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 gyűjtemény-típusok, 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 melléjük 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
```

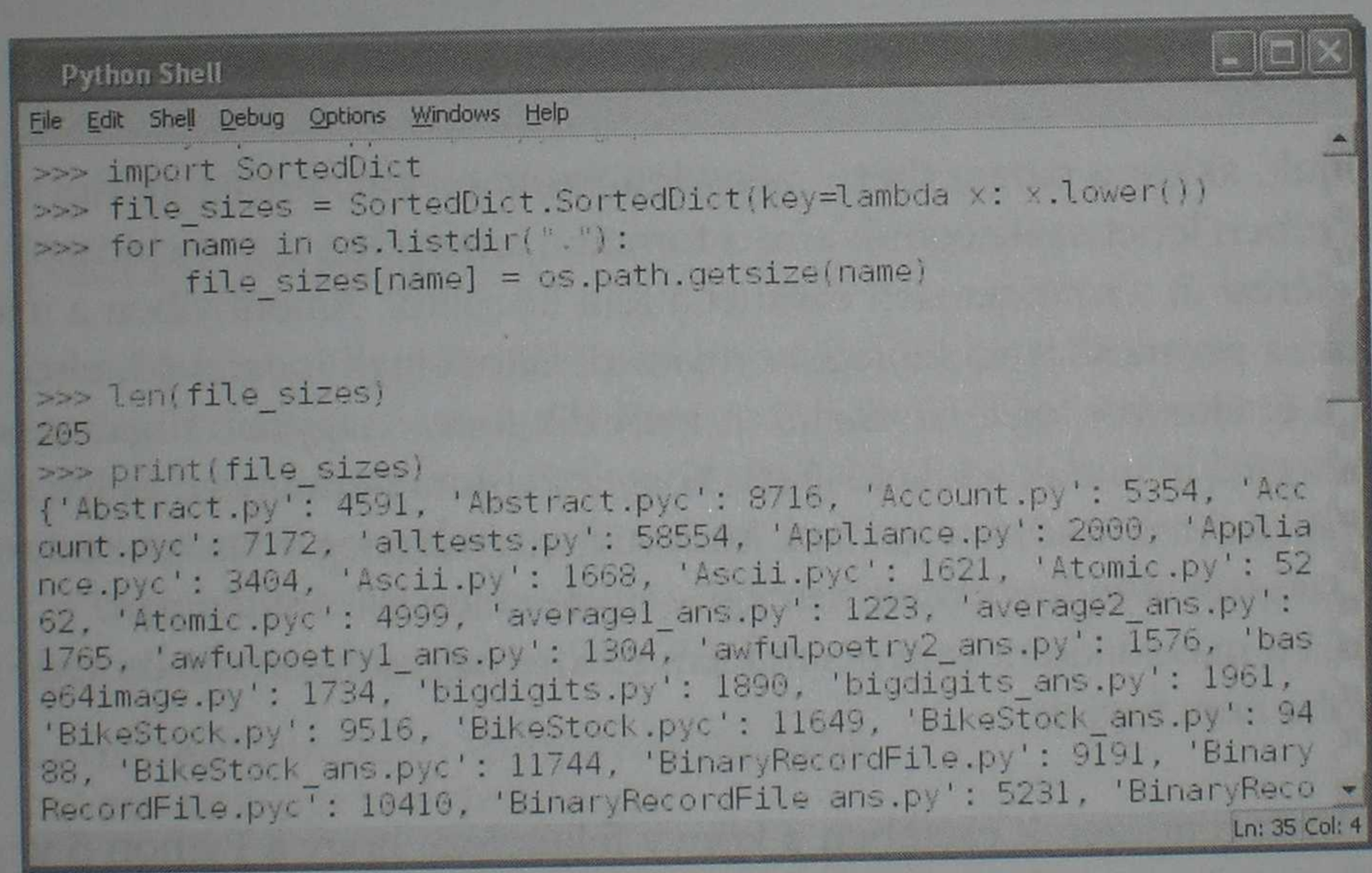
Ha az első formában írjuk, akkor a parancshéj a megadott értelmezőt használja. Olyan Python-programok esetében lehet szükségünk erre a formára, amelyeket egy webkiszolgáló futtat – a pontos elérési út természetesen eltérhet a fent közölttől. Amennyiben a második formát használjuk, a parancshéj a jelenlegi 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
2106245833371143733958360553673408646377901908010982225086219550720
"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.

