kezűleg végeznénk – ahogy mi is tettük az előzőekben –, ugyanakkor pontos hibajelzéseket adni lényegesen nehezebb feladat lehet. Az `import_text_regex()` tagfüggvény áttekintését két lépésben ejtjük meg: először a szabályos kifejezéseket vizsgáljuk meg, azt követően pedig magát a feldolgozást. Az `except` és a `finally` kódrészleteket elhagyjuk, mivel már semmi újat nem tanulhatunk belőlük.

```
def import_text_regex(self, filename):
    incident_re = re.compile(
                  r"\[(?P<id>[^]]+)\](?P<keyvalues>.+?)"
                  r"^\.NARRATIVE_START\.$(?P<narrative>.*?)"
                  r"^\.NARRATIVE_END\.$",
                  re.DOTALL|re.MULTILINE)
    key_value_re = re.compile(r"^\s*(?P<key>[^=]+)\s*=\s*"
                              r"(?P<value>.+)\s*$", re.MULTILINE)
```

Nyers karakterláncok 61 ☜

A szabályos kifejezéseket nyers karakterláncként írjuk le. Ezzel megtakaríthatjuk azt a fáradságot, ami a fordított perjelek megkettőzésével (minden \ helyett \\ írása) járna – például a nyers karakterláncok használata nélkül a második szabályos kifejezést a következő alakban kellene leírnunk: `"^\\s*(?P<key>[^=]+)\\s*=\\s*(?P<value>.+)\\s*$"`. A könyvben a szabályos kifejezések írásakor mindig nyers karakterláncokat használunk.

Az első szabályos kifejezés, az `incident_re` egy teljes baleset feljegyzésének felel meg a szövegfájlban. A szabályos kifejezés használatának egyik következménye, hogy figyelmen kívül hagyunk minden olyan szöveget, ami az adatbázisrekordok *között* áll. Ez a szabályos kifejezés valójában két részből áll. Az első rész a `\[(?P<id>[^]]+)\](?P<keyvalues>.+?)`, amely megkeres egy nyitó szögletes zárójelet, megkeresi és beolvassa az utána álló akárhány karaktert, egészen addig, amíg záró szögletes zárójelet nem talál. Így megkapjuk a jelentésazonosítót. Ez után megkeres néhány – de legalább egy –, a `keyvalues` illeszkedési csoportnak megfelelő karaktert (a `re.DOTALL` jelzés miatt az újsor karakterek is számítanak). A `keyvalues` illeszkedési csoportnak megfelelő karakterek jelentik annak a minimálisan szükséges feltételnek a teljesülését, amellyel továbbléphetünk a szabályos kifejezés második részéhez.

Az első szabályos kifejezés második része a következő: `^\.NARRATIVE_START\.$(?P<narrative>.*?)^\.NARRATIVE_END\.$`, ami betű szerint illeszkedik a `.NARRATIVE_START.` karakterláncra, majd a `narrative` illeszkedési csoportba bekerült, lehető legkevesebb féle karakterre, végül a baleset végénél lévő `.NARRATIVE_END.` karakterláncra. A `re.MULTILINE` jelölő miatt ebben a szabályos kifejezésben a kalap (^) nem csak a karakterlánc elejére, hanem minden sor elejére illeszkedik, és ehhez hasonlóan a „dollár” ($) sem csak a karakterlánc végére, hanem minden sor végére illeszkedik – így a kísérőszöveg elejét és végét jelölő karakterláncokra is a sorok elején érvényes az illeszkedés.

A második szabályos kifejezés, a `key_value_re` a *kulcs=érték* párokat tartalmazó sorokat keresi. Az átadott szövegben csak a sorok elejénél illeszkedik, mégpedig a sor elején lévő akárhány üreshely karakterre (az sem baj, ha nem talál ilyet), amelyet egyenlőségjelet nem tartalmazó karakterek sorozata követ – ezekből lesz a `key` illeszkedési csoport. Utánuk következik az egyenlőségjel, amelyet aztán további karakterek követnek – ezekből az előttük és az utánuk álló szóköz jellegű karakterek elhagyásával a `value` illeszkedési csoportot képezzük.

A fájl feldolgozásának alapvető elve megegyezik azzal, amit az előző részben tárgyalt kézi szövegfeldolgozóban használtunk, azzal a különbséggel, hogy ezúttal nem sorról sorra olvassuk be a baleseteket tároló adatbázisrekordokat, hanem szabályos kifejezésekkel nyerjük ki őket a fájlból.

```
        fh = None
        try:
            fh = open(filename, encoding="utf8")
            self.clear()
            for incident_match in incident_re.finditer(fh.read()):
                data = {}
                data["report_id"] = incident_match.group("id")
                data["narrative"] = textwrap.dedent(
                                    incident_match.group("narrative")).strip()
                keyvalues = incident_match.group("keyvalues")
                for match in key_value_re.finditer(keyvalues):
                    data[match.group("key")] = match.group("value")
                data["date"] = datetime.datetime.strptime(
                                            data["date"], "%Y-%m-%d").date()
                data["pilot_percent_hours_on_type"] = (
                        float(data["pilot_percent_hours_on_type"]))
                data["pilot_total_hours"] = int(
                    data["pilot_total_hours"])
                data["midair"] = bool(int(data["midair"]))
                if len(data) != 9:
                    raise IncidentError("missing data")
                incident = Incident(**data)
                self[incident.report_id] = incident
            return True
```

Az `re.finditer()` tagfüggvény olyan bejárható objektumot ad vissza, ami viszont az egymást át nem fedő illeszkedéseket állítja elő. Ahogy eddig is tettük, létrehozzuk az egy baleset adatait tároló `data` szótárat, de ezúttal a jelentésazonosítót és a kísérőszöveget az `incident_re` szabályos kifejezés egyes illeszkedéseiből kapjuk. Ennek végeztével a `keyvalues` illeszkedési csoporttal egyetlen menetben kinyerjük az összes *kulcs=érték* párt, és a `key_values_re` szabályos kifejezés `re.finditer()` tagfüggvényének alkalmazásával bejárjuk az egyes *kulcs=érték* sorokat. Minden megtalált (kulcs, érték) párt elhelyezünk a `data` szótárban – vagyis minden érték karakterláncként kerül a szótárba. Utóbb azokat az értékeket, amelyek nem igazi karakterláncok, lecseréljük az előző, megfelelő típusú adatra. A karakterláncok átalakítását ugyanazokkal a függvényekkel végezzük, mint amelyeket előzőleg, a kézi feldolgozás során is használtunk.

A kódba bekerült egy ellenőrzés is, ami azért felelős, hogy a `data` szótárba kilenc elem kerüljön – ha ugyanis az adatbázis egy balesetet tartalmazó rekordja megsérül, a `key_value.finditer()` függvény esetleg túl sok vagy túl kevés *kulcs=érték* párt tartalmazó sort talál. A végeredmény ugyanaz, mint az előzőekben – létrehozunk egy új `Incident` objektumot, amelyet behelyezünk a baleseteket tartalmazó szótárba, majd `True` értéket adunk vissza. Ha bárhol hiba történne, az `except` kódrész kiírja a megfelelő hibaüzenetet, és `False` értéket ad vissza, majd a `finally` kódblokk lezárja a fájlt.

A Python nyelv kivételkezelése az egyik dolog, amely mind a kézi, mind a szabályos kifejezések használatán alapuló szövegfeldolgozót olyannyira röviddé és magától értetődővé teszi. A feldolgozóknak nem kell ellenőrizniük egyetlen olyan átalakítást sem, ahol karakterláncból dátumot, számot, vagy logikai értéket képzünk, és nem kell elvégezniük egyetlen tartományellenőrzést sem, mert ez az `Incident` objektumok dolga. Ha az előzőekben megjelölt műveletek közül bármelyik sikertelen, kivétel jön létre, a kivételeket pedig igen takarosan, a kód végén, egyetlen helyen kezeljük. Annak, hogy a közvetlen ellenőrzés helyett a kivételkezelésre támaszkodunk, van még egy előnyös oldala: a kód jó méretezhetősége. A hibakezelést végző kód akkor sem lesz hosszabb, ha az adatbázisrekord formátumának megváltozásával további adatelemek ellenőrzése is szükségessé válik.

XML-fájlok írása és feldolgozása

Néhány program minden adat kezeléséhez az XML fájlformátumot használja, más programok csak a kényelmes adatbetöltéshez és -mentéshez. Az XML formátumú adatok betöltésére és mentésére (kivitelére) való képesség igen hasznos dolog, és akkor is érdemes lehet vele foglalkozni, ha a program fő formátuma egyébként valamilyen szöveges vagy bináris fájlformátum.

A Python három módszert kínál az XML-fájlok írására: megírhatjuk őket saját kezűleg; létrehozhatunk egy elemfát, és használhatjuk az elemfa `write()` tagfüggvényét, illetve létrehozhatunk egy DOM-ot, és használhatjuk a DOM `write()` tagfüggvényét. Ami az XML-fájlok olvasását és feldolgozását illeti, négy megközelítési lehetőség közül választhatunk: beolvashatjuk és értelmezhetjük kézi módszerrel az XML-fájlokat (a módszer ellenjavallt, és nem tárgyaljuk – a homályosabb vagy bonyolultabb részletek kezelése meglehetős nehézségeket okozhat), használhatunk elemfát, illetve DOM- vagy SAX-feldolgozót.

A repülőgép-balesetek tárolására a 7.5. ábrán látható formátumot használjuk. Ebben a részben bemutatjuk, miként lehetséges a formátum kézi módszerrel, elemfával és DOM-mal való írása, illetve elemfával, DOM-mal és SAX-szal való beolvasása és feldolgozása. Ha az Olvasó nem különösebben törődik azzal, hogy melyik módszert használja, akkor elég, ha elolvassa az *Elemfák* című részt, és rögtön továbblapozhat a fejezet utolsó, *Véletlen elérésű bináris fájlok* című részére.

Elemfák

Az adatok elemfa használatával történő írása két lépésben történik. Először létre kell hozni az adatot ábrázoló elemfát, majd a fát egy fájlba kell írni. Lehetnek olyan programok, amelyek az adataikat eleve elemfában tárolják; ebben az esetben az elemfa már meg is van, csak az adatok kiírásával kell törődni. Az `export_xml_etree()` tagfüggvényt két részletben tekintjük át.

```
def export_xml_etree(self, filename):
    root = xml.etree.ElementTree.Element("incidents")
    for incident in self.values():
```

```
<?xml version="1.0" encoding="UTF-8"?>
<incidents>
<incident report_id="20070222008099G" date="2007-02-22"
    aircraft_id="80342" aircraft_type="CE-172-M"
    pilot_percent_hours_on_type="9.09090909091"
    pilot_total_hours="440" midair="0">
<airport>BOWERMAN</airport>
<narrative>
ON A GO-AROUND FROM A NIGHT CROSSWIND LANDING ATTEMPT THE AIRCRAFT HIT
A RUNWAY EDGE LIGHT DAMAGING ONE PROPELLER.
</narrative>
</incident>
<incident>
    ...
</incident>
    :
</incidents>
```

7.5. ábra

A repülőgép-balesetet tároló XML formátumú adatbázisrekord és környezete

```
        element = xml.etree.ElementTree.Element("incident",
                  report_id=incident.report_id,
                  date=incident.date.isoformat(),
                  aircraft_id=incident.aircraft_id,
                  aircraft_type=incident.aircraft_type,
                  pilot_percent_hours_on_type=str(
                                  incident.pilot_percent_hours_on_type),
                  pilot_total_hours=str(incident.pilot_total_hours),
                  midair=str(int(incident.midair)))
        airport = xml.etree.ElementTree.SubElement(element, "airport")
        airport.text = incident.airport.strip()
        narrative = xml.etree.ElementTree.SubElement(element, "narrative")
        narrative.text = incident.narrative.strip()
        root.append(element)
    tree = xml.etree.ElementTree.ElementTree(root)
```

Először az `<incidents>` gyökérelemet hozzuk létre, majd bejárjuk a baleseteket tároló adatbázisrekordokat. Minden balesetnél létrehozzuk a baleset adatait tároló `<incident>` elemet, és kulcsszó-paraméterek használatával adjuk meg a jellemzőket. Minden jellemzőt szövegként adunk meg, ennek megfelelően a dátumokat, számadatokat és logikai értékeket át kell alakítanunk. Az „&”, a „<” és a „>” elemek, illetve a jellemzőkben szereplő idézőjelek levédése miatt ne fájjon a fejünk, ugyanis az elemfa (illetve a DOM és a SAX) modul az ilyen feladatokat automatikusan ellátja.

Minden `<incident>` elem két alelemmel bír: az egyik a repülőtér nevét, a másik a kísérőszöveget tárolja. Alelemek létrehozásakor meg kell adnunk a szülőelemet és a címke nevét. Az egyes elemek olvasható-írható `text` jellemzője használatos a szöveg tárolására.

Ha az <incident> elemet minden jellemzőjével, <airport> és <narrative> alelemeivel együtt létrehoztuk, elhelyezhetjük a hierarchia gyökérelemén, az <incidents> elemen belül. A feldolgozás végeztével olyan elemhierarchiát kapunk, amely a baleseteket tároló adatbázis minden rekordját tartalmazza. A hierarchia ekkorra minden további nélkül elemfává alakítható.

```
try:
    tree.write(filename, "UTF-8")
except EnvironmentError as err:
    print("{0}: import error: {1}".format(
        os.path.basename(sys.argv[0]), err))
    return False
return True
```

A teljes elemfa XML formátumú fájlba való kiírása tehát nem jelent többet, mint megmondani az elemfa-objektumnak, hogy írja ki magát egy fájlba, egy adott karakterkódolással. Eddig, valahányszor karakterkódolást adtunk meg, szinte mindig az „utf8" alakot használtuk. Ez remekül működik a Python nyelv beépített open() függvénye esetében, amely sok karakterkódolást, az egyes karakterkódolások esetében pedig sok elfogadható nevet ismer: az „UTF-8", az „UTF8", az „utf-8" és az „utf8" mind megfelelő. Az XML-fájlok esetében azonban a karakterkódolást csak a hivatalosan használható nevek valamelyikével adhatjuk meg, ezért használtuk az „UTF-8" formát.*

Az XML-fájlok elemfa segítségével történő beolvasása nem sokkal nehezebb, mint a kiírásuk. Az eljárás ismét két lépésből áll: először beolvassuk és feldolgozzuk az XML-fájlt, majd a művelet eredményeként létrejövő elemfa adatainak felhasználásával feltöltjük adatokkal az incidents szótárat. Most is rámutatunk, hogy a második lépés nem szükséges, ha magát az elemfát használjuk adatok tárolására a memóriában is. Íme az import_xml_etree() tagfüggvény, két részre bontva.

```
def import_xml_etree(self, filename):
    try:
        tree = xml.etree.ElementTree.parse(filename)
    except (EnvironmentError,
            xml.parsers.expat.ExpatError) as err:
        print("{0}: import error: {1}".format(
            os.path.basename(sys.argv[0]), err))
        return False
```

Alapértelmezés szerint az elemfa-feldolgozó az expat XML-feldolgozóra támaszkodik a háttérben – így aztán fel kell készülnünk az expat-tól származó kivételek elkapására is.

```
    self.clear()
    for element in tree.findall("incident"):
        try:
```

* Az XML-fájlok karakterkódolásáról a www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl és a www.iana.org/assignments/character-sets webhelyeken találunk információt.

```
            data = {}
            for attribute in ("report_id", "date", "aircraft_id",
                              "aircraft_type",
                              "pilot_percent_hours_on_type",
                              "pilot_total_hours", "midair"):
                data[attribute] = element.get(attribute)
            data["date"] = datetime.datetime.strptime(
                                        data["date"], "%Y-%m-%d").date()
            data["pilot_percent_hours_on_type"] = (
                                    float(data["pilot_percent_hours_on_type"]))
            data["pilot_total_hours"] = int(data["pilot_total_hours"])
            data["midair"] = bool(int(data["midair"]))
            data["airport"] = element.find("airport").text.strip()
            narrative = element.find("narrative").text
            data["narrative"] = (narrative.strip()
                                 if narrative is not None else "")
            incident = Incident(**data)
            self[incident.report_id] = incident
        except (ValueError, LookupError, IncidentError) as err:
            print("{0}: import error: {1}".format(
                os.path.basename(sys.argv[0]), err))
            return False
    return True
```

Ha az elemfánk elkészült, az `xml.etree.ElementTree.findall()` tagfüggvénnyel bejárhatjuk az `<incidents>` elemeket. Az egyes `<incidents>` elemeket `xml.etree.Element` osztályú objektumként kapjuk vissza. Az elemek jellemzőinek kezelésére ugyanazt az eljárást használjuk, mint korábban, az `import_text_regex()` tagfüggvénynél: először minden adatot a `data` szótárban tárolunk, majd a dátumokat, számadatokat és logikai értékeket a megfelelő típusú adattá alakítjuk. Az `airport` és a `narrative` alelemek esetében az `xml.etree.Element.find()` tagfüggvényt használjuk a megkeresésükre és a hozzájuk kapcsolódó `text` jellemző beolvasására. Ha egy szöveges elemben nincs szöveg, a `text` jellemzője a `None` értéket veszi fel, amire számítanunk kell a kísérőszöveg beolvasásakor – ez ugyanis lehet üres. A tagfüggvényektől visszakapott szöveg soha nem tartalmaz XML-védőkaraktereket, mert a kezelésüket a Python automatikusan elvégzi.

Mint minden, a repülőgép-balesetek adatait feldolgozó XML-feldolgozó esetében, kivétel jön létre, amennyiben az `airport` vagy a `narrative` elem, illetve bármely jellemző hiányzik, ha valamelyik átalakítás sikertelen, vagy ha bármely számadat kívül eshet a lehetséges értékek tartományán. Így biztosak lehetünk benne, hogy érvénytelen adat esetén leáll a feldolgozás, és hibaüzenet jelenik meg. A tagfüggvény végén lévő, az `incident` objektumok létrehozására és tárolására, valamint a kivételek kezelésére szolgáló kódrészlet megegyezik azzal, amit már korábban megismertünk.

DOM (Document Object Model)

A DOM az XML-fájlok memóriában történő ábrázolását és kezelését végző szabványos alkalmazásprogramozási felület. A DOM létrehozására és fájlba írására szolgáló, illetve az XML-fájl DOM használatával történő értelmezésére szolgáló kód szerkezetét tekintve nagyon hasonló az elemfával dolgozó megfelelőjéhez, és csak kevéssel hosszabb annál.

Elsőként az `export_xml_dom()` tagfüggvényt tekintjük át, két részletben. A tagfüggvény két lépésben végzi a munkáját: először egy a balesetek adatainak megfelelő DOM készül el, majd a kész DOM-ot kiírjuk egy fájlba. Az elemfához hasonlóan a DOM esetében is létezhetnek olyan programok, amelyek magát a DOM-ot használják belső adatszerkezetként – ilyen esetekben a tárolt adat egyszerűen lemezre írható.

```
def export_xml_dom(self, filename):
    dom = xml.dom.minidom.getDOMImplementation()
    tree = dom.createDocument(None, "incidents", None)
    root = tree.documentElement
    for incident in self.values():
        element = tree.createElement("incident")
        for attribute, value in (
                ("report_id", incident.report_id),
                ("date", incident.date.isoformat()),
                ("aircraft_id", incident.aircraft_id),
                ("aircraft_type", incident.aircraft_type),
                ("pilot_percent_hours_on_type",
                 str(incident.pilot_percent_hours_on_type)),
                ("pilot_total_hours",
                 str(incident.pilot_total_hours)),
                ("midair", str(int(incident.midair)))):
            element.setAttribute(attribute, value)
        for name, text in (("airport", incident.airport),
                           ("narrative", incident.narrative)):
            text_element = tree.createTextNode(text)
            name_element = tree.createElement(name)
            name_element.appendChild(text_element)
            element.appendChild(name_element)
        root.appendChild(element)
```

A tagfüggvény egy DOM-megvalósítás használatba vételével kezdődik. Alapértelmezés szerint a megvalósítást az expat XML-értelmező szolgáltatja. Az `xml.dom.minidom` modul egyszerűbb és kisebb DOM-megvalósítást kínál annál, amit az `xml.dom` modulban találunk, bár az általa használt objektumok az xml.dom modulból származnak. Ha megvan a DOM-megvalósítás, létrehozhatunk egy dokumentumot.

Az `xml.dom.DOMImplementation.createDocument()` tagfüggvény első paramétere a névtérhez tartozó URI, amelyre nincs szükségünk, így aztán `None` értéket adunk át. A második paraméter egy minősített név, a gyökérelemhez tartozó címke. A harmadik paraméter a dokumentumtípus; itt megint `None` értéket adunk meg, ugyanis nem állítunk be dokumentumtípust. Most, hogy hozzájutottunk a dokumentumot ábrázoló fához, kinyerjük a gyökérelemet, és továbblépünk: az `incident` objektumok bejárása következik.

Minden balesethez létrehozunk egy `<incident>` elemet, és minden olyan jellemző esetében, amivel fel akarjuk ruházni az elemet, meghívjuk a `setAttribute()` függvényt a jellemző nevével és értékével paraméterezve. Csakúgy, mint az elemfa esetében, itt sem kell aggódnunk a „&”, a „<” és a „>” elemek, illetve a jellemzők értékében szereplő idézőjelek levédése miatt. Az `airport` és a `narrative` szövegelem esetében előbb létre kell hoznunk a szöveg tárolását végző szövegelemet, majd egy normál elemet – megfelelő címkenévvel –, a szövegelem szülőelemét. Amikor létrejöttek, a normál elemet, illetve a benne lévő szövegelemet elhelyezzük az aktuális `<incident>` elemben. Amikor pedig az `<incident>` elem is létrejött, elhelyezzük a gyökérelemben.

```
fh = None
try:
    fh = open(filename, "w", encoding="utf8")
    tree.writexml(fh, encoding="UTF-8")
    return True
```

Az `except` és a `finally` kódblokkot elhagytuk, ugyanis megegyeznek a korábban megismertekkel. A fenti kódrészlet a beépített `open()` függvénynek átadott karakterkódolás és az XML-fájlokban használt karakterkódolás közötti – az előzőekben már megbeszélt – különbségre hivatott rávilágítani.

Egy XML-dokumentumot egy DOM-ba betölteni hasonló ahhoz, mint amikor elemfába töltünk be, de a kivitelhez hasonlóan hosszabb kódot eredményez. Az `import_xml_dom()` függvényt három részletben tekintjük át. A munkát a `def` utasítást tartalmazó sorral és a beágyazott `get_text()` függvénnyel kezdjük.

```
def import_xml_dom(self, filename):
    def get_text(node_list):
        text = []
        for node in node_list:
            if node.nodeType == node.TEXT_NODE:
                text.append(node.data)
        return "".join(text).strip()
```

A `get_text()` függvény csomópontok listáját járja be – mármint a csomópontok gyermekcsomópontjait –, és minden szövegcsomópontnál kinyeri a csomópont szövegét, amelyet elhelyez egy szövegeket tartalmazó listában. A művelet végén a függvény az összegyűjtött szöveget egyetlen karakterláncként adja vissza, az üreshelyeket a karakterlánc elejéről és végéről egyaránt eltávolítva.

```
try:
    dom = xml.dom.minidom.parse(filename)
except (EnvironmentError,
        xml.parsers.expat.ExpatError) as err:
    print("{0}: import error: {1}".format(
        os.path.basename(sys.argv[0]), err))
    return False
```

Egy XML-fájl DOM-má alakítása könnyű, mivel a modul elvégzi helyettünk a munka nehéz részét, ugyanakkor készen kell állnunk az expat-hibák kezelésére, mert az elemfákhoz hasonlóan a háttérben a DOM-osztályok alapértelmezett feldolgozója is az expat XML-feldolgozó.

```
        self.clear()
        for element in dom.getElementsByTagName("incident"):
            try:
                data = {}
                for attribute in ("report_id", "date", "aircraft_id",
                                  "aircraft_type",
                                  "pilot_percent_hours_on_type",
                                  "pilot_total_hours", "midair"):
                    data[attribute] = element.getAttribute(attribute)
                data["date"] = datetime.datetime.strptime(
                                        data["date"], "%Y-%m-%d").date()
                data["pilot_percent_hours_on_type"] = (
                                    float(data["pilot_percent_hours_on_type"]))
                data["pilot_total_hours"] = int(data["pilot_total_hours"])
                data["midair"] = bool(int(data["midair"]))
                airport = element.getElementsByTagName("airport")[0]
                data["airport"] = get_text(airport.childNodes)
                narrative = element.getElementsByTagName("narrative")[0]
                data["narrative"] = get_text(narrative.childNodes)
                incident = Incident(**data)
                self[incident.report_id] = incident
            except (ValueError, LookupError, IncidentError) as err:
                print("{0}: import error: {1}".format(
                        os.path.basename(sys.argv[0]), err))
                return False
        return True
```

Helyi függvények
☞ 357

Ha a DOM létrejött, töröljük az aktuális baleset adatait, és bejárjuk az `<incident>` elemeket. Minden elemből kinyerjük a jellemzőket, és a dátumot, számot vagy logikai értéket tárolók esetében átalakítjuk azokat a megfelelő adattípusra – pontosan ugyanúgy, ahogy az elemfa használatakor tettük. Az egyetlen igazán jelentős különbség a DOM és az elemfa használata között a szövegcsomópontok kezelésében rejlik.

Az `xml.dom.Element.getElementsByTagName()` tagfüggvényt használjuk az adott névvel rendelkező gyermekelemek felismerésére. Az `<airport>` és a `<narrative>` gyermekelemek esetében tudjuk, hogy mindkettőből pontosan egy létezik, így aztán mindkét típusnál csak az első (és egyetlen) gyermekelemet vesszük figyelembe. Ezt követően a beágyazott `get_text()` függvény használatával bejárjuk a fenti elemek gyermekcsomópontjait, kinyerve a bennük lévő szövegeket.

Ha hiba történik, a tagfüggvény elfogja a megfelelő kivételt, hibaüzenetet ír a felhasználó konzolablakába, és `False` értékkel tér vissza – ahogy azt már megszoktuk.

A DOM és az elemfa használata közötti különbségek nem jelentősek, és minthogy mindkét eljárás esetében az igazi munkát az expat feldolgozó végzi, mindkettő meglehetősen gyorsan dolgozik.

XML-fájl írása saját kezűleg

Ha az elemfánk, illetve a DOM-unk már készen áll, egyetlen tagfüggvény hívásával lemezre írható belőlük az XML-dokumentum. Ha az adatokat azonban még nem rendeztük valamelyik formátumba, akkor először létre kell hoznunk az elemfát, illetve a DOM-ot. Ilyen esetekben lehet, hogy kényelmesebb egyszerűen az adat közvetlen kiírását elvégezni.

Az XML-fájlok írásakor figyelnünk kell arra, hogy a szöveget és a jellemzők értékeit megfelelően levédjük, és arra, hogy jólformált XML-dokumentumot írjunk. Íme a balesetek XML-formátumban való kiírását végző `export_xml_manual()` tagfüggvény:

```
def export_xml_manual(self, filename):
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        fh.write('<?xml version="1.0" encoding="UTF-8"?>\n')
        fh.write("<incidents>\n")
        for incident in self.values():
            fh.write('<incident report_id={report_id} '
                     'date="{0.date!s}" '
                     'aircraft_id={aircraft_id} '
                     'aircraft_type={aircraft_type} '
                     'pilot_percent_hours_on_type='
                     '"{0.pilot_percent_hours_on_type}" '
                     'pilot_total_hours="{0.pilot_total_hours}" '
                     'midair="{0.midair:d}">\n'
                     '<airport>{airport}</airport>\n'
                     '<narrative>\n{narrative}\n</narrative>\n'
                     '</incident>\n'.format(incident,
                report_id=xml.sax.saxutils.quoteattr(
                                              incident.report_id),
                aircraft_id=xml.sax.saxutils.quoteattr(
                                             incident.aircraft_id),
                aircraft_type=xml.sax.saxutils.quoteattr(
                                            incident.aircraft_type),
                airport=xml.sax.saxutils.escape(incident.airport),
                narrative="\n".join(textwrap.wrap(
                        xml.sax.saxutils.escape(
                            incident.narrative.strip()), 70))))
        fh.write("</incidents>\n")
        return True
```

Mint ahogy a fejezetben már gyakran megtettük, ezúttal is elhagytuk az `except` és a `finally` kódblokkot.

A fájl írását az UTF-8 karakterkódolás használatával végezzük, és ezt jeleznünk kell a beépített `open()` függvénynek. Ha pontosan akarunk fogalmazni, akkor azt kell mondanunk, hogy a karakterkódolást nem kell megadnunk az `<?xml?>` bevezetésben, hiszen az UTF-8 az alapértelmezett karakterkódolás, mindazonáltal szeretünk pontosak lenni. Úgy döntöttünk, hogy a jellemzők értékeit idézőjelek közé tesszük, így viszont a kényelem azt diktálta, hogy az egyes baleseteket tároló karakterláncok kerüljenek aposztrófok közé: ezzel elkerülhető az idézőjelek levédése.

A `sax.saxutils.quoteattr()` függvény abban az értelemben hasonló az általunk az XML formátumú szövegek esetében használt `sax.saxutils.escape()` függvényhez, hogy helyesen védi le az „&”, a „<” és a „>” karaktereket. Ezen felül – szükség esetén – levédi az idézőjelet, az aposztrófot, és az olyan karakterláncokat, amelyeket idézőjelek vagy aposztrófok közé zárva adtunk át, használatra készen adja vissza. Ebben rejlik a magyarázata annak, hogy nem tettük idézőjelbe a jelentésazonosítót, illetve más jellemzők karakterlánc típusú értékeit.

A beszúrt újsorok, illetve a kísérőszöveg tördelése pusztán kozmetikai célokat szolgál. Arra valók, hogy a fájlt az ember könnyebben tudja olvasni és szerkeszteni, de akár el is hagyhatók.

A HTML formátumú adatírás nem sokban különbözik az XML formátumban történőtől. A `convert-incidents.py` program ennek egyszerű példájaként tartalmazza az `export_html()` függvényt, amelyet ez alkalommal nem vizsgálunk meg: aligha találunk benne bármi újat.

XML-fájlok feldolgozása a SAX segítségével

Az elemfától és a DOM-tól – amelyek az egész XML-dokumentumot ábrázolják a memóriában – eltérően a SAX (Simple API for XML, azaz egyszerű XML-alkalmazásprogramozási felület) a bemenet feldolgozását fokozatosan végzi, vagyis akár lehet gyorsabb és kevésbé memóriaigényes is. Teljesítményjavulásra mindazonáltal nincs okunk számítani, lévén az elemfa és a DOM egyaránt a gyors expat feldolgozót használja.

A SAX a bemenet feldolgozását „feldolgozási események” bejelentésével végzi, valahányszor kezdő vagy lezáró címkéhez, illetve egyéb XML-elemhez ér. Ha kezelni szeretnénk a számunkra érdekes eseményeket, létre kell hoznunk egy megfelelő, a kezelést végző osztályt, benne néhány előre megadott tagfüggvénnyel, amelyeknek a meghívására akkor kerül sor, amikor a hozzájuk tartozó feldolgozási esemény bekövetkezik. A leggyakrabban megvalósított kezelést végző osztály a tartalomkezelő, bár ha pontosabb ellenőrzésre van szükség, lehetséges hibakezelők és egyéb kezelőosztályok létrehozása is.

Íme a teljes `import_xml_sax()` tagfüggvény. Nagyon rövid, hiszen a munka legnagyobb részét az általunk létrehozott `IncidentSaxHandler` osztály végzi.

```
    def import_xml_sax(self, filename):
        fh = None
        try:
            handler = IncidentSaxHandler(self)
            parser = xml.sax.make_parser()
            parser.setContentHandler(handler)
            parser.parse(filename)
            return True
        except (EnvironmentError, ValueError, IncidentError,
                xml.sax.SAXParseException) as err:
            print("{0}: import error: {1}".format(
                os.path.basename(sys.argv[0]), err))
        return False
```

Létrehozzuk azt az egyetlen kezelőt, amelyet használni szeretnénk, majd létrehozzuk a SAX feldolgozót, tartalomkezelőjéül az imént létrehozott kezelőt állítva be. Ezek után átadjuk a fájl nevét a feldolgozó `parse()` tagfüggvénynek, és `True` értéket adunk vissza, ha a feldolgozásba nem csúszott hiba.

A saját készítésű, `IncidentSaxHandler` osztály előkészítőjének `self`-et, azaz a `dict` osztály `IncidentCollection` nevű alosztályát adjuk át. A kezelő kitakarítja a régi `Incident` objektumokat, majd a fájl feldolgozása közben kiépíti az `Incident` objektumokból álló szótárat. Mire a feldolgozás véget ér, a szótárban megtalálható az összes beolvasott baleset.

```
class IncidentSaxHandler(xml.sax.handler.ContentHandler):
    def __init__(self, incidents):
        super().__init__()
        self.__data = {}
        self.__text = ""
        self.__incidents = incidents
        self.__incidents.clear()
```

Az egyedi SAX-kezelőosztályok mindig a megfelelő ősosztálytól származnak. Így biztosak lehetünk benne, hogy minden olyan tagfüggvény esetében, amelyet nem valósítunk meg újra (hiszen nem érdekelnek bennünket az általuk kezelt események), az ősosztályban található változat kapja meg a hívást. Az ősosztályban található változat pedig semmit sem fog tenni – ami nekünk tökéletesen megfelel.

Az alaposztály előkészítő függvényének meghívásával kezdjük. Ez általában minden alosztály esetében követendő gyakorlat, bár az `object` osztályból közvetlenül származó osztályok esetében erre nincs szükség – ha viszont mégis megtesszük, abból sem lesz baj. A `self.__data` szótárat használjuk egyetlen baleset adatainak tárolására, a `self.__text` karakterlánc pedig a repülőtér nevét, illetve a kísérőszöveget tárolja, attól függően, hogy melyik beolvasásánál tartunk. A `self.__incidents` szótár pedig egy az `IncidentCollection` szótárra mutató objektumhivatkozás. Az `IncidentCollection` szótár frissítését a kezelő közvetlenül végzi. (Más megközelítés esetén megvalósíthatnánk

egy a kezelő belsejében lévő, független szótárat, amelyet a művelet végén a `dict.clear()` és utána a `dict.update()` függvény használatával átmásolunk az `IncidentCollection` szótárba.)

```
def startElement(self, name, attributes):
    if name == "incident":
        self.__data = {}
        for key, value in attributes.items():
            if key == "date":
                self.__data[key] = datetime.datetime.strptime(
                                   value, "%Y-%m-%d").date()
            elif key == "pilot_percent_hours_on_type":
                self.__data[key] = float(value)
            elif key == "pilot_total_hours":
                self.__data[key] = int(value)
            elif key == "midair":
                self.__data[key] = bool(int(value))
            else:
                self.__data[key] = value
    self.__text = ""
```

Minden alkalommal, amikor egy kezdőcímke, illetve a jellemzői beolvasására kerül sor, a címke nevével és a jellemzőkkel meghívjuk az `xml.sax.handler.ContentHandler.startElement()` tagfüggvényt. A repülőgép-baleseteket tároló XML-fájl esetében a kezdőcímkék a következők: az `<incidents>`, amelyet figyelmen kívül hagyunk, az `<incident>`, amelynek a jellemzőivel a `self.__data` szótár egy részét töltjük fel, illetve az `<airport>` és a `<narrative>`, amelyeket megint csak figyelmen kívül hagyunk. A `self.__text` karakterláncot minden kezdőcímke megtalálásakor ürítjük, ugyanis a repülőgép-balesetes XML-fájlban nem használunk beágyazott szöveges címkéket.

Az `IncidentSaxHandler` osztályon belül semmilyen kivételkezelést nem végzünk. Ha kivétel keletkezik, átadjuk a hívónak, esetünkben az `import_xml_sax()` tagfüggvénynek, ami majd elkapja, és kiírja a megfelelő hibaüzenetet.

```
def endElement(self, name):
    if name == "incident":
        if len(self.__data) != 9:
            raise IncidentError("missing data")
        incident = Incident(**self.__data)
        self.__incidents[incident.report_id] = incident
    elif name in frozenset({"airport", "narrative"}):
        self.__data[name] = self.__text.strip()
    self.__text = ""
```

Amikor egy lezáró címke beolvasása megtörténik, az `xml.sax.handler.ContentHandler.endElement()` tagfüggvényt hívjuk meg. Ha egy baleset végére értünk, elvileg megvan minden szükséges adatunk, így létrehozunk

egy új `<incident>` objektumot, amelyet elhelyezünk a baleseteket tároló szótárban. Ha egy szövegelem végére értünk, az elemet és az addig begyűjtött szöveget elhelyezzük a `self.__data` szótárban. Ha ez megtörtént, a `self.__text` karakterláncot kiürítjük, felkészítve a következő használatra. (Szigorú értelemben nem kell kiürítenünk, hiszen a kezdő címke beolvasásakor is kiürítjük. A kiürítés azonban néhány XML-formátum esetében jelentőségre tehet szert, például ott, ahol a címkék egymásba ágyazhatók.)

```
def characters(self, text):
    self.__text += text
```

Amikor a SAX feldolgozó szöveget olvas be, a `xml.sax.handler.ContentHandler.characters()` tagfüggvényt hívja meg. Nincs garancia arra vonatkozólag, hogy a teljes szöveg beolvasása alatt csak egyszer történik meg a tagfüggvény hívása: a szöveg ugyanis érkezhet darabokban is. Így aztán a tagfüggvényt egyszerűen csak arra használjuk, hogy gyűjtse össze a szöveget, és csak a megfelelő lezáró címkét elérve helyezzük el a szöveget a szótárban. (Hatékonyabb megoldást jelentene, ha a `self.__text` objektum egy lista lenne, a mostani tagfüggvény magja pedig a `self.__text.append(text)` tagfüggvény az objektumban; és persze a többi tagfüggvény átalakítását is el kellene végezni.)

A SAX API használata jelentősen különbözik az elemfa vagy a DOM használatától, de ugyanolyan hatékony módszert jelent. Létrehozhatunk további kezelőket is, illetve a tartalomkezelőben újra megvalósíthatunk egyes tagfüggvényeket, így érve el a folyamat általunk óhajtott ellenőrzési fokát. A SAX feldolgozó önmagában az XML-fájlból nyert adatok semmiféle ábrázolását nem tartalmazza – éppen ez teszi ideális eszközzé a fájl adatainak saját adatszerkezeteinkbe való beolvasására. Mindez azt is jelenti, hogy semmiféle „SAX-dokumentumunk" nincs, amit XML-formátumban kiírhatnánk, vagyis az XML-fájl megírásához a fejezet korábbi részében leírt valamelyik módszert kell igénybe vennünk.

Véletlen elérésű bináris fájlok

Az előző részekben abból indultunk ki, hogy a fájl beolvasása egy lépésben történik, ezt követően feldolgozzuk az adatokat, majd ismét egy lépésben írunk ki mindent. A mai számítógépekben annyi memória található, hogy az ilyen megközelítésnek még nagy adatmennyiség esetén is van létjogosultsága. Mindazonáltal vannak helyzetek, amikor jobb megoldás, ha az adatok a lemezen maradnak, csak a szükséges részeket olvassuk be, és csak a változtatásokat írjuk vissza. A lemezen tárolt adatok véletlen elérésének legegyszerűbb módja, ha valamilyen kulcs–érték alapú adatbázist, esetleg egy teljes értékű SQL-adatbázismotort használunk – a 11. fejezetben mindkettőt tárgyaljuk –, de a következő oldalakon azt ismertetjük, hogy miként lehetséges a véletlen elérésű fájlok kézi kezelése.

Elsőként a `BinaryRecordFile.BinaryRecordFile` osztályt mutatjuk be. Az osztályból példányosított objektum egy általános, olvasható és írható bináris fájlt ábrázol, rögzített hosszúságú rekordokként. Ha ezzel megvagyunk, a `BikeStock.BikeStock` osztályt

vesszük szemügyre, amely `BikeStock.Bike` objektumok gyűjteményét tárolja, mégpedig egy `BinaryRecordFile.BinaryRecordFile` objektum rekordjaiként. Tesszük mindezt azért, hogy megismerjük a véletlen elérésű bináris fájlok használatának módját.

Az általános BinaryRecordFile osztály

A `BinaryRecordFile.BinaryRecordFile` osztály alkalmazásprogramozási felülete egy olyan listához hasonlít, amelyben bármely adott sorszámú elemet olvashatjuk, írhatjuk és törölhetjük. Amikor egy rekordot törlünk, azzal spóroljuk meg a többi rekord előrébb mozgatását, hogy a rekordot egyszerűen csak töröltnek jelöljük. Ez azzal is együtt jár, hogy a törlés végeztével minden eredeti elemsorszám érvényes marad. A módszer további előnye, hogy a törlés a „törölt" jelzés eltávolításával egyszerűen visszavonható. Az előnyök ára, hogy a törléssel nem szabadul fel lemezterület. Ezen a problémán úgy segítünk, hogy módot adunk a fájl „tömörítésére", azaz a törölt rekordok eltávolítására – ami viszont az elemsorszámok érvénytelenné válásával jár együtt.

Mielőtt áttekintenénk a megvalósítást, tekintsük át a használatával kapcsolatos alapvető tudnivalókat:

```
Contact = struct.Struct("<15si")
contacts = BinaryRecordFile.BinaryRecordFile(filename, Contact.size)
```

A fenti példában egy olyan, little-endian bájtsorrendű, egy tizenöt bájtos karakterláncból és egy négybájtos előjeles egész számból álló adatszerkezetet hoztunk létre, amely az egyes rekordokat fogja ábrázolni. A példa második sorában létrehozunk egy `BinaryRecordFile.BinaryRecordFile` objektumpéldányt, fájlnevet és a használt adatszerkezetnek megfelelő rekordméretet adva meg. Ha a fájl létezik, a tartalmának megváltoztatása nélkül megnyitjuk, egyébként pedig létrehozzuk. Mindkét esetben bináris olvasás-írás módban nyitjuk meg.

```
contacts[4] = Contact.pack("Abe Baker".encode("utf8"), 762)
contacts[5] = Contact.pack("Cindy Dove".encode("utf8"), 987)
```

A fájllal úgy dolgozhatunk, mintha lista lenne: a listák elemelérési műveletét (`[]`) kell használnunk itt is. A fenti példában két bájtsorozatot (`bytes` objektumokat, mindkettőben egy a megjelölt karakterkódolással kódolt karakterlánccal és egy egész számmal) helyezünk el a fájl két, sorszámmal jelölt területén. A fenti két értékadás a már esetleg ott lévő adatokat felül fogja írni, ha pedig még nincs hat rekord a fájlban, akkor a hiányzó rekordok `0x00` értékkel létrejönnek.

```
contact_data = Contact.unpack(contacts[5])
contact_data[0].decode("utf8").rstrip(chr(0)) # az eredmény: 'Cindy
Dove'
```

Minthogy a „Cindy Dove" karakterlánc tizenöt UTF-8 karakternél rövidebb, a csomagolásakor a rekord végén fennmaradó helyet `0x00` értékű bájtokkal töltjük fel. Így aztán a rekord visszaolvasásakor a `contact_data` változóba egy kételemű rendezett sorozat

`(b'Cindy Dove\x00\x00\x00\x00\x00', 987)` kerül. A név visszanyeréséhez az UTF-8 kódolású karakterláncot Unicode kódolásúvá kell alakítanunk, és el kell távolítanunk a hátul lévő `0x00` bájtokat.

Most, hogy működés közben vetettünk már az osztályra egy pillantást, készen állunk a kód áttekintésére. A `BinaryRecordFile.BinaryRecordFile` osztály a `BinaryRecordFile.py` fájlban kapott helyett. A fájl a szokásos bevezetés után néhány privát, `bytes` típusú változó megadásával kezdődik.

```
_DELETED = b"\x01"
_OKAY = b"\x02"
```

Minden rekord egy állapotjelző bájttal kezdődik, ami vagy `_DELETED` (törölt), vagy `_OKAY` (oké), illetve az üres rekordok esetében `0x00`.

Íme a `class` utasítást tartalmazó sor és az előkészítést végző kódrész:

```
class BinaryRecordFile:
    def __init__(self, filename, record_size, auto_flush=True):
        self.__record_size = record_size + 1
        mode = "w+b" if not os.path.exists(filename) else "r+b"
        self.__fh = open(filename, mode)
        self.auto_flush = auto_flush
```

Két különböző rekordméret létezik. A `BinaryRecordFile.record_size` nevű rekordméret beállítását a felhasználó végzi, és a felhasználó szempontjából ekkora egy rekord. A privát, `BinaryRecordFile.__record_size` nevű változó a valós méretét adja meg a rekordnak, amibe beletartozik az állapotjelző bájt is.

Vigyázunk arra, hogy ha már létezik a fájl, akkor ki ne ürítsük – ezért ilyenkor „`r+b`” módban nyitjuk meg –, és arra is, hogy létrehozzuk, ha még nem létezik – ezért ilyenkor „`w+b`” módban nyitjuk meg. A mód megadásában lévő pluszjel jelenti az olvasásra *és* írásra való megnyitást. Ha a `BinaryRecordFile.auto_flush` változó értéke igaz, a fájl minden olvasás előtt és írás után a lemezre íródik a gyorsítótárból.

```
    @property
    def record_size(self):
        return self.__record_size - 1
    @property
    def name(self):
        return self.__fh.name
    def flush(self):
        self.__fh.flush()
    def close(self):
        self.__fh.close()
```

7.4. táblázat *Fájlobjektumok jellemzői és tagfüggvényei (1)*

Utasításforma	Leírás
`f.close()`	Az `f` fájlobjektumot lezárja, az `f.closed` jellemző értékét `True`-ra állítja.
`f.closed`	Ha `f` fájlobjektum lezárt, `True` értéket ad vissza.
`f.encoding`	A `bytes` és a `str` adattípusok közötti átalakítás során használt karakterkódolást tárolja.
`f.fileno()`	Az `f` fájlobjektumhoz tartozó leírót adja vissza. (Csak olyan fájloknál, amelyeknek van fájlleírójuk.)
`f.flush()`	A gyorsítótárból a háttértárra írja az `f` fájlobjektum tartalmát.
`f.isatty()`	`True` értéket ad vissza, ha a fájlobjektum egy konzolhoz van rendelve. (Csak igazi fájlokhoz tartozó fájlobjektumoknál használható.)
`f.mode`	Az `f` fájlobjektum megnyitásakor használt módot tárolja.
`f.name`	Az `f` fájlobjektum nevét tárolja (ha van).
`f.newlines`	Az `f` szövegfájl-objektumban tárolt újsor karakterek fajtáit tárolja.
`f.__next__()`	Az `f` fájlobjektum következő sorát adja vissza. A legtöbb esetben nem közvetlenül használjuk – például a `for sor in f` utasításban sem.
`f.peek(n)`	`n` bájtot ad vissza, a fájlmutató mozgatása nélkül.
`f.read(szám)`	Az `f` fájlobjektumból legfeljebb *szám* bájtot ad vissza. Ha a *szám* nincs megadva, akkor a fájlban lévő aktuális helyzettől a fájl végéig minden bájtot visszaad. Bináris módú olvasáskor `bytes`, szöveges módú olvasáskor `text` a visszatérési érték típusa. Ha nincs mit beolvasni (vége a fájlnak), üres `bytes`, illetve `str` típusú értéket kapunk vissza.
`f.readable()`	`True` értéket ad vissza, ha a fájlt olvasásra nyitották meg.
`f.readinto(ba)`	Legfeljebb `len(ba)` bájtot olvas be a *ba* `bytearray` típusú változóba, majd a beolvasott bájtok számát – a fájlok végéhez érve nullát – visszaadja. (Csak bináris módban használható.)
`f.readline(szám)`	Beolvassa a következő sort (ha a *szám* meg van adva, akkor legfeljebb az első *szám* bájtot, de csak akkor, ha van elég bájt az `\n` karakter előtt), az `\n` karakterrel együtt.
`f.readlines(méret)`	Beolvassa a fájl végéig hátralévő sorokat, és listaként adja vissza azokat. Ha a *méret* meg van adva, és a használt fájlrendszer támogatja, akkor körülbelül *méret* bájtot olvas be.
`f.seek(eltolás, innen)`	Ha az *innen* nincs megadva, vagy az értéke `os.SEEK_SET`, a fájlmutatót (vagyis ahol a következő írási vagy olvasási művelet kezdődik) az *eltolás*-nak megfelelően eltolja. Ha az *innen* értéke `os.SEEK_CUR`, akkor a jelenlegi helyzettől számítva, ha az *innen* értéke `os.SEEK_END`, akkor a fájl végétől számítva a fájlmutatót *eltolás* (az értéke lehet negatív is) hellyel elmozgatja. Az `a` („append”, azaz hozzáfűzés) módban az írási műveletek a fájlmutató helyétől függetlenül mindig a fájl végén mennek végbe. Szöveges módban az *eltolás* értékéül csak a `tell()` függvény visszatérési értékeit érdemes használnunk.

7.5. táblázat *Fájlobjektumok jellemzői és tagfüggvényei (2)*

Utasításforma	Leírás
`f.seekable()`	`True` értéket ad vissza, ha az `f` fájlobjektum támogatja a véletlen elérést.
`f.tell()`	A fájlmutató értékét a fájl kezdetéhez viszonyítva adja meg.
`f.truncate(méret)`	A fájlt az aktuális pozíciónál (ha a *méret* meg van adva, akkor ott) csonkolja.
`f.writeable()`	`True` értéket ad vissza, ha a fájlt írásra nyitották meg.
`f.write(s)`	Ha a fájl bináris módban van megnyitva, akkor az `s` `bytes/bytearray` objektumot, ha pedig szöveges módban, akkor az *s string* objektumot a fájlba írja.
`f.writelines(sorozat)`	Objektumok sorozatát (szövegfájloknál karakterláncokét, bináris állományoknál `bytes/bytearray` objektumokét) írja a fájlba.

A rekord méretét és a fájlnevet csak olvasható tulajdonságként adtuk meg. A felhasználóval akkora rekordméretet közlünk, amekkorát kért, és amekkora megfelel a rekordoknak. A `flush` és a `close` eljárások egyszerűen továbbadják a feladatot a fájlobjektumnak.

```
def __setitem__(self, index, record):
    assert isinstance(record, (bytes, bytearray)), \
           "binary data required"
    assert len(record) == self.record_size, (
        "record must be exactly {0} bytes".format(
        self.record_size))
    self.__fh.seek(index * self.__record_size)
    self.__fh.write(_OKAY)
    self.__fh.write(record)
    if self.auto_flush:
        self.__fh.flush()
```

Ez a tagfüggvény a *brf*`[`*i*`]` `=` *adat* utasításformát használja, ahol a *brf* egy bináris rekordokból álló fájl, az *i* a rekord sorszáma, az *adat* pedig egy bájtsorozat. Figyeljük meg, hogy a rekordok méretének meg kell egyezniük azzal, amit a bináris fájl létrehozásakor megadtunk. Ha a paraméterek rendben vannak, a fájlmutatót a rekord első bájtjára állítjuk – vegyük észre, hogy ezúttal a valós rekordméretet használtuk, vagyis figyelembe vettük az állapotjelző bájtot. A `seek()` tagfüggvény alapértelmezés szerint a fájlmutatót egy abszolút sorszámú helyre állítja. A második paraméter megadásával tehetjük a helyzetet a jelenlegi helyzethez, illetve a fájl végéhez képest értelmezetté. (A fájlobjektumok jellemzői és tagfüggvényei a 7.4. és a 7.5. táblázatban olvashatók.)

Minthogy az az elem, amelyiknek a beállítását éppen most végezzük, nyilvánvalóan nincs törölve, leírjuk az `_OKAY` állapotjelző bájtot, ezt követően pedig a felhasználótól származó bináris rekordban lévő adatokat. A fájl szempontjából nem érdekes a használt rekordszerkezet, és nem is ismeri azt. Egyetlen dolog számít: a rekordok megfelelő mérete.

Nem ellenőrizzük, hogy a sorszám a megfelelő határok között van-e. A rekord akkor is a megadott helyre kerül, ha a sorszám a fájl végén túlra mutat, a fájl régi vége és a most leírt rekord közötti bájtok pedig automatikusan `b"\x00"` értéket kapnak. Az ilyen üres rekordok állapota se nem `_OKAY`, se nem `_DELETED`, így szükség esetén megkülönböztethetők.

```
def __getitem__(self, index):
    self.__seek_to_index(index)
    state = self.__fh.read(1)
    if state != _OKAY:
        return None
    return self.__fh.read(self.record_size)
```

A rekordok visszaolvasásakor négy különböző helyzet állhat elő, amire kell felkészülnünk. Az első, hogy a rekord nem létezik, vagyis a megadott sorszám a fájl végén túlra mutat. A második, hogy a rekord üres, a harmadik, hogy a rekord törölt, az utolsó pedig az, hogy a rekorddal minden rendben. Ha a rekord nem létezik, akkor a privát `__seek_to_index()` tagfüggvény `IndexError` kivételt vált ki. Egyébként bájtonként haladva a rekord elejére áll, és már olvashatjuk is az állapotjelző bájtot. Ha az állapot nem `_OKAY`, akkor vagy üres, vagy törölt a rekord; mindkét esetben `None` értéket adunk vissza, egyébként pedig magát a rekordot. (Az is járható út lenne, ha `None` érték visszaadása helyett valamilyen egyedi kivételt váltanánk ki az üres és a törölt rekordok esetében.)

```
def __seek_to_index(self, index):
    if self.auto_flush:
        self.__fh.flush()
    self.__fh.seek(0, os.SEEK_END)
    end = self.__fh.tell()
    offset = index * self.__record_size
    if offset >= end:
        raise IndexError("no record at index position {0}".format(
                                                        index))
    self.__fh.seek(offset)
```

A fenti kód egy segédtagfüggvény, amelyet más tagfüggvények használnak a fájlmutató adott sorszámú rekord elejére való állítására. Először megnézzük, hogy a megadott sorszám megfelelő határok közé esik-e. Ezt úgy tesszük meg, hogy a fájl végére megyünk (0 eltolás a fájl végétől), a `tell()` függvénnyel pedig megnézzük, hogy hová jutottunk. Ha a rekord eltolásértéke (a sorszám szorozva a valós rekordmérettel) nem kisebb a kapott értéknél, akkor a sorszám túl magas. Ilyenkor kiváltjuk a megfelelő kivételt. Ha a sorszám megfelelő, a megfelelő helyre állítjuk a fájlmutatót, és készen állunk a következő olvasási vagy írási műveletre.

```
def __delitem__(self, index):
    self.__seek_to_index(index)
    state = self.__fh.read(1)
    if state != _OKAY:
        return
```

```
        self.__fh.seek(index * self.__record_size)
        self.__fh.write(_DELETED)
        if self.auto_flush:
            self.__fh.flush()
```

Először a megfelelő helyre mozgatjuk a fájlmutatót. Ha a sorszáma megfelelő határok között van – vagyis ha nem kaptunk `IndexError` kivételt –, és a rekord nem üres, illetve nem törölt, akkor elvégezzük a rekord törlését. Az állapotjelző bájtot a `_DELETED` értékkel írjuk felül.

```
    def undelete(self, index):
        self.__seek_to_index(index)
        state = self.__fh.read(1)
        if state == _DELETED:
            self.__fh.seek(index * self.__record_size)
            self.__fh.write(_OKAY)
            if self.auto_flush:
                self.__fh.flush()
            return True
        return False
```

A fenti tagfüggvény először megkeresi a rekordot, és beolvassa a benne lévő állapotjelző bájtot. Ha a rekord törölt, az állapotjelző bájtot `_OKAY` értékkel írjuk felül, és a hívónak `True` értéket adunk vissza a siker jeleként. Egyéb esetekben (üres rekordoknál, illetve olyanoknál, amelyek nincsenek törölve) False értéket adunk vissza.

```
    def __len__(self):
        if self.auto_flush:
            self.__fh.flush()
        self.__fh.seek(0, os.SEEK_END)
        end = self.__fh.tell()
        return end // self.__record_size
```

Ez a tagfüggvény azt mondja meg, hogy hány rekord van egy adott bináris rekordokat tartalmazó fájlban. Úgy kapja meg a választ, hogy a fájlvég helyét – vagyis a fájlban lévő bájtok számát – elosztja a rekordmérettel.

Ezzel végeztünk a `BinaryRecordFile.BinaryRecordFile` osztály kínálta alapvető tagfüggvények tárgyalásával. Egyetlen dolgot kell még átgondolnunk: a fájl tömörítését, amivel az üres és a törölt rekordok eltávolítását végezzük. Lényegében két megközelítési lehetőségünk van. Az egyik az, hogy az üres és a törölt rekordokat felülírjuk a magasabb sorszámú helyen tárolt rekordokkal, és ha marad üres vagy törölt rekord a fájl végén, akkor csonkoljuk a fájlt. Az `inplace_compact()` tagfüggvény így dolgozik. A másik lehetőség, hogy azokat a rekordokat, amelyek se nem üresek, se nem töröltek, ideiglenes fájlba másoljuk, amit utóbb átnevezünk az eredeti fájl nevére. Az ideiglenes fájl használata főleg akkor kényelmes, ha meg szeretnénk tartani egy biztonsági másolatot. A `compact()` tagfüggvény ekképpen működik.

Először az `inplace_compact()` tagfüggvény működését tekintjük át, két részletben.

```
def inplace_compact(self):
    index = 0
    length = len(self)
    while index < length:
        self.__seek_to_index(index)
        state = self.__fh.read(1)
        if state != _OKAY:
            for next in range(index + 1, length):
                self.__seek_to_index(next)
                state = self.__fh.read(1)
                if state == _OKAY:
                    self[index] = self[next]
                    del self[next]
                    break
            else:
                break
        index += 1
```

Minden egyes rekordot bejárunk, és bejáráskor mindegyik rekord állapotát is beolvassuk. Ha üres vagy törölt rekordot találunk, megkeressük a fájlban a következő olyan rekordot, amely se nem üres, se nem törölt. Amikor ilyenhez érünk, az üres, illetve a törölt rekordot felülírjuk vele, majd ezt töröljük. Ha nem találunk ilyet, akkor kilépünk az egész `while` ciklusból, hiszen elfogytak az olyan rekordok, amelyek se nem üresek, se nem töröltek.

```
self.__seek_to_index(0)
state = self.__fh.read(1)
if state != _OKAY:
    self.__fh.truncate(0)
else:
    limit = None
    for index in range(len(self) - 1, 0, -1):
        self.__seek_to_index(index)
        state = self.__fh.read(1)
        if state != _OKAY:
            limit = index
        else:
            break
    if limit is not None:
        self.__fh.truncate(limit * self.__record_size)
self.__fh.flush()
```

Ha az első rekord üres vagy törölt, akkor mindegyik üres vagy törölt: az előző kódrészlet ugyanis minden olyan rekordot, ami nem volt üres vagy törölt, a fájl elejére, a többit pedig a fájl végére mozgatta. Ilyen esetben egyszerűen nulla bájt hosszúságúra csonkoljuk a fájlt.

Ha találunk legalább egy olyan rekordot, amely nem üres és nem törölt, az utolsó rekordtól visszafelé haladva bejárjuk a fájlt – tudjuk ugyanis, hogy az üres és a törölt rekordok a fájl végére kerültek. A `limit` változó az első üres vagy törölt rekordra mutat (illetve `None` marad, ha sem üres, sem törölt rekordot nem találunk), a fájlt pedig a megfelelő helyen csonkoljuk.

A helyben végzett tömörítés mellett a másik lehetőség, hogy más fájlba másoljuk az adatokat – ez akkor lehet hasznos módszer, ha meg akarunk tartani egy biztonsági másolatot – mint ahogy a most következő `compact()` tagfüggvény ismertetéséből is kiderül.

```
def compact(self, keep_backup=False):
    compactfile = self.__fh.name + ".$$$"
    backupfile = self.__fh.name + ".bak"
    self.__fh.flush()
    self.__fh.seek(0)
    fh = open(compactfile, "wb")
    while True:
        data = self.__fh.read(self.__record_size)
        if not data:
            break
        if data[:1] == _OKAY:
            fh.write(data)
    fh.close()
    self.__fh.close()
    os.rename(self.__fh.name, backupfile)
    os.rename(compactfile, self.__fh.name)
    if not keep_backup:
        os.remove(backupfile)
    self.__fh = open(self.__fh.name, "r+b")
```

A tagfüggvény két fájlt hoz létre, egy tömörítettet, illetve az eredeti fájl biztonsági másolatát. A tömörített fájl kezdetben ugyanazt a fájlnevet kapja, mint az eredeti, de a fájlnév végére `.$$$` kiterjesztés kerül. Ehhez hasonlóan a biztonsági másolat neve is megegyezik az eredetivel, de a végére a `.bak` kiterjesztés kerül. A létező fájlt rekordonként olvassuk be, és az olyan rekordokat, amelyek se nem üresek, se nem töröltek, átírjuk a tömörített fájlba. (Figyeljük meg, hogy minden esetben az igazi rekordot, vagyis a felhasználó által használható részt és az azt megelőző állapotjelző bájtot írjuk át.)

A `bytes` és a `bytearray` adattípus ☜ 297

Az `if data[:1] == _OKAY:` sor méltó a figyelmünkre. A `data` objektum és az `_OKAY` objektum egyaránt a `bytes` osztályba tartozik. A `data` objektum első bájtját az (egybájtos) `_OKAY` objektummal szeretnénk összehasonlítani. Ha egy `bytes` objektum szeletét vesszük, szintén `bytes` objektumot kapunk, ha azonban csak egyetlen bájtot veszünk belőle, mondjuk a `data[0]` utasítással, akkor `int` objektumot – a bájt értékét. Esetünkben a `data` objektumnak tehát egyetlen bájtnyi részsorozatát – az első, állapotjelző bájtot – hasonlítjuk

össze az egybájtos `_OKAY` objektummal. (Ennek a műveletnek az elvégzésére a másik módszer az lenne, ha a `data[0] == _OKAY[0]` összehasonlítást végeznénk el, két `int` osztályú értéket hasonlítva össze.)

Ha mindezzel megvagyunk, az eredeti fájlt átnevezzük biztonsági másolattá, a tömörített fájl pedig megkapja az eredeti fájl nevét. Utána eltávolítjuk a biztonsági másolatot, ha a `keep_backup` változó értéke `False` (ez az alapértelmezett). Végül megnyitjuk az olvasásra és írásra egyaránt készen álló tömörített fájlt – ami mostanra az eredeti nevét viseli.

A `BinaryRecordFile.BinaryRecordFile` osztály meglehetősen alacsonyszintű, de szolgálhat alapként olyan magasabb szintű osztályok számára, amelyeknek rögzített méretű rekordokból álló véletlen elérésű fájlokat kell kezelniük – ahogy ez a következő részből is kiderül.

Példa: A BikeStock modul osztályai

A `BikeStock` modul egy `BinaryRecordFile.BinaryRecordFile` objektumot használ egy egyszerű raktárkészlet-kezelő osztály megvalósításához. A raktárkészlet elemei kerékpárok; az egyes kerékpárokat `BikeStock.Bike` objektumpéldányok ábrázolják, a teljes raktárkészletet pedig egy `BikeStock.BikeStock` objektumpéldány.

A `BikeStock.BikeStock` osztály egy kerékpár-azonosító kulcsú szótárat képez le egy `BinaryRecordFile.BinaryRecordFile` osztályú fájlobjektumba. A szótár értékei a rekordok sorszámai. Íme a modul használatát bemutató rövid példa, amelyből ráérezhetünk az osztályok működésére:

```
bicycles = BikeStock.BikeStock(bike_file)
value = 0.0
for bike in bicycles:
    value += bike.value
bicycles.increase_stock("GEKKO", 2)
for bike in bicycles:
    if bike.identity.startswith("B4U"):
        if not bicycles.increase_stock(bike.identity, 1):
            print("stock movement failed for", bike.identity)
```

A fenti kódrészlet a kerékpár-raktárkészletet tartalmazó fájl megnyitása után bejárja a benne lévő rekordokat, és a kerékpárok értékeinek és számainak szorzatait összegezve deríti ki a fájlban lévő összes kerékpár értékét. Ha ezzel elkészült, a raktáron lévő „GEKKO" kerékpárok számát kettővel, és minden „B4U" kezdetű kerékpártípusét eggyel növeli. Az összes művelet a lemezen történik meg, vagyis minden olyan folyamat, amely a fájl beolvasásával nyer információt, mindig az aktuális adatokat kapja meg.

Bár a `BinaryRecordFile.BinaryRecordFile` fájlobjektum sorszámokkal dolgozik, a `BikeStock.BikeStock` objektum kerékpár-azonosítókat használ. A megfeleltetést maga a `BikeStock.BikeStock` objektum végzi, mégpedig egy olyan szótárral felszerelkezve, amely a kerékpár-azonosítókat a fájlbeli rekordok sorszámainak felelteti meg.

Vizsgálódásunkat a `BikeStock.Bike` osztály `class` utasítást tartalmazó soránál és az előkészítőnél kezdjük, majd áttekintünk néhány fontosabb tagfüggvényt a `BikeStock.BikeStock` osztályból, végül pedig a kódnak azt a részét nézzük meg, amelyik a `BikeStock.Bike` objektumokat köti össze azokkal a bináris rekordokkal, amelyek az objektumokat a `BinaryRecordFile.BinaryRecordFile` objektumban ábrázolják. (A kód egésze a `BikeStock.py` fájlban található.)

```
class Bike:

    def __init__(self, identity, name, quantity, price):
        assert len(identity) > 3, ("invalid bike identity '{0}'"
                                   .format(identity))
        self.__identity = identity
        self.name = name
        self.quantity = quantity
        self.price = price
```

A `Bike` objektumok minden jellemzője tulajdonságként érhető el: a kerékpár-azonosító (`self.__identity`) a csak olvasható `Bike.identity` tulajdonságként, az összes többi pedig olvasható-írható tulajdonságként – néhány érvényességellenőrzést követően. Ezen felül a `Bike.values` csak olvasható tulajdonság adja vissza a mennyiség és ár szorzatát. (A tulajdonságok megvalósítását nem közöltük, hiszen korábban már láttunk hasonló kódot.)

A `BikeStock.BikeStock` osztályban találunk a `Bike` objektumok kezelésére való tagfüggvényeket, mégpedig olyanokat, amelyek a `Bike` objektumok írható tulajdonságait használják.

```
class BikeStock:

    def __init__(self, filename):
        self.__file = BinaryRecordFile.BinaryRecordFile(filename,
                                                        _BIKE_STRUCT.size)
        self.__index_from_identity = {}
        for index in range(len(self.__file)):
            record = self.__file[index]
            if record is not None:
                bike = _bike_from_record(record)
                self.__index_from_identity[bike.identity] = index
```

A `BikeStock.BikeStock` osztály olyan egyedi gyűjteményes adattípust valósít meg, amely egyszerre kezel egy bináris rekordokból álló fájlt (`self.__file`) és egy szótárat (`self.__index_from_identity`), amelynek a kulcsai kerékpár-azonosítók, az értékei pedig rekordok sorszámai.

Ha a fájlt megnyitottuk (vagy létrehoztuk, ha korábban nem létezett), bejárjuk a tartalmát – már ha van benne valami. Kiolvassuk az egyes kerékpárokat, és `bytes` objektumból a privát `_bike_from_record()` függvény segítségével `BikeStock.Bike` objektummá alakítjuk őket. A kerékpárok azonosítói és sorszámai a `self.__index_from_identity` szótárba kerülnek.

```
def append(self, bike):
    index = len(self.__file)
    self.__file[index] = _record_from_bike(bike)
    self.__index_from_identity[bike.identity] = index
```

Új kerékpár elhelyezése tulajdonképpen a megfelelő rekordsorszám megtalálását, illetve a kerékpár bináris ábrázolásának a sorszámhoz tartozó rekordba töltését jelenti. Arra is figyelünk, hogy a `self.__index_from_identity` szótárat frissítsük.

```
def __delitem__(self, identity):
    del self.__file[self.__index_from_identity[identity]]
```

Egy kerékpárt tartalmazó rekord törlése egyszerű: az azonosító alapján kikeressük a hozzá tartozó rekord sorszámát, és a sorszámnak megfelelő rekordot töröljük.
A `BikeStock.BikeStock` osztályban nem használjuk fel
a `BinaryRecordFile.BinaryRecordFile` osztály törlés visszavonására való képességét.

```
def __getitem__(self, identity):
    record = self.__file[self.__index_from_identity[identity]]
    return None if record is None else _bike_from_record(record)
```

A kerékpárokhoz tartozó rekordok megtalálása kerékpár-azonosító alapján történik.
Ha a keresett azonosító nem létezik, a `self.__index_from_identity` szótáron lefutó keresés `KeyError` kivételt vált ki, ha pedig a rekord üres, illetve törölt,
a `BinaryRecordFile.BinaryRecordFile` objektum `None` értékkel tér vissza. Ha azonban megtaláljuk a rekordot, akkor azt adjuk vissza, `BikeStock.Bike` objektumként.

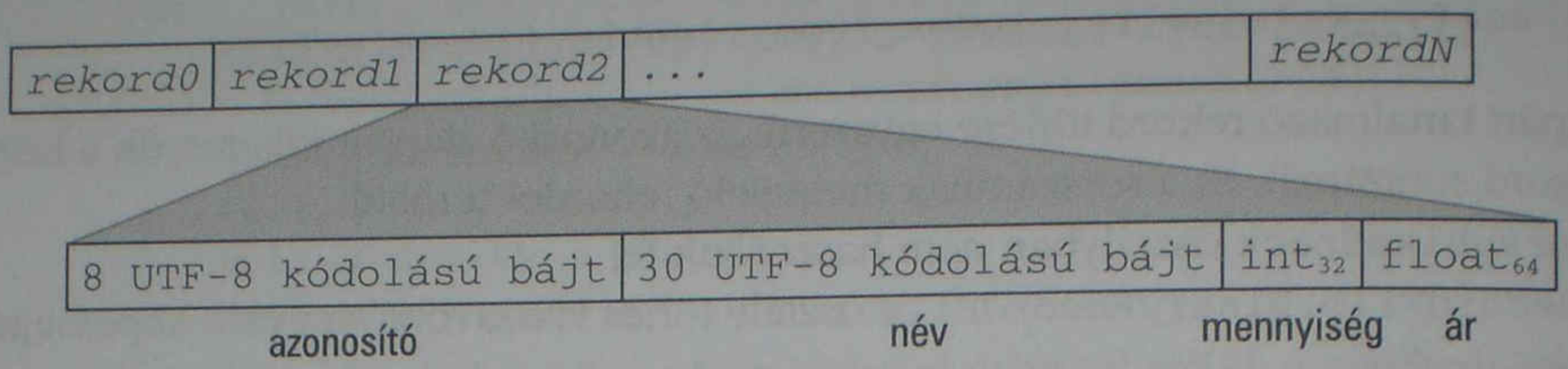
```
def __change_stock(self, identity, amount):
    index = self.__index_from_identity[identity]
    record = self.__file[index]
    if record is None:
        return False
    bike = _bike_from_record(record)
    bike.quantity += amount
    self.__file[index] = _record_from_bike(bike)
    return True

increase_stock = (lambda self, identity, amount:
                  self.__change_stock(identity, amount))
decrease_stock = (lambda self, identity, amount:
                  self.__change_stock(identity, -amount))
```

A privát `__change_stock()` tagfüggvény az `increase_stock()` és a `decrease_stock()` tagfüggvény megvalósítását végzi. A kerékpár rekordjának sorszáma alapján nyers bináris rekordot olvas be, majd az adatot `BikeStock.Bike` objektummá alakítja, az objektumpéldányon elvégzi a szükséges módosítást, és a fájlban lévő rekordot felülírja a frissített `Bike` objektum bináris ábrázolásával. (Van egy `__change_bike()` tagfüggvény is, amely a `change_name()` és a `change_price()` tagfüggvényeket valósítja meg. A könyvben nem mutatjuk be őket, mivel az előzőleg tárgyalt tagfüggvényekhez igen hasonlók.)

```
def __iter__(self):
    for index in range(len(self.__file)):
        record = self.__file[index]
        if record is not None:
            yield _bike_from_record(record)
```

A fenti tagfüggvénynek köszönhető, hogy a `BikeStock.BikeStock` objektumok a listákhoz hasonlóan bejárhatók. A bejárás egyes lépéseinél `BikeStock.Bike` objektumokat kapunk vissza, az üres és a törölt rekordokat átlépjük.



