A `while` ciklusban a program először kiír egy címfeliratot, majd meghívja a `populate_information()` függvényt, átadva neki az `information` szótárat, amelyet frissítünk a függvény belsejében. Ezt követi a `make_html_skeleton()` függvény hívása, amely számos paramétert vár. Ahelyett azonban, hogy ezeket egyenként megadnánk, az `information` szótár kicsomagolt tartalmát adjuk át neki.

Amennyiben a felhasználó megszakítja a program futását, például úgy, hogy nem ad meg valamilyen kötelező adatot, a program a „Cancelled" üzenetet írja ki. Az egyes ciklusok végén (akár megszakította a folyamatot a felhasználó, akár nem), a program rákérdez, hogy szeretnénk-e újabb vázat készíteni – amennyiben nem, kilép a ciklusból, és a futása véget ér.

```
def populate_information(information):
    name = get_string("Enter your name (for copyright)", "name",
                      information["name"])
    if not name:
        raise CancelledError()
    year = get_integer("Enter copyright year", "year",
                       information["year"], 2000,
                       datetime.date.today().year + 1, True)
    if year == 0:
        raise CancelledError()
    filename = get_string("Enter filename", "filename")
    if not filename:
        raise CancelledError()
    if not filename.endswith((".htm", ".html")):
        filename += ".html"
    ...
    information.update(name=name, year=year, filename=filename,
                       title=title, description=description,
                       keywords=keywords, stylesheet=stylesheet)
```

A címfelirat és a leíró szövegek, a HTML-kulcsszavak, valamint a stíluslap beolvasásának kódját nem tüntettük fel – azt azonban elárulhatjuk, hogy mindegyikük a `get_string()` függvényt alkalmazza, amelyről hamarosan szót ejtünk. Elegendő annyit tudnunk, hogy a függvény a kiírandó üzenetet, a megfelelő változó „nevét" (ezt a hibaüzenetekben használjuk), valamint az esetleges alapértelmezett értéket várja. Hasonlóan, a `get_integer()` függvény is egy kiírandó üzenetet, egy változónevet, egy alapértelmezett értéket, illetve egy minimum- és maximumértéket vár, valamint azt is megadhatjuk, hogy a 0 érték megengedett-e.

Végezetül, kulcsszó-paraméterek segítségével frissítjük az `information` szótárat az új értékekkel. Az egyes *`kulcs=érték`* párok esetében a *`kulcs`* a szótár azon kulcsának neve, amelyet az adott *`érték`*-kel helyettesítünk – az *`érték`* ugyanakkor ez esetben egy olyan változót jelent, amelynek a neve megegyezik a szótárban található, neki megfelelő kulcséval.

A függvénynek nem adtunk meg visszatérési értéket (így a `None` értéket adja vissza). A futása megszakadhat, amennyiben a `CancelledError` kivétel bekövetkezik – ezt a rendszer továbbküldi felfelé a hívásveremben, és végül a `main()` függvény kezeli.

A `make_html_skeleton()` függvényt két részre bontva mutatjuk be:

```
def make_html_skeleton(year, name, title, description, keywords,
                       stylesheet, filename):
    copyright = COPYRIGHT_TEMPLATE.format(year,
                                          xml.sax.saxutils.escape(name))
    title = xml.sax.saxutils.escape(title)
    description = xml.sax.saxutils.escape(description)
    keywords = ",".join([xml.sax.saxutils.escape(k)
                         for k in keywords]) if keywords else ""
    stylesheet = (STYLESHEET_TEMPLATE.format(stylesheet)
                  if stylesheet else "")
    html = HTML_TEMPLATE.format(title=title, copyright=copyright,
                                description=description,
                                keywords=keywords,
                                stylesheet=stylesheet)
```

A szerzői jogi szöveg elkészítéséhez a `COPYRIGHT_TEMPLATE` sablonra meghívjuk az `str.format()` függvényt, helyzeti paraméterként átadva neki az aktuális évszámot, valamint (a megfelelően levédett) nevet, amelyek a `{0}` és az `{1}` mezők helyére kerülnek. A cím és a leírás eredeti szövegei helyett a levédett változataikat alkalmazzuk.

A HTML-kulcsszavak esetében két esetet különíthetünk el – erre egy feltételes kifejezést alkalmazunk. Amennyiben a felhasználó nem írt be kulcsszót, a `keywords` karakterláncot üressé tesszük a `""` értéket rendelve hozzá. Egyébként a kulcsszavak listáján végighaladva létrehozzuk a levédett változataikat, majd az `str.join()` függvénnyel egyetlen karakterláncban, vesszővel elválasztva soroljuk fel az így kapott kulcsszavakat.

A stíluslapot a szerzői jogi szöveghez hasonlóan készítjük el, ezúttal azonban mindezt egy feltételes kifejezés belsejében tesszük meg, így ha üres karakterláncot kapunk, egyáltalán nem határozunk meg stíluslapot.

`str.format()`
75

A `html` szöveget a `HTML_TEMPLATE` sablonból készítjük el, most azonban – eltérően más sablonoktól – helyzeti paraméterek helyett kulcsszó-paraméterekkel érjük el a cseremezőket.

```
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        fh.write(html)
    except EnvironmentError as err:
        print("ERROR", err)
```

```
else:
    print("Saved skeleton", filename)
finally:
    if fh is not None:
        fh.close()
```

Miután elkészítettük a HTML-kódot, nem marad más hátra, mint tárolni a megadott nevű fájlban. Tájékoztatjuk a felhasználót, hogy az oldal vázát lemezre mentettük, illetve kiírjuk a megfelelő hibaüzenetet, ha időközben valami rosszul sült el. Végezetül egy `finally` blokkban biztosítjuk, hogy a korábban megnyitott fájlokat be is zárjuk.

```
def get_string(message, name="string", default=None,
               minimum_length=0, maximum_length=80):
    message += ": " if default is None else " [{0}]: ".format(default)
    while True:
        try:
            line = input(message)
            if not line:
                if default is not None:
                    return default
                if minimum_length == 0:
                    return ""
                else:
                    raise ValueError("{0} may not be empty".format(
                                                                name))
            if not (minimum_length <= len(line) <= maximum_length):
                raise ValueError("{0} must have at least {1} and "
                        "at most {2} characters".format(
                        name, minimum_length, maximum_length))
            return line
        except ValueError as err:
            print("ERROR", err)
```

A függvénynek egyetlen paraméterét (`message`) kell feltétlenül meghatároznunk, a többi négy megadása nem kötelező. Ha valahol megadtunk egy alapértelmezett értéket, azt elhelyezzük a `message` karakterláncban is, így a felhasználó látja, mi is kerül a kódba, ha szöveg beírása nélkül lenyomja az ENTER billentyűt. A függvény további része egy végtelen ciklus belsejében található, amelyből a felhasználó úgy léphet ki, ha beír egy érvényes karakterláncot, illetve az ENTER lenyomásával elfogadja az alapértelmezést (amennyiben létezik ilyen).

A felhasználó számára kínálkozik még egy mód a ciklus (és ezzel együtt a teljes program) elhagyására – a CTRL+C billentyűk lenyomása, ami a `KeyboardInterrupt` kivételhez vezet. Mivel pedig ezt a program egyetlen kivételkezelője sem kezeli, megjelenése a program leállásához vezet, és a rendszer egy visszakövetési üzenetet ír ki. Érdemes ilyen „kiskaput" hagynunk? Nos, ha nem tesszük, és a programban benn marad valamilyen szerencsétlen

hiba, meglehet, hogy a felhasználó végtelen ciklusba kerül, amelyből csak a folyamat megsemmisítésével léphet ki. Ezért – ha csak valamilyen lényeges szempont nem indokolja ennek ellenkezőjét – tegyük lehetővé a program megszakítását a CTRL+C billentyűkombinációval – vagyis ne fogjuk el a megfelelő kivételt.

Érdemes észrevennünk, hogy ezt a függvényt nem csak a `make_html_skeleton.py` programban használhatjuk – a lehetőségei jól jöhetnek más esetekben is, amikor a felhasználóval „párbeszédet" kell folytatnunk. Újrahasznosítására alkalmas lehet a másolás és beillesztés módszere, de ez a programok karbantartása során sok fejfájást okozhat. A következő fejezetben ezekre a problémákra is megoldást lelhetünk, hiszen megismerkedünk az egyedi modulokkal, amelyek lehetővé teszik az ilyen és ehhez hasonló függvények megosztását tetszőleges számú program között.

```
def get_integer(message, name="integer", default=None, minimum=0,
                maximum=100, allow_zero=True):
    ...
```

Ez a függvény olyannyira hasonlít a `get_string()`-hez, hogy felesleges itt bemutatnunk. (A könyvhöz mellékelt forráskódban természetesen megtalálható.) Az `allow_zero` paraméter akkor lehet hasznos, ha a 0 nem érvényes érték, de egyetlen érvénytelen érték használatát szeretnénk lehetővé tenni, hogy ezzel jelezhessük, ha a felhasználó megszakította a műveletet. Hasonló megoldás az is, amikor egy érvénytelen alapértelmezett értéket adunk át, és ha ezt kapjuk vissza, azt úgy értelmezzük, hogy a felhasználó a művelet megszakítása mellett döntött.

Programunk utolsó utasítása a `main()` függvény meghívása. A kód összességében kicsit több mint 150 soros, és felidéz számos, ebben és az ezt megelőző fejezetben megtanult módszert.

Összefoglalás

Ebben a fejezetben bemutattuk a Python vezérlési szerkezeteinek használati alakját, emellett megtanultuk, hogy miként váltsunk ki és fogjunk el kivételeket, továbbá hogyan hozzunk létre egyedi kivételtípusokat.

A fejezet legnagyobb részében az egyedi függvényekkel foglalkoztunk. Megismerkedtünk a létrehozásuk módjával, továbbá néhány, a neveikre és paramétereikre vonatkozó elnevezési szabállyal. Megtudtuk azt is, miként adhatunk meg leírást a függvényeinkhez. A Python rugalmas paraméterhasználatáról és a paraméterek átadásáról részletesen beszámoltunk – megtanultuk, mit kell tennünk, ha rögzített, illetve tetszőleges számú helyzeti vagy kulcsszó-paramétert szeretnénk alkalmazni. Szót ejtettünk emellett mind az állandó értékű, mind a változtatható adattípusok alapértelmezett értékeiről is. Röviden emlékeztettünk a * műveletre, amellyel sorozatokat csomagolhatunk ki, és megismerkedtünk a ** művelettel, amely a megfeleltetések kicsomagolására ad módot.

Amennyiben egy függvény belsejében új értéket szeretnénk adni egy globális változónak, ezt úgy tehetjük meg, ha a `global` kulcsszó feltüntetése mellett hivatkozunk rá, egyébként a Python egy azonos nevű helyi változót hoz létre, és ehhez rendeli az értéket. Általánosságban elmondható azonban, hogy a globális változókat jobb, ha kizárólag állandók tárolására használjuk.

A lambdafüggvényeket gyakran használják kulcsfüggvényként, illetve olyan helyeken, ahol paraméterként függvényt kell átadni. Ebben a fejezetben bemutattuk, hogy miként hozhatunk létre lambdafüggvényeket, akár névtelen függvényként alkalmazva őket, akár változókhoz rendelve egysoros, névvel ellátott függvényként hivatkozva rájuk.

A fejezetben megismerkedtünk az `assert` utasítással is, amellyel olyan elő- és utófeltételezéseket rögzíthetünk, amelyeknek az adott függvény minden használatánál teljesülniük kell. Ezek sokat segíthetnek a hibakeresésben, illetve abban, hogy kevésbé sérülékeny programokot készítsünk.

Megismerkedtünk hát a függvények írásának alapjaival, de ezen felül is számos módszer áll a rendelkezésünkre. Így alkalmazhatunk dinamikus függvényeket, vagyis a függvényeket futásidőben hozhatjuk létre, a megvalósításukat az aktuális körülményekhez igazítva (róluk az 5. fejezetben szólunk), helyi (beágyazott) függvényeket (ezek tárgyalására a 7. fejezetben kerül sor), továbbá önhivatkozó és generátorfüggvényeket, valamint más egyebeket is (ezekről a lehetőségekről a 8. fejezetben olvashatunk bővebben).

Jóllehet a Python számos beépített lehetőséggel rendelkezik, és az alapkönyvtára is kiterjedt, mégis jó eséllyel írunk majd olyan függvényeket, amelyeket több programban is fel tudunk használni. Ezeknek a függvényeknek az ide-oda másolgatása komoly gondokat okozhat a programok fenntartásában, de a Python szerencsére egy tiszta és egyszerű megoldással siet a segítségünkre: lehetővé teszi egyedi modulok létrehozását. A következő fejezetben ezekről esik szó, valamint arról, hogy miként helyezhetjük el bennük a saját függvényeinket. Megtanuljuk továbbá, hogyan töltsük be az alapkönyvtár és saját moduljaink lehetőségeit, és röviden áttekintjük, hogy mit kapunk készen az alapkönyvtártól, hogy kreativitásunkat a valódi feladatokra tartogathassuk.

Gyakorlat

Írjunk interaktív programot, amely képes karakterláncok fájlokban tárolt listáinak kezelésére!

A program készítsen listát azokról a fájlokról, amelyek az aktuális könyvtárban találhatók, és a kiterjesztésük `.lst`. Alkalmazzuk az `os.listdir(".")` parancsot az összes fájl nevének beolvasására, és szűrjük ki azokat, amelyek nem rendelkeznek `.lst` kiterjesztéssel. Ha nincs egyetlen találat sem, kérjük meg a felhasználót, hogy írjon be egy fájlnevet – ezt szükség esetén egészítsük ki az `.lst` kiterjesztéssel. Ha léteznek `.lst` fájlok a könyvtár-

ban, egy számozott lista alakjában írjuk a neveiket a képernyőre, az 1-es sorszámmal kezdve. Kérjük meg a felhasználót, hogy válasszon a sorszámok közül, illetve írjon be 0-t, ha új fájlt szeretne meghatározni.

Amennyiben a felhasználó már létező fájlt adott meg, olvassuk be ennek elemeit. Ha a fájl üres, illetve új fájlt hoztunk létre, egy üzenetet kell kiírnunk, ami szerint „nincsenek elemek a listában".

Amennyiben nincsenek listaelemek, az Add (Hozzáadás) és a Quit (Kilépés) lehetőségeket kell elérhetővé tennünk. Ha a lista nem üres, írjuk ki az elemeit egy 1-től kezdve sorszámozott listában, és ajánljuk fel az Add, Delete (Törlés), Save (Mentés; amennyiben vannak még nem mentett beállítások) és Quit lehetőségeket. Ha a felhasználó ez utóbbit választja, és léteznek még nem mentett módosítások, adjunk lehetőséget a mentésre. Íme egy a programmal folytatott munkamenet átirata (a legtöbb üres sort, valamint az egyes listák felett álló List Keeper feliratot elhagytuk).

```
Choose filename: movies

-- no items are in the list --
[A]dd [Q]uit [a]: a
Add item: Love Actually
1: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: a
Add item: About a Boy
1: About a Boy

2: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]:
Add item: Alien

1: About a Boy
2: Alien
3: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: k

ERROR: invalid choice--enter one of 'AaDdSsQq'
Press Enter to continue...
[A]dd [D]elete [S]ave [Q]uit [a]: d
Delete item number (or 0 to cancel): 2

1: About a Boy
2: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: s
Saved 2 items to movies.lst
Press Enter to continue...

1: About a Boy
2: Love Actually
[A]dd [D]elete [Q]uit [a]:
Add item: Four Weddings and a Funeral
```

```
1: About a Boy
2: Four Weddings and a Funeral
3: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: q
Save unsaved changes (y/n) [y]:
Saved 3 items to movies.lst
```

Gondoskodjunk róla, hogy a `main()` függvény elég kicsi legyen (30 sornál rövidebb), és helyezzük el itt a program központi ciklusát. Írjunk egy függvényt, amely beolvassa az új vagy már létező fájlnevet (az utóbbi esetben pedig be is tölti az elemeket), továbbá egy másikat, amely a felhasználó elé tárja a lehetőségeket, és kezeli a választást. Szükségünk lesz függvényekre, amelyek gondoskodnak az egyes elemek hozzáadásáról, törléséről, a lista kiírásáról (legyen szó elemekről vagy fájlnevekről), betöltéséről és mentéséről. Lemásolhatjuk a `make_html_skeleton.py` programban található `get_string()` és `get_integer()` függvényeket, de elkészíthetjük a saját változatainkat is.

A fájlnevek listájának kiírásakor tíznél kevesebb elem esetén alkalmazzunk 1 karakter szélességű mezőt, 100-nál kevesebb elemnél 2, egyébként pedig 3 szélességűt. Tároljuk az elemeket a betűméretre érzéketlen, ábécésorrendbe állított listában, és tartsuk számon, hogy a lista „piszkos-e", azaz vannak-e még nem tárolt módosítások. A Save lehetőséget csak akkor ajánljuk fel, ha a lista piszkos, továbbá csak ilyen esetben kérdezzünk rá a módosítások mentésére, miután a felhasználó a Quit lehetőséget választotta. Az elemek beillesztése és törlése piszkossá teszi a listát, míg a mentés újra „kitisztítja".

Egy lehetséges megoldást megtalálhatunk a `listkeeper.py` fájlban – ez a program 200-nál kevesebb sort tartalmaz.

5

Modulok

- Modulok és csomagok
- A Python alapkönyvtárának áttekintése

Míg a függvényekkel lehetővé válik a kód feldarabolása, illetve az egyes darabok többszöri felhasználása a programon belül, a modulokkal úgy tudjuk a függvényeket (és – ahogy a következő fejezetben látni fogjuk – az egyedi adattípusokat) csoportokba rendezni, hogy akár valamennyi programunkban használhatók lesznek. A Python rendelkezik olyan eszközökkel is, amelyekkel *csomagokat*, vagyis csoportokba szervezett modulokat hozhatunk létre. A csomagokban lévő modulok általában vagy egymáshoz szorosabban kapcsolódó részterületeket fednek le, vagy egymásra támaszkodnak a működésük során.