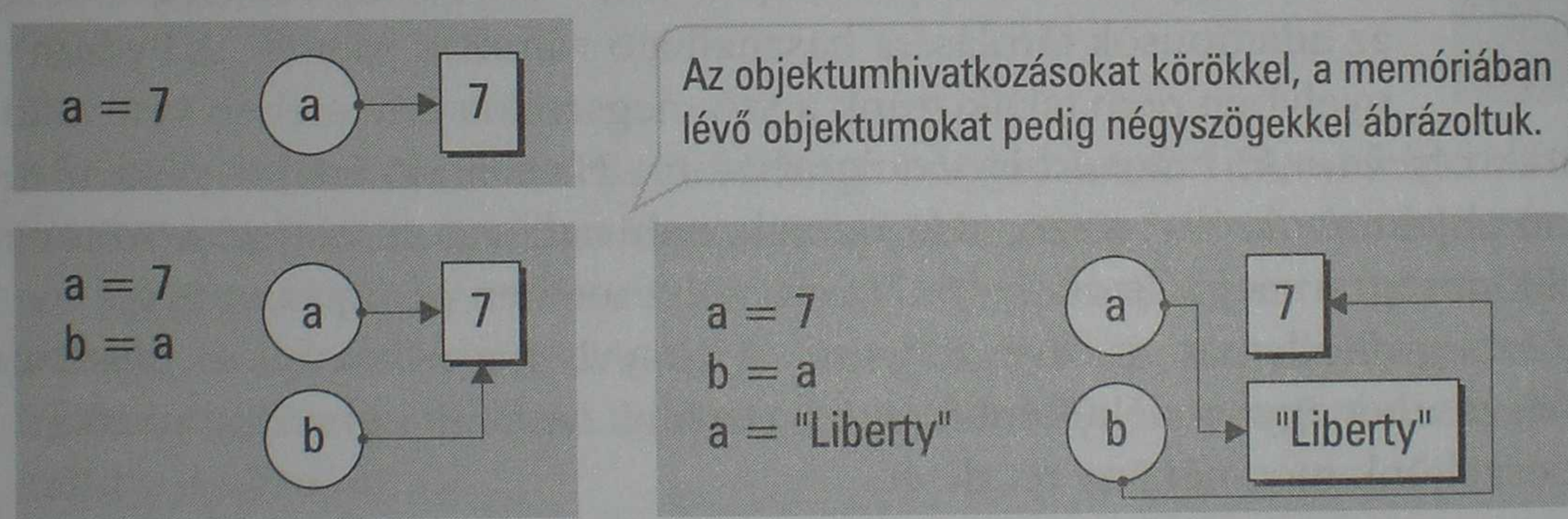
Az egyenlőségjel nem egyenértékű más programozási nyelvek változóknak értéket adó műveleti jelével. Az egyenlőségjel egy objektumhivatkozást kapcsol a memóriában lévő objektumhoz. Ha az objektumhivatkozás már eddig is létezett, akkor a hivatkozás lecserélésével mostantól egy másik, az egyenlőségjel jobb 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 hibakereséskor, 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:

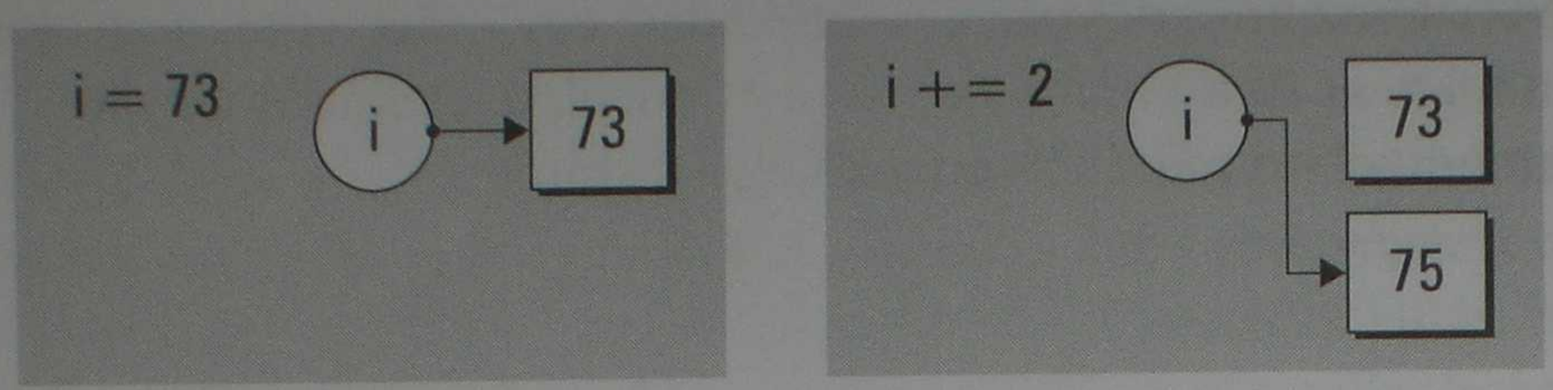
```
>>> 12 / 3
4.0
>>> 3 / 2
1.5
```

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.