7.6. ábra

A kerékpárok tárolására szolgáló rekordokat tartalmazó fájl logikai szerkezete

A privát `_bike_from_record()` és `_record_from_bike()` függvény elrejti a `BikeStock.Bike` osztály bináris ábrázolását a kerékpárok gyűjteményét tároló `BikeStock.BikeStock` osztály elől. A kerékpárok tárolására szolgáló rekordokat tartalmazó fájl logikai szerkezete a 7.6. ábrán látható. A fizikai szerkezet kicsit eltér ettől: minden rekordot egy állapotjelző bájt előz meg benne.

```
_BIKE_STRUCT = struct.Struct("<8s30sid")

def _bike_from_record(record):
    ID, NAME, QUANTITY, PRICE = range(4)
    parts = list(_BIKE_STRUCT.unpack(record))
    parts[ID] = parts[ID].decode("utf8").rstrip("\x00")
    parts[NAME] = parts[NAME].decode("utf8").rstrip("\x00")
    return Bike(*parts)

def _record_from_bike(bike):
    return _BIKE_STRUCT.pack(bike.identity.encode("utf8"),
                             bike.name.encode("utf8"),
                             bike.quantity, bike.price)
```

Amikor egy bináris rekordot `BikeStock.Bike` objektummá alakítunk, először az `unpack()` függvénytől visszakapott rendezett sorozatot alakítjuk át egy listává. Így lehetségessé válik az elemek módosítása, esetünkben az UTF-8 kódolású bájtok olyan karakterlánccá való alakítása, ahonnan a `0x00` értékű bájtok már hiányoznak. Ezt követően a sorozatkicsomagoló művelet (*) használatával átadjuk a részeket a `BikeStock.Bike` objektum előkészítőjének. Az adatok becsomagolása lényegesen egyszerűbb, csak arra kell figyelnünk, hogy a karakterláncokat UTF-8 karakterkódolású bájtokká alakítsuk át.

A modern asztali számítógépekben a memória mennyiségének és a lemezek sebességének növekedésével a véletlen elérésű bináris állományok használatának szükségessége csökken. Ha mégis ilyesmire van szükségünk, gyakran az a legegyszerűbb, ha valamilyen adatbázis-fájlt vagy egy SQL-adatbázist használunk. Ugyanakkor léteznek olyan rendszerek – például beágyazott vagy erőforrásszegény rendszerek –, amelyek esetében a fenti képességekkel bíró programok hasznosak lehetnek.

Összefoglalás

A fejezet a gyűjteményes adattípusok fájlba mentésére és onnan történő visszatöltésére szolgáló legelterjedtebben használt módszereket mutatta be. Láttuk, hogy milyen egyszerű a befőttek használata, és miként kezelhető mind a tömörített, mind a tömörítetlen fájlformátum anélkül, hogy előre tudnánk, hogy tömörített fájllal állunk-e szemben.

Láttuk, hogy a bináris adatok írása és olvasása nagy figyelmet igényel, és azt is láttuk, hogy amennyiben változó hosszúságú karakterláncokat kell kezelnünk, a kód igen hosszúra nyúlhat. Azt is megtanultuk azonban, hogy a bináris fájlok használata rendszerint a lehető legkisebb fájlméretet eredményezi, a leggyorsabb írás és olvasás mellett. Tudjuk azt is, hogy a fájltípus azonosítása végett fontos valamilyen mágikus szám használata, a formátum későbbi megváltoztatását pedig megkönnyíti, ha változatszámot is írunk a fájlba.

A fejezetben láttuk, hogy a felhasználók legkönnyebben a sima szöveges formátumú fájlokat olvassák, és ha ügyesen alakítottuk ki az adatfájl szerkezetét, akkor egyszerűen tudunk egyéb eszközöket is létrehozni az adatok kezelésére. Mindazonáltal a szöveges adatok értelmezése bonyolult lehet. Egyaránt láttunk a szöveges adatok kézi és szabályos kifejezéseket használó beolvasására is példát.

Az XML nagyon népszerű adatcsere-formátum, és általában legalább az adatok mentésére és betöltésére érdemes használni – még akkor is, ha a rendes formátum bináris vagy szöveges. Láttuk, hogy miként írható kézzel egy XML-fájl – láttuk a jellemzők értékeinek és a szöveges adatoknak a levédését is –, és azt is, hogy miként tehető meg mindez elemfa vagy DOM használatával. Megismertük az XML formátumnak a Python alapkönyvtárában lévő elemfával, a DOM-mal és a SAX-szal történő feldolgozását.

A fejezet utolsó részében megfigyelhettük egy általános célú, rögzített méretű rekordokat tartalmazó véletlen elérésű bináris fájlok kezelését végző osztály létrehozását. Megfigyelhettük az általános célú osztály egy adott helyzetben történő használatát is.

A fejezettel a Python-programozás alapjainak végére értünk. Ha most abbahagyjuk az olvasást, az eddig tanultak felhasználásával tökéletesen jó Python-programokat írhatunk. De nagy kár lenne éppen most abbahagyni az olvasást, hiszen olyan sok még a Pythonban rejlő lehetőség, a kód egyszerűsítésére és rövidítésére szolgáló apróságoktól

azokig az igazi agytornát jelentő lehetőségekig, amelyekről még akkor is érdemes legalább egy keveset tudnunk, ha csak ritkán fogjuk használni őket. A következő fejezetben ismét az eljárásközpontú és az objektumközpontú programozásban mélyedünk el, és megízleljük a függvényközpontú programozást is. A rákövetkező fejezetekben pedig olyan általánosabb érvényű témákkal foglalkozunk, mint a többszálúság, a hálózatkezelés, az adatbázis-programozás, a szabályos kifejezések és a grafikus felhasználói felület programozása.

Gyakorlatok

Az első gyakorlat olyan egyszerű, bináris rekordokból álló fájlt kezelő modul megalkotása, amelyet a fejezet korábbi részében is bemutattunk – olyané, amelynek a rekordméretét a felhasználó adhatja meg. A második gyakorlat a `BikeStock` modul módosításával kapcsolatos – az új modul használatba vételét kell megoldanunk. A harmadik gyakorlatban teljesen új programot kell írnunk – a fájlkezelés meglehetősen magától értetődő, de a kimenet formázása némely tekintetben jelentős kihívásokat rejt magában.

1. Alkossuk meg a `BinaryRecordFile` modul új, egyszerűbb változatát – olyat, amelyik nem használ állapotjelző bájtot. Ebben a változatban a felhasználó a valós rekordméretet adja meg. Az új rekordok elhelyezése az új, `append()` nevű tagfüggvénnyel történik; a tagfüggvény a fájl végére mozgatja a fájlmutatót, és leírja az új rekordot. A `__setitem__()` tagfüggvény csak a már létező rekordok cseréjét engedheti meg; a feladat megoldásában nagy segítséget jelenthet a `__seek_to_index()` tagfüggvény használata. Most, hogy nincs állapotjelző bájt, a `__getitem__()` tagfüggvény mindössze három sorocskára zsugorodik. A `__delitem__()` tagfüggvényt teljesen újra kell írnunk, hiszen a keletkező rés feltöltésére minden rekordot előrébb kell hoznia. Mindez nagyjából fél tucat sorral megoldható, de kicsit el kell gondolkodni a megoldáson. Az `undelete()` tagfüggvényt el kell távolítani, lévén a művelet nem támogatott. A `compact()` és az `inplace_compact()` tagfüggvényt is el kell távolítani, de ezeket azért, mert a létük okafogyottá vált.
   Ahogy említettük, a változások az eredeti változathoz képest alig húsz új, és legalább hatvan törölt sort jelentenek – nem számítva a dokumentációs teszteket. A megoldást a `BinaryRecordFile_ans.py` fájlban találjuk.
2. Ha meggyőződtünk róla, hogy a `BinaryRecordFile` osztály egyszerűbb változata működik, készítsünk másolatot a `BikeStock.py` fájlról, és módosítsuk úgy, hogy az osztály új változatát használja. Ez alig pár sor megváltoztatását jelenti.
   A megoldást a `BikeStock_ans.py` fájlban találjuk.
3. A bináris formátumú fájlokban a hibakeresés néha bonyolult. Az egyik eszköz, ami a segítségünkre lehet, a bináris fájl tartalmát írja ki tizenhatos számrendszerben. Alkossunk olyan programot, amely a következő súgószöveget írja a képernyőre:

```
Usage: xdump.py [options] file1 [file2 [... fileN]]

Options:
  -h, --help            show this help message and exit
  -b BLOCKSIZE, --blocksize=BLOCKSIZE
                        block size (8..80) [default: 16]
-d, --decimal           decimal block numbers [default: hexadecimal]
-e ENCODING, --encoding=ENCODING
                        encoding (ASCII..UTF-32) [default: UTF-8]
```

Ha van egy olyan `BinaryRecordFile` formátumú fájlunk, amely a `"<i10s"` szerkezet (little-endian bájtsorrend, négybájtos előjeles egész, tízbájtnyi bájtsorozat) használatával tárolja a rekordokat, a blokkméretet a rekord méretéhez (állapotjelző bájttal együtt tizenöt bájt) igazítva tiszta képet kapunk a fájl tartalmáról. Például:

```
xdump.py -b15 test.dat
Block     Bytes                                     UTF-8 characters
--------  ----------------------------------------  ----------------
00000000  02000000 00416C70 68610000 000000  .....Alpha.....
00000001  01140000 00427261 766F0000 000000  .....Bravo.....
00000002  02280000 00436861 726C6965 000000  .(...Charlie...
00000003  023C0000 0044656C 74610000 000000  .<...Delta.....
00000004  02500000 00456368 6F000000 000000  .P...Echo......
```

Minden bájt kétszámjegyű, tizenhatos számrendszerű számként jelenik meg, a négybájtos (vagyis nyolc tizenhatos számrendszerű számjegynyi) részek közötti szóközök csak az olvashatóságot könnyítik meg. Látható, hogy a második rekord – a „Bravo” – törölt, hiszen az állapotjelző bájtja `0x01`, a nem üres és nem törölt rekordokat jelző `0x02` helyett.

A parancssori paraméterek feldolgozására használjuk az `optparse` modult. (A paraméter „típusának” megadásával az `optparse` modul rávehető a karakterláncok egész számmá alakítására.) A fejlécek blokkméretének megfelelő kiírása meglehetős nehézséget jelenthet, csakúgy, mint az utolsó oszlopban lévő karakterek megfelelő elhelyezése. Mindenképpen hajtsunk végre teszteket különböző blokkméretekkel (például 8, 9, 10, …, 40). Azt se feledjük, hogy a változó hosszúságú fájlok esetében az utolsó blokk esetleg rövidebb a többinél. Ahogy a példa mutatja, használjunk pontokat a nem nyomtatható karakterek helyének kitöltésére.

A program két függvénnyel, hetven sornál rövidebben is megírható. A megoldást az `xdump.py` fájlban találjuk.

8

Haladó programozási eljárások

- Magasabb szintű eljárásközpontú programozás
- Magasabb szintű objektumközpontú programozás
- Függvényközpontú programozás

Ebben a fejezetben a különböző programozási módszerek széles skáláját tekintjük át, illetve a Python nyelv sok újabb – gyakran elmélyültebb tudásszintet képviselő – utasításformájával ismerkedünk meg. A fejezet anyagának bizonyos részei meglehetős kihívást jelentenek, mindazonáltal ne feledjük, hogy a legmagasabb szintű módszerekre ritkán van szükség.
Így ezúttal elegendő lehet, ha éppen csak futólag ismerkedünk meg a témával, bepillantást nyerünk a lehetőségekbe, és az alapos olvasás várhat, amíg szükségünk nem lesz valamelyik, a következőkben tárgyalt módszerre.

A fejezet első részében mélyebbre ássuk magunkat a Python eljárásközpontú (procedurális) lehetőségeibe. Először bemutatjuk, hogy újszerű megközelítésben miként használható a már elsajátított tudásunk, ezt követően pedig visszatérünk a generátoroknak a 6. fejezetben már érintett témájához. A rész a dinamikus programozás bemutatásával folytatódik – vagyis futásidőben fogjuk betöltetni a moduljainkat, miáltal lehetővé válik, hogy futásidőben döntsünk a futtatott kódról. Ezt követően visszakanyarodunk a helyi (beágyazott) függvényekhez, de tárgyaljuk a `nonlocal` kulcsszó és az önhívó függvények használatát is.
Az előzőekben már láttuk, hogy a Python előzetesen megadott díszítői miként használhatók. Ebben a részben azt is megtanuljuk, hogy miként hozhatunk létre saját díszítőket. A részt függvénymagyarázattal zárjuk.

A második rész az objektumközpontú (objektumorientált) programozással kapcsolatos, teljesen új anyagot tárgyal. Először bemutatjuk a `__slots__` osztályszintű változót, amely az egyes objektumok által felhasznált memória csökkentését segíti elő. Ez után azt tanuljuk meg, hogy miként lehet tulajdonságok használata nélkül hozzáférni jellemzőkhöz. Ebben a részben tárgyaljuk még a funktorokat (olyan objektumokról van szó, amelyeket a függvényekhez hasonlóan lehet meghívni), és a környezetkezelőket – ez utóbbiakat a `with` kulcsszóval összefüggésben használják, és sok esetben (például fájlkezeléskor) jók arra, hogy a `try...except...finally` szerkezet helyett az egyszerűbb `try...except` szerke-

zetet használhassuk. A szakasz azt is bemutatja, hogy miként hozhatók létre saját, egyedi környezetkezelőink, és bemutat néhány magasabb szintű objektumközpontú képességet, például az osztályszintű díszítőket, az elvont alaposztályokat, a többszörös öröklést és a metaosztályokat.

A harmadik rész megismertet bennünket a függvényközpontú (funkcionális) programozás néhány alapelvével, és bemutat néhány hasznos, a `functools`, az `itertools` és az `operator` modulban található függvényt. Azt is megtárgyaljuk, hogy miként használható a részleges függvényközpontú programozás a programkód egyszerűsítésére.

Az eddigi fejezetek összességében a „Python-alapismereteket" tartalmazták. Mindazt, amit eddig tárgyaltunk, ez a fejezet egyszeriben „luxusminőségű" ismeretté alakítja át. Minden eredeti módszerünk és utasításformánk megmarad, sőt kiegészül olyanokkal, amelyek a programozást könnyebbé, rövidebbé és hatékonyabbá teszik. Néhány eszköz felcserélhető egymással, vannak például olyan feladatok, amelyeket osztályszintű díszítőkkel és metaosztályokkal egyaránt el lehet végezni. Más eszközök esetében – ilyenek például a leírók – több felhasználási mód létezik, és ezek a hatásukban is különböznek. A következőkben tárgyalt eszközök közül néhányat – például a környezetkezelőket – szinte egyfolytában használni fogjuk, mások viszont talonba kerülnek, és ott várják ki azokat a bizonyos helyzeteket, amelyekben ők jelentik a tökéletes megoldást.

Magasabb szintű eljárásközpontú programozás

Ennek a szakasznak a legnagyobb része az eljárásközpontú programozással kapcsolatos újabb eszközökkel foglalkozik. A legelső rész annyiban különbözik a többitől, hogy anélkül ismertet meg bennünket hasznos programozási módszerekkel, hogy egyetlen új utasításformát is meg kellene ismernünk – csak a már elsajátítottakra támaszkodunk.

Elágazások létrehozása szótárak segítségével

Ahogy már korábban is jeleztük, a Pythonban a függvények minden más elemhez hasonlóan objektumok, a függvény neve pedig a függvényre mutató objektumhivatkozás. Ha egy függvény nevét zárójelek nélkül írjuk le, a Python számára nyilvánvaló, hogy az objektumhivatkozásra gondolunk. Az ilyen objektumhivatkozások a többihez hasonlóan adhatók át a programokban. Ennek a ténynek az ismeretében a sok `elif` záradékot tartalmazó `if` utasítások egyetlen függvény hívásával helyettesíthetők. A 11. fejezetben át fogjuk tekinteni a `dvds-dbm.py` nevű interaktív konzolprogramot, amelynek a menüje az alábbi:

```
(A)dd (E)dit (L)ist (R)emove (I)mport e(X)port (Q)uit
```

A programban van egy függvény, amely a felhasználó választásának beolvasásáért felel, és amely csak érvényes választást ad vissza, esetünkben az „a", „e", „l", „r", „i", „x" és „q" karakterek valamelyikét. Az alábbiakban két egyenértékű kódrészletet ismertetünk; mindkettő a felhasználótól érkező bemenetnek megfelelő függvény hívását végzi el.

```
if action == "a":
    add_dvd(db)
elif action == "e":
    edit_dvd(db)
elif action == "l":
    list_dvds(db)
elif action == "r":
    remove_dvd(db)
elif action == "i":
    import_(db)
elif action == "x":
    export(db)
elif action == "q":
    quit(db)
```

```
functions = dict(a=add_dvd, e=edit_dvd,
l=list_dvds, r=remove_dvd,
i=import_, x=export, q=quit)
functions[action](db)
```

A bemenetet az `action` nevű változóban egyetlen karakterből álló karakterláncként tároljuk, a használandó adatbázis pedig a `db` nevű változóba kerül. Az `import_()` függvény nevében azért van aláhúzás, mert így különböztetjük meg a beépített `import` utasítástól.

A jobb oldalon levő kódrészletben létrehozunk egy szótárat, amelynek a kulcsai a felhasználó által megadható érvényes választási lehetőségek, az értékei pedig objektumhivatkozások. A második utasításban beillesztjük az adott műveletnek megfelelő objektumhivatkozást, és a függvényhívást végző műveleti jellel, vagyis a zárójellel meghívjuk a hivatkozott függvényt. Ebben a példában a `db` paramétert adjuk át.

A jobb oldalon lévő kódrészlet nemcsak lényegesen rövidebb, mint a bal oldali, de könnyen megbirkózik a méretezésből eredő problémákkal is (akár sokkal több elemű szótárat is használhatunk), és teszi mindezt anélkül, hogy a teljesítménye megváltozna. Nem így a bal oldali kódrészlet, amelynek a sebessége attól függ, hogy hány elágazást kell kiértékelni, amíg megtaláljuk azt a függvényt, amelyet meg kell hívnunk.

Az előző fejezetben megismert `convert-incidents.py` program is a fent ismertetett módszert használja az `import_()` tagfüggvényében, amint az a tagfüggvény alábbi részletéből is kiderül:

```
call = {(".aix", "dom"): self.import_xml_dom,
        (".aix", "etree"): self.import_xml_etree,
        (".aix", "sax"): self.import_xml_sax,
        (".ait", "manual"): self.import_text_manual,
        (".ait", "regex"): self.import_text_regex,
        (".aib", None): self.import_binary,
        (".aip", None): self.import_pickle}
result = call[extension, reader](filename)
```

A teljes tagfüggvény mindössze tizenhárom sorból áll, az `extension` paramétert magában a tagfüggvényben számítjuk ki, a `reader` paramétert pedig a hívó adja át. A szótár kulcsai kételemű rendezett sorozatok, az értékek pedig tagfüggvények. Ha `if` utasításokat használtunk volna, a kód huszonkét sornyi lenne, a méretezés pedig mindig gondot okozna.

Generátorkifejezések és -függvények

Generátorfüggvények 277 ☜

A generátorfüggvényekkel és -tagfüggvényekkel még a 6. fejezetben ismerkedtünk meg. Lehetőség van generátorkifejezések létrehozására is, amelyek nyelvtanilag szinte teljesen megegyeznek a listaértelmezésekkel, a különbség mindössze annyi, hogy szögletes zárójelek helyett egyszerű zárójelek közé kell írni őket. Az alábbi utasításformák használhatók:

```
(kifejezés for elem in bejárható_objektum)
(kifejezés for elem in bejárható_objektum if feltétel)
```

Az előző fejezetben `yield` kifejezések használatával hoztunk létre olyan tagfüggvényeket, amelyek adatszerkezetek bejárására szolgálnak. Az alábbiakban következik két egyenértékű kódrészlet, amelyekkel azt mutatjuk be, hogy egy `yield` kifejezés miként használható generátorként, ha `for...in` ciklusba helyezzük:

```
def items_in_key_order(d):
    for key in sorted(d):
        yield key, d[key]
```

```
def items_in_key_order(d):
    return ((key, d[key])
            for key in sorted(d))
```

Mindkét függvény olyan generátort ad vissza, amely egy olyan listát állít elő, amelynek az elemei az adott szótár kulcs-érték párjai. Ha minden elemre egyszerre van szükségünk, akkor a függvényektől visszakapott generátort átadhatjuk a `list()` vagy a `tuple()` függvénynek. Egyéb esetekben a generátor bejárásával kapjuk meg a szükséges elemeket.

A generátorok a „lusta” kiértékelés eszközei lehetnek, ugyanis csak azokat az értékeket számítják ki, amelyekre ténylegesen szükség van. Ez bizonyos körülmények között hatékonyabb, mintha mondjuk például egy lista összes értékét egy menetben számítanánk ki. Néhány generátor annyi értéket állít elő, amennyit kérünk tőle – felső határ nincs. Például:

```
def quarters(next_quarter=0.0):
    while True:
        yield next_quarter
        next_quarter += 0.25
```

A függvény a 0.0, 0.25, 0.5 stb. értékeket állítja elő örökkön örökké. A probléma megoldására az alábbi módon használhatunk generátort:

```
result = []
for x in quarters():
result.append(x)
if x >= 1.0:
break
```

A `break` utasítás létfontosságú: nélküle sohasem fejeződne be a `for...in` ciklus. Az eredménylista: `[0.0, 0.25, 0.5, 0.75, 1.0]`.

Minden alkalommal, amikor a `quarters()` függvényt meghívjuk, egy olyan generátort kapunk, amely 0.0-tól indul, és 0.25-tel emelkedik. Mi történik akkor, ha a generátor aktuális értékét vissza szeretnénk állítani a kezdőértékre? Lehetőség van arra, hogy a generátornak értéket adjunk át, mint ahogy azt a generátort megvalósító függvény alábbi, új változata is tanúsítja:

```
def quarters(next_quarter=0.0):
    while True:
    received = (yield next_quarter)
    if received is None:
        next_quarter += 0.25
    else:
        next_quarter = received
```

A `yield` kifejezés az egyes értékeket a hívónak adja vissza. Ezen felül, amikor a hívó a generátor `send()` tagfüggvényét hívja, az elküldött értéket a generátorfüggvény a `yield` kifejezés eredményeként kapja meg. Az alábbi példa az új generátorfüggvény használatát mutatja be:

```
result = []
generator = quarters()
while len(result) < 5:
    x = next(generator)
    if abs(x - 0.5) < sys.float_info.epsilon:
        x = generator.send(1.0)
    result.append(x)
```

Létrehozunk egy a generátorra hivatkozó változót, majd a beépített `next()` függvényt hívjuk meg, amely a neki átadott generátorból kinyeri a következő elemet. (Ugyanezt a hatást érhetjük el, ha a generátor `__next__()` nevű, különleges szerepű tagfüggvényét hívjuk meg. Ebben az esetben az utasítás formája `x = generator.__next__()`.) Ha az érték 0.5-del egyezik meg, akkor az 1.0 értéket küldjük át a generátornak (amely azonnal ugyanezt az értéket állítja elő). Ezúttal az eredménylista: `[0.0, 0.25, 1.0, 1.25, 1.5]`.

A következőkben a `magic-numbers.py` nevű programot fogjuk áttekinteni, amely a parancssorban megadott fájlokat dolgozza fel. Sajnos a Windows héjprogramja (a `cmd.exe`) nem kezeli tökéletesen a helyettesítő karaktereket (vagyis hiányzik belőle az úgynevezett *file globbing* nevű képesség), így aztán, ha egy programot Windowson a *.* paraméterrel futtatunk, a `sys.argv` lista – az aktuális könyvtárban lévő összes fájl helyett – betű szerint a „*.*” karakterlistát kapja meg. Ezt a problémát úgy oldjuk meg, hogy kétféle `get_files()` függvényt hozunk létre – az egyiket a Windows, a másikat a Unix számára. Mindkét függvény generátort használ. Íme a kód:

```
if sys.platform.startswith("win"):
    def get_files(names):
        for name in names:
            if os.path.isfile(name):
                yield name
            else:
                for file in glob.iglob(name):
                    if not os.path.isfile(file):
                        continue
                    yield file
else:
    def get_files(names):
        return (file for file in names if os.path.isfile(file))
```

A függvény paramétere mindkét esetben egy fájlneveket tartalmazó lista, például: `sys.argv[1:]`.

Windowson a függvény bejárja a listában szereplő neveket, és minden egyes fájlnév esetében előállítja magának a fájlnevet. A nem fájlokat takaró nevek esetében (ezek rendszerint könyvtárak) a `glob` modul `glob.iglob()` függvényét használjuk arra, hogy a helyettesítő karakterek értelmezése során előálló fájlnévlistát tartalmazó bejárható objektumot adjunk vissza. Rendes nevek, például az `autoexec.bat` esetében egyetlen elemből (a névből) álló bejárható objektumot adunk vissza. Azoknak a neveknek az esetében, amelyekben helyettesítő karakterek vannak – ilyen például a `*.txt` –, olyan bejárható objektumot adunk vissza, amely tartalmazza a helyettesítő karakterekkel megadott paraméterre illeszkedő fájlneveket (esetünkben azokat, amelyeknek `.txt` a kiterjesztése). (Létezik a `glob.glob()` tagfüggvény is, amely bejárható objektum helyett listát ad vissza.)

Unixon a parancshéj elvégzi számunkra a helyettesítő karakterek illesztését, így aztán csak annyi a dolgunk, hogy visszaadjuk az átadott nevek alapján létrehozott generátort.*

A generátorfüggvények *társrutinok* (coroutine) létrehozására is használhatók. A társrutinok olyan függvények, amelyek több be- és kilépési ponttal bírnak (ezek a `yield` utasítást tartalmazó kifejezések), és amelyek bizonyos pontoknál (szintén `yield` utasítást tartalmazó kifejezésekről van szó) felfüggeszthetők, illetve folytathatók. A társrutinokat gyakran használják a többszálúság helyett; egyszerűbb és alacsonyabb terheléssel járó megoldást jelenthetnek. A Python Package Index webhelyéről (`pypi.python.org/pypi`) számos, a társrutinokkal foglalkozó modul tölthető le.

---

* A `glob.glob()` függvények a képességeiket tekintve nem lehetnek méltó társai mondjuk a Unix bash parancshéjának, ugyanis a `*`, a `?` és a `[]` műveletek mellett hiányzik belőlük a `{}` művelet támogatása is.

Dinamikus kódvégrehajtás és dinamikus betöltés

Vannak olyan helyzetek, amikor egyszerűbb, ha a kód közvetlen megírása helyett azt a kódot írjuk meg, amelyik előállítja a számunkra szükséges kódot. Vannak olyan helyzetek is, amelyekben kifejezetten hasznos, ha a kódot a felhasználó írja meg (gondolunk itt a táblázatkezelők függvényeire), amelyet majd a Python lefuttat. Így megtakarítható a beírt kód értelmezését végző programrészek megírása – ám azt ne felejtsük el, hogy az ilyen típusú közvetlen kódvégrehajtás természetesen biztonsági kockázatot rejt magában.
A dinamikus kódvégrehajtás másik felhasználási területe olyan bővítmények (beépülő modulok) használata, amelyek egy adott program képességeit hivatottak kiterjeszteni. A bővítmények használatának megvan az a hátránya, hogy a programunk nem tartalmazza beépítetten az összes szükséges képességet – ez a program telepítésekor jelenthet nehézségeket, hiszen magában rejti a bővítmények elvesztésének lehetőségét. Ugyanakkor a módszernek megvan az az előnye, hogy a bővítmények újabb változatai egymástól függetlenül telepíthetők, illetve a modulok telepítése a főprogramtól függetlenül is történhet – akár olyan esetekben is, amikor a programunkat előre nem tervezett képességekkel kell felruháznunk.

Dinamikus kódvégrehajtás

Egy kifejezés kiértékeltetésének legegyszerűbb módja, ha a 6. fejezetben megismert, beépített `eval()` függvényt használjuk. Például:

```
x = eval("(2 ** 31) - 1")     # x == 2147483647
```

Ez a módszer megfelelő, ha a felhasználó által beírt kifejezések futtatásáról van szó. De mi a helyzet akkor, ha a program futása közben kell létrehoznunk egy függvényt? Ilyen esetekben használhatjuk a beépített `exec()` függvényt. Például elképzelhető, hogy a felhasználó a $4\pi r^2$ képletet írja be, azonosítóként pedig a „gömb felszíne” nevet adja meg. Ebből szeretnénk függvényt készíteni. Feltételezve, hogy a π helyére a `math.pi` állandó kerül, a függvény az alábbiak szerint hozható létre:

```
import math
code = '''
def area_of_sphere(r):
    return 4 * math.pi * r ** 2
'''
context = {}
context["math"] = math
exec(code, context)
```

Megfelelő behúzást kell alkalmaznunk, hiszen az idézőjelek között szereplő kód szabványos Python-kód. (Bár ebben az esetben írhattunk volna mindent egyetlen sorba, hiszen maga a képlet egyetlen sorból áll.)

Ha az `exec()` függvény paramétere mindössze némi kód, akkor a kód végrehajtása során létrejövő függvények, illetve változók elérésére nincs mód. Ezen felül az `exec()` függvény képtelen elérni a betöltött modulokat, illetve bármely, a hívás helyének hatókörében lévő

változót, függvényt vagy egyéb objektumot. A fent említett problémák egyaránt megoldódnak, ha az `exec()` függvénynek második paraméterként egy szótárat adunk át. A szótár biztosítja az objektumhivatkozások tárolására szolgáló helyet. Ezek az objektumhivatkozások az `exec()` függvény végrehajtása után is elérhetők maradnak. Például a `context` nevű szótár használata arra szolgál, hogy az `exec()` függvénnyel létrehozott `area_of_sphere()` függvényre mutató objektumhivatkozást tárolni tudjuk. Ebben a példában az `exec()` függvényre azért volt szükség, hogy elérhessük a `math` modult, ezért a `context` nevű szótárban olyan elemet helyeztünk el, amelynek a kulcsa a modul neve, az értéke pedig a megfelelő modulobjektumra mutató objektumhivatkozás. Így az `exec()` függvény hívása alatt is lehetővé válik a `math.pi` állandó elérése.

Néhány esetben kényelmes lehet, ha az `exec()` függvény számára minden globálisan elérhető objektumot elérhetővé teszünk. Ez úgy oldható meg, hogy a `globals()` függvénytől visszakapott szótárat adjuk át az `exec()` függvénynek. A megközelítés egyik hátránya, hogy minden olyan objektum, amely az `exec()` függvény hívása alatt jön létre, bekerül ebbe a szótárba. Megoldást az jelent, ha a globálisan elérhető objektumokat bemásoljuk egy szótárba, például a `context = globals().copy()` utasítással. Így az `exec()` függvény még mindig hozzáfér a betöltött modulokhoz, illetve a hívás helyének hatókörében lévő változókhoz és egyéb objektumokhoz; minthogy azonban másolatot hoztunk létre, az `exec()` függvény futása alatt felmerülő változások a `context` nevű szótárban maradnak, és nem válnak a globális környezet részévé. (Arra gondolhatnánk, hogy biztonságosabb, ha inkább a `copy.deepcopy()` függvényt használjuk, de ha biztonsági megfontolások kerülnek szóba, akkor az a legokosabb, ha teljes egészében tartózkodunk az `exec()` függvény használatától.) A helyi hatókörben lévő változók és objektumok is átadhatók az `exec()` függvénynek – például úgy, hogy harmadik paraméterként a `locals()` függvényt adjuk meg.

Az `exec()` függvény hívása után a `context` szótár tartalmaz egy „`area_of_sphere`” nevű kulcsot, amelynek az értéke az `area_of_sphere()` függvény. Íme a függvény elérése és hívása:

```
area_of_sphere = context["area_of_sphere"]
area = area_of_sphere(5)    # area == 314.15926535897933
```

Az `area_of_sphere` objektum a dinamikusan létrehozott függvényre mutató objektumhivatkozás, és pontosan ugyanúgy használható, mint az összes többi függvény. És ámbár ezúttal az `exec()` függvény hívása során csak egyetlen függvényt hoztunk létre, eltérően az `eval()` függvénytől, amely csak egyetlen kifejezés kezelésére képes, az `exec()` függvénnyel annyi Python-nyelvű kifejezést kezelhetünk, amennyi csak jól esik. Ezalatt érthetünk egész modulokat is, ahogy az a következő részből kiderül.

Modulok dinamikus betöltése

A Pythonban három egyszerűen használható módszer áll a rendelkezésünkre bővítmények létrehozására. Mindhárom módszer használata során futásidőben, név szerint kell modulokat betöltenünk. A modulok betöltésének végeztével a Python vizsgálófüggvényeit használhatjuk arra, hogy megbizonyosodjunk a használni kívánt függvények létéről és elérhetőségéről.

A következőkben a `magic-numbers.py` programot fogjuk áttekinteni. A program a parancssorban megadott fájlokból beolvassa az első ezer bájtot, és megjeleníti az egyes fájlok típusát – illetve az „Unknown" (ismeretlen) szöveget – és a fájl nevét. Íme egy lehetséges parancssor, illetve a program futtatása során keletkezett kimenet részlete:

```
C:\Python30\python.exe magic-numbers.py c:\windows\*.*
...
XML.................c:\windows\WindowsShell.Manifest
Unknown.............c:\windows\WindowsUpdate.log
Windows Executable..c:\windows\winhelp.exe
Windows Executable..c:\windows\winhlp32.exe
Windows BMP Image...c:\windows\winnt.bmp
...
```

A program minden olyan modul betöltésével megpróbálkozik, amely vele azonos könyvtárban található, és a neve tartalmazza a „magic" karaktersorozatot. Az ilyen moduloktól azt várjuk, hogy egyetlen nyilvános függvényt tartalmaznak, amelynek a neve: `get_file_type()`. A könyv példái között szerepel két nagyon egyszerű példamodul, a `StandardMagicNumbers.py` és a `WindowsMagicNumbers.py`. Mindkettő rendelkezik `get_file_type()` nevű függvénnyel. A program `main()` függvényét két részletben tekintjük át.

```
def main():
    modules = load_modules()
    get_file_type_functions = []
    for module in modules:
    get_file_type = get_function(module, "get_file_type")
    if get_file_type is not None:
        get_file_type_functions.append(get_file_type)
```

Hamarosan megismerkedünk a `load_modules()` függvény három különböző megvalósításával. A függvény a modulobjektumok (esetleg üres) listáját adja vissza. Megismerkedünk a `get_function()` függvénnyel is. Minden megtalált modulnál megpróbálkozunk a `get_file_type()` függvény létének ellenőrzésével, és ha találunk ilyet, akkor elhelyezzük a függvényeket tároló listánkban.

```
for file in get_files(sys.argv[1:]):
    fh = None
    try:
        fh = open(file, "rb")
        magic = fh.read(1000)
```

```
        for get_file_type in get_file_type_functions:
            filetype = get_file_type(magic,
                                     os.path.splitext(file)[1])
            if filetype is not None:
                print("{0:.<20}{1}".format(filetype, file))
                break
        else:
            print("{0:.<20}{1}".format("Unknown", file))
    except EnvironmentError as err:
        print(err)
    finally:
        if fh is not None:
            fh.close()
```

A fenti ciklus bejárja a parancssorban megadott fájlokat, és beolvassa az egyes fájlok első ezer bájtját. Ennek végeztével kipróbál minden egyes `get_file_type()` függvényt, és megpróbálja megállapítani az aktuális fájl típusát. Ha a fájltípus megállapítása sikeres, a fájl adatait a képernyőre írjuk, és kilépünk a belső ciklusból, a következő fájlnál folytatva a feldolgozást. Ha a fájl típusának megállapítására egyetlen függvény sem volt képes – illetve, ha nem találtunk egyetlen `get_file_type()` függvényt sem –, az „Unknown" szót írjuk a képernyőre.

A modulok dinamikus betöltésének az alábbiakban három különböző, ámde egyenértékű módszerét tekintjük át. Először a leghosszabb és legbonyolultabb megközelítési módot tekintjük, mivel ez minden egyes lépést bemutat:

```
def load_modules():
    modules = []
    for name in os.listdir(os.path.dirname(__file__) or "."):
        if name.endswith(".py") and "magic" in name.lower():
            filename = name
            name = os.path.splitext(name)[0]
            if name.isidentifier() and name not in sys.modules:
                fh = None
                try:
                    fh = open(filename, "r", encoding="utf8")
                    code = fh.read()
                    module = type(sys)(name)
                    sys.modules[name] = module
                    exec(code, module.__dict__)
                    modules.append(module)
                except (EnvironmentError, SyntaxError) as err:
                    sys.modules.pop(name, None)
                    print(err)
                finally:
                    if fh is not None:
                        fh.close()
    return modules
```

Első lépésként a programban lévő fájlokat járjuk be. Ha az aktuális könyvtárról van szó, az `os.path.dirname(__file__)` függvény üres karakterláncot ad vissza, aminek folyományaként az `os.listdir()` függvény kivételt vált ki. Szükség esetén inkább átadunk egy pontot (`"."`). Minden alkalmasnak látszó fájl esetében (`.py` kiterjesztése van, és a nevében szerepel a „magic" szó) úgy kapjuk meg a modul nevét, hogy a fájl kiterjesztésénél csonkolunk. Ha az így létrejött név valós azonosító – tehát használható modulnévként –, és nincs még a `sys.modules` szótárban tárolt globális modulnévlistában, akkor megpróbálkozunk a betöltésével.

A fájl szövegét a `code` karakterláncba olvassuk be. A következő sor, a `module = type(sys)(name)`, meglehetősen érdekes. Amikor a `type()` függvényt meghívjuk, a neki átadott objektum típusának megfelelő objektumot kapunk vissza. Tehát például a `type(1)` hívás eredménye egy `int` objektum lenne. Ha a típust ábrázoló objektumot le kell írni, akkor valamilyen ember által olvasható kimenetet kapunk (például „int"). Ha azonban a típusobjektumot függvényként hívjuk meg, akkor az adott típusba tartozó objektum lesz a visszatérési érték. Például ha azt írjuk, hogy `x = 5`, akkor az `x` változóban az `5` egész számot kapjuk eredményül. Ugyanez az eredmény, ha az alábbi utasítások egyikét adjuk ki: `x = int(5); x = type(0)(5); int_type = type(0); x = int_type(5)`. Esetünkben a kiadott utasítás: `type(sys)`, a `sys` pedig egy modul. Így aztán modul típusú objektumot kapunk vissza (valójában olyan, mintha osztály típusú objektumot kapnánk), amit arra használhatunk, hogy az adott név alapján új modult hozzunk létre. Az előző, `int` objektumról szóló példához hasonlóan, ahol nem számított, hogy melyik egész számot használtuk az `int` típusú objektum előállítására, most sem számít, hogy melyik modult használjuk (már amennyiben létező modult használunk, vagyis olyat, amelyet sikeresen betöltöttünk) a modul típusú objektum előállításához.

Ha az új – egyelőre üres – modul létrejött, elhelyezzük a modulokat tartalmazó globális listában, mégpedig azért, hogy nehogy véletlenül újból betöltsük. Ez még az `exec()` függvény hívása előtt történik meg, azért, hogy minél inkább megközelítsük az `import` utasítás működését. Ha készen vagyunk, az `exec()` függvényt meghívva végrehajtjuk a beolvasott kódot – és a modul könyvtárát használjuk a kód hatókörébe tartozó objektumok láthatóvá tételére. Ha ezzel is elkészültünk, elhelyezzük a modult abban a listában, amelyet visszaadunk. Ha viszont problémába ütközünk, töröljük a modult a modulokat tartalmazó globális szótárból – már amennyiben belekerült: ha ugyanis hiba történik művelet közben, akkor a modul nem kerül bele a listába. Tartsuk észben, hogy az `exec()` függvény bármekkora mennyiségű kód kezelésére képes (ugyanakkor az `eval()` függvény egyetlen kifejezés kiértékelését tudja elvégezni – lásd a 8.1. táblázatot), ha pedig nyelvtani hibába ütközne, `SyntaxError` kivételt vált ki.

8.1. táblázat *A dinamikus programozást és az objektumok vizsgálatát segítő függvények*

Utasításforma	Leírás
`__import__(...)`	Modult tölt be név alapján; lásd a szöveget.
`compile(`*forrás, fájl, mód*`)`	A *forrás* szöveg fordításából származó kódobjektummal tér vissza, a *fájl* a fájl neve vagy `"<string>"`; a *mód* lehetséges értékei: „single”, „eval” vagy „exec”.
`delattr(`*obj, név*`)`	Az *obj* objektumból törli a *név* jellemzőt.
`dir(`*obj*`)`	Az aktuális hatókör – illetve, ha az *obj* szerepel, akkor az *obj* objektum – jellemzőinek és tagfüggvényeinek neveiből álló listát ad vissza.
`eval(`*forrás, globális_változók, helyi_változók*`)`	A *forrás*-ban lévő egyetlen kifejezés kiértékeléséből származó eredménnyel tér vissza. Ha megadtuk, akkor a *globális_változók*-ban a globális, a *helyi_változók*-ban a helyi környezet található – szótárak formájában.
`exec(`*forrás, globális_változók, helyi_változók*`)`	A *forrás* (ami egy karakterlánc vagy a *compile()* függvény eredményeként előállt kódobjektum lehet) kiértékelése után `None` eredménnyel tér vissza. Ha megadtuk, akkor a *globális_változók*-ban a globális, a *helyi_változók*-ban a helyi környezet található.
`getattr(`*obj, név, érték*`)`	Az *obj* objektum *név* jellemzőjének értékével tér vissza, illetve az *érték*-kel, ha azt megadtuk, és a megadott jellemző nem létezik.
`globals()`	Az aktuális globális környezetből képzett szótárat adja vissza.
`hasattr(`*obj, név*`)`	`True` értékkel tér vissza, ha az *obj* objektum rendelkezik *név* nevű jellemzővel.
`locals()`	Az aktuális helyi környezetből képzett szótárat adja vissza.
`setattr(`*obj, név, érték*`)`	Az *obj* objektum *név* jellemzőjének értékét *érték*-re állítja; a jellemzőt szükség esetén létrehozza.
`type(`*obj*`)`	Az *obj* objektum típusobjektumával tér vissza.
`vars(`*obj*`)`	Az *obj* objektum környezetét adja vissza szótár formájában; ha az *obj* paramétert elhagyjuk, akkor a helyi környezetet.

Íme a második módszer, amellyel egy modul dinamikusan, futásidő alatt tölthető be – az alábbi kódrészlet az első megközelítés `try...except` kódblokkját helyettesíti:

```
try:
    exec("import " + name)
    modules.append(sys.modules[name])
except SyntaxError as err:
    print(err)
```

A fenti megközelítéssel az az elméleti probléma adódik, hogy biztonsági kockázatokat rejt magában. A `name` változó elején szerepelhet a `sys` szó, amelyet akár rossz szándékú kód is követhet.

És itt következik a harmadik megközelítés is, amely megint csak az első módszerhez tartozó kódrészletben lévő `try...except` kódblokkot cseréli le:

```
try:
    module = __import__(name)
    modules.append(module)
except (ImportError, SyntaxError) as err:
    print(err)
```

A fenti megközelítés kínálja a legegyszerűbb módot modulok dinamikus betöltésére. A módszer lényegesen biztonságosabb az `exec()` függvény használatánál, bár egyáltalán nem tekinthető biztonságosnak, mint ahogy a dinamikus betöltési módszerek egyike sem: soha nem lehetünk biztosak abban, hogy egy modul betöltésekor miféle kódot futtatunk.

Az előzőekben bemutatott módszerek egyike sem képes olyan csomagok, illetve modulok kezelésére, amelyek nem az aktuális könyvtárban vannak; és nem jelent nehézséget úgy megváltoztatni a kódot, hogy az ilyen helyzetek kezelésére is alkalmassá váljon. Mindazonáltal, amennyiben kifinomultabb működés a cél, érdemes elolvasni az internetes dokumentációt, különösen az `__import__()` függvény esetében.

Most, hogy a modul betöltésével végeztünk, használatba kell vennünk a benne megvalósított képességeket. Ez úgy oldható meg, hogy kiaknázzuk a Python nyelv beépített vizsgálati képességeit, azaz a `getattr()` és a `hasattr()` függvényeket. Az alábbi kódrészlet bemutatja, hogy miként használtuk ezeket a függvényeket a `get_function()` függvény megvalósításakor.

```
def get_function(module, function_name):
    function = get_function.cache.get((module, function_name), None)
    if function is None:
        try:
            function = getattr(module, function_name)
            if not hasattr(function, "__call__"):
                raise AttributeError()
            get_function.cache[module, function_name] = function
        except AttributeError:
            function = None
    return function
get_function.cache = {}
```

A gyorsítótárral kapcsolatos kódrészletet egy pillanatra hagyjuk figyelmen kívül. A függvény annyit tesz, hogy a modulobjektumra meghívja a `getattr()` függvényt, mégpedig az általunk kívánt névvel. Ha a megjelölt jellemző nem létezik, `AttributeError` kivétel lép fel. Ha azonban megtaláljuk a keresett jellemzőt, akkor a `hasattr()` függvénnyel ellenőrizzük, hogy a jellemző maga rendelkezik-e a `__call__`

`collections.Callable` osztály
☞ 399

nevű jellemzővel. Ez a jellemző olyasmi, amellyel minden hívható objektum (függvény és tagfüggvény) rendelkezik. (A továbbiakban ismertetni fogunk egy elegánsabb módszert annak eldöntésére, hogy egy bizonyos jellemző meghívható-e vagy sem.) Amennyiben a jellemző létezik és hívható, visszaadhatjuk a hívónak, egyéb esetekben pedig `None` objektumot adunk vissza, ezzel jelezve, hogy a függvény nem megfelelő.

Ha fájlok százainak feldolgozását kell elvégezni (például azért, mert a `C:\windows` könyvtárban a `*.*` paraméterrel futtattuk a programot), nem szeretnénk elvégezni a keresést minden fájl és minden modul esetében. Így aztán mihelyt megadjuk a `get_function()` függvényt, el is helyezünk benne egy jellemzőt, méghozzá egy `cache` nevű szótárat. (A Pythonban szinte mindig lehetőségünk nyílik a saját készítésű objektumokat nekünk tetsző nevű jellemzőkkel felruházni.) Amikor a `get_function()` függvényt először hívjuk meg, a `cache` szótár még üres, így a `dict.get()` függvény hívása `None` objektumot ad vissza. Minden olyan alkalommal azonban, amikor megfelelő függvényre találtunk, a függvény bekerül a szótárba, mégpedig kételemű rendezett sorozatként: a modul- és a függvénynév lesz a kulcs, a függvény maga pedig az érték. Így aztán a második esettől kezdve minden olyan alkalommal, amikor egy adott függvényre van szükség, a függvényt a gyorsítótárból olvassuk ki, és semmiféle keresés nem történik a jellemzőkön.*

Azt a módszert, amelyet a `get_function()` függvény egy adott paraméterhalmazra vonatkozó visszatérési értékeinek gyorsítótárazására használtunk, *jegyzetelésnek* (memoizing) nevezzük. Minden olyan függvény esetében használható, amelynek nincsenek mellékhatásai (nem változtat a globális változókon), és amelyek ugyanannak a (nem módosítható) paraméternek az átadásakor mindig ugyanazt az eredményt adják vissza. Mivel a jegyzetelt függvények gyorsítótárainak létrehozásához és kezeléséhez szükséges kód ugyanaz, ideális jelöltnek mutatkozik arra, hogy függvénydíszítőt alakítsunk ki belőle. A Python Cookbook webhelyén (`code.activestate.com/recipes/langs/python/`) sok, ezzel kapcsolatos díszítőreceptet találunk. Minthogy azonban a modulobjektumok módosíthatók, bonyolultabb a dolgunk annál, semhogy egyszerűen leakasztunk a szögről magunknak valamiféle jegyzetelő díszítőt. Az ilyen függvények a mi `get_function()` függvényünkkel nem fognak azonnal működni. Egyszerű megoldásnak kínálkozik, hogy a modul helyett csak a modul `__name__` karakterláncát használjuk a kulcs rendezett sorozatának első elemeként.

Figyelem! A dinamikus modulbetöltés egyszerű, csakúgy, mint az `exec()` függvénnyel történő, tetszőleges Python-kód futtatása. Mindez néhány esetben meglehetősen kényelmes, például lehetőségünk van adatbázisban tárolni kódot. Ne feledjük azonban, hogy nem tudjuk ellenőrizni, hogy a betöltött vagy az `exec()` függvény segítségével végrehajtott kód mit csinál. Jusson eszünkbe, hogy a modulok nem csak változókat, függvényeket és osztályokat tartalmazhatnak, hanem olyan kódot is, amely

* A `magic-numbers.py` programban a fent bemutatott változaton kívül megtaláljuk a `get_function()` függvény egy lényegesen kifinomultabb változatát is, amely ügyesebben kezeli az olyan modulokat, amelyekben nincs meg a szükséges függvény.

lefut a modul betöltésének során. Ha a kód ellenőrizetlen forrásból érkezik, esetleg kellemetlenségeink adódhatnak. Az, hogy ezzel a problémával miként küzdünk meg, az adott helyzettől függ. Természetesen mindez nem feltétlenül okoz gondot – így a személyes projektek esetében sem.

Helyi függvények és önhívó függvények

Gyakran lehet hasznos, ha egy függvény belsejében megírunk egy-két segédfüggvényt. A Pythonban az ilyen esetekben nem kell semmilyen, az eddigiektől eltérő utasításformát használnunk – egyszerűen annyit teszünk, hogy a már létező függvény belsejében megadjuk a szükséges függvényeket. Az ilyen függvényeket gyakran nevezzük *beágyazott függvényeknek* vagy *helyi függvényeknek*. Helyi függvényekkel a 7. fejezetben már találkoztunk.

A helyi függvények igénybe vételének egyik legelterjedtebb esete az önhívás (rekurzió) használata. Ilyenkor egy külső függvényt hívunk meg, amely beállít egy-két dolgot, majd első alkalommal meghív egy helyi önhívó függvényt. Az önhívó függvények (vagy tagfüggvények) olyan függvények, amelyek saját magukat hívják meg. Szerkezetét tekintve minden önhívó függvény két esetet különböztet meg: az *alapesetet* és az *önhívó esetet*. Az alapeset szerepe az önhívás megállítása.

Az önhívó függvények számításigénye általában magas, mivel minden önhívás során új memóriavermet foglalunk, de van néhány olyan algoritmus, amelyeknek a legtermészetesebb megvalósítási formája önhívást kíván. A legtöbb Python-megvalósításban meghatározott az önhívások maximális száma. A maximális szám kiolvasása a `sys.getrecursionlimit()` függvénnyel, beállítása pedig a `sys.setrecursionlimit()` függvénnyel történik. Megjegyeznénk azonban, hogy a maximális érték növelése általában annak a jele, hogy a használt algoritmus nem megfelelő, vagy a megvalósítás hibás.

Az önhívó függvények klasszikus példája faktoriálisok* kiszámítását végzi. Például a `factorial(5)` az 5! kifejezés kiszámítását végzi, visszatérési értéke pedig 120, azaz 1 x 2 x 3 x 4 x 5:

```
def factorial(x):
    if x <= 1:
        return 1
    return x * factorial(x - 1)
```

A megoldás nem hatékony, de bemutatja az önhívó függvények két alapvető jellemzőjét. Ha a megadott szám (`x`) értéke 1 vagy annál kevesebb, a visszaadott érték is 1, önhívás pedig nem történik. Ez az alapeset. Ha viszont az `x` nagyobb, mint 1, akkor a visszaadott érték: `x * factorial(x - 1)`. Ez az önhívó eset, hiszen a `factorial` függvény itt saját magát hívja meg. A függvény befejeződése biztosított, hiszen ha az `x` kezdőértéke nem

* A Python `math` moduljában megtalálható a sokkal hatékonyabb `math.factorial()` függvény.

nagyobb 1-nél, akkor az alapesetet használjuk, és a függvény azonnal véget ér; ha pedig az x nagyobb 1-nél, akkor minden önhívás paramétere az előzőnél eggyel kisebb szám, vagyis előbb-utóbb ismét elérjük az alapesetet.

Ha a helyi, illetve az önhívó függvényeket valós környezetben akarjuk tanulmányozni, megfigyelhetjük az `IndentedList.py` modulban lévő `indented_list_sort()` függvényt. A függvény egyik paramétere egy karakterláncokból álló lista, mégpedig olyan, amelyben behúzásokkal jelezzük a hierarchiát. A másik paraméter egy első szintű behúzást tartalmazó karakterlánc. A kimenet szintén lista, amelyben ugyanazok a karakterláncok találhatók meg, de a kis- és nagybetűket figyelmen kívül hagyó ábécésorrendbe rendezve, és a behúzást tartalmazó elemeket a szülőelem alatt szerepeltetve. A 8.1. ábra `before` és `after` listái ezt szemléltetik.

Az `after` lista a `before` lista alapján az alábbi függvényhívással áll elő: `after = IndentedList.indented_list_sort(before)`. Az alapértelmezett behúzás négy szóköz, ugyanaz, amit a `before` listában is használtunk – így ezt nem kellett külön beállítanunk.

```
before = ["Nonmetals",
          "    Hydrogen",
          "    Carbon",
          "    Nitrogen",
          "    Oxygen",
          "Inner Transitionals",
          "    Lanthanides",
          "        Cerium",
          "        Europium",
          "    Actinides",
          "        Uranium",
          "        Curium",
          "        Plutonium",
          "Alkali Metals",
          "    Lithium",
          "    Sodium",
          "    Potassium"]
```

```
after = ["Alkali Metals",
         "    Lithium",
         "    Potassium",
         "    Sodium",
         "Inner Transitionals",
         "    Actinides",
         "        Curium",
         "        Plutonium",
         "        Uranium",
         "    Lanthanides",
         "        Cerium",
         "        Europium",
         "Nonmetals",
         "    Carbon",
         "    Hydrogen",
         "    Nitrogen",
         "    Oxygen"]
```

8.1. ábra

Behúzásokat tartalmazó lista – rendezés előtt és után

Az `indented_list_sort()` függvényt először egészében, majd két helyi függvényként tekintjük át.

```
def indented_list_sort(indented_list, indent="  "):
    KEY, ITEM, CHILDREN = range(3)
    def add_entry(level, key, item, children):
        ...
    def update_indented_list(entry):
        ...
```

```
    entries = []
    for item in indented_list:
        level = 0
        i = 0
        while item.startswith(indent, i):
            i += len(indent)
            level += 1
        key = item.strip().lower()
        add_entry(level, key, item, entries)
    indented_list = []
    for entry in sorted(entries):
        update_indented_list(entry)
    return indented_list
```

A kód elején létrehozunk három állandót, amelyeket később arra használunk, hogy a helyi függvényekben használt indexhelyeknek nevet adjunk. Ezt követően megadjuk azt a két helyi függvényt, amelyet rövidesen áttekintünk. A rendező algoritmus működése két lépésre osztható. Az első lépésben létrehozzuk a bejegyzések listáját. Minden bejegyzés egy-egy háromelemű rendezett sorozatot jelent. Az első elem a „kulcs", amelyet a rendezéskor fogunk használni; a második az eredeti karakterlánc; a harmadik pedig egy lista, amely a karakterlánc gyermekeit tartalmazza. A kulcs nem egyéb, mint a karakterláncból létrehozott kisbetűs változat, amelynek mindkét végéről eltávolítottuk az üreshelyeket. A szint a behúzás szintjét jelöli – a felső szintű elemek esetében 0, ezek gyermekelemeinél 1, és így tovább. A második lépésben létrehozunk egy új, behúzásokat tartalmazó listát, amelyben elhelyezzük az előző listában rendezetten lévő elemeket, az egyes karakterláncok alatti szinten szereplő karakterláncokat, és így tovább. Eredményül egy behúzásokat tartalmazó, rendezett listát kapunk.

```
def add_entry(level, key, item, children):
    if level == 0:
        children.append((key, item, []))
    else:
        add_entry(level - 1, key, item, children[-1][CHILDREN])
```

A fenti függvényt a listában lévő minden egyes karakterlánc esetében meghívjuk. A függvény `children` paramétere az a lista, amelybe az új bejegyzéseket helyezni kell. Amikor a függvényt a külső – `indented_list_sort()` – függvényből hívjuk meg, a paraméter az `entries` lista. A művelet eredménye az lesz, hogy a karakterláncokból álló lista bejegyzésekből álló listává alakul; minden bejegyzés egy felső szintű (behúzás nélküli) karakterláncból, illetve egy a gyermekelemeket tartalmazó (esetleg üres) listából áll.

Ha a 0. szinten vagyunk (vagyis a felső szinten), új, háromelemű rendezett sorozatot helyezünk el az `entries` listában. Ebben tároljuk a kulcsot (a rendezéshez kell), az eredeti elemet (amely majd a művelet eredményeként létrejövő listába kerül), illetve a gyermekelemek üres listáját. Ez az alapeset, hiszen ilyenkor nem történik önhívás. Ha a szint 0-nál

magasabb, akkor az elem a `children` lista utolsó elemének gyermekeleme (ha tetszik, leszármazottja). Ebben az esetben az `add_entry()` függvényt ismét önhívással hívjuk meg, a szintet eggyel csökkentjük, és a gyermekeket tartalmazó lista utolsó elemének gyermeklistáját adjuk át bővítendő listaként. Ha a szint legalább 2, akkor több önhívás is történik, egészen addig, amíg a szint 0 nem lesz, és a bejegyzés közvetlenül a `children` listába kerül.

Például amikor az „Inner Transitionals" karakterlánchoz érünk, a külső függvény 0 szinttel hívja az `add_entry()` függvényt, a kulcs pedig az „inner transitionals" lesz, az elem maga az „Inner Transitionals", a gyermekelemeket tartalmazó lista pedig az `entries`. Mivel a 0. szinten vagyunk, új, egy kulcsot, magát az elemet és egy üres gyermekelemlistát magában foglaló elem kerül a gyermekelemeket tartalmazó lista (az `entries`) végére. A következő karakterlánc a „ Lanthanides". Van már benne behúzás, vagyis az „Inner Transitionals" karakterlánc gyermekének fogjuk tekinteni. Az `add_entry()` függvényhívás ezúttal első szintet kap, a kulcs a „lanthanides", az elem maga a „ Lanthanides", a gyermekelemeket tartalmazó lista pedig az `entries`. Mivel az első szinten vagyunk, az `add_entry()` függvény meghívja saját magát, ezúttal már a 0. szinttel (1-1=0), ugyanazzal a kulccsal és elemmel, de a gyermekelemeket tartalmazó lista az előző elem gyermekelemeket tartalmazó listája lesz, vagyis az „Inner Transitionals" elemhez tartozó gyermekelemlista. Íme az `entries` lista a karakterláncok elhelyezése után, de még a rendezés előtt:

```
[('nonmetals',
  'Nonmetals',
  [('hydrogen', '   Hydrogen', []),
   ('carbon', '   Carbon', []),
   ('nitrogen', '   Nitrogen', []),
   ('oxygen', '   Oxygen', [])]),
 ('inner transitionals',
  'Inner Transitionals',
  [('lanthanides',
    '    Lanthanides',
    [('cerium', '        Cerium', []),
     ('europium', '        Europium', [])]),
   ('actinides',
    ' Actinides',
    [('uranium', '        Uranium', []),
     ('curium', '        Curium', []),
     ('plutonium', '        Plutonium', [])])]),
 ('alkali metals',
  'Alkali Metals',
  [('lithium', '        Lithium', []),
   ('sodium', '        Sodium', []),
   ('potassium', '        Potassium', [])])]
```

A kimenet előállítására a `pprint` („pretty print") nevű modul `pprint.pprint()` függvényét használjuk. Figyeljük meg, hogy az `entries` listának csak három eleme van (mindhárom elem egy-egy háromelemű rendezett sorozat), és hogy minden háromelemű `tuple` utolsó eleme egy újabb háromelemű `tuple` (esetleg egy üres lista).

Az `add_entry()` függvény egyszerre helyi és önhívó függvény. Mint minden önhívó függvény, ez a függvény is rendelkezik egy az önhívásnak véget vetni képes *alapesettel* (esetünkben arról az esetről van szó, amikor a szint 0), illetve egy önhívó esettel.

A függvényt megírhattuk volna némileg eltérő formában is:

```
def add_entry(key, item, children):
    nonlocal level
    if level == 0:
        children.append((key, item, []))
    else:
        level -= 1
        add_entry(key, item, children[-1][CHILDREN])
```

A fenti esetben a `level` (szint) átadása nem paraméterként történik, hanem a `nonlocal` kulcsszóval jelezzük, hogy a változót egy eggyel magasabb szinten lévő hatókörből is el szeretnék érni. Ha a függvény belsejében nem szeretnénk megváltoztatni a `level` változó értékét, akkor nincs szükség a `nonlocal` kulcsszóra – ilyen helyzetekben a Python ugyanis nem találja meg a változót a helyi (belső függvényre érvényes) hatókörben, és az eggyel kintebbi hatókörben keresi tovább, ahol meg is találja. Az `add_entry()` függvény fenti változatában azonban a `level` változó értékét meg kell változtatnunk. A `global` kulcsszóval jelezzük a Pythonnak, hogy egy globális változó értékén szeretnénk változtatni – így oldható meg, hogy a globális változó értékének frissítése helyett ne új, helyi változó jöjjön létre. Ugyanez vonatkozik minden olyan változóra, amelynek az értékét úgy akarjuk megváltoztatni, hogy maga a változó az aktuális hatókörön kívül helyezkedik el. Ámbár gyakran az a legokosabb, ha a `global` kulcsszó használatát egy az egyben elkerüljük, és szintén alaposan oda kell figyelnünk a `nonlocal` kulcsszó használatára.

```
def update_indented_list(entry):
    indented_list.append(entry[ITEM])
    for subentry in sorted(entry[CHILDREN]):
        update_indented_list(subentry)
```

Az algoritmus első lépésében feltöltjük a bejegyzések háromelemű sorozatokból (kulcs, maga az elem, a gyermekelemek listája) álló listáját – a sorrend nem változik az eredeti listához képest. Az algoritmus második lépésében új, üres, behúzásokat tartalmazó listából indulunk ki, bejárjuk a rendezett bejegyzéseket, és a behúzásokat tartalmazó listát úgy alakítjuk ki, hogy a bejáráskor minden elemnél az `update_indented_list()` függvényt hívjuk meg. Az `update_indented_list()` függvény önhívó. Minden felső szintű bejegyzésnél elhelyez egy elemet az `indented_list` nevű listában, majd a listában lévő egyes elemek gyermekelemeinél saját magát hívja, és így tovább. Az alapeset (amikor az önhívás leáll) akkor következik be, amikor egy elem vagy egy gyermekelem, vagy egy gyermekelem gyermekeleme (és így tovább) nem rendelkezik további saját gyermekelemekkel.

A Python az `indented_list` nevű objektumot a helyi (a belső függvényhez kapcsolódó) hatókörben keresi, de nem találja meg. Így aztán eggyel magasabb szintű hatókörben keresi tovább, ahol aztán meg is találja. Figyeljük meg azonban, hogy elemeket helyezünk el a függvény belsejében – az `indented_list` listában –, holott nem használtuk a `nonlocal` kulcsszót. Ez azért lehetséges, mert a `nonlocal` (valamint a `global`) kulcsszó objektumhivatkozásokkal dolgozik, nem pedig az objektumhivatkozások által hivatkozott objektumokkal. Az `add_entry()` függvény második változatában azért használtuk a `level` változóval kapcsolatban a `nonlocal` kulcsszót, mert ha a `+=` művelet tényezője egy szám, akkor a művelet az objektumhivatkozás célpontjául egy új objektumot ad meg – valójában a `level = level + 1` műveletet hajtja végre a Python – így a `level` változó mostantól egy új, egész számot tartalmazó objektumra hivatkozik. Amikor azonban a `list.append()` függvényt az `indented_list` nevű listával használjuk, a függvény magát a listát módosítja, vagyis nincs szó arról, hogy az objektumhivatkozás új célpontot kapna. Így a `nonlocal` kulcsszó használata szükségtelen. (Olyan esetekben, amikor egy szótárral, listával vagy egyéb más gyűjteményes adattípusú szerkezettel dolgozunk, az elemek elhelyezése a szerkezetben – vagy a kivételük belőle – ugyanilyen okból kifolyólag a `global` kifejezés használata nélkül is lehetséges.)

Függvény- és tagfüggvénydíszítők

A díszítők olyan függvények, amelyeknek egyetlen paramétere egy másik függvény, illetve tagfüggvény, a visszatérési értékük pedig egy új, az átadott függvényt tartalmazó, de azt néhány további képességgel kiegészítő függvény, illetve tagfüggvény. Használtunk már néhány előzetesen megadott díszítőt, például a `@property` és a `@classmethod` nevűeket. Ebben a részben megtanuljuk, hogy miként hozhatunk létre saját függvénydíszítőket, a fejezet későbbi részében pedig az osztálydíszítők létrehozásával is megismerkedünk.

Osztályszintű díszítők ☞ 384

A díszítőkkel kapcsolatos első példánkban tételezzük fel, hogy sok olyan függvényünk van, ami valamilyen számítást végez, és ezek között van néhány olyan, amelynek mindig pozitív eredményt kell adnia. Megtehetnénk, hogy külön-külön végzünk bennük ellenőrzést, de a díszítő használata egyszerűbb és tisztább megoldás. Íme egy függvény, amelyet a `@positive_result` díszítővel díszítettünk. A díszítőt mindjárt létrehozzuk:

```
@positive_result
def discriminant(a, b, c):
    return (b ** 2) - (4 * a * c)
```

A díszítőnek köszönhetően minden olyan esetben, amikor az eredmény nullánál kisebb, `AssertionError` kivétel lép fel, és a program futása megszakad. A díszítőt természetesen annyi függvényen használjuk, amennyin csak jólesik. Íme a díszítő megvalósítása:

```
def positive_result(function):
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
```

```
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result
    wrapper.__name__ = function.__name__
    wrapper.__doc__ = function.__doc__
    return wrapper
```

A díszítő olyan helyi függvényt ad meg, amely az eredeti függvényt hívja meg. A fenti példában a helyi függvény neve `wrapper()`, vagyis burkoló. Ez az eredeti függvényt hívja meg, tárolja az eredményt, majd figyel, hogy az eredmény mindig pozitív legyen – ha nem lesz az, a program futása megszakad. A burkolófüggvény a benne lévő függvény által kiszámított értéket adja vissza. A burkolófüggvény létrehozását követően a nevét és a dokumentációs karakterláncát az eredeti függvénytől kölcsönözzük. Ez a hibakeresést segíti: azt szeretnénk, ha a hibaüzenetek az eredeti függvény nevét írnák ki, nem pedig a burkolóét. Az eredeti függvény helyett mostantól a burkolófüggvényt használjuk.

```
def positive_result(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result
    return wrapper
```

A `@positive_result` díszítőnek ez egy némiképp letisztult változata. A burkolót magát is beburkoljuk, mégpedig a `functools` modul `@functools.wraps` díszítőjével. Ez biztosítja, hogy a `wrapper()` függvény megkapja az eredeti függvény nevét és dokumentációs karakterláncát.

Vannak esetek, amikor hasznos lehet, ha a díszítőt paraméterezni tudjuk. Első látásra ez nem tűnik lehetségesnek, hiszen a díszítő csak egyetlen paramétert vár, ami vagy egy függvény, vagy egy tagfüggvény. De a problémának van egy elegáns megoldása. Meghívhatunk egy olyan függvényt, amelynek átadjuk a kívánt paramétereket, és amely egy olyan függvényt ad vissza, amely képes az őt követő függvény díszítésére. Például:

```
@bounded(0, 100)
def percent(amount, total):
    return (amount / total) * 100
```

A fenti esetben a `bounded()` függvénynek két paramétert adunk át, és olyan díszítőt kapunk vissza, amely aztán a `percent()` függvényt díszíti. Ebben az esetben a díszítő szerepe az, hogy garantálja, hogy a visszaadott érték mindig a 0...100 nyílt intervallumba esik. Íme a `bounded()` függvény megvalósítása:

```
def bounded(minimum, maximum):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            result = function(*args, **kwargs)
```

```
            if result < minimum:
                return minimum
            elif result > maximum:
                return maximum
            return result
        return wrapper
    return decorator
```

A függvény egy díszítőfüggvényt hoz létre, amely viszont egy burkolófüggvényt állít elő. A burkolófüggvény végrehajtja a számítást, majd visszaadja a megfelelő tartományba eső eredményt. A `decorator()` díszítőfüggvény visszatérési értéke a `wrapper()` függvény, a `bounded()` függvényé pedig a díszítőfüggvény.

Érdemes megfigyelnünk azt is, hogy minden olyan alkalommal, amikor a `bounded()` függvény belsejében létrejön egy burkolófüggvény, az éppen létrejött burkolófüggvény a `bounded()` függvénynek átadott minimum- és maximumértékeket használja.

Elérkeztünk ennek a résznek az utolsó díszítőfüggvényéhez, amely azonban az előzőeknél kissé összetettebb. Egy naplózó függvényről van szó, amely feljegyzi az általa díszített függvények nevét, paramétereit és eredményét. Például:

```
@logged
def discounted_price(price, percentage, make_integer=False):
    result = price * ((100 - percentage) / 100)
    if not (0 < result <= price):
        raise ValueError("invalid price")
    return result if not make_integer else int(round(result))
```

Ha a Pythont hibakereső módban futtatjuk (ez az alapeset), minden alkalommal, amikor meghívjuk a `discounted_price()` függvényt, az ideiglenes fájlok tárolására szolgáló helyi könyvtárban lévő `logged.log` fájl végére újabb bejegyzés kerül. Az alábbi fájlrészlet ezt szemlélteti:

```
called: discounted_price(100, 10) -> 90.0
called: discounted_price(210, 5) -> 199.5
called: discounted_price(210, 5, make_integer=True) -> 200
called: discounted_price(210, 14, True) -> 181
called: discounted_price(210, -8) <type 'ValueError'>: invalid price
```

Ha a Pythont optimalizált módban (a `-O` parancssori paramétert használva, illetve a `PYTHONOPTIMIZE` környezeti változóban a `-O` értéket tárolva) futtatjuk, nem történik naplózás. Íme a naplózást megvalósító függvény, illetve a díszítőfüggvény:

```
if __debug__:
    logger = logging.getLogger("Logger")
    logger.setLevel(logging.DEBUG)
    handler = logging.FileHandler(os.path.join(
                                  tempfile.gettempdir(), "logged.log"))
    logger.addHandler(handler)
```

```
def logged(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        log = "called: " + function.__name__ + "("
        log += ", ".join(["{0!r}".format(a) for a in args] +
                         ["{0!s}={1!r}".format(k, v)
                          for k, v in kwargs.items()])
        result = exception = None
        try:
            result = function(*args, **kwargs)
            return result
        except Exception as err:
            exception = err
        finally:
            log += (") -> " + str(result)) if exception is None
                    else ") {0}: {1}".format(type(exception),
                                             exception))
            logger.debug(log)
            if exception is not None:
                raise exception
    return wrapper
else:
    def logged(function):
        return function
```

Hibakereső módban a `__debug__` globális változó értéke `True`. Ilyenkor a `logging` modul segítségével elindítjuk a naplózást, majd létrehozzuk a `@logged` díszítőt. A naplózó modul igen sokoldalú és rugalmas – képes fájlokba, forgatott fájlokba, email-be, hálózatra, HTTP-kiszolgálóra és egyéb helyekre naplózni. Ezúttal csak az alapvető eszközeit használtuk: létrehoztunk egy naplózó objektumot, beállítottuk a naplózás szintjét – számos szint támogatott –, és megjelöltük a kimenet tárolására szolgáló fájlt.