A fejezet első része a modulokban és csomagokban tárolt függvények használatba vételéről szól, az alapkönyvtárban lévő és a saját magunk írta modulokra és csomagokra egyaránt vonatkozóan. Ezt követően bemutatjuk a saját csomagok és saját modulok létrehozásának mikéntjét. Két saját készítésű modult mutatunk be; az első példa inkább bevezető jellegű, a második pedig a gyakorlatban felmerülő problémák sokaságának kezelését mutatja be – például a rendszerfüggetlenség és a tesztelés kérdését.

Elektronikus dokumentáció
167 ☞

A fejezet második része a Python alapkönyvtáráról (standard library) nyújt rövid áttekintést. Fontos tisztában lennünk azzal, hogy milyen lehetőségeket nyújt az alapkönyvtár, hiszen a programozás lényegesen gyorsabb a már megírt függvények használatával, mintha mindent a nulláról indulva írnánk meg. Ezen felül az alapkönyvtár moduljait általában sokan használják – így széleskörű a tesztelésük, és kellően megbízhatók. Az áttekintésen túl néhány rövid példával szemléltetjük az alapkönyvtár mindennapi használatát, valamint utalunk a többi fejezetben tárgyalt modulokra.

Modulok és csomagok

Ha egyszerűen fogalmazunk, a Python-modulok nem egyebek `.py` kiterjesztésű fájloknál. A modulokban olyan Python nyelvű kódot helyezünk el, amilyet csak akarunk. Az eddig megírt programjaink mind egy-egy `.py` kiterjesztésű fájlt jelentenek, tehát egyszerre modul és program valamennyi. A legfőbb különbség abban rejlik, hogy amíg a programokat futtatásra szánjuk, addig a modulok célja, hogy a programjainkba betöltve használhassuk őket.

Nem minden modul esetében találjuk meg a modulnak megfelelő `.py` kiterjesztésű fájlt – példának okáért a `sys` modul a Pythonba beépítve létezik, és van néhány olyan modul is, amelyet más nyelven, főleg C-ben írtak. Ugyanakkor a Python alapkönyvtárának legnagyobb része Python nyelven készült, így például amikor az `import collections` utasítást szerepeltetjük a programunkban, a `collections.namedtuple()` tagfüggvény hívásával nevesített rendezett sorozatokat hozhatunk létre, a tagfüggvény maga pedig a `collections.py` fájlban található. A programjaink szempontjából nem érdekes, hogy egy modult melyik programozási nyelven írtak, ugyanis minden modul betöltése és használata azonos módon történik. A modulok betöltésekor használt utasításforma többféle lehet, például:

```
import valami
import valami1, valami2, …, valamiN
import valami as nekünk_tetsző_név
```

A feni példában a *`valami`* általában egy modult jelöl – például a `collections` modult. Lehet azonban csomag is vagy egy csomagon belüli modul – ez esetben a név részeit pont (`.`) választja el, mint az `os.path` esetében is. A könyvben az első két utasításformát használjuk, mert ezek a legegyszerűbbek és a legbiztonságosabbak, és minthogy így rákényszerülünk a teljes nevek használatára, biztosan nem lesz a nevek keveredéséből származó problémánk.

A harmadik forma lehetővé teszi, hogy a betöltött modult, illetve csomagot az általunk választott névvel lássuk el – ez elvileg névütközéshez is vezethetne, de a gyakorlatban ez elkerülhető az `as` kulcsszó használatával. Az átnevezés különösen akkor jöhet jól, amikor egy modul különféle megvalósításaival kísérletezgetünk. Példának okáért olyankor, amikor két, egyforma API-val (Application Programming Interface, alkalmazásprogramozási felület) rendelkező modulunk van – `MyModuleA` és `MyModuleB` – a programba elegendő az `import MyModuleA as MyModule` sort írnunk, később pedig minden különösebb nehézség nélkül átválthatunk a `MyModuleB` használatára.

Hol érdemes az `import` utasítást tartalmazó kifejezéseket elhelyeznünk? Általánosan elterjedt gyakorlat, hogy az `import` utasításokat a `.py` kiterjesztésű állományok elején, a shebang-sor és a modul dokumentációja után szerepeltetjük. Ahogy azt régebben, az első fejezetben már említettük, azt tanácsoljuk, hogy az alapkönyvtár moduljait töltsük be elsőként, ezt kövessék a máshonnan származó modulok, és a sort a saját moduljainkkal zárjuk.

Íme néhány egyéb forma az `import` utasítás használatára:

```
from valami import objektum as nekünk_tetsző_név
from valami import objektum1, objektum2, ..., objektumN
from valami import (objektum1, objektum2, objektum3, objektum4,
objektum5,
    objektum6, ..., objektumN)
from valami import *
```

A fenti utasításformák vezethetnek névütközéshez, ugyanis a használatukkal a betöltött objektumok (változók, függvények, adattípusok és modulok) közvetlenül elérhetőkké válnak. Ha a `from...import` utasítással sok objektum betöltését szeretnénk elvégezni, többsoros utasításokat is írhatunk: az utolsó kivételével levédjük az új sorokat, vagy a harmadik utasításformának megfelelően zárójelbe tesszük az objektumok neveit.

Az utolsó példában előkerülő csillag azt jelenti, hogy „tölts be mindent, ami nem privát". Ez a gyakorlatban az aláhúzással kezdődő nevű objektumok kivételével minden objektum betöltését jelenti, illetve amennyiben a modulban található egy névlistát tartalmazó globális `__all__` változó, akkor az ebben a változóban megnevezett objektumokat töltjük be.

`__all__` ☞ 197

Íme néhány, az `import` utasítás használatát bemutató példa:

```
import os
print(os.path.basename(filename))  # biztonságos,
                                   # teljes nevet használó elérés

import os.path as path
print(path.basename(filename))  # a path objektummal való
                                # névütközést kockáztatjuk

from os import path
print(path.basename(filename))  # a path objektummal való
                                # névütközést kockáztatjuk

from os.path import basename
print(basename(filename))  # a basename objektummal való
                           # névütközést kockáztatjuk

from os.path import *
print(basename(filename))  # sok névütközést kockáztatunk
```

A `from valami import *` formával a modul valamennyi objektumát (vagy a csomag valamennyi modulját) betöltjük, és ez nevek százait jelentheti. A `from os.path import *` utasítás esetében majdnem negyven nevet töltünk be – ezek között megtalálható a `dirname`, az `exists` és a `split`, csupa olyan, amelyet akár saját változóinak, illetve függvényeinek is névül választ az ember.

Például, ha kiadjuk az `from os.path import dirname` utasítást, kényelmesen meghívhatjuk a `dirname()` függvényt, anélkül, hogy a teljes nevét megadnánk. Ha azonban a programunkban később kiadjuk a `dirname = "."` utasítást, a `dirname` nevű objektumhivatkozás

a `dirname()` függvény helyett immár a „.“ karakterláncra fog hivatkozni. Ha ekkor a `dirname()` függvény hívásával próbálkozunk, `TypeError` kivétel lép fel, hiszen a `dirname` ekkorra egy karakterláncot jelöl, és a karakterláncokat nem lehet meghívni.

Az `import *` utasításforma által okozható névütközésekre való tekintettel néhány programozói csapatban alapelv, hogy csak az `import valami` utasításformát használják. Mindazonáltal néhány terjedelmes csomag esetében – különös tekintettel a GUI (Graphical User Interface – grafikus felhasználói felület) programozására szolgáló könyvtárakra – gyakran használják az `import *` utasításformát, ugyanis ezekben olyan rengeteg függvény és osztály (saját adattípus) fordul elő, hogy a kézzel történő begépelésük könnyen hosszadalmassá válhat.

Azonnal adódik a kérdés: honnan tudja a Python, hogy hol kell keresnie a betöltendő modulokat, illetve csomagokat? A beépített `sys` modulban található egy `sys.path` nevű lista: az itt szereplő könyvtárak együttese a Python elérési útvonalainak jegyzéke, a *Python path*. Az első mappa az, amelyik magát a programot tartalmazza – abban az esetben is, ha a programot egy másik mappából indítottuk el. Ha a `PYTHONPATH` környezeti változónak van beállított értéke, akkor az itt megadott mappák következnek a listában. A lista végén lévő mappák a Python alapkönyvtárának eléréséhez szükségesek – ezek a Python telepítése során kapnak értéket.

Amikor először töltünk be egy nem beépített modult, a Python egymás után átvizsgálja a `sys.path` listában feltüntetett könyvtárakat. Ennek a műveletnek az egyik eredménye többek között az, hogy amennyiben a Python alapkönyvtárának valamelyik moduljával megegyező nevű modult készítünk, a saját modulunkat találja meg először a Python, ami nyilvánvalóan problémák forrása lesz. Ez úgy kerülhető el, ha soha nem készítünk a Python alapkönyvtárának felső szintű moduljaival vagy mappáival megegyező nevű programot és modult – kivéve, ha az adott modul saját megvalósítását készítjük el, és szándékosan bíráljuk felül az eredetit. (A felső szintű modulok olyan modulok, amelyek `.py` kiterjesztésű állománya valamelyik, a `PYTHONPATH`-ban lévő mappában, és nem ezek egyik almappájában található.) Windowson dolgozva a `PYTHONPATH` például rendszerint tartalmaz egy `C:\Python30\Lib` nevű mappát, vagyis ezen az operációs rendszeren ne készítsünk sem `Lib.py` nevű modult, sem olyat, amelynek a neve megegyezik valamelyik, a `C:\Python30\Lib` nevű mappában lévő modulokéval.

Figyelem! *Az egyik gyors módszer annak ellenőrzésére, hogy egy modulnév már foglalt-e, ha megpróbálkozunk a modul betöltésével. Ezt a konzolban úgy tehetjük meg, hogy a parancsértelmezőt a* `-c` *(„kódvégrehajtás”) parancssori paraméterrel indítjuk el, amelyet a betöltési utasítás követ. Ha például azt akarjuk kideríteni, hogy létezik-e modul* `Music.py` *néven (vagy egy felső szintű,* `Music` *nevű mappa a Python elérési útvonalain), a konzolban a következő utasítást kell kiadnunk.*

```
python -c "import Music"
```

Ha `ImportError` kivételt kapunk, tudhatjuk, hogy ilyen nevű modul, illetve felső szintű mappa nincs használatban. Bármilyen más kimenet – vagy a kimenet hiánya – annyit jelent, hogy a név már foglalt. Sajnos az, hogy a név jelenleg még használható, semmiféle garanciát nem jelent arra nézve, hogy ez mindig így lesz. Később ugyanis elképzelhető, hogy olyan, más fejlesztőtől származó Python-csomagot vagy -modult telepítünk, amelyben a mienkkel ütköző név található – mindenesetre a gyakorlatban ez igen ritkán okoz gondot.

Ha például egy `os.py` nevű modult készítenénk, névütközés lépne fel az alapkönyvtár `os` nevű modulja miatt. Ha azonban `path.py` nevű modult készítünk, akkor minden rendben, ugyanis az alapkönyvtár modulját `os.path` néven tölthetjük be. Ebben a könyvben a saját moduljaink fájlneveit nagy kezdőbetűvel írjuk, ezzel a névütközés – legalábbis Unixon – elkerülhető; az alapkönyvtár fájlnevei ugyanis kis kezdőbetűsek.

Egy program betölthet néhány modult, amelyek aztán további modulokat tölthetnek be, akár olyanokat is, amelyeket már előzőleg betöltöttünk – ez nem okoz problémát.
A Python minden betöltési utasítás végrehajtása előtt ellenőrzi, hogy a modult korábban nem töltötte-e be. Ha nem, akkor a Python végrehajtja a bájtkódúra fordított kódot, létrehozva ezzel a változókat, függvényeket és a kódban lévő egyéb objektumokat, és emellett feljegyzi magának, hogy ezt a modult már betöltötte. A Python minden további betöltés alkalmával felismeri, hogy a modult már betöltötte, és tétlen marad.

Amikor a Pythonnak a bájtkódúra fordított kódra van szüksége, automatikusan létrehozza azt. Ennyiben eltér például a Javától, ahol a bájtkódú fordítást külön kérnünk kell.
A Python először a modul `.py` kiterjesztésű állományát keresi, de `.pyo` kiterjesztéssel, vagyis az optimalizált, bájtkódúra fordított változatot. Ha a Python `.pyo` kiterjesztésű fájlt nem talál – vagy talál ugyan, de a `.py` kiterjesztésűnél régebbit, tehát nem a legfrissebb változatról készültet –, akkor `.pyc` kiterjesztésűt keres, ami a nem optimalizált bájtkódú változatot tartalmazza. Ha ez eléggé friss, akkor betölti, egyébként pedig betölti, és bájtkódúvá fordítja a `.py` kiterjesztésű változatot. A végeredmény minden esetben az, hogy a Python a memóriába tölti a bájtkódú modulváltozatot.

Ha a Pythonnak kellett elvégeznie a `.py` kiterjesztésű változat bájtkódra fordítását, a létrejött `.pyc` (ha a Python parancssorában megadtuk a `-O` kapcsolót, illetve a `PYTHONOPTIMIZE` környezeti változót úgy állítottuk be, akkor `.pyo`) kiterjesztésű fájlt menti is, már amennyiben a mappa írható. A bájtkódú változat mentése a `-B` kapcsolóval, illetve a `PYTHONDONTWRITEBYTECODE` környezeti változó beállításával letiltható.

A bájtkódú változatok használata gyorsabban induló programokat eredményez, lévén a parancsértelmezőnek a kód betöltése, fordítása, lehetőség szerinti mentése és futtatása helyett csak a kód betöltése és futtatása a dolga. A program maga természetesen nem fut gyorsabban. Az alapkönyvtár moduljait rendszerint a telepítési folyamat részeként fordítja bájtkódúra a Python.

Csomagok

A csomag (package) nem egyéb, mint egy olyan mappa, amely néhány modult, valamint egy `__init__.py` nevű fájlt tartalmaz. Tegyük fel például, hogy van egy képzeletbeli modulkészletünk, amellyel különféle grafikus formátumok írása és olvasása valósítható meg. A modulok: a `Bmp.py`, a `Jpeg.py`, a `Png.py`, a `Tiff.py`, és az `Xpm.py` mindegyike rendelkezik `load()`, `save()` és egyéb függvényekkel.* A modulok lehetnének a programunkkal egy mappában, de egy nagy program esetében, amely saját készítésű modulok sokaságára támaszkodik, a grafikai modulokat nehéz lenne így kezelnünk. Ha azonban egy külön alkönyvtárba, mondjuk a `Graphics` nevűbe kerülnek, akkor könnyebb az egységes kezelés. Ha még egy üres `__init__.py` fájlt is elhelyezünk közöttük, a mappa csomaggá lép elő:

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
    Tiff.py
    Xpm.py
```

Amennyiben a `Graphics` mappa a programunk mappájának egyik alkönyvtára, vagy megtalálható a Python elérési útvonalait rögzítő listában, a fenti modulok bármelyike betölthető és használható. Óvatosnak kell lennünk, nehogy a felső szintű modulnév (a `Graphics`) névütközésbe kerüljön az alapkönyvtár valamelyik felső szintű nevével. (Unix rendszereken ez könnyen kivédhető, ha a modul neve nagybetűvel kezdődik, lévén az alapkönyvtár nevei kisbetűvel kezdődnek. A modul betöltésének, illetve használatba vételének a módja a következő:

```
import Graphics.Bmp
image = Graphics.Bmp.load("bashful.bmp")
```

Rövid programok esetében néhány programozó előnyben részesíti a rövidebb nevek használatát. A Python kétféle módszert kínál erre.

```
import Graphics.Jpeg as Jpeg
image = Jpeg.load("doc.jpeg")
```

A fenti példában a `Graphics` csomag `Jpeg` moduljának betöltésekor szóltunk a Pythonnak, hogy a teljes név (`Graphics.Jpeg`) használata szerint jobb szeretnénk az egyszerűbb, `Jpeg` elnevezést használni.