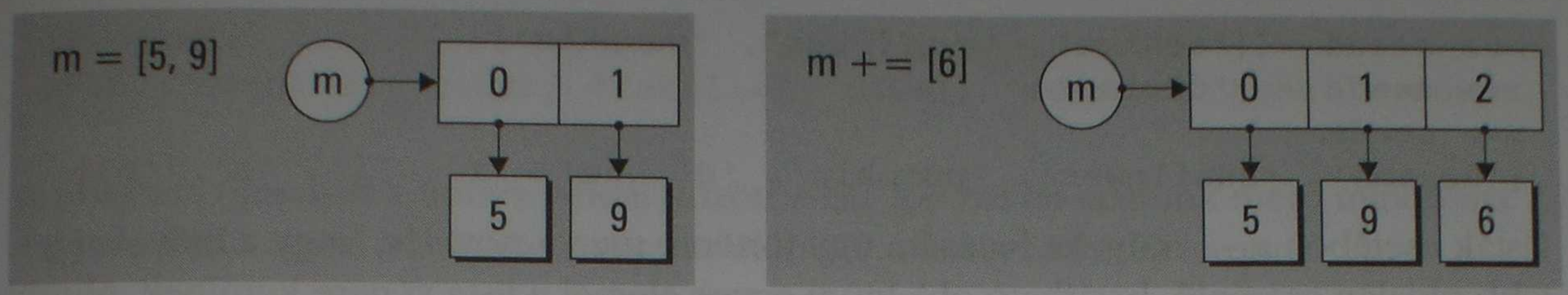
A `+` és a `+=` műveleti jeleket a Python a karakterláncok és listák esetében túlterheli (vagyis más adattípusnál más értelemben is használja). Az első jel az összefűzés (konkatenáció) műveletét hajtja végre, a második pedig karakterláncok esetében hozzáfűzést, listák esetében bővítést (másik lista hozzáfűzését) 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 iszonytató 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 tesztadatokként, 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
`divmod(x, y)`	Az `x/y` osztás eredményét és osztási maradékát két egész szám rendezett párjaként adja vissza.
`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).

Rendezett sorozatok 12 ☞

Rendezett sorozatok ☞ 102

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, 5.4000000000000004, -2.5, 0.00088999999999999995)
```

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 $\sqrt{x^2+y^2}$.
`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 π 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)
78.539816339744831
>>> math.hypot(5, 12)
13.0
>>> math.modf(13.732)
(0.73199999999999932, 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 számművelete 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"))
21.904761904761904761904761 90
>>> decimal.Decimal(23) / decimal.Decimal("1.05")
Decimal('21.904761904761904761904761 90')
```

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-

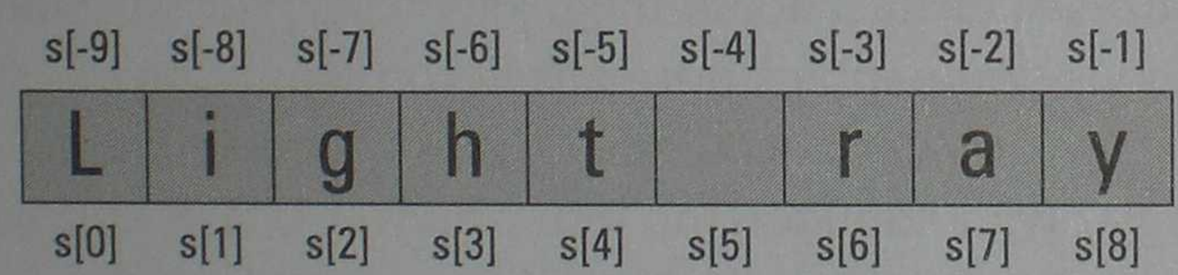
sággal a szükségességéről. Természetesen, amint a 3. fejezetben majd láthatjuk, a Python rendezési eljárásait igényeink szerint testreszabhatjuk. Ha bővebben szeretnénk olvasni a Unicode-karakterláncok rendezéséről, a Unicode Collation Algorithm dokumentum (`unicode.org/reports/tr10`) kielégítheti a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

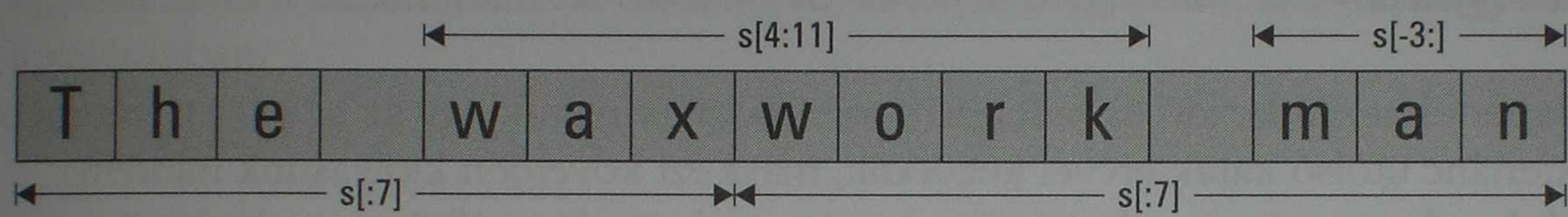
A szeletelő műveletet háromféle alakban használhatjuk:

```
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'
```

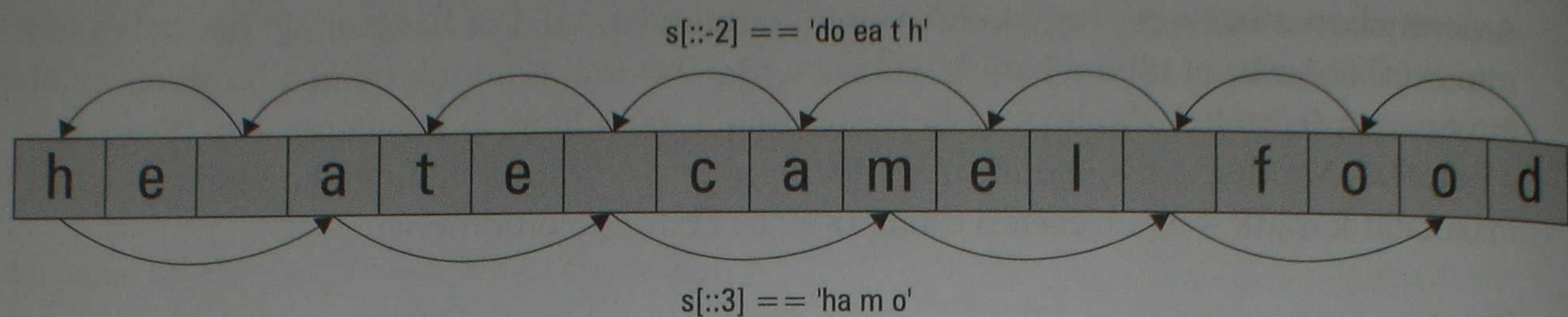
Mivel a „wo" az eredeti karakterláncban is megjelenik, ugyanezt az eredményt elérhettük volna az `s[:12] + s[7:9] + s[12:]` hozzárendelésével is.