Szótárértelmezések 122, 130 ☜

A burkolófüggvény kódjának elején a naplóba írandó karakterláncot állítjuk elő, benne a függvény nevével és paramétereivel. Ez után megpróbálkozunk a függvény meghívásával, illetve az eredmény tárolásával. Ha kivétel lépne fel, azt is tároljuk. Minden esetben sor kerül a `finally` kódblokk végrehajtására. Itt helyezzük el a visszatérési értéket (vagy a kivételt) a naplóba kerülő karakterláncban, majd a karakterláncot a naplóba írjuk. Ha nem lépett fel kivétel, az eredményt adjuk vissza, egyéb esetben pedig újra kiváltjuk a kivételt, hogy az eredeti függvény viselkedését minél pontosabban utánozzuk.

Amikor a Python optimalizált módban fut, a `__debug__` változó értéke `False`. Ilyen esetben úgy adjuk meg a `logged()` függvényt, hogy egyszerűen a neki átadott függvényt adja vissza – vagyis eltekintve attól a kis terhelésnövekedéstől, amit a függvény létrehozásakor tapasztalhatunk, a program futása egyáltalán nem jár többletterheléssel.

Jegyezzük meg, hogy az alapkönyvtár `trace` és `profile` modulja programok futtatására és elemzésére szolgál; velük különféle nyomkövetési és összefoglaló jelentések állíthatók elő. Mindkettő közvetlenül a kódba tekint bele, azaz az előzőekben használt `logged()` függvénytől eltérően sem a `trace`, sem a `profile` modul használatakor nincs szükség a forráskód módosítására.

Függvénymagyarázatok

A függvények és a tagfüggvények magyarázatokkal (annotation) is megadhatók – a magyarázatok olyan kifejezések, amelyeket a függvény bevezetésében használhatunk. Íme az általános utasításforma:

```
def függvényNév(rész1 : kif1, rész2 : kif2, ..., részN : kifN) -> vkif:
    kódblokk
```

Minden kettősponttal elválasztott *rész* `(:` *kifX*`)` egy elhagyható magyarázatot jelképez, csakúgy, mint a nyilat követő „visszaadott kifejezés" rész (`->` *vkif*). Az utolsó – esetleg egyetlen – helyzeti paraméter (már amennyiben van ilyen) `*`*paraméterek* formában is megadható, magyarázattal vagy anélkül. Hasonlóan, az utolsó – esetleg egyetlen – kulcsszóparaméter (már amennyiben van ilyen) `*`*kulcsszó-paraméterek* formában is megadható, megint csak magyarázattal vagy anélkül.

Ha vannak magyarázatok, akkor a függvény `__annotations__` szótárába kerülnek. Ha nincsenek magyarázatok, akkor ez a szótár üres. A szótár kulcsai a paraméterek nevei, az értékek pedig a megfelelő kifejezések. Az utasításforma lehetővé teszi csak egy, csak néhány, vagy akár az összes paraméter magyarázattal való ellátását, illetve azt is, hogy egyáltalán ne adjunk meg magyarázatot. A visszatérési érték is ellátható magyarázattal, vagy hagyható magyarázat nélkül. A magyarázatoknak a Pythonban nincs megkülönböztetett jelentőségük. A Python egyetlen dolgot művel a magyarázatokkal, nevezetesen elhelyezi őket az `__annotations__` szótárában; minden más rajtunk áll. Íme egy magyarázatokkal ellátott, az `Util` modulból származó függvényt bemutató példa:

```
def is_unicode_punctuation(s : str) -> bool:
    for c in s:
        if unicodedata.category(c)[0] != "P":
            return False
    return True
```

Minden Unicode-karakter egy adott kategóriába tartozik, és minden kategóriát egy kétkarakteres azonosító azonosít. Minden olyan kategória, amelynek a neve *P*-vel kezdődik, írásjelkaraktereket tartalmaz.

A fenti példában a Python adattípusait használjuk a magyarázatok kifejezéseiként. Mindazonáltal a Python számára ez nem bír semmilyen megkülönböztetett jelentéssel, ahogy ennek az alábbi hívásokból ki kell derülnie:

```
Util.is_unicode_punctuation("zebr\a")       # az eredmény: False
Util.is_unicode_punctuation(s="!@#?")       # az eredmény: True
Util.is_unicode_punctuation(("!", "@"))     # az eredmény: True
```

Az első hívás egy helyzeti paramétert használ, a második pedig egy kulcsszóparamétert; mindezt csak azért, hogy megmutassuk, hogy mindkét paraméterfajta az elvárásainknak megfelelően működik. Az utolsó hívás karakterlánc helyett egy rendezett sorozatot ad át, amit a függvény elfogad, hiszen a Python semmi egyebet nem tesz, mint feljegyzi a magyarázatokat az `__annotations__` szótárába.

Ha a magyarázatokat értelmessé szeretnénk tenni, például típusellenőrzést akarunk végezni, akkor az egyik lehetőség az, hogy az ellenőrizendő függvényeket megfelelő díszítővel látjuk el. Íme egy meglehetősen alapszintű, típusellenőrzést végző díszítő:

```
def strictly_typed(function):
    annotations = function.__annotations__
    arg_spec = inspect.getfullargspec(function)

    assert "return" in annotations, "missing type for return value"
    for arg in arg_spec.args + arg_spec.kwonlyargs:
        assert arg in annotations, ("missing type for parameter '" +
                                    arg + "'")
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        for name, arg in (list(zip(arg_spec.args, args)) +
                          list(kwargs.items())):
            assert isinstance(arg, annotations[name]), (
                   "expected argument '{0}' of {1} got {2}".format(
                   name, annotations[name], type(arg)))
        result = function(*args, **kwargs)
        assert isinstance(result, annotations["return"]), (
                   "expected return of {0} got {1}".format(
                   annotations["return"], type(result)))
        return result
    return wrapper
```

Ez a díszítő akkor használható, ha a függvény minden paramétere és a visszatérési értéke is el van látva a megfelelő típust jelölő magyarázattal. Az átadott függvény a létrehozásakor ellenőrzi, hogy a függvény paramétereit és a visszatérési értéket ellátták-e a típusokat megjelölő magyarázattal, a program futtatásakor pedig azt, hogy az aktuális paraméterek megfelelnek-e a kívánalmaknak.

Az `inspect` modul sokoldalú, az objektumoknál használható vizsgálati szolgáltatást nyújt. Ezúttal csak egy kis részét használtuk az általa visszaadott paraméterleíró objektumnak: ennek segítségével nyertük ki a helyzeti és a kulcsszóparaméterek nevét – a helyzeti paraméterek esetében a pontos sorrenddel együtt. A neveket aztán a magyarázatokat tartalmazó szótárral együtt használva megbizonyosodunk arról, hogy minden paraméter és a visszatérési érték is kapott magyarázatot.

A díszítő belsejében létrehozott burkolófüggvény először bejárja a megadott helyzeti és kulcsszóparaméterekhez tartozó név-paraméter párokat. Minthogy a `zip()` függvény visszatérési értéke egy bejárható objektum, a `dictionary.items()` függvényé pedig egy szótár, nem oldható meg az, hogy azonnal egymáshoz fűzzük őket; először mindkettőt listává kell alakítanunk. Ha bármely, futáskor kapott paraméter típusa eltér a megfelelő magyarázatban tárolttól, az ellenőrzés hibát fog jelezni; egyéb esetben pedig meghívjuk a függvényt, ellenőrizzük a visszatérési érték típusát, és amennyiben megfelelőnek találjuk, visszaadjuk az értéket a hívónak. A `strictly_typed()` függvény végén a szokásos módon visszaadjuk a burkolt függvényt. Figyeljük meg, hogy az ellenőrzés csak hibakereső módban történik meg (ami a Python alapértelmezett módja, és a `-O` parancssori paraméterrel, illetve a `PYTHONOPTIMIZE` környezeti változóval állítható be).

Ha az `is_unicode_punctuation()` függvényt a `@strictly_typed` díszítővel díszítjük, és megpróbáljuk a függvényt az előző, a díszítetlen változat futtatása során használt példákkal futtatni, megfigyelhetjük a magyarázatok okozta változást:

```
is_unicode_punctuation("zebr\a")      # az eredmény: False
is_unicode_punctuation(s="!@#?")      # az eredmény: True
is_unicode_punctuation(("!", "@"))    # AssertionError kivételt vált ki
```

Ezúttal a paraméterek típusát ellenőrizzük, így az utolsó esetben `AssertionError` kivétel lép fel, hiszen a rendezett sorozat sem a karakterlánc típusnak, sem a `str` típus valamelyik alosztályának nem feleltethető meg.

A következőben a magyarázatok egy teljesen különböző használati módját fogjuk megismerni. Íme egy rövid függvény, amely a működését tekintve megegyezik a `range()` függvénnyel, a különbség mindössze annyi, hogy a mi függvényünk mindig lebegőpontos számokat ad vissza:

```
def range_of_floats(*args) -> "author=Reginald Perrin":
    return (float(x) for x in range(*args))
```

A függvény maga nem használja a magyarázatot, de könnyen elképzelhetünk egy olyan eszközt, amely a projekthez tartozó modulok betöltését követően listát készít a függvények és a függvények szerzőinek neveiről, mindezt az `__annotations__` szótár `"return"` elemét felhasználva.

A magyarázatok a Pythonban újdonságnak számítanak, és minthogy jelentésük a Python nyelvben nincs előírva, csak a képzeletünk szabhat határt a felhasználás lehetséges módjainak. A `www.python.org/dev/peps/pep-3107` webhelyen elérhető PEP 3107 (*Function Annotations*) dokumentumban további ötletekre lelhetünk a felhasználás lehetséges módjait illetőleg; illetve találhatunk itt néhány hasznos hivatkozást is.

Magasabb szintű objektumközpontú programozás

Ebben a részben a Python nyelv objektumközpontú programozást támogató oldalát ismerjük meg mélyebben, olyan módszereket tanulmányozva, amelyek a megírandó kód mennyiségét csökkentik, illetve tovább tágítják az általunk használható programozási eszközök lehetőségeit. Először azonban lássunk egy kicsi és egyszerű újdonságot. Íme a `Point` osztályt meghatározó kódrészlet eleje; az osztály viselkedése tökéletesen megegyezik a 6. fejezetben elkészített változatokéval:

```
class Point:
__slots__ = ("x", "y")
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
```

Amikor az osztályt a `__slots__` osztályszintű jellemző használata nélkül hozzuk létre, a Python a színfalak mögött minden új példány számára létrehoz egy privát, `__dict__` nevű szótárat; és ez a szótár tárolja a példány adatjellemzőit. Így válik lehetségessé jellemzők elhelyezése és eltávolítása az egyes objektumokból. (Például a `get_function()` függvényben a fejezet korábbi részében így helyeztünk el egy `cache` nevű jellemzőt.)

Ha csak olyan objektumokra van szükségünk, amelyeknek az eredeti jellemzőit kell elérnünk, és nem merül fel igény jellemzők elhelyezésére és eltávolítására, létrehozhatunk olyan osztályokat is, amelyek nem rendelkeznek `__dict__` szótárral. Ez úgy valósítható meg, hogy megadunk egy `__slots__` nevű osztályszintű jellemzőt, amelynek az értéke egy a jellemzők neveit tartalmazó rendezett sorozat. Minden, ebbe az osztályba tartozó objektum rendelkezni fog a megadott nevű jellemzőkkel, de nem lesz `__dict__` nevű szótára; ekképpen az ilyen osztályokban nem lehetséges új jellemzők elhelyezése, sem a már meglévők eltávolítása. Az ilyen objektumok a hagyományos objektumokhoz képest kevesebb memóriát használnak, bár ez feltehetőleg nem vezet jelentésbeli különbségekhez, hacsak nem használunk igen nagy számú ilyen objektumot.

A jellemzők elérésének szabályozása

Előfordulnak olyan helyzetek, amikor kényelmes lehet egy olyan osztállyal rendelkezni, amelyben a jellemzők értékét nem tároljuk, hanem szükség esetén kiszámítjuk. Íme egy ilyen osztály teljes megvalósítása:

```
class Ord:
def __getattr__(self, char):
        return ord(char)
```

Most, hogy létrehoztuk az `Ord` osztályt, példányosíthatjuk is: `ord = Ord()`. Ezzel létrehoztuk a beépített `ord()` függvény működését megvalósító objektumot, amely minden olyan karakter esetében működik, amely használható azonosítóként.

Például az `ord.a` visszatérési értéke 97, az `ord.Z`-é 90, az `ord.á`-é pedig 229. (De az `ord.!` és társai nyelvtanilag hibásak.)

Gondoljunk arra, hogy ha az `Ord` osztályt az IDLE fejlesztői környezetben adtuk volna meg, gondjaink támadtak volna az `ord = Ord()` utasítás kiadásakor. Ennek oka abban keresendő, hogy az objektumpéldány neve megegyezett volna a beépített `ord()` függvény nevével, amelyet az `Ord` osztály is használ. Így az `ord()` hívás valójában az `ord` nevű objektumpéldányt hívta volna meg, és `TypeError` kivételt eredményezett volna. A probléma nem jelentkezne, ha az `Ord` osztály egy betöltött modulban kapna helyet, ugyanis a fejlesztői környezetben közvetlenül megadott `ord` objektum, illetve a beépített `ord()` függvény – amelyet az `Ord` osztály is használ – két különböző modulban kapna helyet, így nem írnák egymást felül. Ha valaha is tényleg szükségünk van arra, hogy egy osztályt közvetlenül a fejlesztői környezetben adjunk meg, illetve egy beépített objektum nevét használjuk saját objektumunk nevéül, akkor ezt úgy tudjuk megtenni, hogy figyelünk arra, hogy az osztály a beépített objektumot hívja – esetünkben be kell töltenünk a `builtins` modult, amely egyértelművé teszi a beépített függvények hívását, mégpedig úgy, hogy a sima `ord()` hívás helyett a `builtins.ord()` hívást hajtja végre.

Íme egy újabb aprócska, ámde teljes osztály. Ez az osztály „állandók" létrehozását teszi lehetővé. Az osztály megkerülésével nem jelenthet gondot az értékek megváltoztatása, de a legkönnyebben elkövethető hibák kivédésénél jó szolgálatot tehet.

```
class Const:

    def __setattr__(self, name, value):
        if name in self.__dict__:
            raise ValueError("cannot change a const attribute")
        self.__dict__[name] = value

    def __delattr__(self, name):
        if name in self.__dict__:
            raise ValueError("cannot delete a const attribute")
        raise AttributeError("'{0}' object has no attribute '{1}'"
                             .format(self.__class__.__name__, name))
```

Az osztály segítségével állandó objektumokat hozhatunk létre, például így: `const = Const()`; és lehetőség nyílik a jellemzők beállítására is, például: `const.limit = 591`. Ha egyszer beállítottuk egy jellemző értékét, annyiszor olvassuk ki, ahányszor úri kedvünk diktálja, ugyanakkor minden, az érték megváltoztatására, illetve törlésére irányuló kísérlet eredménye `ValueError` kivétel lesz. Nem valósítottuk meg újra a `__getattr__()` tagfüggvényt, hiszen az alaposztály `object.__getattr__()` tagfüggvénye épp azt teszi, amire szükségünk van – visszaadja az átadott jellemző értékét, vagy a megfelelő jellemző hiányában `AttributeError` kivételt vált ki. A `__delattr__()` tagfüggvény megvalósításakor a `__getattr__()` tagfüggvény nem létező jellemzők esetén adott hibaüzenetét utánozzuk. Ezt úgy tudjuk megtenni, hogy kinyerjük annak az osztálynak a nevét, amelyikben éppen

tartózkodunk, csakúgy, mint a nem létező jellemző nevét. Az osztály működik, hiszen az osztály `__dict__` nevű szótárát használjuk, és ezt használja az alaposztály `__getattr__()`, `__setattr__()` és `__delattr__()` tagfüggvénye is. Mindazonáltal megjegyeznénk, hogy mi csak az alaposztály `__getattr__()` tagfüggvényét használtuk. A jellemzők elérését megvalósító különleges szerepű tagfüggvények listája a 8.2. táblázatban található.

8.2. táblázat *A jellemzők elérésére szolgáló különleges szerepű tagfüggvények*

Tagfüggvény	Használat	Leírás
`__delattr__(self, name)`	`del x.n`	Törli az `x` objektum `n` jellemzőjét.
`__dir__(self)`	`dir(x)`	Az `x` objektum jellemzőinek neveiből álló listával tér vissza.
`__getattr__(self, name)`	`v = x.n`	Az `x` objektum `n` jellemzőjének értékét adja vissza, ha a közvetlen keresés sikertelen.
`__getattribute__(self, name)`	`v = x.n`	Az `x` objektum `n` jellemzőjének értékét adja vissza; lásd a szöveget.
`__setattr__(self, name, value)`	`x.n = v`	Az `x` objektum `n` jellemzőjének `v`-t adja értékül.

Az állandók megvalósításának másik módja is kínálkozik: nevesített rendezett sorozatokat használhatunk. Íme néhány példa:

```
Const = collections.namedtuple("_", "min max")(191, 591)
                                        # az eredmény: (191, 591)
Const.min, Const.max
Offset = collections.namedtuple("_", "id name description")(*range(3))
Offset.id, Offset.name, Offset.description   # az eredmény: (0, 1, 2)
```

A nevesített rendezett sorozatokat mindkét esetben csak egy egyszer használatos névvel jelöltük, mivel egyszerre csak egy nevesített rendezett sorozatra van szükségünk, nem pedig a `tuple` típus egy alosztályára, amelyből nevesített `tuple`-példányokat hozhatunk létre. Bár a Python nem támogatja a felsoroló (enum) adattípust, hasonló hatást érhetünk el nevesített rendezett sorozatok használatával, mint ahogy azt fent is tettük.

Image 262

A jellemzők elérését megvalósító különleges szerepű tagfüggvényekre vetett utolsó pillantásunkkal visszatérünk egy olyan példához, amellyel először a 6. fejezetben találkoztunk. Abban a fejezetben hoztuk létre az `Image` osztályt, amelynek a szélessége, magassága és háttérszíne a példányosítás során később meg nem változtatható értéket kap (bár, ha betöltünk egy képet, az értékek megváltoznak). Az értékek elérését csak olvasható tulajdonságokkal valósítottuk meg. Például így:

```
@property
def width(self):
    return self.__width
```

A kódolás elvégzése egyszerű, ám ha sok csak olvasható tulajdonsággal dolgozunk, könnyen válhat unalmassá. Íme egy eltérő megoldás, amely az `Image` csak olvasható tulajdonságait egyetlen tagfüggvényben kezeli:

```
def __getattr__(self, name):
    if name == "colors":
        return set(self.__colors)
    classname = self.__class__.__name__
    if name in frozenset({"background", "width", "height"}):
        return self.__dict__["_{0}__{1}".format(classname, name)]
    raise AttributeError("'{0}' object has no attribute '{1}'"
                         .format(classname, name))
```

Ha megpróbáljuk elérni egy objektum jellemzőjét, de a jellemzőt nem találjuk, a Python a `__getattr__()` tagfüggvényt hívja meg (amennyiben a tagfüggvény létezik, és nem végeztük el a `__getattribute__()` tagfüggvény újbóli megvalósítását), mégpedig a jellemzőt adva át paraméterként. A `__getattr__()` tagfüggvény megvalósításaival szemben elvárás, hogy amennyiben az átadott jellemzőt nem képesek kezelni, `AttributeError` kivételt kell kiváltaniuk.

Például ha az `image.colors` utasítást tekintjük, a Python keresni fogja a `colors` jellemzőt, és miután nem találja meg, kiadja az `Image.__getattr__(image,"colors")` hívást. Esetünkben a `__getattr__()` tagfüggvény kezeli a `colors` jellemzőnevet, és a gép által használt színeket tartalmazó halmaz másolatát adja vissza.

A többi jellemző nem módosítható, így aztán biztonságos őket közvetlenül visszaadni a hívónak. Írhattunk volna minden további jellemzőhöz önálló `elif`-ágat, például így:

```
elif name == "background":
    return self.__background
```

Ehelyett egy tömörebb megközelítés mellett döntöttünk. Minthogy tudjuk, hogy valójában minden olyan jellemző, amely nem bír különleges szereppel az objektumban, a `self.__dict__` szótárba kerül, úgy döntöttünk, hogy közvetlenül fogunk hozzájuk férni. A privát jellemzők esetében (azokról van szó, amelyeknek a neve két aláhúzással kezdődik) a nevek úgy alakulnak át, hogy *`_osztályNév__jellemzőNév`* alakot kapjanak. Amikor az objektum privát szótárából előhívjuk a jellemző értékét, erre figyelnünk kell.

Ahhoz tehát, hogy a privát jellemzőket meg tudjuk keresni, illetve elő tudjuk állítani a szokásos `AttributeError` hibaüzenet szövegét, névátalakítás szükséges. Ha ezt meg szeretnénk oldani, tudnunk kell annak az osztálynak a nevét, amelyben tartózkodunk. (Ez esetleg nem az `Image` osztály lesz, ugyanis a kérdéses objektum esetleg az `Image` osztály egy alosztályának a példánya.) Minden objektum rendelkezik egy `__class__` nevű különleges szerepű jellemzővel, vagyis a tagfüggvények belsejében mindig használhatjuk a `self.__class__` kifejezést, a nemkívánatos önhívás pedig a `__getattr__()` függvény használatával kerülhető el.

Figyeljünk fel arra a parányi különbségre, hogy a `__getattr__()` és a `self.__class__` használatakor a jellemzőt az objektumpéldány osztályában érjük el (ez az osztály lehet alosztály is), ugyanakkor a jellemző közvetlen elérésekor azt az osztályt használjuk, amelyikben a jellemző megadása történt.

Eddig nem tárgyaltuk a `__getattribute__()` különleges szerepű tagfüggvényt. Míg a `__getattr__()` tagfüggvényt olyankor hívjuk meg, amikor egy (különleges szereppel nem bíró) jellemzőt keresünk, a `__getattribute__()` jellemzőt minden olyan alkalommal elsőként hívjuk, amikor hozzá kell férnünk egy osztályhoz. Bár vannak esetek, amikor hasznos vagy akár létfontosságú lehet a `__getattribute__()` tagfüggvény hívása, újbóli megvalósítása néhány buktatót rejt magában. Az ilyen megvalósításokat esetében figyelnünk kell arra, hogy nehogy saját magát hívja meg a tagfüggvény – ilyen esetekben gyakori a `super().__getattribute__()` és az `object.__getattribute__()` hívás használata. És minthogy a `__getattribute__()` tagfüggvényt minden elem elérésekor meghívjuk, az újbóli megvalósítás könnyen vezethet teljesítménycsökkenéshez közvetlen elérés esetén. A könyvben bemutatott egyetlen osztály sem valósítja meg újra a tagfüggvényt.

Funktorok

Figyelem!

A Python nyelvben a *függvényobjektumok* olyan objektumhivatkozások, amelyek egy hívható objektumra (függvény, lambdafüggvény, tagfüggvény) mutatnak. A meghatározásba beleférnek az osztályok is, hiszen egy osztályra mutató objektumhivatkozás is hívható objektumra mutat: ha az osztályokat hívjuk, a visszatérési érték egy az adott osztályba tartozó objektum lesz. Ilyen hívás például az `x = int(5)`. A számítástudomány a *functor* névvel az olyan objektumokat illeti, amelyek függvényként hívhatók. A Pythonban tehát a funktor nem egyéb, mint egy újabb függvényobjektum-féleség. Minden olyan osztály funktor, amely rendelkezik a `__call__()` nevű különleges szerepű tagfüggvénnyel. A funktorok legfontosabb előnye, hogy képesek néhány, az állapotukra vonatkozó információ kezelésére. Például létrehozhatunk olyan funktort, amely karakterláncok végéről távolít el írásjeleket. Az alábbi példa szemlélteti a létrehozását és a használatát:

```
strip_punctuation = Strip(",;:.!?")
strip_punctuation("Land ahoy!")          # az eredmény: 'Land ahoy'
```

A fenti példában a `Strip` funktort példányosítjuk, majd az `",;:.!?"` kezdőértéket adjuk neki. Amikor a példányt meghívjuk, a neki átadott karakterláncot adja vissza, de előbb eltávolítja a végéről az írásjeleket. Íme a `Strip` osztály teljes megvalósítása:

```
class Strip:
    def __init__(self, characters):
        self.characters = characters
    def __call__(self, string):
        return string.strip(self.characters)
```

Ugyanez a hatás elérhető lett volna egy sima függvénnyel vagy egy lambdafüggvénnyel is, de ha kicsivel több állapotinformáció tárolására van szükség, illetve bonyolultabb az elvégzendő feladat, gyakran a funktorok használata jelenti a helyes megoldást.

Az, hogy a funktorok egy osztály használatával képesek állapotok rögzítésére, igen nagy sokoldalúságot és érdekes lehetőségeket rejt magában, de előfordul, hogy nincs szükségünk ilyen sok mindenre. Az állapot rögzítésének másik módszerét a *záradékok* használata jelenti. A záradék olyan függvény, illetve tagfüggvény, amely valamilyen külső állapot rögzítésére képes. Például:

```
def make_strip_function(characters):
    def strip_function(string):
        return string.strip(characters)
    return strip_function

strip_punctuation = make_strip_function(",;:.!?")
strip_punctuation("Land ahoy!")          # az eredmény: 'Land ahoy'
```

A `make_strip_function()` függvény egyetlen paraméterét a levágandó karakterek jelentik, a visszatérési értéke pedig egy függvény, a `strip_function()` nevű. Ez utóbbi paramétere egy karakterlánc, feladata pedig a záradék létrehozásakor megadott karakterek törlése. Vagyis, hasonlóan ahhoz, ahogy a `Strip` osztályból annyi példányt hozunk létre, amennyi nekünk tetszik, méghozzá mindegyiket a saját levágandó karaktereivel, a karakterek eltávolítását végző függvényből is sokat hozhatunk létre, és mindegyik saját levágandó karaktereket kaphat.

A funktorok klasszikus felhasználási területét jelentik a rendező algoritmusok kulcsfüggvényei. Íme a `SortKey.py` fájlban található általános célú `SortKey` funktorosztály:

```
class SortKey:

    def __init__(self, *attribute_names):
        self.attribute_names = attribute_names

    def __call__(self, instance):
        values = []
        for attribute_name in self.attribute_names:
            values.append(getattr(instance, attribute_name))
        return values
```

Amikor létrejön egy a `SortKey` osztályba tartozó objektumpéldány, egy rendezett sorozatban tárolja azokat a jellemzőneveket, amelyeket az előkészítés során kapott meg. Amikor az objektum hívása megtörténik, létrehoz egy listát a neki átadott példány jellemzőinek értékeiből, abban a sorrendben, amit a `SortKey` osztály előkészítésekor megadtunk. Például képzeljük el, hogy van egy `Person` nevű osztályunk:

```
class Person:
    def __init__(self, forename, surname, email):
        self.forename = forename
        self.surname = surname
        self.email = email
```

Tegyük fel, hogy a `people` listában `Person` osztályú objektumok vannak. A listát vezetéknevek alapján a következőképpen rendezhetjük: `people.sort(key=SortKey("surname"))`. Ha sok ember van, biztosan lesznek egyező vezetéknevűek, így a vezetékneveken belül keresztnév szerint is rendezhetünk. A módszer a következő: `people.sort(key=SortKey("surname", "forename"))`. Ha sok, teljesen egyező nevű ember van, az e-mail cím jelenthet újabb rendezési szempontot, és természetesen megoldható az is, hogy a rendezés először történjen keresztnév, aztán vezetéknév alapján: ehhez a `SortKey` funktornak átadott jellemzők neveinek sorrendjét kell megváltoztatnunk.

Ugyanez az eredmény más módon is elérhető, anélkül, hogy akár egyetlen funktort is létre kellene hoznunk. A megoldást az `operator` modul `operator.attrgetter()` függvényének használata jelenti. Ekkor a vezetéknév alapján történő rendezés a következő módon történne: `people.sort(key=operator.attrgetter("surname"))`. Ehhez hasonlóan, a vezeték- és keresztnév alapján történő rendezés a következőképp történne: `people.sort(key=operator.attrgetter("surname", "forename"))`.

Az `operator.attrgetter()` függvény visszatérési értéke egy függvény (egy záradék), amely egy objektumra meghívva az objektumnak a záradék létrehozásakor megadott jellemzőit adja vissza.

A funktorokat alighanem ritkábban használják a Pythonban, mint az egyéb olyan nyelvekben, amelyek támogatják a használatukat. Ennek oka feltehetőleg abban keresendő, hogy a Python más, hasonló feladatok ellátására alkalmas eszközöket is rendelkezésre bocsát a funktorok mellett – például a záradékokat, illetve az elemek és a jellemzők kiolvasását megvalósító függvényeket.

Környezetkezelők

A környezetkezelők a kód egyszerűsítését azáltal teszik lehetővé, hogy bizonyos műveleteket adott kódblokkok végrehajtása előtt és után mindig elvégeznek. Ez a viselkedés arra vezethető vissza, hogy a környezetkezelők megadnak két különleges szerepű tagfüggvényt – `__enter__()` és `__exit__()` –, amelyeket a Python a `with` utasítás hatókörében különlegesen kezel. Amikor a környezetkezelő a `with` utasításon belül jön létre, automatikusan meghívjuk az `__enter__()` tagfüggvényét, amikor pedig kiérünk a környezetkezelő hatóköréből, az `__exit__()` tagfüggvény automatikus hívására kerül sor.

Megírhatjuk a saját környezetkezelőinket, vagy használhatunk előre megadottakat – ahogy a későbbiekből kiderül, a beépített `open()` függvénytől visszakapott fájlobjektumok is környezetkezelők. A környezetkezelők az alábbi utasításformával használhatók:

```
with kifejezés as változó:
    kódblokk
```

A *kifejezés* vagy eleve egy környezetkezelő, vagy a kiértékelésének kell környezetkezelőt eredményeznie. Ha az elhagyható `as` *változó* részt megadjuk, a változó értéke a környezetkezelő `__enter__()` tagfüggvényétől visszakapott objektumra fog hivatkozni. A visszakapott objektum gyakran maga a környezetkezelő. Mivel a környezetkezelő biztosan végrehajtja a kilépési kódját (ebben a kivételek sem gátolják), a környezetkezelők sok helyzetben használhatók a `finally` blokkok megszüntetésére.

A Python típusai közül néhány tulajdonképpen környezetkezelő – például az `open()` függvénytől visszakapott valamennyi fájlobjektum –, így szükségtelenné válnak a fájlkezelés során használt `finally` blokkok. Az alábbi két egyenértékű kódrészlet is ezt érzékelteti (feltételezzük, hogy a `process()` egy más helyen megadott függvény):

```
fh = None
try:
fh = open(filename)
for line in fh:
process(line)
except EnvironmentError as err:
print(err)
finally:
if fh is not None:
fh.close()
```

```
try:
with open(filename) as fh:
for line in fh:
process(line)
except EnvironmentError as err:
print(err)
```

A fájlobjektumok olyan környezetkezelők, amelyeknek a kilépéskor végrehajtandó kódja minden esetben lezárja a megnyitott állományokat. A kilépési kód a kivételek esetleges létrejöttétől függetlenül végrehajtódik, bár az utóbbi esetben a kivételről tudomást szerzünk. Így biztosítható, hogy a fájl le lesz zárva, és módunkban áll a felmerülő hibák kezelése is – ezúttal egy hibaüzenetet írunk a felhasználó képernyőjére.

Az az igazság, hogy a környezetkezelőkkel nem kell feltétlenül továbbadatnunk a kivételeket, de ha így járunk el, akkor lényegében elrejtjük a kivételeket, ami szinte bizonyosan kódolási hiba. A beépített környezetkezelők, csakúgy, mint az alapkönyvtárban lévők, továbbadják a kivételeket.

Vannak esetek, amikor egyszerre több környezetkezelőt is használnunk kell. Például:

```
try:
    with open(source) as fin:
        with open(target, "w") as fout:
            for line in fin:
                fout.write(process(line))
except EnvironmentError as err:
    print(err)
```

A fenti példában a forrásfájl sorait olvassuk be, amelyeket feldolgozás után a célfájlba írunk.

Az egymásba ágyazott `with` utasítások használata gyorsan vezethet rengeteg behúzáshoz. Szerencsére az alapkönyvtár `contextlib` modulja további támogatással segíti a környezetkezelők használatát, többek között a `contextlib.nested()` függvény formájában, amely lehetővé teszi, hogy kettő vagy akár több környezetkezelőt alkalmazzunk ugyanazon a `with` utasításon belül – így elkerülhető az egymásba ágyazásuk. Íme a fent bemutatott kód átírt változata – elhagyva azokat a sorokat, amelyek nem változtak az előző formához képest.

```
try:
    with contextlib.nested(open(source), open(target, "w")) as (
            fin, fout):
        for line in fin:
```

Nem csak a fájlobjektumok környezetkezelők. Például számos, a többszálúsággal kapcsolatos és a zárolások során használt osztály is környezetkezelő. Környezetkezelőket használhatunk a `decimal.Decimal` típusú számok esetében – ez különösen akkor hasznos, ha adott beállítások mellett (mondjuk egy adott pontossággal) szeretnénk számításokat végezni.

Többszálúság ☞ 420

Ha saját környezetkezelőt szeretnénk, egy olyan osztályt kell létrehoznunk, amely két tagfüggvényt valósít meg – az `__enter__()` és az `__exit__()` tagfüggvényről van szó. Minden olyan alkalommal, amikor egy az osztályból példányosított objektumot egy `with` utasításban használunk, az `__enter__()` tagfüggvény hívására kerül sor, és ennek a visszatérési értéke kerül az `as` kulcsszó után megadott változóba (ha nem adtunk meg változót, akkor a visszatérési érték elvész). Amikor a vezérlés kikerül a `with` hatóköréből, az `__exit__()` tagfüggvény hívása történik meg – ha közben kivétel lép fel, akkor ennek az adatai lesznek a tagfüggvény paraméterei.

Tegyük fel, hogy műveletek sorát szeretnénk végrehajtani egy adott listán, mégpedig tranzakciószerűen – azaz azt szeretnénk, hogy vagy minden művelet legyen végrehajtva, vagy egyik sem, és ekképp az eredménylista legyen mindig ismert állapotú. Például ha egy egész számokból álló listánk van, amelyhez egy újabb egész számot szeretnénk adni, egyet törölni, néhányat pedig megváltoztatni, és mindezt egyetlen műveletként, akkor az alábbihoz hasonló kódot írhatunk:

```
try:
    with AtomicList(items) as atomic:
        atomic.append(58289)
        del atomic[3]
        atomic[8] = 81738
        atomic[index] = 38172
except (AttributeError, IndexError, ValueError) as err:
    print("no changes applied:", err)
```

Ha nem lép fel kivétel, akkor minden műveletet elvégzünk az eredeti listán – az `items` nevűn –, amennyiben azonban kivétel keletkezik, semmilyen módosítás nem történik. Íme az `AtomicList` környezetkezelő kódja:

```
class AtomicList:

    def __init__(self, alist, shallow_copy=True):
        self.original = alist
        self.shallow_copy = shallow_copy

    def __enter__(self):
        self.modified = (self.original[:] if self.shallow_copy
                         else copy.deepcopy(self.original))
        return self.modified

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is None:
            self.original[:] = self.modified
```

Sekély és mély másolás 143 ☜

Amikor egy `AtomicList` objektum létrejön, megtartjuk az eredeti listára mutató hivatkozást, és átgondoljuk, hogy sekély másolást kell-e használnunk. (A sekély másolás megfelelő módszer számokból vagy karakterláncokból álló lista esetén; de olyan listáknál nem, amelyek további listákat vagy egyéb gyűjteményes típusú szerkezeteket tartalmaznak.)

Később, amikor az `AtomicList` környezetkezelő objektumot a `with` utasítás belsejében használjuk, az objektum `__enter__()` tagfüggvényét hívjuk meg. Ezen a ponton az eredeti listát lemásoljuk, és a másolatot adjuk vissza, így minden változtatás a másolaton fog megtörténni.

Amikor elérjük a `with` hatókörének végét, az `__exit__()` tagfüggvény hívódik meg. Ha nem lépett fel kivétel, az `exc_type` („exception type”, kivétel típusa) változó értéke `None` lesz, és tudjuk, hogy biztonságosan cserélhetjük le az eredeti lista elemeit a módosított listából származókra. (Nem adhatjuk ki a `self.original = self.modified` utasítást, ugyanis az csak az eredeti objektum objektumhivatkozását cseréli le, és az eredeti listán semmit nem változtatna.) Ha azonban kivétel történt, nem nyúlunk az eredeti listához, a módosítottat pedig elvetjük.

Az `__exit__()` tagfüggvény visszatérési értéke arra jó, hogy jelezze, hogy történt-e továbbadandó kivétel. A `True` érték azt jelenti, hogy minden kivételt kezeltünk, továbbadásra nincs szükség. Normális esetben mindig `False` értéket – vagy olyasmit, ami logikai értelemben `False` értékké értékelődik ki – adunk vissza, lehetővé téve, hogy az esetlegesen létrejött kivételek továbbadódjanak. Ha nem adunk meg konkrét visszatérési értéket, az `__exit__()` tagfüggvényünk `None` objektumot ad vissza, amelynek a kiértékelésekor `False` értéket kapunk, és ez – helyesen – ahhoz vezet, hogy a kivételek továbbadódnak.

A 10. fejezetben arra használjuk a saját környezetkezelőket, hogy megbizonyosodjunk róla, hogy a csatolókapcsolatokat, illetve a gzip tömörítésű fájlokat lezártuk. A 9. fejezetben a többszálúsággal kapcsolatos modulok pár környezetkezelőjét azért vesszük igénybe, hogy megbizonyosodjunk róla, hogy a kölcsönösen kizáró zárolások fel vannak-e oldva. Az Olvasó a fejezet gyakorlataiban lehetőséget kap egy az előzőnél általánosabb, tranzakciókkal dolgozó környezetkezelő megírására.

Leírók

A leírók (descriptor) olyan osztályok, amelyek a más osztályok jellemzőihez történő hozzáférést szabályozzák. Minden olyan osztály, amely a `__get__()`, a `__set__()` és a `__delete__()` különleges szerepű tagfüggvények közül legalább egyet megvalósít, leírónak tekinthető és akként használható.

A beépített `property()` és `classmethod()` függvények megvalósítása leírók használatával történt. A leírók megértésének kulcsa abban rejlik, hogy bár a leíró osztályból az objektumokat az osztályon belül osztályjellemzőként példányosítjuk, a Python a leírókat az osztályból létrehozott példányokon keresztül éri el.

Hogy világosabb legyen, tegyük fel, hogy van egy osztályunk, amelynek a példányai bizonyos karakterláncokat tárolnak. A megszokott módon szeretnénk hozzáférni a karakterláncokhoz, például tulajdonságként, ugyanakkor szeretnénk egy az XML-nek megfelelően levédett változatot is kapni a karakterláncokból, amikor csak kedvünk tartja. Az egyik egyszerű megoldás az lenne, hogy minden alkalommal, amikor értékül adunk egy karakterláncot, azonnal létrehozzuk az XML-ben használatos másolatát is. Amennyiben azonban több ezer karakterlánccal dolgozunk, és ezek közül csak néhánynak szeretnénk az XML-ben használatos változatát beolvasni, azokat is csak egyszer-egyszer, rengeteg felesleges számítást végeznénk, és rengeteg memóriát pocsékolnánk el. Így inkább létrehozunk egy olyan leírót, amely az XML-nek megfelelő, levédett változatokat csak szükség esetén állítja elő, és nem tároljuk azokat. Elsőként azt az osztályt írjuk meg, amelyik a leírót használni fogja:

```
class Product:
    __slots__ = ("__name", "__description", "__price")

    name_as_xml = XmlShadow("name")
    description_as_xml = XmlShadow("description")

    def __init__(self, name, description, price):
        self.__name = name
        self.description = description
        self.price = price
```

A fentiekben csak a tulajdonságokat megvalósító kódot nem közöltük. A `name` tulajdonság csak olvasható, a `description` és a `price` pedig írható-olvasható tulajdonság; mindegyiket a szokásos módszerrel állítottuk be. (A kód egésze az `XmlShadow.py` fájlban található.)

A `__slots__` változóval biztosítottuk, hogy az osztály ne rendelkezzen `__dict__` szótárral, és csak a három megadott privát jellemző tárolására legyen képes; mindez nem áll kapcsolatban a leírók használatával, és nem is szükséges hozzá. A `name_as_xml` és a `description_as_xml` osztályszintű jellemzőket úgy állítottuk be, hogy az `XmlShadow` leíróosztály példányait kapják értékül. Bár egyetlen `Product` objektum sem rendelkezik sem `name_as_xml`, sem `description_as_xml` nevű jellemzővel, a leírónak köszönhetően a kód az alábbi formát kaphatja (ezúttal a modul dokumentációs tesztjeiből idézve):

```
>>> product = Product("Chisel <3cm>", "Chisel & cap", 45.25)
>>> product.name, product.name_as_xml, product.description_as_xml
('Chisel <3cm>', 'Chisel &lt;3cm&gt;', 'Chisel &amp; cap')
```

Azért járhatunk el így, mert amikor például a `name_as_xml` tulajdonság elérésével próbálkozunk, a Python rájön, hogy a `Product` osztálynak van ilyen nevű leírója, és így a leíróval olvassa ki a jellemző értékét. Íme az `XmlShadow` leíróosztály teljes kódja:

```
class XmlShadow:

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name

    def __get__(self, instance, owner=None):
        return xml.sax.saxutils.escape(
                                getattr(instance, self.attribute_name))
```

Amikor a `name_as_xml` és a `description_as_xml` objektumok létrejönnek, a `Product` osztály megfelelő jellemzőjét az `XmlShadow` osztály előkészítőjének adjuk át, így aztán a leíró tudja, hogy melyik jellemzővel kell dolgoznia. Később, amikor a `name_as_xml` és a `description_as_xml` jellemzőket keressük, a Python a leíró `__get__()` tagfüggvényét hívja meg. A `self` paraméter a leíróosztály egy példánya, az `instance` a `Product`-é (a `Product` `"self"`-je), az `owner` paraméter pedig a leírót birtokló osztályt jelzi (esetünkben a `Product`-ról van szó). A `getattr()` függvényt használjuk arra, hogy a `Product`-ból a megfelelő jellemzőt (esetünkben a megfelelő tulajdonságot) kiolvassuk, és az XML-nek megfelelően levédett változatát adjuk vissza.

Ha a használat módja az lenne, hogy a `Product` objektumoknak csak egy kis hányadából olvassuk ki az XML-nek megfelelően levédett karakterláncokat, de ezek a karakterláncok gyakran hosszúak, és sokszor olvassuk ki ugyanazokat, érdemes lenne gyorsítótárat használni. Például így:

```
def __init__(self, attribute_name):
    self.attribute_name = attribute_name
    self.cache = {}

def __get__(self, instance, owner=None):
    xml_text = self.cache.get(id(instance))
    if xml_text is not None:
        return xml_text
```

```
        return self.cache.setdefault(id(instance),
                xml.sax.saxutils.escape(
                                getattr(instance, self.attribute_name)))
```

A szótár kulcsaként nem magukat az objektumpéldányokat, hanem csak az egyedi azonosítójukat tároljuk, mivel a szótár kulcsainak kivonatolhatóknak kell lenniük – az azonosítók pedig ilyenek –, ugyanezt a követelményt azonban nem szeretnénk a `CachedXmlShadow` leírót használó osztályokkal szemben is megfogalmazni. A kulcsra azért van szükség, mert a leírók nem példányonként, hanem osztályokként jönnek létre. (A `dict.setdefault()` tagfüggvény számunkra igen kényelmesen a neki átadott kulcs értékét adja vissza, amennyiben pedig ilyen kulcsú elem nem létezik, új elemet hoz létre az átadott kulccsal és értékkel, és ilyenkor az értéket kapjuk vissza.)

Most, hogy láttuk, hogy a leírók miként képesek olyan adatok létrehozására, amelyeket nem feltétlenül tárolnak, megvizsgálunk egy olyan leírót, amelyet egy objektumpéldány jellemzőiben lévő adatok tárolására használhatunk – magának az objektumpéldánynak semmit sem kell tárolnia. Ebben a példában ismét egy szótárat használunk, de valósághűbb környezetben az adatot fájlban vagy adatbázisban tárolnánk. Az alábbiakban ismertetjük a `Point` osztály módosított változatát, amely az `ExternalStorage.py` fájlban található leírót használja:

```
class Point:
    __slots__ = ()
    x = ExternalStorage("x")
    y = ExternalStorage("y")
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Azzal, hogy a `__slots__` változó értékéül egy üres rendezett sorozatot adunk meg, biztosítjuk, hogy az osztály egyáltalán ne legyen képes adatjellemzők tárolására. Amikor a `self.x` kap értéket, a Python rájön, hogy van „x” nevű leíró, és a leíró `__set__()` tagfüggvényét használja. Az osztály további részét nem közöljük, mert megegyezik a 6. fejezetben ismertetett eredeti `Point` osztályban lévőkkel. Íme a teljes `ExternalStorage` leíróosztály:

```
class ExternalStorage:
    __slots__ = ("attribute_name",)
    __storage = {}
    def __init__(self, attribute_name):
        self.attribute_name = attribute_name
    def __set__(self, instance, value):
        self.__storage[id(instance), self.attribute_name] = value
    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.__storage[id(instance), self.attribute_name]
```

Minden `ExternalStorage` objektum egyetlen adatjellemzővel, az `attribute_name` nevűvel rendelkezik. A jellemző az objektumpéldányt birtokló osztály adatjellemzőjét tárolja. Minden alkalommal, amikor egy jellemző értéket kap, az értéket a `__storage` nevű privát, osztályszintű szótárban tároljuk. Amikor egy adat kiolvasására kerül sor, azt is a `__storage` nevű szótárból olvassuk ki.

A leírók minden tagfüggvényéhez hasonlóan a `self` is a leíróosztály egy példánya, az `instance` pedig azon objektum `self` tagfüggvényének feleltethető meg, amely a leírót tartalmazza, azaz ebben a helyzetben a `self` egy `ExternalStorage` objektum, az `instance` pedig egy `Point` objektum.

Bár a `__storage` osztályszintű jellemző, a `self.__storage` utasítással érhetjük el (pontosan úgy, mint ahogy a tagfüggvények hívását is a `self.`*`tagfüggvény`*`()` utasításformával oldhatjuk meg), hiszen a Python egy objektumpéldány jellemzőjeként fogja keresni, és miután nem talált semmi, a keresést az osztályszintű jellemzők között folytatja. Az ilyen megközelítés egyetlen elméleti hátránya abban rejlik, hogy amennyiben egyforma nevű osztályszintű és objektumpéldány-szintű jellemzőnk van, az egyik el fogja takarni a másikat. (Ha ebbe a problémába a valóságban is beleütközünk, az osztályszintű jellemzőre az osztály nevével hivatkozhatunk, vagyis így: `ExternalStorage.__storage`.) Bár az osztály nevének bedrótozása az alosztályok kialakítása során általában rossz ötlet, a privát jellemzők esetében nem igazán van jelentősége: a Python így is, úgy is végrehajt bizonyos névváltoztatásokat.

A `__get__()` különleges szerepű tagfüggvény megvalósítása azért valamennyivel kifinomultabb az előzőeknél, ugyanis biztosítjuk annak lehetőségét, hogy maga az `ExternalStorage` objektum elérhető legyen. Például ha van egy `p = Point(3, 4)` objektumunk, az *x* koordinátát a `p.x` formában érhetjük el, és a `Point.x` utasítással férhetünk hozzá az `x` értékeket tároló `ExternalStorage` objektumhoz.

A leírók tárgyalását azzal tesszük teljessé, hogy létrehozzuk azt a `Property` leíróosztályt, amely a beépített `property()` függvény viselkedését utánozza, legalábbis ami a beállítófüggvényeket és a kiolvasófüggvényeket illeti. A kód a `Property.py` fájlban található. Íme a leíróosztályt használó `NameAndExtension` osztály teljes kódja:

```
class NameAndExtension:
    def __init__(self, name, extension):
        self.__name = name
        self.extension = extension
    @Property                   # Az egyéni Property leírót használja
    def name(self):
        return self.__name
    @Property                   # Az egyéni Property leírót használja
    def extension(self):
        return self.__extension
```

```
    @extension.setter                   # Az egyéni Property leírót használja
    def extension(self, extension):
        self.__extension = extension
```

A használat megegyezik a beépített `@property` díszítőével, illetve a `@tulajdonságNév.setter` díszítőével. Íme a `Property` leíróosztály megvalósításának eleje:

```
class Property:
    def __init__(self, getter, setter=None):
        self.__getter = getter
        self.__setter = setter
        self.__name__ = getter.__name__
```

Az osztály előkészítő függvénye egy vagy két függvényt vár paraméterül. Ha díszítőként használjuk, akkor csak a díszítendő függvényt kéri, és belőle jön létre a kiolvasófüggvény; a beállítófüggvény értéke pedig a `None` objektum lesz. A kiolvasófüggvény nevét fogjuk a tulajdonság neveként használni, így aztán minden tulajdonság kap egy kiolvasófüggvényt, esetleg egy beállítófüggvényt is, illetve egy nevet.

```
def __get__(self, instance, owner=None):
    if instance is None:
        return self
    return self.__getter(instance)
```

Valahányszor valami hozzányúl egy tulajdonsághoz, a kiolvasófüggvény meghívásával adjuk vissza az eredményt. A kiolvasófüggvény paramétere az objektumpéldány. Első ránézésre a `self.__getter()` utasítás úgy néz ki, mint egy tagfüggvény hívása, de nem az. A valóságban a `self.__getter` egy jellemző, mégpedig olyan, amely történetesen az átadott tagfüggvényre mutató objektumhivatkozást tárolja. Lényegében tehát annyi történik, hogy kiolvassuk a jellemzőt (`self.__getter`), majd függvényként hívjuk meg `()`. Minthogy nem tagfüggvényként, hanem függvényként hívjuk meg, magunknak kell átadnunk a megfelelő `self` objektumot. Leíróról lévén szó, a leírót használó osztályból származó `self` objektum neve valójában `instance` – hiszen a `self` a leíróobjektum. Ugyanez érvényes a `__set__()` tagfüggvényre is.

```
def __set__(self, instance, value):
    if self.__setter is None:
        raise AttributeError("'{0}' is read-only".format(
                             self.__name__))
    return self.__setter(instance, value)
```

Ha a beállítófüggvény nincs megadva, `AttributeError` kivételt váltunk ki, egyéb esetben pedig a példánnyal és az új értékkel hívjuk meg a beállítófüggvényt.

```
def setter(self, setter):
    self.__setter = setter
    return self.__setter
```

A tagfüggvény hívására akkor kerül sor, amikor a parancsértelmező például az `@extension.setter` utasításhoz ér. A `setter` paraméter helyén a díszítendő függvény áll. A tagfüggvény elraktározza az átadott beállítófüggvényt (mostantól a `__set__()` tagfüggvényben tudjuk használni), és visszaadja a `setter` paramétert – ugyanis a díszítők visszatérési értéke az általuk díszített függvény, illetve tagfüggvény.

Ezzel áttekintettük a leírók mindhárom, meglehetősen különböző felhasználási területét. A leírók olyan sokoldalú és rugalmas képességekkel gazdagítják a Python nyelvet, amelyek segítségével sok háttérmunka végezhető el, pedig az őket használó osztályok szempontjából csupán egyszerű jellemzőknek tűnnek.

Osztályszintű díszítők

Ahogy a függvények és tagfüggvények elláthatók díszítőkkel, egész osztályok díszítése is megoldható. Az osztályszintű díszítők paramétere egy osztályobjektum (a `class` kifejezés eredménye), visszatérési értékük pedig egy másik osztály – rendes esetben az általuk díszített osztály módosított változata. Ebben a részben két osztályszintű díszítőt tanulmányozunk, így ismerve meg az osztályszintű díszítők megvalósításának módját.

`SortedList` 268 ☜

A 6. fejezetben `SortedList` néven létrehoztunk egy egyedi gyűjteményes adattípust. Az osztály a `self.__list` nevű privát jellemzőben egy egyszerű listát tárolt. Az osztály tagfüggvényei közül nyolc egyszerűen ennek a privát jellemzőnek adta tovább az elvégzendő munkát. Lássuk például a `SortedList.clear()` és a `SortedList.pop()` tagfüggvény megvalósítását:

```
def clear(self):
    self.__list = []

def pop(self, index=-1):
    return self.__list.pop(index)
```

A `clear()` tagfüggvénnyel semmi teendőnk, mivel a `list` típus nem rendelkezik megfelelő tagfüggvénnyel, de a `pop()` és a másik hat, a `SortedList` osztály által átruházott tagfüggvény esetében egyszerűen meghívhatjuk a `list` osztály megfelelő tagfüggvényét. Ez a könyv `Util` moduljában található `@delegate` osztályszintű díszítő hívásával oldható meg. Íme a `SortedList` osztály új változatának eleje:

```
@Util.delegate("__list", ("pop", "__delitem__", "__getitem__",
    "__iter__", "__reversed__", "__len__", "__str__"))
class SortedList:
```

Az első paraméter annak a jellemzőnek a neve, amelyiknek a munkát át szeretnénk adni, a második pedig egy legalább egy tagfüggvényből álló sorozat. A sorozat azokat a tagfüggvényeket tartalmazza, amelyeket a `delegate()` függvénnyel meg szeretnénk valósíttatni – mi magunk nem kívánunk munkát végezni. A `SortedListDelegate.py` fájlban levő `SortedList` osztály is így közelít a problémához, ezért a listában szereplő

tagfüggvényeknek megfelelő kódot nem tartalmaz, még akkor sem, ha gondoskodik a teljes támogatásukról. Íme az az osztályszintű díszítő, amelyik gondoskodik helyettünk a tagfüggvények megvalósításáról:

```
def delegate(attribute_name, method_names):
    def decorator(cls):
        nonlocal attribute_name
        if attribute_name.startswith("__"):
            attribute_name = "_" + cls.__name__ + attribute_name
        for name in method_names:
            setattr(cls, name, eval("lambda self, *a, **kw: "
                                    "self.{0}.{1}(*a, **kw)".format(
                                    attribute_name, name)))
        return cls
    return decorator
```

Azért nem használhatunk sima díszítőt, mert a díszítőknek paramétereket szeretnénk átadni. Így aztán inkább létrehoztunk egy olyan függvényt, amely átveszi a paramétereket, és osztályszintű díszítőt ad vissza. Maga a díszítő egyetlen paramétert vár: egy osztályt – pontosan úgy, ahogy a függvénydíszítő egyetlen függvényt, illetve tagfüggvényt vár paraméterül.

Azért kell használnunk a `nonlocal` kulcsszót, hogy a beágyazott függvény a külső hatókörből származó `attribute_name` változót használhassa, és ne próbálkozzon a saját hatókörében lévő hasonló nevű változó használatával. Arra is figyelnünk kell, hogy a privát jellemzők névváltozásai miatt képesnek kell lennünk a jellemző nevének helyesbítésére. A díszítő viselkedése meglehetősen egyszerű: bejárja a `delegate()` függvénynek átadott valamennyi tagfüggvénynevet, és mindegyikhez új tagfüggvényt hoz létre. Az újonnan létrehozott tagfüggvényeket az osztály jellemzőiként állítja be, mégpedig a megadott tagfüggvény nevével.

Az `eval()` függvényt használtuk a szükséges tagfüggvények létrehozására, mivel ez egyetlen utasítás végrehajtására alkalmas, a `lambda` utasítás pedig tagfüggvény, illetve függvény előállítására való. A `pop()` függvényt előállító kód például az alábbi:

```
lambda self, *a, **kw: self._SortedList__list.pop(*a, **kw)
```

A paramétereket azért adjuk meg a * és a ** forma használatával, hogy minden paraméter számára lehetővé tegyük adott paraméterlista használatát – még azoknak a tagfüggvényeknek is, amelyekre éppen átruházzuk a munkát. Például a `list.pop()` tagfüggvény egyetlen sorszámot vár paraméterül (ha semmit nem kap, akkor alapértelmezés szerint az utolsó elemmel foglalkozik). Ez rendben is van, hiszen ha nem megfelelő számú vagy típusú paramétert adunk át, a munka elvégzéséhez meghívott `list` tagfüggvény majd kiváltja a megfelelő kivételt.

FuzzyBool
240, 247 ☜

Szintén a 6. fejezetben került elő a másodikként vizsgált díszítő. A `FuzzyBool` osztály megvalósításakor említettük, hogy csak az `__lt__()` és az `__eq__()` különleges szerepű tagfüggvényeket adtuk meg (a „<”és az „==”műveletet), minden más összehasonlító műveletet automatikusan állítottunk elő. Nem ismertettük viszont teljes egészében az osztály meghatározásának elejét:

```
@Util.complete_comparisons
class FuzzyBool:
```

A másik négy összehasonlító műveletet a `complete_comparisons()` osztályszintű díszítővel állítottuk elő. Ha adott egy osztály, amely csak a „<” (illetve a „<” és a „==”) műveleteket valósítja meg, a díszítő az alábbi logikai egyenletek alapján állítja elő a hiányzó műveleteket:

$x = y \quad \square \quad \neg (x < y \text{ Ú } y < x)$
$x \square y \quad \square \quad \neg (x = y)$
$x > y \quad \square \quad y < x$
$x \leq y \quad \square \quad \neg (y < x)$
$x \geq y \quad \square \quad \neg (x < y)$

Ha a díszítendő osztály rendelkezik a „<” és az „==” műveletekkel, a díszítő mindkettőt használni fogja, ha azonban csak a „<” megvalósított az osztályban, akkor annak alapján is képes a többi művelet előállítására. (Az igazság az, hogy a Python automatikusan előállítja a „>” műveletet, ha a „<” adott, és a „!=” műveletet is, ha az „==” művelet ismert, a „<=” művelet ismeretében pedig a „>=” művelet előállítására is képes. Mindez annyit jelent, hogy elegendő a „<”, a „<=” és az „==” műveleteket megadnunk, a Python kikövetkezteti a többit. A fent ismertetett osztályszintű díszítő az általunk megvalósítandó műveleteket azonban mindössze a „<” műveletre korlátozza. Azon felül, hogy a dolog kényelmes, azt is biztosítja, hogy az összehasonlító műveletek ugyanazt a logikát kövessék.)

```
def complete_comparisons(cls):
    assert cls.__lt__ is not object.__lt__, (
                "{0} must define < and ideally ==".format(cls.__name__))
    if cls.__eq__ is object.__eq__:
        cls.__eq__ = lambda self, other: (not
                (cls.__lt__(self, other) or cls.__lt__(other, self)))
    cls.__ne__ = lambda self, other: not cls.__eq__(self, other)
    cls.__gt__ = lambda self, other: cls.__lt__(other, self)
    cls.__le__ = lambda self, other: not cls.__lt__(other, self)
    cls.__ge__ = lambda self, other: not cls.__lt__(self, other)
    return cls
```

A díszítő a következő problémával szembesül: az `object` osztály, amelyből végső soron minden egyéb osztály származik, mind a hat összehasonlító műveletet megadja. Ha ezek közül bármelyiket használjuk, `TypeError` kivétel lép fel. Vagyis a díszítőnek meg kell

tudnia, hogy létezik-e felülírása a „<” és az „==” műveleteknek, magyarán: szabad-e használni őket. Ez könnyen megoldható, ha a díszített osztály megfelelő különleges szerepű tagfüggvényeit összehasonlítjuk az `object` objektumban találhatókkal.

Ha a díszítendő osztályban nem találunk egyedi „<” műveletet, a feltételezés megbukik, ez ugyanis a minimális kívánalom a díszítő működéséhez. Ha van egyedi „==” művelet, akkor használjuk, ha nem, akkor létrehozunk egyet. Ezt követően létrejön a többi művelet is, és visszakapjuk a díszített, immáron mind a hat művelettel rendelkező osztályt.

Az osztályok módosításának talán legegyszerűbb és legközvetlenebb módja a díszítők használata. A másik lehetőség a metaosztályok használata – ez a téma a fejezet későbbi részében kerül sorra.

Metaosztályok ☞ 397

Elvont alaposztályok

Az elvont alaposztály olyan osztály, amelyből nem lehet objektumokat példányosítani, hanem arra használjuk, hogy felületeket adjunk meg, vagyis lényegében felsoroljuk azokat a tagfüggvényeket és tulajdonságokat, amelyeket az alaposztályból származó osztályoknak meg kell valósítaniuk. Az elvont alaposztályokat egyfajta ígéretként használhatjuk – ígéretet teszünk rá, hogy a belőle származó osztályok megvalósítják mindazokat a tagfüggvényeket és tulajdonságokat, amelyeket az alaposztály megad.*

Az elvont alaposztályok legalább egy elvont tagfüggvénnyel, illetve tulajdonsággal bírnak. Az elvont alaposztályok megadhatók megvalósítás nélkül is (vagyis a kódblokkjuk a `pass` utasítás, illetve `raise NotImplementedError()`, ha a felülírást a leszármazott osztályokban ki akarjuk kényszeríteni), de konkrét megvalósítás is jelen lehet, amely a leszármazott osztályok objektumaiból azonnal meghívható. Ez utóbbi eljárásnak főleg akkor van értelme, ha a megvalósítás sok leszármazott esetében azonos. Az elvont alaposztályokban megadhatunk konkrét – nem elvont – tagfüggvényeket és tulajdonságokat is.

Az elvont alaposztályok leszármazottjaiban újra meg kell valósítanunk minden, az alaposztálytól örökölt elvont tagfüggvényt, illetve tulajdonságot – csak ennek végeztével válik példányosíthatóvá az osztály. Az olyan elvont tagfüggvények esetében, amelyek konkrét megvalósítással bírnak (még akkor, is, ha ez a megvalósítás a `pass` utasításra korlátozódik), az osztály a `super()` függvénnyel egyszerűen használatba veheti az elvont alaposztályban lévő változatot. A rendes tagfüggvények és tulajdonságok az öröklődés szabályainak figyelembe vétele mellett a megszokott módon használhatók. Minden elvont alaposztálynak rendelkeznie kell az `abc` modulban

Metaosztályok ☞ 397

* A Python elvont alaposztályainak leírását a `www.python.org/dev/peps/pep-3119` webhelyen lévő PEP 3119 dokumentum tartalmazza. A dokumentumot igazán érdemes elolvasnunk, már a benne lévő remek magyarázat miatt is.

található `abc.ABCMeta` osztályba vagy annak valamelyik leszármazottjába tartozó metaosztállyal (*ABC: Abstract Base Class*, elvont alaposztály). A metaosztályok tárgyalására nemsokára sort kerítünk.

8.3. táblázat *A numbers modul elvont alaposztályai*

Elvont alaposztály	Alaposztálya	Felület	Példák
`Number`	`object`		`complex`, `decimal.Decimal`, `float`, `fractions.Fraction`, `int`
`Complex`	`Number`	`==`, `!=`, `+`, `-`, `*`, `/`, `abs()`, `bool()`, `complex()`, `conjugate()`; illetve a `real` és az `imag` tulajdonság	`complex`, `decimal.Decimal`, `float`, `fractions.Fraction`, `int`
`Real`	`Complex`	`<`, `<=`, `==`, `!=`, `>=`, `>`, `+`, `-`, `*`, `/`, `//`, `%`, `abs()`, `bool()`, `complex()`, `conjugate()`, `divmod()`, `float()`, `math.ceil()`, `math.floor()`, `round()`, `trunc()`; illetve a `real` és az `imag` tulajdonság	`decimal.Decimal`, `float`, `fractions.Fraction`, `int`
`Rational`	`Real`	`<`, `<=`, `==`, `!=`, `>=`, `>`, `+`, `-`, `*`, `/`, `//`, `%`, `abs()`, `bool()`, `complex()`, `conjugate()`, `divmod()`, `float()`, `math.ceil()`, `math.floor()`, `round()`, `trunc()`; illetve a `real`, az `imag`, a `numerator` és a `denominator` tulajdonság	`fractions.Fraction`, `int`
`Integral`	`Rational`	`<`, `<=`, `==`, `!=`, `>=`, `>`, `+`, `-`, `*`, `/`, `//`, `%`, `<<`, `>>`, `~`, `&`, `^`, `|`, `abs()`, `bool()`, `complex()`, `conjugate()`, `divmod()`, `float()`, `math.ceil()`, `math.floor()`, `pow()`, `round()`, `trunc()`; illetve a `real`, az `imag` a `numerator` és a `denominator` tulajdonság	`int`

A Pythonban két csoportját találjuk meg az elvont alaposztályoknak: az egyiket a `collections` modulban, a másikat a `numbers` modulban. Segítségükkel egy objektumra vonatkozó kérdéseket tehetünk fel. Például ha adott az *x* változó, az `isinstance(x, collections.MutableSequence)` kifejezéssel eldönthető, hogy egy sorozatról van-e szó,

az `isinstance(x, numbers.Integral)` kifejezés pedig arra ad választ, hogy egész-e a szám. Ez főként a Python nyelv dinamikus típuskezelését tekintve hasznos, mivel ilyenkor nem feltétlenül ismerjük, vagy nem feltétlenül fontos egy objektum típusa, de esetleg érdekel bennünket, hogy vajon támogatja-e az objektum a használni kívánt műveletet. A két említett modul elvont alaposztályai a 8.3. és a 8.4. táblázatban kaptak helyet. Az `io.IOBase` szintén jelentős alaposztály: belőle származik valamennyi fájl- és folyamkezelő osztály.

Ahhoz, hogy a saját gyűjteményes, illetve szám típusú szerkezeteket megvalósító osztályaink tökéletesen illeszkedjenek a képbe, ki kell alakítanunk a kapcsolatukat a szabványos elvont alaposztályokkal. Például a `SortedList` osztály egy sorozat, de az `isinstance(L, collections.Sequence)` kifejezés értéke `False` (az `L` egy `SortedList` objektum). Ez könnyen rendbehozható, ha a származtatás a megfelelő elvont alaposztályból történik:

```
class SortedList(collections.Sequence):
```

Minthogy alaposztályként a `collections.Sequence` osztályt jelöltük meg, az `isinstance()`-vizsgálat immár `True` értéket ad vissza. Ezen felül mostantól meg kell valósítanunk az `__init__()` (vagy a `__new__()`), a `__getitem__()` és a `__len__()` tagfüggvényeket. Ezt meg is tesszük. A `collections.Sequence` elvont alaposztály rendelkezik rendes (vagyis nem elvont) megvalósításokkal is, mégpedig a `__contains__()`, az `__iter__()`, a `__reversed__()`, a `count()` és az `index()` tagfüggvényeket illetőleg. Ami a `SortedList` osztályt illeti, ezek mindegyikét újból megvalósítjuk, de ha kedvünk tartaná, használhatnánk az elvont alaposztályból származó változatokat is – ehhez mindössze annyit kellene tennünk, hogy nem írjuk meg az új változatokat. A `SortedList` osztály nem lehet a `collections.MutableSequence` alaposztály leszármazottja, hiába változtatható. Ennek az az oka, hogy a `SortedList` osztály nem rendelkezik mindazokkal a tagfüggvényekkel, amelyekkel a `collections.MutableSequence` alaposztály leszármazottjainak rendelkezniük kell – ilyen például a `__setitem__()` és az `append()` tagfüggvény. (A `SortedList` osztály kódja a `SortedListAbc.py` fájlban található.) A metaosztályokról szóló részben újabb olyan módszert ismerhetünk meg, amellyel a `SortedList` osztály a `collections.Sequence` elvont alaposztály leszármazottjává tehető.

Metaosztályok ☞ 397

Most, hogy láttuk a saját osztályaink és a szabványos elvont alaposztályok közötti kapcsolat kialakításának módját, az elvont alaposztályok újabb felhasználási lehetőségével ismerkedünk meg: a felületről adunk előzetes képet saját készítésű osztályaink számára. Három meglehetősen különböző példát tekintünk át, az elvont alaposztályok létrehozásának és használatának három oldalát megismerve.

Kezdetnek vegyünk egy igen egyszerű példát, amely az írható-olvasható tulajdonságok kezelését mutatja be. Az osztály háztartási készülékek ábrázolására használható. Minden létrehozott készüléknek rendelkeznie kell egy csak olvasható karakterlánc-tulajdonsággal (a gyártmány jelölésére), illetve egy írható-olvasható árral. Azt is biztosítani szeretnénk,

hogy az elvont alaposztály `__init__()` tagfüggvényét újra meg kelljen valósítani. Íme az `Appliance.py` fájlban lévő elvont alaposztály. Az `abstractmethod()` és az `abstractproperty()` tagfüggvény használatához szükséges `import abc` utasítás nem látható – a függvények díszítőként használatosak.