```
from Graphics import Png
image = Png.load("dopey.png")
```

Ez a kódrészlet a `Graphics` csomag `Png` modulját közvetlenül tölti be a programba. A `from...import` utasításforma közvetlenül elérhetővé teszi a `Png` modult.

---

* A grafikus állományok kezelését a külső fejlesztők által készített modulok sokasága támogatja. A legfigyelemre-

méltóbb alighanem a Python Imaging Library (www.pythonware.com/products/pil).

Semmi sem kötelez bennünket arra, hogy a programunkban a csomag eredeti nevét használjuk. Például:

```
from Graphics import Tiff as picture
image = picture.load("grumpy.tiff")
```

Ebben a példában a Tiff modult használjuk, de más néven: a program picture néven hivatkozik rá.

Vannak olyan helyzetek, amikor kényelmes egy csomag összes modulját egyetlen utasítással betöltenünk. Ha ezt szeretnénk tenni, a csomag __init__.py állományát úgy kell átszerkesztenünk, hogy tartalmazza a betölteni kívánt modulok neveit megadó utasítást. Az utasításnak a modulokat az __all__ különleges szerepű változónak kell értékül adnia. Például a Graphics/__init__.py fájlban az alábbi sort kell elhelyeznünk:

```
__all__ = ["Bmp", "Jpeg", "Png", "Tiff", "Xpm"]
```

Mindössze ennek a sornak kell mindenképpen benne lennie az __init__.py fájlban, de igazából azt írunk még bele, ami nekünk tetszik. Most már használható az import utasítás egy újabb formája:

```
from Graphics import *
image = Xpm.load("sleepy.xpm")
```

A from *csomag* import * utasításformával az __all__ listában megnevezett valamennyi modult közvetlenül betöltjük a programunkba, vagyis az utasítás végrehajtása után az Xpm modulon felül az összes többi modul is közvetlenül elérhető lesz.

Ahogy korábban jeleztük, ez az utasításforma modulok esetében is alkalmazható, vagyis a from *modul* import * utasítással a modulban megadott összes függvény, változó és egyéb objektum (kivéve azokat, amelyeknek aláhúzással kezdődik a neve) betöltése megvalósul. Ha pontosan szabályozni szeretnénk, hogy minek a betöltése történik meg, amikor a from *modul* import * utasításformát használjuk, a modulban magában is megadhatunk egy __all__ nevű listát. Ebben az esetben a from *modul* import * utasítás csak az __all__ listában megnevezett objektumok betöltését végzi el.

Eddig az egymásba ágyazhatóságnak csak egyetlen szintjét mutattuk be, de a Python nyelv lehetővé teszi, hogy olyan mélységig ágyazzunk egymásba csomagokat, ameddig csak szeretnénk. Így a Graphics könyvtárban lehet mondjuk egy Vector nevű alkönyvtárunk, ebben pedig további modulok, például az Eps.py és az Svg.py:

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
```

```
Tiff.py

Vector/
    __init__.py
    Eps.py
    Svg.py
Xpm.py
```

Hogy a `Vector` könyvtár is csomagnak minősüljön, kell benne lennie egy `__init__.py` állománynak, amely, ahogy már jeleztük, lehet üres, illetve tartalmazhat egy `__all__` nevű listát, amely azoknak a programozóknak teszi kényelmesebbé az életét, akik a betöltést a `from Graphics.Vector import *` utasítással szeretnék elvégezni. A beágyazott csomagok eléréséhez a már használt utasításformából indulunk ki:

```
import Graphics.Vector.Eps
image = Graphics.Vector.Eps.load("sneezy.eps")
```

A teljes név meglehetősen hosszú, emiatt sok programozó igyekszik a modulok hierarchiáját minél kevesebb szinttel megvalósítani.

```
import Graphics.Vector.Svg as Svg
image = Svg.load("snow.svg")
```

Egy modulra mindig hivatkozhatunk az általunk megadott rövid néven, ahogy a fenti példában is tettük, ez a módszer azonban növeli a névütközés veszélyét.

Egyedi modulok

Minthogy a modulok tulajdonképpen egyszerű, `.py` kiterjesztésű fájlok, a létrehozásuk sem jelent különösebb gondot. Ebben a részben két saját készítésű modult tekintünk át. Az első, a `TextUtil.py` fájlban található `TextUtil` modul, mindössze három függvényt tartalmaz. Az `is_balanced()` függvény `True` értékkel tér vissza, ha a neki átadott függvényben a nyitó és záró zárójelek típusban és számban megfelelnek egymásnak. A `shorten()` függvényt korábban már bemutattuk, a `simplify()` pedig szóközöket és egyéb karaktereket távolít el egy karakterláncból. A modul tárgyalása során azt is megtudjuk, hogy miként lehetséges a dokumentációs karakterláncban lévő kód végrehajtása egységtesztek elvégzésekor.

A második, a `CharGrid.py` fájlban lévő `CharGrid` modul egy karakterekből álló mintát (rácsot) tárol, és lehetővé teszi, hogy vonalakat, négyszögeket és szöveget „rajzoljunk" ebbe a rácsba, majd mindezt a konzolon is megjelenítsük. A modul pár olyan módszert is bemutat, amellyel korábban még nem találkoztunk, és egészében véve is jellemzőbb példáját nyújtja a nagyszabású, összetett moduloknak.

A TextUtil modul

Ennek a modulnak – és a legtöbb más modulnak is – különbözik a felépítése a programoknál megszokottól. Az első sor tartalmazza a shebang-et, majd pár megjegyzés következik (rendszerint a szerzői jogi és a felhasználási engedélyekkel kapcsolatos információ). Ezeket

általában egy három idézőjelbe zárt karakterlánc követi, amely áttekintést nyújt a modul tartalmáról, gyakran a felhasználás módját bemutató példákkal – ez a rész a modulban lévő dokumentációs karakterlánc (docstring). Íme a `TextUtil.py` fájl eleje, a felhasználási engedélyhez tartozó megjegyzéseket elhagyva:

```
#!/usr/bin/env python3
# Copyright (c) 2008 Qtrac Ltd. All rights reserved.
"""
This module provides a few string manipulation functions.

>>> is_balanced("(Python (is (not (lisp))))")
True
>>> shorten("The Crossing", 10)
'The Cro...'
>>> simplify(" some   text   with spurious whitespace ")
'some text with spurious whitespace'
"""

import string
```

A modulban lévő dokumentációs karakterlánc akkor válik hozzáférhetővé a programok, illetve más modulok számára, ha azok a betöltést a `TextUtil.__doc__` formában végzik el. A modulban a docstring után következnek a betöltést végző sorok – esetünkben egyetlen egy –, és ezt a modul többi része követi.

A `shorten()` függvény már szerepelt a könyvben, így most nem ismételjük meg. És minthogy a függvények helyett ezúttal a modulokra összpontosítunk, a `simplify()` függvényt teljes egészében, a dokumentációs karakterlánccal együtt mutatjuk be, de az `is_balanced()` függvénynek már csak a kódot tartalmazó része fog szerepelni. Íme a `simplify()` függvény, két részletben:

```
def simplify(text, whitespace=string.whitespace, delete=""):
r"""Returns the text with multiple spaces reduced to single spaces

The whitespace parameter is a string of characters, each of which
is considered to be a space.
If delete is not empty it should be a string, in which case any
characters in the delete string are excluded from the resultant
string.

>>> simplify("   this and\n that\t too")
'this and that too'
>>> simplify(" Washington  D.C.\n")
'Washington D.C.'
>>> simplify(" Washington  D.C.\n", delete=",;:.")
'Washington DC'
>>> simplify(" disemvoweled ", delete="aeiou")
'dsmvwld'
"""
```

Nyers karakterláncok
☞ 61

A `def` utasítást tartalmazó sor után következik a függvény dokumentációs karakterlánca, amely a hagyomány szerint egy egysoros leírásból, egy ezt követő üres sorból és további leírásból, majd néhány példából áll. A példákat úgy adjuk meg, mintha interaktívan írtuk volna be őket. Minthogy az idézőjelben lévő karakterláncok a dokumentációs karakterláncon belül vannak, a bennük lévő fordított perjeleket vagy le kell védenünk, vagy ahogy mi is tettük, nyers, háromszoros idézőjelbe tett karakterláncot kell használnunk.

```
result = []
word = ""
for char in text:
    if char in delete:
        continue
    elif char in whitespace:
        if word:
            result.append(word)
            word = ""
    else:
        word += char
if word:
    result.append(word)
return " ".join(result)
```

A `result` nevű listában tároljuk a „szavakat", vagyis az olyan karakterláncokat, amelyek nem tartalmaznak szóközöket, illetve törlésre kijelölt karaktereket. Az átadott szöveget karakterenként járjuk be, a törlésre jelölt karaktereket átlépve. Ha szóközként megadott karakterbe botlunk, és éppen egy szó belsejében vagyunk, a szót hozzáadjuk az eredményeket tároló listához, majd a szót eddig tároló változót kiürítjük. Minden más karakter fellelésekor a karakter az éppen kiépülőben lévő szó végére kerül. A függvény végül egyetlen karakterláncot ad vissza, amelyben az eredményeket tartalmazó lista szavai szerepelnek, egy-egy szóközzel elválasztva.

Az `is_balanced()` függvény hasonló mintát követ: a `def` utasítást tartalmazó sort követi a docstring, benne az egysoros, illetve az üres sort követő bővebb leírás, néhány példa, majd maga a kód. Íme a dokumentációs karakterlánc nélküli kód:

```
def is_balanced(text, brackets="()[]{}<>"):
    counts = {}
    left_for_right = {}
    for left, right in zip(brackets[::2], brackets[1::2]):
        assert left != right, "the bracket characters must differ"
        counts[left] = 0
        left_for_right[right] = left
    for c in text:
        if c in counts:
            counts[c] += 1
        elif c in left_for_right:
            left = left_for_right[c]
```

```
        if counts[left] == 0:
            return False
        counts[left] -= 1
    return not any(counts.values())
```

A függvény két szótárat épít fel. A `counts` szótár kulcsai a nyitó zárójelek („(", „[", „{" és „<"), az értékei pedig egész számok. A `left_for_right` szótár kulcsai a záró zárójelek („)", „]", „}" és „>"), az értékei pedig a megfelelő nyitó zárójelek. Ha a szótárak elkészültek, a függvény karakterenként bejárja a szöveget. Amikor nyitó zárójelet találunk, a megfelelő mérőszámot eggyel növeljük. Ehhez hasonlóan, amikor záró zárójelhez érünk, a függvény megkeresi a megfelelő nyitó zárójelet. Ha pedig annak nulla a száma, az annyit tesz, hogy fölös számú záró zárójelre akadtunk, vagyis azonnal `False` értéket kapunk vissza. Ha az érték nullától különböző, akkor eggyel csökkentjük. A szöveg bejárásának végeztével elvileg minden mérőszám nulla értékű – már amennyiben ugyanannyi nyitó és záró zárójel van. Ha tehát bármelyik szám nullától eltérő értékű, a függvény `False` értéket ad vissza, és `True` a visszaadott érték, ha mindegyik nulla.

Eddig a pontig minden majdnem olyan volt, mint bármely Python-programban. Ha a `TextUtil.py` fájl programfájl lenne, alighanem lenne benne néhány további függvény, a fájl vége felé pedig egy önmagában álló függvényhívás, ami a feldolgozás megkezdését végezné. Ez a fájl azonban modul, vagyis azért készítettük, hogy más fájlokba lehessen betölteni, és ekképpen elég, ha csak megadjuk benne a függvényeket. Immár bármely program és bármely modul betöltheti és használhatja a `TextUtil` modult:

```
import TextUtil

text = " a     puzzling conundrum "
text = TextUtil.simplify(text) # text == 'a puzzling conundrum'
```

Ha azt szeretnénk, hogy egy bizonyos program használhassa a `TextUtil` modult, elég, ha a `TextUtil.py` fájlt a programmal azonos könyvtárban helyezzük el. Ha azt szeretnénk, hogy a `TextUtil.py` fájlban lévő modul minden programunk számára elérhető legyen, többféleképpen is megközelíthető a probléma. Az egyik lehetőség, hogy a modult elhelyezzük a Python `site-packages` alkönyvtárában – ez Windows rendszereken rendszerint a `C:\Python30\Lib\site-packages` könyvtár, Mac OS X és Unix rendszereken azonban különféle helyekre szokott kerülni. A könyvtár megtalálható a Python elérési útvonalán, vagyis az ide helyezett modulok mindig elérhetők. A második lehetőség, ha az olyan modulok számára, amelyeket minden programunkban használni szeretnénk, külön mappát hozunk létre, és a mappát beállítjuk a `PYTHONPATH` környezeti változóban. A harmadik lehetőség, ha a modul a *helyi* `site-packages` alkönyvtárba kerül; ez Windowson az `%APPDATA%/Python/Python30/site-packages` mappa, Unixon (a Mac OS X operációs rendszert is ideértve) pedig a `~/.local/lib/python3.0/site-packages` mappa. Ez a mappa is benne van a Python elérési útvonalait tartalmazó listában. A második és harmadik módszernek megvan az az előnye, hogy a saját kódunkat a hivatalos telepítéstől elkülönítve tároljuk.

Az, hogy elkészült a `TextUtil` modul, igazán remek dolog, de ha sok programban szeretnénk használni a modult, talán pontosabban meg kellene bizonyosodnunk a helyes működéséről. Ennek egy igazán egyszerű módja, ha a dokumentációs karakterláncban lévő példákat végrehajtjuk, és megbizonyosodunk róla, hogy a várt eredményt kapjuk. Mindez elvégezhető, ha a modult tartalmazó, `.py` kiterjesztésű fájl végén elhelyezünk három sort:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Ha egy Python-programba betöltünk egy modult, a Python a modul számára létrehoz egy `__name__` nevű változót, és a modul nevét elhelyezi ebben a változóban. A modul neve megegyezik az őt tartalmazó fájl nevével, mindössze a `.py` kiterjesztést hagyjuk el. Így a példánkban a modul betöltésének megtörténtével a `__name__` változó értéke `TextUtil` lesz. Így az `if` utasításnál megadott feltétel nem teljesül, vagyis a két utolsó sor végrehajtása elmarad. Ez annyit tesz, hogy a modul betöltésekor az utolsó három sor gyakorlatilag nem jelent költséget.

Amikor egy `.py` kiterjesztésű fájlt futtatunk, a Python a program számára létrehoz egy `__name__` nevű változót, amely a `"__main__"` karakterláncot kapja értékül. Vagyis, ha a `TextUtil.py` állományt *futtatjuk*, mintha program lenne, a Python a `__name__` nevű változó értékéül beállítja a `"__main__"` karakterláncot, az `if` utasítás feltételének kiértékelésekor pedig `True` értéket kapunk, és az utolsó két sort végrehajtjuk.

A `doctest.testmod()` függvény a Python önellenőrző képességeit kihasználva megkeresi a modulban lévő valamennyi függvényt, illetve a függvényekhez tartozó dokumentációs karakterláncokat, és megpróbálja végrehajtani az összes olyan kódrészletet, amit azokon belül talál. Ha ezen a módon futtatjuk a modult, csak akkor kapunk kimenetet, ha a futás során hiba merül fel. Ez első közelítésben zavaró lehet, mivel úgy tűnik, mintha mi sem történne, de ha a `-v` parancssori kapcsolót is használjuk, az alábbihoz hasonló kimenetet kapunk:

```
Trying:
    is_balanced("(Python (is (not (lisp))))")
Expecting:
    True
ok
...
Trying:
    simplify(" disemvoweled ", delete="aeiou")
Expecting:
    'dsmvwld'
ok
4 items passed all tests:
    3 tests in __main__
    5 tests in __main__.is_balanced
    3 tests in __main__.shorten
```

```
    4 tests in __main__.simplify
15 tests in 4 items.
15 passed and 0 failed.
Test passed.
```

Ahol kihagytunk sorokat, ott ezt három ponttal jeleztük. Ha a `-v` kapcsolót használjuk, azoknak a függvényeknek (osztályoknak, tagfüggvényeknek) a listáját is megkapjuk, amelyekhez nem adtunk meg tesztet. Figyeljük meg, hogy a `doctest` modul a modul és az egyes függvények dokumentációs karakterláncaiban lévő teszteket egyaránt megtalálja.

A dokumentációs karakterláncokban lévő végrehajtható példák neve *doctest*. Vegyük észre, hogy ha doctest-et írunk, sem a `simplify()` függvényt, sem az egyéb függvényeket nem kell a teljes nevükkel hívnunk, hiszen a tesztek magában a modul belsejében kapnak helyet. A modulon kívül – feltételezve, hogy a modult betöltöttük – a függvények teljes nevét kell használnunk, például: `TextUtil.is_balanced()`.

A következő részben megmutatjuk, hogy miként végezhetünk alaposabb tesztelést, különös tekintettel az olyan esetekre, amikor hibákra számítunk – például érvénytelen adatok okozta kivételekre. Emellett áttekintünk pár egyéb, a modulok létrehozása során felmerülő kérdést. Ilyen például a modulok előkészítése, az operációs rendszerek okozta különbségek figyelembe vétele, és annak kezelése, hogy a `from` *`modul`* `import *` utasításforma használata esetén csak azok az objektumok kerüljenek át a betöltő programba, amelyeket kívülről elérhetővé akarunk tenni.

A CharGrid modul

A `CharGrid` modul a memóriában karakterekből álló rácsot alakít ki. Arra való, hogy a rácsra karakterekből álló vonalakat, négyszögeket és szöveget „rajzoljunk", és ezt a konzolon megjelenítsük. Íme a modul dokumentációs karakterláncában lévő dokumentációs tesztek (doctest):

```
>>> resize(14, 50)
>>> add_rectangle(0, 0, *get_size())
>>> add_vertical_line(5, 10, 13)
>>> add_vertical_line(2, 9, 12, "!")
>>> add_horizontal_line(3, 10, 20, "+")
>>> add_rectangle(0, 0, 5, 5, "%")
>>> add_rectangle(5, 7, 12, 40, "#", True)
>>> add_rectangle(7, 9, 10, 38, " ")
>>> add_text(8, 10, "This is the CharGrid module")
>>> add_text(1, 32, "Pleasantville", "@")
>>> add_rectangle(6, 42, 11, 46, fill=True)
>>> render(False)
```

A `CharGrid.add_rectangle()` függvény legalább négy paramétert vár: a bal felső sarok és a jobb alsó sarok sorának és oszlopának számait. A körvonal megrajzolásához használni kívánt karakter lehet az ötödik paraméter, a hatodikként átadott logikai érték pedig azt szabályozza, hogy a körvonalat ki kell-e tölteni (a körvonalhoz használt karakterrel).

Amikor a függvényt először hívjuk, a harmadik és a negyedik paramétert egy kételemű rendezett sorozat (szélesség, magasság) kicsomagolásával adjuk át. A rendezett sorozatot a `CharGrid.get_size()` függvény visszatérési értékeként kapjuk meg.