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.



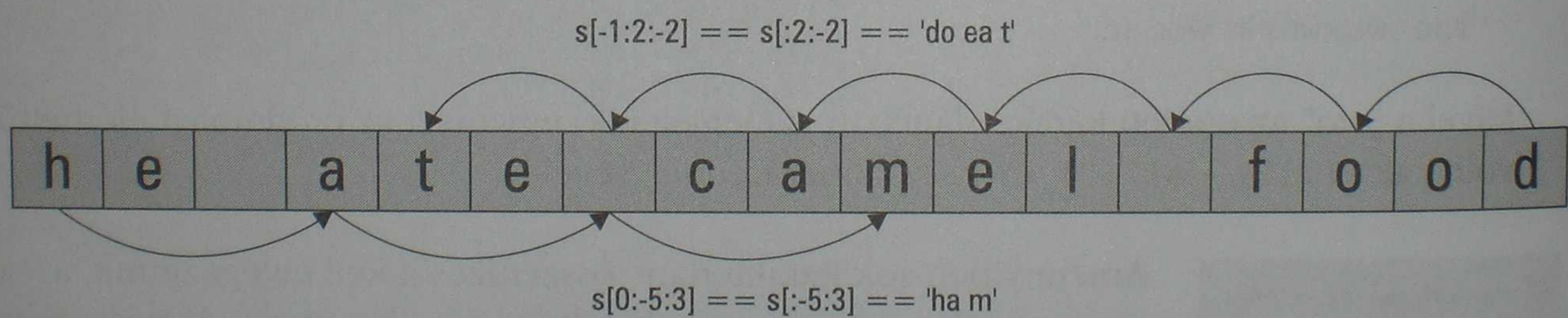
2.3. ábra

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. `bytes` típus ☞ 297
`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()`. 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. `str.format` ☞ 75, 83
`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()`.

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

csv2-html.py example
☞ 91

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.

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'
```

Mikként 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'
```

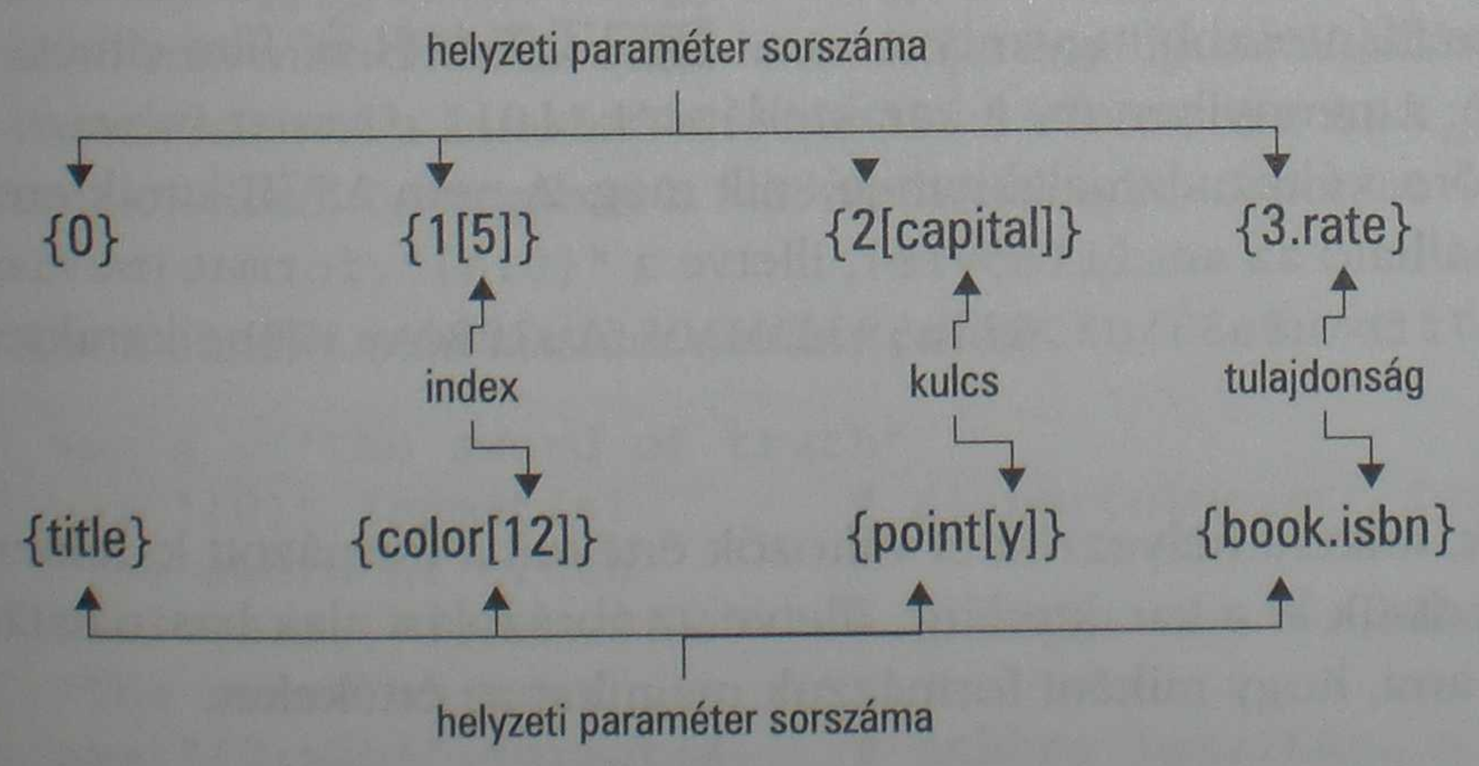
Összegezve a látottakat elmondhatjuk, hogy a mezőnevek használati alakja lehetővé teszi, hogy hivatkozzunk bármely helyzeti, illetve kulcsszó-paraméterre, amelyet az `str.format()` tagfüggvénynek átadtunk. Amennyiben a paraméterek