```
class Appliance(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def __init__(self, model, price):
        self.__model = model
        self.price = price
    def get_price(self):
        return self.__price
    def set_price(self, price):
        self.__price = price
    price = abc.abstractproperty(get_price, set_price)

    @property
    def model(self):
        return self.__model
```

8.4. táblázat *A collections modul főbb elvont alaposztályai*

Elvont alaposztály	Alaposztálya	Felület	Példák
`Callable`	`object`	`()`	Valamennyi függvény, tagfüggvény és a `lambda`-függvények
`Container`	`object`	`in`	`bytearray`, `bytes`, `dict`, `frozenset`, `list`, `set`, `str`, `tuple`
`Hashable`	`object`	`hash()`	`bytes`, `frozenset`, `str`, `tuple`
`Iterable`	`object`	`iter()`	`bytearray`, `bytes`, `collections.deque`, `dict`, `frozenset`, `list`, `set`, `str`, `tuple`
`Iterator`	`Iterable`	`iter()`, `next()`	
`Sized`	`object`	`len()`	`bytearray`, `bytes`, `collections.deque`, `dict`, `frozenset`, `list`, `set`, `str`, `tuple`
`Mapping`	`Container`, `Iterable`, `Sized`	`==`, `!=`, `[]`, `len()`, `iter()`, `in`, `get()`, `items()`, `keys()`, `values()`	`dict`

8.4. táblázat *A collections modul főbb elvont alaposztályai* *(folytatás)*

Elvont alaposztály	Alaposztálya	Felület	Példák
`MutableMapping`	`Mapping`	`==, !=, [], del, len(), iter(), in, clear(), get(), items(), keys(), pop(), popitem(), setdefault(), update(), values()`	`dict`
`Sequence`	`Container, Iterable, Sized`	`[], len(), iter(), reversed(), in, count(), index()`	`bytearray, bytes, list, str, tuple`
`MutableSequence`	`Container, Iterable, Sized`	`[], +=, del, len(), iter(), reversed(), in, append(), count(), extend(), index(), insert(), pop(), remove(), reverse()`	`bytearray, list`
`Set`	`Container, Iterable, Sized`	`<, <=, ==, !=, =>, >, &, |, ^, len(), iter(), in, isdisjoint()`	`frozenset, set`
`MutableSet`	`Set`	`<, <=, ==, !=, =>, >, &, |, ^, &=, |=, ^=, -=, len(), iter(), in, add(), clear(), discard(), isdisjoint(), pop(), remove()`	`set`

A fenti kóddal az osztály metaosztályává az `abc.ABCMeta` osztályt tettük – azért, mert az elvont alaposztályok esetében ez kívánalom; természetesen az `abc.ABCMeta` osztály bármely leszármazottja megfelelt volna. Az `__init__()` tagfüggvényből elvont tagfüggvényt csináltunk, ezzel téve szükségessé az újbóli megvalósítását. Elkészítettünk egy olyan megvalósítást, amelyet a várakozásaink szerint a leszármazottak hívni fognak – ezt azonban nem tudjuk kötelezővé tenni. Elvont írható-olvasható osztályok létrehozásakor nem használhatjuk a díszítő utasításformát, és a kiolvasó-, illetve beállítófüggvények sem kaptak privát nevet: ha így tettünk volna, az az alosztályoknál okozna kényelmetlenséget. A `model` tulajdonság sem elvont, így az alosztályokban nem kell újból megvalósítani. Az `Appliance` osztály nem példányosítható, ugyanis az osztályban elvont jellemzőket találunk. Íme egy példaként szolgáló alosztály:

```
class Cooker(Appliance):
    def __init__(self, model, price, fuel):
        super().__init__(model, price)
        self.fuel = fuel
        price = property(lambda self: super().price,
                         lambda self, price: super().set_price(price))
```

A `Cooker` osztályban újra meg kell valósítanunk az `__init__()` tagfüggvényt és a `price` tulajdonságot. Ami a tulajdonságot illeti, minden munkát átadtunk az alaposztálynak. A `model` csak olvasható tulajdonságot örökölte az osztály. Az `Appliance` („készülék") osztály alapján sok egyéb osztály létrehozható, mondjuk a `Fridge` („hűtő"), a `Toaster` („pirító"), és így tovább.

A következőként áttekintett elvont alaposztály még rövidebb. A `TextFilter.py` fájlban lévő, szöveg szűrésére használható elvont alaposztályról van szó:

```
class TextFilter(metaclass=abc.ABCMeta):

    @abc.abstractproperty
    def is_transformer(self):
        raise NotImplementedError()

    @abc.abstractmethod
    def __call__(self):
        raise NotImplementedError()
```

A `TextFilter` elvont alaposztály semmiféle működésre nem képes, létének egyetlen oka, hogy egy felületet ad meg – az `is_transformer` csak olvasható tulajdonságot és a `__call__()` tagfüggvényt, amelyeket minden leszármazott osztályban újra meg kell valósítani. Minthogy az elvont tulajdonság és az elvont tagfüggvény megvalósítása egyaránt hiányzik, nem szeretnénk, ha az alosztályok meghívnák őket – egy ártalmatlanul megbúvó `pass` utasítás helyett kivételt váltunk ki, ha valami használná őket (például a `super()` függvény meghívásával). Íme egy egyszerű alosztály:

```
class CharCounter(TextFilter):

    @property
    def is_transformer(self):
        return False

    def __call__(self, text, chars):
        count = 0
        for c in text:
            if c in chars:
                count += 1
        return count
```

Ez a szövegszűrő nem végez átalakítást – az átadott szövegben a megadott karaktereket számolja meg – vagyis az `is_transformer()` tagfüggvény `False` értéket ad vissza. Íme a használatát bemutató példa:

```
vowel_counter = CharCounter()
vowel_counter("dog fish and cat fish", "aeiou") # visszatérési érték: 5
```

Két másik szövegszűrő osztályt is megadunk – a `RunLengthEncode` (futamhossz-kódoló) és a `RunLengthDecode` (futamhossz-visszafejtő) osztályokat –; mindkettő átalakítást végez. Így használjuk őket:

```
rle_encoder = RunLengthEncode()
rle_text = rle_encoder(text)
...
rle_decoder = RunLengthDecode()
original_text = rle_decoder(rle_text)
```

A futamhossz-kódoló egy karakterláncot alakít UTF-8 kódolású bájtsorozattá; a `0x00` bájtokat a `0x00`, `0x01`, `0x00` sorozatra cseréli, illetve bármely, 3-255 karakter hosszúságú ismétlődő bájtot a `0x00`, *darabszám*, *bájt* sorozatra cserél. Ha a karakterláncban sok a legalább négy egymást követő azonos bájtból álló „futam”, az eljárással a sima UTF-8 kódolású karakterláncnál rövidebbet kapunk. A futamhossz-visszafejtő paramétere egy az előzőekben ismertetett eljárással kódolt karakterlánc, visszatérési értéke pedig az eredeti karakterlánc. Íme a `RunLengthDecode` osztály eleje:

```
class RunLengthDecode(TextFilter):
    @property
    def is_transformer(self):
    return True
def __call__(self, rle_bytes):
    ...
```

A `__call__()` tagfüggvény belső részét elhagytuk, de a könyv mellékleteként érkező forráskódok között megtalálható. A `RunLengthDecode` osztály szerkezete pontosan ugyanilyen.

Az utolsó áttekintett elvont alaposztály egy visszavonási művelet alkalmazásprogramozási felületét (Application Programming Interface, API) adja meg, az alapértelmezett megvalósítással együtt. Íme az `Abstract.py` fájlban lévő teljes elvont alaposztály:

```
class Undo(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def __init__(self):
        self.__undos = []
    @abc.abstractproperty
    def can_undo(self):
        return bool(self.__undos)
    @abc.abstractmethod
    def undo(self):
        assert self.__undos, "nothing left to undo"
        self.__undos.pop()(self)
    def add_undo(self, undo):
        self.__undos.append(undo)
```

Minthogy az `__init__()` és az `undo()` tagfüggvények egyaránt elvontak, újbóli megvalósításuk kötelező. Ugyanez áll a csak olvasható `can_undo` tulajdonságra is. Az alosztályokban nem kötelező az `add_undo()` tagfüggvény újbóli megvalósítása, bár nem is tiltja semmi.

Az `undo()` tagfüggvény rejt némi érdekességet magában. A `self.__undos` listának tagfüggvényekre mutató objektumhivatkozásokat kell tárolnia. Minden tagfüggvénynek az a feladata, hogy a meghívásakor visszavonja a megfelelő műveletet – ez remélhetőleg jobb megvilágításba kerül, ha áttekintettük az `Undo` alosztályt. A visszavonás úgy végezhető el, hogy a `self.__undos` lista végéről a `pop()` tagfüggvénnyel kivesszük az utolsó visszavonást végző tagfüggvényt, amelyet aztán függvényként meghívunk. A függvény paramétere a `self`. (Azért van szükség a `self` átadására, mert függvényként és nem tagfüggvényként hívjuk meg a tagfüggvényt.)

Lássuk a `Stack` osztály elejét! Az osztály az `Undo` osztály leszármazottja, így a rajta végzett műveletek a `Stack.undo()` tagfüggvény paraméterek nélküli meghívásával visszavonhatók.

```
class Stack(Undo):
    def __init__(self):
        super().__init__()
        self.__stack = []

    @property
    def can_undo(self):
        return super().can_undo

    def undo(self):
        super().undo()

    def push(self, item):
        self.__stack.append(item)
        self.add_undo(lambda self: self.__stack.pop())

    def pop(self):
        item = self.__stack.pop()
        self.add_undo(lambda self: self.__stack.append(item))
        return item
```

A `Stack.top()` és a `Stack.__str__()` tagfüggvényt elhagytuk, mivel egyikük sem szolgál semmiféle újdonsággal, és az `Undo` alaposztállyal sincsenek kapcsolatban. Ami a `can_undo` tulajdonságot, illetve az `undo()` tagfüggvényt illeti, a munkát egyszerűen átadjuk az alaposztálynak. Ha a tulajdonság és a tagfüggvény nem lenne elvont, nem kellene újra megvalósítani őket, a hatásuk pedig ugyanaz maradna. Mi azonban ez esetben kényszeríteni akartuk a leszármazott osztályokat az újbóli megvalósításra, hogy a visszavonás lehetőségét mindenképpen figyelembe vegyék. A `push()` és a `pop()` tagfüggvények esetében elvégezzük, majd a visszavonható műveletek között elhelyezzük a műveletet – így válik a frissen elvégzett művelet visszavonhatóvá.

Az elvont alaposztályok a nagy méretű programok, könyvtárak és alkalmazás-keretrendszerek esetében a leghasznosabbak – segítségükkel megoldható, hogy bárki is a program írója, bármilyen is a megvalósítás, a program egyes részei képesek együttműködni egymással, mert az elvont alaposztályaik ugyanazt az alkalmazásprogramozási felületet tartalmazzák.

Többszörös öröklés

Többszörös öröklésnek azokat az eseteket tekintjük, amikor egy osztály legalább két osztálytól örököl. Bár a Python (és például a C++) tökéletesen támogatja a többszörös öröklést, vannak nyelvek – a legfigyelemreméltóbb köztük a Java –, amelyek nem engedik meg a használatát. A többszörös örökléssel kapcsolatos egyik probléma, hogy kialakulhat az a helyzet, amikor egy osztályból is többszörösen örököl az osztályunk (például két alaposztálya is ugyanannak az osztálynak a leszármazottja). Mindez annyit tesz, hogy ha olyan tagfüggvényt hívunk meg, amely nem az osztályban, hanem annak legalább két alaposztályában szerepel (vagy akár az ő alaposztályaikban, és így tovább), akkor az, hogy a tagfüggvénynek pontosan melyik változatát hívjuk, a tagfüggvények megtalálásának sorrendjétől függ – így viszont a többszörös öröklést használó osztályaink némileg törékennyé válnak.

A többszörös öröklés általában elkerülhető, ha egyszeres öröklés (egyetlen alaposztály) használata mellett egy metaosztály megadásával oldjuk meg a további alkalmazásprogramozási felületek támogatását – ugyanis, ahogy az a következőkből kiderül, a metaosztályokkal anélkül adhatunk meg felületeket, hogy egyetlen tagfüggvény, illetve adatjellemző tényleges öröklése megvalósulna. Dönthetünk úgy is, hogy a többszörös öröklést egyetlen konkrét osztállyal valósítjuk meg, és a további felületekhez akár több elvont alaposztályt használunk. További választási lehetőség, hogy egyszeres öröklés mellett objektum-összetétellel dolgozunk.

Mindazonáltal vannak olyan esetek, amelyekben a többszörös öröklés jelenti a kényelmes megoldást. Tételezzük fel például, hogy a korábbi `Stack` osztály új változatát szeretnénk létrehozni, de úgy, hogy az osztály támogassa a befőttek betöltését és mentését. Könnyen elképzelhető, hogy a befőttek kezelésének képességét számos osztályban meg szeretnénk valósítani, így saját, önálló osztályban valósítjuk meg.

```
class LoadSave:

    def __init__(self, filename, *attribute_names):
        self.filename = filename
    self.__attribute_names = []
    for name in attribute_names:
        if name.startswith("__"):
            name = "_" + self.__class__.__name__ + name
        self.__attribute_names.append(name)

def save(self):
    with open(self.filename, "wb") as fh:
        data = []
        for name in self.__attribute_names:
            data.append(getattr(self, name))
        pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)
```

```
    def load(self):
        with open(self.filename, "rb") as fh:
            data = pickle.load(fh)
            for name, value in zip(self.__attribute_names, data):
                setattr(self, name, value)
```

Az osztálynak két jellemzője van: az egyik a `filename`, amely nyilvános, és bármikor megváltoztatható, a másik az `__attribute_names`, amely rögzített, és csak a példány létrejöttekor kaphat értéket. A `save()` tagfüggvény minden jellemzőnevet bejár, és azt a `data` nevű listát hozza létre, amely minden mentendő jellemző értékét tárolja. Ennek végeztével a lista adatai a befőttbe kerülnek. A `with` utasítás biztosítja, hogy amennyiben sikeresen megnyitottuk a fájlt, akkor be is zárjuk, illetve azt, hogy bármely, a fájllal, illetve a befőttel kapcsolatos kivétel eljusson a hívóhoz. A `load()` tagfüggvény a jellemzők neveit és a megfelelő betöltött adatelemeket járja be, és minden jellemző értékéül a betöltött értéket adja meg.

Az alábbiakban ismertetjük a `FileStack` osztályt, amely az előző részben megismert `Undo` és az itt szereplő `LoadSave` osztálytól egyaránt örököl.

```
class FileStack(Undo, LoadSave):

    def __init__(self, filename):
        Undo.__init__(self)
        LoadSave.__init__(self, filename, "__stack")
        self.__stack = []

    def load(self):
        super().load()
        self.clear()
```

Az osztály további része pontosan megegyezik a `Stack` osztállyal, így az itteni közlésétől eltekintettünk. Az `__init__()` tagfüggvényben nem használjuk a `super()` függvényt; helyette – minthogy a `super()` függvény nem képes a szándékaink kitalálására – nekünk kell megadnunk az alaposztályokat. Ami a `LoadSave` osztály előkészítését illeti, átadjuk a használandó fájlnevet, csakúgy, mint azoknak a jellemzőknek a nevét, amelyeket menteni szeretnénk – ebben az esetben csak a `__stack` nevű privát jellemzőről van szó. (Az `__undos` jellemzőt nem akarjuk menteni, sőt ebben az esetben nem is tudnánk, mert egy tagfüggvénylistát tárolunk benne, így ez a lista nem alakítható befőtté.)

A `FileStack` osztály rendelkezik az `Undo` osztály összes tagfüggvényével, valamint a `LoadSave` osztály `save()` és `load()` tagfüggvényeivel is. A `save()` tagfüggvény újbóli megvalósításától eltekintettünk, hiszen a mostani is remekül működik. Ami azonban a `load()` tagfüggvényt illeti, betöltés után gondoskodnunk kell a visszavonásokat tároló változó kitakarításáról. Ez azért szükséges, mert esetleg egy mentés után teszünk egyet-mást, amit akár betöltés is követhet. A betöltés eredményeképpen törlődik minden, ami addig történt, ezen műveletek visszavonásának tehát a továbbiakban nincs értelme. Az eredeti `Undo` osztály nem rendelkezik `clear()` tagfüggvénnyel, vagyis azt most kell létrehozni:

```
    def clear(self):                    # Az Undo osztályban
        self.__undos = []
```

A `Stack.load()` tagfüggvényben a `LoadSave.load()` tagfüggvény hívására azért használtuk a `super()` függvényt, mert nem létezik olyan `Undo.load()` tagfüggvény, amely kétséget okozhatna. Ha mindkét alaposztályban lenne `load()` tagfüggvény, hogy melyiket hívnánk, azon múlna, hogy a Python milyen sorrendben találja meg őket. A `super()` függvény használatát igyekszünk azokra az esetekre korlátozni, amikor nem merülhet fel kétség. Más esetekben megadjuk a megfelelő alaposztály nevét is, és soha nem hagyatkozunk a megtalálási sorrendre. Ami a `self.clear()` hívást illeti, megint csak nem merül fel kétség, hiszen `clear()` tagfüggvény csak az `Undo` osztályban található. A `super()` függvény használatára azért nincs szükség, mert – a `load()` tagfüggvénnyel ellentétben – a `Filestack` osztály nem rendelkezik `clear()` tagfüggvénnyel.

Mi történik, ha a későbbiekben a `FileStack` osztály is kiegészül egy `clear()` nevű tagfüggvénnyel? Nos, ez tönkretenné a `load()` tagfüggvény működését. A probléma egyik megoldását az jelentheti, ha a `load()` tagfüggvény belsejében a sima `self.clear()` hívás helyett a `super().clear()` hívással dolgoznánk. Ennek eredménye az lenne, hogy az elsőként megtalált alaposztály `clear()` tagfüggvényét hívnánk meg. Az efféle problémák elkerülése végett érdemes alapelvként elfogadnunk, hogy a többszörös öröklésű osztályok esetében az alaposztályok nevét beledrótozzuk az osztályunkba – a fenti példában az `Undo.clear(self)` hívást használjuk. Illetve megpróbálkozhatunk azzal, hogy egyáltalán nem alkalmazzuk a többszörös öröklést, hanem összetétellel (aggregation) oldjuk meg az ilyen eseteket – például az `Undo` osztályból öröklünk, a `LoadSave` osztályt pedig már a létrehozásakor felkészítjük az összetétel alkalmazására.

A többszörös öröklés eredményeként két meglehetősen különböző osztály keverékét hoztuk létre, anélkül, hogy akár a visszavonást, akár a betöltést, akár a mentést meg kellett volna valósítanunk – helyette az alaposztályok által biztosított képességekre hagyatkoztunk. Ez igen kényelmes lehet, és különösen akkor használható remekül, amikor a szülőosztályok felületében nincs átfedés.

Metaosztályok

A metaosztály úgy viszonyul az osztályhoz, mint az osztály a példányhoz, vagyis a metaosztályokat osztályok létrehozására használjuk, pontosan úgy, ahogy az osztályokat használjuk példányok létrehozására. Ahogy az `isinstance()` függvénnyel kideríthető, hogy egy példány egy adott osztály leszármazottja-e, az `issubclass()` függvény megmondja, hogy egy osztály – `dict`, `int`, `SortedList` – leszármazottja-e egy másik osztálynak.

A metaosztályok használatának legegyszerűbb esete, amikor velük illesztjük az egyéni osztályainkat a Python elvont alaposztályainak szabványos hierarchiájába. Például ha azt akarjuk, hogy a `SortedList` osztály a `collections` osztály alá tartozzon, nem öröklést állítunk be – ezt korábban már bemutattuk –, hanem egyszerűen bejegyezzük a `SortedList` osztályt a `collections.Sequence` osztály alá. Így:

```
class SortedList:
    ...
collections.Sequence.register(SortedList)
```

Az osztály szokásos megadását követően bejegyezzük az osztályt a `collections.Sequence` osztály alá. A bejegyzés az osztályt *virtuális alosztállyá** teszi. A virtuális alosztályok az `isinstance()`, illetve az `issubclass()` függvény kérdésére válaszolva az adott alaposztály alosztályának mondják magukat, de sem a tagfüggvényeket, sem az adatokat nem öröklik.

Az osztályok efféle bejegyzése elvárássá teszi, hogy az osztályunk támogassa annak az osztálynak a felületét, amelyik alá bejegyeztük – biztosítékot azonban nem nyújt minderre. A metaosztályok használatának egyik oka, hogy az elvárás mellett a biztosíték is megjelenjék. A másik használati lehetőség, hogy egy osztályt módosítunk a segítségével – éppúgy, mint az osztályszintű díszítőkkel. Természetesen egy metaosztály egyidejűleg szolgálhatja mindkét cél elérését.

Tegyük fel, hogy osztályok egy olyan csoportját szeretnénk létrehozni, ahol minden osztály rendelkezik `load()` és `save()` tagfüggvényekkel. Ezt úgy tehetjük meg, hogy létrehozunk egy osztályt, amely, ha metaosztályként használjuk, ellenőrzi, hogy az említett tagfüggvények jelen vannak-e.

```
class LoadableSaveable(type):
    def __init__(cls, classname, bases, dictionary):
        super().__init__(classname, bases, dictionary)
        assert hasattr(cls, "load") and \
               isinstance(getattr(cls, "load"),
                          collections.Callable), ("class '" +
               classname + "' must provide a load() method")
        assert hasattr(cls, "save") and \
               isinstance(getattr(cls, "save"),
                          collections.Callable), ("class '" +
               classname + "' must provide a save() method")
```

Azoknak az osztályoknak, amelyeket metaosztályként használunk, mindenképpen a hierarchia tetején álló `type` metaosztálytól vagy annak egy alosztályától kell származniuk. Figyeljük meg, hogy a fent megadott osztály meghívására akkor kerül sor, amikor az őt használó *osztályokat* példányosítjuk, tehát minden valószínűség szerint nem túl gyakran, vagyis a futásidejű költség meglehetősen alacsony. Figyeljük meg azt is, hogy az ellenőrzéseket az osztály létrehozása után kell foganatosítanunk (a `super()` függvényt meghívva), mivel az osztály jellemzői magán az osztályon belül csak ekkor válnak elérhetővé. (A jellemzők benne vannak a szótárban, de jobban szeretjük az ellenőrzéseket magán az előkészített osztályon elvégezni.)

---

* A Python szóhasználatában a *virtuális* nem ugyanazt jelenti, mint a C++ esetében.

A `hasattr()` függvénnyel is ellenőrizhettük volna, hogy a `load` és a `save` jellemzők hívhatók-e – vagyis van-e `__call__` jellemzőjük –, de inkább azt néztük meg, hogy a `collections.Callable` példányairól van-e szó. A `collections.Callable` elvont osztály megkívánja – de nem biztosítja – hogy az alosztályai, illetve a virtuális alosztályai hívhatók legyenek.

Ha az osztály – a `type.__new__()` tagfüggvény vagy egy újbóli megvalósítása segítségével – létrejött, az `__init__()` tagfüggvényének meghívásával a metaosztályt is előkészítettük. Az `__init__()` tagfüggvény paraméterei az éppen létrehozott osztály; az osztály neve (ami az *`osztály.__név__`* tulajdonságból is kiolvasható), az osztály alaposztályainak listája (kivéve az `object` osztályt, így akár üres is lehet), és a szótár, amely azokat a jellemzőket tárolja, amelyek akkor váltak osztályszintű jellemzőkké, amikor az osztály létrejött, hacsak a metaosztály `__new__()` tagfüggvényének egy esetleges újbóli megvalósításakor másképp nem rendelkeztünk.

Nézzünk pár interaktív példa annak szemléltetésére, hogy mi történik, ha a `LoadableSaveable` metaosztály segítségével hozunk létre osztályokat:

```
>>> class Bad(metaclass=Meta.LoadableSaveable):
...     def some_method(self): pass
...
Traceback (most recent call last):
  ...
AssertionError: class 'Bad' must provide a load() method
```

A metaosztályban megadottak miatt az őt használó osztályoknak bizonyos tagfüggvényeket meg kell valósítaniuk, ha pedig nem tennék – ahogy most sem –, akkor `AssertionError` kivétel lép fel.

```
>>> class Good(metaclass=Meta.LoadableSaveable):
...     def load(self): pass
...     def save(self): pass
>>> g = Good()
```

A `Good` osztály eleget tesz a metaosztály felületre vonatkozó kívánalmainak, bár a bennünk kialakult, a viselkedésével kapcsolatos elvárásoknak nem felel meg.

A metaosztályok az őket használó osztályok megváltoztatására is használhatók. Ha a változtatás az éppen létrehozott osztály nevét, az alaposztályát, vagy a szótárát (a `__slots__` -ra gondolunk) érinti, akkor a metaosztály `__new__()` tagfüggvényét kell újból megvalósítanunk, de más változtatások esetében – új tagfüggvénnyel vagy adatjellemzővel történő kiegészítés – elegendő az `__init__()` tagfüggvény újbóli megvalósítása – bár ez a `__new__()` tagfüggvényben is elvégezhető. A következőkben egy olyan metaosztályt tekintünk át, amely az őt használó osztályok módosítását pusztán a `__new__()` tagfüggvényével végzi.

A `@property` és a `@name.setter` díszítők használatán felüli lehetőségként létrehozhatunk olyan osztályokat is, ahol egyszerű elnevezési irányelveket léptetünk életbe a tulajdonságok azonosítására. Például ha az osztály rendelkezik `get_`*`név`*`()` és `set_`*`név`*`()` tagfüggvénnyel, akkor várhatóan találunk egy privát `__`*`név`* nevű tulajdonságot az osztályban, amelyet kiolvasáskor és beállításkor a *`példány`*`.`*`név`* alakban érünk el. Mindez megoldható egy metaosztály használatával. Íme egy a fenti irányelveket követő példaosztály:

```
class Product(metaclass=AutoSlotProperties):
    def __init__(self, barcode, description):
        self.__barcode = barcode
        self.description = description

    def get_barcode(self):
        return self.__barcode

    def get_description(self):
        return self.__description

    def set_description(self, description):
        if description is None or len(description) < 3:
            self.__description = "<Invalid Description>"
        else:
            self.__description = description
```

A `__barcode` („vonalkód”) privát tulajdonságnak az előkészítés során kell értéket adnunk, mivel nem tartozik hozzá beállítófüggvény – a dolog másik folyománya, hogy így a tulajdonság csak olvashatóvá válik. A `description` viszont írható-olvasható tulajdonság. Íme néhány interaktív példa:

```
>>> product = Product("101110110", "8mm Stapler")
>>> product.barcode, product.description
('101110110', '8mm Stapler')
>>> product.description = "8mm Stapler (long)"
>>> product.barcode, product.description
('101110110', '8mm Stapler (long)')
```

Ha a `__barcode` tulajdonságnak próbálunk értéket adni, `AttributeError` kivétel lép fel, és megjelenik a „can't set attribute” (a jellemző értékét nem lehet beállítani) hibaüzenet.

Ha egy pillantást vetünk a `Product` osztály jellemzőire – például a `dir()` függvénnyel –, a nyilvános jellemzők közül csak a `barcode` és a `description` nevűeket találjuk meg. A `get_`*`név`*`()` és `set_`*`név`*`()` tagfüggvény már nincs ott – a *`név`* tulajdonság került a helyükre. A vonalkódot és a leírást tároló változók szintén privát változók (`__barcode` és `__description`), és memóriarekeszként kaptak helyet, hogy csökkenjen az osztály memóriaigénye. Mindezt az `AutoSlotProperties` metaosztály végzi el, amely egyetlen tagfüggvényből áll:

```
class AutoSlotProperties(type):
    def __new__(mcl, classname, bases, dictionary):
        slots = list(dictionary.get("__slots__", []))
        for getter_name in [key for key in dictionary
                            if key.startswith("get_")]:
            if isinstance(dictionary[getter_name],
                          collections.Callable):
                name = getter_name[4:]
                slots.append("__" + name)
                getter = dictionary.pop(getter_name)
                setter_name = "set_" + name
                setter = dictionary.get(setter_name, None)
                if (setter is not None and
                    isinstance(setter, collections.Callable)):
                    del dictionary[setter_name]
                dictionary[name] = property(getter, setter)
        dictionary["__slots__"] = tuple(slots)
        return super().__new__(mcl, classname, bases, dictionary)
```

A metaosztályok `__new__()` osztályszintű tagfüggvényének a metaosztály és a létrehozandó osztály neve, alaposztályai és szótára a paraméterei. Azért kell az `__init__()` tagfüggvény helyett a `__new__()` tagfüggvény újbóli megvalósítását használnunk, mert az osztály létrehozása előtt szeretnénk megváltoztatni a szótárat.

Először a `__slots__` gyűjteményt másoljuk át, ha pedig nem rendelkezünk ilyennel, létrehozunk egy üreset. Figyelünk, hogy lista legyen, nem pedig `tuple`, különben nem tudnánk módosítani. A szótárban lévő jellemzők közül kiválasztjuk azokat, amelyeknek a neve „`get_`”-tel kezdődik és hívhatók, vagyis a kiolvasófüggvényeket. A `__slots__` gyűjteménybe a megfelelő adatok tárolására minden kiolvasófüggvényhez egy-egy privát név kerül, a `get_`*`név`*`()` tagfüggvény esetében például a `__`*`név`*. Ennek végeztével vesszük a kiolvasófüggvényre mutató hivatkozást, és az eredeti nevére vonatkozó bejegyzést töröljük a szótárból (ezt egy menetben végezzük, a `pop()` függvénnyel). Ha van beállítófüggvény, akkor ugyanezt tesszük vele is, majd új elemet helyezünk el a szótárban. A kulcs a kívánt tulajdonságnév lesz, például ha a kiolvasófüggvény neve `get_`*`név`*`()`, akkor a tulajdonság neve *név*. Az elem értékéül egy tulajdonságot adunk meg, a szótárban fellelt és onnan eltávolított kiolvasófüggvénnyel és beállítófüggvénnyel (ami a `None` objektum is lehet).

Amikor ezzel is megvagyunk, az eredeti `__slots__` gyűjteményt lecseréljük a módosítottra, amelyben minden elhelyezett tulajdonság privát helyet kap, majd folytatjuk a hívást az alaposztálynál, amely elvégzi az osztály tényleges létrehozását, de már a módosított szótárral. Gondoljuk arra, hogy ez esetben a `super()` függvény meghívásakor pontosan meg kell adnunk a metaosztályt – a `__new__()` tagfüggvény hívásakor mindig ez a helyzet, ugyanis nem példányszintű, hanem osztályszintű tagfüggvényről van szó.

A fenti példában azért nem írtunk `__init__()` tagfüggvényt, mert minden munkát a `__new__()` tagfüggvényben végeztünk el. Mindazonáltal semmi sem gátolja mind a `__new__()`, mind az `__init__()` tagfüggvény újbóli megvalósítását, külön-külön feladatot adva nekik.

Ha párhuzamot vonunk az összetétel és az öröklés, illetve a kézifúrók, valamint a díszítők és a leírók, illetve az elektromos ütvefúrók között, akkor a metaosztályok valahol a lézersugárral egyenértékűek – ha a használhatóságot és sokoldalúságot tekintjük. A metaosztályok használata nem az első, amihez a fejlesztő fordul, hanem sokkal inkább az utolsó – kivéve azt az esetet, amikor alkalmazás-keretrendszerek fejlesztéséről van szó, és nem akarjuk, hogy a felhasználóink szenvedjenek, amíg felismerik a kínált előnyöket.

Függvényközpontú programozás

A függvényközpontú programozás a programozás olyan megközelítése, amelyben a számítások alapegységei olyan függvények, amelyek nem módosítják a saját paramétereiket, nem hivatkoznak a program állapotjelzőire, és nem is változtatják meg azokat, az eredményüket pedig visszatérési érték formájában adják meg. Az efféle programozási megközelítés mellett szóló érvek közül az egyik legerősebb az, hogy – legalábbis elméletben – sokkal egyszerűbb elszigetelt függvényeket fejleszteni, és így az ilyen programokban a hibakeresés is könnyebbé válik. Ezt az is elősegíti, hogy az ilyen programoknak nincsenek állapotváltozásaik, tehát a működésük matematikai eszközökkel is kifejezhető.

A függvényközpontú programozáshoz szorosan kapcsolódik a *megfeleltetés* (mapping), a *szűrés* (filtering) és az *elhagyás* (reducing) fogalma. A megfeleltetés lényege, hogy egy adott függvénnyel és egy bejárható típusú objektummal (iterable) olyan új bejárható objektumot (vagy listát) állítunk elő, ahol az egyes elemeket az eredeti bejárható objektum megfelelő elemeit a függvénynek átadva visszatérési értékként kapjuk meg. Ennek kivitelezésére a beépített `map()` függvény szolgál. Például:

```
list(map(lambda x: x ** 2, [1, 2, 3, 4]))    # visszatérési érték:
                                                [1, 4, 9, 16]
```

A `map()` függvény paramétere egy függvény és egy bejárható objektum, a visszatérési értéke pedig a hatékonyság okán nem lista, hanem szintén egy bejárható objektum. Ezúttal az eredmény érthetőbbé tétele miatt listát kértünk:

```
[x ** 2 for x in [1, 2, 3, 4]]   # visszatérési érték: [1, 4, 9, 16]
```

A `map()` függvény helyén gyakran használhatunk generátorkifejezést. Ezúttal a `list()` használatának elkerülése végett listaértelmezést használtunk – a kifejezés úgy alakítható generátorrá, hogy a külső szögletes zárójelek helyett rendes zárójelet használunk.

A szűrés lényege, hogy egy függvény és egy bejárható objektum segítségével új bejárható objektumot állítunk elő. A régi objektum elemei akkor kerülnek át az újba, ha a függvény `True` értékkel tér vissza. A `filter()` beépített függvény ezt valósítja meg:

```
list(filter(lambda x: x > 0, [1, -2, 3, -4]))     # visszatérési érték:
                                                  [1, 3]
```

A `filter()` függvény paramétere egy függvény és egy bejárható objektum, a visszatérési értéke pedig egy másik bejárható objektum.

```
[x for x in [1, -2, 3, -4] if x > 0]     # visszatérési érték: [1, 3]
```

A `filter()` függvény mindig helyettesíthető egy generátorkifejezéssel vagy egy listaértelmezéssel.

Az elhagyás lényege, hogy egy függvény és egy bejárható objektum segítségével egyetlen visszatérési értéket állítunk elő. A visszatérési érték előállítása úgy történik, hogy a függvényt az első két értékkel paraméterezve hívjuk meg, majd a paraméterek az előző hívás visszatérési értéke és a harmadik érték lesznek, és így tovább, amíg valamennyi érték sorra nem került. A `functools` modul `functools.reduce()` függvénye ezt valósítja meg. Íme két kódsor, amelyek ugyanazt a számítást végzik el:

```
functools.reduce(lambda x, y: x * y, [1, 2, 3, 4])  # visszatérési
                                                    # érték: 24
functools.reduce(operator.mul, [1, 2, 3, 4])  # visszatérési érték: 24
```

Az `operator` modulnak minden, a Python nyelvben használatos művelethez van megfelelő függvénye, mégpedig elsősorban a függvényközpontú programozás megkönnyítése végett. A fenti példa második sorában ahelyett, hogy az első sorhoz hasonlóan a `lambda` utasítással hoztuk volna létre a szorzás műveletét, inkább az `operator.mul()` függvényt használtuk.

A Python beépített elhagyófüggvényekkel is rendelkezik. A bejárható objektummal paraméterezett `all()` függvény akkor ad vissza `True` értéket, ha a `bool()` függvényt az objektum egyes elemeivel meghívva a visszatérési érték mindig `True`. Az `any()` függvény visszatérési értéke `True`, ha bármelyik elem kiértékelése `True` értéket ad. A `max()` függvény a bejárható objektum legnagyobb elemét adja vissza, a `min()` függvény pedig a legkisebbet. A `sum()` függvény a bejárható objektum elemeinek összegével tér vissza.

Az alapelvek tisztázása után tekintsünk meg néhány további példát. Először több módszerrel is kiszámoljuk a `files` listában lévő fájlok együttes méretét.

```
functools.reduce(operator.add, (os.path.getsize(x) for x in files))
functools.reduce(operator.add, map(os.path.getsize, files))
```

A `map()` függvény használata gyakran eredményez a listaértelmezésnél, illetve a generátorkifejezésnél rövidebb megoldást, kivéve, ha feltétel is van. A hozzáadás függvénye ezúttal nem a `lambda x, y: x + y`, hanem az `operator.add()` volt.

Ha csak a `.py` kiterjesztésű fájlok méretére vagyunk kíváncsiak, a más kiterjesztésű fájlokat kiszűrhetjük. A feladat az alábbi három módszerrel oldható meg:

```
functools.reduce(operator.add, map(os.path.getsize,
                 .filter(lambda x: x.endswith(".py"), files)))
functools.reduce(operator.add, map(os.path.getsize,
                (x for x in files if x.endswith(".py"))))
functools.reduce(operator.add, (os.path.getsize(x)
                for x in files if x.endswith(".py")))
```

Könnyen belátható, hogy a második és harmadik módszer jobb az elsőnél, ugyanis ezekben nem kell megírnunk a lambdafüggvényt. A generátorkifejezés (illetve a listaértelmezés), valamint a `map()` és a `filter()` függvény használata közötti döntés azonban a legtöbb esetben pusztán személyes megfontolásokon alapul.

A `map()`, a `filter()` és a `functools.reduce()` függvény használata gyakran oda vezet, hogy szükségtelenné válnak a ciklusok – a fenti példák is ezt szemléltetik. Ezek a függvények olyankor hasznosak, amikor függvényközpontú nyelven írt programokat alakítunk át, de a Python nyelvben a `map()` függvény helyett rendszerint alkalmazhatunk listaértelmezést, a `filter()` függvény helyett pedig feltételt tartalmazó listaértelmezést.

A `functools.reduce()` függvény pedig sok esetben kiküszöbölhető, ha a Python beépített függvényei közül az `all()`, az `any()`, a `max()`, a `min()` vagy a `sum()` valamelyikét használjuk. Például:

```
sum(os.path.getsize(x) for x in files if x.endswith(".py"))
```

A fenti kifejezés eredménye megegyezik az előző háromével, de ez lényegesen tömörebb.

`operator.attrgetter()` függvény
375 ☜

Az `operator` modul, azon felül, hogy a Python műveleteinek megfelelő függvényeket találunk benne, az `operator.attrgetter()` és az `operator.itemgetter()` függvényeket is tartalmazza. Az előzőről a fejezet korábbi részében futólag már szóltunk. Mindkét függvény olyan függvényeket ad vissza, amelyeket a megadott jellemzők, illetve elemek kinyerésére használhatunk.

Amíg szeleteléssel egy lista egy része kapható meg, szeleteléssel együtt alkalmazott egyeléssel pedig részek egy sorozata (például az `L[::3]` kifejezéssel a lista minden harmadik eleme), addig az `operator.itemgetter()` függvénnyel egy lista tetszőleges elemeit kaphatjuk meg (például: `operator.itemgetter(4, 5, 6, 11, 18)(L)`). Az `operator.itemgetter()` függvénytől visszakapott függvényt nem kötelező azonnal meghívni, sem pedig elvetni, mint ahogy az előző példában tettük – ha megtartjuk, paraméterül adható a `map()`, a `filter()` vagy a `functools.reduce()` függvények bármelyikének, használhatjuk szótárban, listában, vagy halmazértelmezésben.

Ha adatok rendezése a cél, megadhatunk egy kulcsfüggvényt. Ez a függvény bármilyen függvény lehet: lambdafüggvény, beépített függvény, tagfüggvény (például az `str.lower()`), vagy az `operator.attrgetter()` függvény kapott függvény. Ha például feltételezzük, hogy az `L` listában `priority` jellemzővel bíró objektumok vannak, a listát rendezhetjük a prioritás szerint, mégpedig így: `L.sort(key=operator.attrgetter("priority"))`.

A már említett `functools` és `operator` modulokon felül a függvényközpontú programozás során az `itertools` modul is jó szolgálatot tehet. Több lista bejárása például nem csak a listák egymáshoz fűzésével lehetséges, hanem az `itertools.chain()` függvény használatával is:

```
for value in itertools.chain(data_list1, data_list2, data_list3):
    total += value
```

Az `itertools.chain()` függvény visszatérési értéke egy olyan bejárható objektum, amelyben először az első lista egymásra következő értékei vannak, aztán a másodiké, és így tovább, amíg az összes lista összes értéke sorra nem kerül. Az `itertools` modulban sok egyéb függvény található, a dokumentációja pedig számos rövid, ámde hasznos és elolvasásra érdemes példával szolgál.

Részleges függvények alkalmazása

A részleges függvények (parciális függvények) alkalmazása azt jelenti, hogy egy már létező függvényből és annak pár paraméteréből új függvényt alakítunk ki. Az új függvény ugyanazt teszi, mint az eredeti, de néhány paraméter rögzítettsége mellett – amelyeket így a hívónak nem kell átadnia. Íme egy nagyon egyszerű példa:

```
enumerate1 = functools.partial(enumerate, start=1)
for lino, line in enumerate1(lines):
    process_line(i, line)
```

Az első sorban `enumerate1()` néven új függvényt hozunk létre, amely az átadott `enumerate()` függvény burkolója. A megadott kulcsszóparaméter (a `start=1`) miatt az `enumerate1()` az eredeti függvényt – esetünkben a `lines` objektumot – egy rögzített paraméterrel hívja meg; és természetesen átadja a híváskor kapott további paramétereket. A fenti példában az `enumerate1()` függvényt arra használjuk, hogy a sorok számozását a hagyományos módon, 1-től kezdve végezzük el.

A részleges függvények segítségével egyszerűsíthetjük a kódot, különösen akkor, amikor újra meg újra ugyanazokkal a paraméterekkel kell meghívnunk egy függvényt. Például amikor UTF-8 kódolású fájl feldolgozására van szükség, ahelyett, hogy minden alkalommal megadnánk a kódolásra, illetve a megnyitásra vonatkozó paramétereket, az említett paraméterek rögzítésével létrehozhatunk egy függvénypárt:

```
reader = functools.partial(open, mode="rt", encoding="utf8")
writer = functools.partial(open, mode="wt", encoding="utf8")
```

Innentől kezdve a `reader(`*fájlnév*`)` hívással olvasásra, a `writer(`*fájlnév*`)` hívással pedig írásra nyithatunk meg egy fájlt.

A részleges függvényeket a grafikus felhasználói felületek programozásában (a témakört a 17. fejezet tárgyalja) igen elterjedten használják. Gyakran kényelmes, ha pár gomb közül egyet lenyomva ugyanazt a függvényt hívhatjuk meg. Például:

```
loadButton = tkinter.Button(frame, text="Load",
                            command=functools.partial(doAction, "load"))
saveButton = tkinter.Button(frame, text="Save",
                            command=functools.partial(doAction, "save"))
```

A fenti példa azt a `tkinter` nevű, grafikus felhasználói felület programozására alkalmas könyvtárat használja, amely alapértelmezés szerint együtt települ a Pythonnal.
A `tkinter.Button` osztállyal gombok hozhatók létre – ezúttal kettőt készítettünk, mindkettőt egyazon kereten belül. Mindkettőre ráírtuk, hogy mire való. Mindkét gomb `command` jellemzőjének értéke az a függvény, amelyet a gomb lenyomásakor a `tkinter` objektumnak meg kell hívnia – esetünkben a `doAction()` függvényről van szó. Részleges függvény használatával biztosítottuk, hogy a `doAction()` függvénynek átadott első paraméter egy olyan karakterlánc legyen, amely arról tájékoztatja a függvényt, hogy melyik gombot nyomtuk le. Így a `doAction()` függvény el tudja dönteni, hogy mit kell tennie.

Példa: Valid.py

Leírók
☞ 379

Ebben a részben a leírókat az osztályszintű díszítőkkel együttesen arra fogjuk használni, hogy egy érvényesített jellemzők létrehozására szolgáló, jól használható eljárást alakítsunk ki.

Osztályszintű díszítők
☞ 384

Egészen eddig, ha biztosak akartunk benne lenni, hogy egy jellemző érvényes értéket kap, csak a tulajdonságokra, vagy a beállító és kiolvasó tagfüggvények használatára támaszkodhattunk. Az efféle megközelítés hátránya abban rejlik, hogy érvényességvizsgáló kódot kell elhelyeznünk minden olyan osztályban és minden olyan jellemzőnél, ahol ennek szükségét érezzük.
Kényelmesebb és egyszerűbben módosítható eljárás lenne, ha az osztályok olyan jellemzőket kapnának, amelyekbe a szükséges ellenőrzőkód be van építve. Íme a használni kívánt utasításforma példája:

```
@valid_string("name", empty_allowed=False)
@valid_string("productid", empty_allowed=False,
              regex=re.compile(r"[A-Z]{3}\d{4}"))
@valid_string("category", empty_allowed=False, acceptable=
          frozenset(["Consumables", "Hardware", "Software", "Media"]))
```

```
@valid_number("price", minimum=0, maximum=1e6)
@valid_number("quantity", minimum=1, maximum=1000)
class StockItem:

    def __init__(self, name, productid, category, price, quantity):
        self.name = name
        self.productid = productid
        self.category = category
        self.price = price
        self.quantity = quantity
```

A `StockItem` osztály minden jellemzőjét érvényesítettük. A `productid` jellemző például csak olyan karakterláncot kaphat értékül, ami nem üres, három nagybetűvel kezdődik, és négy számra végződik, a `category` jellemző pedig csak olyat, ami nem üres, és a megadott értékek valamelyikével megegyezik. A `quantity` jellemző egy szám, amely nem kisebb egynél, és nem nagyobb ezernél. Ha érvénytelen érték beállításával próbálkozunk, kivétel lép fel.

Szabályos kifejezések ☞ 463

Osztályszintű díszítők 384 ☞

Az érvényesítést osztályszintű díszítők és leírók együttes használatával oldottuk meg. Ahogy korábban már jeleztük, az osztályszintű díszítők egyetlen paramétert kaphatnak – a díszítendő osztályt. Így aztán ezúttal is ahhoz a módszerhez folyamodtunk, amelyet az osztályszintű díszítőkkel együtt ismertünk meg. A `valid_string()` és a `valid_number()` függvény az általunk megadott összes paramétert elfogadja, majd egy díszítőt ad vissza, amely viszont egy osztályt vár paraméterül, és az osztály módosított változatával tér vissza.

Az alábbiakban a `valid_string()` függvény működését tekintjük át.

```
def valid_string(attr_name, empty_allowed=True, regex=None,
                 acceptable=None):
    def decorator(cls):
        name = "__" + attr_name
        def getter(self):
            return getattr(self, name)
        def setter(self, value):
            assert isinstance(value, str), (attr_name +
                              " must be a string")
            if not empty_allowed and not value:
                raise ValueError("{0} may not be empty".format(
                                 attr_name))
            if ((acceptable is not None and value not in acceptable) or
                (regex is not None and not regex.match(value))):
                raise ValueError("{0} cannot be set to {1}".format(
                                 attr_name, value))
            setattr(self, name, value)
        setattr(cls, attr_name, GenericDescriptor(getter, setter))
    return cls
return decorator
```

A függvény először egy osztályszintű díszítőfüggvényt hoz létre, amelynek egyetlen paramétere egy osztály. A díszítő a dísztett osztályt két jellemzővel egészíti ki: egy privát adatjellemzővel és egy leíróval. Például ha a `valid_string()` függvényt a „productid" névvel hívjuk meg, a `StockItem` osztályba bekerül a termékazonosítót tároló `__productid` jellemző és a leíró `productid` jellemző, amely az érték elérésére szolgál. Ha például az `item = StockItem("TV", "TVA4312", "Electrical", 500, 1)` utasítással hozunk létre egy elemet, a termékazonosító az `item.productid` utasítással olvasható ki, a beállítására pedig mondjuk az `item.productid = "TVB2100"` utasítás használható.

A díszítő által létrehozott kiolvasófüggvény a privát adatjellemző értékének visszaadására egyszerűen a globális `getattr()` függvényt használja. A beállítófüggvény elvégzi az érvényességvizsgálatot, majd a privát adatjellemző értékéül az új – és érvényes – értéket a `setattr()` függvénnyel állítja be. Valójában a privát adatjellemző csak akkor jön létre, amikor értéket kap.

Amikor a kiolvasó- és beállítófüggvény elkészült, ismét a `setattr()` függvényt használjuk, ezúttal azért, hogy a megadott név (például a `productid`) alapján új, osztályszintű jellemzőt hozzunk létre, amelynek az értéke egy `GenericDescriptor` típusú leíró lesz. Ha ez is megvan, a díszítő a módosított osztályt adja vissza, a `valid_string()` függvény pedig a díszítőfüggvényt.

Felépítését tekintve a `valid_number()` és a `valid_string()` függvény egyforma; az egyetlen különbség a paraméterezésükben és a beállítófüggvény érvényesítést végző kódjában rejlik – így az ismertetésétől ezúttal eltekintünk. (A teljes forráskód megtalálható a `Valid.py` modulban.)

Az utolsó dolog, amit tárgyalnunk kell, a `GenericDescriptor` típus, ami történetesen a legegyszerűbb rész:

```
class GenericDescriptor:
    def __init__(self, getter, setter):
        self.getter = getter
        self.setter = setter
    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.getter(instance)
    def __set__(self, instance, value):
        return self.setter(instance, value)
```

A leírót az egyes jellemzők kiolvasó- és beállítófüggvényeinek tárolására használjuk. A kiolvasással és a beállítással kapcsolatos munkát egyszerűen átadja a megfelelő függvényeknek.

Összefoglalás

A fejezet során lényegesen bővültek a Python eljárásközpontú és objektumközpontú programozásáról szerzett ismereteink, és belekóstoltunk a függvényközpontú programozásba is. Az első részben megismertük a generátorkifejezések létrehozásának módját, és mélyebben tárgyaltuk a generátorkifejezéseket. Azt is megtanultuk, hogy miként lehet dinamikusan modulokat betölteni, miként használhatjuk az ilyen modulok képességeit, és miként történik a dinamikus kódvégrehajtás. Megismertük a `nonlocal` utasítás használatát, és az önhívó függvényeket, és azt is megtanultuk, hogy miként írhatjuk meg saját függvény- és tagfüggvénydíszítőinket, illetve miként írhatunk és használhatunk jegyzetelt függvényeket.

A fejezet második részében az objektumközpontú programozás néhány bonyolultabb területét tanulmányoztuk. Először a jellemzők elérését, illetve a `__getattr__()` különleges szerepű tagfüggvény használatát ismertük meg jobban, majd a funktorokról tanultunk, valamint arról, hogy a funktorok miként használhatók a függvények állapottal való kiegészítésére – ez olyasmi, amit úgy is megoldhatunk, hogy a függvényeket tulajdonságokkal ruházzuk fel, vagy záradékokat használunk. A fejezetben ezekről is volt szó. Megtanultuk, hogy a `with` utasítás miként használatos a környezetkezelők vonatkozásában, és azt is, hogy mi a módja az egyéni környezetkezelők létrehozásának. Minthogy a Python fájlobjektumai egyben környezetkezelők is, a fájlkezelést mostantól a `try with...except` szerkezettel végezzük, biztosítva, hogy a megnyitott fájlokat a `finally` blokkok használata nélkül is lezárjuk.

A második részt további, az objektumközpontú programozás elmélyült ismeretét igénylő képességek ismertetésével folytattuk. Először a leírókkal ismerkedtünk meg, amelyek sokféle módon használhatók, és amelyek a Python alapvető díszítőinek – többek között a `@property` és a `@classmethod` díszítőkre gondolunk – használhatóságát nagyban emelik. Megismertük az egyéni leírók létrehozásának módját, és három, meglehetősen különböző példán keresztül barátkoztunk meg a használatukkal. Ezt követően az osztályszintű díszítőket tanulmányoztuk, és láttuk, hogy miként módosítható velük egy osztály szinte ugyanúgy, ahogy a függvénydíszítők módosítanak egy függvényt.

A szakasz utolsó három részében a Python elvont alaposztályairól, a többszörös öröklésről és a metaosztályokról tanultunk. Megtudtuk, hogy a saját osztályaink miként egyezethetők össze a Python elvont alaposztályaival, és azt is, hogy miként hozhatók létre egyéni elvont alaposztályok. Azt is láttuk, hogy a többszörös örökléssel különböző osztályok képességei egyetlen osztályban egyesíthetők. A metaosztályok tárgyalása során fény derült arra is, hogy egy osztály – nem egy objektumpéldány – létrejöttekor és előkészítésekor is elvégeztethetünk a programunkkal egyet-mást.

Az utolsó előtti részben néhány olyan függvényt és modult ismertünk meg, amelyekkel a Python a függvényközpontú programozást segíti. Megtanultuk a megfeleltetés, a szűrés és az elhagyás, vagyis a függvényközpontú programozás három legközönségesebb műveletének használatát. A részleges függvények létrehozásának módját is megismertük.

Végül az utolsó részben megtanultunk olyan eljárást kialakítani, amely – az osztályszintű díszítők és a leírók együttes használatával – jól használható és rugalmas megoldást kínál a jellemzőértékek érvényességének ellenőrzésére.

A fejezettel lezárult magának a Python nyelvnek a tárgyalása. A mostani, illetve az előző fejezetben nem minden képességével ismerkedtünk meg a nyelvnek, de csak a homályos és ritkán használt területeket nem érintettük. A következő fejezetek egyike sem ismertet nyelvi újdonságokat, bár az alapkönyvtár moduljai közül mindegyikben találkozunk eleddig ismeretlennel, és néhányban továbblépünk a már megismert módszerek használatát illetőleg is. Mindezen felül a következő fejezetekben bemutatásra váró programoknál nem alkalmazzuk az előzőekben használt megszorításokat – eddig ugyanis vigyáztunk, hogy csak a már megismert nyelvi eszközökkel dolgozzunk fel egy problémát –, vagyis ezekben a fejezetekben találjuk meg a nyelv sajátságait leginkább kiaknázó példákat.

Gyakorlatok

Az alábbi három gyakorlat egyike sem jár hosszú kódrészletek megírásával – és egyikükről sem állítjuk, hogy egyszerű.

1. Készítsünk másolatot a `magic-numbers.py` programról, és töröljük belőle a `get_function()` függvényeit, valamint egy kivételével a `load_modules()` függvényeit is. Vegyünk fel egy új `GetFunction` funktorosztályt, amely két gyorsítótárral bír. Az egyikben a megtalált, a másikban a fellelhetetlen függvényeket tároljuk, így kerülve el, hogy egy olyan modulban keressünk újra meg újra egy adott függvényt, amelyikben az nincs meg. A `main()` függvényben mindössze annyi módosítást kell elvégeznünk, hogy a ciklus előtt elhelyezünk egy `get_function = GetFunction()` utasítást, és egy `with` utasítást használunk a `finally` blokk helyett. Azt is ellenőrizzük, hogy a modul függvényei meghívhatók-e, mégpedig a `hasattr()` függvény helyett a `collections.Callable` osztállyal. Az osztály nagyjából húsz sorban megírható. A megoldást a `magic-numbers_ans.py` fájlban találjuk.
2. Hozzunk létre egy új modult, és adjunk meg benne három függvényt. Az `is_ascii()` függvény `True` értékkel tér vissza, ha a neki átadott karakterlánc minden karakterének kódja 127-nél kisebb. Az `is_ascii_punctuation()` függvény akkor tér vissza `True` értékkel, ha valamennyi karakter megtalálható a `string.punctuation` karakterláncban. Az `is_ascii_printable()` függvény visszatérési értéke pedig akkor `True`, ha minden karakter benne van

a `string.printable` karakterláncban. Szerkezetét tekintve az utolsó két függvény egyforma. Minden függvényt a `lambda` utasítás használatával hozzunk létre; ez függvényközpontú megközelítéssel két sorban megoldható. Mindenképp tartozzon mindegyikhez docstring és doctest. A dokumentációs teszteket a modul futassa is le. A függvények egyenként három-öt sor hosszúak, így a modul egésze – a dokumentációs tesztekkel együtt – a huszonöt sort sem éri el. A megoldást az `Ascii.py` fájlban találjuk.

3. Hozzunk létre egy új modult, és határozzuk meg benne az `Atomic` környezetkezelő osztályt. Az osztály működése mindenben egyezzen meg a fejezetben bemutatott `AtomicList` osztályéval, de nem csak listával, hanem bármilyen változtatható gyűjteményes adattípussal működnie kell. A tároló megfelelőségét az `__init__()` tagfüggvénynek kell ellenőriznie, és nem azt kell jeleznie, hogy sekély vagy mély másolás szükséges, hanem ennek fényében a `self.copy` jellemzőnek kell megfelelő függvényt értékül adnia. A másolást végző függvényt az `__enter__()` tagfüggvényben hívjuk meg. Az `__exit__()` is bővül, mivel a halmazok és a szótárak tartalmának cseréje nem úgy történik, mint a listáké – és értékadást sem használhatunk, mert az az eredeti tárolón nem változtatna. Az osztály nagyjából harminc sorban megírható, bár dokumentációs tesztekre is szükség lenne. A megoldást – dokumentációs tesztekkel együtt – a nagyjából százötven soros `Atomic.py` fájlban találjuk.

9

Folyamatok és szálak

- Feladatok végrehajtása folyamatokkal
- Feladatok végrehajtása szálakkal

Amióta a többmagos processzorok nemhogy kivételt nem jelentenek, de egyenesen alapvetőnek számítanak, a programozót még inkább kísérti az az immáron gyakorlati érvekkel is könnyebben alátámasztható igény, hogy az adatok feldolgozásának terhét megossza, a többmagos gépeknél így érve el a lehető legnagyobb teljesítményt. A munkateher megosztásának alapvetően két módja kínálkozik. Az egyik a több folyamat, a másik a több szál használata. A fejezet mindkét megközelítési mód használatát bemutatja.

A több folyamat használatának, azaz különálló programok futtatásának megvan az az előnye, hogy minden folyamat egymástól függetlenül fut. Így az erőforrásokért való versengés kezelésével járó minden terhet az operációs rendszerre hagyunk. A hátrányos oldal, hogy a kommunikáció és az adatmegosztás könnyen válik kényelmetlenné a főprogram, illetve az általa létrehozott különálló folyamat között. Unix rendszereken az exec-fork megosztás megoldást jelenthet, de a rendszerfüggetlen programok esetében más megoldást kell találnunk. A legegyszerűbb és az itt bemutatott egyetlen eljárás az, ha a főprogram látja el az általa létrehozott folyamatokat adatokkal, de a folyamatok a kimenetet önállóan állítják elő. Rugalmasabb, a kétirányú kommunikációt lényegesen leegyszerűsítő megközelítés a hálózat használata. Természetesen számos esetben nincs szükség efféle kommunikációra, mindössze arra, hogy egy-két másik programot futtassunk a folyamatok vezénylését végző főprogram segítségével.

Ha nem akarjuk a munkát független folyamatok között elosztani, készíthetünk olyan, többszálú programot, amely a feladatokat több önálló végrehajtású szál között osztja szét. Ennek az eljárásnak az az előnye, hogy a kommunikáció pusztán az adatok megosztásával is megvalósul – feltételezve, hogy az adathoz egy adott pillanatban egyetlen szál fér hozzá –, de az erőforrások párhuzamos használatának kezelésével járó teher a programozó nyakában marad. A Python remekül támogatja a többszálú programok készítését – az elvégzendő munka a lehető legkevesebb. A többszálú programok mindazonáltal lényegükből fakadóan összetettebbek egyszálú társaiknál, és mind a program kialakításánál, mind a karbantartás során lényegesen nagyobb odafigyelést kívánnak meg.

A fejezet első részében két rövid programot fogunk megírni. Az első programot a felhasználó indítja el, a másodikat pedig az első program. A második program annyiszor indul el, ahányszor szükség van erre. A második rész elején megismerkedünk a többszálú programok írásának alapjaival, ezt követően pedig megírunk egy olyan többszálú programot, amely egyenértékű az előzőleg megírt két program együttesével. Tesszük mindezt úgy, hogy rávilágítunk a több folyamat és a több szál használata közötti főbb különbségekre. Végezetül áttekintjük egy másik, az előzőnél kifinomultabb többszálú program működését, amely kiosztja az elvégzendő feladatokat, és az eredményeket is összegyűjti.

Feladatok végrehajtása folyamatokkal

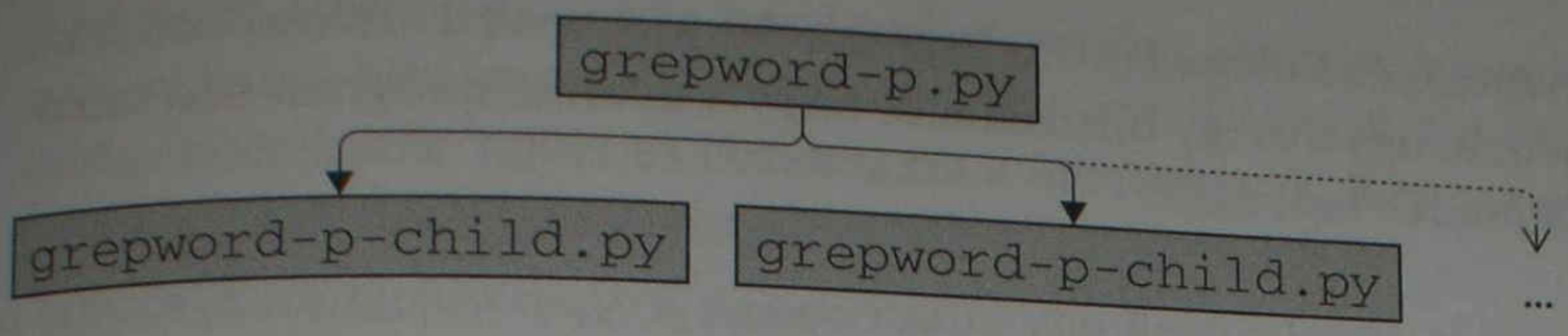
Vannak helyzetek, amikor rendelkezésünkre állnak az elvégzendő feladatot ellátni képes programok, de automatizálni szeretnénk a futtatásukat. Ez megoldható a Python `subprocess` moduljával, amelyben megtaláljuk a más programok futtatásához és a megfelelőnek talált parancssori paraméterek átadásához szükséges eszközöket, sőt azokat is, amelyek csővezetékek használatával biztosítják a folyamatok közötti kommunikációt. Már láttunk egy igen egyszerű példát, mégpedig az 5. fejezetben, amikor a `subprocess.call()` függvényt használtuk a konzolablak operációs rendszertől függő letakarítására. Ugyanezek az eszközök használhatók „szülő-gyermek" programpárok megalkotására is. Ilyen esetekben a felhasználó futtatja a szülőprogramot, amely annyi példányban indít különböző feladatokat ellátó gyermekprogramokat, ahányszor azt szükségesnek ítéli. A fenti megközelítési lehetőséget tárgyaljuk ebben a részben.

A 3. fejezetben bemutattuk a `grepword.py` nevű, igen egyszerű programot, amely a parancssorba írt szót keresi meg a parancssorban a szó után álló fájlokban. Ebben a részben a program egy kifinomultabb változatát írjuk meg, amely alkönyvtárakban is képes a beolvasandó fájlok megtalálására, és a munkát tetszőleges számú önálló gyermekfolyamat között képes szétosztani. A kimenet egy elérési útvonalakat is tartalmazó fájlnévlista, a megadott szót tartalmazó fájlok neveivel.

A szülőprogram neve `grepword-p.py`, a gyermekprogramé pedig `grepword-p-child.py`. A programok futása alatt fennálló kapcsolatokat a 9.1. ábra szemlélteti.

A `grepword-p.py` program szíve a `main()` függvényben rejlik, amelyet három részben tekintünk át:

```
def main():
    child = os.path.join(os.path.dirname(__file__),
                         "grepword-p-child.py")
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    files_per_process = len(filelist) // opts.count
    start, end = 0, files_per_process + (len(filelist) % opts.count)
    number = 1
```



9.1. ábra

Szülőprogram és gyermekprogramok

A munkát a gyermekprogram nevének kiolvasásával kezdjük, majd a felhasználó által megadott parancssori paraméterek feldolgozása következik. A `parse_options()` függvény az `optparse` modult használja; visszatérési értéke pedig az `opts` nevű nevesített rendezett sorozat, amely jelzi, hogy a programnak vizsgálnia kell-e az alkönyvtárakat, és tudatja a használandó folyamatok számát – hét az alapértelmezett, és a program legfeljebb húszat indíthat. A visszaadott értékben megtaláljuk a keresett szót, illetve a parancssorban megadott könyvtár- és fájlnevek listáját is. A `get_files()` függvény visszatérési értéke a beolvasandó fájlok listája.

Ha megvan minden, a feladat elvégzéséhez szükséges információ, kiszámítjuk, hogy az egyes folyamatoknak hány fájlt adjunk ki feldolgozásra. A `start` és az `end` változó jelzi a `filelist` listában annak a résznek az elejét és végét, amelyet a következő gyerekfolyamatnak adunk oda. A fájlok száma rendszerint nem lesz pontos többszöröse a folyamatok számának, a maradékot tehát az első folyamat kapja. A `number` változó egyetlen szerepe a hibakeresés segítése; ennek segítségével láthatjuk, hogy a kimenet egyes sorai melyik folyamattól származnak.

```
pipes = []
while start < len(filelist):
    command = [sys.executable, child]
    if opts.debug:
        command.append(str(number))
    pipe = subprocess.Popen(command, stdin=subprocess.PIPE)
    pipes.append(pipe)
    pipe.stdin.write(word.encode("utf8") + b"\n")
    for filename in filelist[start:end]:
        pipe.stdin.write(filename.encode("utf8") + b"\n")
    pipe.stdin.close()
    number += 1
    start, end = end, end + files_per_process
```

A `filelist` minden `start:end` közötti részéhez létrehozunk egy parancslistát, amely a Python parancsértelmezőből (a `sys.executable` változóban találjuk), abból a gyermekprogramból, amelyet a Pythonnal akarunk elindíttatni és a parancssori paraméterekből – esetünkben, hibakereséskor csak a gyermekfolyamatok számából – áll. Ha a gyermekprogram megfelelő shebang-sorral vagy fájltársítással rendelkezik, lehetne ő az első a listában, és

nem kellene a Python-értelmező megadásával bajlódni. Mindazonáltal szívesebben közelítjük meg így a problémát, ugyanis így biztosítható, hogy a gyermekprogram és a szülő ugyanazt a Python-értelmezőt használja.

Amikor elkészült a parancs, létrehozunk egy `subprocess.Popen` objektumot, ezúttal úgy, hogy tudjunk írni a folyamat szabványos bemenetére. (Hasonló kulcsszó-paraméter megadásával a folyamat szabványos kimenetének olvasására is van lehetőség.) Ezután leírjuk a keresett szót, majd egy újsor karakter, végül a fájllista megfelelő részében lévő fájlnevek következnek. A `subprocess` modul karakterláncok helyett bájtokat ír és olvas, vagyis a leírt karakterláncokat kódolnunk kell (a beolvasott bájtokat pedig visszafejtenünk), mégpedig megfelelő kódolás használatával. Ezúttal az UTF-8 kódolást választottuk. Ha a gyermekfolyamatnak átadtuk a fájlok listáját, lezárjuk a hozzá tartozó szabványos bemenetet, és továbblépünk.

Nem feltétlenül kötelező minden folyamat objektumhivatkozását megtartanunk – a `pipe` változó a ciklus minden ismétlődésekor az új `subprocess.Popen` osztályú objektumra hivatkozik –, hiszen mindegyik önállóan fut. Mégis mindegyiket elhelyezzük egy listában, így ugyanis módunkban áll megszakítani egy-egy folyamatot. Az eredményeket sem gyűjtjük be, inkább minden folyamattal a megfelelő időben kiíratjuk az eredményt a folyamat saját konzoljára. Ez azzal jár, hogy a különböző folyamatok kimenete átfedésben lehet. (A gyakorlatoknál megismerjük az átfedések elkerülésének módját.)

```
while pipes:
    pipe = pipes.pop()
    pipe.wait()
```

Amikor minden folyamat elindult, megvárjuk, amíg az egyes folyamatok befejeződnek. Ez nem feltétlenül szükséges, de a Unix-szerű rendszereken így lehetünk biztosak abban, hogy az összes folyamat lefutása után visszakapjuk a konzol parancsjelét – egyéb esetben meg kell nyomnunk az ENTER billentyűt a folyamatok lefutása után. A várakozás további előnye, hogy ha megszakítjuk a főprogram futását – mondjuk a CTRL+C billentyűk lenyomásával –, azok a folyamatok is megszakadnak, amelyek ekkor még futnak. A leállás egy el nem kapott `KeyboardInterrupt` kivételt eredményez. Ha nem várakoznánk, a főprogram befejeződne – vagyis nem lenne megszakítható –, a gyermekfolyamatok pedig tovább futnának, egészen addig, amíg a `kill` programmal vagy a feladatkezelővel véget nem vetnénk a futásuknak.

A megjegyzések és a modulok betöltését végző sorok kivételével következzék a teljes `grepword-p-child.py` program, két részletben.

```
BLOCK_SIZE = 8000

number = "{0}: ".format(sys.argv[1]) if len(sys.argv) == 2 else ""
word = sys.stdin.readline().rstrip()
```

A program először a number karakterlánc értékét állítja be a megfelelő számra, illetve ha nem végzünk hibakeresést, üres karakterláncot ad értékül. Ezt követően az első sor beolvasása következik, mivel ez tartalmazza a keresett szót. Ez a sor és az ezt követők karakterláncként lesznek beolvasva.