A `CharGrid.render()` függvény alapértelmezés szerint a rács kiírása előtt kiüríti a képernyőt, de ez meggátolható a `False` paraméter átadásával – mi is ezt tettük. Íme az előző dokumentációs tesztek eredményéből származó minta:

```
%%%%%*************************************************
%   %                                @@@@@@@@@@@@@@@  *
%   %                                @Pleasantville@  *
%   %                     ++++++++++ @@@@@@@@@@@@@@@  *
%%%%%                                                 *
*       ##################################          *
*       ##################################    ****  *
*       ##                              ##    ****  *
*       ## This is the CharGrid module ##    ****  *
* !     ##                              ##    ****  *
* ! |   ##################################    ****  *
* ! |   ##################################          *
*   |                                               *
*************************************************
```

A modul a `TextUtil` modulnál megismert módon kezdődik: shebang-sor, szerzői jogi és a felhasználási engedélyekkel kapcsolatos megjegyzések, majd a modulszintű dokumentációs karakterlánc, amely leírja a modul működését, és tartalmazza a korábban bemutatott dokumentációs teszteket. Ezt követi maga a kód, két, `import` utasítást tartalmazó sorral, amelyek közül az egyik a `sys`, a másik a `subprocess` modult tölti be. A `subprocess` modulról a 9. fejezetben szólunk részletesebben.

A modulon belül két hibakezelési házirend érvényesül. Számos függvénynek van `char` típusú paramétere, aminek valójában mindig egy olyan karakterláncnak kell lennie, amely pontosan egy karaktert tartalmaz. Ha ez a kívánalom sérül, azt végzetes programhibának tekintjük, így `assert` utasítást tartalmazó kifejezésekkel ellenőrizzük a karakterlánc hosszát. Ha túl magas értékeket kapunk sor- és oszlopszámokként, azt olyan hibának tekintjük, ami előfordulhat, vagyis ha ilyen esemény történik, egyedi kivételeket hozunk létre.

Az alábbiakban a kód néhány jellemző, kulcsszerepű részét tekintjük át, kezdve az egyedi kivételeknél:

```
class RangeError(Exception): pass
class RowRangeError(RangeError): pass
class ColumnRangeError(RangeError): pass
```

A modulban lévő, kivételeket létrehozni képes függvények egyike sem fog `RangeError` kivételt adni, mindig `RowRangeError` vagy `ColumnRangeError` jön létre, attól függően, hogy a sor vagy az oszlop száma esik kívül a megadható tartományon. Így azonban, hogy

a kivételeket hierarchiába szerveztük, a modul felhasználójának a kezébe adjuk a választás lehetőségét: vagy a pontosan a hibának megfelelő kivételt fogja el, vagy mindkettőt egyszerre, ha a `RangeError` alaposztályba tartozó kivételeket figyeli. Figyeljük meg, hogy bár a dokumentációs tesztekben a kivételek nevét úgy használjuk, ahogy az előző sorokban is írtuk, a `CharGrid` modult betöltő programban a kivételek neve természetesen módosul: a `CharGrid.RangeError`, a `CharGrid.RowRangeError`, és a `CharGrid.ColumnRangeError` neveket kell használnunk.

```
_CHAR_ASSERT_TEMPLATE = ("char must be a single character: '{0}' "
                         "is too long")
_max_rows = 25
_max_columns = 80
_grid = []
_background_char = " "
```

A fenti sorokban néhány privát változónak adtunk értéket; ezeket csak a modul használja. A változók nevei aláhúzással kezdődnek, így amennyiben a modul betöltését a `from CharGrid import *` utasítással végezzük, ezek a változók kimaradnak a betöltésből. (A probléma másik megközelítési módja az lenne, ha értéket adnánk az `__all__` nevű listának.) A `_CHAR_ASSERT_TEMPLATE` egy karakterlánc, amelyet a `str.format()` függvény használ. A modulban az a szerepe, hogy az `assert` utasítások hibaüzenetét tárolja. A többi változót az első előfordulásuk helyén tárgyaljuk.

```
if sys.platform.startswith("win"):
    def clear_screen():
        subprocess.call(["cmd.exe", "/C", "cls"])
else:
    def clear_screen():
        subprocess.call(["clear"])
clear_screen.__doc__ = """Clears the screen using the underlying \
window system's clear screen command"""
```

A konzolképernyő kiürítésének módja az operációs rendszertől függ. Windowson a `cmd.exe` programot kell a megfelelő paraméterekkel futtatnunk, a legtöbb Unix rendszeren pedig a `clear` programot futtatjuk. A subprocess modul `subprocess.call()` függvénye lehetővé teszi egy külső program futtatását, így használható arra, hogy az operációs rendszer szempontjából megfelelő módszerrel ürítsük ki a képernyőt. A `sys.platform` karakterlánc tárolja a programot futtató operációs rendszer nevét – például: win32 vagy linux2. Így az operációs rendszerek eltéréséből eredő különbségek kezelésének egyik módja az lenne, ha megadnánk egy az alábbihoz hasonló `clear_screen()` függvényt:

```
def clear_screen():
    command = (["clear"] if not sys.platform.startswith("win") else
               ["cmd.exe", "/C", "cls"])
    subprocess.call(command)
```

A megközelítés hátránya, hogy bár tisztában vagyunk azzal, hogy a program futása alatt nem változhat az operációs rendszer, mégis minden függvényhívás alkalmával lefut az ellenőrzés.

Ha el akarjuk kerülni azt, hogy a `clear_screen()` függvény minden egyes hívásakor megtörténjen a programot futtató operációs rendszer ellenőrzése, a modul betöltésekor létrehozunk egy az operációs rendszernek megfelelő `clear_screen()` függvényt, és onnantól fogva mindig azt használjuk. Mindez azért lehetséges, mert a `def` utasítás is olyan, mint a Python nyelv többi utasítása: amikor a parancsértelmező eléri az `if` utasítást, vagy az első, vagy a második `def` utasítást hajtja végre, *dinamikusan* hozva létre vagy az egyik, vagy a másik `clear_screen()` függvényt. Minthogy a függvényt nem egy másik függvény belsejében (vagy másik osztály belsejében, ahogy a következő fejezetben látni fogjuk) hozzuk létre, a függvényünk globális, vagyis éppen úgy érhető el, mint a modulban lévő többi függvény.

A függvény létrehozása után pontos utasítással adjuk meg a hozzá tartozó dokumentációs karakterláncot, ezzel elkerüljük ugyanannak a docstring-nek a két helyen való megadását, és megmutatjuk azt is, hogy a dokumentációs karakterlánc nem egyéb, mint a függvény jellemzőinek egyike. A jellemzők között megtalálható a függvény modulja és a neve.

```
def resize(max_rows, max_columns, char=None):
    """Changes the size of the grid, wiping out the contents and
    changing the background if the background char is not None
    """
    assert max_rows > 0 and max_columns > 0, "too small"
    global _grid, _max_rows, _max_columns, _background_char
    if char is not None:
        assert len(char) == 1, _CHAR_ASSERT_TEMPLATE.format(char)
        _background_char = char
        _max_rows = max_rows
        _max_columns = max_columns
        _grid = [[_background_char for column in range(_max_columns)]
                 for row in range(_max_rows)]
```

Ez a függvény egy `assert` utasítással juttatja érvényre a kódolási hibává nyilvánítást, amikor a rácsot 1x1-es méretnél kisebbre próbáljuk állítani. Ha a háttérül szolgáló karakter is meg van adva, szintén egy `assert` utasítással ellenőrizzük, hogy pontosan egy karakterről van-e szó. Ha nem, akkor a művelet hibaüzenete a `_CHAR_ASSERT_TEMPLATE` karakterláncban tárolt szöveg lesz, a `{0}` helyére az átadott karakterlánc kerül.

Sajnos szükség van a `global` utasítás használatára, hiszen a függvény belsejében számos globális változót kell frissítenünk. Ez olyasmi, amit objektumközpontú megközelítéssel elkerülhetünk, ahogy azt a 6. fejezetben látni fogjuk.

Listaértelmezések 113

A `_grid` lista egy listaértelmezés belsejében lévő listaértelmezéssel jön létre. A `[[char] * columns] * rows` alakú listamásolási művelet nem vezet eredményre, hiszen ez csak a belső lista megosztását (sekély másolását) végzi el. Használhattunk volna helyette egymásba ágyazott `for...in` ciklusokat:

```
_grid = []
for row in range(_max_rows):
    _grid.append([])
    for column in range(_max_columns):
        _grid[-1].append(_background_char)
```

A fenti kódot viszonylag nehezebb megérteni, mint a listaértelmezést, ugyanakkor lényegesen hosszabb is nála.

Elsődleges célunk a modulok megvalósításának megértése, ezért a rajzolást végző függvények közül csak egyet tekintünk át, és éppen csak belekóstolunk a rajzolás mikéntjébe. Íme az `add_horizontal_line()` függvény, két részletben közölve:

```
def add_horizontal_line(row, column0, column1, char="-"):
    """Adds a horizontal line to the grid using the given char

    >>> add_horizontal_line(8, 20, 25, "=")
    >>> char_at(8, 20) == char_at(8, 24) == "="
    True
    >>> add_horizontal_line(31, 11, 12)
    Traceback (most recent call last):
    ...
    RowRangeError
    """
```

A dokumentációs karakterláncokban két teszt található: az egyiktől azt várjuk, hogy működjön, a másiktól pedig azt, hogy hozzon létre kivételt. Ha a dokumentációs tesztekben lévő kivételekkel bajlódunk, a módszer az, hogy a „Traceback" kezdetű sort adjuk meg – ez ugyanis mindig ugyanaz. Így hozhatjuk tudomására a `doctest` modulnak, hogy kivételre számítunk. Ezek után három ponttal jelezzük a közbeeső sorokat, amelyek mindig mások lehetnek, végül pedig a várt kivétel nevét tartalmazó sor következik. A `char_at()` függvény egyike a modulban szereplő függvényeknek, amely a rács adott sorában és oszlopában álló karaktert adja vissza.

```
assert len(char) == 1, _CHAR_ASSERT_TEMPLATE.format(char)
try:
    for column in range(column0, column1):
        _grid[row][column] = char
except IndexError:
    if not 0 <= row <= _max_rows:
        raise RowRangeError()
    raise ColumnRangeError()
```

A kód ugyanazzal a karakterlánc hosszúságát ellenőrző résszel kezdődik, amelyet a `resize()` függvényben is használtunk. A sor- és az oszlopszámot hordozó paraméter külön ellenőrzése helyett a függvény feltételezi, hogy a paraméterek érvényesek. Ha nem létező sort vagy oszlopot próbálunk elérni, `IndexError` kivétel lép fel, amit elkapunk, és helyette egy a modulra jellemző kivételt hozunk létre. Az efféle programozási stílust szokás „könnyebb elnézést kérni, mint engedélyt" stílusnak nevezni, és általában *„pythonosabb"* módszernek számít, mint a „lassan járj, tovább érsz" módszer, azaz az ellenőrzések előzetes elvégzése. Ha kivételes esetek ritkán fordulnak elő, hatékonyabb a létrejövő kivételekre támaszkodni, mint előre ellenőrizni a dolgokat. (Az `assert` utasításokat tartalmazó kifejezések használata másféle elbírálás alá esik, mivel jó esetben velük soha nem történik meg a baj, és a kész kódban gyakran megjegyzéssé is alakítják őket.)

A modulnak majdnem a legvégén – miután valamennyi függvény megadása megtörtént –, egyetlen egyszer hívjuk meg a `resize()` függvényt:

```
resize(_max_rows, _max_columns)
```

Ez a hívás előkészíti az alapértelmezett (25x80) méretű rácsot, és figyel arra is, hogy a kód, amely a modult betölti, azonnal használatba is vehesse. A hívás nélkül a betöltő programnak, illetve modulnak mindig meg kellene hívnia a `resize()` függvényt a rács előkészítéséhez, valahányszor a modult betöltjük. Erre a programozóknak mindig emlékezniük kellene, és a dolog többszörös előkészítéshez is vezethet.

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

A modul három utolsó sora a szokásos három sor azokban az esetekben, amikor a `doctest` modullal ellenőrizzük a modul dokumentációs tesztjeit.

A `CharGrid` modulnak van egy fontos képességbeli hiányossága: csak egyetlen karakteres rács alakítható ki vele. A probléma egyik megoldását az jelentené, ha a modulban rácsok gyűjteményét tárolnánk, de ez azzal járna, hogy a modul használóinak a függvény minden hívásakor meg kellene adniuk a rács azonosítására szolgáló sorszámot vagy kulcsot. Olyan esetekben, amikor egy objektumból több példányra is szükség van, jobb megoldást jelent olyan modult készíteni, amely megad egy objektumosztályt (egy egyedi adattípust), ugyanis az objektumosztályt annyiszor példányosítjuk (annyi, az adott típusba tartozó objektumot hozunk létre), ahányszor csak kedvünk tartja. Az osztály létrehozásának további előnye lenne, hogy elkerülhetnénk a `global` utasítás használatát az osztályszintű (statikus) adatok tárolásakor. A következő fejezetben meg is ismerkedünk az osztályok létrehozásának módjával.

A Python alapkönyvtárának áttekintése

A Python alapkönyvtárárárról szólva gyakran mondják, hogy minden benne van. És tényleg: a majdnem kétszáz csomag és modul a felmerülő problémák széles körére kínál kész megoldásokat. Ebben a részben tematikus megközelítésben vesszük sorra, hogy mi mindenre találunk megoldást, kihagyva a túlzottan speciális problémák kezelésére való, illetve a csak egyetlen operációs rendszeren futó csomagokat és modulokat. Sok esetben rövid példával adunk ízelítőt az adott csomag vagy modul használatából, a könyv más részein tárgyalt csomagokra és modulokra pedig a megfelelő helyen hivatkozunk.

Karakterláncok kezelése

A `string` modulban megtalálható néhány olyan hasznos állandó, mint a `string.ascii_letters` (ASCII-betűk) és a `string.hexdigits` (a tizenhatos – hexadecimális – számrendszerben használt számok). Itt találjuk a `string.Formatter` osztályt, amelyből alosztályokat képezve egyedi formázókat hozhatunk létre.* A `textwrap` modul arra használható, hogy egy szöveg sorait adott szélességűvé alakítsa, és a behúzásokat csökkentse.

A `struct` modulban számok, logikai értékek és karakterláncok `bytes` osztályú objektumokba való csomagolására és a belőlük való kicsomagolására szolgáló, a kettes számrendszerű (bináris) ábrázolásukat használó függvényeket találunk. Ez akkor lehet hasznos, amikor alacsony szintű, C nyelven írt könyvtárakkal cserélünk adatokat. A 7. fejezetben tárgyalt `convert-incidents.py` program a `struct` és a `textwrap` modult egyaránt használja.

`bytes` típus ☞ 297

`struct` modul ☞ 296

A `difflib` modulban olyan osztályokat és tagfüggvényeket találunk, amelyekkel sorozatok – például karakterláncok – összehasonlítása végezhető el, a kimenetet pedig a `diff` parancs megszokott formájában vagy akár HTML-ben is kérhetjük. A Python legügyesebb karakterlánc-kezelő modulja az `re` (regular expression) modul. Erről a modulról a 12. fejezet szól.

Az `io.StringIO` osztályból olyan, karakterlánchoz hasonló objektumot példányosíthatunk, amely a memóriában tárolt szövegfájlként viselkedik. Ez igen kényelmes lehet, ha a fájlok írására használatos kódot szeretnénk karakterlánc írására használni.

Példa: Az io.StringIO osztály

A Python nyelv két különböző módon teszi lehetővé szöveg fájlba írását. Az egyik módszer a fájlt ábrázoló objektum `write()` tagfüggvényének használata, a másik pedig az, ha a `print()` függvényt használjuk, és a `file` kulcsszó-paraméter értékéül az írásra megnyitott fájlt adjuk meg. Például:

---

* Az *alosztályképzés* vagy a *specializáció* szakkifejezést arra használjuk, amikor egy egyedi adattípust – vagyis egy osztályt – egy másik objektumosztály alapján hozunk létre. A 6. fejezet teljes körűen tárgyalja a témát.

```
print("An error message", file=sys.stdout)
sys.stdout.write("Another error message\n")
```

Mindkét kiírt szövegsor a `sys.stdout` objektumra kerül, ami egy fájlobjektum, és a „szabványos kimeneti csatornának" felel meg. Ez rendes esetben a konzol, és annyiban különbözik a „szabványos hibacsatornától", hogy ez utóbbi nem rendelkezik átmeneti tárral. (A Python a programok elindulásakor automatikusan létrehozza a `sys.stdin` (szabványos bemenet), a `sys.stdout`, és a `sys.stderr` csatornát.) A `print()` függvény alapértelmezés szerint a sor végén sortörést is elhelyez, bár ez meggátolható, ha átadjuk neki az üres karakterlánc értékűre állított `end` kulcsszó-paramétert.

Vannak helyzetek, amikor hasznos lehet, ha a fájlba irányított kimenetet egy karakterlánccal elkapjuk. Ez a `io.StringIO` osztály segítségével tehető meg, amelyből a fájlobjektumokkal azonos módon használható, de a beleírt adatot karakterláncban tároló objektum példányosítható. Ha a `io.StringIO` osztályú objektumnak átadunk egy kezdeti karakterláncot, akkor az is úgy olvasható vissza, mintha egy fájlba került volna. A `io.StringIO` osztály az `import io` parancs kiadása után érhető el, és használható például a `sys.stdout` fájlobjektumnak átadott kimenet elfogására:

```
sys.stdout = io.StringIO()
```

Ha a fenti sort elhelyezzük egy program elején, a betöltések után, de még mielőtt a `sys.stdout` objektumot használnánk, minden, a `sys.stdout` objektumba küldött szöveg valójában a fenti sorral létrehozott és az alapértelmezett `sys.stdout` objektumot lecserélő `io.StringIO` osztályú, fájlként viselkedő objektumba kerül. Ha a korábban bemutatott, a `print()` és a `sys.stdout.write()` utasítást tartalmazó sorokat ekkor hajtjuk végre, a kimenetük a konzol helyett az `io.StringIO` osztályú objektumba kerül. (Az eredeti `sys.stdout` objektum a `sys.stdout = sys.__stdout__` utasítással visszaállítható.)