gyűjtemény típusúak – például listák vagy szótárak –, illetve rendelkeznek 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 karakterlánctagfüggvény, 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 karakter 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 kódolással 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.

Példák

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

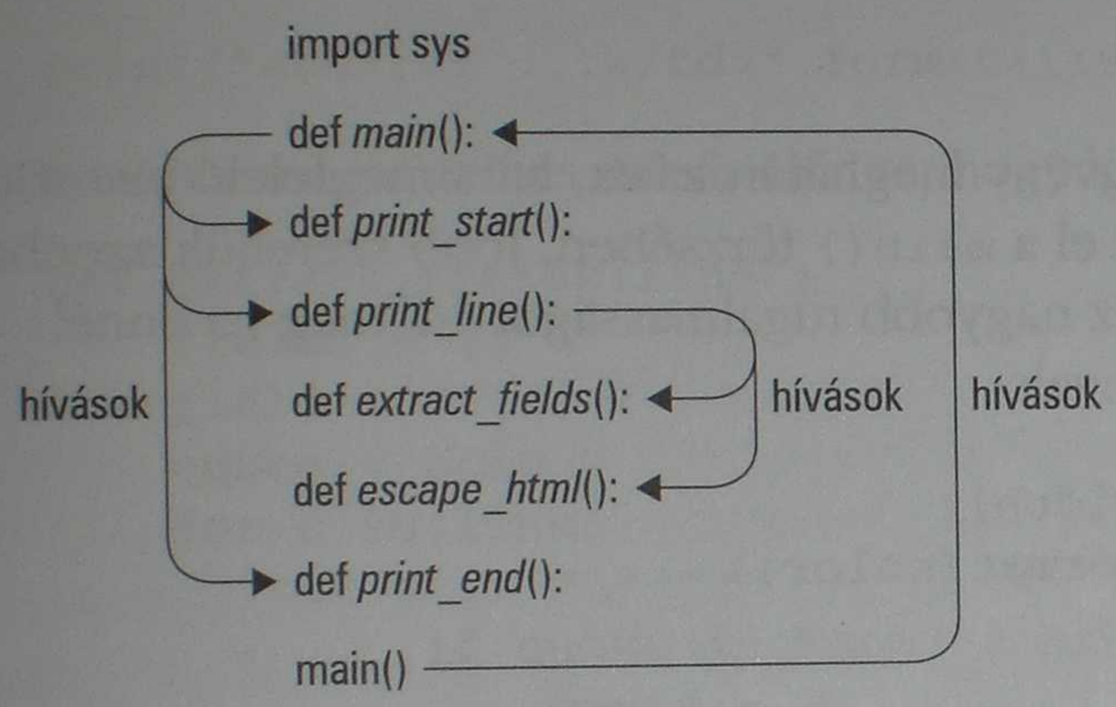
Láttuk hát, hogyan használhatjuk a programot, és milyen eredményt kaphatunk, így nekiláthatunk a kód áttekintésének. A program a `sys` modul betöltésével veszi kezdetét – ezt és az egyéb betöltéseket a következőkben nem mutatjuk be, hacsak nem valamilyen szokatlan vagy figyelemreméltó esetről van szó.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

range()
☞ 136

```
for i in range(len(numbers)):
    numbers[i] += 1
```

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.)