```
for filename in sys.stdin:
    filename = filename.rstrip()
    previous = ""
    try:
        with open(filename, "rb") as fh:
            while True:
                current = fh.read(BLOCK_SIZE)
                if not current:
                    break
                current = current.decode("utf8", "ignore")
                if (word in current or
                    word in previous[-len(word):] +
                            current[:len(word)]):
                    print("{0}{1}".format(number, filename))
                    break
                if len(current) != BLOCK_SIZE:
                    break
                previous = current
    except EnvironmentError as err:
        print("{0}{1}".format(number, err))
```

Az első sort a fájlnevek követik, az elérési utakkal együtt. Minden sornál megnyitjuk a megfelelő fájlt, beolvassuk, és ha tartalmazza a keresett szót, kiírjuk a nevét. Lehetséges, hogy pár fájl igen méretes, ami problémákhoz is vezethet, főleg ha húsz gyermekfolyamat fut egyszerre, és mindegyik nagy fájlok beolvasását végzi. A problémán úgy igyekszünk úrrá lenni, hogy a fájlok beolvasását blokkokban végezzük, az előzőleg beolvasott blokkot megtartva – így vigyázunk arra, hogy az olyan esetek se sikkadjanak el, amikor a szó egyetlen előfordulása blokkhatárra esik. A blokkonkénti beolvasás további előnye, hogy amennyiben a keresett szót már a fájl eleje felé megtaláljuk, nem kell végigolvasnunk a fájlt – hiszen csak annyi érdekel bennünket, hogy a szó benne van-e az adott állományban, vagy sem.

Karakterkódolás 86 ☜

A fájlok beolvasása bináris módban történik, vagyis mielőtt elolvasnánk, minden blokkot karakterlánccá kell alakítanunk – a keresett szó ugyanis karakterlánc típusú. Feltételezzük, hogy minden állomány az UTF-8 karakterkódolást használja, bár minden valószínűség szerint lesznek ettől eltérő esetek. Egy kifinomultabb program megpróbálkozna a használt karakterkódolás meghatározásával, majd bezárná, és újra megnyitná a fájlt, ezúttal már a helyes karakterkódolást használva. Ahogyan a 2. fejezetben már említettük, a Python Package Index webhelyén (`pypi.python.org/pypi`) legalább két olyan Python-csomag található, amely a fájlok karakterkódolásának automatikus megállapítását célozza. (Kísértést érezhetünk arra, hogy

a keresett szót `bytes` típusú objektummá alakítsuk, majd bájtsorozatok között végezzük el az összehasonlítást, de ez nem megbízható módszer, vannak ugyanis olyan karakterek, amelyeknek több érvényes UTF-8 kódolású ábrázolásuk van.)

A `subprocess` modul lényegesen több képességgel rendelkezik annál, mint amit itt felhasználtunk. Képes például a parancshéj fordított aposztrófjának (`) és csővezetékének, illetve az `os.system()` függvénynek megfelelő működés ellátására, valamint függvények előállítására.

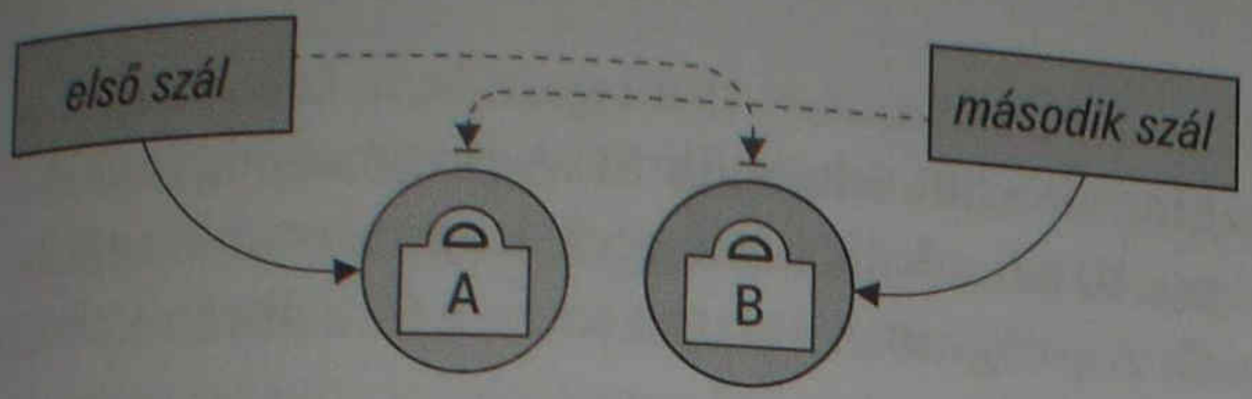
A következő részben megismerkedünk a `grepword-p.py` program többszálú változatával, és lesz módunk összehasonlítani a szülő-gyermek folyamatokon alapulóval. Áttekintjük egy olyan, kifinomultabb többszálú program működését is, amelyik kiosztja a feladatokat, majd összeszedi az eredményeket, így szabályozva még pontosabban a kimenetet.

Feladatok végrehajtása szálakkal

A Pythonban meglehetősen egyszerű egy kettő vagy akár több végrehajtási szálat tartalmazó program írása. A dolog onnantól válik bonyolulttá, ha azt is szeretnénk, hogy az egyes szálak osztozzanak az adatokon. Képzeljük el, hogy két szál osztozik egy listán. Az egyik szál a lista bejárásához fog a `for x in L` ciklussal, és amikor a közepe felé jár, egy másik szál töröl pár elemet a listából. Ez legjobb esetben is a program gyanús elhalálozásához vezet – a legrosszabb eset pedig az, hogy hamis eredményt kapunk.

Az egyik legáltalánosabban elterjedt megoldás valamiféle zárolási rendszer használata. Például, az első szál zárolja a listát, és csak *ez után* kezdi el bejárni. Ilyenkor a zárolás miatt egyetlen más szál sem fér a listához. A valóságban a helyzet messze nem ennyire egyszerű. Az adat és az azt zároló zár közötti kapcsolat ugyanis kizárólag a mi képzeletünk szülötte. Ha egy szál zárol valamit, és egy másik szál megpróbálja ugyanazt a dolgot zárolni, a második szál egészen addig kénytelen várni, amíg az első szál meg nem szünteti a zárolást. Ha az adatok hozzáférhetőségének szabályozásakor a zárolási eljárást is figyelembe vesszük, biztosítható, hogy a megosztott adathoz egy adott időben csak egy adott szál férjen hozzá – még akkor is, ha a védelem csak közvetett.

A zárolással kapcsolatos a holtpont kialakulásának kockázata. Tegyük fel, hogy az *első szál* egy a megosztott adathoz való hozzáférés okán létrehozza az *A* zárat, és amikor az létrejön, megpróbál a *b* megosztott adathoz is hozzáférni. A *B* zár létrehozására azonban képtelen, ugyanis azt időközben egy *második szál* már létrehozta, hogy hozzáférhessen a *b* megosztott adathoz. A második szál most éppen az *A* zár létrehozásával próbálkozik, hogy hozzáférjen az a megosztott adathoz. Vagyis az *első szál* birtokolja az *A* zárat, és próbálkozik a *B* zár kialakításával, miközben a második szál birtokolja a *B* zárat, és próbálkozik az *A* zár kialakításával. Végül mindkét szál várakozásra kényszerül, a program pedig holtpontra jut – ahogy a 9.2. ábra is mutatja.



9.2. ábra

Holtpont: legalább két szál várakozik arra, hogy a másik által zárolt adathoz hozzáférhessen

Bár a holtpont lerajzolása és elképzelése egyszerű, a gyakorlatban a holtpontok komoly gondokat okozhatnak – ugyanis a létük nem mindig nyilvánvaló. Néhány többszálúságot kezelő programkönyvtár figyelmeztetni próbálja a programozót a lehetséges holtpontokra, de a holtpontok elkerülése emberi figyelmet és beavatkozást igényel.

A holtpontok elkerülésének egyik egyszerű, ámde hatékony módja, ha felállítunk egy rendet, amely megadja a zárolások sorrendjét. Például, ha a rend azt mondja ki, hogy az *A* zárolásnak mindig meg kell előznie a *B* zárolást, és éppen meg akarjuk valósítani a *B* zárolást, a rend miatt előbb az *A* zárolást kell megvalósítanunk. Ezzel a módszerrel biztosítható, hogy a fent leírt holtpont sohasem alakul ki, mivel mindkét szál az *A* zárolást akarja elvégezni, és amelyiknek sikerül, csak az léphet tovább a *B* zároláshoz. Mindez remekül működik, amíg valaki el nem felejtkezik a rend betartásáról.

A zárolás további problémája, hogy ha több szál várakozik egy zárolásnál, akkor csak állnak, és semmi hasznosat nem végeznek. Ennek hatása jelentősen csökkenthető, ha a kódolási szokásainkat úgy változtatjuk meg, hogy csak minimális munkát végeztessünk a programmal, miközben valamit zárolva tartunk.

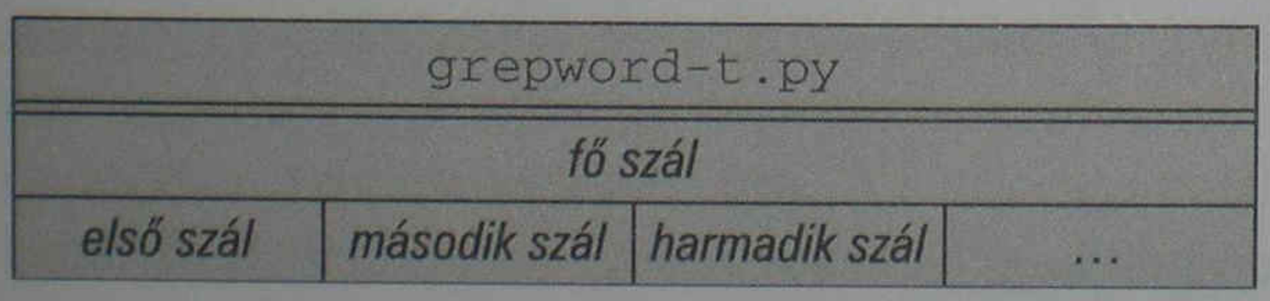
Minden Python-programban van legalább egy szál, a fő szál. További szálak létrehozása előtt be kell töltenünk a `threading` modult, amellyel aztán annyi szálat hozhatunk létre, amennyit csak akarunk. Szálak létrehozására két mód kínálkozik. Meghívhatjuk a `threading.Thread()` tagfüggvényt, valamilyen hívható objektumot adva át neki, illetve alosztályt alakíthatunk ki a `threading.Thread` osztály alapján – a fejezetben mindkét megközelítést bemutatjuk. Az alosztály létrehozása a rugalmasabb és egyszerűbb módszer. Az alosztályok lecserélhetik az `__init()__` tagfüggvényt (ebben az esetben az alaposztály-beli megvalósítást *kell* meghívniuk), és le *kell* cserélniük a `run()` tagfüggvényt – ez a tagfüggvény felelős a szál munkájának elvégzéséért. A kódunk *soha* nem hívhatja meg a `run()` tagfüggvényt – a szál indítása a `start()` tagfüggvény meghívásával történik, és ha készen áll, a `start()` tagfüggvény hívja majd meg a `run()` tagfüggvényt. A `threading.Thread` osztály semelyik másik tagfüggvénye nem cserélhető le, ugyanakkor lehetőségünk van új tagfüggvények megadására.

Példa: Többszálú szókereső program

Ebben a részben a `grepword-t.py` program kódját tekintjük át. A program ugyanazt a feladatot látja el, mint a `grepword-p.py`, azzal a különbséggel, hogy a munkát több folyamat helyett több szál között osztja el. A program vázlatos rajza a 9.3. ábrán látható.

A program különösen érdekes tulajdonsága, hogy láthatólag egyáltalán nem használ zárolást. Ez azért lehetséges, mert a megosztott adat egy fájllista, amelynek a tárolására egy `queue.Queue` osztályú objektumot használunk, a `queue.Queue` osztály különlegessége pedig az, hogy mindenféle zárolást belül végez el, vagyis minden olyan alkalommal, amikor új elemek elhelyezését vagy meglévők törlését akarjuk elvégezni, az objektum maga fogja *sorba állítani* az eléréseket. A többszálúság szempontjából az elérések sorba állítása azzal jár, hogy egyidejűleg biztosan nem fogja több szál is elérni az adatot. A `queue.Queue` osztály használatának további előnye, hogy nem nekünk kell megoldanunk a munka elosztását – egyszerűen elhelyezzük a sorban a feladat egyes részeit, és a munkát végző szálak majd jelentkeznek érte, ha az előzővel elkészültek.

A `queue.Queue` osztály „elsőnek be – elsőnek ki" (first in, first out, FIFO) alapon dolgozik. A `queue` modulban találjuk a `queue.LifoQueue` osztályt, amely „utolsónak be – elsőnek ki" (last in, first out, LIFO) módszerrel végzi a feladatot, illetve a `queue.PriorityQueue` (prioritási sor) osztályt is, mely objektumainak kételemű (prioritás, elem) rendezett sorozatokat adunk át, és a feldolgozás a legalacsonyabb prioritású elemmel kezdődik. Minden sorféleség létrehozásakor megadható a maximális méret – ha ezt elérjük, egészen addig nem helyezhető új elem a sorba, amíg néhányat el nem távolítunk.



9.3. ábra
Többszálú program

A `grepword-t.py` program áttekintését három részletben végezzük el. Elsőként a teljes `main()` függvény következik:

```
def main():
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    work_queue = queue.Queue()
    for i in range(opts.count):
        number = "{0}: ".format(i + 1) if opts.debug else ""
        worker = Worker(work_queue, word, number)
        worker.daemon = True
        worker.start()
    for filename in filelist:
        work_queue.put(filename)
    work_queue.join()
```

A felhasználó által megadott paraméterek és a fájllista bekérése ugyanúgy történik, mint eddig. Ha a szükséges információ a birtokunkban van, létrehozunk egy `queue.Queue` objektumot, a ciklus pedig annyiszor fut le, ahány szálat szeretnénk: alapértelmezés szerint ha nem végzünk hibakeresést, a karakterlánc üres –, majd létrehozunk egy `Worker` objektumpéldányt (a `Worker` osztály alaposztálya a `threading.Thread` osztály), amelynek `daemon` tulajdonságához rövidesen visszatérünk. Ennek végeztével elindítjuk a szálat, bár ebben a pillanatban még nincs feladata, hiszen a munkalista még üres. Így aztán a szál azonnal meg is áll, és nem is indul tovább, amíg feladatot nem kap.

Most, hogy minden szál munkára készen áll, bejárjuk a fájlokat, és elhelyezzük őket a munkasorban. Amint az első fájl bekerül a sorba, valamelyik szál lecsap rá, és dolgozni kezd rajta – és ez így folytatódik, amíg van feladat nélkül lévő szál. Ahogy egy szál befejezi a munkát egy fájlon, másikat vesz magának a sorból, egészen addig, amíg a feldolgozandó fájlok el nem fogynak.

Figyeljük meg, hogy a programunk ebben a tekintetben különbözik a `grepword-p.py` programtól, ahol nekünk kellett átadnunk a fájllista szeleteit az egyes gyermekfolyamatoknak, amelyeknek az indulása – és így a listák átvétele is – egymás után történt. A szálak használata ekképpen a nagyobb hatékonyság lehetőségét rejti magában. Példának okáért, ha az első öt fájl nagyon nagy, a többi pedig kicsi, akkor – minthogy minden szál egyszerre csak egy feladatot kap meg – a nagy fájlok különböző szálakhoz kerülnek, és így megvalósul a munka egyenlő elosztása. A többfolyamatos megközelítés alkalmazásával – amit a `grepword-p.py` program esetében is tettünk – minden nagy fájl az első folyamathoz kerülne, a kicsik meg a többihez, és végül az első folyamat végezné a munka túlnyomó részét, a többiek meg gyorsan végeznének, és alig-alig dolgoznának.

Amíg van futó szál, a programunk nem áll le. Ez azért baj, mert a szálak technikai értelemben tovább futnak akkor is, ha már befejezték a feladatukat. A megoldás az, ha a szálakat démonokká alakítjuk, ami arra jó, hogy a programunk le tud állni, ha már csak olyan szálai vannak, amelyek nem démonok. A fő szál nem démon, vagyis amikor a fő szál befejezi a futását, a program szépen megszakítja a démonszálak futását, és végül saját maga is leáll. Természetesen ez esetleg az előzőekben vázolt problémával ellentétesen rossz helyzethez vezethet. Így ha a munkát végző szálak elindultak, vigyáznunk kell arra, hogy a fő szál is tovább fusson – egészen addig, amíg van elvégzendő feladat. Ezt a problémát a `queue.Queue.join()` tagfüggvény meghívásával oldhatjuk meg: a tagfüggvény addig várakozik, amíg ki nem ürül a sor. Íme a `Worker` alosztály eleje:

```
class Worker(threading.Thread):
    def __init__(self, work_queue, word, number):
        super().__init__()
        self.work_queue = work_queue
        self.word = word
        self.number = number
```

```
    def run(self):
        while True:
            try:
                filename = self.work_queue.get()
                self.process(filename)
            finally:
                self.work_queue.task_done()
```

Az `__init__()` tagfüggvénynek az ősosztály `__init__()` tagfüggvényét kell meghívnia. A munkasor egyetlen `queue.Queue` objektum; ezen osztozik valamennyi szál.

A `run()` tagfüggvényt végtelen ciklusként írtuk meg. Ez a démonszálak esetében általánosan elterjedt módszer, és azért járunk el így, mert nem tudjuk, hogy a szálnak hány fájlt kell feldolgoznia. Minden ismétlődés alkalmával a `queue.Queue.get()` tagfüggvénnyel kapjuk meg a következő feldolgozandó fájlt. Ha a sor üres, a hívásnak várakoznia kell, és azért nem kell zárolással ügyelnünk rá, mert a `queue.Queue` objektumok ezt automatikusan elvégzik helyettünk. Ha van feldolgozható fájlunk, fel is dolgozzuk, de utána meg kell mondanunk a sornak, hogy az adott feladatot befejeztük – a `queue.Queue.join()` tagfüggvény helyes működésének elengedhetetlen feltétele, hogy a `queue.Queue.task_done()` tagfüggvény meghívása megtörténjen.

A `process()` függvényt nem mutattuk be, ugyanis a `previous = ""` sortól a kód végéig megegyezik a `grepword-p-child.py` fájlban lévővel – a `def` utasítást tartalmazó sortól eltekintve.

Megjegyeznénk még egy utolsó dolgot: a könyv példái között megtaláljuk a `grepword-m.py` nevű programot, amely majdnem teljesen megegyezik a `grepword-t.py` nevű, az imént áttekintett társával, azzal a különbséggel, hogy a `threading` modul helyett a `multiprocessing` modult használja. A kódban mindössze három helyen van eltérés. Először is a `queue` és a `threading` modul helyett a `multiprocessing` modult töltjük be. Másodszor, a `Worker` osztályt a `threading.Thread` osztály helyett a `multiprocessing.Process` osztályból származtatjuk. Harmadszor pedig, a munkasor nem `queue.Queue`, hanem `processing.JoinableQueue` objektumként jön létre.

A `multiprocessing` modul a szálak használatához hasonló viselkedést elágazással valósítja meg az elágazást támogató (Unix) rendszereken, és gyermekfolyamatokkal az elágazást nem támogató (Windows) rendszereken. Így nem kell mindig zárolást használnunk, a folyamataink pedig annyi processzormagon fognak futni, ahány az operációs rendszer rendelkezésére áll. A csomag a folyamatok közti adatátadásnak számos módját kínálja – többek között olyan sor is található benne, amely ahhoz hasonlóan osztja szét a feladatokat a folyamatok között, ahogy a `queue.Queue` objektumok a szálakat látják el tennivalóval.

A `multiprocessing` modulon alapuló változat legfőbb előnye, hogy a szálakkal dolgozó változatnál a többmagos gépek esetében elvileg gyorsabb, ugyanis a folyamatait képes az összes elérhető magon futtatni. Tekintsük mindezt annak a fényében, hogy a szabványos Python-értelmező (C nyelven írták, így CPython néven is ismert) egy GIL (Global Interpreter Lock) nevű zárolási eljárást használ, ami annyit tesz, hogy egyszerre csak egy szál hajthat végre Python-kódot. Ez a megszorítás egy megvalósítási részletkérdés, és más Python-értelmezők esetében – amilyen például a Jython* – nem szükségszerűen igaz.

Példa: A többször előforduló fájlokat kereső program többszálú változata

Első pillantásra a többszálúságot bemutató második példaprogram felépítése az előzőhöz hasonló, maga a program ugyanakkor sok szempontból lényegesen kifinomultabb. Két sort használ, az egyik a munkasor, a másikban pedig az eredményeket tárolja. Önálló eredményfeldolgozó szállal bír, így oldva meg azt, hogy az eredményeket az elérhetővé válásukat követően azonnal megjeleníthesse. A program egyaránt igénybe veszi a `threading.Thread` ősosztályból származó alosztályt, és a `threading.Thread()` tagfüggvény hívását, illetve zárolást használ a megosztott adathoz (egy `dict` típusú objektumhoz) történő hozzáférések sorba állítására.

A `findduplicates-t.py` program az 5. fejezetben megismert `finddup.py` program továbbfejlesztett változata. Bejárja az aktuális könyvtár – illetve a megadott elérési útvonal – fájljait, illetve az alkönyvtárakat is. A `finddup.py` programmal megegyező módon összehasonlítja egymással az egyező nevű fájlok hosszát. Az egyező nevű és méretű állományoknál MD5 (Message Digest) algoritmussal megállapítja, hogy a fájlok tényleg megegyeznek-e, és az egyezéseket a képernyőre írja.

Először a `main()` függvényt tekintjük át, mégpedig négy részletben.

```
def main():
    opts, path = parse_options()
    data = collections.defaultdict(list)
    for root, dirs, files in os.walk(path):
        for filename in files:
            fullname = os.path.join(root, filename)
            try:
                key = (os.path.getsize(fullname), filename)
            except EnvironmentError:
                continue
            if key[0] == 0:
                continue
            data[key].append(fullname)
```

* Annak rövid magyarázatát, hogy a CPython miért a GIL eljárást használja, az alábbi helyeken találjuk meg: www.python.org/doc/faq/library/#can-t-we-get-rid-of-the-global-interpreter-lock és docs.python.org/api/threads.html.

A `data` alapértelmező szótár minden kulcsa egy kételemű (méret, fájlnév) rendezett sorozat, amelyben a fájlnév nem tartalmazza az elérési utat. Az értékek pedig fájlnévlisták, mégpedig az elérési utakat tartalmazó fájlnevekkel. Minden olyan elemről, amelynek az értéklistája egynél több nevet tartalmaz, elképzelhető, hogy többször előforduló fájlt jelent. A szótár feltöltésekor egy adott elérési útvonalon található fájlokat járjuk be, kihagyva azokat, amelyeknek nem tudjuk lekérni a méretét – hiányzó engedélyek miatt, vagy esetleg azért, mert nem rendes állományokról van szó –, illetve azokat, amelyeknek a mérete 0, hiszen az ilyenek ugyebár bizonyosan egyezőek lennének.

```
work_queue = queue.PriorityQueue()
results_queue = queue.Queue()
md5_from_filename = {}
for i in range(opts.count):
    number = "{0}: ".format(i + 1) if opts.debug else ""
    worker = Worker(work_queue, md5_from_filename, results_queue,
                    number)
    worker.daemon = True
    worker.start()
```

Amikor minden adat a helyére került, készen állunk a munkát végző szálak kialakítására. Először a munkasort és az eredménysort hozzuk létre. A munkasor prioritási sor, vagyis mindig a legalacsonyabb prioritású elemet adja elsőként vissza. A mi esetünkben ez a legkisebb méretű fájlt jelenti. Létrehozunk egy szótárat is, amelynek a kulcsai elérési útvonalat tartalmazó fájlnevek, az értékei pedig a fájlok MD5-ellenőrzőösszegei. A szótár arra jó, hogy ne kelljen egynél többször kiszámolni ugyanannak a fájlnak az ellenőrzőösszegét – ez ugyanis meglehetősen erőforrás-igényes feladat.

Most, hogy a megosztott adatgyűjtemények a helyükre kerültek, annyiszor futtatjuk a ciklusmagot, ahány szálat létre szeretnénk hozni – alapértelmezés szerint hétszer. A `Worker` alosztály hasonló ahhoz, mint amit az előzőekben alkottunk, azzal a különbséggel, hogy ezúttal a sorokat és az MD5-összegeket tartalmazó szótárat egyaránt átadjuk neki. Ahogy korábban is tettük, minden feldolgozó szálat azonnal elindítunk, és majd várakoznak, amíg feladatot tudunk adni nekik.

```
results_thread = threading.Thread(
                        target=lambda: print_results(results_queue))
results_thread.daemon = True
results_thread.start()
```

Ahelyett, hogy a `threading.Thread` osztályon alapuló alosztályt hoznánk létre az eredmények feldolgozására, létrehoztunk egy függvényt, amelyet átadunk a `threading.Thread()` tagfüggvénynek. A visszatérési érték olyan egyedi szál, amely a szál elindulása után az átadott függvényt hívja meg. Az eredménysort (ami természetesen üres) adjuk át neki, így a szál azonnal várakozásra kényszerül.

Mostanra elkészült a munkát elvégző összes szál, sőt az eredményfeldolgozó szál is, és mindannyian arra várnak, hogy feladatot kapjanak.

```
for size, filename in sorted(data):
    names = data[size, filename]
    if len(names) > 1:
        work_queue.put((size, names))
work_queue.join()
results_queue.join()
```

A fenti kóddal bejárjuk a `data` objektumot. Minden olyan (méret, fájlnév) kételemű rendezett sorozat esetében, amelyben legalább két olyan fájlt találunk, amelyekről elképzelhető, hogy egymás másolatai, a méretet és az elérési utakat is tartalmazó fájlneveket új feladatként elhelyezzük a munkasorban. Minthogy a sor osztálya a `queue` modulból származik, a zárolással nem kell foglalkoznunk.

Végül a `work_queue.join()` és a `results_queue.join()` tagfüggvény hívásával megoldjuk, hogy a sorok kiürüléséig várakozni kelljen. Így biztosítható, hogy a program addig fusson, amíg minden feladatát el nem végzi, és minden eredmény a képernyőre nem kerül. Ha ezzel elkészült, a futása tisztán ér véget.

```
def print_results(results_queue):
    while True:
        try:
            results = results_queue.get()
            if results:
                print(results)
        finally:
            results_queue.task_done()
```

A fenti függvényt a `threading.Thread()` tagfüggvénynek adjuk át paraméterül, és akkor fut le, amikor a szál, amelynek átadtuk, elindul. Azért találunk benne végtelen ciklust, mert démonszálként fogjuk használni. A feladata abban rejlik, hogy megkapja az eredményeket – egy többsoros karakterláncról van szó –, és ha a kapott karakterlánc nem üres, a képernyőre írja a tartalmát – amíg csak van mit kiírni.

A `Worker` osztály eleje hasonló a korábban megismerthez:

```
class Worker(threading.Thread):
    Md5_lock = threading.Lock()
    def __init__(self, work_queue, md5_from_filename, results_queue,
                 number):
        super().__init__()
        self.work_queue = work_queue
        self.md5_from_filename = md5_from_filename
        self.results_queue = results_queue
        self.number = number
```

```
    def run(self):
        while True:
            try:
                size, names = self.work_queue.get()
                self.process(size, names)
            finally:
                self.work_queue.task_done()
```

A különbséget az jelenti, hogy több olyan megosztott adatunk van, amelyet nyomon kell követnünk, valamint az, hogy az általunk készített `process()` függvényt más paraméterekkel hívjuk meg. A sorok miatt nem kell aggódnunk, mivel azok majd megoldják az elérések sorba állítását, de más adatelemek esetében – gondolunk itt az `md5_from_filename` szótárra – ezt a sorba állítást egy zárolás kialakításával magunknak kell megoldanunk. A zárolást az osztály egy jellemzőjeként valósítottuk meg, ugyanis azt szeretnénk, ha a `Worker` osztály minden példánya ugyanazt a zárolást használná. Így aztán, ha valamelyik példány érvénybe juttatja a zárolást, a többi példánynak addig kell várakoznia, amíg ugyanaz a példány fel is oldja a zárat.

A `process()` függvényt két részletben tekintjük át.

```
    def process(self, size, filenames):
        md5s = collections.defaultdict(set)
        for filename in filenames:
            with self.Md5_lock:
                md5 = self.md5_from_filename.get(filename, None)
            if md5 is not None:
                md5s[md5].add(filename)
            else:
                try:
                    md5 = hashlib.md5()
                    with open(filename, "rb") as fh:
                        md5.update(fh.read())
                        md5 = md5.digest()
                        md5s[md5].add(filename)
                    with self.Md5_lock:
                        self.md5_from_filename[filename] = md5
                except EnvironmentError:
                    continue
```

Kezdetben még üres az az alapértelmező szótár, amelynek később az MD5-ellenőrzőösszegek lesznek a kulcsai, az értékek pedig halmazok lesznek: azoknak a fájlneveknek a halmazai, amelyeknek az MD5-ellenőrzőösszegei megegyeznek a kulcsban tárolttal. A fájlok bejárásakor kiolvassuk az ellenőrzőösszeget, ha már kiszámoltuk, és kiszámoljuk, ha még nem tettük meg.

Környezetkezelők
375 ☜

Az `md5_from_filename` szótárat akkor is zároljuk, ha olvasás céljából nyúlunk bele, és akkor is, ha beleírunk valamit. A `threading.Lock` osztály példányai belépéskor zárolást hajtanak végre, kilépéskor pedig feloldják a zárolást. A `with` utasítások várakoznak, amíg egy másik szál birtokában van az `Md5_lock` zár, egészen addig, amíg a zárolás meg nem szűnik. Az első `with` utasítással zárolást hajtunk végre, hogy a szótárból ki tudjuk olvasni az MD5-összeget – ha a szótár még üres, a `None` különleges szerepű objektumot olvassuk ki. Ha a kiolvasott MD5-összeg értéke a `None` objektum, akkor az összeget nekünk kell kiszámolnunk. Ilyenkor az összeget elhelyezzük az `md5_from_filename` szótárban, így kerülve el a többszöri számítást egyetlen fájl esetében.

Figyeljük meg, hogy a várakozás időtartamát úgy próbáljuk a lehető legrövidebbé tenni, hogy minden lehetséges esetben a zárolás időtartama alatt végzett munka mennyiségének csökkentésére törekszünk. Esetünkben minden egyes zároláskor éppen csak egy szótárműveletet hajtunk végre.

GIL
423 ☜

Szigorúan véve, ha a CPython értelmezőt használjuk, nem szükséges zárolnunk, a GIL ugyanis elvégzi a szótárműveletek felügyeletét helyettünk. Mindazonáltal úgy döntöttünk, hogy a program írása során nem támaszkodunk a GIL létére, és inkább mi magunk végezzük a zárolásokat.

```
for filenames in md5s.values():
    if len(filenames) == 1:
        continue
    self.results_queue.put("{0}Duplicate files ({1:n} bytes):"
                           "\n\t{2}".format(self.number, size,
                             "\n\t".join(sorted(filenames))))
```

Végül egy ciklussal végignézzük az `md5s` nevű helyi alapértelmező szótárat, és minden olyan halmaz esetében, amely legalább két fájlnevet tartalmaz, egy többsoros karakterláncot fűzünk az eredményeket tároló sorhoz. A karakterlánc tartalmazza a szál számát (alapértelmezés szerint egy üres karakterláncot), a fájl bájtban kifejezett méretét, és a többször előforduló fájl neveit. Amikor az eredményeket tároló sorral végzünk műveleteket, nem kell zárolást használnunk. A sor ugyanis `queue.Queue` objektum, és mint ilyen, a kulisszák mögött automatikusan ellátja a zárolással kapcsolatos teendőket.

A `queue` modul osztályai nagymértékben leegyszerűsítik a többszálú alkalmazások készítését, és amikor mégis magunknak kell elvégeznünk a zárolást, a `threading` modul siet a segítségünkre. Példánkban a lehetőségek közül a legegyszerűbbet, a `threading.Lock` osztályt használtuk. Vannak további módszerek is, például a `threading.RLock` osztály (olyan zárolás, amelyet a zárolást már birtokló szál újra és újra végrehajthat), a `threading.Semaphore` osztály (olyan zárolás, amely adott számú forrás védelmére alkalmas) és a `threading.Condition` osztály használata, amely a várakozás feltételét adja meg.

GIL
423 ☜

Többszálú programok írásával gyakran elegánsabban oldhatjuk meg a felmerülő problémákat, mint ha a `subprocess` modult használnánk, de sajnos ha a több folyamatot használó programokkal hasonlítjuk össze őket, a többszálú Python-programok nem szükségszerűen nyújtják a lehető legjobb teljesítményt. Ahogy korábban már jeleztük, ez a probléma a Python alapmegvalósításában rejlik, a CPython értelmező ugyanis egyszerre csak egy processzormagon tudja futtatni a kódot, akkor is, ha a kód több szálat használ.

A problémát megoldani hivatott csomagok egyike a `multiprocessing` modul. Ahogy már korábban is jeleztük, a `grepword-m.py` program valójában a `grepword-t.py` programnak a `multiprocessing` modulra épülő változata, és a két program mindössze három sorban különbözik egymástól. Az előzőekben áttekintett `findduplicates-t.py` program esetében is lehetséges lenne hasonló átalakítás, de a gyakorlatban ez nem tanácsos. Bár a `multiprocessing` modulban lévő API igen-igen hasonlít a `threading` modul API-jára – ezzel is könnyítve a programok átalakítását –, a két API mégsem azonos, és mindkettőnek megvannak a maga bogarai. Ezen felül elmondható, hogy bár egy olyan kicsi és egyszerű programot, mint a `grepword-t.py`, bizonyosan egyszerű mechanikusan a `threading` modul használatáról a `multiprocessing` modul használatára átírni, a megközelítés túlságosan durva lenne a `findduplicates-t.py` program esetében. A legjobb eredmény általában akkor várható, ha a tervezés megkezdésétől fogva szem előtt tartjuk a `multiprocessing` modul használatát. (A `findduplicates-m.py` program megtalálható a program példái között. Ugyanazt a feladatot látja el, mint a `findduplicates-t.py` program, de tőle igencsak eltérően működik, és a működését a `multiprocessing` modul használatára alapozza.)

A CPython értelmező egy a többszálú programok írását jobban támogató változatának már folyik a fejlesztése. A munka állapotáról a legfrissebb hírek a `http://code.google.com/p/python-safethread/` címen olvashatók.

Összefoglalás

A fejezet az olyan programok készítésének módját mutatta be, amelyek az alapkönyvtár `subprocess` moduljának használatával más programok elindítására képesek. A `subprocess` modul használatával futtatott programok kaphatnak parancssori paramétereket, kaphatnak adatokat a szabványos bemenetükre, a szabványos kimenetük és hibacsatornájuk pedig kiolvasható. A gyermekfolyamatok használata lehetővé teszi, hogy a többmagos processzorok minden előnyét kihasználjuk, a párhuzamos futásból eredő erőforrás-megosztási problémák kezelését pedig az operációs rendszerre hagyhatjuk. A módszer hátránya, hogy ha adatokat kell megosztanunk, illetve a folyamatok futását kell egymáshoz igazítanunk, ki kell agyalnunk valamilyen kommunikációs eljárást – például meg kell osztanunk a memória egy területét (ez többek között az `mmap` modullal oldható meg), megosztott fájlokat kell kialakítanunk, vagy a hálózatot kell igénybe vennünk. Ez pedig nem mindig egyszerű.

A fejezet azt is bemutatta, hogy miként készíthetők többszálú programok. Az ilyen programok sajnos nem képesek élni a többmagos processzorok minden előnyével (mármint, ha az alap CPython értelmezőt használjuk), vagyis amikor a teljesítmény fontos szempont, és a Pythonnal dolgozunk, a több folyamat használata gyakran előnyösebb. Mindezek mellett láthattuk, hogy a `queue` modul használata és a Python zárolási módszerei annyira egyszerűen használhatók, amennyire ez egyáltalán elképzelhető. Tanúi lehettünk annak is, hogy az olyan egyszerű programok esetében, amelyeknek a `queue.Queue`, illetve a `queue.PriorityQueue` objektumoknál bonyolultabbra nincs szükségük az adattároláshoz, a zárolással egyáltalán nem kell magunknak foglalkoznunk.

Bár a többszálú programoknak kétségkívül vannak előnyeik, a megírásuk és karbantartásuk, illetve a bennük végzett hibakeresés az egyszálú programokhoz képest sokszor jelentős többletterhet ró a programozóra. A többszálú programoknál ugyanakkor lényegesen egyszerűbb a szálak közötti kapcsolattartás – már amennyiben zárolást vagy valamilyen `queue` objektumot használunk –, illetve az összehangolás – például az eredmények összegyűjtése – , mint a gyermekfolyamatokat használóknál. A többszálúság az olyan grafikus felületű programok esetében is igen hasznos lehet, amelyeknek hosszú ideig tartó számításokat kell végezniük, és közben folyamatosan reagálniuk kell a felhasználó felől érkező parancsokra – például a futó feladat megszakítását célzókra. Ha azonban megosztott memórián, több oldalról is elérhető sor adatszerkezeten, vagy hasonlóan hatékony csatornán át kommunikálhatnak, a több szálon alapuló megoldásokkal szemben érdemes lehet a több folyamaton alapuló megoldások mellett dönteni.

A következő fejezetben ismét egy többszálú programmal ismerkedünk meg: egy olyan kiszolgálóval, amely a hozzá intézett kéréseket külön szálon kezeli, a megosztott adat védelméről pedig zárolással gondoskodik.

Gyakorlatok

1. Másoljuk le a `grepword-p.py` programot, és módosítsuk úgy, hogy ne a gyermekfolyamatok írják a kimenetet a képernyőre, hanem gyűjtse össze mindet a főprogram, és miután minden gyermekfolyamat futása véget ért, a főprogram rendezze, majd írja a képernyőre az eredményeket. A feladat elvégzéséhez csak a `main()` függvényt kell módosítanunk: három során változtatnunk kell, és három új sort kell hozzáírnunk. A gyakorlat elvégzése igényel némi gondolkodást és odafigyelést, valamint el kell olvasnunk a `subprocess` modul dokumentációját. A megoldást a `grepword-p_ans.py` fájlban találjuk.
2. Írjunk olyan többszálú programot, amely beolvassa a parancssorban megadott fájlokat, illetve a parancssorban megadott könyvtárak állományait is. Minden XML-fájlt (vagyis az „`<?xml`” karaktersorozattal kezdődőket) olvassunk be és dolgozzuk fel valamelyik XML-feldolgozóval. Állítsuk elő az egyedi címkék listáját, ha pedig a feldolgozás hibába ütközik, jelenítsünk meg hibaüzenetet. Példaként íme a program egy adott futásának kimenete:

```
./data/dvds.xml is an XML file that uses the following tags:
    dvd
    dvds
./data/bad.aix is an XML file that has the following error:
    mismatched tag: line 7889, column 2
./data/incidents.aix is an XML file that uses the following tags:
    airport
    incident
    incidents
    narrative
```

A program megírásának legegyszerűbb módja, ha a `findduplicates-t.py` program másolatát írjuk át, bár természetesen lehetőségünk van a program írását teljesen elölről elkezdeni. Kisebb mértékben módosítanunk kell a `Worker` osztály `__init__()` és `run()` tagfüggvényét, a `process()` tagfüggvényt pedig teljesen újra kell írnunk (de ez mindössze nagyjából húsz sort jelent). A program `main()` függvényét számos helyen egyszerűsítenünk kell, ami a `print_results()` függvény egyik sorára is igaz. A használati utasítást is át kell alakítanunk, az alábbiaknak megfelelően:

```
Usage: xmlsummary.py [options] [path]
outputs a summary of the XML files in path; path defaults to .

Options:
  -h, --help   show this help message and exit
  -t COUNT, --threads=COUNT
                the number of threads to use (1..20) [default 7]
  -v, --verbose
  -d, --debug
```

Mindenképpen próbáljuk meg a `--debug` kapcsolóval futtatni a programot, így ugyanis ellenőrizhetjük, hogy a szálak rendben elindultak, és mindegyik szál elvégzi a munka rá eső részét. A megoldást a száz sornál alig hosszabb, közvetlen zárolást nem végző `xmlsummary.py` fájlban találjuk.

10

Hálózatkezelés

- TCP-ügyfél létrehozása
- TCP-kiszolgáló létrehozása

A hálózatok lehetővé teszik a számítógépprogramok számára, hogy kapcsolatba lépjenek egymással, még akkor is, ha különböző gépeken futnak. Egyes programoknak (ide tartoznak például a webböngészők) ez a kapcsolatteremtés a lényegi feladatuk, másoknál viszont a hálózati lehetőségek csak kiegészítik az eredeti működést, például a távoli működtetés, illetve naplózás biztosításával, vagy éppen azzal, hogy a hálózatok révén adatokat tárolhatunk egy másik rendszeren, illetve betölthetjük őket onnan. A legtöbb hálózati program egyenrangú partnerek között (ilyenkor ugyanaz a program fut különböző gépeken), illetve még gyakrabban ügyfél-kiszolgáló alapokon működik, ahol az ügyfélprogram kérelmeket küld a kiszolgálónak.

Fejezetünkben elkészítünk egy igen egyszerű ügyfél-kiszolgáló alkalmazást. Az ilyen típusú alkalmazásokat rendszerint két külön programban valósítják meg – a kiszolgáló fogadja a kérelmeket és válaszol rájuk, míg egy vagy több ügyfél kérelmeket küld a kiszolgálónak, és várakozik a kiszolgáló válaszára. Mindehhez az ügyfeleknek tudniuk kell, hol találják a kiszolgálót, vagyis ismerniük kell az IP-címét, valamint a kapuszámát.* Ráadásul az ügyfélnek és a kiszolgálónak meg kell egyeznie az adatátviteli protokollban, és olyan formátumban kell elküldeniük az adataikat, amelyet kölcsönösen megértenek.

A Python alacsonyszintű `socket` modulja (erre épül a nyelv összes magasabb szintű hálózatkezelő modulja) támogatja mind az IPv4, mind az IPv6 formátumú címek használatát. Mindemellett ismeri a leggyakrabban használt hálózati protokollokat, mint az UDP (User Datagram Protocol – ez egy könnyűsúlyú, de megbízhatatlan, kapcsolat nélküli protokoll, amelyben az adatok csomagok alakjában utaznak, de semmi sem garantálja, hogy célba is érkeznek) és a TCP (Transmission Control Protocol – ez egy megbízható, adatfolyamköz-

* A gépek kiépíthetik a kapcsolatot a szolgáltatás feltárása révén is, például a bonjour API használatával – a megfelelő modulokat megtalálhatjuk a Python Package Indexen, a `pypi.python.org/pypi` címen.

pontú protokoll). A TCP-vel tetszőleges adatmennyiség átvihető – a csatoló feladata, hogy hálózati átvitelre alkalmas darabokra bontsa az adatokat, majd a megérkezésük után újra összeállítsa a darabokból az eredeti alakot.

Befőttek 293 ☜

Az UDP-t gyakran használják olyan műszerek adatainak továbbítására, amelyek folyamatosan szolgáltatnak eredményt, viszont nem számít, ha olykor-olykor kimarad egy adat – hasonlóképpen, a protokoll alkalmazható hang- és videófolyamok átvitelére is abban az esetben, amennyiben nem gond, ha egy-egy képkocka elvész az út során. Mind az FTP, mind a HTTP a TCP-re épül, hasonlóan a legtöbb ügyfél-kiszolgáló alkalmazáshoz, amelyeknek szükségük van a TCP kapcsolatközpontú működésére és megbízhatóságára. Ebben a fejezetben egy ügyfél-kiszolgáló programot készítünk, következésképpen mi is a TCP lehetőségeit vesszük igénybe.

Döntenünk kell emellett arról is, hogy az adatainkat szöveges sorok vagy bináris blokkok alakjában szeretnénk küldeni és fogadni, továbbá – amennyiben az utóbbit választottuk – a formátumot is meg kell határoznunk. Fejezetünkben olyan bináris blokkokat használunk, amelyekben az első négy bájt az utánuk következő adatcsomag hosszát adja meg (egy előjel nélküli egész alakjában, amelyet a `struct` modullal állítunk elő), ez az adatcsomag pedig nem más, mint egy befőtt. Ennek a megközelítésnek nagy előnye, hogy ugyanazt a küldő és fogadó kódot használhatjuk bármely alkalmazás esetében, hiszen szinte tetszőleges adatokat tárolhatunk befőtt formájában. Hátránya ugyanakkor, hogy a befőttek használatához mind az ügyfelet, mind a kiszolgálót Python nyelven kell elkészítenünk, illetve mindenképpen lehetőséget kell biztosítanunk a Python elérésére (erre alkalmas a Jython könyvtár a Javában, illetve a Boost.Python a C++-ban). Mindemellett a befőttekkel kapcsolatos biztonsági meggondolások, amelyekről már szót ejtettünk, itt is érvényesek.

A hálózatok lehetőségeit egy autóregisztrációs program példáján mutatjuk be. A kiszolgáló tárolja a regisztrációs adatokat (a rendszámot, az ülések számát, a megtett kilométereket és a tulajdonos nevét), az ügyféllel pedig kiolvashatjuk az autó adatait, módosíthatjuk a kilométerek számát és a tulajdonos nevét, illetve létrehozhatunk egy új bejegyzést. Tetszőleges számú ügyfelet munkába állíthatunk, és ezek nem zavarják egymás működését, még akkor sem, ha ketten egyszerre próbálják elérni a kiszolgálót. Mindez úgy lehetséges, hogy a kiszolgáló minden ügyfél kérelmét külön szállal kezeli. (Látjuk majd, hogy ugyanilyen könnyedén különböző folyamatokat is használhatnánk.)

A példában a könnyebb megvalósítás érdekében az ügyfeleket és a kiszolgálót egyazon gépen futtatjuk, vagyis IP-címként használhatjuk a „localhost" kulcsszót (ha a kiszolgáló nem ugyanazon a gépen fut, az ügyfeleknek átadhatjuk az IP-címét a parancssorban, és a rendszer gond nélkül működik, ha csak egy tűzfal az útjába nem áll). Meghatároztuk a kapuszámot is – esetünkben a 9653 bizonyult szimpatikusnak. A választásban azonban igen tág lehetőségeink vannak – a kapuszámok általában az 5001–32767 tartományból kerülnek ki, de elméletben 1024 és 65 535 között bármelyik számot kiválaszthatjuk.

A kiszolgáló a következő öt típusú kérelmet képes fogadni: `GET_CAR_DETAILS`, `CHANGE_MILEAGE`, `CHANGE_OWNER`, `NEW_REGISTRATION` és `SHUTDOWN` – ezek mindegyikére megfelelő válasszal is rendelkezik, ami a kért adatokat, a kért művelet visszaigazolását, illetve egy hibajelzést tartalmazhat.

TCP-ügyfél létrehozása

Ügyfélprogramunk a `car_registration.py` fájlban kap helyet. Lássuk most, hogyan is folyik a „beszélgetés" a kiszolgálóval (a kiszolgáló már fut, a menüt pedig egy kicsit átszerkesztettük, hogy elférjen az oldalon):

```
(C)ar  (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License: 024 hyr
License: 024 HYR
Seats:   2
Mileage: 97543
Owner:   Jack Lemon
(C)ar  (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]: m
License [024 HYR]:
Mileage [97543]: 103491
Mileage successfully changed
```

A beírt adatokat **félkövérrel** jelöltük – ahol nem látszik a bemenet, ott a felhasználó egyszerűen ENTER-t nyomott, és ezzel elfogadta az alapértelmezést. Jelen esetben a felhasználó egy autó regisztrációs adatait nézte meg, majd módosította a megtett kilométerek számát.

Annyi ügyfelet futtathatunk, amennyit csak akarunk, és ha a felhasználó bezárja az egyik ügyfelet, ez semmilyen hatással nincs a kiszolgálóra. Ha azonban a kiszolgálót leállítjuk, az az ügyfél, amelyikben az erre vonatkozó parancsot kiadtuk, leáll, a többiek pedig „Connection refused" (A kapcsolat elutasítva) hibaüzenetet kapnak, és a működésük megszakad, amint legközelebb kapcsolódni próbálnak a kiszolgálóhoz. Alkalmazásunk kifinomultabb változatában elérhetnénk, hogy a leállításra csak bizonyos felhasználóknak legyen lehetősége, esetleg csak egyes gépeken, de most egyszerűen az ügyfélprogramba építettük be ezt a lehetőséget, hogy lássuk, miként is élhetünk vele.

A következőkben áttekintjük a kódot – a `main()` függvénnyel és a felhasználói felület kezelésével kezdjük, végére hagyva magát a hálózatkezelési kódot.

```
def main():
    if len(sys.argv) > 1:
        Address[0] = sys.argv[1]
    call = dict(c=get_car_details, m=change_mileage, o=change_owner,
                n=new_registration, s=stop_server, q=quit)
    menu = ("(C)ar Edit (M)ileage Edit (O)wner (N)ew car "
            "(S)top server (Q)uit")
    valid = frozenset("cmonsq")
```

```
previous_license = None
while True:
    action = Console.get_menu_choice(menu, valid, "c", True)
    previous_license = call[action](previous_license)
```

Az `Address` egy globális változó, amely kételemű listában tárolja az IP-címet és a kapuszámot, `["localhost", 9653]` alakban – az IP-cím azonban a parancssorból felülírható. A `call` szótár a menüpontokat függvényeknek felelteti meg.

A `Console` modul, amelyhez ezzel a könyvvel juthatunk hozzá, hasznos függvényeket tartalmaz, amelyekkel a konzolon keresztül értékeket olvashatunk be a felhasználóktól – ilyenek például a `Console.get_string()` és a `Console.get_integer()`. Ezek a függvények hasonlóak a korábbi fejezetekben elkészítettekhez; a könnyebb elérhetőség kedvéért helyeztük el őket egy modulban.

A felhasználók kényelmét szolgálja, hogy a program megjegyzi a legutóbb kért rendszámot, amelyet alapértelmezettként használhatunk, hiszen a legtöbb parancs első kérdése éppen erre az adatra irányul. Ha a felhasználó meghozta a döntését, meghívjuk a megfelelő függvényt, átadva az előző rendszámot, és elvárjuk, hogy a függvény működése végén az általa használt rendszámot adja vissza. Mivel a parancsokat kezelő ciklus végtelen, a programot egy függvényből kell végül befejeznünk – ezt majd láthatjuk a későbbiekben.

```
def get_car_details(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is not None:
        print("License: {0}\nSeats: {1[0]}\nMileage: {1[1]}\n"
              "Owner:   {1[2]}".format(license, car))
    return license
```

Ezzel a függvénnyel az egyes autókról szerezhetünk adatokat. Mivel a legtöbb függvénynek szüksége van egy rendszámra a felhasználótól, valamint gyakran az autó egyéb adataira is, amelyekkel műveleteket végezhet, ezek megszerzését egy `retrieve_car_details()` nevű függvényre bíztuk. Ez egy rendezett párt ad vissza, amelyben szerepel a felhasználó által megadott rendszám, valamint egy névvel ellátott rendezett sorozat, a `CarTuple`, amely tartalmazza az autó üléseinek számát, a megtett kilométereket, valamint a tulajdonos nevét (ismeretlen rendszám esetén az eredményként kapott rendezett párban az előző rendszám és a `None` szerepel). Itt azonban mindössze kiírjuk a kiolvasott adatokat, és visszaadjuk azt a rendszámot, amelyet a program alapértelmezésként használ a következő olyan függvény hívásakor, amelynek szüksége van rendszámra.

```
def retrieve_car_details(previous_license):
    license = Console.get_string("License", "license",
                                 previous_license)
    if not license:
        return previous_license, None
```

```
    license = license.upper()
    ok, *data = handle_request("GET_CAR_DETAILS", license)
    if not ok:
        print(data[0])
        return previous_license, None
    return license, CarTuple(*data)
```

Ez az első olyan függvényünk, amely kihasználja a hálózatkezelési lehetőségeket. Mindebből pillanatnyilag csak annyit látunk, hogy a kód meghívja a `handle_request()` függvényt, amelyet a későbbiekben veszünk szemügyre közelebbről. A működése röviden annyiból áll, hogy elküldi a számára átadott adatokat a kiszolgálónak, és visszaadja a kapott választ. Nem törődik azzal, hogy milyen adatok haladnak át rajta: tisztán a hálózati forgalmat biztosítja.

Az autóregisztrációk esetében a protokoll egyszerűen abban áll, hogy első paraméterként átküldjük a kiszolgálónak az elvégzendő művelet kódját, másodikként pedig a szükséges adatokat – esetünkben mindössze a rendszámra lesz szükség. A válasz protokollja szerint a kiszolgáló mindig egy rendezett párt küld vissza, amelynek az első eleme egy logikai siker/kudarc jelző. Amennyiben ennek értéke `False`, egy rendezett párt kaptunk a kiszolgálótól, amelynek a második eleme egy hibaüzenet. Ha az érték `True`, két eset lehetséges. Kaphattunk egy rendezett párt, amelynek a második eleme egy visszaigazoló üzenet, vagy egy rendezett n-est, amelyben a második és az azt követő elemek a kért adatokat tartalmazzák.

Itt tehát, ha a rendszám ismeretlen, az `ok` értéke `False`, így kiírjuk a `data[0]`-ban található hibaüzenetet, és változatlan alakban visszaadjuk az előző rendszámot. Egyébként viszont visszaadjuk az új rendszámot (ami ezt követően az „előző” lesz), valamint egy `CarTuple` értéket, amelyet a `data` listából (ülések száma, kilométerszám, tulajdonos) nyertünk ki.

```
def change_mileage(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is None:
        return previous_license
    mileage = Console.get_integer("Mileage", "mileage",
                                  car.mileage, 0)
    if mileage == 0:
        return license
    ok, *data = handle_request("CHANGE_MILEAGE", license, mileage)
    if not ok:
        print(data[0])
    else:
        print("Mileage successfully changed")
    return license
```

A függvény működése alapjaiban a `get_car_details()` esetében látott mintát követi, azzal a különbséggel, hogy ha rendelkezésre állnak az adatok, az egyiküket módosítjuk is. A kódban valójában két hálózati művelet is szerepel, hiszen a `retrieve_car_details()`

meghívja a `handle_request()` függvényt, hogy hozzájusson az autó adataihoz – erre szükségünk van egyrészt azért, hogy igazoljuk a rendszám érvényességét, másrészt hozzá kell jutnunk a kilométerek számához, hogy ezt használhassuk alapértelmezésként. A válasz itt minden esetben egy rendezett pár, amelynek a második tagja egy hibaüzenet vagy a `None` érték.

A `change_owner()` függvény bemutatására nem térünk ki, hiszen a szerkezete azonos a `change_mileage()` kódjában látottal, de nem foglalkozunk a `new_registration()` függvénnyel sem, ugyanis ez csak annyiban tér el az előzőektől, hogy itt nem töltjük be kezdetben az autó adatait (hiszen új autót veszünk fel a rendszerbe), és minden adatot elkérünk a felhasználótól, nem csak egyet-egyet módosítunk. Mindez semmiféle új hálózati ismeretet nem ad, így inkább tovább is lépünk egy érdekesebb függvényhez.

```
def quit(*ignore):
    sys.exit()
def stop_server(*ignore):
    handle_request("SHUTDOWN", wait_for_reply=False)
    sys.exit()
```

Ha a felhasználó maga dönt úgy, hogy kilép a programból, a `sys.exit()` hívásával „tisztán" fejezzük be a program futtatását. Minden menüponthoz tartozó függvényt az előző rendszámmal hívunk meg, de ebben az esetben nem foglalkozunk a paraméterrel. Mindazonáltal a `def quit():` meghatározás hibás működéshez vezetne, ugyanis ez esetben a függvény nem várna paramétert – így az előző rendszámot a számára átadva `TypeError` kivételt kapnánk, ami azt jelzi, hogy a várt nulla helyett egy paramétert adtunk át a függvénynek. Ezért hát a `*ignore` paramétert használjuk, így tetszőleges számú hely szerinti paramétert megadhatunk. Az `ignore` név mindössze arra utal, hogy figyelmen kívül hagyjuk az átadott paramétereket, a Python számára nem bír semmilyen jelentőséggel.

Ha a felhasználó úgy dönt, hogy leállítja a kiszolgálót, ezt az utasítást a `handle_request()` függvénnyel adjuk át – ez esetben nem várunk választ. Ha tehát elküldtük az adatokat, a `handle_request()` visszatér anélkül, hogy a válaszra várna, és végrehajthatunk egy tiszta leállást a `sys.exit()` segítségével.

```
def handle_request(*items, wait_for_reply=True):
    SizeStruct = struct.Struct("!I")
    data = pickle.dumps(items, 3)
    try:
        with SocketManager(tuple(Address)) as sock:
            sock.sendall(SizeStruct.pack(len(data)))
            sock.sendall(data)
            if not wait_for_reply:
                return
            size_data = sock.recv(SizeStruct.size)
            size = SizeStruct.unpack(size_data)[0]
            result = bytearray()
```

```
            while True:
                data = sock.recv(4000)
                if not data:
                    break
                result.extend(data)
                if len(result) >= size:
                    break
        return pickle.loads(result)
    except socket.error as err:
        print("{0}: is the server running?".format(err))
        sys.exit(1)
```

A fenti függvény felel az ügyfélprogram hálózati működéséért. A kódban először létrehozunk egy `struct.Struct` változót, amely egy előjel nélküli egészet tartalmaz hálózati bájtsorrendben, majd pedig létrehozunk egy befőttet, amelyben a paraméterként kapott adatokat tároljuk. A függvény nem tudja, miféle adatokhoz jutott hozzá, de nem is foglalkozik vele. Figyeljük meg, hogy a 3-as befőttprotokollt állítottuk be – tettük ezt azért, hogy a kiszolgáló és az ügyfelek biztosan ugyanazt a változatot használják, még akkor is, ha időközben valamelyiküknél a Python újabb változatát vették használatba.

Ha a protokollt jobban fel szeretnénk készíteni a jövőre, változatkezelést alkalmazhatunk (hasonlóan a bináris lemezformátumokhoz), amit megtehetünk a hálózat vagy az adatok szintjén. Amennyiben az előbbit választjuk, a korábbi egy helyett adjunk át két előjel nélküli egészet, amelyek közül a második a protokoll változatszámát tartalmazza. Az adatok szintjén ragaszkodhatunk ahhoz, hogy a befőttbe mindig egy lista (vagy mindig egy szótár) kerüljön, amelynek az első („változat”) eleme a tartalmazza a változatszámot. (A változatkezelés alkalmazására lesz még módunk a gyakorlatok során.)

A `SocketManager` egy egyedi környezetkezelő, amely biztosítja számunkra az adatátvitelhez szükséges csatolót – hamarosan közelebbről is megismerkedhetünk vele. A `socket.socket.sendall()` tagfüggvény minden adatot elküld, amit csak megadunk neki, esetleg több `socket.socket.send()` hívást végezve a háttérben. Minden esetben két adatelemet küldünk át: a befőtt hosszát, valamint magát a befőttet. Amennyiben a `wait_for_reply` paraméter értéke `False`, nem várunk a válaszra, hanem azonnal visszatérünk a függvényből – a környezetkezelő biztosítja, hogy lezárjuk a csatolót, mielőtt a visszatérés valóban megtörténne.

Az adatok elküldése után a `socket.socket.recv()` tagfüggvénnyel hozzájutunk a válaszhoz (amennyiben szükségünk van rá). A tagfüggvény blokkolja a programot, amíg hozzá nem jut a várt adatokhoz. Az első hívással négy bájtot kérünk – ennyi ugyanis annak az egésznek a mérete, amely az utána álló befőtt hosszát tárolja. Ha hozzájutottunk, a `struct.Struct` segítségével elkészítjük belőle a `size` nevű egész számot. Ezt követően létrehozunk egy üres `bytearray` típusú változót, és megkíséreljük betölteni a befőttet legfeljebb 4000 bájtos blokkokban. Ha beolvastunk `size` mennyiségű bájtot (illetve ha

az adatfolyam még ez előtt véget ért), kiszállunk a ciklusból, kicsomagoljuk az adatokat a `pickle.loads()` függvénnyel (amely `bytes` vagy `bytearray` objektumot képes fogadni), és visszaadjuk az eredményt. Ez esetben tökéletesen tisztában vagyunk azzal, hogy egy rendezett sorozatot kapunk, hiszen ezt rögzíti az autóregisztrációs kiszolgálóval közös protokoll, de maga a `handle_request()` nem tudja, hogy milyen adatokról van szó, és ez nem is érdekli.

Ha valami gond adódik a hálózati kapcsolattal, például leáll a kiszolgáló, vagy megszakad a kapcsolat, `socket.error` kivételhez jutunk. Ilyenkor a programunk elfogja a kivételt, kiír egy hibaüzenetet, és leáll.

```
class SocketManager:
    def __init__(self, address):
        self.address = address
    def __enter__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect(self.address)
        return self.sock
    def __exit__(self, *ignore):
        self.sock.close()
```

Az `address` objektum egy rendezett pár (IP-cím, kapuszám), amelyet akkor állítunk be, amikor a környezetkezelőt létrehozzuk. Mihelyt a környezetkezelőt egy `with` utasításban alkalmazzuk, létrehoz egy csatolót, és megkísérli a kapcsolat kiépítését – mindaddig blokkol, amíg a kapcsolat létre nem jön, vagy nem kap csatolókivételt. A `socket.socket()` kezdeti értékadó tagfüggvény első paramétere a címcsalád – esetünkben a `socket.AF_INET` (IPv4) beállítást használtuk, de mást is választhatunk, például a `socket.AF_INET6` (IPv6), a `socket.AF_UNIX` vagy a `socket.AF_NETLINK` beállítást. A második paraméter általában az itt is használt `socket.SOCK_STREAM` (TCP) vagy a `socket.SOCK_DGRAM` (UDP).

Ha a programvezérlés elhagyja a `with` utasítás hatókörét, a rendszer meghívja a környezeti objektum `__exit__()` tagfüggvényét. Ez esetben nem törődünk azzal, hogy bekövetkezett-e valamilyen kivétel (így a kivételparamétereket figyelmen kívül hagyjuk), egyszerűen csak bezárjuk a csatolót. Mivel a tagfüggvény visszatérési értéke `None` (logikai környezetben `False`), minden kivétel felfelé vándorol a hívásveremben – ez nagyszerűen beválik, hiszen a csatolókivételek kezelésére elhelyeztünk egy alkalmas `except` blokkot a `handle_request()` kódjában.

TCP-kiszolgáló létrehozása

Mivel a kiszolgálók létrehozásának kódja gyakran igen hasonló szerkezetű, az alacsonyszintű `socket` modul helyett használhatjuk a magasszintű `socketserver` modult, amely elvégzi a háttérmunkát helyettünk. Nekünk mindössze az a feladatunk, hogy egy kérelemkezelő osztályt hozzunk létre egy `handle()` nevű tagfüggvénnyel, amelyet a kérelmek kiadására és

a válaszok fogadására használhatunk. A `socketserver` modul kezeli az adatcserét, kiszolgálja a kapcsolati kérelmeket, akár sorban egymás után, akár minden kérelmet külön szálnak vagy folyamatnak adva át – és mindezt áttetszően teszi, vagyis nem kell foglalkoznunk az alacsonyszintű részletekkel.

Alkalmazásunk kódját a `car_registration_server.py` fájl tartalmazza.* A programban egy igen egyszerű, `Car` nevű osztályt találhatunk, amely tulajdonságok alakjában tárolja az ülések számát, a megtett kilométereket és a tulajdonos adatait (az ülések száma természetesen csak olvasható). A rendszámokat azonban ne itt keressük – az autókat a program egy szótárban tárolja, amelyben a kulcsok a rendszámok.

Először a `main()` függvényt vizsgáljuk meg, majd röviden áttekintjük, hogyan tölti be a rendszer a kiszolgáló adatait, ezt követően pedig létrehozzuk az egyedi kiszolgálóosztályunkat, végül pedig létrehozunk egy osztályt az ügyfelek kérelmeinek kezelésére.

```
def main():
    filename = os.path.join(os.path.dirname(__file__),
                            "car_registrations.dat")
    cars = load(filename)
    print("Loaded {0} car registrations".format(len(cars)))
    RequestHandler.Cars = cars
    server = None
    try:
        server = CarRegistrationServer(("", 9653), RequestHandler)
        server.serve_forever()
    except Exception as err:
        print("ERROR", err)
    finally:
        if server is not None:
            server.shutdown()
            save(filename, cars)
            print("Saved {0} car registrations".format(len(cars)))
```

Az autóregisztrációs adatokat ugyanabban a könyvtárban tároltuk, mint magát a programot. A `cars` objektumban egy szótárat helyezünk el, amelynek a kulcsai a rendszámok karakterláncai, az értékei pedig `Car` objektumok. A kiszolgálók általában nem írnak ki semmit a képernyőre, hiszen az indításuk és a leállításuk rendszerint automatikus, és a háttérben működnek – így az állapotukról naplókban tudósítanak (például a `logging` modul használatával). Most azonban a tesztek és a kísérletezés megkönnyítésére kiírunk egy üzenetet a kiszolgáló indításakor és leállításakor.

* Windows rendszereken a kiszolgáló első futtatása alkalmával kaphatunk egy párbeszédablakot a tűzfaltól, amely figyelmeztet, hogy letiltotta a hálózatelérést a Python számára – a kiszolgáló helyes működése érdekében kattintsunk az Unblock (Tiltás feloldása) gombra.

A kérelemkezelő osztálynak el kell érnie a `cars` szótárat, de ezt nem adhatjuk át egy példánynak, mivel ezeket a példányokat a kiszolgáló hozza létre a számunkra – minden kérelem kezeléséhez egyet. Ezért a szótárat a `RequestHandler.Cars` osztályváltozóban tároljuk, ahol minden példány számára elérhető lesz.

Létrehozzuk a kiszolgáló egy példányát, és átadjuk neki azt a címet és kaput, ahol működni fog, továbbá a `RequestHandler` osztályobjektumot – nem pedig ennek egy példányát. A cím helyén egy üres karakterláncot adunk át, jelezve, hogy bármely elérhető IPv4-cím megfelel (ide tartozik a helyi gép címe, vagyis a `localhost` is). Ez után arra utasítjuk a kiszolgálót, hogy fogadja a kérelmeket örökkön örökké. Amikor a kiszolgáló leáll (arról, hogy ezt miként teszi, hamarosan szólunk), lemezre mentjük a `cars` szótárat, hiszen ezt az ügyfelek időközben megváltoztathatták.

```
def load(filename):
    try:
        with contextlib.closing(gzip.open(filename, "rb")) as fh:
            return pickle.load(fh)
    except (EnvironmentError, pickle.UnpicklingError) as err:
        print("server cannot load data: {1}".format(err))
        sys.exit(1)
```

A betöltés kódja egyszerű, mivel az alapkönyvtár `contextlib` moduljának környezetkezelőjét használtuk annak biztosítására, hogy a rendszer mindenképpen bezárja a fájlt akkor is, ha időközben kivétel lép fel. Ugyanezt az eredményt saját környezetkezelővel is elérhetjük, amelynek a kódja valahogy így alakul:

```
class GzipManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
    def __enter__(self):
        self.fh = gzip.open(self.filename, self.mode)
        return self.fh
    def __exit__(self, *ignore):
        self.fh.close()
```

Egyedi `GzipManager` osztályunkkal a `with` utasítás alakja így fest:

```
with GzipManager(filename, "rb") as fh:
```

A `save()` függvény (amelyet nem mutatunk be) szerkezetileg megegyezik a `load()` függvénnyel, az eltérés mindössze annyi, hogy a fájlt bináris módban nyitjuk meg, az adatokat a `pickle.dump()` függvénnyel mentjük lemezre, és nem adunk vissza semmit.

```
class CarRegistrationServer(socketserver.ThreadingMixIn,
                            socketserver.TCPServer): pass
```

Többszörös öröklés 395 ☜

Bármily meglepő, ez az egyéni kiszolgálóosztályunk teljes kódja. Ha olyan kiszolgálót szerettünk volna készíteni, amely nem szálakat, hanem folyamatokat alkalmaz, mindössze annyiban kellene eltérnünk a fenti kódtól, hogy az osztályunkat a `socketserver.ThreadingMixIn` helyett a `socketserver.ForkingMixIn` osztályból származtatjuk. A *mixin* szócska jobbára olyan osztályra utal, amelyet eleve többszörös öröklésre szántak. A `socketserver` modul osztályai segítségével különféle egyedi kiszolgálókat készíthetünk a megfelelő alaposztályok párjaiból leszármaztatva – így létrehozhatunk UDP-kiszolgálókat, de unixos TCP- és UDP-kiszolgálókat is készíthetünk.

Fontos megjegyeznünk, hogy az általunk használt `socketserver` bekeveredő (mixin) osztálynak az öröklési sorban mindig elöl kell állnia. Ezzel biztosíthatjuk, hogy a rendszer a bekeveredő osztály tagfüggvényeit használja a második osztálybeli társaik helyett, amennyiben névegyezés fordul elő. Ennek az az oka, hogy egy tagfüggvény hívásakor a Python a megadásuk sorrendjében halad végig az alaposztályokon, és azt a tagfüggvényt használja, amelyik elsőként megfelel a hívottnak.

A csatolókiszolgáló az általunk megadott osztály révén kérelemkezelőt hoz létre minden egyes kérelem kezelésére. Egyedi `RequestHandler` osztályunk a kérelmek minden kezelhető típusához külön tagfüggvényt biztosít, emellett meg kell valósítania a `handle()` tagfüggvényt is – hiszen ez az egyetlen olyan tagfüggvény, amelyet a csatolókiszolgáló használ. Mielőtt azonban közelebbről is megismerkednénk a tagfüggvényekkel, tekintsük át az osztály meghatározását és osztályváltozóit.

```
class RequestHandler(socketserver.StreamRequestHandler):

    CarsLock = threading.Lock()
    CallLock = threading.Lock()
    Call = dict(
            GET_CAR_DETAILS=(
                    lambda self, *args: self.get_car_details(*args)),
            CHANGE_MILEAGE=(
                    lambda self, *args: self.change_mileage(*args)),
            CHANGE_OWNER=(
                    lambda self, *args: self.change_owner(*args)),
            NEW_REGISTRATION=(
                    lambda self, *args: self.new_registration(*args)),
            SHUTDOWN=lambda self, *args: self.shutdown(*args))
```

Létrehoztunk egy `socketserver.StreamRequestHandler` nevezetű alosztályt, hiszen egy TCP-adatfolyamkiszolgálót használunk. Az UDP-kiszolgálók esetében hasonló lehetőségeket biztosít a `socketserver.DatagramRequestHandler`, de származtathatjuk az alosztályunkat a `socketserver.BaseRequestHandler` osztályból is, amennyiben alacsonyabb szintű elérésre van szükségünk.

A `RequestHandler.Cars` szótár egy osztályváltozó, amelyet a `main()` függvényben illesztettünk be – ez tartalmazza a regisztrációs adatokat. A Python lehetővé teszi, hogy mindenféle bonyolult kódolás nélkül további jellemzőkkel egészítsünk ki objektumokat (így osztályokat és példányokat) az osztályon kívülről (amennyiben az objektum rendelkezik a `__dict__` jellemzővel), ami igen kényelmes lehetőség. Mivel ismert, hogy az osztály működése ezen a változón áll vagy bukik, egyes programozók szükségesnek érezhetik, hogy a `Cars = None` sorral jelezzék az osztályváltozó jelenlétét az osztályban.

Szinte minden kérelemkezelő tagfüggvénynek szüksége van a `Cars` adataira, de valahogy biztosítanunk kell, hogy egyszerre két tagfüggvény (két különböző szálból) ne érhesse el – ha ugyanis erről nem gondoskodunk, a szótár tartalma károsodhat, és a program akár össze is omolhat. Mindennek elkerülésére bevezettünk egy zárolási osztályváltozót, amellyel biztosíthatjuk, hogy a `Cars` szótárhoz egyszerre csak egy szál férjen hozzá.* (A szálak és a zárolás használatáról a 9. fejezetben olvashattunk bővebben.)

A `Call` szótár szintén osztályváltozó, amelyben a kulcsok a kiszolgáló által végezhető műveletek, az értékek pedig a műveletekhez tartozó függvények. Az utóbbiakat nem használhatjuk közvetlenül, mint az ügyfél menüjének szótárában, ugyanis az osztály szintjén nem érhető el a `self` változó. Ezért burkolófüggvényeket készítettünk, amelyek a hívásuk idején hozzájutnak a `self` objektumhoz, majd meghívják a megfelelő tagfüggvényt, átadva neki a `self` objektumot és a többi szükséges paramétert. Megoldás lehetne az is, ha a `Call` szótárat a tagfüggvények után hoznánk létre. Így már alkalmazhatnánk a `GET_CAR_DETAILS=get_car_details` és ehhez hasonló értékadásokat, hiszen a Python megtalálja a `get_car_details()` tagfüggvényt, amelyet így már létrehoztunk. Azért döntöttünk az előbbi megoldás mellett, mert az alkalmazásának a célja világosabban látszik, és így a kód nem függ a szótár és a tagfüggvények létrehozásának sorrendjétől.

GIL 423 ☜

Jóllehet a `Call` szótár olvasására csak az után kerül sor, hogy az osztályt létrehoztuk, változtathatósága miatt mégis különösen ügyeltünk a biztonságára, és egy zárat hoztunk létre, amely meggátolja, hogy egyszerre két szál férjen hozzá. (A GIL jelenléte miatt a CPythonban erre a zárolásra nincs igazán szükség.)

```
def handle(self):
SizeStruct = struct.Struct("!I")
size_data = self.rfile.read(SizeStruct.size)
size = SizeStruct.unpack(size_data)[0]
data = pickle.loads(self.rfile.read(size))
try:
    with self.CallLock:
        function = self.Call[data[0]]
    reply = function(self, *data[1:])
```

GIL 423 ☜

* A GIL (Global Interpreter Lock) képes biztosítani, hogy a `Cars` szótár elérései összhangban legyenek, de – amint azt a korábbiakban is említettük – nem használjuk ki ezt a lehetőséget, ugyanis ez a CPython megvalósításhoz kötődik.

```
except Finish:
    return
data = pickle.dumps(reply, 3)
self.wfile.write(SizeStruct.pack(len(data)))
self.wfile.write(data)
```

Ahányszor csak egy ügyfél kérelmet ad ki, létrehozunk egy új szálat a `RequestHandler` osztály új példányával, majd meghívjuk ennek a példánynak a `handle()` tagfüggvényét. A tagfüggvény belsejében az ügyféltől érkező adatokat a `self.rfile` objektum révén érhetjük el, a válasz adatait pedig a `self.wfile` objektumon keresztül küldhetjük el – mindkét objektumot a `socketserver` modultól kapjuk, és ilyenkor már nyitva állnak, készen a használatra.

A `struct.Struct` egy egész értékű bájtszámláló, amelyre a „hossz + befőtt" formátumhoz szükségünk van – ilyen alakban cserélünk adatokat az ügyfél és a kiszolgáló között.

Először beolvasunk négy bájtot, és az eredményt a `size` egész változóba csomagoljuk ki, így már tudjuk az átküldött befőtt hosszát. Ezt követően beolvasunk `size` darab bájtot, és tároljuk őket a `data` változóban. Az olvasás alatt blokkoljuk a többi hálózati műveletet. Esetünkben tudjuk, hogy az adatok egy rendezett sorozatot adnak ki, amelynek az első eleme a művelet, az utána következők pedig annak paraméterei – ebben a protokollban állapodtunk ugyanis meg az ügyfelekkel.

A `try` blokk belsejében hozzájutunk a kért műveletnek megfelelő lambdafüggvényhez. A zárolás segítségével letiltjuk a `Call` szótár elérését, jóllehet sokan ezt túlzott óvatosságnak tekinthetik. Szokásainkhoz híven a zárolás hatókörén belül a műveleteinket a lehető legszükségesebbekre korlátozzuk, ami esetünkben egy szótárbeli keresést jelent, amellyel hozzájutunk a kívánt függvény hivatkozásához. Ha megkaptuk, meg is hívjuk, első paraméterként átadva számára a `self` objektumot, ezt követően pedig a `data` sorozat további elemeit. Fontos megértenünk, hogy itt egy egyszerű függvényhívásról van szó, ezért a Python nem adja át automatikusan a `self` paramétert. Persze ez nem jelent gondot, hiszen itt, a lambdafüggvényen belül ezt egyszerű paraméterként használjuk a híváshoz. A végeredmény a `self.`*`tagfüggvény`*`(*data[1:])` hívás, ahol a *tagfüggvény* megfelel a `data[0]`-ban átadott műveletnek.

Amennyiben a művelet a leállítás, a `Finish` egyéni kivételt váltjuk ki a `shutdown()` tagfüggvényben – ilyenkor tudjuk, hogy az ügyfél nem várhat semmiféle választ, így egyszerűen visszatérünk a `return` utasítással. A többi művelet esetében azonban befőzzük a megfelelő tagfüggvény hívásának eredményét (a 3-as befőttprotokollal), majd kiírjuk a befőtt méreteit és magát a befőzött adatcsomagot.