Az `io.StringIO` osztályú objektumba került valamennyi karakterlánchoz hozzáférhetünk, ha az `io.StringIO.getvalue()` tagfüggvényt hívjuk, esetünkben a `sys.stdout.getvalue()` formában. A tagfüggvény visszatérési értéke a beírt összes sort tartalmazó karakterlánc. Ezt a karakterláncot kiírhatjuk a képernyőre, menthetjük egy naplófájlba, vagy továbbíthatjuk a hálózaton – bármely más karakterlánchoz hasonlóan. Az `io.StringIO` osztály használatára a fejezetben kicsit később újabb példát is látunk majd.

Parancssori programozás

Ha olyan programra van szükségünk, amely a konzolon átirányított vagy a parancssorban felsorolt fájlokban lévő szöveg feldolgozására képes, érdemes lehet a `fileinput` modul `fileinput.input()` függvényét használnunk. Ez a függvény bejárja a konzolról átirányított sorokat – ha vannak ilyenek –, illetve a parancssorban felsorolt fájlokat, méghozzá sorok folyamatos sorozataként. A modul a `fileinput.filename()` és a `fileinput.lineno()` függvényekkel képes bármely pillanatban megadni az aktuális fájlnevet és sort, és képes néhány tömörített fájlformátum kezelésére is.

A parancssori paraméterek kezelésére két külön modul áll rendelkezésre: az `optparse` és a `getopt`. A `getopt` modul azért népszerű, mert egyszerű, és már régóta része az alapkönyvtárnak. Az `optparse` modul újabb és sokoldalúbb.

Példa: Az optparse modul

A `csv2html.py` példaprogram
☞ 91

Még a második fejezetben tárgyaltuk a `csv2html.py` programot. A fejezet gyakorlatai között ajánlottuk, hogy a programot tegyük képessé a `maxwidth` (maximális szélesség) és a `format` (formátum) paraméterek kezelésére; az előbbi egy egész szám, az utóbbi egy karakterlánc. A megadott megoldás (`csv2html2_ans.py`) 26 sorban dolgozza fel a paramétereket. Íme a `csv2html2_opt.py` program `main()` függvényének eleje, amelyben a parancssori paraméterek feldolgozására saját függvény helyett az `optparse` modult használjuk:

```
def main():
    parser = optparse.OptionParser()
    parser.add_option("-w", "--maxwidth", dest="maxwidth", type="int",
        help=("the maximum number of characters that can be "
              "output to string fields [default: %default]"))
    parser.add_option("-f", "--format", dest="format",
            help=("the format used for outputting numbers "
                  "[default: %default]"))
    parser.set_defaults(maxwidth=100, format=".0f")
    opts, args = parser.parse_args()
```

Mindössze kilenc sornyi kódra volt szükség, meg persze az `import optparse` kifejezésre. Mi több, a továbbiakban nincs szükség a `-h` és a `-help` kapcsoló külön megadására. Ezeket mostantól az `optparse` modul kezeli, megfelelő használati utasítást készítve a `help` kulcsszó-paraméterekből, a `%default` szövegeket a kapcsoló alapértelmezett értékével helyettesítve.

Figyeljük meg azt is, hogy a kapcsolók immár a hagyományos Unix-stílust követik: rövid és hosszú, kötőjelekkel kezdődő nevekkel rendelkeznek. A rövid nevek a konzolban végrehajtott interaktív műveletek során kényelmesek, a hosszú nevek pedig érthetőbbek, ha héjprogramokban használjuk őket. Például a maximális szélesség értékének beállításához egyaránt használható a `-w80`, a `-w 80`, a `–width=80` és a `–width 80` forma. Miután a parancssort feldolgoztuk, a paraméterek a `dest` paranccsal megadott neveken érhetők el, például `opts.maxwidth` és `opts.format` néven. Azok a parancssori paraméterek, amelyeknek a feldolgozása nem történt meg (ezek rendszerint fájlnevek), az `args` listába kerülnek.

Ha a parancssor feldolgozása során valami nem sikerül, az `optparse` modul elvégzi a `sys.exit(2)` függvényhívást. Ez a program tiszta megszakításához vezet, az operációs rendszer pedig a programtól a 2-es visszatérési értéket kapja. A 2-es visszatérési érték hagyományosan felhasználói hibát jelez, az 1 egyszerűen csak hibát, a 0 pedig sikeres futást. Ha a `sys.exit()` függvényt paraméterek nélkül hívjuk meg, 0-át ad vissza az operációs rendszernek.

Matematika és számok

A beépített `int` (egész), `float` (lebegőpontos) és `complex` (komplex) számokat tároló típusokon felül az alapkönyvtár lehetőséget nyújt a tizedestörtek és az emeletes törtek kezelésére is (`decimal.Decimal`, `fractions.Fraction`). Három számfeldolgozó könyvtár áll a rendelkezésünkre: a `math` a racionális számokkal dolgozó matematikai függvényeket, a `cmath` a komplex számokat használó függvényeket tartalmazza, a `random` pedig a véletlenszámok előállítására használható függvények sokaságával szolgál. Ezeket a modulokat a második fejezetben ismertük meg.

A Python nyelv számfeldolgozó elvont alaposztályai (olyan objektumosztályok, amelyekből származtatható osztály, de a közvetlen példányosításuk nem lehetséges) a `numbers` modulban találhatók. Ezek akkor hasznosak, ha egy objektumról, mondjuk *x*-ről, el akarjuk dönteni, hogy valamilyen szám-e: `isinstance(x, numbers.Number)`, vagy valamilyen adott típusú szám-e: `isinstance(x, numbers.Rational)`, `sinstance(x, numbers.Integral)`.

Azok, akik a tudományos vagy mérnöki számítások elvégzésében érdekeltek, hasznosnak fogják találni a `NumPy` nevű, külső fejlesztőktől származó csomagot. Ebben igen hatékony n-dimenziós tömböket, alapvető lineáris algebrai függvényeket, Fourier-transzformációkat, illetve a C, C++ és Fortran nyelvekkel való együttműködést segítő eszközöket találunk. A `SciPy` csomagnak is része a `NumPy` csomag, és ezen felül találhatók benne statisztikai számításokat, jel- és képfeldolgozást segítő eszközök, genetikai algoritmusok, és sok minden más. Mindkét csomag ingyenesen elérhető a `www.scipy.org` webhelyen.

Dátumok és időpontok

A `calendar` és a `datetime` modul az idő- és a dátumkezelés megkönnyítésére szolgáló osztályokat és függvényeket tartalmaz. Mindkét modul a javított Gergely-naptáron alapul, a korábbi dátumok kezelésére alkalmatlanok. Az idő- és a dátumkezelés meglehetősen összetett probléma: a használatban lévő naptárak eltérő helyeken, eltérő időpontokban mások és mások, egy nap nem pontosan 24 óra, egy év nem pontosan 365 nap, és még akkor nem is beszéltünk a téli-nyári időszámításból és az időzónák létéből eredő problémákról. A `datetime.datetime` osztályt felkészítették az időzónák kezelésére (nem így a `datetime.date` osztályt), de ezt nem végzi el automatikusan. Külső fejlesztőktől származó modulok próbálják orvosolni a problémát, például a `www.labix.org/python-dateutil` webhelyről származó `dateutil`, illetve a `www.egenix.com/products/python/mxBase/mxDateTime` webhelyről származó `mxDateTime` modul.

A `time` modul az időbélyegeket kezeli. Az időbélyegek egyszerű számok, amelyek a Unix rendszerek időszámításának kezdetétől, vagyis 1970-01-01 00:00:00-tól kezdve eltelt másodperceket jelentik. Ez a modul használható arra, hogy megtudjuk a számítógép aktuális, UTC (koordinált világidő) szerinti időbélyegét, illetve a helyi, téli-nyári időszámítást figyelembe vevő időt. A modul képes továbbá különbözőképpen formázott dátumok és időpontok megjelenítésére, valamint olyan karakterláncok feldolgozására, amelyek dátumokat és időpontokat tárolnak.

Példa: A calendar, a datetime és a time modul

A `datetime.datetime` osztály objektumait rendszerint programból állítjuk elő, ugyanakkor azok az objektumok, amelyek UTC szerinti dátumokat és időpontokat tárolnak, rendszerint külső forrásból, például fájlok időbélyegéből érkeznek. Íme néhány példa:

```
import calendar, datetime, time
moon_datetime_a = datetime.datetime(1969, 7, 20, 20, 17, 40)
moon_time = calendar.timegm(moon_datetime_a.utctimetuple())
moon_datetime_b = datetime.datetime.utcfromtimestamp(moon_time)
moon_datetime_a.isoformat()     # visszaadott időpont:
                                # '1969-07-20T20:17:40'
moon_datetime_b.isoformat()     # visszaadott időpont:
                                # '1969-07-20T20:17:40'
time.strftime("%Y-%m-%dT%H:%M:%S", time.gmtime(moon_time))
```

A `moon_datetime_a` változó a `datetime.datetime` típusba tartozik, és azt az időpontot tárolja, amikor az Apollo 11 leszállt a Holdon. A `moon_time` változó `int` típusú, és a holdraszállástól számított másodperceket tárolja – a számot a `calendar.timegm()` függvénytől nyeri, amely a `datetime.datetime.utctimetuple()` függvény által visszaadott `time_struct` objektumot kapja paraméterül, és a benne ábrázolt másodpercek számát adja vissza. (Minthogy a holdraszállás a Unix időszámítás kezdete előtt történt, a szám negatív.) A `moon_datetime_b` változó is a `datetime.datetime` típusba tartozik, és a `moon_time` egész típusú változóból hozzuk létre, bemutatva az időszámítás kezdőpontja óta eltelt másodpercek átalakítását `datetime.datetime` osztályú objektummá.* Az utolsó három sor ugyanazt az ISO 8601 szabványban leírt formátumú dátumot, illetve időt adja vissza.

Az aktuális, UTC szerinti dátum és idő megkapható `datetime.datetime` osztályú objektumként, ha a `datetime.datetime.utcnow()` függvényt, illetve az időszámítás kezdőpontjától eltelt másodpercekben kifejezve, ha a `time.time()` függvényt hívjuk. A helyi idő kinyeréséhez a `datetime.datetime.now()`, illetve a `time.mktime(time.localtime())` függvények valamelyikét érdemes használnunk.

Algoritmusok és gyűjteményes adattípusok

A `bisect` modulban olyan függvények találhatók, amelyek rendezett sorozatokban, például rendezett listákban tudnak keresni, illetve képesek a rendezettség megtartása mellett új elemek beszúrására. A `modul` függvényei a bináris keresés algoritmusát használják, így nagyon gyorsak. A `heapq` modulban sorozatok, például listák kupacokká alakításához találunk függvényeket (a kupac – `heap` – olyan gyűjteményes adattípus, ahol az első, vagyis 0. sorszámú elem mindig a legkisebb), valamint a kupacforma megtartása mellett történő elembeszúráshoz és törléshez.

---

* A Windows-felhasználókat érinti, hogy a `datetime.datetime.utcfromtimestamp()` függvény képtelen a negatív, vagyis az 1970. január 1. előtti időbélyegek kezelésére.

Alapértelmező szótár 179 ☜

Nevesített rendezett sorozat 105 ☜

A `collections` csomagban kapott helyet a korábban már tárgyalt `collections.defaultdict` (alapértelmező szótár) és a `collections.namedtuple` (nevesített rendezett sorozat) gyűjteményes adattípus. A csomagban ezen felül megtaláljuk a `collections.UserList` és a `collections.UserDict` nevű típusokat, bár talán gyakrabban használják a beépített `list` és `dict` osztályokból származtatott osztályokat, mint a fenti típusokat. Szintén itt találjuk a listához hasonló `collections.deque` (kétvégű sor) típust, amely a listához hasonló, de míg a listának csak a végén tudunk gyorsan új elemet beszúrni és eltávolítani, a kétvégű sornál ezek a műveletek a sor elején és végén is nagyon gyorsan működnek.

A Python nem számokkal foglalkozó elvont alaposztályai (olyan osztályok, amelyekből származtatható osztály, de a közvetlen példányosításuk nem lehetséges) szintén a `collections` csomagban találhatók. Ezeket a 8. fejezetben tárgyaljuk.

Az `array` (tömb) modulba került az `array.array` sorozattípus, amely a számokat és a karaktereket igen helyhatékony módon tárolja. A `list` típushoz hasonló a viselkedése, azzal a kivétellel, hogy a létrehozásakor eldől, hogy mely osztályba tartozó objektumok tárolására képes, vagyis a listáktól eltérően csak egyetlen típusba tartozó objektumokat tárolhat. A már korábban említett, külső fejlesztésű `NumPy` csomag szintén hatékony tömböket tartalmaz.

Programunkat a `weakref` modul tudja felruházni a gyenge hivatkozások létrehozásának képességével. A gyenge hivatkozások a rendes objektumhivatkozásokhoz hasonlóan működnek, kivéve, hogy amennyiben az objektumra már csak gyenge hivatkozás mutat, akkor a szemétgyűjtő felszabadíthatja az objektum által elfoglalt memóriát. Ezen a módon megakadályozható, hogy egy objektum csak azért maradjon a memóriában, mert mutat rá objektumhivatkozás. Természetesen mielőtt megpróbálnánk elérni, ellenőrizhető, hogy a gyenge hivatkozás által hivatkozott objektum létezik-e még.

Példa: A heapq modul

A `heapq` modulban találjuk azokat a függvényeket, amelyekkel egy listát kupaccá alakíthatunk, illetve a kupachoz új elemet adhatunk, vagy kivehetjük belőle a már ott lévőket – mindezt a kupactulajdonság megőrzése mellett. A kupac tulajdonképp olyan bináris fa, amelyben igaz, hogy az első (0. sorszámú) elem mindig a legkisebb.* A kupac részfái is kupacok, vagyis ezek is megőrzik a kupactulajdonságot. Íme egy kupac létrehozása a semmiből:

```
import heapq
heap = []
heapq.heappush(heap, (5, "rest"))
heapq.heappush(heap, (2, "work"))
heapq.heappush(heap, (4, "study"))
```

* Szigorú értelemben a `heapq` modul *minimumkupacok* kezelésére képes. Azokat a kupacokat, amelyeknek az első eleme mindig a legnagyobb, *maximumkupacoknak* nevezzük.

Ha már van egy rendelkezésre álló listánk, akkor azt a `heapq.heapify(lista)` hívással alakíthatjuk kupaccá. A tagfüggvény a szükséges újrarendezést helyben végzi el. A kupacból a legkisebb elemet a `heapq.heappop(kupac)` hívással távolíthatjuk el.

```
for x in heapq.merge([1, 3, 5, 8], [2, 4, 7], [0, 1, 6, 8, 9]):
    print(x, end=" ") # prints: 0 1 1 2 3 4 5 6 7 8 8 9
```

A `heapq.merge()` függvénynek akárhány bejárható adatot átadhatunk paraméterül, a visszatérési értéke pedig az összes átadott adat elemeit a megfelelő sorrendben tartalmazó bejárható objektum lesz.

Fájlformátumok, kódolások és az adatok megőrzése

Karakterkódolások 86 ☜

Az alapkönyvtár kiterjedt támogatással rendelkezik a szabványos fájlformátumok és kódolások terén. A `base64` modul az RFC 3548 dokumentumban* megadott Base16, Base32 és Base64 kódolású adatok olvasására és írására való függvények helye. A `quopri` modulban találjuk a „quoted-printable" formátum írására és olvasására való függvényeket. A formátumot az RFC 1521 dokumentum adja meg, és a MIME- (Multipurpose Internet Mail Extensions) adatok kapcsán használatos. Az `uu` modulban találjuk az `uu`, vagyis Unix-Unix kódolású adatok írására és olvasására való függvényeket. Az RFC 1832 dokumentum adja meg az External Data Representation Standard (külső adatok megjelenítésének szabványa) követelményeit; az ilyen formátumú adatok írására és olvasására szolgáló függvényeket az `xdrlib` modulban találjuk.

Az alapkönyvtárban megtalálhatók a legnépszerűbb archívum-formátumok kezelését végző modulok is. A `bz2` modul a `.bz2` kiterjesztésű fájlok kezelését végzi, a `gzip` modul a `.gz` kiterjesztésűekét, a `tarfile` modul a `.tar`, a `.tar.gz`, illetve `.tgz`, valamint a `.tar.bz2` formátumúakét, a `zipfile` modul pedig a `.zip` kiterjesztésűekét. A `tarfile` modul használatára még ebben a részben látunk példát, később pedig egy rövid példa mutatja be a `gzip` modul néhány lehetőségét. A `gzip` modul működését a 7. fejezetben is tanulmányozhatjuk.

Az alapkönyvtár néhány hangfájlformátum kezelését is támogatja: az `aifc` modul az AIFF (Audio Interchange File Format) formátumot, a `wave` modul pedig a tömörítetlen, `.wav` kiterjesztésű fájlok használatát segíti. Egyes hangformátumok az `audioop` modullal kezelhetők, az `sndhdr` modul pedig néhány, a fájlban tárolt hangadatok jellemzőit (például mintavételezési gyakoriság), és egyéb tulajdonságait kideríteni képes függvénnyel szolgál.

A beállítófájlok – a Windows operációs rendszer régi `.ini` fájljaihoz hasonló – egyik formátumát adja meg az RFC 822 dokumentum; az ilyen fájlok írására és olvasására való függvényeket a `configparser` modulban találjuk.

---

* Az RFC, vagyis Request for Comments (véleménynyilvánításra beterjesztett) dokumentumok írják le az Interneten használatos különféle technológiákat. Mindegyik dokumentum saját azonosítószámmal bír, és sokból hivatalosan is elfogadott szabvány lett.