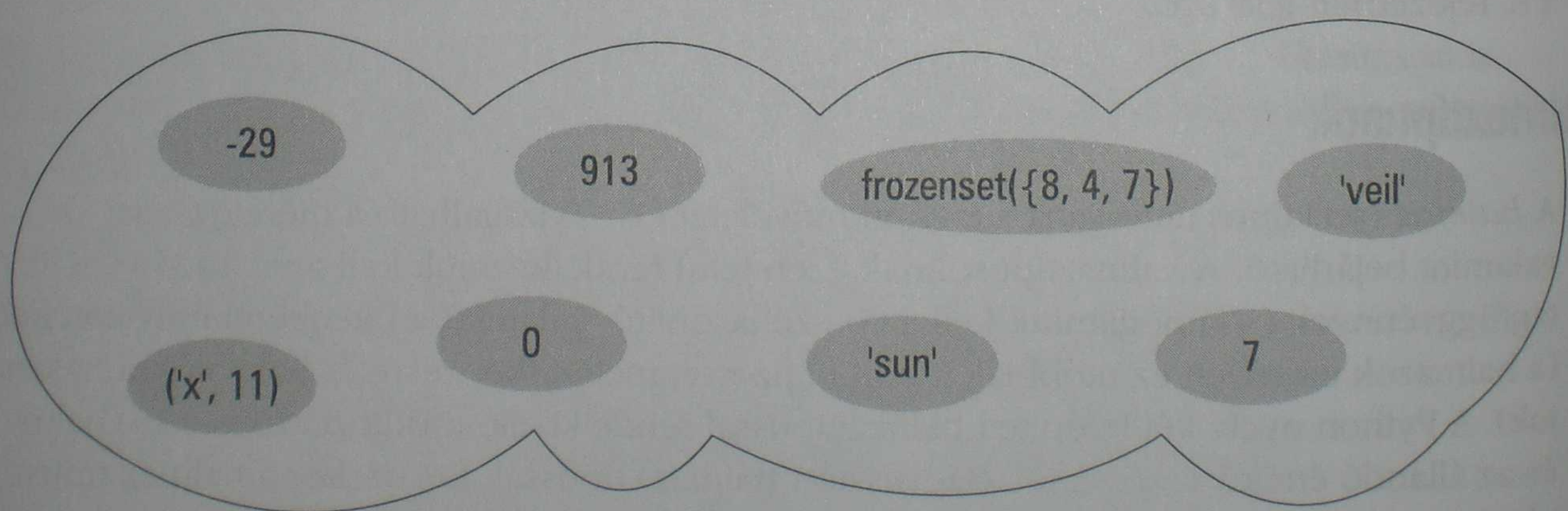
A beépített állandó értékű adattípusok (mint a `float`, a `frozenset`, az `int`, az `str` és a `tuple`) mindegyike kivonatolható, és így halmazba helyezhető. A beépített változtatható adattípusok (a `dict`, a `list` és a `set`) nem kivonatolhatók, hiszen a kivonatértékük (hash) a bennük tárolt objektumok 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:

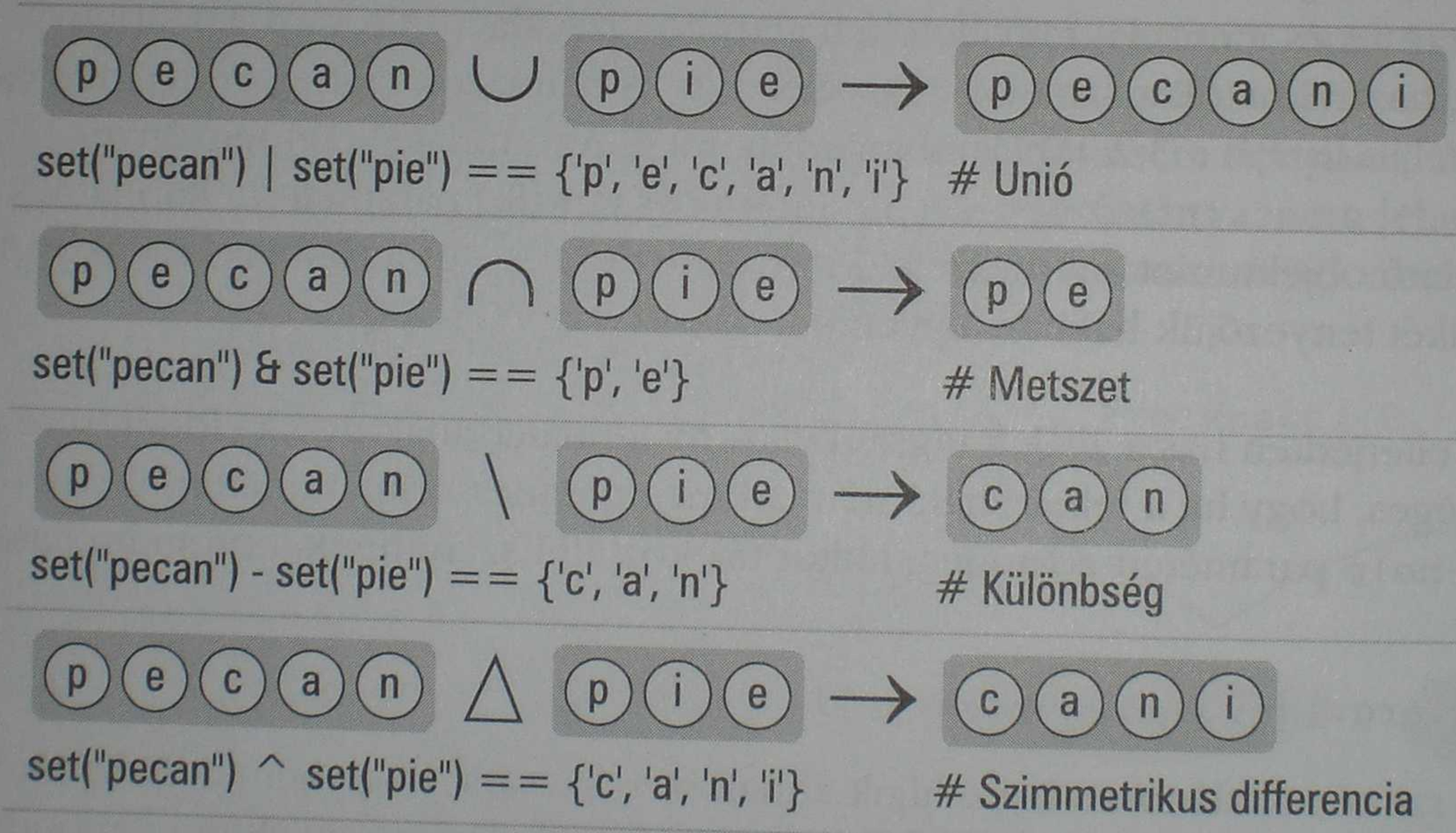
```
seen = set()
for ip in ips:
if ip not in seen:
seen.add(ip)
process_ip(ip)
```

```