```
def get_car_details(self, license):
    with self.CarsLock:
        car = copy.copy(self.Cars.get(license, None))
```

```
        if car is not None:
            return (True, car.seats, car.mileage, car.owner)
        return (False, "This license is not registered")
```

A fenti tagfüggvény először megpróbál hozzájutni az autók adatainak zárolásához – amíg ezt meg nem szerzi, magánál tartja a vezérlést. A megadott rendszámhoz tartozó autó adatait a `dict.get()` tagfüggvénnyel olvassa ki, második paraméterként a `None` értéket adva át – ha nincs ilyen autó, az eredmény `None`. A program azonnal átmásolja az autó adatait, és a `with` utasítás véget is ér, ezzel a zárolás a lehető legrövidebb ideig van érvényben. Jóllehet az olvasás nem módosítja a kiolvasott adatokat, mégis csak egy módosuló gyűjteménnyel dolgozunk, amelyet egy másik tagfüggvény egy másik szálban esetleg meg szeretne változtatni éppen az olvasás közben – a zárolással ennek elejét vesszük. A zárolás hatókörén kívül ebben a pillanatban rendelkezünk az autó adatainak másolatával (illetve, ha nincs meg az autó, a `None` értékkel), amelyeket nyugodtan kezelhetünk anélkül, hogy bármely szálat feltartanánk.

Az autóregisztráció műveletkezelő tagfüggvényeihez hasonlóan egy rendezett sorozatot adunk vissza, amelynek az első eleme egy logikai siker/kudarc jelző, a többi elem pedig változó lehet az adott művelettől függően. E tagfüggvények egyikének sem kell aggódnia amiatt, hogy milyen adatokat kap, sőt egyáltalán nem kell foglalkoznia vele azon túl, hogy amit kap, az „egy rendezett sorozat, amelynek az első eleme logikai érték” – a hálózati műveleteket ugyanis a `handle()` tagfüggvény végzi.

```
def change_mileage(self, license, mileage):
    if mileage < 0:
        return (False, "Cannot set a negative mileage")
    with self.CarsLock:
        car = self.Cars.get(license, None)
        if car is not None:
            if car.mileage < mileage:
                car.mileage = mileage
                    return (True, None)
                    return (False, "Cannot wind the odometer back")
            return (False, "This license is not registered")
```

Ebben a tagfüggvényben az első vizsgálatot zárolás nélkül is elvégezhetjük – ha azonban a kilométerszám nem negatív, szükségünk lesz a zárolásra, majd hozzá kell jutnunk a megfelelő autóhoz, és ha ez megvan (vagyis ismert rendszámot adtunk meg), még a zárolás hatókörében maradva megváltoztatjuk az óraállást, vagy visszaadunk egy hibaüzenetet. Ha nincs a rendszámhoz tartozó autó a rendszerben (vagyis a `car` értéke `None`), kilépünk a `with` hatóköréből, és egy hibaüzenetet adunk vissza.

Úgy tűnhet, hogy ha az adatok ellenőrzését az ügyfélnél végeznénk el, némi hálózati forgalmat megtakaríthatnánk, így például az ügyfél maga adhatna hibaüzenetet, ha negatív óraállással találkozik (vagy egyáltalán nem tenné lehetővé ilyen adat beírását). Jóllehet

az ügyfél megteheti ezt, továbbra is el kell végeznünk az ellenőrzést a kiszolgálón, hiszen sosem lehetünk biztosak az ügyfelek hibamentes működésében. Ráadásul, még ha tudjuk is, hogy az ügyfél az autó eredeti óraállását használja alapértelmezésként, akkor sem tekinthetjük a felhasználó által beírt adatot érvényesnek (hiába nagyobb az eredeti értéknél), hiszen időközben egy másik ügyfél módosíthatta. Következésképpen szükség van az ellenőrzésre a kiszolgálón, mégpedig a zárolás hatókörén belül.

A `change_owner()` tagfüggvény kódja hasonló, így nem foglalkozunk vele részletesebben.

```
def new_registration(self, license, seats, mileage, owner):
    if not license:
        return (False, "Cannot set an empty license")
    if seats not in {2, 4, 5, 6, 7, 8, 9}:
        return (False, "Cannot register car with invalid seats")
    if mileage < 0:
        return (False, "Cannot set a negative mileage")
    if not owner:
        return (False, "Cannot set an empty owner")
    with self.CarsLock:
        if license not in self.Cars:
            self.Cars[license] = Car(seats, mileage, owner)
            return (True, None)
        return (False, "Cannot register duplicate license")
```

Ezúttal is lehetőségünk van arra, hogy több hibaellenőrzési lépést elvégezzünk a zárolás előtt, de ha minden adat érvényes, a zárolás nem kerülhető el. Ha a rendszám nem található a `RequestHandler.Cars` szótárban (márpedig ha minden rendben folyik, ez a helyzet, hiszen itt új rendszámról van szó), létrehozunk egy új `Car` objektumot, és tároljuk a szótárban. Mindezt egyazon zárolás hatókörében kell megtennünk, ugyanis nem szabad megengednünk, hogy más ügyfél is bejegyezzen egy autót ugyanezzel a rendszámmal a rendszám ellenőrzése és az új autó bejegyzése közti holtidőben.

```
def shutdown(self, *ignore):
    self.server.shutdown()
    raise Finish()
```

Amennyiben a kért művelet a leállítás, meghívjuk a kiszolgáló `shutdown()` tagfüggvényét – ez meggátolja a további kérelmek fogadását, jóllehet a kiszolgáló mindaddig nem áll le, amíg még kezeli a korábban beérkezett kérelmeket. Ezt követően kiváltunk egy egyedi kivételt, amellyel jelezzük a `handler()` számára, hogy végeztünk – ennek hatására a `handler()` függvény futása véget ér anélkül, hogy választ adna az ügyfélnek.

Összefoglalás

Ebben a fejezetben meggyőződhettünk arról, hogy a hálózati ügyfelek és kiszolgálók létrehozása nem különösebben nehéz feladat a Pythonban, köszönhetően az alapkönyvtár hálózatkezelési moduljainak, de sokat segít a `struct` és a `pickle` modul is.

Az első részben egy olyan ügyfélprogramot írtunk, amely egyetlen, `handle_request()` nevű függvénnyel éri el a hálózatot – ez intézi a befőzhető adatok küldését és fogadását a kiszolgáló felé az általános „hossz + befőtt" adatformátumban. A második részben egy kiszolgáló-alaposztályt hoztunk létre a `socketserver` modul osztályainak leszármazottjaként, és megtanultuk, hogy miként valósítsunk meg egy kérelemkezelő osztályt, amely képes az ügyfelektől érkező kérelmek feldolgozására. A hálózati működés lebonyolítása egyetlen, `handle()` nevezetű tagfüggvényre hárul, amely képes az ügyféllel kapcsolatba lépve tetszőleges befőzhető adatok küldésére és fogadására.

A `socket`, a `socketserver`, valamint más, a témakörben érintett modulok (mint az `asyncore`, az `asynchat` vagy az `ssl`) a bemutatottaknál jelentősen tágabb lehetőségekkel rendelkeznek. Ha pedig az alapkönyvtár hálózati képességei nem bizonyulnak elegendőnek, vagy nem elég magas szintűek, érdemes egy pillantást vetnünk a külső fejlesztésű Twisted hálózati környezetre (`www.twistedmatrix.com`).

Gyakorlatok

A következő gyakorlatokban a korábbiakban elkészített ügyfél- és kiszolgálóprogramokat kell módosítanunk. Mindehhez kevesebb gépelésre, de kicsit több furfangra lesz szükség.

1. Készítsünk másolatot a `car_registration_server.py` és a `car_registration.py` fájlokról, majd módosítsuk őket úgy, hogy az adatcseréhez használt protokoll esetében hálózati szintű változatkezelést alkalmazzanak. Ezt megtehetjük például úgy, hogy két egész számot (hossz, protokollváltozat) adunk át a struktúrában egy helyett.
   A feladat végrehajtásához nagyjából tíz sor beillesztésére, illetve módosítására lesz szükség az ügyfélprogram `handle_request()` függvényében, valamint körülbelül tizenhat sor beillesztésére, illetve módosítására a kiszolgáló `handle()` tagfüggvényében. Mindebbe beleértjük azt a kódrészletet is, amely gondoskodik arról az esetről, amikor a beolvasott protokollváltozat nem egyezik a várttal.
   Ennek és az ezt követő feladatoknak egy lehetséges megoldását megtalálhatjuk a `car_registration_ans.py` és a `car_registration_server_ans.py` fájlokban.
2. Készítsünk másolatot a `car_registration_server.py` programról (vagy használjuk az 1. feladatban elkészült változatot), és tegyünk benne elérhetővé egy új, `GET_LICENSES_STARTING_WITH` nevű műveletet, amely egyetlen paramétert, egy karakterláncot vár, és megkeres minden olyan rendszámot, amely ezzel a karakterlánccal kezdődik. A műveletet megvalósító tagfüggvénynek mindig egy (`True`,

rendszámok listája) rendezett párt kell visszaadnia – ezúttal nem következhet be hiba (`False`), ugyanis ha nincs találat, az nem hibát jelent: ilyenkor az eredmény egyszerűen a `True` és egy üres lista.
A rendszámok (vagyis a `RequestHandler.Cars` szótár kulcsai) kiolvasását zárolás hatálya alatt végezzük, a többi műveletet azonban a zárolás hatókörén kívül helyezzük el, a blokkolás időtartamának leszorítása érdekében. A megfelelő rendszámok megkeresésének egyik igen hatékony módszere, ha rendezzük a kulcsokat, majd a `bisect` modullal megkeressük az első egyezést, és innen haladunk tovább. Választhatjuk azt a megközelítést is, amelynek során egymás után végighaladunk a rendszámokon, és tároljuk azokat, amelyek az adott karakterlánccal kezdődnek, például listaértelmezés használatával.
Az új `import` műveleten túl a `Call` osztályban szükségünk lesz pár új sorra a művelet megvalósításához. A műveletet levezénylő tagfüggvény tíznél kevesebb sorban megírható – nem különösebben nehéz feladat, de azért oda kell figyelnünk. Egy lehetséges megoldást (amely a `bisect` modult alkalmazza) találhatunk a `car_registration_server_ans.py` fájlban.

3. Készítsünk másolatot a `car_registration.py` programról (vagy használjuk az 1. feladatban elkészült változatot), és módosítsuk úgy, hogy kihasználja az új kiszolgáló (`car_registration_server_ans.py`) lehetőségeit. Mindehhez meg kell változtatnunk a `retrieve_car_details()` függvényt úgy, hogy ha a felhasználó ismeretlen rendszámot ír be, lehetővé tegyük a számára, hogy egy a rendszám kezdetének beírása után kapott listából válasszon. Íme egy lehetséges párbeszéd a felhasználó és az ügyfél között. (Feltételezzük, hogy a kiszolgáló már fut, továbbá némiképp átszerkesztettük a menüt, hogy elférjen az oldalon, a felhasználó válaszait pedig félkövérrel emeltük ki.)

```
(C)ar  (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License: da 4020
License: DA 4020
Seats:   2
Mileage: 97181
Owner:   Jonathan Lynn
(C)ar  (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License [DA 4020]: z
This license is not registered
Start of license: z
No licence starts with Z
Start of license: a
(1) A04 4HE
(2) A37 4791
(3) ABK3035
Enter choice (0 to cancel): 3
License: ABK3035
Seats:   5
Mileage: 17719
Owner:   Anthony Jay
```

A módosításhoz egy sor törlésére, és nagyjából húsz új sor beillesztésére van szükség. A feladat nehézségét az adja, hogy lehetővé kell tennünk, hogy a felhasználó bármely szinten kiléphessen, vagy folytathassa a keresést. Fontos, hogy az új lehetőséget minden elképzelhető esetben ellenőrizzük (nincs olyan rendszám, amely a megadott karakterlánccal kezdődik; egy ilyen van; illetve kettő vagy több találat adódik).
Egy lehetséges megoldást a `car_registration_ans.py` fájlban találhatunk.

11

Adatbázis-programozás

- DBM-adatbázisok
- SQL-adatbázisok

A legtöbb programfejlesztő számára az *adatbázis* kifejezés valójában relációsadatbázis-kezelő rendszert (RDBMS, Relational Database Management System) jelent, amelyben táblák (vagyis a táblázatokhoz hasonló rácsok) szerepelnek, amelyek sorai a bejegyzéseknek, oszlopai pedig a mezőknek felelnek meg. A táblákat és az azokat megtöltő adatokat az SQL segítségével hozhatjuk létre és kezelhetjük. A Python biztosít egy API-t (alkalmazásprogramozási felületet) az SQL-adatbázisok kezeléséhez, emellett a nyelv szokásos programcsomagjában hozzájutunk az SQLite 3 adatbázishoz is.

Az adatbázisok másik típusa a *DBM* (Database Manager, adatbázis-kezelő), amely tetszőleges számú kulcs-érték párt tartalmazhat. A Python alapkönyvtára felületet biztosít számos DBM-hez, amelyek között kifejezetten Unix rendszereken működőket is találhatunk.
A DBM-ek működése hasonló a Pythonban megismert szótárakéhoz, azzal a különbséggel, hogy a memória helyett jellemzően a merevlemezen érhetjük el őket, kulcsaik és értékeik pedig minden esetben `bytes` objektumok, amelyek hosszára olykor korlátozások állhatnak fenn. A fejezetünk első részében bemutatott `shelve` modul kényelmes DBM-felületet ad, amely lehetővé teszi, hogy kulcsok gyanánt karakterláncokat, értékekként pedig (befőzhető) objektumokat szerepeltessünk.

Ha a rendelkezésre álló adatbázis-kezelők és az SQLite adatbázis nem bizonyulnak elegendőnek, a Python Package Indexen (`pypi.python.org/pypi`) számos adatbázisokhoz kapcsolódó programcsomagot találhatunk, például a bsddb-t („Berkeley-adatbázis”), továbbá olyan objektumrelációs leképezőket, mint az SQLAlchemy (`www.sqlalchemy.org`), de kaphatunk felületeket az ismertebb ügyfél-kiszolgáló adatbázisokhoz is, mint a DB2, az Informix, az Ingres, a MySQL, az ODBC, valamint a PostgreSQL.

Fejezetünkben egy DVD-ket nyilvántartó adatbázis két változatát készítjük el, amely a lemezek listájában tárolja a DVD-k címét, kiadási évét, percben mért hosszát, valamint a film rendezőjét. Az első változatban egy DBM-et használunk (a `shelve` modul révén)

az adatok tárolására, míg a másodikban az SQLite adatbázis szolgáltatásait vesszük igénybe. Mindkét program lehetőséget ad arra, hogy az adatainkat egyszerű XML formátumban tároljuk és töltsük be, így a DVD-k adatait könnyedén átvihetjük egyik programból a másikba. Az SQL alapú változat lehetőségei némiképp tágabbak, mint a DBM-re épülő társánál elérhetők, és az adatok szerkezete is valamelyest átláthatóbb.

DBM-adatbázisok

A `shelve` modul burkolót biztosít egy DBM köré, amely lehetővé teszi, hogy úgy érintkezzünk vele, mintha egy szótárral tennénk – amennyiben kizárólag karakterláncokat használunk kulcsok és befőzhető objektumokat értékek gyanánt. A háttérben a modul a kulcsokat és értékeket `bytes` objektumokká alakítja, és viszont.

Mivel a `shelve` modul az elérhető legjobb háttér-DBM-et használja, előfordulhat, hogy az egyik gépen mentett DBM-fájlt egy másik gép programja nem képes olvasni, ugyanis az nem rendelkezik ugyanazzal a DBM-mel, mint a másik gépen futó társa. Gyakori megoldás erre, hogy lehetővé teszik az XML-fájlok ki- és bevitelét, amelyeknek a formátuma minden rendszeren értelmezhető. Jelen esetben, a `dvds-dbm.py` programban éppen ezt tettük.

A kulcsok szerepét a DVD-k címei játsszák, az értékek pedig a rendezőt, a megjelenés évét, valamint a film hosszát tartalmazó rendezett sorozatok. A `shelve` modulnak köszönhetően nem kell az adatok átalakításával foglalkoznunk – a DBM objektumot pontosan úgy kezelhetjük, mint egy szótárat.

Mivel a program szerkezete hasonló a korábban látott interaktív, menüvezérelt programokéhoz, most csak azokra a részeire összpontosítunk, amelyek részt vesznek az adatbáziskezelésben. Az alábbiakban ezért csak egy részletet mutatunk be a `main()` függvényből, amelyből kihagytuk a menükezelés kódját:

```
db = None
try:
    db = shelve.open(filename, protocol=pickle.HIGHEST_PROTOCOL)
    ...
finally:
    if db is not None:
        db.close()
```

Itt megnyitjuk (illetve, ha még nem létezik, létrehozzuk) a megadott DBM-fájlt írásra és olvasásra. Az egyes elemek értékeit befőttként tároljuk a megadott befőttprotokollal – a már meglevő elemek akkor is beolvashatók, ha korábbi protokollal mentettük őket, ugyanis olvasás esetén a Python képes önállóan kitalálni az alkalmazott protokollt. Ha mindennel elkészültünk, bezárjuk a DBM-et – vagyis kiürítjük a DBM belső átmeneti tárát, meggyőződünk róla, hogy a lemezen tárolt fájlba bekerültek a módosítások, majd bezárjuk a fájlt.

A program lehetővé teszi, hogy beillesszük, szerkesszük, felsoroljuk, töröljük, betöltsük és mentsük a DVD-k adatait. Az XML formátumú be- és kivitel leírásától most eltekintünk, hiszen hasonló feladattal már találkoztunk a 7. fejezetben. Mindemellett a beillesztésen kívül elhagyjuk a felület kódjának többségét, ugyanis ezeket is láthattuk már más környezetben.

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director = Console.get_string("Director", "director")
    if not director:
        return
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   minimum=0, maximum=60*48)
    db[title] = (director, year, duration)
    db.sync()
```

Ez a függvény, hasonlóan a menüből meghívott társaihoz, egyedüli paraméterként egy `db` nevű DBM-objektumot vár. Kódjának nagy részében a DVD adatainak beolvasásával foglalkozik, az utolsó előtti sor pedig tárolja a kapott kulcs-érték elemet a DBM-fájlban, ahol a DVD címe szerepel kulcsként, az érték pedig a rendező, a megjelenés éve és a hossz hármasa egy a `shelve` által előállított befőttben.

A Python szokásos egységesítési lehetőségeinek folyományaként a DBM-ek ugyanazt az API-t adják, mint a szótárak, így nem találkozhatunk sok újdonsággal a korábban már látott `shelve.open()` függvényen, valamint a `shelve.Shelf.sync()` tagfüggvényen kívül, amellyel kiürítjük a `shelve` belső átmeneti tárát, és összhangba hozzuk a lemezen tárolt adatokat a módosításokkal – ez esetben egy új elem beillesztéséről van szó.

```
def edit_dvd(db):
    old_title = find_dvd(db, "edit")
    if old_title is None:
        return
title = Console.get_string("Title", "title", old_title)
if not title:
    return
director, year, duration = db[old_title]
...
db[title] = (director, year, duration)
if title != old_title:
    del db[old_title]
db.sync()
```

Ahhoz, hogy a felhasználó szerkeszthessen egy DVD-t, először ki kell választania, hogy melyikhez szeretne hozzáférni. Itt mindössze a címét kell megadnia, hiszen a címek jelentik a kulcsokat, míg a hozzájuk tartozó értékek adják meg a további adatokat. Mivel az itt alkalmazandó kódra máshol is szükség van (nevezetesen a DVD-k eltávolításánál), ezt egy `find_dvd()` nevű függvényben különítettük el, amelyet az alábbiakban mutatunk be.
Ha megtaláltuk a DVD-t, beolvassuk a felhasználó módosításait, a már létező adatokat alkalmazva alapértelmezésként, hogy némiképpen felgyorsítsuk a párbeszédet. (A felhasználói felület kezelésének kódját nagyrészt elhagytuk, ugyanis ez szinte megegyezik a DVD beillesztésénél látottakkal.) Ha mindennel végeztünk, tároljuk az adatokat, mint a DVD beillesztésénél. Amennyiben a cím változatlan, felülírjuk a hozzá tartozó értéket, ha pedig megváltozott, új kulcs-érték párt hozunk létre, és töröljük az eredetit.

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    while True:
        matches = []
        start = Console.get_string(message, "title")
        if not start:
            return None
        for title in db:
            if title.lower().startswith(start.lower()):
                matches.append(title)
        if len(matches) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(matches) == 1:
            return matches[0]
        elif len(matches) > DISPLAY_LIMIT:
            print("Too many dvds start with {0}; try entering "
                  "more of the title".format(len(matches)))
            continue
        else:
            for i, match in enumerate(sorted(matches, key=str.lower)):
                print("{0}: {1}".format(i + 1, match))
            which = Console.get_integer("Number (or 0 to cancel)",
                                        "number", minimum=1, maximum=len(matches))
            return matches[which - 1] if which != 0 else None
```

A DVD-k gyors és egyszerű felkutatásának érdekében elkérjük a felhasználótól a keresett cím első néhány karakterét, majd végighaladunk az adatbázison, egyezéseket keresve. Ha egyetlen találat van, egyszerűen visszaadjuk. Ha több egyezést is találunk (de kevesebbet, mint a `DISPLAY_LIMIT` értéke, amit másutt adunk meg a programban), kiírjuk őket a képernyőre a betűállást figyelmen kívül hagyó sorrendben, mindegyiküket egy számmal azonosítva – a felhasználó így egy szám beírásával választhatja ki a megfelelő címet. (A `Console.get_integer()` függvény elfogadja a 0 értéket még akkor is, ha a minimum nullánál nagyobb, így a 0-t használhatjuk kilépési érték gyanánt. Ezt a viselkedést

kikapcsolhatjuk az `allow_zero=False` átadásával. Az ENTER lenyomásával – ha nem írunk be semmit – nem érhetjük el ugyanezt a hatást, hiszen ez eleve azt jelenti, hogy elfogadjuk az alapértelmezést.)

```
def list_dvds(db):
    start = ""
    if len(db) > DISPLAY_LIMIT:
        start = Console.get_string("List those starting with "
                                   "[Enter=all]", "start")
    print()
    for title in sorted(db, key=str.lower):
        if not start or title.lower().startswith(start.lower()):
            director, year, duration = db[title]
            print("{0} ({1}) {2} minute{3}, by {4}".format(
                title, year, duration, Util.s(duration), director))
```

Az összes DVD felsorolása (illetve azoké, amelyek egy adott karakterlánccal kezdődnek) egyszerűen abból áll, hogy sorra vesszük az adatbázis elemeit.

A `Util(s)` függvény kódja egyszerűen `s = lambda x: "" if x == 1 else "s"`; így a visszatérési érték az „`s`“, amennyiben az `x` értéke nem éppen egy perc.

```
def remove_dvd(db):
    title = find_dvd(db, "remove")
    if title is None:
        return
    ans = Console.get_bool("Remove {0}?".format(title), "no")
    if ans:
        del db[title]
        db.sync()
```

Egy adott DVD eltávolításához keressük meg, hogy melyiket választotta a felhasználó, kérjünk tőle megerősítést, majd ha valóban törölni akarja, távolítsuk el az elemet a DBM-ből.

Láttuk tehát, miként nyithatunk meg (vagy hozhatunk létre) egy DBM-fájlt a `shelve` modul segítségével, hogyan illesszünk be elemeket ebbe az adatbázisba, miként szerkesszük, vegyük sorra, illetve töröljük őket. Sajnos az adatszerkezetünkben rejlik egy apró hiba. A rendezők nevei több változatban is szerepelhetnek, ami könnyen ellentmondásokhoz vezethet – például előfordulhat, hogy Danny DeVito nevét egyszer „Danny De Vito”, egy másik filmnél pedig „Danny deVito” alakban adják meg. Az egyik lehetséges megoldás, ha két DBM-fájlt hozunk létre: egy központi DVD-tárolót, amelyben a címek a kulcsok, és az (év, hossz, rendezőazonosító) hármasok az értékek, továbbá egy rendezőket tartalmazó adatbázist, amelyben a rendezők azonosítói (egész számok) a kulcsok, a neveik pedig az értékek. Ezeket a gondokat orvosoljuk programunk következőkben bemutatott, SQL-adatbázist alkalmazó változatában is, amelyben két tábla szerepel: egy a DVD-k, egy pedig a rendezők számára.

SQL-adatbázisok

A legtöbb népszerű SQL-adatbázishoz létezik külső fejlesztőktől származó modul, a Pythonnal pedig hozzájutunk az `sqlite3` modulhoz (és az SQLite 3 adatbázishoz), így hát, ha kedvünk tartja, minden különösebb előkészület nélkül nekikezdhetünk az adatbázis-programozásnak. Az SQLite-ról tudnunk kell, hogy könnyűsúlyú SQL-adatbáziskezelő, és mint ilyen, számos lehetőség hiányzik belőle, ami például a PostgreSQL-ben elérhető, de igen kényelmes a prototípus-készítéshez, és sok esetben tökéletesen elegendő a saját céljainkra.

Annak érdekében, hogy könnyen válthassunk a háttéradatbázisok között, a PEP 249 (Python Database API Specification v2.0) meghatároz egy a DB-API 2.0 nevű API-szabványt, amelyet minden adatbázis-felületnek követnie illik – az `sqlite3` modul megteszi ezt a szívességet, de a külső modulok nem feltétlenül. Az API két központi objektumot – nevezetesen egy kapcsolati és egy kurzorobjektumot – határoz meg, amelyeknek a 11.1. és a 11.2. táblázatban bemutatott felületeket kell támogatniuk. Az `sqlite3` esetében a két objektum a szabványban megköveteltek mellett számos egyéb jellemzőt és tagfüggvényt is a rendelkezésünkre bocsát.

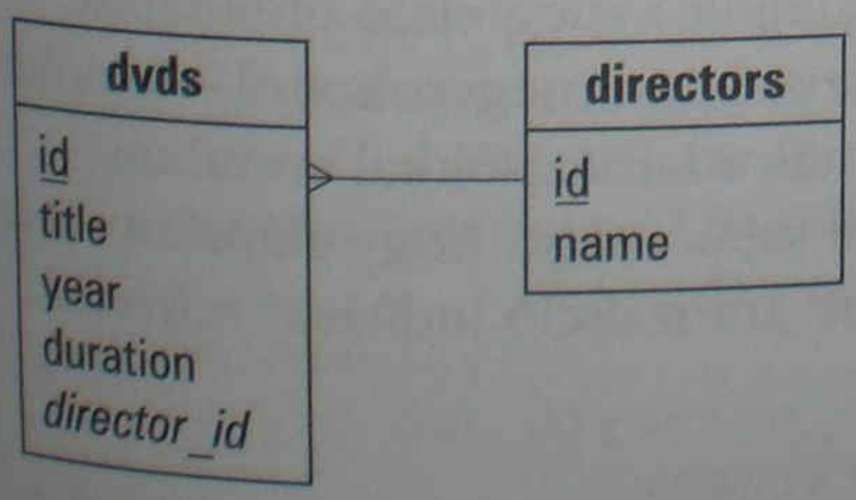
DVD-nyilvántartó programunk SQL-változata, amely a `dvds-sql.py` névre hallgat, a DVD-k adataitól elkülönítve tárolja a rendezőket a névváltozatok megjelenésének elkerülésére, és egy újabb menüponttal lehetővé teszi a rendezők listázását. A két táblát a 11.1. ábrán figyelhetjük meg. Programunk közel 300 soros, míg az előző részben bemutatott `dvds-dbm.py` kódja nem érte el a 200 sort. A különbség nagyrészt annak tudható be, hogy ezúttal a szótárműveletek helyett SQL-lekérdezéseket kell használnunk, és a program első futtatásakor létre kell hoznunk a táblákat.

11.1. táblázat *DB-API 2.0 kapcsolati objektumának tagfüggvényei*

Használati alak	Leírás
`db.close()`	Lezárja a kapcsolatot az adatbázissal (ezt a `db` objektum jeleníti meg, amelyhez a `connect()` függvény hívásával jutottunk hozzá).
`db.commit()`	Jóváhagyja az összes függő tranzakció eredményét az adatbázisban – abban az esetben, ha az adatbázis nem támogatja a tranzakciókat, a tagfüggvény nem végez semmilyen műveletet.
`db.cursor()`	Visszaad egy kurzorobjektumot az adatbázishoz, amelynek révén lekérdezéseket hajthatunk végre.
`db.rollback()`	Visszagörgeti az összes függőben levő tranzakciót a kezdeti állapotba – abban az esetben, ha az adatbázis nem támogatja a tranzakciókat, a tagfüggvény nem végez semmilyen műveletet.

11.2. táblázat *A DB-API 2.0 kurzorobjektumának jellemzői és tagfüggvényei*

Használati alak	Leírás
`c.arraysize`	A sorok (írható/olvasható) száma, amelyet a `fetchmany()` akkor ad vissza, ha nem határoztunk meg méretet.
`c.close()`	Bezárja a `c` kurzort. Ez automatikusan megtörténik, amikor a kurzor kikerül a hatókörből.
`c.description`	Hételemű rendezett sorozatok csak olvasható sora, amelynek elemei (`name`, `type_code`, `display_size`, `internal_size`, `precision`, `scale`, `null_ok`) alakban írják le a `c` kurzor egymást követő oszlopait.
`c.execute(sql, paraméterek)`	Végrehajtja az `sql` karakterláncban megadott SQL-lekérdezést, amelyben a helykitöltőket az átadott *`paraméterek`* sorozat vagy megfeleltetés elemeivel helyettesíti.
`c.executemany(sql, paramétersorozat)`	Végrehajtja az `sql` karakterláncban megadott SQL-lekérdezést sorozatok sorozatának minden elemével. Nem alkalmazhatjuk olyan műveletek esetében, amelyek eredményhalmazt adnak (ilyenek a `SELECT` utasítások).
`c.fetchall()`	Visszaadja azoknak a soroknak a sorozatát, amelyeket még nem olvastunk ki (itt szó lehet akár az összes sorról is).
`c.fetchmany(méret)`	Sorok sorozatát adja vissza (az egyes sorok önmagukban is sorozatok). A *`méret`* alapértelmezett értéke a `c.arraysize`.
`c.fetchone()`	Visszaadja a lekérdezés eredményhalmazának következő sorát, illetve a `None` értéket, ha az eredményhalmaz végére ért. Amennyiben nincs eredményhalmaz, kivételt vált ki.
`c.rowcount`	A legutóbbi műveletben (`SELECT`, `INSERT`, `UPDATE` vagy `DELETE`) visszakapott sorok számának csak olvasható értéke, illetve -1, amennyiben az érték nem érhető el, vagy nem meghatározható.



11.1. ábra
A DVD-nyilvántartó program adatbázisának felépítése

A `main()` függvény felépítése hasonló az előzőekben látotthoz, de ezúttal az egyedi `connect()` függvénnyel kapcsolódunk az adatbázishoz.

```
def connect(filename):
    create = not os.path.exists(filename)
    db = sqlite3.connect(filename)
    if create:
```

```
        cursor = db.cursor()
    cursor.execute("CREATE TABLE directors ("
        "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
        "name TEXT UNIQUE NOT NULL)")
    cursor.execute("CREATE TABLE dvds ("
        "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
        "title TEXT NOT NULL, "
        "year INTEGER NOT NULL, "
        "duration INTEGER NOT NULL, "
        "director_id INTEGER NOT NULL, "
        "FOREIGN KEY (director_id) REFERENCES directors)")
        db.commit()
    return db
```

Az `sqlite3.connect()` függvény létrehoz egy adatbázisobjektumot, miután megnyitotta a számára átadott adatbázisfájlt, illetve, amennyiben az még nem létezett, létrehozta. Ezért, mielőtt meghívnánk ezt a tagfüggvényt, feljegyezzük, hogy az adatbázisunkat a kezdetektől építjük-e fel, ugyanis ez esetben létre kell hoznunk azokat a táblákat, amelyekre az adatbázis működése épül. A lekérdezéseket az adatbázis kurzorobjektumán végezhetjük el, amelyhez az adatbázisobjektum `cursor()` tagfüggvényével juthatunk hozzá.

Vegyük észre, hogy mindkét táblát olyan ID mezővel hoztuk létre, amelyre érvényes az `AUTOINCREMENT` megszorítás. Ez azt jelenti, hogy az SQLite automatikusan egyedi értékekkel tölti fel ezt az oszlopot, így új bejegyzések beillesztésénél ennek a mezőnek a kitöltését ráhagyhatjuk a rendszerre.

Az SQLite csak bizonyos adattípusokat támogat – lényegében csak logikai értékeket, számokat és karakterláncokat használhatunk –, de az elérhető típusok skálája bővíthető az úgynevezett „átalakítók" segítségével, legyen szó beépített átalakítókról, mint a dátum és a dátum-idő értékeknél, illetve egyediekről, amelyekkel olyan adattípust jeleníthetünk meg, amilyet csak akarunk. A DVD-nyilvántartó programhoz nincs szükség erre a lehetőségre, de ha élnénk vele, az `sqlite3` leírásában megtaláljuk a megfelelő útmutatást. Az idegen kulcsok használati alakja eltér a más adatbázisokban megszokottól – leginkább a szándékainkat hivatottak jelezni, ugyanis az SQLite más adatbázisokkal szemben kikényszeríti a kapcsolati épséget. Az `sqlite3` további sajátossága, hogy alapállapotban támogatja a beleértett tranzakciókat, így nincs kifejezett „tranzakció indítása" művelet.

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director = Console.get_string("Director", "director")
    if not director:
        return
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
```

```
duration = Console.get_integer("Duration (minutes)", "minutes",
                               minimum=0, maximum=60*48)
director_id = get_and_set_director(db, director)
cursor = db.cursor()
cursor.execute("INSERT INTO dvds "
               "(title, year, duration, director_id) "
               "VALUES (?, ?, ?, ?)",
               (title, year, duration, director_id))
db.commit()
```

A függvény kódjának első sorai megegyeznek a `dvds-dbm.py` programbeli társánál látottakkal, de amint összegyűjtöttük az adatokat, a működése más irányt vesz. A felhasználó által megadott rendező szerepelhet a `directors` táblában, de megeshet az is, hogy nincs jelen – a `get_and_set_director()` függvény nagyszerűen kezeli a helyzetet, hiszen utóbbi esetben beilleszti a rendező nevét az adatbázisba, és bármi történjen is, visszaadja a rendező azonosítóját, amelyet azután elhelyezhetünk a `dvds` táblában. Ha minden adat a rendelkezésünkre áll, végrehajtunk egy `INSERT` SQL-utasítást. A bejegyzés azonosítóját nem nekünk kell megadnunk, hiszen az SQLite automatikusan szolgáltatja.

A lekérdezésben helykitöltőként kérdőjeleket használtunk, amelyek mindegyikének helyére az SQL-utasítás karakterlánca után álló sorozat megfelelő eleme kerül. Mindemellett névvel ellátott helykitöltőket is használhatunk, amire a bejegyzések szerkesztésénél láthatunk majd példát. Jóllehet módunk van arra is, hogy elkerüljük a helykitöltők használatát, és magunk formázzuk az SQL-karakterláncot a megfelelő adatokkal együtt, mégis ajánlatos helykitöltőket alkalmaznunk, ugyanis így az adatbázismodul leveszi a vállunkról az adatelemek kódolásának és levédésének terhét. A helykitöltők használatának további előnye a megnövekedett biztonság, hiszen ez meggátolja a rosszindulatú behatolókat abban, hogy tetszőleges SQL-kódot helyezzenek el a lekérdezéseinkben.

```
def get_and_set_director(db, director):
    director_id = get_director_id(db, director)
    if director_id is not None:
        return director_id
    cursor = db.cursor()
    cursor.execute("INSERT INTO directors (name) VALUES (?)",
                   (director,))
    db.commit()
    return get_director_id(db, director)
```

A fenti függvény visszaadja a paraméterként meghatározott rendező azonosítóját, szükség esetén létrehozva számára egy új bejegyzést az adatbázisban. Ha beillesztettünk egy bejegyzést, az azonosítóját a `get_director_id()` függvénnyel olvassuk ki – ahogy ezzel a függvény első sorában is megpróbálkoztunk.

```
def get_director_id(db, director):
    cursor = db.cursor()
    cursor.execute("SELECT id FROM directors WHERE name=?",
```

```
                    (director,))
    fields = cursor.fetchone()
    return fields[0] if fields is not None else None
```

A `get_director_id()` függvény a megadott rendező azonosítójával tér vissza, illetve ha a rendező nem szerepel az adatbázisban, a `None` értéket adja. A kiolvasáshoz a `fetchone()` tagfüggvényt használjuk, hiszen tudjuk, hogy legfeljebb egy találatot kapunk. (Egyetlen rendező neve sem szerepelhet többször, mivel a `directors` tábla `name` mezőjére a `UNIQUE` megszorítás érvényes, ráadásul új rendező beillesztése előtt mindig megvizsgáljuk, hogy szerepel-e már az adatbázisban.) A kiolvasó tagfüggvények mindig mezők sorozatát adják vissza (illetve a `None` értéket, ha az eredményhalmaz végére értek), még akkor is, ha egyébként egyetlen mezőt szeretnénk kiolvasni, mint a mi esetünkben.

```
def edit_dvd(db):
    title, identity = find_dvd(db, "edit")
    if title is None:
        return
    title = Console.get_string("Title", "title", title)
    if not title:
        return
    cursor = db.cursor()
    cursor.execute("SELECT dvds.year, dvds.duration, directors.name "
                   "FROM dvds, directors "
                   "WHERE dvds.director_id = directors.id AND "
                   "dvds.id=:id", dict(id=identity))
    year, duration, director = cursor.fetchone()
    director = Console.get_string("Director", "director", director)
    if not director:
        return
    year = Console.get_integer("Year", "year", year, 1896,
                               datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   duration, minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor.execute("UPDATE dvds SET title=:title, year=:year, "
                   "duration=:duration, director_id=:director_id "
                   "WHERE id=:id", locals())
    db.commit()
```

Ahhoz, hogy szerkesszük az adott DVD-hez tartozó bejegyzést, mindenekelőtt meg kell találnunk. Ha megvan, először lehetővé tesszük, hogy a felhasználó megváltoztassa a címét. Ezt követően kiolvassuk a többi mező tartalmát, így alapértelmezett értékekkel tehetjük kényelmesebbé a szerkesztést – ha a felhasználó leüti az ENTER-t, egyszerűen elfogadja az előző, alapértelmezett értéket. Ez esetben névvel ellátott helykitöltőket használtunk (a `:név` alakban), így a megfelelő értékeket egy megfeleltetés alakjában kell biztosítanunk. A `SELECT` utasításoknál egy frissen létrehozott szótárat alkalmazunk, az `UPDATE` esetében pedig ahhoz a szótárhoz folyamodunk, amelyet a `locals()` meghívásával kaptunk.

Használhattunk volna új szótárat mindkét esetben, de ekkor az UPDATE műveletnek a `locals()` eredménye helyett a `dict(title=title, year=year, duration=duration, director_id=director_id, id=identity)` értéket kellett volna átadnunk.

Ha megkaptuk minden mező értékét, és a felhasználó elvégezte a kívánt módosításokat, kiolvassuk a megfelelő rendezőazonosítót (illetve szükség esetén új bejegyzést hozunk létre a rendező számára), majd frissítjük az adatbázist az új adatokkal. A lehető legegyszerűbb megoldás mellett döntöttünk, így az összes mezőt frissítjük, nem csak azokat, amelyek megváltoztak.

A DBM-fájlra épülő megoldásban a DVD címét alkalmaztuk kulcsként, így ha ez megváltozott, létre kellett hoznunk egy új kulcs-érték párt, és törölnünk az eredetit. Ez esetben azonban minden DVD-bejegyzés egyedi azonosítóval rendelkezik, amelyet a beillesztésekor kapott, így minden további erőfeszítés nélkül megváltoztathatjuk bármelyik mező értékét.

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    cursor = db.cursor()
    while True:
        start = Console.get_string(message, "title")
        if not start:
            return (None, None)
        cursor.execute("SELECT title, id FROM dvds "
                       "WHERE title LIKE ? ORDER BY title",
                       (start + "%"))
        records = cursor.fetchall()
        if len(records) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(records) == 1:
            return records[0]
        elif len(records) > DISPLAY_LIMIT:
            print("Too many dvds ({0}) start with {1}; try entering "
                  "more of the title".format(len(records), start))
            continue
        else:
            for i, record in enumerate(records):
                print("{0}: {1}".format(i + 1, record[0]))
            which = Console.get_integer("Number (or 0 to cancel)",
                                        "number", minimum=1, maximum=len(records))
            return records[which - 1] if which != 0 else (None, None)
```

Ez a függvény ugyanazt a feladatot végzi el, mint a `dvds-dbm.py` program `find_dvd()` függvénye, eredményként pedig egy (cím, DVD azonosító) rendezett párt ad, ha a keresés sikeres volt, egyébként pedig a `(None, None)` párhoz jutunk. Ezúttal azonban használhatjuk az SQL % műveletét, így csak azokat a bejegyzéseket olvassuk ki, amelyekre valóban szükségünk van.

Mivel arra számítunk, hogy a találatok száma kicsi lesz, az összeset egyszerre olvassuk ki sorozatok sorozata alakjában. Amennyiben a megfelelő bejegyzések száma egynél több, viszont nem túlzottan sok a megjelenítéshez, mindegyiküket kiírjuk egy-egy számmal azonosítva, amelyeket a felhasználó kiválaszthat – hasonlóan a `dvds-dbm.py` programban látottakhoz.

```
def list_dvds(db):
    cursor = db.cursor()
    sql = ("SELECT dvds.title, dvds.year, dvds.duration, "
           "directors.name FROM dvds, directors "
           "WHERE dvds.director_id = directors.id")
    start = None
    if dvd_count(db) > DISPLAY_LIMIT:
        start = Console.get_string("List those starting with "
                                   "[Enter=all]", "start")
        sql += " AND dvds.title LIKE ?"
    sql += " ORDER BY dvds.title"
    print()
    if start is None:
        cursor.execute(sql)
    else:
        cursor.execute(sql, (start + "%",))
    for record in cursor:
        print("{0[0]} ({0[1]}) {0[2]} minutes, by {0[3]}".format(
              record))
```

Az egyes DVD-k adatainak kiírásához egy `SELECT` lekérdezést végzünk, amely egyesíti a két táblát, egy második elemet is elhelyezve a `WHERE` záradékban, amennyiben a bejegyzések száma (ezt a `dvd.count()` függvénnyel kaphatjuk meg) több, mint amennyi megjeleníthető. Ezt követően végrehajtunk egy lekérdezést, és végighaladunk az eredményeken. A kapott bejegyzések olyan sorozatok, amelyeknek a mezői egyezést adnak a `SELECT` lekérdezéssel.

```
def dvd_count(db):
cursor = db.cursor()
cursor.execute("SELECT COUNT(*) FROM dvds")
return cursor.fetchone()[0]
```

Ezeket a kódsorokat egy önálló függvényben helyeztük el, mivel a program több pontján is szükségünk van rájuk.

A `list_directors()` függvény kódját nem mutatjuk be, mivel szerkezetében igen hasonló a `list_dvds()` függvényhez, mindössze egyszerűbb, hiszen csak egy mezőt (ez pedig a `name`) ír ki.

```
def remove_dvd(db):
    title, identity = find_dvd(db, "remove")
    if title is None:
```

```
        return
    ans = Console.get_bool("Remove {0}?".format(title), "no")
    if ans:
        cursor = db.cursor()
        cursor.execute("DELETE FROM dvds WHERE id=?", (identity,))
```

Ezt a függvényt akkor hívjuk meg, ha a felhasználó törölni kíván egy bejegyzést. A szerkezete nagyban hasonlít a `dvds-dbm.py` programbeli megfelelőjéhez.

Ezzel végére értünk a `dvds-sql.py` program bemutatásának, amelynek során megtanultuk, hogy miként hozhatunk létre adatbázis-táblákat, hogyan választhatunk ki bejegyzéseket és vehetjük sorra őket, továbbá miként illesszük be, frissítsük vagy töröljük a bejegyzéseket. Az `execute()` tagfüggvénnyel tetszőleges SQL-utasítást végrehajthatunk, amelyet a háttérben működő adatbázis támogat.

Az SQLite az itt szükségesnél jóval többre képes, így alkalmazhatjuk az automatikus jóváhagyási módot (és más tranzakcióvezérlési lehetőségeket), továbbá olyan függvényeket is írhatunk, amelyeket az SQL-kódban futtathatunk. Létrehozhatunk továbbá gyártófüggvényeket is, amelyekkel meghatározhatjuk, hogy milyen alakban kapjuk vissza a kiolvasott bejegyzéseket (a mezők sorozata helyett szótárat vagy egyedi típust is meghatározhatunk így). Ráadásul lehetőséget kapunk arra is, hogy a memóriában tárolt SQLite adatbázisokat használjunk, csak adjuk át a `":memory:"` karakterláncot fájlnév gyanánt.

Összefoglalás

A 7. fejezetben számos módot megismertünk a lemezen tárolt fájlok adatainak mentésére és betöltésére, itt pedig azt is megtudtuk, hogyan kezeljük azokat az adattípusokat, amelyek a tartalmukat a memória helyett lemezen raktározzák.

A DBM-fájlok használatánál a `shelve` modul igen kényelmes megoldást ad, hiszen karakterlánc–objektum párok tárolására alkalmas. Amennyiben teljes mélységében át szeretnénk látni a rendszert, természetesen közvetlenül is hozzáférhetünk a háttérben működő bármely DBM-hez. A `shelve` modul és a DBM-ek nagyszerű tulajdonsága, hogy a szótárak API-ját használják, így az elemek kiolvasása, beillesztése, szerkesztése és törlése gyerekjáték, sőt így azokat a programokat, amelyek szótárakat használnak, könnyedén rávehetjük az adatbázisok elérésére. A DBM-ek használatának apró szépséghibája, hogy a relációs adatok esetében minden kulcs-érték táblához külön DBM-fájlt kell készítenünk, míg az SQLite ezeket az adatokat egyetlen fájlban tárolja.

Az SQL-adatbázisok esetében az SQLite igen hasznos lehet a prototípus-készítésnél, de gyakran önállóan is használható, mindemellett az az előnye mindenképp megvan, hogy megkapjuk a Python szokásos programcsomagjával. Láthattuk, hogyan juthatunk hozzá

az adatbázisobjektumhoz a `connect()` függvénnyel, és miként hajthatunk végre SQL-lekérdezéseket (mint a `CREATE TABLE`, a `SELECT`, az `INSERT`, az `UPDATE` vagy a `DELETE`) az adatbázis kurzorának `execute()` tagfüggvényével.

A Python az adattárolási lehetőségek széles skáláját biztosítja a számunkra, legyen szó memóriabeli vagy merevlemezes tárolásról. A bináris, szöveg- és XML-fájloktól a befőtteken át a DBM- és az SQL-adatbázisokig minden rendelkezésre áll, hogy az adott feladathoz a megfelelő megoldást válasszuk.

Gyakorlat

Készítsünk interaktív konzolprogramot könyvjelzők listájának kezelésére, amelyben minden elem két adatot – egy URL-t és egy nevet – tartalmaz. Lássuk, miként is működhet egy ilyen program:

```
Bookmarks (bookmarks.dbm)
(1) Programming in Python 3..... http://www.qtrac.eu/py3book.html
(2) PyQt........................ http://www.riverbankcomputing.com
(3) Python...................... http://www.python.org
(4) Qtrac Ltd................... http://www.qtrac.eu
(5) Scientific Tools for Python.... http://www.scipy.org
(A)dd (E)dit (L)ist (R)emove (Q)uit [l]: e
Number of bookmark to edit: 2
URL [http://www.riverbankcomputing.com]:
Name [PyQt]: PyQt (Python bindings for GUI library)
```

Tegyük lehetővé a felhasználóknak, hogy új könyvjelzőt illesszenek be, továbbá adjunk lehetőséget a könyvjelzők szerkesztésére, felsorolására és törlésére. A szerkesztéshez, illetve az eltávolításhoz azonosítsuk a könyvjelzőket számokkal, így a felhasználó ezek valamelyikét választva végezheti el a kívánt műveletet. Tároljuk az adatokat egy DBM-fájlban a `shelve` modul segítségével, amelyben a kulcsok szerepét a nevek játsszák, az értékeknek pedig az URL-ek felelnek meg. A program a szerkezetét tekintve igencsak hasonlít majd a `dvds-dbm.py` fájlra, azzal a különbséggel, hogy a `find_dvd()` helyett ezúttal a `find_bookmark()` szerepel, ami jóval egyszerűbb a társánál, hiszen ezúttal mindössze egy egész számot kell beolvasnunk a felhasználótól, és ez alapján megkeresni a megfelelő könyvjelző nevét.

A felhasználók kényelme kedvéért a beillesztett vagy szerkesztett URL előtt helyezzük el a `http://` karakterláncot, amennyiben nem adnak meg semmilyen más protokollt.

A teljes program száznál kevesebb sort igényel (feltéve, hogy igénybe vesszük a `Console` modul szolgáltatásait, köztük a `Console.get_string()` függvényt). Egy lehetséges megoldást megtekinthetünk a `bookmarks.py` fájlban.

12

Szabályos kifejezések

- A Python szabályoskifejezés-nyelve
- A Python szabályoskifejezés-modulja

A szabályos kifejezések olyan tömör jelölések, amelyek karakterláncok egy csoportját írják le. A szabályos kifejezéseket az teszi olyan hatékonnyá, hogy egyetlen szabályos kifejezés korlátlan számú karakterláncot ábrázolhat – feltéve, hogy azok megfelelnek a szabályos kifejezés követelményeinek. A szabályos kifejezéseket (angolul regular expression – ezért általában csak „regex"-nek hívjuk őket) egy mininyelv segítségével határozzuk meg, ami teljesen eltér a Pythontól, a Python részét képező `re` modulon keresztül azonban zökkenőmentesen hozhatunk létre és használhatunk szabályos kifejezéseket.* Szabályos kifejezéseket általában a következő négy célra szoktunk használni:

- Érvényesség-ellenőrzés: annak vizsgálata, hogy egy adott szövegrész megfelel-e bizonyos feltételeknek, például tartalmazza-e egy pénznem jelét, amelyet számjegyek követnek.
- Keresés: olyan rész-karakterláncok keresése, amelyek részben különbözhetnek; például a „pet.png", a „pet.jpg", a „pet.jpeg" vagy a „pet.svg" keresése, de nem a „carpet.png" és hasonló karakterláncoké.
- Keresés és csere: minden olyan szövegrész lecserélése, amelyre a szabályos kifejezés illeszkedik, például hogy a „bicycle" és a „human powered vehicle" karakterláncokból egyaránt „bike" legyen.
- Karakterláncok felbontása: karakterlánc szétvágása minden olyan ponton, amely illeszkedik a szabályos kifejezésre, például minden „:" vagy „=" karakternél.

Legegyszerűbb formájában a szabályos kifejezés egy kifejezés (vagyis literális karakter), amelyet egy elhagyható mennyiségjelző követ. Az összetettebb szabályos kifejezések tetszőleges számú, mennyiségjelzővel ellátott kifejezésből állhatnak, és feltételezések (assertion, „állítás") is szerepelhetnek bennük, valamint jelzőkkel (flag) szabályozhatók.

---

* Az egyik jó könyv, amely a szabályos kifejezésekről szól, Jeffrey E. F. Friedl *Mastering Regular Expressions* című kötete (ISBN 0596528124). A mű nem foglalkozik kifejezetten a Pythonnal, de a Python `re` modulja nagyon hasonló szolgáltatásokat nyújt, mint a Perl nyelvnek a könyvben részletesen ismertetett szabályoskifejezés-motorja.

Fejezetünk első részében bemutatjuk és elmagyarázzuk a szabályos kifejezésekkel kapcsolatos összes fontos fogalmat. Tisztán a szabályos kifejezések nyelvtanával foglalkozunk – magára a Pythonra csak annyiban utalunk, amennyire feltétlenül szükséges. A fejezet második részében aztán a korábban tanultakra építve megmutatjuk, hogy miként használhatók a szabályos kifejezések a Python programozási környezetében. Azok, akik már ismerik a szabályos kifejezéseket, és csak azt szeretnék megtanulni, hogy miként működnek a Pythonban, rögtön a fejezet második felére ugorhatnak. A fejezet a `re` modul által kínált teljes szabályoskifejezés-nyelvet bemutatja, beleértve az összes feltételezést és jelzőt. A szövegben a szabályos kifejezéseket **`félkövérrel`** szedtük, az illeszkedéseket (találatokat) aláhúzással, az elfogott találatokat pedig árnyékolással jelöltük.

A Python szabályoskifejezés-nyelve

Ebben a részben négy témakörre bontva tárgyaljuk a szabályos kifejezések nyelvét. Az első szakaszban azt mutatjuk meg, hogy miként lehet egyes karakterekre (például a-ra vagy b-re) vagy karakterek egy csoportjára (például „*a vagy b*”-re) illeszteni. A második szakaszban az illesztések mennyiségjelzőivel foglalkozunk (például egyszeres illesztés, legalább egyszeres illesztés vagy a lehető legtöbbszöri illesztés). A harmadik szakasz az alkifejezések csoportosításának és az illeszkedő szövegek elfogásának módját mutatja be, a negyedik, utolsó szakasz pedig a nyelv feltételezéseinek és jelzőinek használatát, amelyekkel szabályozhatjuk a szabályos kifejezések működését.

Karakterek és karakterosztályok

A legegyszerűbb kifejezések csupán literális karakterek, mint az `a` vagy az `5`, és ha nem adunk meg kifejezetten mennyiségjelzőt, a nyelv egyszeri illesztésként értelmezi őket. A **`tune`** szabályos kifejezés például négy, beleértett módon egyszeri illesztést meghatározó kifejezésből áll: arra illeszkedik, ha egy *`t`*-t egy *`u`*, azt egy *`n`*, azt pedig egy `e` követi, így a `tune` és az `attuned` karakterláncokra is találatot ad.

Bár a legtöbb karakter használható literálként, vannak „különleges karakterek” is – olyan jelek, amelyek a szabályoskifejezés-nyelv sajátjai, ezért ha literális értelemben szeretnénk használni őket, le kell védenünk egy fordított perjellel (`\`). A különleges karakterek a következők: `\.^$?+*{}[]()|`. A Python szabványos védőkaraktereinek többsége – például az újsor karaktert jelző `\n` vagy a tabulátor karaktert jelző `\t` – ugyancsak használható a szabályos kifejezéseken belül, akárcsak az `\x`*`HH`*, az `\u`*`HHHH`* vagy az `\U`*`HHHHHHHH`* formát követő hexadecimális védőkarakterek.

Sok esetben nem egy bizonyos karaktert keresünk, hanem egy karakterhalmaz bármelyik tagját. Ezt egy *karakterosztály* használatával tehetjük meg, ami nem más, mint egy vagy több, szögletes zárójelek közé zárt karakter. (A karakterosztályoknak semmi közük a Python osztályaihoz, csupán így hívják a szabályos kifejezések nyelvében a karakterek halmazait.) A karakterosztály is kifejezés, és mint ilyen, rá is érvényes, hogy ha nem adunk

meg hozzá mennyiségjelzőt, akkor pontosan egy karakterre illeszkedik (ami a karakterosztályban szereplő karakterek bármelyike lehet). A `r[ea]d` szabályos kifejezés például mind a red, mind a radar szóra találatot ad, de a read-re nem. Ehhez hasonlóan, ha egyetlen számjegyre szeretnénk illeszteni, akkor a `[0123456789]` szabályos kifejezést használhatjuk. A kényelem kedvéért karakterek egy tartományát egy kötőjel segítségével is meghatározhatjuk, így a `[0-9]` szabályos kifejezés szintén egy számjegyre illeszkedik. Ha a nyitó zárójel után egy kalapjelet írunk, a karakterosztályt tagadhatjuk is, tehát a `[^0-9]` szabályos kifejezés minden olyan karakterre illeszkedni fog, ami *nem* számjegy.

Megjegyzendő, hogy a karakterosztályokon belül a \ kivételével a különleges karakterek elveszítik az eredeti (különleges) jelentésüket – bár a ^ esetében azt figyelhetjük meg, hogy a karakter új jelentést (tagadás) kap, ha a karakterosztály első karaktereként szerepel (minden más esetben egyszerűen ^ karakternek minősül). Emellett a – karaktertartományt jelöl, kivéve, ha ez az első karakter, mely esetben egyszerű kötőjelként értelmezendő.

Mivel egyes karakterhalmazokra gyakran szükség van, ezek gyorsírásos alakkal is rendelkeznek, amelyeket a 12.1. táblázatban láthatunk. Egy kivétellel valamennyi gyorsírásos alak a karakterhalmazokon belül használható, tehát a `[\dA-Fa-f]` szabályos kifejezés például bármely hexadecimális számjegyre ad találatot. A kivételt a . jelenti, amely a karakterosztályokon kívül minősül gyorsírásos alaknak, karakterosztályon belül azonban a literális pont (.) karakterre illeszkedik.

12.1. táblázat *Karakterosztályok gyorsírásos alakjai*

Jel	Jelentés
.	Az újsor kivételével bármely karakterre illeszkedik, illetve a `re.DOTALL` jelzővel megadva minden karakterre, karakterosztályon belül azonban a pont (.) karakterre.
\d	Az Unicode-számjegyekre illeszkedik, illetve a `re.ASCII` jelzővel megadva a `[0-9]` tartományra.
\D	A nem számjegy Unicode-karakterekre illeszkedik, illetve a `re.ASCII` jelzővel megadva a `[^0-9]` tartományra.
\s	Az Unicode-üreshelyekre illeszkedik, illetve a `re.ASCII` jelzővel megadva a `[ \t\n\r\f\v]` tartományra.
\S	A nem üreshely Unicode-karakterekre illeszkedik, illetve a `re.ASCII` jelzővel megadva a `[^ \t\n\r\f\v]` tartományra.
\w	Az Unicode-„szókarakterekre” illeszkedik, illetve a `re.ASCII` jelzővel megadva az `[a-zA-Z0-9]` tartományra.
\W	A nem „szókarakter” Unicode-karakterekre illeszkedik, illetve a `re.ASCII` jelzővel megadva a `[^a-zA-Z0-9]` tartományra.

Mennyiségjelzők

A mennyiségjelzők alakja `{m,n}`, ahol az *m* és az *n* azt adják meg, hogy legalább, illetve legfeljebb hány alkalommal kell illeszkednie annak a kifejezésnek, amelyikre a mennyiségjelző vonatkozik. Az **`e{1,1}e{1,1}`** és az **`e{2,2}`** szabályos kifejezések például egyaránt illeszkednek a `feel`-re, de a `felt`-re egyik sem.

Ha minden kifejezés után mennyiségjelzőt kellene írnunk, az hamar fárasztóvá válna, és a kód olvashatóságát is rontaná. Szerencsére a szabályos kifejezések nyelve többféle kényelmes gyorsírásos alakot is támogat. Ha a mennyiségjelzőben csak egy számot adunk meg, akkor úgy veszi, hogy ez a minimum és a maximum is egyben, tehát az **`e{2}`** ugyanazt jelenti, mint az **`e{2,2}`**. Ahogy pedig az előző részben már megjegyeztük, ha nem adunk meg kifejezetten mennyiségjelzőt, a nyelv egyszeri illesztést feltételez (tehát **`{1,1}`**-et vagy **`{1}`**-et) – az **`ee`** tehát ugyanazt jelenti, mint az **`e{1,1}e{1,1}`**, illetve az **`e{1} e{1}`**, ezért az **`e{2}`** és az **`ee`** egyaránt illeszkedik a `feel`-re, de a `felt`-re nem.

Gyakran hasznos más-más minimum- és maximumértéket megadni. Ahhoz például, hogy a `travelled` és a `traveled` szóra egyaránt illeszthessünk (mindkettő egyformán elfogadott írásmód), a **`travel{1,2}ed`** és a **`travell{0,1}ed`** alakot is használhatjuk. A **`{0,1}`** mennyiségjelzőt olyan gyakran használják, hogy saját gyorsírásos alakja van: a kérdőjel (**`?`**). Az előző szabályos kifejezést tehát így is írhatjuk (és a gyakorlatban általában így is szoktak tenni): **`travell?ed`**.

Még két gyorsírásos mennyiségjelző áll rendelkezésre: a pluszjel (**`+`**), amely az **`{1,`***n*`**`} („legalább egy”) alakot helyettesíti, illetve a csillag (**`*`**), amelynek a jelentése **`{0,`***n*`**`} („tetszőleges számú”). Az *n* mindkét esetben az adott mennyiségjelző esetében lehetséges előfordulások maximális száma, ami általában legalább 32 767. A mennyiségjelzőket a 12.2. táblázatban láthatjuk.

A **`+`** mennyiségjelző nagyon hasznos. Az egész számok illesztéséhez például használhatjuk a **`\d+`** alakot is, hiszen ez egy vagy több számjegyre illeszkedik. A `4588.91` karakterláncban például ez a szabályos kifejezés két helyen is illeszkedik: `4588`.`91` és `4588.91`.

Az elírásoknak néha az az oka, hogy egy billentyűt túl sokáig tartunk lenyomva. A **`bevel+ed`** szabályos kifejezéssel kiszűrhetjük a helyes `beveled` és `bevelled` alakokat, míg a helytelen bevellled nem fog illeszkedni. Ha egységesen az egy *l*-es írásmódot szeretnénk használni, és csak azokat az előfordulásokat keressük, amelyekben kettő vagy több l található, a **`bevell+ed`** szabályos kifejezés lehet a segítségünkre.

A **`*`** mennyiségjelzőnek kevésbé vesszük hasznát, egyszerűen azért, mert gyakran nem várt eredményt ad. Tegyük fel például, hogy azokat a sorokat keressük néhány Python-fájlban, amelyek megjegyzéseket tartalmaznak. Erre a célra megpróbálkozhatunk a **`#*`** kifejezéssel, de ez a szabályos kifejezés minden sorra illeszkedni fog, beleértve az üres sorokat is, mert a jelentése „keress tetszőleges számú #-et”, és a „tetszőleges szám”-ba

a nulla is beleértendő. Azoknak, akik csak most ismerkednek a szabályos kifejezésekkel, alapszabályként a * teljes kerülését javasoljuk, ha viszont mégis használnánk (akár a csillagot, akár a ? regexet), akkor gondoskodjunk róla, hogy legalább még egy olyan kifejezés szerepeljen a szabályos kifejezésben, amelynek a mennyiségjelzője nem nulla, vagyis egy olyan mennyiségjelző, amely se nem *, se nem ?, mivel ez a kettő a megadott kifejezés nulla előfordulását is keresi.

Ahol csillagot használnánk, ott gyakran használhatjuk a + mennyiségjelzőt is, és viszont. A legalább egy *l*-t tartalmazó „tasselled" szót például a `tassell*ed` és a `tassel+ed` kifejezéssel is megtaláljuk, míg a kettő vagy több *l*-t tartalmazó előfordulások illesztésére a `tasselll*ed` és a `tassell+ed` is használható.

A `\d+` szabályos kifejezés a 136 karakterláncra is illeszkedik. De miért az összes számjegyre, miért nem csak az elsőre? Nos, a mennyiségjelzők alapértelmezés szerint kivétel nélkül *mohók*, vagyis annyi karakterre illeszkednek, amennyire csak tudnak. A mennyiségjelzők azonban nem mohóvá (úgy is mondják, hogy „*minimálissá*") is tehetők, ha egy kérdőjelet írunk utánuk. (A kérdőjelnek két különböző jelentése van: önmagában a `{0,1}` mennyiségjelző gyorsírásos alakja, míg ha egy mennyiségjelzőt követ, akkor arra utasítja azt, hogy ne legyen mohó.) A `\d+?` szabályos kifejezés például három helyen illeszkedik a 136 karakterláncra: 136, 136 és 136. Íme még egy példa: a `\d??` nulla vagy egy számjegyre illeszkedik, de a nulla előfordulást részesíti előnyben, mert nem mohó – önmagában állva ugyanaz a gond vele, mint a csillaggal: a semmire is találatot jelez, vagyis „minden szövegre" illeszkedik.

12.2. táblázat *A szabályos kifejezések mennyiségjelzői*

Alak	Jelentés
`e?` vagy `e{0,1}`	Mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e??` vagy `e{0,1}?`	Nem mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e+` vagy `e{1,}`	Mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e+?` vagy `e{1,}?`	Nem mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e*` vagy `e{0,}`	Mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e*?` vagy `e{0,}?`	Nem mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e{m}`	Az *e* kifejezés pontosan *m* előfordulására illeszkedik.
`e{m,}`	Mohón illeszkedik az *e* kifejezés legalább *m* előfordulására.
`e{m,}?`	Nem mohón illeszkedik az *e* kifejezés legalább *m* előfordulására.
`e{,n}`	Mohón illeszkedik az *e* kifejezés legfeljebb *n* előfordulására.
`e{,n}?`	Nem mohón illeszkedik az *e* kifejezés legfeljebb *n* előfordulására.
`e{m,n}`	Mohón illeszkedik az *e* kifejezés legalább *m* és legfeljebb *n* előfordulására.
`e{m,n}?`	Nem mohón illeszkedik az *e* kifejezés legalább *m* és legfeljebb *n* előfordulására.

A nem mohó mennyiségjelzők akkor hasznosak, ha „gyors és piszkos" XML- vagy HTML-feldolgozásra van szükségünk. Például ha az összes képcímkét keressük, az **<img.*>** („keress egy „<", majd egy „i", egy „m", egy „g" és nulla vagy több tetszőleges karaktert az újsor karakter kivételével, végül pedig egy „>" karaktert") nem fog működni, mert a **.*** rész mohó, így mindenre illeszkedni fog, beleértve a címke záró > karakterét, és addig ad találatokat, amíg el nem érte a teljes szöveg utolsó > karakterét.

Három megoldás kínálkozik (egy megfelelő feldolgozó használatán kívül). Az első az **<img[^>]*>** (ez akkor illeszkedik, ha az <img karaktereket tetszőleges számú olyan karakter követi, ami nem a >, majd a címke záró > karaktere), a második az **<img.*?>** (ez akkor illeszkedik, ha az <img karaktereket tetszőleges karakter követi – nem mohón értelmezve, ezért a címke záró > karaktere előtt megáll –, majd a > karakter következik), a harmadik pedig az **<img[^>]*?>**, amely az előző kettőt ötvözi. Ugyanakkor egyik megoldás sem helyes, mert mind találatot ad az <img> karakterláncra is, ami nem érvényes címke. Mivel tudjuk, hogy minden képelemnek rendelkeznie kell az src jellemzővel, a következő szabályos kifejezés pontosabb lenne: **<img\s+[^>]*?src=\w+[^>]*?>**. Ez a kifejezés literálisan illeszkedik az <img karakterekre, majd egy vagy több üreshelyre, aztán nem mohón nulla vagy több tetszőleges karakterre a > kivételével (így átugrik minden más jellemzőt, például az alt-ot), ezt követően az src jellemzőre (vagyis a literálisan értelmezett src= karakterláncra, amelyet legalább egy „szókarakter" követ), aztán az egyéb jellemzők megtalálása érdekében bármely olyan karakterre, ami nem a > (a „semmi" is számít), végül pedig a záró > karakterre.

Csoportosítás és elfogás

A valós alkalmazásokban gyakran van szükség olyan szabályos kifejezésekre, amelyek két vagy több alternatíva közül bármelyikre illeszkednek, és a találatot vagy annak egy részét is gyakran kell elfognunk további feldolgozás céljából. Az is előfordul, hogy azt szeretnénk, ha egy mennyiségjelző egyszerre több kifejezésre vonatkozna. Ezek mindegyikét elérhetjük csoportosítással, amire a () jelek szolgálnak, az alternatívák meghatározására pedig a | („vagy") jelet használhatjuk.

A „vagy" művelet különösen akkor hasznos, ha azt szeretnénk, hogy a szabályos kifejezés több teljesen különböző lehetőség közül bármelyikre illeszkedjen.

Az **aircraft|airplane|jet** szabályos kifejezés például bármilyen szöveget visszaad, amely tartalmazza akár az „aircraft", akár az „airplane", akár a „jet" karaktereket. Ugyanez elérhető az **air(craft|plane)|jet** szabályos kifejezéssel is. Itt a kerek zárójelek a kifejezések csoportosítására szolgálnak, tehát két „külső" kifejezésünk van – az **air(craft|plane)** és a **jet** –, az elsőben pedig egy belső kifejezés (**craft|plane**) található. Mivel ezt a belső kifejezést az **air** előzi meg, az első külső kifejezés csak az „aircraft", illetve az „airplane" szavakra illeszkedik.

A (kerek) zárójelek két különböző célt szolgálnak: kifejezések csoportosítására, illetve egy kifejezésre illeszkedő szöveg elfogására használhatók. A *csoport* szót az elfogott és el nem fogott csoportosított kifejezésekre egyaránt használjuk, a *fogás* vagy *elfogási csoport* szót viszont csak az elfogni szándékozott csoportokra. Ha az `(aircraft|airplane|jet)` szabályos kifejezést használnánk, az nem csak illeszkedne a benne szereplő három szó bármelyikére, hanem el is fogná a megtalált kifejezést későbbi használatra. Hasonlítsuk ezt össze az `(air(craft|plane)|jet)` szabályos kifejezéssel, amely két fogást tartalmaz, ha az első kifejezés illeszkedik (az első fogás az „aircraft” vagy „airplane”, a második pedig a „craft” vagy „plane”), és egy fogást („jet”), ha a második kifejezés ad találatot. Az elfogást kikapcsolhatjuk, ha a nyitó zárójel után a `?:` karaktereket írjuk, tehát az `(air(?:craft|plane)|jet)` szabályos kifejezésnek például csak egy fogása lesz, ha illeszkedik („aircraft” vagy „airplane” vagy „jet”).

A csoportosított kifejezés is kifejezés, tehát ellátható mennyiségjelzővel. Mint minden más kifejezés esetében, a nyelv egyszeri illesztést feltételez, hacsak kifejezetten meg nem adjuk, hogy hányszor szeretnénk illeszteni. Például ha egy olyan szövegfájlt olvasunk be, amelyben *kulcs=érték* alakú sorok találhatók, ahol minden *kulcs* alfanumerikus, a `(\w+)=(.+)` szabályos kifejezés minden olyan sorra illeszkedni fog, amelyben sem a kulcs, sem az érték mező nem üres. (Emlékezzünk vissza, hogy a . mindenre illeszkedik, kivéve az újsor karaktereket.) Ezen felül minden illeszkedő sor esetében két fogásunk lesz: az első a kulcs, a második pedig az érték.

A *kulcs=érték* fenti szabályos kifejezése például a teljes `topic= physical geography` sorra illeszkedni fog (a két fogást az árnyékolás mutatja). Figyeljük meg, hogy a második fogásban egy szóköz (üreshely karakter) is szerepel, az egyenlőségjel előtt azonban nem lehet üreshely. A szabályos kifejezést a következőképpen írhatnánk át egy kicsit hosszabb formára, hogy rugalmasabb legyen az üreshelyek elfogadása tekintetében, és eltávolítsa a nemkívánatos szóközöket:

```
[ \t]*(\w+)[ \t]*=[ \t]*(.+)
```

Ez a kifejezés is illeszkedik ugyanarra a sorra, mint a korábbi, de azokra is, amelyekben az egyenlőségjel körül szóközök találhatók, csak éppen az első fogásba nem kerül bele a bevezető vagy záró szóköz, a második fogásba pedig a bevezető üreshely. Például: `topic= physical geography`.

Szabályoskifejezés-jelzők
☞ 479

Az üreshelyekre illeszkedő részeket gondosan az elfogó zárójeleken kívül helyeztük el, és arra is ügyeltünk, hogy azokat a sorokat is megtaláljuk, amelyekben egyáltalán nincs szóköz. Az üreshelyekre nem az `\s` kifejezéssel illesztettünk, mert ez az újsor karakterekre (`\n`) is illeszkedik, ami helytelenül több sort átfogó találatokat eredményezett volna (például ha a `re.MULTILINE` jelzőt használjuk). Az érték esetében pedig azért nem az `\S` kifejezést alkalmaztuk a nem üreshely karakterek illesztésére, mert azokat az értékeket (például

a mondatokat) is meg kívántuk engedni, amelyek szóközt is tartalmaznak. Ahhoz, hogy elkerüljük, hogy a második fogásban záró szóköz is lehessen, kifinomultabb szabályos kifejezésre van szükség – erre a következő részben láthatunk majd példát.

Az elfogott kifejezésekre *visszirányú hivatkozásokkal* hivatkozhatunk, vagyis úgy, hogy visszautalunk egy korábbi elfogási csoportra.* Magukon a szabályos kifejezéseken belül az egyik lehetséges hivatkozási forma az `\i`, ahol az `i` a fogás száma. A fogások egytől számozódnak, és a szám balról jobbra egyesével nő, ahogy az értelmező egy új (elfogó) bal zárójellel találkozik. A kétszer szereplő szavakat például leegyszerűsítve úgy találhatjuk meg, ha a **`(\w+)\s+\1`** szabályos kifejezést használjuk, amely egy „szóra" illeszkedik, amelyet legalább egy üreshely követ, majd ismét az elfogott szó. (A 0 elfogási szám automatikusan jön létre, anélkül, hogy szükség lenne a zárójelekre – a 0. fogás a teljes találatot tartalmazza, vagyis azt, amit aláhúzással jelölünk.) Az ismétlődő szavak illesztésére később egy kifinomultabb módszert is látni fogunk.

A hosszú vagy összetett szabályos kifejezésekben gyakran kényelmesebb nevekkel, mint számokkal azonosítani a fogásokat. Ez a kód karbantartását is megkönnyíti, mivel az elfogó zárójelek hozzáadása vagy eltávolítása a számozást megváltoztatja, a nevekre azonban nincs hatással. Ha nevet szeretnénk adni egy fogásnak, a nyitó zárójel után a **`?P<`***`név`***`>`** kifejezést kell írnunk. A **`(?P<key>\w+)=(?P<value>.+)`** szabályos kifejezésben például két fogás szerepel: a `"key"` és a `"value"`. A nevesített fogásokra mutató visszirányú hivatkozások formája a szabályos kifejezéseken belül **`(?P=`***`név`***`)`**. A **`(?P<word>\w+)\s+(?P=word)`** például a `"word"` nevű fogás segítségével az ismétlődő szavakra illeszkedik.

Feltételezések és jelzők

Az eddig látott szabályos kifejezések nagy részét érintő probléma, hogy több, sőt egészen eltérő szövegekre is illeszkedhetnek, mint amit szerettünk volna.

Az **`aircraft|airplane|jet`** szabályos kifejezés például a „jet" mellett a „waterjet" és a „jetski" szavakat is visszaadja. Ezt a fajta problémát feltételezések (assertion, „állítás") segítségével oldhatjuk meg. A feltételezés nem illeszt semmilyen szöveget, csupán állít valamit a szövegről azon a ponton, ahol a feltételezés szerepel.

Ilyen feltételezés például a **`\b`** (szóhatár), amely azt mondja ki, hogy az előtte álló karakternek „szókarakternek" (**`\w`**) kell lennie, az utána állónak pedig nem szókarakternek (**`\W`**), vagy fordítva. A **`jet`** szabályos kifejezés például a „the jet and jetski are noisy" szövegben kétszer illeszkedik – `the jet and jetski are noisy` –, a **`\bjet\b`** viszont csak egyszer: `the jet and jetski are noisy`. Az eredeti szabályos kifejezés összefüggésében tehát a **`\baircraft\b|\bairplane\b|\bjet\b`** alakot, vagy hogy átláthatóbb legyen, a **`\b(?:aircraft|airplane|jet)\b`** szabályos kifejezést írhatjuk (vagyis: szóhatár, nem elfogó kifejezés, szóhatár).

---

* Jegyezzük meg, hogy a karakterosztályokon belül (vagyis egy `[]` belsejében) nem használhatunk visszirányú hivatkozásokat.

Ahogy a 12.3. táblázat is mutatja, a nyelv számos egyéb feltételezést támogat. Ezeket felhasználhatjuk például egy *kulcs=érték* szabályos kifejezés világosabbá tételére, ha a regexet a `^(\w+)=([^\n]+)` alakra módosítjuk, és beállítjuk a `re.MULTILINE` jelzőt, hogy biztosítsuk, hogy minden *kulcs=érték* pár egyetlen sorból származzon, és ne legyen lehetőség több sor átfogására. (A jelzőket a 12.5. táblázatban találhatjuk, használati alakjukat pedig ennek a résznek a végén, illetve a következő részben ismertetjük.) Ha a bevezető és záró üreshelyeket is el szeretnénk távolítani, valamint nevesített fogásokat szeretnénk használni, a szabályos kifejezés teljes alakja ez lesz:

```
^[ \t]*(?P<key>\w+)[ \t]*=[ \t]*(?P<value>[^\n]+)(?<![ \t])
```

Szabályoskifejezés-jelzők
☞ 479

Bár ez a szabályos kifejezés meglehetősen egyszerű feladatot lát el, elég bonyolultnak tűnik. A későbbi módosítás megkönnyítésének egyik módja, ha megjegyzésekkel látjuk el. Ezt megtehetjük soron belüli megjegyzések hozzáadásával is, `(?#`*megjegyzés*`)` alakban, de a gyakorlatban az ilyen megjegyzések a szabályos kifejezést még nehezebben olvashatóvá tehetik. Ennél sokkal elegánsabb megoldás a `re.VERBOSE` jelző használata, amely lehetővé teszi, hogy a szabályos kifejezésekben szabadon használhassunk üreshelyeket és szokványos Python-megjegyzéseket, azzal az egy megkötéssel, hogy ha üreshely karakterre szeretnénk illeszteni, akkor vagy az `\s` kifejezést, vagy egy olyan karakterosztályt kell alkalmaznunk, mint a `[ ]`. Íme a *kulcs=érték* szabályos kifejezés megjegyzésekkel ellátott formában:

```
^[ \t]*                 # a sor kezdete és az esetleges bevezető üreshely
(?P<key>\w+)            # a kulcs szövege
[ \t]*=[ \t]*           # az egyenlőségjel a körülötte levő esetleges
                        # üreshelyekkel
(?P<value>[^\n]+)       # az érték szövege
(?<![ \t])              # negatív visszatekintés a záró üreshelyek elkerülésére
```

Nyers karakterláncok
☞ 61

Egy Python-programban egy ehhez hasonló szabályos kifejezést általában egy nyers, háromszoros idézőjelek közé zárt karakterláncban helyeznénk el – azért nyers karakterláncban, hogy ne kelljen megkettőzni a fordított perjeleket, és azért háromszoros idézőjelek közé zártban, hogy több sorra tördelhessük.

Az eddig bemutatott feltételezéseken kívül továbbiak is léteznek, amelyek az állítás előtt (vagy után) álló szöveget vizsgálják, hogy megfelel-e (vagy éppen ellenkezőleg) egy általunk megadott kifejezésnek. A visszatekintő állításokban (utófeltételezésekben) szereplő kifejezéseknek rögzített hosszúságúaknak kell lenniük (tehát a `?`, `+` és `*` mennyiségjelzők nem használhatók, a számszerű mennyiségjelzőknek pedig adott értékűnek kell lenniük, például: `{3}`).

12.3. táblázat *A szabályos kifejezések feltételezései*

Jel	Jelentés
`^`	A szöveg kezdetére illeszkedik, illetve a `re.MULTILINE` jelző használata esetén minden újsor karakter után.
`$`	A szöveg végére illeszkedik, illetve a `re.MULTILINE` jelző használata esetén minden újsor karakter előtt.
`\A`	A szöveg elejére illeszkedik.
`\b`	A szóhatárokra illeszkedik, a `re.ASCII` jelzőnek megfelelően. Egy karakterosztályon belül a „backspace” karakter levédésére szolgál.
`\B`	Nem szóhatárra illeszkedik, a `re.ASCII` jelzőnek megfelelően.
`\Z`	A szöveg végére illeszkedik.
`(?=`*e*`)`	Akkor illeszkedik, ha az *e* kifejezés illeszkedik a feltételezésnél, de nem halad át rajta. *Előfeltételezésnek* vagy *pozitív előretekintésnek* is hívják.
`(?!`*e*`)`	Akkor illeszkedik, ha az *e* kifejezés nem illeszkedik a feltételezésnél, és nem halad át rajta. *Negatív előretekintésnek* is hívják.
`(?<=`*e*`)`	Akkor illeszkedik, ha az *e* kifejezés közvetlenül a feltételezés előtt illeszkedik. *Pozitív visszatekintésnek* is hívják.
`(?<!`*e*`)`	Akkor illeszkedik, ha az *e* kifejezés nem illeszkedik közvetlenül a feltételezés előtt. *Negatív visszatekintésnek* is hívják.

Szabályoskifejezés-jelzők ☞ 479

A *kulcs=érték* szabályos kifejezés esetében a negatív visszatekintés azt jelenti, hogy ahol a feltételezés szerepel, ott az *előző* karakter nem lehet szóköz vagy tabulátor. Ez biztosítja, hogy a `"value"` elfogási csoportba olvasott utolsó karakter ne lehessen szóköz vagy tabulátor (ugyanakkor azt nem akadályozza meg, hogy az elfogott szövegen belül szóköz vagy tabulátor álljon).

Nézzünk meg még egy példát! Tegyük fel, hogy egy többsoros szöveget olvasunk, amely a „Helen Patricia Sharman”, „Jim Sharman”, „Sharman Joshi”, „Helen Kelly” stb. neveket tartalmazza, és a „Helen Patricia” nevet keressük, de csak akkor, ha „Helen Patricia Sharman”-ról van szó. Ennek a `\b(Helen\s+Patricia)\s+Sharman\b` szabályos kifejezés használata a legegyszerűbb módja, de ugyanezt az eredményt egy előretekintő állítással is elérhetjük, például így: `\b(Helen\s+Patricia)(?=\s+Sharman\b)`. Ez a kifejezés csak akkor illeszkedik a „Helen Patricia” névre, ha azt egy szóhatár előzi meg, utána pedig egy szóköz és a „Sharman” szó áll, amelynek a végén szóhatár található.

A szövegben szereplő keresztnév-változatok („Helen”, „Helen P.” vagy „Helen Patricia”) elfogásához a szabályos kifejezésen kissé bonyolítanunk kell, például így: `\b(Helen(?:\s+(?:P\.|Patricia))?)\s+(?=Sharman\b)`. Ez a kifejezés egy szóhatárra illeszkedik, amelyet valamelyik keresztnév-változat követ, de csak akkor, ha a keresztnév után egy szóköz és a „Sharman” szó áll, amelynek a végén szóhatár található.

Jegyezzük meg, hogy az elfogásnak csak két formája lehetséges: az `(e)` és a `(?P<név>e)`. Az egyéb zárójeles alakok egyike sem hajt végre elfogást. Ez tökéletesen érthető az előmi előzi meg, illetve követi őket – nem részei az illesztésnek, csupán azt befolyásolják, és utófeltételezések esetében, hiszen ezek csak egy állítást fogalmaznak meg arról, hogy hogy kapunk-e egyezést. Ezen kívül a következőkben bemutatott két utolsó zárójeles alaknál is ésszerű.

Korábban láttuk, hogy miként hivatkozhatunk visszafelé egy fogásra egy szabályos kifejezés belsejében, akár számmal (például `\1`), akár név szerint (például `(?P=név)`). A feltételes illesztés azonban attól függően is lehetséges, hogy korábban kaptunk-e már egyezést. Ennek utasításformája a következő: `(?(azonosító)igen_kifejezés)` vagy `(?(azonosító)igen_kifejezés|nem_kifejezés)`. Az *azonosító* annak a korábbi fogásnak a sorszáma vagy neve, amelyikre hivatkozunk. Ha az elfogás sikerült, az *igen_kifejezés*-re illesztünk, ha viszont meghiúsult, a *nem_kifejezés*-re (amennyiben megadtunk ilyet).

Vegyünk egy példát: tegyük fel, hogy az `img` HTML-elemek `src` jellemzőjéből szeretnénk kinyerni az ott hivatkozott fájlneveket. Először csak az `src` jellemzőre próbálunk illeszteni, de a korábbi kísérletünktől eltérően figyelembe vesszük a jellemző értéke által felvehető három alakot (aposztrófos, idézőjeles, idézőjel nélküli). Íme az első változat: `src=(["'])([^"'>]+)\1`. A `([^"'>]+)` rész legalább egy nem idézőjel vagy > karakter mohó illesztését fogja el. Ez a szabályos kifejezés az idézőjeles fájlnevek esetében megfelelően működik, és az `\1`-nek köszönhetően csak akkor ad egyezést, ha a nyitó és záró idézőjelek egyformák. Az idézőjel nélküli fájlneveket azonban nem engedi meg. Ennek kijavításához a nyitó idézőjelet elhagyhatóvá kell tennünk, és csak akkor szabad illesztenünk rá, ha jelen van. Íme a javított szabályos kifejezés: `src=(["'])?([^"'>]+)(?(1)\1)`. A *nem_kifejezés*-t nem adtuk meg, mivel nincs mire illeszteni, ha nincs idézőjel. Most már készen állunk arra, hogy a szabályos kifejezést összefüggésében helyezzük el – íme az `img` elemekre írt teljes szabályos kifejezés, nevesített csoportok és megjegyzések használatával:

```
<img\s+                          # a címke kezdete
[^>]*?                           # az src-t megelőző esetleges jellemzők
src=                             # az src jellemző kezdete
(?P<quote>["'])?                 # elhagyható nyitó idézőjel
(?P<image>[^"'>]+)               # a kép fájlneve
(?(quote)(?P=quote))             # záró idézőjel (ha van nyitó idézőjel,
                                 # azzal egyező)
[^>]*?                           # az src-t követő esetleges jellemzők
>                                # a címke vége
```

Az elfogott fájlnév elfogási neve `"image"` (a fogás sorszáma pedig 2).

Természetesen van egyszerűbb és kifinomultabb megoldás is: `src=(["']?)([^"'>]+)\1`. Itt, ha van nyitó idézőjel karakter, akkor elfogjuk az 1-es elfogási csoportban, és a nem idézőjel karakterek után illesztjük. Ha nincs nyitó idézőjel, az 1-es csoport akkor is illeszkedni

fog – mégpedig egy üres karakterláncra, mivel nem kötelező (a mennyiségjelzője nulla vagy egy), mely esetben a visszirányú hivatkozás ugyancsak egy üres karakterláncra fog illeszkedni.

A Python szabályoskifejezés-motorja által kínált szabályos kifejezések nyelvtanában az utolsó rész, aminek még a helyére kell kerülnie, a jelzők (flag) beállításának módja. A jelzőket általában kiegészítő paraméterekként átadva állítjuk be, amikor meghívjuk a `re.compile()` függvényt, de néha kényelmesebb magának a szabályos kifejezésnek a részeként beállítani őket. Az utasításforma egyszerűen `(?`*jelzők*`)`, ahol a *jelzők* egy vagy több jelzőt jelent az `a` (ez ugyanazt jelenti, mint a `re.ASCII` átadása) az `i` (`re.IGNORECASE`), az `m` (`re.MULTILINE`), az `s` (`re.DOTALL`) és az `x` (`re.VERBOSE`) közül.* Ha a jelzőket így állítjuk be, a szabályos kifejezés elejére kell tennünk őket. Semmire sem illeszkednek, ezért a hatásuk csupán annyi a szabályos kifejezésre, hogy beállítják a jelzőket.

Szabályoskifejezés-jelzők
☞ 479

A Python szabályoskifejezés-modulja

A `re` modul kétféle módját teszi lehetővé a szabályos kifejezésekkel való munkának. Az egyik a 12.4. táblázatban felsorolt függvények használata, amikor is az adott függvénynek első paraméterként egy szabályos kifejezést adunk át. A függvény a szabályos kifejezést egy belső formátumra alakítja át – ezt hívják *fordításnak* (compiling) –, és ezt követően hajtja végre a feladatát. Ez egyszeri használat esetén nagyon kényelmes, de ha ugyanazt a szabályos kifejezést többször is használni szeretnénk, a minden használatkori fordítás költségét célszerű elkerülnünk: ezt teszi lehetővé a `re.compile()` függvény, amellyel csak egyszer kell lefordítanunk a szabályos kifejezést. Ezt követően annyiszor hívhatunk meg tagfüggvényeket a lefordított szabályos kifejezésre, ahányszor csak szeretnénk. A lefordított szabályos kifejezések tagfüggvényeit a 12.6. táblázatban láthatjuk.

```
match = re.search.(r"#[\dA-Fa-f]{6}\b", text)
```

A fenti kódrészlet egy `re`-modulfüggvény használatát mutatja be. A szabályos kifejezés a HTML-stílusban (például `#C0C0AB` formában) megadott színeket keresi. Ha egyezést talál, a `re.search()` függvény egy találati objektumot ad vissza, egyébként pedig `None`-t. A találati objektumok által biztosított tagfüggvényeket a 12.7. táblázatban tekinthetjük meg.

Ha ezt a szabályos kifejezést többször szeretnénk használni, akkor egyszer lefordítjuk, majd szükség esetén a lefordított szabályos kifejezést használjuk:

```
color_re = re.compile(r"#[\dA-Fa-f]{6}\b")
match = color_re.search(text)
```

* A jelzőkhöz használt betűk ugyanazok, mint amelyeket a Perl szabályoskifejezés-motorja használ, ezért az s a re.DOTALL, illetve az x a re.VERBOSE jele.

Ahogy korábban megjegyeztük, nyers karakterláncokat használunk, hogy elkerüljük a fordított perjelek levédésének szükségességét. A fenti szabályos kifejezést másképp is felírhatnánk a `[\dA-F]` karakterosztály használatával és a `re.IGNORECASE` jelzőt a `re.compile()` hívás utolsó paramétereként átadva, vagy használhatnánk a `(?i)#[\dA-F]{6}\b` szabályos kifejezést, amely a kis- és nagybetűk figyelmen kívül hagyására utasító jelzővel kezdődik.

Ha egynél több jelzőre van szükség, az OR („vagy") művelettel (`|`) kapcsolhatjuk össze őket, például így: `re.MULTILINE|re.DOTALL` (vagy `(?ms)`, ha magába a szabályos kifejezésbe ágyazzuk be).

Ezt a részt néhány példa áttekintésével zárjuk. Néhány olyan szabályos kifejezéssel kezdjük, amelyeket a fejezet korábbi részeiben láttunk, hogy szemléltessük a `re` modul által kínált lehetőségek közül a leggyakrabban használtakat. Elsőként lássuk a kétszer szereplő szavakat kiszűrő szabályos kifejezést:

```
double_word_re = re.compile(r"\b(?P<word>\w+)\s+(?P=word)(?!\w)",
                            re.IGNORECASE)
for match in double_word_re.finditer(text):
    print("{0} is duplicated".format(match.group("word")))
```

Ez a szabályos kifejezés némileg bonyolultabb, mint a korábban elkészített változata. A kifejezés egy szóhatárral kezdődik (hogy biztosítsuk, hogy minden illeszkedés egy szó elején kezdődjön), majd mohón illeszkedik egy vagy több „szókarakterre", egy vagy több üreshely karakterre, majd ismét ugyanarra a szóra – de csak akkor, ha a szó második előfordulását nem szókarakter követi.

Ha a bemeneten érkező szöveg „win in vain" volt, az első feltételezés *nélkül* egyetlen egyezést (találatot) és két fogást kapnánk: `win in vain`. A szóhatár-feltételezés használata biztosítja, hogy az első egyezésként kapott karakterlánc egy teljes szó legyen, így nem kapunk egyezést vagy fogást, hiszen nem egy szó ismétlődik. Ehhez hasonlóan, ha a bemeneten érkező szöveg „`one and and two let's s say`" volt, az utolsó feltételezés *nélkül* két egyezésünk és két fogásunk lenne: `one and and two let's say`. Az előretekintő feltételezés használata azt jelenti, hogy a második egyezésként kapott karakterláncnak teljes szónak kell lennie, ezért egyetlen egyezést és egyetlen fogást kapunk: `one and and two let's say`.

A `for` ciklus végigmegy a `finditer()` tagfüggvény által visszaadott összes találati objektumon, a találati objektum `group()` tagfüggvényével pedig kinyerjük az elfogási csoport szövegét. Ugyanilyen könnyen használhattuk volna a `group(1)` kifejezést is (bár ez később nehezebben módosítható), mely esetben egyáltalán nem kellett volna nevet adnunk az elfogási csoportnak, és egyszerűen használhattuk volna a `(\w+)\s+\1(?!\w)` szabályos kifejezést. Azt is érdemes megjegyezni, hogy a kifejezés végén a `(?!\w)` helyett egy szóhatárt (`\b`) is alkalmazhattunk volna.

Egy másik korábbi példa egy olyan szabályos kifejezés volt, amely a fájlneveket kereste a HTML-képelemekben. Lássuk, hogyan fordíthatnánk le ezt a szabályos kifejezést, jelzőket hozzáadva, hogy a kis- és nagybetűket ne különböztesse meg, és lehetővé tegye megjegyzések beágyazását is:

```
image_re = re.compile(r"""
                <img\s+                     # a címke kezdete
                [^>]*?                      # nem-src jellemzők
                src=                        # az src jellemző kezdete
                (?P<quote>["'])?            # elhagyható nyitó idézőjel
                (?P<image>[^"'>]+)          # a kép fájlneve
                (?(quote)(?P=quote))        # záró idézőjel
                [^>]*?                      # nem-src jellemzők
                >                           # a címke vége
                """, re.IGNORECASE|re.VERBOSE)
image_files = []
for match in image_re.finditer(text):
    image_files.append(match.group("image"))
```

Itt ismét a `finditer()` tagfüggvényt használtuk a találatok kinyerésére, és a találati objektum `group()` tagfüggvényét az elfogott szövegek kiolvasására. Mivel a kis- és nagybetűkre való érzéketlenség csak az **`img`** és **`src`** elemekre vonatkozik, a `re.IGNORECASE` jelzőt el is hagyhatnánk, és használhatnánk helyette az **`[Ii][Mm][Gg]`**, illetve az **`[Ss][Rr][Cc]`** kifejezéseket. Ez a szabályos kifejezést némileg kevésbé átláthatóvá tenné, ugyanakkor gyorsítana rajta, mivel az illesztett szöveget nem kellene nagybetűssé (vagy kisbetűssé) alakítani – de ez valószínűleg csak akkor számít, ha a szabályos kifejezést igen nagy mennyiségű szövegre alkalmaznánk.

Gyakori feladat, hogy egy HTML-szövegből a címkéket kihagyva sima szöveges kimenetet kell készítenünk. Ezt temészetesen elvégezhetnénk a Python valamelyik feldolgozójának segítségével is, de szabályos kifejezések használatával is elkészíthetünk egy egyszerű eszközt. Három műveletet kell végrehajtanunk: törölnünk kell az összes címkét, a karakteregyedeket az általuk jelölt tényleges karakterekre kell cserélnünk, valamint üres sorokat kell beszúrnunk a bekezdések elválasztására. Íme egy függvény (a `html2text.py` programból), amely elvégzi mindezt:

```
def html2text(html_text):
    def char_from_entity(match):
        code = html.entities.name2codepoint.get(match.group(1), 0xFFFD)
        return chr(code)

    text = re.sub(r"<!--(?:.|\n)*?-->", "", html_text)                    #1
    text = re.sub(r"<[Pp][^>]*?(?!</)>", "\n\n", text)                    #2
    text = re.sub(r"<[^>]*?>", "", text)                                  #3
    text = re.sub(r"&#(\d+);", lambda m: chr(int(m.group(1))), text)
    text = re.sub(r"&([A-Za-z]+);", char_from_entity, text)               #5
    text = re.sub(r"\n(?:[ \xA0\t]+\n)+", "\n", text)                     #6
    return re.sub(r"\n\n+", "\n\n", text.strip())                         #7
```

Az első szabályos kifejezés, a `<!-(?:.|\n)*?->`, a HTML-megjegyzésekre illeszkedik, beleértve azokat is, amelyekbe más HTML-elemeket ágyaztak. A `re.sub()` függvény az összes találatot a cserekarakterekre cseréli, illetve törli őket, ha a csereszöveg egy üres karakterlánc, mint itt. (A találatok számának maximumát a kifejezés végén paraméterként átadott egész számmal határozhatjuk meg.)

Ügyeltünk rá, hogy nem mohó (minimális) illesztést végezzünk, hogy minden találat esetében csak egy megjegyzést töröljünk. Ha nem így tettünk volna, az első megjegyzés elejétől az utolsó megjegyzés végéig mindent törölnénk.

A `re.sub()` függvény paraméterként nem fogad el jelzőket, tehát a pont (`.`) azt jelenti, hogy „az újsor kivételével bármely karakter”, ezért pontot vagy `\n`-t kell keresnünk, mégpedig nem karakterosztály, hanem „vagy” művelet segítségével, mivel a karakterosztályokon belül a . literális jelentésű, tehát egy pontot jelöl. Egy másik lehetőség, ha a szabályos kifejezést a beágyazott jelzővel kezdjük (például: `(?s)<!-.*?->`), vagy ha a `re.DOTALL` jelzővel lefordítunk egy szabályoskifejezés-objektumot, mely esetben a szabályos kifejezés egyszerűen ennyi lenne: `<!-.*?->`.

A második szabályos kifejezés, a `<[Pp][^>]*?(?!</)>`, a nyitó bekezdéscímkékre (például `<P>` vagy `<p align=center>`) illeszkedik. Megkeresi a nyitó `<p` (vagy `<P`) karaktereket, majd (nem mohó illesztéssel) a jellemzőket, végül pedig a záró `>` karaktert, feltéve, hogy azt nem `/` előzi meg (erről egy negatív visszatekintő feltételezés gondoskodik), mivel az záró bekezdéscímkét jelölne. A `re.sub()` függvény második hívása ezt a szabályos kifejezést használja arra, hogy a nyitó bekezdéscímkéket két újsor karakterre cserélje (ami a sima szöveges fájlokban a bekezdések elválasztásának szabványos módja).

A harmadik szabályos kifejezés, a `<[^>]*?>`, minden címkére illeszkedik, és a harmadik `re.sub()` függvényhívásban használjuk a fennmaradó címkék törlésére.

A HTML-egyedek (HTML-karakteregyedek) a nem ASCII karakterek ASCII-karakterekkel történő leírására szolgálnak. Kétféle alakjuk lehetséges: az egyik az `&`*`név`*`;` – ahol a *`név`* a karakter neve (például a © karakter esetében `©`), a másik pedig az `&#`*`számjegyek`*`;` – ahol a *`számjegyek`* az Unicode-kódpontot azonosító decimális számjegyek (a ¥ karakter esetében például `¥`). A `re.sub()` negyedik hívása az `&#(\d+);` szabályos kifejezést használja, amely a számjegyes alakra illeszkedik, és a számjegyeket az 1-es elfogási csoportba olvassa. Literális csereszöveg helyett egy lambdafüggvényt adunk át. Amikor a `re.sub()` egy függvényt kap, minden egyezés megtalálásakor meghívja azt, és egyetlen paraméterként átadja neki a találati objektumot. A `lambda` függvényben (karakterláncként) kinyerjük a számjegyeket, a beépített `int()` függvénnyel egész számmá alakítjuk őket, majd a szintén beépített `chr()` függvénnyel kiolvassuk az adott kódpont Unicode-karakterét. Csereszövegként a függvény visszatérési értékét (illetve a `lambda` kifejezések esetében a kifejezés eredményét) használjuk.

Az ötödik `re.sub()` hívás az `&([A-Za-z]+);` szabályos kifejezéssel a nevesített egyedeket fogja el. Az alapkönyvtár `html.entities` modulja tartalmazza az egyedek szótárait, köztük a `name2codepoint`-ot, amelynek a kulcsai egyednevek, az értékei pedig egész értékű kódpontok. A `re.sub()` függvény minden találat esetében a helyi `char_from_entity()` függvényt hívja meg. A `char_from_entity()` függvény a `dict.get()` függvényt használja, `0xFFFD` alapértelmezett paraméterrel (ez a szabványos Unicode-cserekarakter kódpontja, amelyet gyakran ❓ formában ábrázolnak). Ez biztosítja, hogy mindig kapjunk egy kódpontot, amelyet a `chr()` függvénynek átadva visszaadhatjuk a nevesített egyed lecserélésére alkalmas karaktert – illetve az Unicode-cserekaraktert, amennyiben az egyed neve nem érvényes.

A hatodik `re.sub()` hívás szabályos kifejezése, a **`\n(?:[ \xA0\t]+\n)+`**, a csak üreshelyeket tartalmazó sorok törlésére szolgál. A karakterosztályunk egy szóközt, egy nem törő szóközt (amelyre az előző szabályos kifejezésben az ` ` egyedeket cseréltük) és egy tabulátort tartalmaz. A szabályos kifejezés akkor illeszkedik, ha olyan újsor karaktert talál (egy sor végén, amelyet egy vagy több, csak üreshelyeket tartalmazó sor követ), amely után legalább egy csak üreshelyeket tartalmazó sor következik. Mivel a találatnak az újsor karakter is része, a csak üreshelyeket tartalmazó sorokat megelőző sorban a találatot egyetlen újsor karakterre kell cserélnünk, máskülönben nem csak a csak üreshelyeket tartalmazó sorokat törölnénk, hanem a megelező sor újsor karakterét is.

A hetedik és egyben utolsó `re.sub()` hívás eredményét visszaadjuk a hívónak. Ez a szabályos kifejezés, a **`\n\n+`**, arra szolgál, hogy két vagy több újsor karakter sorozatát pontosan két újsor karakterre cseréljük, vagyis hogy biztosítsuk, hogy minden bekezdést csak egyetlen üres sor válasszon el.

A HTML-es példában egyik csereszöveget sem közvetlenül a találatból vettük (bár HTML-egyedneveket és -számokat használtunk), egyes esetekben azonban a csereszövegben szerepelnie kell az illeszkedő szövegnek vagy annak egy részének is. Például ha van egy nevekből álló listánk, amelyben a nevek *Keresztnév Középsőnév1…KözépsőnévN Családnév* alakban állnak, és tetszőleges számú középső név (akár nulla is) lehet bennük, és a listából olyan új változatot szeretnénk készíteni, amelynek az elemei *Családnév,KeresztnévKözépsőnév1…KözépsőnévN* alakban szerepelnek, ezt egy szabályos kifejezés segítségével könnyen megtehetjük:

```
new_names = []
for name in names:
    name = re.sub(r"(\w+(?:\s+\w+)*)\s+(\w+)", r"\2, \1", name)
    new_names.append(name)
```

A szabályos kifejezés első része, a **`(\w+(?:\s+\w+)*)`**, a keresztnévre illeszkedik az első **`\w+`** kifejezéssel, és nulla vagy több középső névre a **`(?:\s+\w+)*`** kifejezéssel. Ez utóbbi nulla vagy több szóközt is megtalál, ha egy szó következik utána. A szabályos kifejezés második része, a **`\s+(\w+)`**, a keresztnevet (és a középső neveket) követő szóközre, illetve a családnévre illeszkedik.

Ha a szabályos kifejezés kissé zavarosnak tűnik, nevesített elfogási csoportokkal javíthatunk az olvashatóságán, és megkönnyíthetjük a későbbi módosítását:

```
name = re.sub(r"(?P<forenames>\w+(?:\s+\w+)*)"
              r"\s+(?P<surname>\w+)",
              r"\g<surname>, \g<forenames>"; name)
```

Az elfogott szövegre egy `sub()` vagy `subn()` függvényből vagy tagfüggvényből hivatkozhatunk, `\i` vagy `\g<azonosító>` formában, ahol az *i* az elfogási csoport sorszáma, az *azonosító* pedig az elfogási csoport neve vagy száma – az `\1` tehát ugyanaz, mint a `\g<1>`, a példánkban pedig a `\g<forenames>`-nek felel meg. Ugyanezt az alakot a találati objektumok `expand()` tagfüggvényének átadott karakterláncban is használhatjuk.

Miért nem a teljes nevet olvassa ki a szabályos kifejezés első része (hiszen mohó illesztést végez)? Nos, valójában a teljes nevet kiolvassa, de az illesztés meghiúsul, mert bár a középső nevek nulla vagy több alkalommal illeszkedhetnek, a családnévnek pontosan egyszer kell, a középső neveket illesztő rész azonban mohón mindent kiolvas. Miután nem járt sikerrel, a szabályoskifejezés-motor „visszateker", és elereszti az utolsó „középső nevet", így lehetővé válik a családnév illeszkedése. A mohó illesztés annyi egyezést keres, amennyit csak lehet, de megáll, ha a további találatok meghiúsítanák az illesztést.

12.4. táblázat *A szabályoskifejezés-modul függvényei*

Utasításforma	Leírás
`re.compile(r,f)`	Az *r* lefordított szabályos kifejezést adja vissza, a jelzőit *f*-re állítva, ha ezt a paramétert is megadjuk.
`re.escape(s)`	Az *s* karakterláncot adja vissza, minden nem alfanumerikus karaktert fordított perjellel levédve, ezért a visszaadott karakterláncban nem lesznek különleges szabályoskifejezés-karakterek.
`re.findall(r,s,f)`	Az *r* szabályos kifejezés minden egymást át nem fedő illeszkedését visszaadja az *s* karakterláncból (az *f* jelzőkkel módosítva, ha ezt a paramétert is megadjuk). Ha a szabályos kifejezés elfogást is tartalmaz, minden találatot fogások rendezett sorozataként kapunk vissza.
`re.finditer(r,s,f)`	Egy találati objektumot ad vissza az *r* szabályos kifejezés minden egymást át nem fedő illeszkedésére az *s* karakterláncban (az *f* jelzőkkel módosítva, ha ezt a paramétert is megadjuk).
`re.match(r,s,f)`	Egy találati objektumot ad vissza, ha az *r* szabályos kifejezés az *s* karakterlánc elejére illeszkedik (az *f* jelzőkkel módosítva, ha ezt a paramétert is megadjuk), máskülönben a visszatérési érték `None` lesz.
`re.search(r,s,f)`	Egy találati objektumot ad vissza, ha az *r* szabályos kifejezés bárhol illeszkedik az *s* karakterláncban (az *f* jelzőkkel módosítva, ha ezt a paramétert is megadjuk), máskülönben a visszatérési érték `None` lesz.

12.4. táblázat *A szabályoskifejezés-modul függvényei* *(folytatás)*

Utasításforma	Leírás
`re.split(r,s,m)`	Azoknak a karakterláncoknak a listáját adja vissza, amelyeket az *s* karakterláncnak az *r* szabályos kifejezés minden előfordulásánál történő szétvágásával kapunk (*m* számú vágásig, illetve ha ezt a paramétert nem adjuk meg, annyi darabig, amennyi csak lehetséges). Ha a szabályos kifejezés elfogást is tartalmaz, a fogások bekerülnek a listába az általuk szétvágott részek közé.
`re.sub(r,x,s,m)`	Az *s* karakterlánc másolatát adja vissza, az *r* szabályos kifejezés minden illeszkedését (vagy amennyit az *m* paraméterrel meghatározunk) *x*-re cserélve, ami egy karakterlánc vagy egy függvény lehet (lásd a szövegben).
`re.subn(r,x,s,m)`	Ugyanaz, mint a `re.sub()`, azzal a különbséggel, hogy az eredményként kapott karakterláncot és a végrehajtott cserék számát adja vissza, kételemű rendezett sorozatként (rendezett párként).

12.5. táblázat *A szabályoskifejezés-modul jelzői*

Jelző	Jelentés
`re.A` vagy `re.ASCII`	Hatására a `\b`, `\B`, `\s`, `\S`, `\w` és `\W` feltételezések ASCII-ként kezelik a karakterláncokat. Ezek a gyorsírásos karakterosztályok alapértelmezés szerint az Unicode-szabványra támaszkodnak.
`re.I` vagy `re.IGNORECASE`	Hatására a szabályos kifejezés a kis- és nagybetűk különbözőségét figyelmen kívül hagyva illeszkedik.
`re.M` vagy `re.MULTILINE`	Hatására a kalapjel (`^`) minden új sor elejére, illetve minden újsor karakter után illeszkedik, a dollárjel (`$`) pedig az újsor karakterek elé, illetve a sorok végére.
`re.S` vagy `re.DOTALL`	Hatására a pont (`.`) minden karakterre illeszkedik, az újsor karaktereket is beleértve.
`re.X` vagy `re.VERBOSE`	Lehetővé teszi üreshelyek és megjegyzések beágyazását.

12.6. táblázat *A szabályoskifejezés-objektumok tagfüggvényei*

Utasításforma	Leírás
`rx.findall(s,kezdet,vég)`	A szabályos kifejezés minden egymást át nem fedő illeszkedését visszaadja az *s* karakterláncból (illetve annak a *kezdet*`:`*vég* szeletéből). Ha a szabályos kifejezés elfogást is tartalmaz, minden találatot fogások rendezett sorozataként kapunk vissza.
`rx.finditer(s,kezdet,vég)`	Egy találati objektumot ad vissza a szabályos kifejezés minden egymást át nem fedő illeszkedésére az *s* karakterláncban (illetve annak a *kezdet*`:`*vég* szeletében).

12.6. táblázat *A szabályoskifejezés-objektumok tagfüggvényei* *(folytatás)*

Utasításforma	Leírás
`rx.flags`	Azok a jelzők, amelyeket a szabályos kifejezés lefordításakor beállítottunk.
`rx.groupindex`	Szótár, amelynek a kulcsai elfogási csoportok nevei, az értékei pedig a csoportok sorszámai. Ha nem használunk neveket, akkor üres.
`rx.match(` `s,kezdet,vég)`	Egy találati objektumot ad vissza, ha a szabályos kifejezés az *s* karakterlánc elejére (vagy az *s* `kezdet:vég` szeletének elejére) illeszkedik, különben a visszatérési érték `None` lesz.
`rx.pattern`	A karakterlánc, amelyből a szabályos kifejezést lefordítottuk.
`rx.search(` `s,kezdet,vég)`	Egy találati objektumot ad vissza, ha a szabályos kifejezés bárhol illeszkedik az *s* karakterláncban (illetve annak a `kezdet:vég` szeletében), máskülönben a visszatérési érték `None` lesz.
`rx.split(s,m)`	Azoknak a karakterláncoknak a listáját adja vissza, amelyeket az *s* karakterláncnak a szabályos kifejezés minden előfordulásánál történő szétvágásával kapunk (*m* számú vágásig, illetve ha ezt a paramétert nem adjuk meg, annyi darabig, amennyi csak lehetséges). Ha a szabályos kifejezés elfogást is tartalmaz, a fogások bekerülnek a listába az általuk szétvágott részek közé.
`rx.sub(x,s,m)`	Az *s* karakterlánc másolatát adja vissza, a szabályos kifejezés minden illeszkedését (vagy amennyit az *m* paraméterrel meghatározunk) *x*-re cserélve, ami egy karakterlánc vagy egy függvény lehet (lásd a szövegben).
`rx.subn(` `x,s,m)`	Ugyanaz, mint a `re.sub()`, azzal a különbséggel, hogy az eredményként kapott karakterláncot és a végrehajtott cserék számát adja vissza, kételemű rendezett sorozatként (rendezett párként).

Ha a név például „James W. Loewen”, a szabályos kifejezés először a teljes névre illeszkedik, vagyis így: `James W. Loewen`. Ez kielégíti a szabályos kifejezés első részét, de a családnevet kereső résznek nem hagy semmit, és mivel a családnév kötelező (a beleértett mennyiségjelzője 1), a szabályos kifejezés nem jár sikerrel. A középső neveket kiolvasó rész mennyiségjelzője a `*`, tehát nulla vagy több alkalommal illeszkedhet (esetünkben kétszer: „ W.” és „ Loewen”), ezért a szabályoskifejezés-értelmező motor ráveheti, hogy adjon fel valamennyit a találataiból, anélkül, hogy ez meghiúsítaná a rész illeszkedését. A szabályos kifejezés tehát visszateker, feladja az utolsó `\s+\w+`-t (mármint a „ Loewen”-t), így a találat `James W.` `Loewen` lesz, ami kielégíti a teljes szabályos kifejezést, és a két találati csoport a megfelelő szövegeket fogja tartalmazni.

Ha a „vagy” műveletet (`|`) alkalmazzuk két vagy több alternatívát elfogva, nem tudjuk, hogy melyik illeszkedett, ezért azt sem tudjuk, hogy melyik elfogási csoportból kell kinyernünk a szöveget. Természetesen végigjárhatjuk az összes csoportot, hogy megkeresse

ressük azt, amelyik nem üres, de ilyen helyzetben gyakran a találati objektum `lastindex` jellemzője adhatja meg a kívánt csoport sorszámát. Ennek szemléltetésére, illetve hogy kicsit több gyakorlatot szerezzünk a szabályos kifejezések használatában, még egy utolsó példát szemügyre veszünk.

Tegyük fel, hogy azt szeretnénk kideríteni, hogy milyen kódolást használ egy HTML-, XML- vagy Python-fájl. Megnyithatnánk a fájlt bináris módban, és egy `bytes` objektumba olvashatnánk belőle mondjuk az első 1000 bájtot, majd a fájlt bezárva megnézhetnénk a kódolást a `bytes` objektumban, és szöveges módban a talált kódolással vagy egy alapértelmezettel (például az UTF-8 kódolással) újra megnyithatnánk az állományt. A szabályoskifejezés-motor karakterláncként várja a szabályos kifejezéseket, a szöveg azonban, amelyre a szabályos kifejezést alkalmazzuk, `str`, `bytes` vagy `bytearray` objektum is lehet, és az utolsó kettő esetében minden függvény és tagfüggvény `bytes` objektumokat ad vissza karakterláncok helyett, a `re.ASCII` jelző pedig automatikusan bekapcsolódik.

A HTML-fájlok esetében a kódolást általában egy `<meta>` elemben határozzák meg (ha egyáltalán megadják), például így: `<meta http-equiv='Content-Type' content='text/html;charset=ISO-8859-1' />`. Az XML-fájlok alapértelmezés szerint UTF-8 kódolásúak, de ez felülbírálható, például így: `<?xml version="1.0" encoding="Shift_JIS"?>`. A Python 3-fájlok alapértelmezett kódolása ugyancsak az UTF-8, de ez ismét csak felülbírálható egy közvetlenül a shebang-sor után írt sorral, valahogy így: `#encoding: latin1` vagy `#-*-coding: latin1-*-`.

A kódolást a következőképpen deríthetjük ki, amennyiben a `binary` változó egy `bytes` objektum, amely egy HTML-, XML- vagy Python-fájl első 1000 bájtját tartalmazza:

```
match = re.search(r"""(?<![-\w])                    #1
                      (?:(?:en)?coding|charset)     #2
                      (?:=(["']))?([-\w]+)(?(1)\1)  #3
                      |:\s*([-\w]+))""".encode("utf8"),
                  binary, re.IGNORECASE|re.VERBOSE)
encoding = match.group(match.lastindex) if match else b"utf8"
```

Ahhoz, hogy egy `bytes` objektumban kereshessünk, meg kell adnunk egy mintát, ami ugyancsak egy `bytes` objektum. Ebben az esetben a kényelem kedvéért egy nyers karakterláncot szeretnénk használni, ezért azt is használunk, és a `re.search()` függvény első paramétereként átadva alakítjuk `bytes` objektummá.

Magának a szabályos kifejezésnek az első része egy utófeltételezés, ami azt mondja ki, hogy a találatot nem előzheti meg kötőjel vagy szókarakter. A kifejezés második része az „encoding”, „coding” vagy „charset” szóra illeszkedik, és így is írhattuk volna: `(?:encoding|coding|charset)`. A harmadik részt két sorra tördeltük, hogy kihangsú-

lyozzuk, hogy két „vagy-vagy” részből áll – `=([„'])?([-\w]+)(?(1)\1)` és `:\s*([-\w]+)` –, amelyek közül csak az egyik illeszkedhet. Az első egy egyenlőségjel, amelyet egy vagy több szókarakter vagy kötőjel követ (ezek megegyező idézőjelek között is állhatnak, amit egy feltételes illesztéssel vizsgálunk), a második pedig egy kettőspont, utána egy nem kötelező üreshely, majd egy vagy több szókarakter vagy kötőjel. (Emlékezzünk vissza, hogy a kötőjel a karakterosztályokon belül literális értelmű, amennyiben első karakterként szerepel, más esetben azonban egy olyan karaktertartományt jelöl, mint például a `[0-9]`).

12.7. táblázat *A találati objektumok jellemzői és tagfüggvényei*

Utasításforma	Leírás
`m.end(`*g*`)`	A találat végének helyét adja vissza a szövegen belül a *g* csoport számára (ha megadjuk a csoportot), illetve a 0. csoport esetében a teljes találatot. Ha a csoport nem vett részt az illesztésben, a visszatérési érték –1 lesz.
`m.endpos`	A keresés végének helye (a szöveg vége vagy a `match()`, illetve a `search()` függvénynek átadott *vég*).
`m.expand(`*s*`)`	Az *s* karakterláncot adja vissza, a fogásjelzőket (`\1`, `\2`, `\g<`*név*`>` és hasonlók) a megfelelő fogásokra cserélve.
`m.group(`*g*`,...)`	A *g* sorszámozott vagy nevesített elfogási csoportot adja vissza. Ha egynél több csoportot adunk meg, a megfelelő elfogási csoportok rendezett sorozatát kapjuk vissza (a teljes találat a 0. csoport).
`m.groupdict(`*alapértelmezés*`)`	Az összes nevesített elfogási csoport szótárát adja vissza, amelyben a nevek a kulcsok, a fogások pedig az értékek. Ha megadjuk az alapértelmezést, akkor az illesztésben részt nem vevő elfogási csoportok ezt az értéket kapják.
`m.groups(`*alapértelmezés*`)`	Az 1-estől az összes elfogási csoport rendezett sorozatát adja vissza. Ha megadjuk az alapértelmezést, akkor az illesztésben részt nem vevő elfogási csoportok ezt az értéket kapják.
`m.lastgroup`	A legmagasabb számú illeszkedő elfogási csoport neve, vagy `None`, ha nem volt ilyen, illetve ha nem használtunk neveket.
`m.lastindex`	A legmagasabb számú illeszkedő elfogási csoport sorszáma, vagy `None`, ha nem volt ilyen.
`m.pos`	A keresés kezdetének helye (a szövege eleje vagy a `match()`, illetve a `search()` függvénynek átadott *kezdet*).
`m.re`	Az a szabályos kifejezés, amely az adott találati objektumot adta.
`m.span(`*g*`)`	A találat kezdetének és végének helyét adja vissza a szövegen belül a *g* csoport számára (ha megadjuk a csoportot), illetve a 0. csoport esetében a teljes találatot. Ha a csoport nem vett részt az illesztésben, a visszatérési érték `(-1, -1)` lesz.

12.7. táblázat *A találati objektumok jellemzői és tagfüggvényei* *(folytatás)*

Utasításforma	Leírás
`m.start(g)`	A találat kezdetének helyét adja vissza a szövegen belül a `g` csoport számára (ha megadjuk a csoportot), illetve a 0. csoport esetében a teljes találatot. Ha a csoport nem vett részt az illesztésben, a visszatérési érték –1 lesz.
`m.string`	A `match()`, illetve a `search()` függvénynek átadott karakterlánc.

A `re.IGNORECASE` jelzőt használtuk, hogy ne azt kelljen írnunk, hogy `(?:(?:[Ee][Nn])?[Cc][Oo][Dd][Ii][Nn][Gg]|[Cc][Hh][Aa][Rr][Ss][Ee][Tt])`, illetve a `re.VERBOSE` jelzőt, hogy szépen elrendezhessük a szabályos kifejezést, és megjegyzéseket is fűzhessünk hozzá (ebben az esetben csak sorszámokat, hogy könnyebben hivatkozhassunk a kifejezés egyes részeire a szövegben).

Három elfogási csoportunk van; mindegyik a kifejezés harmadik részében: a `(["'])?`, ami a nem kötelező nyitó idézőjelet fogja el, a `([-\w+])`, ami egy egyenlőségjelet követő kódolást, és (a következő sorban) a második `([-\w+])`, ami egy kettőspontot követő kódolást. Minket csak a kódolás érdekel, ezért vagy a második, vagy a harmadik elfogási csoportot kell kinyernünk, amelyek közül csak az egyik illeszkedhet, mivel „vagy” kapcsolatban állnak egymással. Az utolsó *illeszkedő* elfogási csoport (ebben a példában találat esetén ez a 2. vagy 3. csoport) sorszámát a `lastindex` jellemző tárolja, amelyből kiolvassuk a megfelelőt, illetve ha nem volt illeszkedés, egy alapértelmezett kódolást alkalmazunk.

Most, hogy a `re` modul leggyakrabban használt szolgáltatásai közül az összeset láttuk működés közben, a rész lezárásaként csupán megemlítünk még egy utolsó függvényt. A `re.split()` függvény (vagy a szabályoskifejezés-objektum `split()` tagfüggvénye) karakterláncok felbontására képes egy szabályos kifejezés alapján. Gyakori, hogy egy szöveget a szóközöknél kell szétvágni, hogy megkapjuk a szavak listáját.
Ezt a `re.split(r"\s+", text)` hívással hajthatjuk végre, amely egy szólistát ad vissza (pontosabban egy karakterlánc-listát, amelyben minden karakterlánc illeszkedik az `\s+` kifejezésre). A szabályos kifejezések nagyon hatékonyak és hasznosak, és ha egyszer megtanultuk a használatukat, könnyen úgy tekinthetünk minden szövegfeldolgozási feladatra, mint aminek a megoldása egy szabályos kifejezést igényel. Néha azonban a karakterlánc-függvények használata is elegendő, sőt talán helyénvalóbb. A szóközöknél történő szétvágás például ugyanilyen könnyen megoldható a `text.split()` függvénnyel is, mivel a `str.split()` tagfüggvény (így, vagy első paraméterként a `None` értékkel) alapértelmezett viselkedése, hogy az `\s+`-nál vág.

Összefoglalás

A szabályos kifejezések hatékony módját kínálják annak, hogy olyan karakterláncokat keressünk szövegekben, amelyek egy bizonyos mintára illeszkednek, illetve hogy ezeket a karakterláncokat más karakterláncokra cseréljük, amelyek maguk is függhetnek attól, hogy milyen egyezést kerestünk.

Ebben a fejezetben láttuk, hogy a legtöbb karaktert literális értelemben illesztjük, alapértelmezés szerint egyszer. Azt is megtanultuk, hogyan határozhatunk meg karakterosztályokat – illesztendő karakterek egy halmazát –, és hogyan negálhatjuk ezeket a halmazokat, illetve adhatunk meg bennük karaktertartományokat anélkül, hogy minden karaktert külön-külön ki kellene írnunk.

Azt is megtudtuk, hogy miként láthatjuk el mennyiségjelzővel a kifejezéseket, hogy meghatározzuk az illesztések pontos számát, vagy hogy a kifejezés legalább és legfeljebb hányszor illeszkedhet, valamint megtanultuk a mohó és a nem mohó illesztés módját. Arról is tanultunk, hogy miként képezhetünk csoportot egy vagy több kifejezésből, hogy egy egységként adhassuk meg az illesztéseik számát, illetve foghassuk el őket.

A fejezetben azt is megmutattuk, hogy az illesztést hogyan szabályozhatjuk különféle feltételezésekkel, például pozitív vagy negatív előretekintéssel és visszatekintéssel, illetve különböző jelzőkkel, amelyekkel például befolyásolhatjuk a pont jelentését, vagy hogy az illesztés során különbözőnek tekintjük-e a kis- és nagybetűket.

A fejezet utolsó részében azt mutattuk meg, hogy miként használhatjuk a szabályos kifejezéseket a Python-programok környezetében. Megtanultuk, hogyan használhatjuk a `re` modul által biztosított függvényeket, illetve a lefordított szabályos kifejezésekből, valamint a találati objektumokból elérhető tagfüggvényeket. Arról is szót ejtettünk, hogy miként cserélhetjük le a találatokat literális karakterláncokkal, visszirányú hivatkozásokat tartalmazó literális karakterláncokkal, valamint függvényhívások vagy lambdakifejezések eredményével, és megtanultuk, hogyan könnyíthetjük meg a szabályos kifejezések későbbi módosítását nevesített fogások és megjegyzések segítségével.

Gyakorlatok

1. Sok esetben (például egyes webes űrlapokon) a felhasználóknak meg kell adniuk egy telefonszámot, és előfordul, hogy számukra bosszantó módon csak egy adott formátumot fogadnak el. Írjunk olyan programot, amely amerikai telefonszámokat olvas be, a három számjegyű körzetszámot és a hét számjegyből álló helyi számot a következő formátumokban elfogadva: tíz számjegyként; kötőjelekkel, illetve szóközökkel tagolva; valamint a körzetszámot nem kötelező módon zárójelbe téve. A következő alakok közül tehát mindegyik érvényes: 555-555-5555, (555)5555555,

(555) 555 5555 és 5555555555. A telefonszámokat a `sys.stdin`-ről olvassuk be, és mindegyiket írjuk ki az `echo` utasítással „(555) 555 5555" formában, az érvénytelen számok esetében pedig adjunk hibaüzenetet.
A fenti telefonszámokat illesztő szabályos kifejezés mintegy nyolc sorban megírható (bőbeszédű módban), és meglehetősen egyszerű. Egy megoldást a `phone.py` fájlban láthatunk, amely körülbelül huszonöt sor hosszú.

2. Írjunk egy olyan rövid programot, amely egy a parancssorban megadott XML- vagy HTML-fájlt olvas be, és minden jellemzővel rendelkező elem esetében kiírja az elem nevét, alatta a jellemzőivel. Példaként álljon itt egy részlet a program kimenetéből, amelyet akkor kapunk, ha a Python dokumentációjának valamelyik `index.html` fájlját olvassuk be:

```
html
    xmlns = http://www.w3.org/1999/xhtml
meta
    http-equiv = Content-Type
    content = text/html; charset=utf-8
li
    class = right
    style = margin-right: 10px
```

Az egyik megközelítés az lehet, ha két szabályos kifejezést használunk: egyet az elemek és jellemzőik elfogására, egy másikat pedig az egyes jellemzők nevének és értékének kinyerésére. A jellemzőértékek aposztrófok vagy (kettős) idézőjelek között is állhatnak (mely esetben üreshelyeket, valamint olyan idézőjeleket is tartalmazhatnak, amelyek nem egyeznek meg a közrezárásukra használttal), de lehetnek idézőjel nélküliek is (ebben az esetben azonban nem lehet bennük üreshely vagy idézőjel). Valószínűleg az a legegyszerűbb, ha egy olyan szabályos kifejezés megírásával kezdjük, amely külön kezeli az idézőjeles és az idézőjel nélküli értékeket, majd a két szabályos kifejezést egyetlen kifejezésben egyesítjük, hogy mindkét esetet lefedjük. Az a legjobb, ha nevesített csoportokat használunk, mert a szabályos kifejezés így olvashatóbb lesz. A feladat nem egyszerű, különösen mert karakterosztályokon belül nem használhatunk visszirányú hivatkozásokat.
Egy megoldást az `extract_tags.py` fájlban láthatunk, amely kevesebb mint 35 sor hosszú. Az elemeket és a jellemzőket kinyerő szabályos kifejezés mindössze egy sor, a jellemzőneveket és -értékeket kiolvasó kifejezés azonban féltucat sorból áll, és „vagy" művelet, (két egymásba ágyazott) feltételes illesztés, valamint mohó és nem mohó mennyiségjelzők is szerepelnek benne.

13

Bevezetés a grafikus felhasználói felület programozásába

- Párbeszédablakos programok
- Főablakos programok

A Python alapállapotban nem támogatja a grafikus felhasználói felület (GUI, Graphical User Interface) programozását, de ez nem jelent gondot, hiszen más nyelveken számos olyan GUI-programkönyvtárat írtak, amelyeket a Python-programozók felhasználhatnak. Ezt az teszi lehetővé, hogy sok GUI-könyvtárhoz létezik Python-*burkoló* vagy *-kötés*, vagyis olyan csomagok és modulok, amelyeket ugyanúgy tölthetünk be és használhatunk, mint bármely más Python-csomagot és modult, csak éppen olyan szolgáltatásokat tesznek elérhetővé, amelyek nem Python-könyvtárakban találhatók.

A Python alapkönyvtárának része a Tcl/Tk: a Tcl egy szinte nyelvtanmentes parancsnyelv, a Tk pedig egy GUI-könyvtár, amit Tcl-ben és C-ben írtak. A Python `tkinter` modulja a Tk GUI-könyvtárhoz biztosít Python-kötéseket. A Tk-nak három előnye van a Pythonból hozzáférhető más GUI-könytárakkal szemben. Először is, a Pythonnal együtt települ, tehát mindig elérhető. Másodszor, kis méretű (még a Tcl-lel együtt is). Harmadszor pedig, része az IDLE környezetnek, mely környezet nagyon hasznos, ha kísérletezni szeretnénk a Pythonnal, illetve Python-programokat szeretnénk szerkeszteni és hibakeresést végezni bennük.

Sajnos a Tk megjelenése a 8.5-ös változat előtt igencsak elavult volt, és a Tk igen korlátozott elemkészlettel („widget”-ekkel vagy „mütyürökkel”, a Windows nyelvén „vezérlőkkel” vagy „tárolókkal”) rendelkezett. A Tk-ban ugyan meglehetősen könnyű egyéni vezérlőket létrehozni más vezérlők egyesítésével, ám a Tk nem nyújt semmilyen közvetlen módot arra, hogy a semmiből hozzunk létre egyéni vezérlőket, és úgy rajzoljuk meg azokat, ahogy csak szeretnénk. További Tk-megfelelő vezérlőkhöz juthatunk a Tix könyvtár használatával, amely szintén része a Python alapkönyvtárának, de a Windowstól eltérő rendszereken nem mindig áll rendelkezésre – például az Ubuntun, amely a könyv írásakor csak nem támogatott bővítményként tette lehetővé a használatát. Ezen kívül mind a Tk, mind

a Tix híján van egy Python-központú leírásnak – a dokumentáció nagy részét Tcl/Tk-programozóknak szánták, ezért a Tcl-ben járatlan programozók számára nehezen megfejthető lehet.*

Ha olyan grafikus felületű programokat szeretnénk fejleszteni, amelyeknek minden asztali Python-platformon (vagyis Windowson, Mac OS X-en és Linuxon is) futniuk kell, és csak a Python alaptelepítése áll a rendelkezésünkre, kiegészítő könyvtárak nélkül, egyetlen lehetőségünk van: a Tk használata.

Ha külső könyvtárakat is használhatunk, a lehetőségek száma jelentősen megnő. Választhatjuk például a WCK (Widget Construction Kit, vagyis „vezérlőépítő készlet”, `www.effbot.org/zone/wck.htm`) könyvtárat, amely további Tk-megfelelő szolgáltatásokat kínál, például olyan egyéni vezérlők létrehozásának lehetőségét, amelyeknek a tartalmát kóddal határozzuk meg.

A Tk-t mellőző lehetőségek két csoportra oszthatók: azokra, amelyek csak egy adott rendszeren működnek, illetve azokra, amelyek rendszerfüggetlenek. A rendszerfüggő GUI-könyvtárak elérhetővé teszik az adott rendszerre jellemző szolgáltatásokat, de ennek az az ára, hogy a rendszerhez kötnek minket. A három legszélesebb körben elterjedt, Python-kötésekkel rendelkező rendszerfüggetlen GUI-könyvtár a PyGtk (`www.pygtk.org`), a PyQt (`www.riverbankcomputing.com/software/pyqt`) és a wkPython (`www.wkpython.org`). Mindhárom lényegesen több vezérlőt kínál, mint a Tk, szebb megjelenésű grafikus felületet állít elő (bár a Tk 8.5 megjelenésével a különbség csökkent), és lehetővé teszi a kóddal megrajzolt egyéni vezérlők létrehozását is. Ezen felül mindegyiket könnyebb megtanulni és használni, mint a Tk-t; a leírásuk is bővebb és sokkal inkább Python-központú, mint a Tk-é; és általában véve a PyGtk-t, a PyQt-t vagy a wxPythont használó programok kevesebb kódot igényelnek, és jobb eredményt adnak, mint a Tk-val írtak.

Korlátai és bosszantó hiányosságai ellenére azonban a Tk segítségével is készíthetünk hasznos grafikus felületű programokat – a Python világában az IDLE a legismertebb. Továbbá könyvünk írásakor a Tk fejlesztése meglódult, és a témákat kínáló Tk 8.5 már olyan Tk-programok készítését tette lehetővé, amelyek sokkal jobban hasonlítanak az adott rendszer sajátjaira, valamint számos régen várt új vezérlő is belekerült.

Ebben a fejezetben csupán ízelítőt nyújthatunk a Tk-programozásból. Ha komoly grafikus felületeket szeretnénk építeni, a legjobb, ha átugorjuk ezt a fejezetet (mivel itt a grafikus felület programozásának csak a hagyományos Tk-megközelítését mutatjuk be), és valamelyik

---

*Az egyetlen Python/Tk-könyv, amiről a szerzőnek tudomása van, John Grayson *Python and Tkinter Programming* című, 2000-ben kiadott (és ezért egyes részeit tekintve elavult) műve (ISBN 1884777813). Tcl/Tk témában Brent Welch és Ken Jones *Practical Programming in Tcl and Tk* című kötetét (ISBN 0130385603) érdemes elolvasni. A Tcl/Tk teljes leírását a www.tcl.tk internetcímen érhetjük el.

külső könyvtárat vesszük igénybe. Ha azonban a Tk az egyetlen választási lehetőségünk, akkor legalább annyi tudással kell rendelkeznünk a Tcl nyelvről, amennyi elég ahhoz, hogy megértsük a Tk leírását.

A következőkben a Tk segítségével két grafikus felületű programot fogunk elkészíteni. Az első egy igen kis méretű párbeszédablakos program, amely kamatos kamatok kiszámítását végzi. A második, főablakot használó program már kidolgozottabb lesz, és könyvjelzők (nevek és URL-ek) listáját kezeli majd. Mivel az adatok egyszerűek lesznek, semmi nem fogja elterelni a figyelmünket a grafikus felület programozását érintő kérdésekről. A könyvjelző-kezelő programot bemutató részben látni fogjuk, hogyan hozhatunk létre egy egyéni párbeszédablakot, illetve egy menükkel és eszköztárakkal rendelkező főablakot, valamint azt is, hogy miként kombinálhatjuk a kettőt egy teljes értékű, működőképes program előállításához. Először azonban át kell tekintenünk a grafikus felület programozásának néhány alapkérdését, mivel a grafikus programok készítése némileg eltér a konzolos programokétól.

A Pythonban a konzolos programok és a modulfájlok kiterjesztése mindig `.py`, a grafikus felületű programok esetében azonban a `.pyw` kiterjesztést használjuk (kivéve a modulfájlokat, amelyek mindig `.py` kiterjesztésűek). Mind a `.py`, mind a `.pyw` megfelelően működik Linuxon is, Windowson viszont a `.pyw` kiterjesztés biztosítja, hogy a `python.exe` helyett a `pythonw.exe` legyen az értelmező, amely gondoskodik róla, hogy amikor grafikus felületű Python-programot hajtunk végre, ne jelenjen meg egy szükségtelen konzolablak. A Mac OS X a Windowshoz hasonlóan működik, és a GUI-programokhoz a `.pyw` kiterjesztést használja.

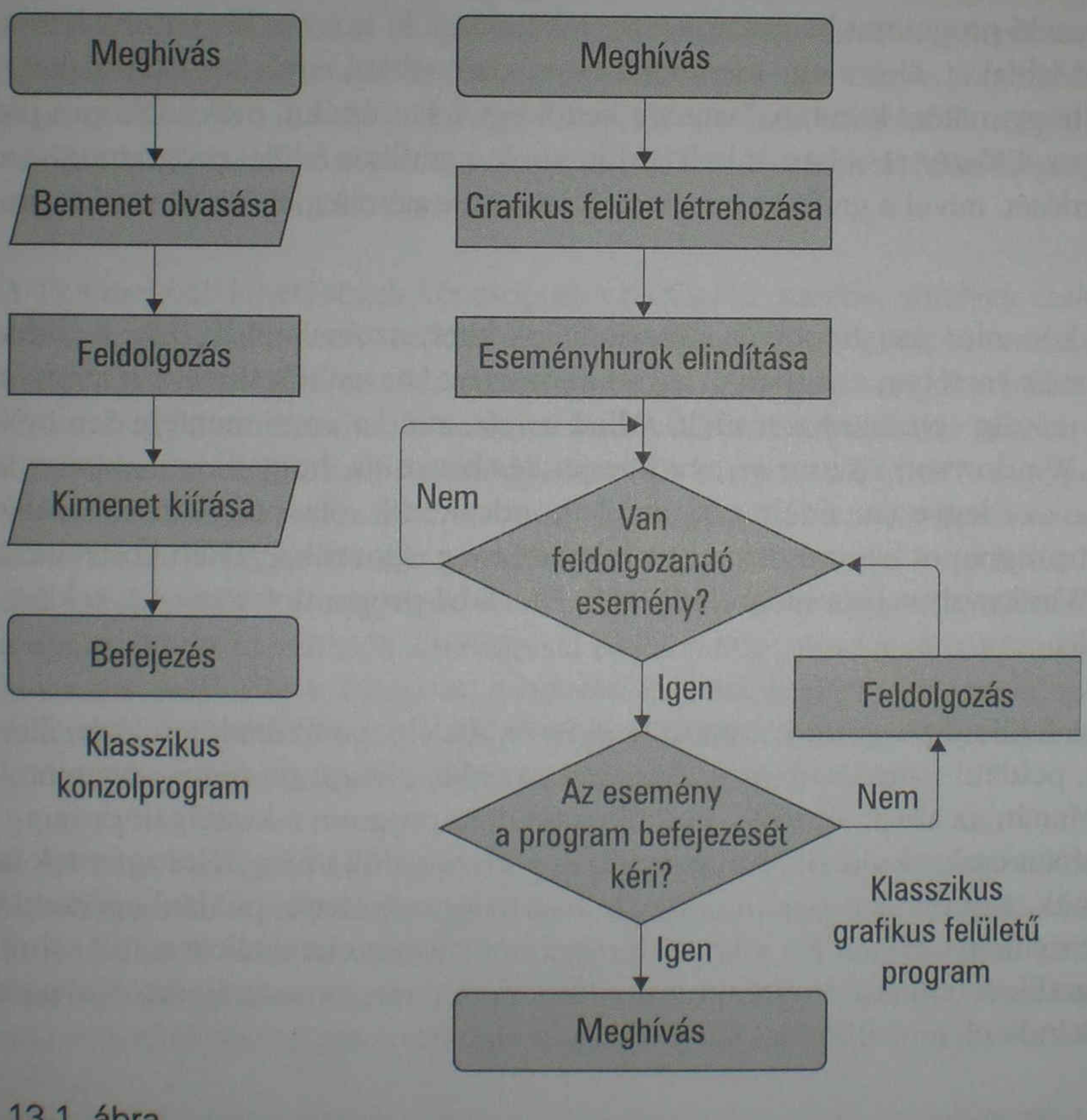
Amikor grafikus felületű programot futtatunk, először általában a főablak jön létre, illetve annak vezérlői, például a menüsor, az eszköztárak, az ablak központi része, valamint az állapotsor. Miután az ablak létrejött, a grafikus felületű program a kiszolgálóprogramokhoz hasonlóan csak várakozik. Amíg azonban a kiszolgálók az ügyfélprogramok csatlakozására várnak, a grafikus programok egy felhasználói műveletre, például egérkattintásra vagy billentyűlenyomásra. Ezt a konzolprogramokkal összehasonlítva a 13.1. ábra szemlélteti. A grafikus felületű program ugyanakkor nem passzívan várakozik, hanem egy *eseményhurkot* indít el, ami álkóddal leírva valahogy így fest:

```
while True:
    event = getNextEvent()
    if event:
        if event == Terminate:
        break
    processEvent(event)
```

Amikor a felhasználó valamilyen műveletet hajt végre a programban, vagy más esemény történik, például lejár egy időzítő, vagy a program ablaka aktívvá válik (például mert egy másik programot bezártak), egy esemény jön létre a GUI-könyvtárban, és bekerül

az eseménysorba. A program eseményhurka folyamatosan ellenőrzi, hogy van-e feldolgozandó esemény, és ha talál ilyet, feldolgozza azt (vagy feldolgozásra átadja az eseményhez társított függvénynek vagy tagfüggvénynek).

GUI-programozóként a GUI-könyvtárra bízhatjuk az eseményhurok létrehozását. A mi feladatunk az, hogy létrehozzuk azokat az osztályokat, amelyek a programunk által igényelt ablakokat és vezérlőket jelképezik, és ellássuk azokat a felhasználói műveletekre válaszolni képes tagfüggvényekkel.



13.1. ábra

Konzolos programok és grafikus felületű programok

Párbeszédablakos programok

Az első program, amelyet megvizsgálunk, az Interest (Kamat) nevű kamatszámító program. Ez egy párbeszédablakos program (ami úgy értendő, hogy nincsenek menüi), amellyel a felhasználó kamatos kamatok kiszámítását hajthatja végre. A programot a 13.2. ábrán láthatjuk.



13.2. ábra
Az Interest program

A legtöbb objektumközpontú grafikus programban egy egyéni osztályt használnak, amely egyetlen főablakot vagy párbeszédablakot ábrázol, a rajta elhelyezett vezérlők többsége pedig szabványos vezérlők egy-egy példánya; ilyen például a gomb vagy a jelölőnégyzet, amelyet a könyvtár kínál. A legtöbb rendszerfüggetlen GUI-könyvtárhoz hasonlóan a Tk nem igazán tesz különbséget az ablakok és a vezérlők között, hanem az ablakot egyszerűen olyan vezérlőnek tekinti, amelynek nincs szülővezérlője (vagyis nem egy másik vezérlő belsejében található). A szülővezérlővel nem rendelkező vezérlők (vagyis ablakok) automatikusan keretet és díszítőelemeket (például címsort és bezáró gombot) kapnak, és általában más vezérlőket tartalmaznak.

A legtöbb vezérlő egy másik vezérlő gyermekeként jön létre (és a szülő belsejében tárolódik), míg az ablakok a `tkinter.Tk` objektum gyermekei. Ez egy olyan objektum, amely az alkalmazást jelképezi – hamarosan visszatérünk rá. A vezérlők és az ablakok (amelyeket felső szintű vezérlőknek is hívnak) megkülönböztetésén túl a szülő-gyermek viszonyok biztosítják, hogy a vezérlők a megfelelő sorrendben törlődjenek, illetve hogy a szülő törlésekor a gyermekvezérlői is automatikusan megsemmisüljenek.

A felhasználói felületet az előkészítő kód hozza létre (tehát ez végzi el a vezérlők hozzáadását és elrendezését, valamint az egér- és billentyűkötések meghatározását), míg a többi tagfüggvény a felhasználó műveleteire válaszol. A Tk lehetővé teszi egyéni vezérlők létrehozását is, vagy úgy, hogy egy kész vezérlőből, például a `tkinter.Frame`-ből alosztályt származtatunk, vagy úgy, hogy egy szokványos osztályt hozunk létre, és jellemzőként vezérlőket adunk hozzá. Mi most az alosztályok létrehozásának módszerét fogjuk alkalmazni, a következő példában azonban mindkét megközelítést bemutatjuk.

Mivel az Interest programnak csak egyetlen főablaka van, egyetlen osztályban fogjuk megvalósítani. Kezdjük az osztály előkészítő függvényének vizsgálatával, amelyet öt részre bontottunk, mivel elég hosszú:

```
class MainWindow(tkinter.Frame):

    def __init__(self, parent):
        super(MainWindow, self).__init__(parent)
        self.parent = parent
        self.grid(row=0, column=0)
```

Először az alaposztály előkészítését hajtjuk végre. (Ideális esetben a `super().__init__(parent)` kódot használhatnánk, de kötetünk írásakor ez nem működött, ezért a `super()`-nek kifejezetten át kellett adnunk paraméterként a szülőosztályt és a `self` objektumot.) Ezt követően elteszünk egy másolatot a szülőről későbbi használatra. Abszolút hely és méret megadása helyett elrendezéskezelők segítségével más vezérlőkön belül rendezzük el a vezérlőket. A `grid()` hívás a keret elhelyezéséről gondoskodik a rács-elrendezés-kezelő használatával. Minden látható vezérlőt el kell rendeznünk, még a legfelső szintűeket is. A Tk számos elrendezéskezelőt kínál, de a rács (grid) a legkönnyebben érthető és használható, bár az olyan legfelső szintű elrendzések esetében, ahol csak egy vezérlőt kell elhelyeznünk, a csomagoló (packer) elrendezéskezelőt is használhatnánk; ekkor a `grid(row=0,column=0)` helyett a `pack()` hívást kellene használnunk ugyanennek a hatásnak az eléréséhez.