Sok alkalmazás – például az Excel – képes olvasni a CSV (Comma Separated Values, vesszővel tagolt értékek) formátumú fájlokat, illetve ezek változatait – például a tabulátorokkal tagolt adatokat tartalmazókat. A `csv` modul képes az ilyen formátumú fájlok olvasására és írására, és kezeli az olyan egyéni hóbortokat, amelyek megnehezítik a CSV formátumú fájlok egyszerű, azonnali kezelését.

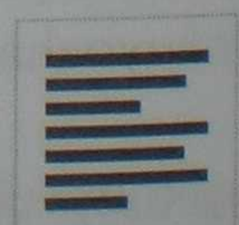
Azon felül, hogy az alapkönyvtár különféle fájlformátumok kezelését támogatja, találunk benne az adatok megőrzését szolgáló modulokat is. A `pickle` modul használható Python-objektumok, akár teljes gyűjtemények lemezre írására és onnan történő visszaolvasására – a modult a 7. fejezetben tárgyaljuk.

Az alapkönyvtár támogatja a különféle adatbázis-formátumokat (DBM) is – ezek a szótár típushoz hasonlóak, de az elemeiket nem a memóriában, hanem lemezen tárolják, a kulcsaik és értékeik pedig egyaránt vagy `bytes` típusú objektumok, vagy karakterláncok. A `shelve` modul, amelyet a 11. fejezetben tárgyalunk, olyan DBM-fájlok kialakítására használható, amelyeknek a kulcsa karakterlánc, az értéke pedig bármely nekünk tetsző, a Pythonban megvalósított osztályba tartozó objektum lehet – a modul a színfalak mögött gond nélkül elvégzi a Python objektumai és a `bytes` formátum közötti átalakításokat. A DBM-modulokról, a Python adatbázis-kezelési alkalmazásprogramozási felületéről és a beépített SQLite adatbázis-kezelőről a 11. fejezet szól.

Példa: A base64 modul

A `base64` modult leginkább az elektronikus levelekbe ASCII formátumú szövegként beágyazott bináris adatok kezelésére használják, de használható bináris adatok Python-programállományokban való tárolására is. Az alábbi kódrészlet feltételezi, hogy a `base64` modult előzőleg betöltöttük, illetve hogy a `left_align_png` változó egy `.png` formátumú fájl elérési útját és nevét tárolja.

```
binary = open(left_align_png, "rb").read()
ascii_text = ""
for i, c in enumerate(base64.b64encode(binary)):
if i and i % 68 == 0:
ascii_text += "\\\n"
ascii_text += chr(c)
```



left_align.png

bytes típus
☞ 297

A fenti kódrészlet bináris fájlként olvassa be az állományt, majd ASCII karakterekből álló, Base64 kódolásúvá alakítja. Minden hatvannyolcadik karakternél elhelyez egy fordított perjelből és egy újsor karakterből álló kombinációt. Ezzel az ASCII-karaktereket tartalmazó sorok hosszát 68 karakterben korlátozza, és vigyáz arra is, hogy az adat visszaolvasásakor az újsor karakterek ne legyenek figyelembe véve (a fordított perjel ugyanis levédi azokat). Az ekképpen nyert, ASCII-kódolású szöveget a Python-programjainkban `bytes` típusú adatként közvetlenül is tárolhatjuk, például így:

```
LEFT_ALIGN_PNG = b"""\
iVBORw0KGgoAAAANSUhEUgAAACAAAAAgCAYAAABzenr0AAAABGdBTUEAALGPC/xhBQAA\
...
bmquu8PAmVT2+CwVV6rCyA9UfFMCkI+bN6p18tCWqcUzrDOwBh2zVCR+JZVeAAAAAElF\
TkSuQmCC"""
```

Ahogy a három pont jelzi, a sorok nagy részét a fenti részletben elhagytuk.

Az adatok az alábbi módszerrel alakíthatók vissza az eredeti formájukra:

```
binary = base64.b64decode(LEFT_ALIGN_PNG)
```

Az `open(fájlnév, "wb").write(binary)` utasítással az adatok fájlba írhatók. A bináris adatok Python-programfájlokban való tárolása lényegesen több helyet igényel, mintha az eredeti formájukban tárolnánk őket, de hasznos lehet, ha olyan programot akarunk írni, amelynél a programfájlban ilyen adatokat kell elhelyeznünk.

Példa: A tarfile modul

A Windows legtöbb változata nem támogatja a Unix rendszereken olyannyira elterjedt `.tar` formátumot. Az ebből eredő kényelmetlenség gyorsan orvosolható a Python `tarfile` moduljával, amely képes `.tar` és `.tar.gz` formátumú archívumok (*tarball* néven is hivatkoznak rájuk) létrehozására és kicsomagolására. Ha a megfelelő könyvtárak telepítve vannak, a `.tar.bz2` formátum is kezelhető a modullal. Az `untar.py` program a `tarfile` modul használatával képes a fenti formátumú archívumok kicsomagolására. Az alábbiakban csak a program kulcsfontosságú részeit mutatjuk be, kezdve az első `import` utasítást tartalmazó kifejezéssel:

```
BZ2_AVAILABLE = True
try:
    import bz2
except ImportError:
    BZ2_AVAILABLE = False
```

A `bz2` modult használjuk a `bzip2` tömörített formátum kezelésére, de ha a Python fordítása úgy történt, hogy a bzip2 programkönyvtár nem volt hozzáférhető, a modul sem lesz betölthető. (A Python Windowsra telepíthető változata mindig tartalmazza a bzip2 tömörítést, legfeljebb néhány Unix-telepítés esetében hiányozhat a tömörítési módszer támogatása.) Annak lehetőségét, hogy a modul esetleg elérhetetlen, a `try...except` utasítások közötti kódrésszel vesszük figyelembe, a vizsgálat eredményét egy később is használható logikai változóban tárolva (bár a változót használó kódrészlet a fentiekben nem szerepel).

```
UNTRUSTED_PREFIXES = tuple(["/", "\\"] +
    [c + ":" for c in string.ascii_letters])
```

A fenti kifejezés a `('/', '\', 'A:', 'B:', ..., 'Z:', 'a:', 'b:', ..., 'z:')` rendezett sorozatot hozza létre. A kicsomagolás alatt lévő, `.tar` formátumú fájlban lévő minden olyan állomány, amely a rendezett sorozatban lévő karakterek valamelyikével kezdődik,

gyanús. A `.tar` formátumú fájloknak nem lenne szabad abszolút elérési utakat tartalmazniuk, hiszen így azt kockáztatjuk, hogy felülírjuk valamelyik rendszerfájlt. Így elővigyázatosságból kicsomagolatlanul hagyjuk azokat a fájlokat, amelyeknek a neve a rendezett sorozatban lévő karakterek valamelyikével kezdődik.

```
def untar(archive):
    tar = None
    try:
        tar = tarfile.open(archive)
        for member in tar.getmembers():
            if member.name.startswith(UNTRUSTED_PREFIXES):
                print("untrusted prefix, ignoring", member.name)
            elif ".." in member.name:
                print("suspect path, ignoring", member.name)
            else:
                tar.extract(member)
                print("unpacked", member.name)
    except (tarfile.TarError, EnvironmentError) as err:
        error(err)
    finally:
        if tar is not None:
            tar.close()
```

A `.tar` formátumú fájlban lévő valamennyi állomány az archívum *tagja* (member). A `tarfile.getmembers()` függvény `tarfile.TarInfo` osztályú objektumokból álló listát ad vissza – tagonként egy objektumot. A taghoz tartozó fájlnév, beleértve az elérési utat, a `tarfile.TarInfo.name` jellemzőből olvasható ki. Ha a név valamilyen gyanús karakterrel kezdődik, vagy `..` található az elérési útban, hibaüzenetet írunk ki, egyéb esetben a `tarfile.extract()` függvény hívásával lemezre mentjük a tagot. A `tarfile` modul saját kivételek teljes készletével bír, mi azonban azt az egyszerű megközelítési módot választottuk, hogy akármilyen kivételes esemény bekövetkeztekor kiírjuk a hibaüzenetet, és kilépünk.

```
def error(message, exit_status=1):
    print(message)
    sys.exit(exit_status)
```

Az `error()` függvény csak a teljesség kedvéért került ide. A fentiekben nem szereplő `main()` függvény felhasználási utasítást ír ki, ha a `-h`, vagy a `-help` kapcsolóval indítják a programot, egyéb esetekben pedig végrehajt néhány alapvető ellenőrzést, és meghívja az `untar()` függvényt.

Fájlok, könyvtárak és folyamatok kezelése

Az `shutil` modul magas szintű fájl- és könyvtárkezelő függvényeket bocsát a rendelkezésünkre, többek között az `shutil.copy()` és az `shutil.copytree()` nevűt, amelyekkel fájlok, illetve teljes könyvtárfák másolása lehetséges. Az `shutil.move()` függvény könyvtárfák mozgatását végzi, az `shutil.rmtree()` függvénnyel pedig teljes könyvtárfák eltávolítása lehetséges – olyanoké is, amelyek nem üresek.

Ideiglenes fájlok létrehozására a `tempfile` modult érdemes használni, amelyben megtalálhatók a szükséges függvények – például a `tempfile.mkstemp()` –, és a fájlok létrehozásánál a lehető legnagyobb biztonsági követelményeknek igyekszik megfelelni.

A `filecmp` modul `filecmp.cmp()` függvényével fájlokat, a `filecmp.cmpfiles()` függvényével pedig teljes könyvtárakat hasonlíthatunk össze.

A Python nyelvű programok igen hatékonyak és sokoldalúak lehetnek egyéb programok futásának vezénylőjeként. Ilyenkor használhatjuk a `subprocess` modult, amely képes egyéb folyamatok elindítására, a velük csővezetéken keresztül való kommunikációra, és a tőlük kapott eredmények fogadására. A modullal a 9. fejezetben foglalkozunk. Még ennél a modulnál is többre képes a `multiprocessing` modul, amely kiterjedt eszköztárával képes egy feladatot több folyamat között elosztani, és az eredményeket összesíteni – eképpen többszálú program készítése helyett gyakran lehet megfontolandó választás.

Az `os` modul operációs rendszertől független hozzáférést biztosít az operációs rendszer bizonyos részeihez. Az `os.environ` változó egy megfeleltetési típusú objektumot tárol, amelynek elemei a környezeti változók nevei és értékei. A program munkakönyvtárát az `os.getcwd()` függvény mutatja meg, a megváltoztatása pedig az `os.chdir()` függvénnyel lehetséges. A modulban találhatók olyan függvények is, amelyek az alacsonyszintű, fájlleíró alapú fájlkezelést végzik. Az `os.access()` függvény használható egy fájl létének, olvashatóságának, írhatóságának ellenőrzésére, az `os.listdir()` a neki átadott könyvtár tartalmát adja vissza (vagyis a fájljait és könyvtárait, de a `.` és a `..` könyvtárbejegyzéseket nem), az `os.stat()` függvény pedig különféle információkat ad vissza az aktuális fájlról, illetve könyvtárról, például a módját (ki olvashatja, ki írhatja, ki hajthatja végre), az utolsó elérése idejét és a méretét.

Könyvtárak létrehozására az `os.mkdir()` függvény használható, ha pedig köztes könyvtárakat is létre kell hozni, használjuk az `os.makedirs()` függvényt. Az üres könyvtárak eltávolítása az `os.rmdir()` függvénnyel, a csak üres könyvtárakat tartalmazó könyvtárak eltávolítása az `os.removedirs()` függvénnyel lehetséges. Fájlok és könyvtárak eltávolítására az `os.remove()`, átnevezésére az `os.rename()` függvény használható.

Az `os.walk()` függvény teljes könyvtárfák bejárására alkalmas, és az egyes fájlok, illetve könyvtárak nevét adja vissza.

Az `os` modul tartalmaz néhány alacsonyszintű, csak egy-egy operációs rendszerre jellemző függvényt is, például a fájlleírókkal való munkára, folyamatok elágaztatására (csak Unix-rendszereken), más folyamat indítására, illetve leszármaztatására.

Míg az `os` modul függvényei főleg az operációs rendszerrel való érintkezésre valók, különös tekintettel a fájlrendszerre, az `os.path` modul egyedi keverékét tartalmazza az elérési utak karakterláncokként való kezelésének, illetve a fájlrendszerrel kapcsolatos, a kényelmünket

szolgáló függvényeknek. Az `os.path.abspath()` függvény a neki átadott paraméter abszolút elérési útját adja vissza, a többször szereplő elválasztók, illetve a .. elemek nélkül. Az `os.path.split()` függvény visszatérési értéke egy kételemű rendezett sorozat, amelynek az első eleme az elérési utat, a második a fájlnevet tartalmazza – ez utóbbi üres, ha fájlnév nélküli elérési utat adtunk meg. A két rész közvetlenül is hozzáférhető az `os.path.basename()` és az `os.path.dirname()` függvénnyel.
Az `os.path.splitext()` függvény két részre osztja a fájlneveket: névre és kiterjesztésre.
Az `os.path.join()` függvény paraméterei olyan karakterláncok, amelyek elérési utakban szerepelhetnek, a visszatérési értéke pedig egyetlen elérési út, amelynek a részei között az adott operációs rendszerre jellemző elválasztó karakter található.

Ha sok különféle információra vagyunk kíváncsiak egy adott fájlt, illetőleg könyvtárat illetőleg, az `os.stat()` függvényt, ha azonban csak egy adott információt szeretnénk kideríteni, az `os.path` modul megfelelő függvényét használhatjuk. A fájl, illetve könyvtár létéről az `os.path.exists()`, a méretéről az `os.path.getsize()` függvény nyújt felvilágosítást, az `os.path.isfile()` és az `os.path.isdir()` függvénnyel pedig eldönthető, hogy az éppen vizsgált objektum fájl vagy könyvtár-e.

A `mimetypes` modulba került a `mimetypes.guess_type()` függvény, amely az átadott fájl MIME-típusát próbálja kitalálni.

Példa: Az os és az os.path modul

Az alábbiakban bemutatjuk, hogy miként hozhatunk létre az `os`, illetve az `os.path` modullal olyan szótárat, amelynek a kulcsai az elérési utat is tartalmazó fájlnevek, az értékei pedig az elérési útban megadott fájlok utolsó módosítási idejét megmutató időbélyegek (a Unix-időszámítás kezdete óta eltelt másodpercek száma):

```
date_from_name = {}
for name in os.listdir(path):
    fullname = os.path.join(path, name)
    if os.path.isfile(fullname):
        date_from_name[fullname] = os.path.getmtime(fullname)
```

A kód meglehetősen magától értetődő, és csak egyetlen könyvtárban lévő fájlok dolgozhatók fel vele. Ha egy egész könyvtárfát kell végigjárnunk, az `os.walk()` függvény lehet a segítségünkre.

Egy kódrészlet következik a többször előforduló fájlok keresését végző `finddup.py` programból.* A kód egy olyan szótárat alakít ki, amelynek minden kulcsa egy-egy kételemű, egy fájl méretét és elérési út nélküli nevét tároló rendezett sorozat (tuple), az egyes értékek pedig olyan listák, amelyek a kulcsban tárolt fájllal megegyező méretű és nevű fájlok elérési utat is tartalmazó neveit tárolják:

* Egy szintén a többször előforduló fájlok keresését végző, ám lényegesen kifinomultabb programot tárgyalunk a 9. fejezetben. A `finddupliсates-t.py` többszálú, és a működése MD5 ellenőrzőösszegek képezésén alapul.

```
data = collections.defaultdict(list)
for root, dirs, files in os.walk(path):
for filename in files:
fullname = os.path.join(root, filename)
key = (os.path.getsize(fullname), filename)
data[key].append(fullname)
```

Az egyes könyvtárak tekintetében az `os.walk()` függvény visszaadja a gyökérkönyvtárat, valamint két listát is: az egyik az adott könyvtár alkönyvtárait, a másik a benne lévő fájlokat tartalmazza. A teljes elérési út képzése a gyökérkönyvtár és a tárolt fájl nevének kombinációjával oldható meg. Figyeljük meg, hogy nem a mi feladatunk a könyvtárak alkönyvtárainak bejárása – az `os.walk()` függvény megteszi ezt helyettünk. Ha az adatgyűjtéssel végeztünk, a végigjárás következik, amelynek az eredménye az a jelentés, amely az esetleg többször előforduló fájlok neveit tartalmazza:

```
for size, filename in sorted(data):
    names = data[(size, filename)]
    if len(names) > 1:
        print("{0} ({1} bytes) may be duplicated "
            "({2} files):".format(filename, size, len(names)))
        for name in names:
            print("\t{0}".format(name))
```

Minthogy a szótár kulcsai (*méret, fájlnév*) alakú rendezett sorozatok, nem kell a `key` függvénnyel méret szerinti sorrendbe állítani az adatokat. Ha bármely (*méret, fájlnév*) rendezett sorozat listájában egynél több fájlnév szerepel, valószínűleg többszöri előfordulással állunk szemben.

```
...
shell32.dll (8460288 bytes) may be duplicated (2 files):
        \windows\system32\shell32.dll
        \windows\system32\dllcache\shell32.dll
```

Az előző sorok az utolsók abból a 3282 sornyi kimenetből, amely egy Windows XP rendszeren futtatott `finddup.py \windows` parancs kimenetéből származik.