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

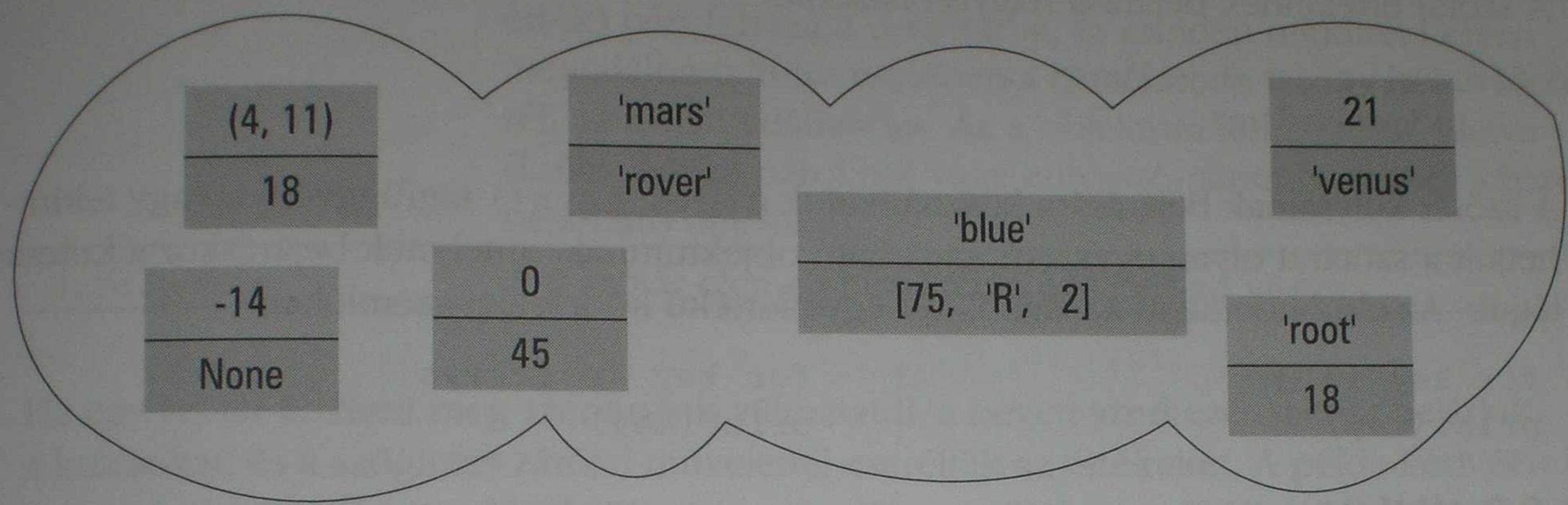
Íme néhány példa a különféle utasításformák szemléltetésére – mindegyik ugyanazt a szótárat hozza létre:

```
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

7. fejezet: *Fájlkezelés* ☞ 287

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.

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 tartalmazza, 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. Feltételezve, 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.