```
self.principal = tkinter.DoubleVar()
self.principal.set(1000.0)
self.rate = tkinter.DoubleVar()
self.rate.set(5.0)
self.years = tkinter.IntVar()
self.amount = tkinter.StringVar()
```

A Tk lehetővé teszi, hogy vezérlőkhöz társított változókat határozzunk meg. Ha egy ilyen változó értékét a program kódja megváltoztatja, a változás tükröződni fog a hozzá társított vezérlőn, és ehhez hasonlóan az érték felhasználói módosítása a vezérlőben is megváltoztatja a változó értékét. Itt két „double" típusú változót hoztunk létre (a „double" típusú változók lebegőpontos – `float` – értékeket tárolnak), valamint egy egész értékűt és egy karakterlánc értékűt, és kettőnek a kezdőértékét is beállítottuk.

```
principalLabel = tkinter.Label(self, text="Principal $:",
                               anchor=tkinter.W, underline=0)
principalScale = tkinter.Scale(self, variable=self.principal,
        command=self.updateUi, from_=100, to=10000000,
        resolution=100, orient=tkinter.HORIZONTAL)
rateLabel = tkinter.Label(self, text="Rate %:", underline=0,
                          anchor=tkinter.W)
rateScale = tkinter.Scale(self, variable=self.rate,
        command=self.updateUi, from_=1, to=100,
        resolution=0.25, digits=5, orient=tkinter.HORIZONTAL)
yearsLabel = tkinter.Label(self, text="Years:", underline=0,
                           anchor=tkinter.W)
yearsScale = tkinter.Scale(self, variable=self.years,
        command=self.updateUi, from_=1, to=50,
        orient=tkinter.HORIZONTAL)
amountLabel = tkinter.Label(self, text="Amount $",
                            anchor=tkinter.W)
actualAmountLabel = tkinter.Label(self,
        textvariable=self.amount, relief=tkinter.SUNKEN,
        anchor=tkinter.E)
```

Az előkészítő kódnak ez az a része, ahol létrehozzuk a vezérlőket. A `tkinter.Label` vezérlő csak olvasható szöveget jelenít meg a felhasználó számára. Ugyanúgy, mint a többi vezérlőt, ezt is egy szülővel hozzuk létre (ebben az esetben – a szokásos módon – a szülő a tartalmazó vezérlő), a vezérlő megjelenésének és viselkedésének különböző jellemzőit pedig kulcsszó-paraméterekkel állítjuk be. Beállítottuk a `principalLabel` szövegét, az `anchor` tulajdonságának pedig a `tkinter.W` értéket adtuk, aminek a hatására a szöveg címke szövege balra igazodik (W=west, vagyis „nyugat"). Az `underline` paramétert annak meghatározására használjuk, hogy a címkében melyik karakter legyen az aláhúzott, ami egy *gyorsbillentyűt* jelöl (például ALT+P). Később azt is látni fogjuk, hogy miként bírhatjuk működésre a gyorsbillentyűt. (A gyorsbillentyű olyan billentyűsorozat, amelynek az alakja ALT+BETŰ, ahol a BETŰ egy aláhúzott betűt jelent, amellyel a billentyűfókuszt a gyorsbillentyűhöz kapcsolt vezérlőre állíthatjuk, ami rendszerint a gyorsbillentyűvel rendelkező címke alatti vagy attól jobbra található vezérlő.)

A `tkinter.Scale` vezérlők szülőjeként a szokásos módon a `self` objektumot adjuk meg, és mindegyikhez egy változót társítunk. Ezen kívül parancsként hozzájuk kapcsoljuk egy függvény (illetve ebben az esetben egy tagfüggvény) objektumhivatkozását – ez a tagfüggvény fog automatikusan meghívódni, ha a csúszka (skála) értéke megváltozik –, beállítjuk a skála minimumát (`from_`, a végén aláhúzással, mert a `from` önmagában egy kulcsszó) és maximumát (`to`), valamint vízszintesen tájoljuk a csúszkákat. Egyes csúszkák esetében felbontást (resolution; itt: lépésköz) is megadunk, a `rateScale` számára pedig azt is meghatározzuk, hogy hány számjegyet kell tudnia megjeleníteni.

Az `actualAmountLabel`-hez ugyancsak társítunk egy változót, hogy később egyszerűen módosíthassuk a címke által megjelenített szöveget. Ezt a címkét besüllyesztjük (`relief`), hogy képileg jobban illeszkedjen az ablakra.

```
principalLabel.grid(row=0, column=0, padx=2, pady=2,
                    sticky=tkinter.W)
principalScale.grid(row=0, column=1, padx=2, pady=2,
                    sticky=tkinter.EW)
rateLabel.grid(row=1, column=0, padx=2, pady=2,
               sticky=tkinter.W)
rateScale.grid(row=1, column=1, padx=2, pady=2,
               sticky=tkinter.EW)
yearsLabel.grid(row=2, column=0, padx=2, pady=2,
                sticky=tkinter.W)
yearsScale.grid(row=2, column=1, padx=2, pady=2,
                sticky=tkinter.EW)
amountLabel.grid(row=3, column=0, padx=2, pady=2,
                 sticky=tkinter.W)
actualAmountLabel.grid(row=3, column=1, padx=2, pady=2,
                       sticky=tkinter.EW)
```

Miután létrehoztuk a vezérlőket, el kell rendeznünk őket. A használt rácselrendezést a 13.3. ábra mutatja.

principalLabel	principalScale
rateLabel	rateScale
yearsLabel	yearsScale
amountLabel	actualAmountLabel

13.3. ábra

Az Interest program elrendezése

Minden vezérlő ismeri a `grid()` tagfüggvényt (valamint más elrendezőfüggvényeket, például a `pack()`-et is). A `grid()` meghívásával a vezérlőt a szülőn belül rendezzük el, a megadott számú sorban és oszlopban. További kulcsszó-paraméterekkel (`rowspan` és `columnspan`) úgy is beállíthatjuk a vezérlőket, hogy több sort, illetve oszlopot fogjanak át, a `padx` és `pady` kulcsszó-paraméterekben egész képpontértékeket meghatározva pedig margót adhatunk köréjük (a `padx` a bal és jobb margót, a `pady` a felső és alsó margót állítja be). Ha egy vezérlőnek több helyet foglalunk, mint amennyire szüksége van, a `sticky` („ragadós") paraméterrel határozhatjuk meg, hogy mi történjen a fölös hellyel – ha nem adjuk meg, a vezérlő a számára lefoglalt terület közepére kerül. Az első oszlop minden vezérlőjét `sticky`-re állítottuk, mégpedig `tkinter.EW` (east-west, „kelet-nyugat") értékkel, aminek a hatására megnyúlnak, hogy a teljes rendelkezésükre álló szélességet kitöltsék.

Minden vezérlőt helyi változókban tárolunk, de szemétgyűjtésre nem kell kijelölnünk őket, mert a szülő–gyermek kapcsolatok gondoskodnak róla, hogy ne törlődjenek, amikor kikerülnek a hatókörből az előkészítő kód végén, hiszen mindegyikünknek a főablak a szülője. A vezérlőket néha példányváltozóként hozzuk létre, például akkor, ha az előkészítő függvényen kívül kell hivatkoznunk rájuk, de itt a vezérlőkhöz társított változókat (`self.principal`, `self.rate` és `self.years`) tettük példányváltozókká, mert ezeket fogjuk az előkészítő függvényen kívül használni.

```
principalScale.focus_set()
self.updateUi()
parent.bind("<Alt-p>", lambda *ignore: principalScale.focus_set())
parent.bind("<Alt-r>", lambda *ignore: rateScale.focus_set())
parent.bind("<Alt-y>", lambda *ignore: yearsScale.focus_set())
parent.bind("<Control-q>", self.quit)
parent.bind("<Escape>", self.quit)
```

Az előkészítő függvény végén a billentyűfókuszt a `principalScale` vezérlőre állítjuk, hogy amint a program elindult, a felhasználó beállíthassa a nyitó pénzösszeget. Ez után meghívjuk a `self.updateUi()` tagfüggvényt, hogy kiszámítsuk a kezdő végösszeget.

A következő lépés néhány billentyűkötés beállítása. (Sajnos a *kötés* szónak három különböző jelentése van. Az első a változókötés, amikor egy nevet, vagyis egy objektumhivatkozást kapcsolunk egy objektumhoz; a második a billentyűkötés, amikor egy billentyűműveletet, például egy billentyű lenyomását vagy felengedését társítjuk egy függvényhez vagy tagfügg-

vényhez, amelyet akkor kell meghívni, amikor a műveletet végrehajtják; a harmadik pedig a könyvtárkötés, ami azt a ragasztókódot jelenti, amely egy a Pythontól eltérő nyelven írt könyvtárat Python-modulokon keresztül elérhetővé tesz a Python-programozók számára.) A billentyűkötések létfontosságúak a fogyatékkal élők számára, akik nehezen vagy egyáltalán nem tudják használni az egeret, és azoknak a munkáját is sokkal kényelmesebbé teszik, akik nem akarják igénybe venni az egeret, mert gyors gépelők lévén az egér használata csak lelassítaná őket.

Az első három billentyűkötéssel a billentyűfókuszt állítjuk egy csúszkára. A `principalLabel` feliratként például a `Principal $:` szöveget állítottuk be, az aláhúzást pedig a 0. karakterre, a címke tehát `Principal $:` formában jelenik meg. Az első billentyűkötés beállításával a billentyűfókusz az Alt+P billentyűk lenyomásának hatására a `principalScale` csúszkára kerül. Ugyanígy működik a másik két billentyűkötés is. Megjegyzendő, hogy a `focus_set()` tagfüggvényt nem közvetlenül kötjük a vezérlőkhöz, mert amikor egy eseménykötés eredményeként függvények vagy tagfüggvények meghívására kerül sor, első paraméterként megkapják az őket hívó eseményt, nekünk viszont nincs szükségünk erre az eseményre. Ezért egy lambdafüggvényt használunk, amely elfogja az eseményt, és meghívja a tagfüggvényt, de a nemkívánatos paraméter nélkül.

Két *billentyűparancsot* is meghatároztunk – ezek olyan billentyűkombinációk, amelyek egy bizonyos műveletet indítanak el. Itt a Ctrl+Q, illetve az Esc billentyűparancsokat adtuk meg, és mindkettőt a `self.quit()` tagfüggvényhez kötöttük, amely tisztán befejezi a program futtatását.

Billentyűkötéseket az egyes vezérlők számára is beállíthatunk, de itt mindet a szülőhöz (az alkalmazáshoz) adtuk meg, hogy attól függetlenül működjenek, hogy hol van éppen a billentyűfókusz.

A Tk `bind()` tagfüggvényét egérkattintások és billentyűleütések, sőt a programozó által meghatározott események kötésére is használhatjuk. A Ctrl-hez és az Esc-hez hasonló különleges billentyűknek a Tk-ben adott nevük van (`Control` és `Escape`), míg a közönséges betűk önmagukat jelölik. Billentyűsorozatot úgy adhatunk meg, hogy a részeit szögletes zárójelek közé írjuk, és kötőjelekkel választjuk el őket.

Most, hogy létrehoztuk és elrendeztük a vezérlőket, és beállítottuk a billentyűkötéseket, a program megjelenése és alapműködése a helyére került. A program megvalósításának teljessé tételéhez a következőkben azokat a tagfüggvényeket tekintjük át, amelyek a felhasználói műveletekre reagálnak.

```
def updateUi(self, *ignore):
    amount = self.principal.get() * (
             (1 + (self.rate.get() / 100.0)) ** self.years.get())
    self.amount.set("{0:.2f}".format(amount))
```

Ez a tagfüggvény minden alkalommal meghívódik, amikor a felhasználó megváltoztatja a tőke összegét, a kamatlábat vagy a futamidőt, mivel mindegyik csúszkához ez a parancs társul. A függvény csupán annyit tesz, hogy kiolvassa a csúszkához tartozó változó értékét, végrehajtja a kamatos kamat kiszámítását, és az eredményt (karakterláncként) a végösszeget kiíró címkéhez társított változóba írja. Ennek eredményeképpen a végösszeget kiíró címke mindig naprakész összeget mutat.

```
def quit(self, event=None):
    self.parent.destroy()
```

Ha a felhasználó úgy dönt, hogy kilép a programból (a Ctrl+Q vagy az Esc lenyomásával, vagy az ablak bezáró gombjára kattintva), ez a tagfüggvény hívódik meg. Mivel nincsenek mentendő adatok, csak utasítjuk a szülőt (az alkalmazás objektumát), hogy semmisítse meg önmagát. A gyermekek törléséről a szülő gondoskodik – ez az összes ablakot jelenti, amelyek pedig az összes vezérlőjüket pusztítják el –, így tisztán kiléphetünk a programból.

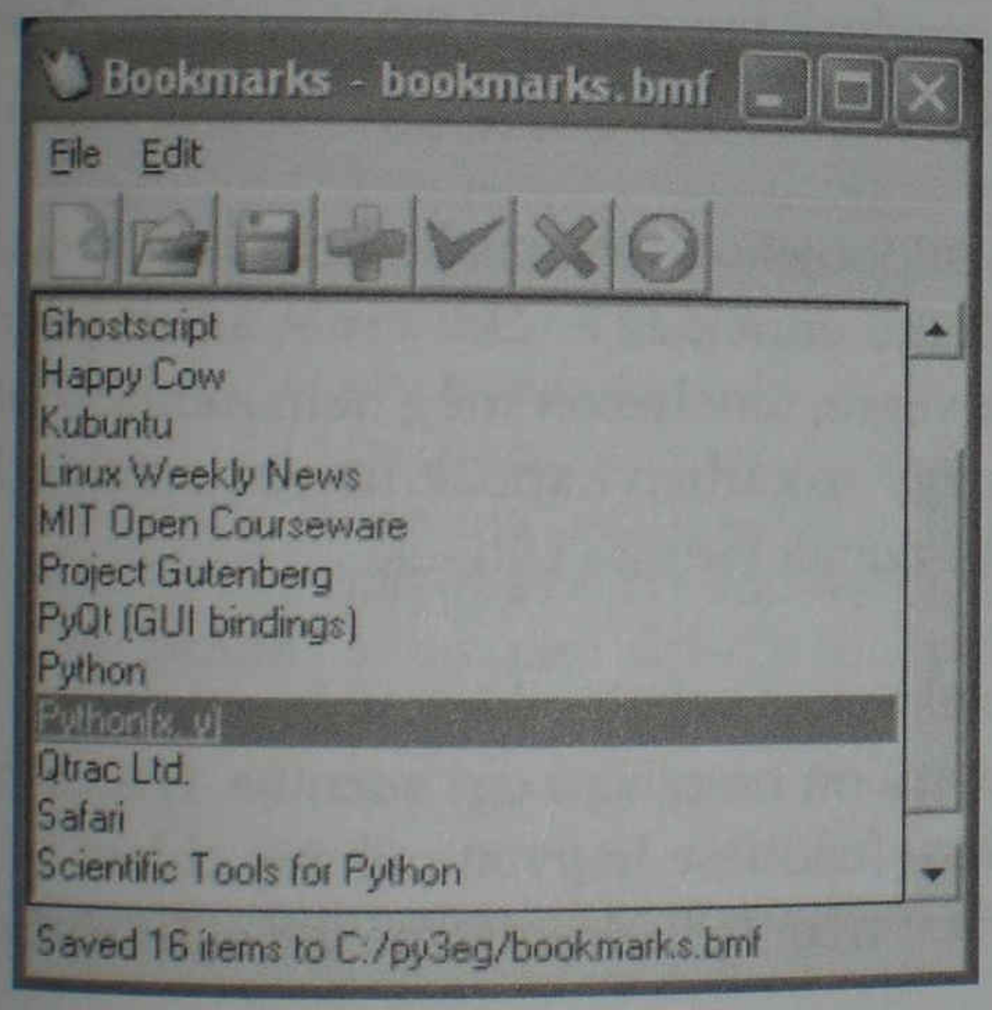
```
application = tkinter.Tk()
path = os.path.join(os.path.dirname(__file__), "images/")
if sys.platform.startswith("win"):
    icon = path + "interest.ico"
else:
    icon = "@" + path + "interest.xbm"
application.iconbitmap(icon)
application.title("Interest")
window = MainWindow(application)
application.protocol("WM_DELETE_WINDOW", window.quit)
application.mainloop()
```

Ha meghatároztuk a főablak (ami ebben az esetben az egyetlen ablak) osztályát, már rendelkezünk azzal a kóddal, ami elindítja a programot. Ez után először létrehozzuk az alkalmazás egészét jelképező objektumot. Windowson ahhoz, hogy egy programhoz ikont rendeljünk, egy `.ico` fájlra van szükségünk, amelynek a fájlnevét (teljes elérési úttal) átadjuk az `iconbitmap()` tagfüggvénynek. Unix rendszereken azonban egy bitképet (pontosabban egy monokróm képet) kell biztosítanunk. A Tk számos beépített bitképet kínál, amelyeket a fájlrendszerből származó képektől a nevük elé írt `@` jellel különböztethetünk meg. Ezt követően nevet (címet) adunk az alkalmazásnak (ez fog megjelenni az ablak címsorában), majd létrehozzuk a `MainWindow` osztályunk egy példányát, a szülőjeként az alkalmazásobjektumot megadva. Végül meghívjuk a `protocol()` tagfüggvényt, hogy megadjuk, hogy mi történjen, ha a felhasználó a bezáró gombra kattint – mi azt mondtuk, hogy ilyenkor a `MainWindow.quit()` tagfüggvényt kell meghívni. Végül elindítjuk az eseményhurkot – az ablak csak ekkor jelenik meg, és innentől fogva képes válaszolni a felhasználói műveletekre.

Főablakos programok

Bár a párbeszédablakos programok gyakran elegendőek az egyszerű feladatokhoz, ha egy program képességeinek köre nő, általában célszerű teljes, főablakos alkalmazást készíteni belőle, menükkel és eszköztárakkal. Az ilyen alkalmazásokat általában könnyebb bővíteni, mint a párbeszédablakos programokat, mivel anélkül adhatunk hozzájuk további menüket, menüpontokat vagy eszköztárgombokat, hogy ez befolyásolná a főablak elrendezését.

Ebben a részben a 13.4. ábrán látható `bookmarks-tk.pyw` programot vizsgáljuk meg. A program könyvjelzőket tárol (név, URL) formájú karakterláncok alakjában, és lehetőséget nyújt a felhasználónak a könyvjelzők felvételére, szerkesztésére és eltávolítására, valamint egy adott könyvjelző által hivatkozott weboldal megnyitására a webböngészőben.



13.4. ábra

A Bookmarks program

A programnak két ablaka van: a főablak a menüsorral, az eszköztárral, a könyvjelzők listájával és az állapotsorral, valamint egy párbeszédablak, amely a könyvjelzők felvételére és szerkesztésére szolgál.

Főablak létrehozása

A főablak annyiban hasonlít egy párbeszédablakhoz, hogy vezérlői vannak, amelyeket létre kell hozni és el kell rendezni. Ezen kívül hozzá kell adnunk a menüsort, a menüket, az eszköztárat és az állapotsort, valamint tagfüggvényeket, amelyek végrehajtják a felhasználó által kért műveleteket. A felhasználói felületet teljes egészében a főablak előkészítő függvényében alakítjuk ki, amelyet öt részletben tekintünk át, mert elég hosszú.

```
class MainWindow:

    def __init__(self, parent):
        self.parent = parent

        self.filename = None
        self.dirty = False
        self.data = {}

        menubar = tkinter.Menu(self.parent)
        self.parent["menu"] = menubar
```

Ehhez az ablakhoz – ellentétben az előző példával, ahol egy meglevő vezérlőtől örököltünk – egyszerűen egy szokványos Python-osztályt hozunk létre. Ha öröklést alkalmazunk, új megvalósítást adhatunk a szülőosztálytól örökölt tagfüggvényeknek, de ha erre nincs szükségünk, használhatunk egyszerűen összetételt, ahogy itt is tettük. Az ablak megjelenését vezérlő-példányváltozók létrehozásával alakítjuk ki, amelyek mindegyike egy `tkinter.Frame` objektumban kap helyet, ahogy hamarosan látni fogjuk.

Négyféle információt kell nyomon követnünk: a szülőobjektumot (az alkalmazást), az aktuális könyvjelzőfájl nevét, a „piszkos” (`dirty`) jelzőt (ha ennek az értéke `True`, az azt jelenti, hogy az adatokon olyan módosításokat hajtottunk végre, amelyeket még nem mentettünk lemezre), valamint magukat az adatokat, amelyek egy szótárban kapnak helyet, amelynek a kulcsai a könyvjelzők nevei, az értékei pedig a hozzájuk tartozó URL-ek.

A menüsor létrehozásához egy `tkinter.Menu` objektumot kell létrehoznunk, amelynek a szülője az ablak szülője, és tájékoztatnunk kell a szülőt, hogy van egy menüje. (Furcsának tűnhet, hogy a menüsor is egy menü, de a Tk hosszú fejlődése hagyott néhány ehhez hasonló furcsaságot.) Az így létrehozott menüsorokat nem kell elrendezni; ezt a Tk elvégzi helyettünk.

```
        fileMenu = tkinter.Menu(menubar)
        for label, command, shortcut_text, shortcut in (
                ("New...", self.fileNew, "Ctrl+N", "<Control-n>"),
                ("Open...", self.fileOpen, "Ctrl+O", "<Control-o>"),
                ("Save", self.fileSave, "Ctrl+S", "<Control-s>"),
                (None, None, None, None),
                ("Quit", self.fileQuit, "Ctrl+Q", "<Control-q>")):
            if label is None:
                fileMenu.add_separator()
            else:
                fileMenu.add_command(label=label, underline=0,
                        command=command, accelerator=shortcut_text)
                self.parent.bind(shortcut, command)
        menubar.add_cascade(label="File", menu=fileMenu, underline=0)
```

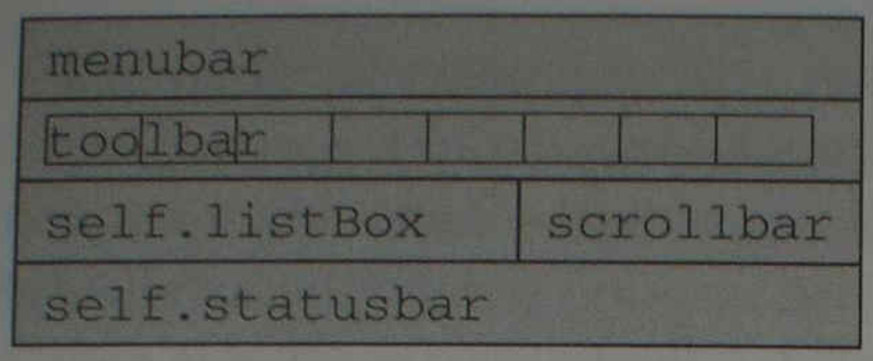
A menüsor minden menüjét ugyanúgy kell létrehozni. Először létrehozunk egy `tkinter.Menu` objektumot, amely a menüsor gyermeke, majd parancsokat (menüpontokat) és elválasztókat adunk a menühöz. (Megjegyzendő, hogy az „accelerator” – „gyorsító” –

a Tk nyelvezetében billentyűparancsot jelent, az `accelerator` paraméter azonban csak a billentyűparancs-hivatkozás szövegét állítja be, tényleges billentyűkötést nem hoz létre.) Az `underline` paraméter az aláhúzott karakter megadására szolgál – esetünkben minden menüpontnak az első betűjét húzzuk alá, és ez a betű lesz az adott parancs gyorsbillentyűje. A menüpontok (parancsok, command) hozzáadása mellett billentyűparancsokat is megadunk, mégpedig úgy, hogy egy billentyűsorozatot kötünk ahhoz a parancshoz, amelyik a kiválasztott menüponthoz tartozik. Végül a menüt a menüsorhoz adjuk az `add_cascade()` tagfüggvénnyel.

Az Edit (Szerkesztés) menüt kihagytuk, mert a szerkezetét tekintve megegyezik a File (Fájl) menü kódjával.

```
frame = tkinter.Frame(self.parent)
self.toolbar_images = []
toolbar = tkinter.Frame(frame)
for image, command in (
        ("images/filenew.gif", self.fileNew),
        ("images/fileopen.gif", self.fileOpen),
        ("images/filesave.gif", self.fileSave),
        ("images/editadd.gif", self.editAdd),
        ("images/editedit.gif", self.editEdit),
        ("images/editdelete.gif", self.editDelete),
        ("images/editshowwebpage.gif", self.editShowWebPage)):
    image = os.path.join(os.path.dirname(__file__), image)
    try:
        image = tkinter.PhotoImage(file=image)
        self.toolbar_images.append(image)
        button = tkinter.Button(toolbar, image=image,
                                command=command)
        button.grid(row=0, column=len(self.toolbar_images) -1)
    except tkinter.TclError as err:
        print(err)
toolbar.grid(row=0, column=0, columnspan=2, sticky=tkinter.NW)
```

Ez után első lépésként létrehozunk egy keretet, amely az ablak összes vezérlőjét tartalmazni fogja, majd egy másikat `toolbar` néven, amelyben olyan gombok vízszintes sora kap helyet, amelyeken szöveg helyett képek láthatók – ezek lesznek az eszköztár gombjai. Az eszköztárgombokat egymás után egy rácsban rendezzük el, amelynek egyetlen sora van, és annyi oszlopa, ahány gombbal rendelkezünk. Végül magának az eszköztárnak a keretét a főablak keretének első sorává tesszük, „északnyugatra" ragasztva, hogy mindig az ablak bal felső sarkához igazodjon. (A Tk automatikusan az ablakban található többi vezérlő felett helyezi el a menüsort.) Az elrendezést a 13.5. ábra szemlélteti, amelyen a Tk által elhelyezett menüsort fehér, míg az általunk elrendezett vezérlőket szürke háttérrel jeleztük.



13.5. ábra

A Bookmarks program főablakának elrendezése

Amikor egy képet adunk egy gombhoz, gyenge hivatkozás jön létre, ami azt jelenti, hogy ha a kép kikerül a hatókörből, akkor kijelölődik szemétgyűjtésre. Ezt el kell kerülnünk, mert azt szeretnénk, ha a gombokon megjelennének a képek, amikor az előkészítő befejezte a munkáját, ezért létrehozunk egy `self.toolbar_images` nevű példányváltozót, amelynek csak annyi a feladata, hogy tárolja a képekre mutató hivatkozásokat, hogy életben tartsa azokat a program teljes élete során.

Alapállapotában a Tk csupán néhány képfájlformátumot képes olvasni, ezért `.gif`* képeket kellett használnunk. Ha valamelyik kép nem található, `tkinter.Exception` kivétel keletkezik, ezért ügyelnünk kell rá, hogy elfogjuk ezt a kivételt, nehogy a program kilépjen, csak azért, mert nincs meg egy kép.

Figyeljük meg, hogy a menükben elérhető műveletek közül nem mindet tettük eszköztárgombként is hozzáférhetővé, ami elterjedt gyakorlat.

```
scrollbar = tkinter.Scrollbar(frame, orient=tkinter.VERTICAL)
self.listBox = tkinter.Listbox(frame, yscrollcommand=scrollbar.set)
self.listBox.grid(row=1, column=0, sticky=tkinter.NSEW)
self.listBox.focus_set()
scrollbar["command"] = self.listBox.yview
scrollbar.grid(row=1, column=1, sticky=tkinter.NS)

self.statusbar = tkinter.Label(frame, text="Ready...",
                               anchor=tkinter.W)
self.statusbar.after(5000, self.clearStatusBar)
self.statusbar.grid(row=2, column=0, columnspan=2, sticky=tkinter.EW)
frame.grid(row=0, column=0, sticky=tkinter.NSEW)
```

A főablak központi részét (az eszköztár és az állapotsor közötti területet) egy listamező és egy hozzá tartozó gördítősáv foglalja el. A listamezőt minden irányban ragadóssá tettük, a gördítősávot azonban csak északra és délre (vagyis függőlegesen). Mindkét vezérlőt az ablakkeret rácsához adtuk, egymás mellett.

---

* Ha telepítettük a Python Imaging Library Tk bővítményét, minden mai képformátum támogatottá válik. A részleteket lásd a `www.pythonware.com/products/pil/` címen.

Gondoskodnunk kell róla, hogy ha a felhasználó a LE és FEL nyílbillentyűkkel mozog a listában, vagy a gördítősávval görgeti a listát, a két vezérlő összhangban maradjon. Ezt úgy érjük el, hogy a listamező `yscrollcommand` tulajdonságát a gördítősáv `set()` tagfüggvényére állítjuk (így a listamezőben való mozgás a gördítősávot is elmozdítja, amennyiben szükséges), a gördítősáv `command` tulajdonságát pedig a listamező `yview()` tagfüggvényére (ennek eredményeképpen a gördítősáv mozgatása a lista megfelelő görgetését is maga után vonja).

Az állapotsor csupán egy szövegcímke. Az `after()` tagfüggvény egy egyszeri időzítő (vagyis egy olyan időzítő, amely egyszer jár le a megadott időtartam után), amelynek az első paramétere a lejárati időtartam ezredmásodpercben, a második paramétere pedig egy függvény vagy tagfüggvény, amelyet akkor kell meghívni, amikor az időtartam lejárt. Ez azt jelenti, hogy amikor a program elindul, az állapotsorban öt másodpercig a „Ready…” („Kész…”) szöveg lesz látható, majd az állapotsor tartalma törlődik. Az állapotsort az ablak utolsó soraként helyezzük el, és nyugat–kelet irányban (vízszintesen) tesszük ragadóssá.

Végül elrendezzük magát az ablakkeretet. Már végeztünk a létrehozással és a főablak vezérlőinek elrendezésével, de jelen állapotban a vezérlők rögzített alapértelmezett méretet tételeznek fel, ezért ha az ablakot átméretezzük, a vezérlők nem nyúlnak meg vagy zsugorodnak össze, hogy igazodjanak a méretváltozáshoz. A következő kódrészlet megoldja ezt a problémát, és teljessé teszi az előkészítő függvényt:

```
frame.columnconfigure(0, weight=999)
frame.columnconfigure(1, weight=1)
frame.rowconfigure(0, weight=1)
frame.rowconfigure(1, weight=999)
frame.rowconfigure(2, weight=1)

window = self.parent.winfo_toplevel()
window.columnconfigure(0, weight=1)
window.rowconfigure(0, weight=1)

self.parent.geometry("{0}x{1}+{2}+{3}".format(400, 500, 0, 50))
self.parent.title("Bookmarks - Unnamed")
```

A `columnconfigure()` és `rowconfigure()` tagfüggvények lehetővé teszik a rács súlyozását. Az ablakkerettel kezdjük, ahol az összes „súlyt” az első oszlopnak és (a listamező által elfoglalt) második sornak adjuk, így ha a keretet átméretezzük, minden új területet a listamező fog megkapni. Önmagában azonban ez nem elég – a keretet tartalmazó legfelső szintű ablakot átméretezhetővé is kell tennünk. Ezt úgy érjük el, hogy a `wininfo_toplevel()` tagfüggvénnyel kinyerünk egy hivatkozást az ablakra, majd az ablakot úgy tesszük átméretezhetővé, hogy a sor- és oszlopsúlyozását 1-re állítjuk.

Az előkészítő függvény végén egy *`szélességxmagasság+x+y`* alakú karakterlánc segítségével beállítunk egy kezdeti ablakméretet és -helyzetet. (Ha csak a méretet szeretnénk meghatározni, a *`szélességxmagasság`* alakot használhatjuk helyette.) Végül megadjuk az ablak címét, amivel az ablak felhasználói felülete teljessé válik.

Ha a felhasználó egy eszköztárgombra kattint, vagy kiválaszt egy menüpontot, egy tagfüggvény hívódik meg, amely végrehajtja a kért műveletet, ezeknek a tagfüggvényeknek egy része pedig segédtagfüggvényekre támaszkodik. A következőkben valamennyi tagfüggvényt áttekintjük, azzal kezdve, amelyiknek a meghívására öt másodperccel a program elindulása után kerül sor:

```
def clearStatusBar(self):
    self.statusbar["text"] = ""
```

Az állapotsor egy egyszerű `tkinter.Label` objektum. A tartalmát törölhettük volna egy lambdakifejezéssel is az `after()` tagfüggvényhívásban, de mivel az állapotsor kiürítését több helyről is tudnunk kell elvégezni, külön tagfüggvényt hoztunk létre erre a célra:

```
def fileNew(self, *ignore):
    if not self.okayToContinue():
        return
    self.listBox.delete(0, tkinter.END)
    self.dirty = False
    self.filename = None
    self.data = {}
    self.parent.title("Bookmarks - Unnamed")
```

Ha a felhasználó új könyvjelzőfájlt akar létrehozni, először lehetőséget kell adnunk neki, hogy mentse a még nem mentett módosításokat a meglevő fájlba (ha van olyan). Ezt a lehetőséget a `MainWindow.okayToContinue()` tagfüggvényben teremtjük meg, mivel több különböző helyen is használni fogjuk. A tagfüggvény `True` értéket ad vissza, ha továbbléphetünk, más esetben pedig `False`-t. Ha folytathatjuk a műveletet, akkor kiürítjük a listamezőt az elsőtől az utolsóig az összes bejegyzés törlésével. A `tkinter.END` egy állandó, amellyel az utolsó elemet jelölhetjük egy olyan környezetben, ahol a vezérlő több elemet is tartalmazhat. Ezt követően töröljük a „piszkos” jelzőt, a fájlnevet és az adatokat, mivel a fájl új és módosítatlan, az ablak címét pedig átállítjuk, hogy tükrözze, hogy egy új, még mentetlen fájllal dolgozunk.

Az `ignore` („figyelmen kívül hagy”) változó egy nulla vagy több helyparaméterből álló sorozatot tárol, amellyel nem törődünk. A menüpontok kiválasztásával vagy eszköztárgombok lenyomásával meghívott tagfüggvények esetében nincsenek figyelmen kívül hagyott paraméterek, de ha a felhasználó egy billentyűparancsot adott ki (például a Ctrl+N-t), akkor a hívó esemény is átadódik, de mivel minket nem érdekel, hogy a felhasználó hogyan hívta meg a műveletet, figyelmen kívül hagyjuk a kérelmező eseményt.

```
def okayToContinue(self):
    if not self.dirty:
        return True
    reply = tkinter.messagebox.askyesnocancel(
                    "Bookmarks - Unsaved Changes",
                    "Save unsaved changes?", parent=self.parent)
```

```
    if reply is None:
        return False
    if reply:
        return self.fileSave()
    return True
```

Ha a felhasználó olyan műveletet szeretne végrehajtani, amely törli a listamező tartalmát (például létre szeretne hozni vagy meg szeretne nyitni egy új fájlt), lehetőséget kell adnunk neki, hogy mentse a még nem mentett módosításokat. Amennyiben a fájl nem „piszkos", nincsenek mentendő módosítások, ezért azonnal `True`-t adunk vissza, egyéb esetben pedig megjelenítünk egy szabványos üzenetablakot, amelyen `Yes` (Igen), `No` (Nem) és `Cancel` (Mégse) gombok találhatók. Ha a felhasználó visszavonja a parancsot (Mégse), a válasz (`reply`) `None` lesz; ezt úgy vesszük, hogy a felhasználó nem kívánja folytatni a megkezdett műveletet, és nem akar menteni sem, ezért egyszerűen `False` értéket adunk vissza. Amennyiben a felhasználó igennel válaszol, a válasz `True` lesz, így lehetőséget adunk a mentésre, és `True`-t adunk vissza, ha a mentés megtörtént, és `False`-t, ha nem. Ha pedig a felhasználó a Nem gombot nyomja meg, a válasz `False` lesz, ami arra utasítja a programot, hogy ne mentsen, de ettől függetlenül `True`-t adunk vissza, mert a felhasználó folytatni szeretné a megkezdett műveletet, csak éppen el kívánja vetni a még nem mentett módosításokat.

A Tk szabványos párbeszédablakait az `import tkinter` utasítás nem tölti be, ezért ezen a beemelésen felül ki kell adnunk az `import tkinter.messagebox`, illetve a következő tagfüggvényhez az `import tkinter.filedialog` utasítást is. Windowson és Mac OS X-en a rendszer beépített párbeszédablakainak a használatára kerül sor, míg más rendszereken a Tk-ra jellemző párbeszédablakok jelennek meg. A szabványos párbeszédablakoknak mindig átadjuk a szülőjüket, mert ez biztosítja, hogy megjelenésükkor automatikusan a szülőablak közepére kerüljenek.

Minden szabványos párbeszédablak *modális ablak*, ami azt jelenti, hogy ha megjelentek, a felhasználó csak ezen az egy ablakon végezhet műveleteket a programban, tehát ha hozzá szeretne férni a program többi részéhez, akkor előbb be kell zárnia a párbeszédablakot (az `OK`, `Open`, `Cancel` és más hasonló gombok egyikére kattintva). A programozók számára a modális párbeszédablakok használata a legkönnyebb, mert a felhasználó nem változtathatja meg mögöttük a program állapotát, és blokkolják a programot (maguknál tartják a vezérlését) egészen addig, amíg be nem zárják őket. A blokkolás azt jelenti, hogy amikor létrehozunk vagy meghívunk egy modális párbeszédablakot, az utána következő utasítás csak akkor hajtódik végre, amikor a párbeszédablakot bezárják.

```
def fileSave(self, *ignore):
    if self.filename is None:
        filename = tkinter.filedialog.asksaveasfilename(
                            title="Bookmarks - Save File",
                            initialdir=".",
                            filetypes=[("Bookmarks files", "*.bmf")],
```

```
                                defaultextension=".bmf",
                                parent=self.parent)
            if not filename:
                return False
            self.filename = filename
            if not self.filename.endswith(".bmf"):
                self.filename += ".bmf"
        try:
            with open(self.filename, "wb") as fh:
                pickle.dump(self.data, fh, pickle.HIGHEST_PROTOCOL)
            self.dirty = False
            self.setStatusBar("Saved {0} items to {1}".format(
                              len(self.data), self.filename))
            self.parent.title("Bookmarks - {0}".format(
                              os.path.basename(self.filename)))
        except (EnvironmentError, pickle.PickleError) as err:
            tkinter.messagebox.showwarning("Bookmarks - Error",
                    "Failed to save {0}:\n{1}".format(
                    self.filename, err), parent=self.parent)
        return True
```

Ha nincs fájl megnyitva, meg kell kérnünk a felhasználót, hogy válasszon egy fájlnevet. Ha a felhasználó a Mégse gombra kattint, `False`-t adunk vissza, hogy jelezzük, hogy a teljes műveletet vissza kell vonni, egyébként pedig meggyőződünk róla, hogy a megadott fájlnév kiterjesztése megfelelő-e. Ha igen, a meglevő vagy új fájlnév használatával a fájlba mentjük a befőzött `self.data` szótárt. A könyvjelzők mentését követően töröljük a „piszkos” jelzőt, hiszen most már nincsenek nem mentett módosítások, kiírunk egy üzenetet az állapotsorban (amely csak egy ideig látható, ahogy hamarosan látni fogjuk), és frissítjük az ablak címsorát, hogy a fájlnév is megjelenjen benne (az elérési út nélkül). Ha a fájl mentése nem sikerül, egy figyelmeztető üzenetablakot jelenítünk meg (ezen automatikusan helyet kap egy `OK` gomb), hogy tájékoztassuk a felhasználót.

```
def setStatusBar(self, text, timeout=5000):
    self.statusbar["text"] = text
    if timeout:
        self.statusbar.after(timeout, self.clearStatusBar)
```

A fenti tagfüggvény állítja be az állapotsor szövegét, és ha megadtunk egy lejárati időt (az alapértelmezés öt másodperc), a tagfüggvény beállít egy egyszeri időzítőt, amely törli az állapotsor tartalmát a lejárati idő letelte után.

```
def fileOpen(self, *ignore):
    if not self.okayToContinue():
        return
    dir = (os.path.dirname(self.filename)
           if self.filename is not None else ".")
    filename = tkinter.filedialog.askopenfilename(
                title="Bookmarks - Open File",
```

```
                initialdir=dir,
                filetypes=[("Bookmarks files", "*.bmf")],
                defaultextension=".bmf", parent=self.parent)
        if filename:
            self.loadFile(filename)
```

Ez a tagfüggvény ugyanúgy kezdődik, mint a `MainWindow.fileNew()` – lehetőséget ad a felhasználónak, hogy mentse a még nem mentett módosításokat, vagy visszavonja a fájlmegnyitási műveletet. Ha a felhasználó úgy dönt, hogy folytatja, egy ésszerű kezdőkönyvtárba kell irányítanunk, ezért az aktuális fájl (ha van ilyen) könyvtárát használjuk, illetve egyéb esetben a jelenlegi munkakönyvtárat. A `filetypes` paraméter (leírás, helyettesítő karakter) formájú rendezett párok listája, amelyet a fájlmegnyitási párbeszédablaknak meg kell jelenítenie. Ha a felhasználó kiválasztott egy fájlnevet, az aktuális fájlnevet a választottra állítjuk, és meghívjuk a `loadFile()` tagfüggvényt, amely végrehajtja a fájl tényleges beolvasását.

A `loadFile()` tagfüggvényt általában el szokták különíteni, hogy könnyebb legyen anélkül is betölteni egy fájlt, hogy a felhasználótól kérnénk egy fájlnevet. Egyes programok például induláskor betöltik az utoljára megnyitott fájlt, mások pedig a legutóbb megnyitott fájlokat felsorolják egy menüben, így ha a felhasználó kiválaszt egyet közülük, a `loadFile()` tagfüggvényt közvetlenül meg lehet hívni az adott menüponthoz társított fájlnévvel.

```
    def loadFile(self, filename):
        self.filename = filename
        self.listBox.delete(0, tkinter.END)
        self.dirty = False
        try:
            with open(self.filename, "rb") as fh:
                self.data = pickle.load(fh)
            for name in sorted(self.data, key=str.lower):
                self.listBox.insert(tkinter.END, name)
            self.setStatusBar("Loaded {0} bookmarks from {1}".format(
                              self.listBox.size(), self.filename))
            self.parent.title("Bookmarks - {0}".format(
                              os.path.basename(self.filename)))
        except (EnvironmentError, pickle.PickleError) as err:
            tkinter.messagebox.showwarning("Bookmarks - Error",
                    "Failed to load {0}:\n{1}".format(
                    self.filename, err), parent=self.parent)
```

Ha a fenti tagfüggvény meghívódik, tudhatjuk, hogy minden módosítást mentettek vagy elvetettek, tehát nyugodtan törölhetjük a listamező tartalmát. Az aktuális fájlnevet az átadott fájlnévre állítjuk, kiürítjük a listát, töröljük a „piszkos" jelzőt, majd megkíséreljük megnyitni a fájlt és a befőttet kibontani a `self.data` szótárba. Ha megvannak az adatok, végigmegyünk a könyvjelzők nevein, és egymás után a listához fűzzük azokat. Végül tájékoztató

üzenetet jelenítünk meg az állapotsorban, és frissítjük az ablak címsorát. Ha a fájlt nem sikerült beolvasni, vagy a befőttet nem tudtuk kibontani, egy üzenetablakban figyelmeztetést jelenítünk meg, hogy értesítsük a felhasználót.

```
def fileQuit(self, event=None):
    if self.okayToContinue():
        self.parent.destroy()
```

Ez a File menü utolsó menüpontjának tagfüggvénye. Lehetőséget adunk a felhasználónak, hogy mentse a még nem mentett módosításokat – ha a `Cancel` gombra kattint, nem csinálunk semmit, és a program folytatja a futását, egyébként azonban utasítjuk a szülőt, hogy semmisítse meg önmagát, ami a program tiszta befejezéséhez vezet. Ha a felhasználó beállításait is menteni szeretnénk, azt is itt tennénk meg, közvetlenül a `destroy()` hívás előtt.

```
def editAdd(self, *ignore):
    form = AddEditForm(self.parent)
    if form.accepted and form.name:
        self.data[form.name] = form.url
        self.listBox.delete(0, tkinter.END)
        for name in sorted(self.data, key=str.lower):
            self.listBox.insert(tkinter.END, name)
        self.dirty = True
```

Ezt a tagfüggvényt akkor hívjuk meg, amikor a felhasználó új könyvjelzőt szeretne felvenni (az Edit, Add menüpontra vagy az eszköztár ➕ gombjára kattintva, esetleg a CTRL+A billentyűparancsot kiadva). Az `AddEditForm` egy egyéni párbeszédablak, amelyet a következő részben tárgyalunk – a használatához csak annyit kell tudnunk róla, hogy van egy `accepted` nevű jelzője, amelyet `True`-ra állítunk, ha a felhasználó az `OK` gombra kattintott, és `False`-ra, ha a `Cancel` gombot választotta, valamint két jellemzője, a `name` és az `url`, amelyek a felhasználó által felvett vagy szerkesztett könyvjelző nevét és URL-jét tárolják.

Létrehozunk egy új `AddEditForm`-példányt, amely azonnal megjelenik, mégpedig modális párbeszédablakként, vagyis blokkolja a programot – tehát az `if form.accepted...` utasítás a párbeszédablak bezárásáig nem hajtódik végre. Ha a felhasználó az `OK` gombra kattint az `AddEditForm` párbeszédablakban, és nevet ad a könyvjelzőnek, az új könyvjelző nevét és URL-jét hozzáaadjuk a `self.data` szótárhoz. Ez után töröljük a listamező tartalmát, és újra beszúrjuk az adatokat a megfelelő sorrendben. Persze hatékonyabb lenne egyszerűen az új könyvjelzőt beszúrni a megfelelő helyre, de egy mai számítógépen a különbség még könyvjelzők százainak esetén is alig lenne észrevehető. Végül beállítjuk a `dirty` jelzőt, mivel most van egy nem mentett módosításunk.

```
def editEdit(self, *ignore):
    indexes = self.listBox.curselection()
    if not indexes or len(indexes) > 1:
        return
    index = indexes[0]
    name = self.listBox.get(index)
```

```
form = AddEditForm(self.parent, name, self.data[name])
if form.accepted and form.name:
    self.data[form.name] = form.url
    if form.name != name:
        del self.data[name]
        self.listBox.delete(0, tkinter.END)
        for name in sorted(self.data, key=str.lower):
            self.listBox.insert(tkinter.END, name)
    self.dirty = True
```

A könyvjelzők szerkesztése valamivel bonyolultabb, mint a hozzáadásuk, mert először meg kell keresnünk azt a könyvjelzőt, amelyet a felhasználó szerkeszteni szeretne. A `curselection()` tagfüggvény egy (valószínűleg üres) listát ad vissza a kijelölt elemek sorszámaival. Ha a felhasználó egyetlen elemet jelölt ki, akkor kiolvassuk a szövegét, mivel ez a neve annak a könyvjelzőnek, amelyet a felhasználó módosítani szeretne (és egyben a `self.data` szótár kulcsa). Ezt követően létrehozunk egy új `AddEditForm`-ot, a szekeszteni kívánt könyvjelző nevét és URL-jét átadva.

Miután az ablakot bezárták, frissítjük a `self.data` szótárt – amennyiben a felhasználó az `OK` gombra kattintott, és nem „üres" nevet adott a könyvjelzőnek. Ha az új név megegyezik a régivel, akkor egyszerűen beállítjuk a „piszkos" jelzőt, és készen is vagyunk (ez esetben ugyanis a felhasználó feltehetőleg az URL-t módosította), ha viszont a könyvjelző neve megváltozott, akkor töröljük a szótárból azt az elemet, amelynek a kulcsa a régi név, kiürítjük a listamezőt, majd újra feltöltjük a listát a könyjelzőkkel, ugyanúgy, ahogy egy könyvjelző hozzáadásakor tettük.

```
def editDelete(self, *ignore):
    indexes = self.listBox.curselection()
    if not indexes or len(indexes) > 1:
        return
    index = indexes[0]
    name = self.listBox.get(index)
    if tkinter.messagebox.askyesno("Bookmarks - Delete",
                                   "Delete '{0}'?".format(name)):
        self.listBox.delete(index)
        self.listBox.focus_set()
        del self.data[name]
        self.dirty = True
```

Ha egy könyvjelzőt törölni szeretnénk, először ki kell derítenünk, hogy a felhasználó melyik könyvjelzőt választotta, ezért a fenti tagfüggvény ugyanazokkal a sorokkal kezdődik, mint a `MainWindow.editEdit()`. Ha a felhasználó egyetlen könyvjelzőt jelölt ki, megjelenítünk egy üzenetablakot, amelyben a felhasználót a törlés megerősítésére kérjük. Ha igennel válaszol, az üzenetablak függvénye `True`-t ad vissza, ezért töröljük a könyvjelzőt a listából, valamint a `self.data` szótárból, illetve beállítjuk a `dirty` jelzőt, és a billentyűfókuszt visszaadjuk a listamezőnek.

```
    def editShowWebPage(self, *ignore):
        indexes = self.listBox.curselection()
        if not indexes or len(indexes) > 1:
            return
        index = indexes[0]
        url = self.data[self.listBox.get(index)]
        webbrowser.open_new_tab(url)
```

Ha a felhasználó kiváltja ennek a tagfüggvénynek a meghívását, akkor megkeressük az általa kijelölt könyvjelzőt, és kinyerjük a megfelelő URL-t a `self.data` szótárból. Ez után a `webbrowser` modul `webbrowser.open_new_tab()` függvényével megnyitjuk az adott URL-hez tartozó oldalt a felhasználó böngészőjében. Ha a böngésző még nem fut, elindítjuk azt.

```
application = tkinter.Tk()
path = os.path.join(os.path.dirname(__file__), "images/")
if sys.platform.startswith("win"):
    icon = path + "bookmark.ico"
    application.iconbitmap(icon, default=icon)
else:
    application.iconbitmap("@" + path + "bookmark.xbm")
window = MainWindow(application)
application.protocol("WM_DELETE_WINDOW", window.fileQuit)
application.mainloop()
```

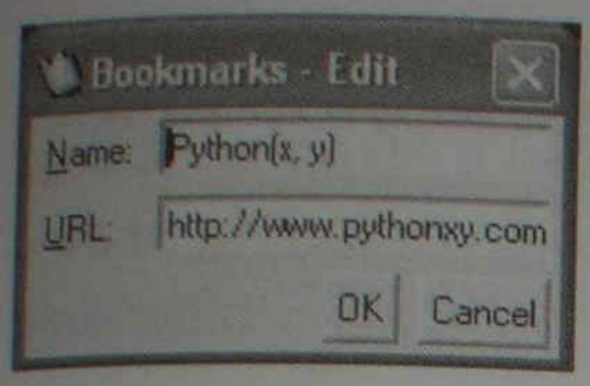
A program utolsó sorai megegyeznek a korábban látott `interest-tk.pyw` programban használtakkal, három eltéréssel. Az első az, hogy ha a felhasználó a programablak bezáró gombjára kattint, a Bookmarks programban más tagfüggvényt hívunk meg, mint az Interest programban. A második, hogy Windowson az `iconbitmap()` tagfüggvénynek van még egy paramétere, amelynek a segítségével alapértelmezett ikont határozhatunk meg a program összes ablaka számára – erre Unixon nincs szükség, mert ott ez önműködően történik. Az utolsó különbség pedig az, hogy az alkalmazás címsorát nem itt, hanem a `MainWindow` osztály tagfüggvényeiben állítjuk be. Az Interest programban a cím soha nem változik meg, ezért csak egyszer kell beállítanunk, a Bookmarks programban azonban a címet mindig annak megfelelően kell módosítanunk, hogy melyik könyvjelzőfájlon dolgozunk.

Most, hogy láttuk a főablak osztályának megvalósítását, valamint a programot előkészítő és az eseményhurkot elindító kódot, a figyelmünket az `AddEditForm` párbeszédablakra fordíthatjuk.

Egyéni párbeszédablak létrehozása

Az `AddEditForm` párbeszédablak nyújt lehetőséget a felhasználóknak a könyvjelzők neveinek és URL-jeinek a szerkesztésére, illetve hozzaádására. Az ablakot a 13.6. ábrán láthatjuk, ahol éppen egy meglevő könyvjelzőt szerkesztünk (ezért szerepel „Edit", vagyis

„Szerkesztés" a címsorban). Ugyanezt a párbeszédablakot könyvjelzők felvételére is használhatjuk. Először tekintsük át a párbeszédablak előkészítő kódját, mégpedig négy részre bontva.



13.6. ábra

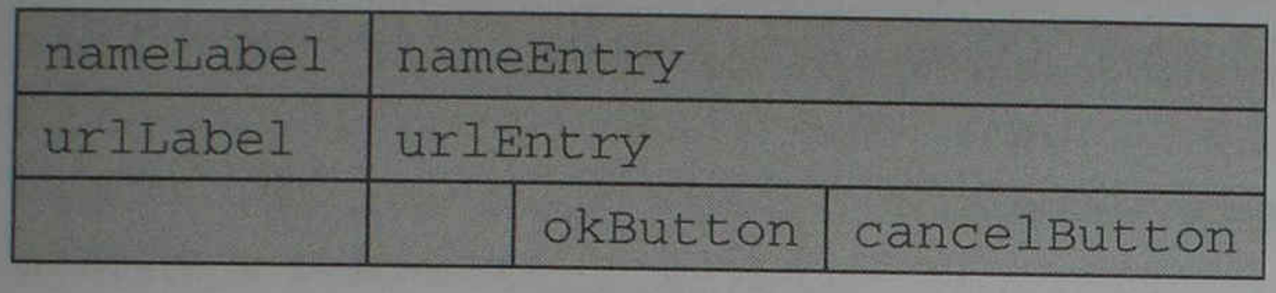
A Bookmarks program Add/Edit párbeszédablaka

```
class AddEditForm(tkinter.Toplevel):
    def __init__(self, parent, name=None, url=None):
        super(AddEditForm, self).__init__(parent)
        self.parent = parent
        self.accepted = False
        self.transient(self.parent)
        self.title("Bookmarks - " + (
                   "Edit" if name is not None else "Add"))
        self.nameVar = tkinter.StringVar()
        if name is not None:
            self.nameVar.set(name)
        self.urlVar = tkinter.StringVar()
        self.urlVar.set(url if url is not None else "http://")
```

Úgy döntöttünk, hogy a `tkinter.TopLevel`-től öröklünk, amely egy olyan csupasz vezérlő, amely a legfelső szintű ablakként használt vezérlők alaposztályaként szolgál. (Ahogy korábban említettük, a `super().__init__(parent)` kódot szerettük volna írni, de mivel ez nem működött, a `super()`-t kellett meghívnunk, kifejezetten átadva neki paraméterként a szülőosztályt és a `self` objektumot.) Elraktározunk egy a szülőre mutató hivatkozást, létrehozzuk a `self.accepted` jellemzőt, és `False`-ra állítjuk azt. A `transient()` tagfüggvényt azért hívjuk meg, hogy tájékoztassuk a szülőablakot, hogy ennek az ablaknak mindig a szülő felett kell megjelennie. A címsor beállításával jelezzük, hogy éppen hozzáadunk vagy szerkesztünk – a beállítást attól függően végezzük el, hogy kaptunk-e nevet és URL-t. A könyvjelző nevét és URL-jét két `tkinter.StringVar` követi nyomon, amelyekbe az átadott értékek kerülnek, ha a párbeszédablakot szerkesztésre használják.

```
        frame = tkinter.Frame(self)
        nameLabel = tkinter.Label(frame, text="Name:", underline=0)
        nameEntry = tkinter.Entry(frame, textvariable=self.nameVar)
        nameEntry.focus_set()
        urlLabel = tkinter.Label(frame, text="URL:", underline=0)
        urlEntry = tkinter.Entry(frame, textvariable=self.urlVar)
        okButton = tkinter.Button(frame, text="OK", command=self.ok)
        cancelButton = tkinter.Button(frame, text="Cancel", command=self.close)
        nameLabel.grid(row=0, column=0, sticky=tkinter.W, pady=3, padx=3)
```

```
nameEntry.grid(row=0, column=1, columnspan=3,
               sticky=tkinter.EW, pady=3, padx=3)
urlLabel.grid(row=1, column=0, sticky=tkinter.W, pady=3, padx=3)
urlEntry.grid(row=1, column=1, columnspan=3,
              sticky=tkinter.EW, pady=3, padx=3)
okButton.grid(row=2, column=2, sticky=tkinter.EW, pady=3, padx=3)
cancelButton.grid(row=2, column=3, sticky=tkinter.EW, pady=3, padx=3)
```



13.7. ábra

A Bookmarks program Add/Edit párbeszédablakának elrendezése

Ahogy a 13.7. ábrán láthatjuk, a vezérlőket egy rácsban hozzuk létre és rendezzük el. A név és az URL bevitelére szolgáló szövegmezők vezérlőit a megfelelő `tkinter.StringVar` változókhoz társítjuk, a két gomb pedig a később bemutatandó `self.ok()` és `self.close()` tagfüggvényeket fogja meghívni.

```
frame.grid(row=0, column=0, sticky=tkinter.NSEW)
frame.columnconfigure(1, weight=1)
window = self.winfo_toplevel()
window.columnconfigure(0, weight=1)
```

A párbeszédablakot csak vízszintesen van értelme átméretezni, ezért az ablakkeret második oszlopát vízszintesen méretezhetővé tesszük az oszlop súlyozását 1-re állítva – ez azt jelenti, hogy ha a keretet vízszintesen megnyújtják, az 1-es oszlopban levő vezérlők (a név és az URL bevitelére szolgáló szövegmezők) is megnőnek, hogy kitöltsék a többlethelyet. Ugyanígy az ablak oszlopát is vízszintesen átméretezhetővé tesszük a súlya 1-re állításával. Ha a felhasználó a párbeszédablak magasságát módosítja, a vezérlők megtartják az egymáshoz viszonyított helyzetüket, és az ablakon belül középre igazodnak, de ha a párbeszédablak szélessége változik meg, a név és az URL bevitelére szolgáló szövegmezők a vízszintesen rendelkezésre álló hely függvényében összezsugorodnak vagy megnyúlnak.

```
self.bind("<Alt-n>", lambda *ignore: nameEntry.focus_set())
self.bind("<Alt-u>", lambda *ignore: urlEntry.focus_set())
self.bind("<Return>", self.ok)
self.bind("<Escape>", self.close)
self.protocol("WM_DELETE_WINDOW", self.close)
self.grab_set()
self.wait_window(self)
```

Itt két szövegcímkét hozunk létre Name: és URL: szöveggel, amelyekre kattintva a billentyűfókusz a megfelelő szövegbeviteli vezérlőre kerül (az aláhúzások jelzik, hogy a két vezérlőhöz az ALT+N, illetve az ALT+U gyorsbillentyű tartozik). A szükséges billentyűkötéseket is

megadjuk, hogy mindez működjön. Lambdafüggvényeket használunk a `focus_set()` tagfüggvények közvetlen átadása helyett, hogy az eseményparamétert figyelmen kívül hagyhassuk. Ezen kívül az `OK` és `Cancel` gombok szabványos billentyűkötéseit (ENTER és Esc) is megadjuk.

A `protocol()` tagfüggvénnyel határozzuk meg, hogy melyik tagfüggvényt kell meghívni, ha a felhasználó a bezáró gombra kattintva bezárja a párbeszédablakot. A `grab_set()` és a `wait_window()` hívásokra azért van szükség, hogy az ablakot modális párbeszédablakká alakítsuk.

```
def ok(self, event=None):
    self.name = self.nameVar.get()
    self.url = self.urlVar.get()
    self.accepted = True
    self.close()
```

Ez a tagfüggvény hívódik meg, amikor a felhasználó az `OK` gombra kattint (vagy lenyomja az ENTER billentyűt). A függvény a `tkinter.StringVar` változókból a megfelelő példányváltozókba másolja a szövegeket (mely példányváltozókat csak most hozzuk létre), a `self.accepted` változót `True` értékre állítja, és meghívja a `self.close()` tagfüggvényt a párbeszédablak bezárásához.

```
def close(self, event=None):
    self.parent.focus_set()
    self.destroy()
```

Ezt a tagfüggvényt a `self.ok()` tagfüggvényből hívjuk meg, illetve akkor, ha a felhasználó az ablak bezáró gombjára vagy a `Cancel` gombra kattint (vagy lenyomja az Esc billentyűt). A függvény visszaadja a billentyűfókuszt a szülőnek, és a párbeszédablakot önmaga megsemmisítésére utasítja. Ebben az összefüggésben a megsemmisítés csak annyit jelent, hogy az ablak és annak vezérlői törlődnek – az `AddEditForm`-példány továbbra is létezni fog, mert a hívó még hivatkozik rá.

Miután a párbeszédablak bezáródott, a hívó megvizsgálja az `accepted` változót, és ha annak értéke `True`, akkor kiolvassa a felhasználó által megadott vagy módosított nevet és URL-t. Ezt követően, ha a `MainWindow.editAdd()` vagy a `MainWindow.editEdit()` tagfüggvény befejezte a működését, az `AddEditForm` objektum kikerül a hatókörből, és kijelölődik szemétgyűjtésre.

Összefoglalás

Ez a fejezet ízelítőt adott a grafikus felület programozásából a Tk GUI-könytár segítségével. A Tk nagy előnye, hogy része a Python alapcsomagjának, ugyanakkor számos hátránya is van – nem utolsósorban az, hogy régi könyvtár, amely némileg másképp működik, mint mai versenytársai többsége.

Ha még kezdők vagyunk a grafikus programozás terén, ne feledjük, hogy a Tk főbb rendszerfüggetlen versenytársai – a PyGtk, a PyQt és a wxPython – sokkal könnyebben megtanulhatók és használhatók, mint a Tk, és mindegyikkel jobb eredmény érhető el, ráadásul kevesebb kóddal. Ezek a Tk-riválisok továbbá kivétel nélkül bővebb és jobb Python-központú leírással, lényegesen több vezérlővel és szebb megjelenéssel rendelkeznek, és azt is lehetővé teszik, hogy úgy hozzunk létre a semmiből új vezérlőket, hogy közben a megjelenésüket és a viselkedésüket teljesen kézben tartjuk.

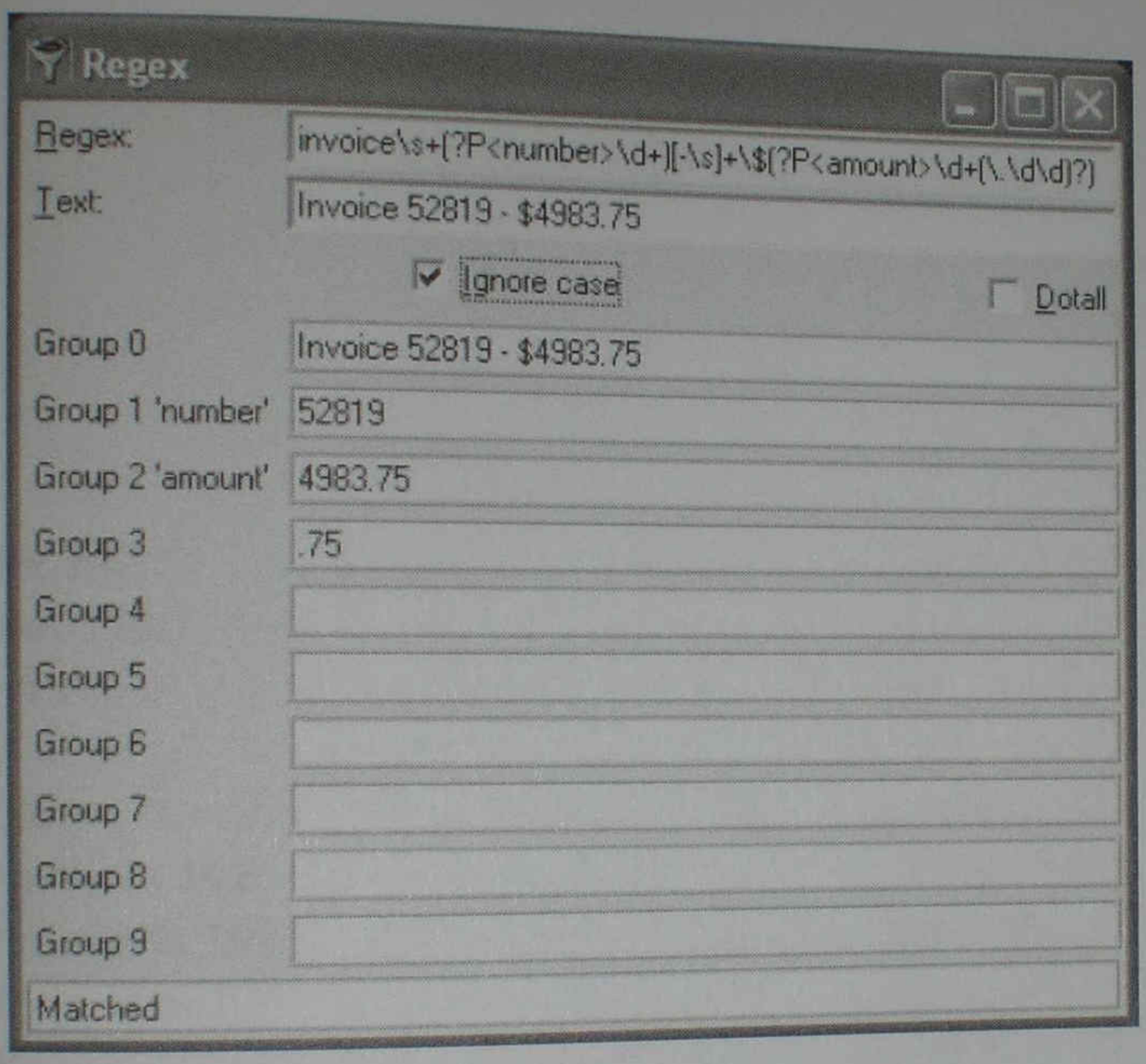
Bár a Tk-nak aprócska programok írásakor, illetve olyan helyzetekben, amikor csak a Python alapkönyvtára áll a rendelkezésünkre, hasznát vehetjük, minden más esetben bármely másik rendszerfüggetlen programkönyvtár sokkal jobb választás.

Gyakorlatok

Az első gyakorlatban az ebben a fejezetben bemutatott Bookmarks programot kell lemásolni és módosítani, míg a másodikban a semmiből kell elkészíteni egy grafikus felületű programot.

1. Másoljuk le a `bookmarks-tk.pyw` programot, és módosítsuk úgy, hogy képes legyen a (11. fejezetben gyakorlatként elkészített) `bookmarks.py` konzolprogram által használt DBM-fájlok betöltésére és mentésére. Ehhez adjunk két menüpontot – `Import` és `Export` – a File menühöz, és ne felejtsünk el billentyűparancsokat kapcsolni hozzájuk (ne feledjük, hogy a CTRL+E billentyűparancsot már használjuk az Edit, Edit menüponthoz). Ugyanígy hozzuk létre a két megfelelő eszköztárgombot is. Ehhez körülbelül öt sornyi kóddal kell kiegészítenünk a főablak előkészítő függvényét. Az említett lehetőségek megvalósításához két tagfüggvényre lesz szükség: a `fileImport()` és a `fileExport()` nevűekre, amelyek kevesebb mint 60 kódsorral megvalósíthatók, a hibakezelést is beleértve. Betöltéskor eldönthetjük, hogy a betöltött könyvjelzőket össze szeretnénk-e fésülni a meglevőkkel, vagy inkább lecseréljük a meglevő könyvjelzőket a betöltöttekre. A kód nem bonyolult, de a megírása némi körültekintést igényel. Egy megoldást (amely a betöltött könyvjelzőket egyesíti a meglevőkkel) a `bookmarks-tk_ans.py` fájlban találhatunk.
2. A 12. fejezetben láttuk, hogyan hozhatunk létre és használhatunk szabályos kifejezéseket szövegek illesztésére. Készítsünk olyan párbeszédablakos grafikus felületű programot, amelyet szabályos kifejezések beírására és tesztelésére használhatunk (lásd a 13.8. ábrát).
   A feladat megoldásához el kell olvasnunk a `re` modul leírását, mert a programnak akkor is helyesen kell működnie, ha érvénytelen szabályos kifejezéssel találkozik, vagy ha végig kell járnia az illesztési csoportokat, mivel a legtöbb esetben a szabályos kifejezés nem fog annyi illesztési csoportot tartalmazni, mint ahány szövegcímke rendelkezésre áll. Gondoskodjunk róla, hogy a program teljes támogatást nyújtson azoknak, akik a billentyűzetet használják: a szövegbeviteli vezérlők között lehessen

mozogni az Alt+R és Alt+T billentyűkkel, a jelölőnégyzeteket kezelni az Alt+I és Alt+D billentyűparancsok segítségével, ki lehessen lépni a programból a Ctrl+Q és az Esc billentyűkkel, és végezzünk újraszámítást , ha a felhasználó lenyom és felenged egy billentyűt bármelyik szövegbeviteli vezérlőben, illetve ha bejelöl egy jelölőnégyzetet, vagy törli belőle a jelet.



13.8. ábra

A Regex program

A program megírása nem túl nehéz, bár a találatokat és a csoportszámokat (illetve neveket, ha meg vannak adva) megjelenítő kód egy icipicit trükkös. Egy megoldást a `regex-tk.pyw` fájlban találhatunk, amely körülbelül 140 sorból áll.