Hálózatkezelés és internetes programozás

A hálózatkezeléssel és az Internettel kapcsolatos programozási feladatok ellátásában szerepet játszó csomagok és modulok a Python alapkönyvtárának jelentős részét képezik. A `socket` modulban található függvények a legalacsonyabb szinten valósítják meg az alapvető hálózati műveleteket. Találunk csatolók (socket) létrehozására, DNS-keresések (Domain Name System) végrehajtására, és IP-címek (Internet Protocol) kezelésére alkalmas függvényeket. Az `ssl` modullal titkosított és azonosítást kívánó csatolók alakíthatók ki. A `socketserver` modul TCP- (Transmission Control Protocol) és UDP- (User Datagram Protocol) kiszolgálókkal lát el bennünket. A kiszolgálók alkalmasak a kérelmek közvetlen kezelésére, de az egyes kérelmek kezelésére létre tudnak hozni önálló folyamatot (elágaztatással), sőt akár önálló szálat is. Az aszinkron ügyfél-kiszolgáló csatolókezelés az `asyncore`, illetve a rá épülő, magasabb szintű `asynchat` modul használatával valósítható meg.

A Python a WSGI (Web Server Gateway Interface, webkiszolgálói átjárófelület) felületet azért alakította ki, hogy a webkiszolgálók és a Python nyelven írt alkalmazások közötti kommunikációnak szabványos felületet biztosítson. A szabványt támogatandó, a `wsgiref` csomag tartalmazza a WSGI egy példaként szolgáló megvalósítását, modulokkal a WSGI felülettel együttműködni képes HTTP-kiszolgálók kialakításához, illetve a válaszfejlécek és a CGI-parancsfájlok (Common Gateway Interface) kezeléséhez. Ezen felül a `http.server` modul olyan HTTP-kiszolgálót ad a kezünkbe, amelynek átadható egy CGI-parancsfájlok futtatásához szükséges kérelemkezelő. A `http.cookies` és a `http.cookiejar` modul tartalmazza a sütik kezelését végző függvényeket, a CGI-parancsfájlok támogatását pedig a `cgi` és a `cgitb` modul végzi.

A HTTP-kérelmekhez ügyféloldalról a `http.client` modullal férhetünk hozzá, bár a magasabb szintű `urllib` csomag `urllib.parse`, `urllib.request`, `urllib.response`, `urllib.error` és `urllib.robotparser` moduljai könnyebb és kényelmesebb URL-kezelést tesznek lehetővé. Egy fájl letöltése az Internetről ilyen egyszerű:

```
fh = urllib.request.urlopen("http://www.python.org/index.html")
html = fh.read().decode("utf8")
```

Az `urllib.request.urlopen()` függvény olyan objektumot ad vissza, amelynek a viselkedése leginkább egy bináris módban olvasásra megnyitott fájléra hasonlít. A fenti példában a Python webhelyének `index.html` fájlját `bytes` osztályú objektumként töltjük le, majd a tartalmát a `html` változóban tároljuk, immár karakterláncként. Fájlok letöltése és helyi fájlként való tárolása is lehetséges; ilyenkor a `urllib.request.urlretrieve()` függvényt érdemes használnunk.

A HTML és az XHTML dokumentumok feldolgozását végezhetjük a `html.parser` modullal, az URL-ek feldolgozását és létrehozását az `urllib.parser` modullal, a `robots.txt` állományok feldolgozását pedig az `urllib.robotparser` modullal. A JSON (JavaScript Object Notation) formátum használatával ábrázolt adatok olvasását és írását a `json` modullal oldhatjuk meg.

A HTTP-kiszolgálók és -ügyfelek támogatásán túl az alapkönyvtár az XML-RPC (Remote Procedure Call, távoli eljáráshívás) technológiát is támogatja, mégpedig a `xmlrpc.client` és az `xmlrpc.server` modulokkal. További ügyféloldali támogatás található az FTP (File Transfer Protocol, fájlátviteli protokoll) protokollhoz az `ftplib` modulban, az NNTP (Network News Transport Protocol, hálózati hírcsatornák kezelésére szolgáló protokoll) protokollhoz az `nntplib` modulban és a TELNET protokollhoz a `telnetlib` modulban.

Az `smtpd` modul egy SMTP-kiszolgálót (Simple Mail Transport Protocol, egyszerű levéltovábbítási protokoll) bocsát a rendelkezésünkre, a levelezés ügyféloldali moduljai pedig az `smtplib` az SMTP esetében, az `imaplib` az IMAP4 (Internet Message Access Protocol, internetes üzenethozzáférési protokoll) esetében, és a `poplib` a POP3 (Post Office

Protocol, postahivatal-protokoll) esetében. A különböző formátumú postafiókok elérését a `mailbox` modul biztosítja. Egy-egy üzenet (akár több részes is lehet) létrehozására és kezelésére az `email` modul szolgál.

Ha az alapkönyvtár csomagjai és moduljai elégtelennek bizonyulnak a területen, a Twisted nevű, átfogó, külső fejlesztésű hálózatkezelő könyvtár (`www.twistedmatrix.com`) lehet a megoldás. Sok további külső fejlesztésű könyvtár található az Interneten, ilyen például a webalkalmazások készítésére való Django (`www.djangoproject.com`) és Turbogears (`www.turbogears.org`), valamint a teljes webes keretrendszert és tartalomkezelő rendszert kínáló Plone (`www.plone.org`) és Zope (`www.zope.org`). Az előzőekben felsorolt könyvtárakat mind Python nyelven írták.

XML

Az XML nyelvű dokumentumok feldolgozására két széles körben elterjedt módszer áll a rendelkezésünkre. Az egyik a DOM (Document Object Model, dokumentum-objektum-modell), a másik pedig a SAX (Simple API for XML) használata. Az alapkönyvtárral két DOM-feldolgozót kapunk; az egyik az `xml.dom` modulban, a másik az `xml.dom.minidom` modulban található. A SAX-feldolgozást az `xml.sax` modul biztosítja. A benne lévő `xml.sax.saxutils.escape()` tagfüggvény miatt már használtuk az `xml.sax.saxutils` modult (az „&", a „>" és a „<" jelek XML-ben szokásos levédésére). Létezik az `xml.sax.saxutils.quoteattr()` függvény is, amely ugyanezt teszi, de ezen felül levédi az idézőjeleket is (alkalmassá téve a szöveget arra, hogy egy címke jellemzőjeként használjuk). Az ellenkező irányú átalakítást a `xml.sax.saxutils.unescape()` függvény végzi.

Két további feldolgozót is használhatunk. Amennyiben az `expat` könyvtár hozzáférhető, az `xml.parsers.expat` modul használható az XML-dokumentumok `expat`-tal való feldolgozására. Az `xml.etree.ElementTree` modul segítségével pedig egyfajta szótár/lista-felületet használva végezhetjük el az XML-dokumentumok feldolgozását. (Alapértelmezés szerint a DOM és az elemfa-feldolgozók maguk is az `expat` feldolgozót használják a háttérben.)

Az XML-dokumentumok saját kezű, a DOM-mal, illetve elemfák használatával való megírása, valamint az XML-dokumentumok DOM-, SAX- vagy elemfa-feldolgozóval történő értelmezése a 7. fejezet témája.

Példa: Az xml.etree.ElementTree modul

A Python DOM- és SAX-feldolgozói szolgáltatják azt az alkalmazásprogramozási felületet, amihez a tapasztalt XML-programozók szoktak hozzá, az `xml.etree.ElementTree` modul pedig egy „pythonosabb" megközelítését kínálja az XML-dokumentumok olvasásának és írásának. Az elemfa-modul csak mostanában került be az alapkönyvtárba*, így néhány olvasónk esetleg nem ismeri. Ennek tükrében az alábbiakban ízelítőt nyújtunk belőle,

* A `xml.etree.ElementTree` modul a 2.5-ös Python-változatban szerepelt először.

mégpedig egy igen rövid példa segítségével. A 7. fejezetben találunk egy alaposabb példát, és az a fejezet mutat majd be egy összehasonlításra is alkalmas kódot, a DOM és a SAX technológiák használatával.

Az Egyesült Államok kormányzatának NOAA (National Oceanic and Atmospheric Administration) webhelye meglehetősen sokféle adatot szolgáltat, többek között egy olyan XML-állományt, amelyben az Egyesült Államok időjárás-megfigyelő állomásainak listája található. A fájl 20 000 sornál is hosszabb, és nagyjából kétezer állomás részletes adatait tartalmazza. Íme egy átlagos állomás:

```
<station>
    <station_id>KBOS</station_id>
    <state>MA</state>
    <station_name>Boston, Logan International Airport</station_name>
    ...
    <xml_url>http://weather.gov/data/current_obs/KBOS.xml</xml_url>
</station>
```

Pár sort kihagytunk, és a fájlban használt behúzást is csökkentettük. A fájl mérete körülbelül 840 kilobájt, így a gzip használatával 72 kilobájtos, használható méretűre tömörítettük. Sajnos az elemfa-feldolgozó fájlnevet vagy fájlobjektumot vár olvasásra, így tömörített fájlt nem adhatunk át neki, az ugyanis a feldolgozó szempontjából nem lenne egyéb, mint véletlenszerű bináris adatok halmaza. A probléma megoldását szolgálja a két kezdeti lépés:

```
binary = gzip.open(filename).read()
fh = io.StringIO(binary.decode("utf8"))
```

io.StringIO 74, 209 ☞

bytes típus ☞ 297

A `gzip` modul `gzip.open()` függvénye hasonló a beépített `open()` függvényhez, azzal a különbséggel, hogy a gzip tömörítésű (`.gz` kiterjesztésű) állományokat nyers bináris adatként képes olvasni. Az adatokra olyan fájl formájában van szükségünk, amellyel az elemfa-feldolgozó dolgozni tud. Így a `bytes.decode()` tagfüggvénnyel a bináris adatot karakterlánccá alakítjuk, ugyanannak az UTF-8 karakterkódolásnak a használatával, amit az XML-fájl is használ, majd létrehozunk egy fájlként viselkedő `io.StringIO` osztályú objektumot, benne az XML-fájlban lévő összes adatot tartalmazó karakterlánccal.

```
tree = xml.etree.ElementTree.ElementTree()
root = tree.parse(fh)
stations = []
for element in tree.getiterator("station_name"):
    stations.append(element.text)
```

A fenti kódrészletben új `xml.etree.ElementTree.ElementTree` osztályú objektumot hozunk létre, majd a feldolgozni kívánt XML-adatokat fájlobjektum formájában átadjuk neki. Az elemfa-feldolgozó szempontjából az átadott adat egy olvasásra megnyitott fájlt tartalmazó objektum, még akkor is, ha a valóságban egy `io.StringIO` osztályú objektumban tá-

rolt karakterláncot kapott. Szeretnénk kinyerni az egyes időjárásjelző állomások neveit, és ennél mi sem egyszerűbb, ha a `xml.etree.ElementTree.ElementTree.getiterator()` tagfüggvényt használjuk, amelynek a visszatérési értéke egy olyan bejárható objektum, amely viszont az átadott címkenévvel bíró `xml.etree.ElementTree.Element` osztályú objektumokat adja vissza. A szöveg kinyeréséhez egyszerűen az elem `text` jellemzőjéhez fordulunk. Csakúgy, mint az `os.walk()` függvény esetében, itt sem magunknak kell elvégeznünk a bejárást: a bejárható objektum megteszi ezt helyettünk. Címkét sem kell megadnunk – a bejárható objektum így ugyanis a teljes XML-dokumentum minden egyes elemét visszaadja.

Egyéb modulok

Nem áll rendelkezésünkre elegendő hely az alapkönyvtár majd' kétszáz csomagjának és moduljának tárgyalására. Mindazonáltal ez az általános áttekintés elvileg elég volt arra, hogy belekóstoljunk az alapkönyvtár nyújtotta szolgáltatásokba, és megismerjük az általa lefedett főbb területek legfontosabb csomagjait. A fejezet utolsó részéhez érve néhány további, érdeklődésünkre számot tartó témakör megismerésével folytatjuk.

Az előzőkben láthattuk, hogy mennyire egyszerűen tudunk dokumentációs karakterláncokat létrehozni, és milyen egyszerű őket a `doctest` modullal futtatni. Az alapkönyvtárnak is van egy egységtesztek lebonyolítására szolgáló keretrendszere: a `unittest` modul, amely a Java nyelv JUnit teszt-keretrendszerének Python nyelvű változata. A `doctest` modul képes alapszinten együttműködni a `unittest` modullal. Számos külső fejlesztésű tesztelési keretrendszer is a rendelkezésünkre áll, például a `codespeak.net/py/dist/` webhelyről letölthető `py.test` és a `www.somethingaboutorange.com/mrl/projects/nose/` webhelyről letölthető `nose`.

A közvetlen felhasználói beavatkozás nélkül futó alkalmazások, például a kiszolgálók, gyakran használnak naplófájlokat a felmerülő problémák jelzésére. A `logging` modul egységes, a naplózás feladatának ellátását segítő felületet nyújt. Azon felül, hogy fájlokba képes naplózni, a GET és a POST HTTP-módszerrel is el tudja végezni a naplózást, sőt elektronikus levelek vagy csatolók használatával is.

Az alapkönyvtárnak vannak olyan moduljai is, amelyek az önelemzést, illetve a kód módosítását szolgálják. A legtöbbjük tárgyalása túlmutat könyvünk keretein, mégis megemlítünk egyet: a `pprint` nevűt, amely a Python-objektumok és -adattípusok kódjának formázására (angol nyelvű szakzsargonban „pretty printing" – innen a modul neve) való, és mint ilyen, gyakran van a segítségünkre a hibakeresés során is. Az `inspect` modul használatát a 8. fejezetben mutatja be egy egyszerű példa; a modul az élő objektumok (live objects) önelemzését segíti.

A `threading` modul a többszálú alkalmazások készítését támogatja, a `queue` pedig három különböző szálbiztos sortípus-megvalósítást kínál. A többszálúsággal a 9. fejezet foglalkozik.

A Python nem támogatja beépítetten a grafikus felhasználói felületek készítését, de Python nyelvű programjainkban számos ilyen feladatot ellátó programkönyvtárat használhatunk. A Tk könyvtárat a rendszerint az alaptelepítés részeként a gépünkre kerülő `tkinter` modul támogatja. A grafikus felhasználói felületek programozásába a 13. fejezet nyújt betekintést.

Az `abc` (Abstract Base Class, elvont alaposztály) modul az elvont alaposztályok létrehozásához szükséges függvények helye. A modullal a 8. fejezetben foglalkozunk.

A `copy` modulban találjuk a 3. fejezetben megismert `copy.copy` és `copy.deepcopy` függvényeket.

Sekély és mély másolás 143 ☜

A *külső függvények*, vagyis a megosztott könyvtárakban (a Windows `.dll`, a Mac OS X `.dylib` és a Linux `.so` állományaiban) lévő függvények a `ctypes` modul használatával válnak hozzáférhetővé.

A Pythonnak része a C nyelvhez való alkalmazásprogramozási felület, vagyis lehetőségünk van arra, hogy a C nyelven létrehozott egyedi adattípusokat és függvényeket a Python nyelvű programjainkban is használjuk. A `ctypes` modul és a C-alkalmazásprogramozási felület tárgyalása egyaránt kívül esik könyvünk keretein.

Ha az előzőekben említett csomagok, illetve modulok egyikében sem találjuk meg, amit keresünk, ne fogjunk hozzá azonnal a megírásához. Előbb érdemes körülnéznünk a Python dokumentáció *Global Module Index*-ében, hátha létezik az a modul, amelyet éppen megírni készülünk – a fejezetben nem volt lehetőség minden modul megemlítésére. Ha itt sem találnánk, amit keresünk, keresgéljünk a *Python Package Index* webhelyen (`pypi.python.org/pypi`), ahol több ezer Python-bővítményt lelhetünk, a rövid, egyetlen fájlból álló moduloktól a hatalmas programkönyvtárakig, illetve a pár sorból álló csomagocskáktól a több száz modulból álló keretrendszerekig.

Összefoglalás

A fejezet elején bemutattuk a csomagok, modulok, illetve a modulok belsejében lévő objektumok betöltésekor használatos utasításformákat. Rámutattunk, hogy a névütközések elkerülése végett sok programozó az `import valami` utasításformát használja. Ügyelnünk kell arra, hogy a programjaink, illetve moduljaink a felső szintű Python-moduloktól, illetve -mappáktól eltérő nevet kapjanak.

A Python csomagjairól szintén esett szó. Egyszerűen olyan mappákról beszélünk, amelyekben egy `__init__.py` nevű fájlt és egy vagy több `.py` kiterjesztésű modulfájlt találunk. Az `__init__.py` nevű fájl üres is lehet, de ha szeretnénk támogatni a `from valami import *` utasításformát, akkor az `__init__.py` nevű fájlban létre kell hoznunk egy `__all__` nevű, különleges szerepű változót, amely értékül a modulok neveiből álló listát kapja.

Az `__init__.py` fájlban emellett elhelyezhetünk általánosan használt előkészítő kódot is. Megemlítettük, hogy a csomagok egymásba ágyazása is járható út: almappák létrehozásával és az egyes almappák saját `__init__.py` fájljainak létrehozásával történik.

Két egyedi modult tárgyaltunk részletesebben. Az első mindössze néhány függvényt tartalmaz, `doctest`-jei pedig igen egyszerűek. A második kidolgozása alaposabb: saját kivételek, futásidő alatti függvénylétrehozás az operációs rendszernek megfelelő egyedi függvény kialakítására, privát globális adatok, egy előkészítő függvény meghívása, valamint alaposabban átgondolt `doctest`-ek jellemzik.

A fejezetnek nagyjából a felét a Python alapkönyvtárának madártávlatú áttekintésére áldoztuk. Több karakterlánc-kezelő modult említettünk, és bemutattunk néhány, az `io.StringIO` osztállyal kapcsolatos példát. Az egyik példa bemutatta, hogy miként lehetséges szöveg fájlba írása, akár a beépített `print()` függvény, akár a fájlobjektumokra jellemző `write()` tagfüggvény használatával. Megtudtuk azt is, hogy miként használható igazi fájl helyett egy `io.StringIO` osztályú objektum. Az előző fejezetekben a `sys.argv` objektum feldolgozását saját magunk végeztük, de amikor az alapkönyvtár parancssori programozást segítő részéről volt szó, megismertük a parancssori paraméterek kezelését nagyban leegyszerűsítő `optparse` modult – amelyet mostantól kiterjedten használunk.