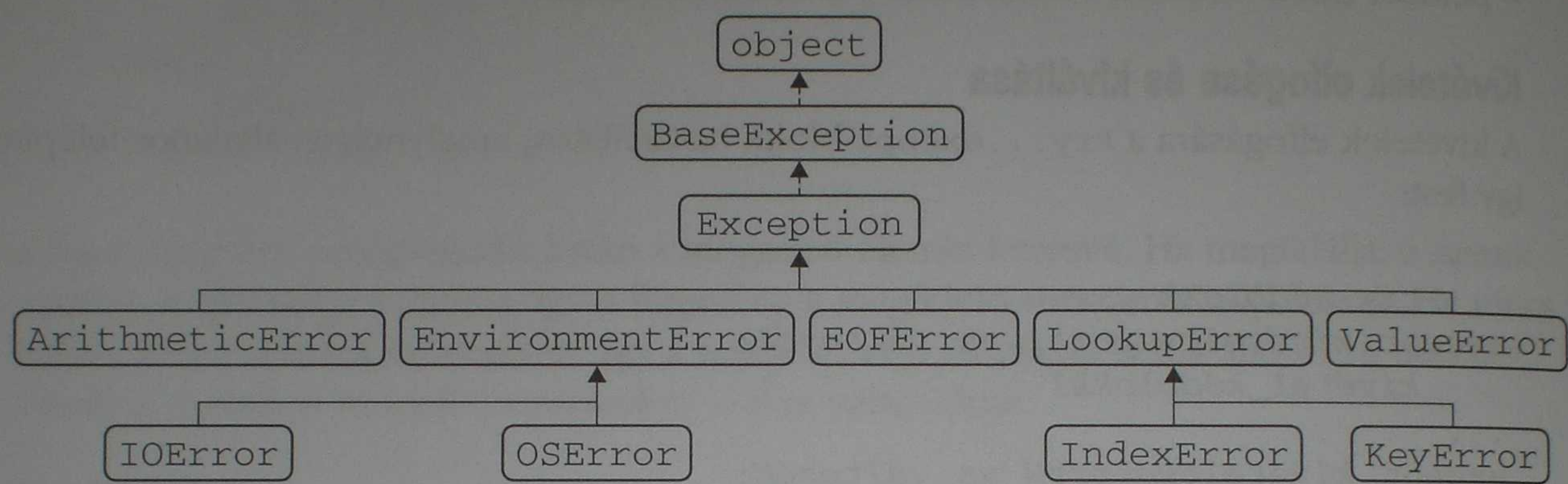
Az `except` utasítások kivételcsoportjaiban szerepelhet egyetlen kivétel, vagy kivételek rendezett sorozata. A változó rész feltüntetése nem kötelező – amennyiben használatba vesszük, ezen keresztül hivatkozhatunk a megfelelő blokkban magára a kivételre.

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.*

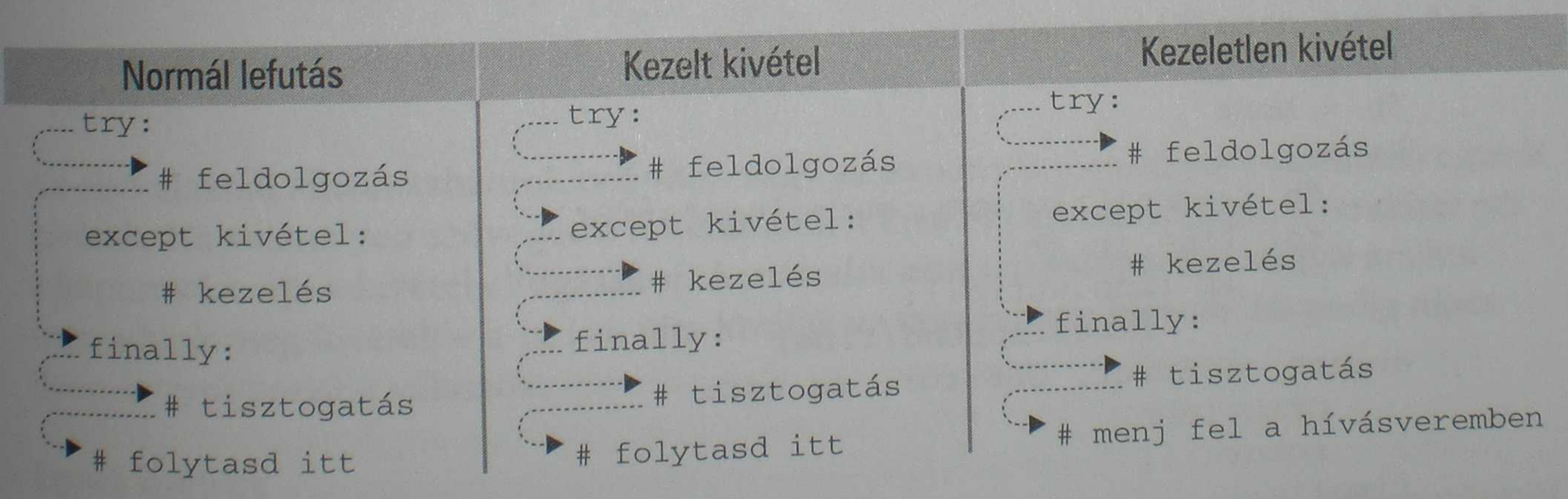
```
try:
    x = d[k / n]
except Exception: # HELYTELEN GYAKORLAT
    print("Something happened")
```

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.



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:
        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
```

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.