Szóba került az is, hogy a Python nyelv remekül alkalmas számokkal való munkára. Átfutottuk az alapkönyvtár számok tárolására való adattípusait és a matematikai függvényeknek otthont adó három modult csakúgy, mint a `SciPy` projekt által kínált, tudományos és mérnöki számítások elvégzését segítő lehetőségeket. Futólag említettük az alapkönyvtár és a külső fejlesztőktől származó, dátum- és időkezeléssel foglalkozó osztályokat, láttunk példát az aktuális dátum és idő kinyerésére, illetve a Unix-időszámítás kezdete óta eltelt másodpercek, valamint a `datetime.datetime` tagfüggvény kimenete közötti átváltásra. Megtárgyaltuk az alapkönyvtárban lévő gyűjteményes adattípusokat és a rendezett sorozatokkal való munkát támogató algoritmusokat, illetve áttekintettünk néhány, a `heapq` modul függvényeit bemutató példát.

Előkerültek azok a modulok, amelyek – a karakterkódolásokon felül – a különféle fájlkódolásokat támogatják, valamint azok is, amelyek a legnépszerűbb archívumformátumok kezelését biztosítják, sőt a hangadatokat kezelő modulokról is esett szó. Láttunk példát arra, hogy miként tárolható Python nyelvű állományban Base64 kódolással átalakított bináris adat, és megvizsgáltunk egy `.tar` kiterjesztésű fájlok kicsomagolására szolgáló programot is. A fájlok és mappák kezelésének támogatottsága jelentős – ráadásul mindez operációs rendszertől független függvényekkel valósul meg. Mutattunk példát olyan szótár létrehozására, amelynek a kulcsai fájlnevek, az értékei pedig az utolsó módosítást jelző időbélyegek, és értelmeztük egy könyvtárak rekurzív (alkönyvtárakat is bejáró) átvizsgálását végző, fájlnév és -méret alapján többször előforduló fájlokat kereső program működését.

Az alapkönyvtár nagy részét a hálózatkezelésnek és az Internettel kapcsolatos programok írásának szentelték. Éppen csak érintettük a rendelkezésünkre álló lehetőségeket, amelyek között a nyers csatolók (beleértve a kódolt csatolókat) éppúgy megtalálhatók, mint a TCP- és UDP-kiszolgálók vagy a HTTP-kiszolgálók és a WSGI támogatása. Megemlítettük a sütik, a CGI-parancsfájlok, a HTTP-adatok kezelésére, a HTML-, az XHTML- és az URL-értelmezésre szolgáló modulokat is. A további megemlített modulok között szerepeltek az XML-RPC kezelését, illetve a magasabb szintű protokollok – az FTP, az NNTP, az SMTP protokoll ügyfél- és kiszolgálóoldala, illetve az IMAP4 és a POP3 ügyféloldala – támogatását végző modulok.

Szintén megemlítettük az alapkönyvtár kiterjedt XML-író és -értelmező támogatását, beleértve a DOM-, a SAX- és az elemfa-értelmezőket, illetve az `expat` modult, és példát is láttunk az elemfa-értelmező modul működésére. Emellett megemlítettünk néhány további, az alapkönyvtárban található csomagot és modult.

A Python alapkönyvtára hihetetlenül hasznos forrást jelent, elképesztő mennyiségű idő és erőfeszítés spórolható meg a használatával, sok esetben pedig lényegesen rövidebb programok írhatók az alapkönyvtár nyújtotta lehetőségek kihasználásával. Azoknak a területeknek a lefedésére, amelyeken az alapkönyvtár esetleg nem kielégítő, szó szerint sok ezer külső fejlesztésű csomag létezik. Ennek a rengeteg dolognak a segítségével úgy összpontosíthatunk a programjaink működésére, hogy a részletekkel való törődést a programkönyvtárakban lévő modulokra hagyhatjuk.

A fejezettel befejeztük az eljárásközpontú programozás alapjainak áttekintését. A következő fejezetek, különösen a 8., az eljárásközpontú programozás mélyebb, kevésbé ismert területeire kalauzolnak, a soron következő fejezetben pedig bemutatjuk az objektumközpontú programozást. A Python nyelv tisztán eljárásközpontú nyelvként való használata lehetséges, és – főként kis programok esetében – tanácsos is, de a közepes és nagy programoknál, az egyedi csomagoknál és moduloknál, és azokban az esetekben, ahol a hosszú időn át karbantartható kód fontos szempont, az objektumközpontú megközelítés a befutó. Szerencsére minden, amit eddig tárgyaltunk, egyaránt hasznos és helytálló az objektumközpontú programozás világában is, így a következő fejezetek az eddig megalapozott Python-tudásunkra építve fejleszthetik tovább programozói képességeinket.

Gyakorlat

Írjunk olyan programot, amely könyvtárak tartalmát írja ki, vagyis a Windows `dir`, illetve a Unix `ls` parancsához hasonló feladatot végez. A saját könyvtártartalom-kiíró program előnye, hogy megadhatjuk a saját alapértelmezéseinket, és a programot minden operációs rendszeren használhatjuk, tehát nem kell emlékeznünk a `dir` és az `ls` közötti különbségekre. Készítsük el az alábbiaknak megfelelő felhasználói felülettel rendelkező programot:

```
Usage: ls.py [options] [path1 [path2 [... pathN]]]

The paths are optional; if not given . is used.

Options:
-h, --help        show this help message and exit
-H, --hidden      show hidden files [default: off]
-m, --modified    show last modified date/time [default: off]
-o ORDER, --order=ORDER
                  order by ('name', 'n', 'modified', 'm', 'size', 's')
                  [default: name]
-r, --recursive recurse into subdirectories [default: off]
-s, --sizes       show sizes [default: off]
```

(A kimenetet a könyv oldalszélességéhez igazítottuk.)

Íme egy kis könyvtár tartalma, amelyet az `ls.py -ms -os misc/` paranccsal írattunk ki:

```
2007-04-10 15:49:01          322 misc/chars.pyw
2007-08-01 11:24:57        1,039 misc/pfa-bug.pyw
2007-10-12 09:00:27        2,445 misc/test.lout
2007-04-10 15:50:31        2,848 misc/chars.png
2008-02-11 14:17:03       12,184 misc/abstract.pdf
2008-02-05 14:22:38      109,788 misc/klmqtintro.lyx
2007-12-13 12:01:14    1,359,950 misc/tracking.pdf
                                 misc/phonelog/

7 files, 1 directory
```

A parancssorban a paramétereket csoportokba foglaltuk (az `optparse` modul önműködően kezeli mindezt), és ugyanezt az eredményt értük volna el a paraméterek szétválasztásával (`ls.py -m -s -os misc/`), egyetlen csoportba foglalásával (`ls.py -msos misc/`), a hosszú paraméterformák használatával (`ls.py –modified –sizes –order=size misc/`), illetve ezek bármely kombinációjával. Megjegyeznénk, hogy „rejtettnek" azokat a mappákat és fájlokat tekintjük, amelyeknek ponttal (`.`) kezdődik a nevük.

A gyakorlat sok kihívást rejt magában. Át kell olvasnunk az `optparse` modul dokumentációját, és meg kell értenünk, hogy milyen paraméterekkel állítható be `True` érték, és hogy miként ajánlható fel lehetőségek egy adott listája. Ha a felhasználó rekurzív bejárást kér, a fájlok feldolgozását (a könyvtárakét nem) az `os.walk()` függvénnyel kell elvégeznünk, egyébként az `os.listdir()` függvény használatával magunknak kell gondoskodnunk mind a fájlok, mind a könyvtárak feldolgozásáról.

A rejtett könyvtárak elkerülése egy kis ügyességet igényel a rekurzív feldolgozás során. A lista módosításával eltávolíthatók az `os.walk()` tagfüggvény `dirs` listájából – így az `os.walk()` majd kihagyja őket. De vigyázzunk: ne adjuk értéket közvetlenül a `dirs` változónak – hiszen ezzel nem a hivatkozott listát fogjuk megváltoztatni, csak egyszerűen

(és értelmetlenül) lecseréljük egy másikra. Érdemes a receptkönyvi megoldást használnunk: adjunk a teljes lista egy részlistájának értéket, vagyis gondoljunk a következő alakú utasításra: `dirs[:] = [dir for dir in dirs if not dir.startswith(".")]`.

`locale.setlocale()`
81 ☜

A fájlméretek kiírásánál a megfelelő számcsoportképző karakter kinyerésének legjobb módja, ha betöltjük a `locale` modult; a `locale.setlocale()` függvény hívásával megkapjuk a felhasználó alapértelmezett beállításait, majd az `n` formázókaraktert használjuk. Az `ls.py` nagyjából 130 sor hosszú, a 130 sor négy függvény között oszlik meg.

6

Objektumközpontú programozás

- Az objektumközpontú megközelítés
- Egyedi osztályok
- Egyedi gyűjteményosztályok

A korábbi fejezetekben rengeteg objektumot használtunk, a programozási stílusunk azonban szigorúan eljárásközpontú volt. A Python azonban többféle megközelítést alkalmazó nyelv, vagyis egyaránt lehetővé teszi az eljárásközpontú, az objektumközpontú és a függvényközpontú programozási stílus alkalmazását, sőt ezek keverékét is – hiszen semmiféle módon nem korlátozza a programozási lehetőségeinket.

Tetszőleges programot megírhatunk eljárásközpontú alakban, és ez a kisebb programoknál (mondjuk úgy 500 sorig) egyáltalán nem jár semmiféle nehézséggel. A legtöbb program esetében azonban – különösképpen a közepes, illetve nagy méretűeknél – az objektumközpontú programozás számos előnnyel jár.

Ebben a fejezetben bemutatjuk mindazokat az alapelveket és módszereket, amelyek lehetővé teszik az objektumközpontú programozást a Pythonban. Az első rész leginkább azoknak szól, akik kevesebb tapasztalattal rendelkeznek ezen a téren, illetve eljárásközpontú nyelveket (mint a C vagy a Fortran) használtak a korábbiakban. Először olyan kérdésekre világítunk rá, amelyek gondot okozhatnak az eljárásközpontú programozás világában, de az objektumközpontú módszerek választ adnak rájuk. Ezt követően röviden bemutatjuk az objektumközpontú programozás alapjait a Pythonban, valamint megismerkedünk néhány alapvető kifejezéssel. Ha mindezeken túlestünk, rátérünk a fejezet két fő részére.

A második részben megtudjuk, hogyan hozhatunk létre egyedi adattípusokat, amelyek egy-egy elemet tárolnak (jóllehet ezek az elemek több jellemzővel is rendelkezhetnek), majd megismerkedünk az egyedi gyűjteményes adattípusokkal, amelyek tetszőleges számú és típusú objektumot tartalmazhatnak. Ezeken az oldalakon szinte mindent megtalálunk, amit az objektumközpontú programozásról a Pythonban tudnunk kell, bár néhány komolyabb felkészültséget igénylő témakört csak a 8. fejezetben fejtünk ki bővebben.

Az objektumközpontú megközelítés

A következőkben egy példa kapcsán világítunk rá a tisztán eljárásközpontú megközelítés gyenge pontjaira. A példában köröket kell ábrázolnunk, amelyekből esetleg igen sok is lehet. Egy kör meghatározásához legalább három adatra van szükség – a középpontjának (x, y) a koordinátája és a sugara (radius) éppen megteszi. Az adatok tárolására alkalmazhatunk rendezett hármasokat, például így:

```
circle = (11, 60, 8)
```

Ennek a módszernek az az egyik hátulütője, hogy nem tudhatjuk pontosan, hogy a hármas melyik tagja mit jelent. Gondolhattunk az `(x, y, radius)` hármasra, de éppúgy tárolhattuk az adatokat `(radius, x, y)` alakban is. További gondokat okozhat, hogy az elemeket csak a sorszámuk alapján érhetjük el. Ha ugyanis van két függvényünk – a `distance_from_origin(x, y)` és az `edge_distance_from_origin(x, y, radius)` –, amelyek körök adatait várják paraméterként, a sorozatkicsomagoló műveletet kell használnunk az elérésükhöz, valahogy így:

```
distance = distance_from_origin(*circle[:2])
distance = edge_distance_from_origin(*circle)
```

Itt mindkét esetben feltételeztük, hogy a rendezett hármasok `(x, y, radius)` alakban tartalmazzák a körök adatait. A sorrend ismeretének igényén és a kicsomagolás bonyodalmain túllendülhetünk, ha nevesített rendezett sorozatot használunk:

```
import collections
Circle = collections.namedtuple("Circle", "x y radius")
circle = Circle(13, 84, 9)
distance = distance_from_origin(circle.x, circle.y)
```

Így most már létrehozhatunk `Circle` rendezett hármasokat, amelyek jellemzőire a nevükkel hivatkozhatunk – így a függvényhívások érthetőbbé válnak. Sajnos a gondjainknak ezzel még nem szakad vége. Semmi sem tiltja meg például, hogy érvénytelen adatokkal hozzunk létre köröket, mint alább:

```
circle = Circle(33, 56, -5)
```

Természetesen semmi értelme egy negatív sugarú körnek, de a `circle` hármas mindenféle gond (kivétel kiváltása) nélkül létrejön – éppúgy, mintha a sugarat egy negatív értéket tartalmazó változóban adtuk volna át. A hiba csak abban az esetben válik nyilvánvalóvá, ha meghívjuk az `edge_distance_from_origin()` függvényt, sőt ez esetben is csak akkor, ha a kódban kiszűrjük a negatív sugárértékeket. Az objektumok létrehozásánál így nem tudjuk ellenőrizni az érvényességüket – talán ez a tisztán eljárásközpontú megközelítés igazi Achilles-sarka.

Ha azt szeretnénk, hogy a körök módosulhassanak, vagyis elmozdíthassuk őket a középpontjuk áthelyezésével, illetve átméretezhessük a sugaruk módosításával, a `collections.namedtuple._replace()` tagfüggvényt alkalmazhatjuk:

```
circle = circle._replace(radius=12)
```

A `Circle` objektumok létrehozásánál látottakhoz hasonlóan itt sem tiltja semmi az érvénytelen adatok használatát (és figyelmeztetést sem kapunk).

Ha a körök adatai előreláthatóan gyakran módosulnak, a kényelmesebb munka kedvéért választhatunk valamilyen változtatható adattípust, például listát az adatok tárolásához:

```
circle = [36, 77, 8]
```

Ez továbbra sem véd meg az érvénytelen adatok használatától, ha pedig név szerint kívánjuk elérni az elemeket, a legtöbb, amit tehetünk, hogy meghatározunk néhány alkalmas állandót, mint a `circle[RADIUS] = 5` kifejezésben. A listák használatánál azonban újabb gondok merülnek fel – nyugodtan használhatjuk például a `circle.sort()` hívást! Szótárakat is alkalmazhatunk (például `circle = dict(x=36, y=77, radius=8)`), de itt sem kínálkozik semmilyen mód a sugár érvényes értékének biztosítására, és a nem megfelelő tagfüggvények hívásának megakadályozására.

Az objektumközpontú programozás alapelvei és alapfogalmai

Az előzőekből leszűrhető, hogy valójában a kört leíró adatok egységbe zárására van szükségünk, emellett biztosítanunk kell, hogy csak olyan tagfüggvényeket alkalmazhassunk rájuk, amelyek érvényes műveleteket végeznek rajtuk. Mindkét célunk elérhető egy egyedi `Circle` adattípussal. Létrehozásának módját bemutatjuk a későbbiekben, előbb azonban tisztáznunk kell néhány alapfogalmat. Ha a következőkben leírtak első pillantásra nehezen érthetőnek is tűnnek, ne aggódjunk, a példák majd mindent megvilágítanak.

Az *osztály*, a *típus* és az *adattípus* fogalmakat szinonimáknak tekintjük. A Pythonban létrehozott egyedi osztályok teljes mértékben beépülnek a nyelv szerkezetébe, vagyis ugyanúgy használhatjuk őket, mint a beépített adattípusokat. Osztályokkal már többször is találkozhattunk – ilyen volt például a `dict`, az `int` és az `str` is. Egy adott osztály példányára az *objektum*, illetve esetenként a *példány* elnevezéssel hivatkozunk. Így az `5` egy `int`, az `"oblong"` pedig egy `str` objektum.

A legtöbb osztály egyaránt tartalmaz adatokat, valamint tagfüggvényeket, amelyek ezekre az adatokra alkalmazhatók. Így például az `str` osztály egy Unicode-karakterláncot képes adatként tárolni, amelyre tagfüggvényeket – mint mondjuk az `str.upper` – alkalmazhatunk. Számos osztály további lehetőségeket is elérhetővé tesz – így például a `+` művelettel egymáshoz fűzhetünk két karakterláncot (vagy bármely más sorozatot), a beépített `len()` függvénnyel pedig megkaphatjuk egy sorozat hosszát. Ezeket a lehetőségeket *különleges tagfüggvények* biztosítják, amelyek annyiban térnek el a „rendes" tagfüggvényektől, hogy előre meghatározottak, emellett a nevük minden esetben két aláhúzással kezdődik és

végződik. Így ha olyan osztályt szeretnénk létrehozni, amely lehetővé teszi az összefűzést a `+` művelettel és a hossz megállapítását a `len()` függvénnyel, meg kell valósítanunk benne az `__add__()` és a `__len__()` különleges tagfüggvényeket. Mindebből ugyanakkor az is következik, hogy magunk nem írhatunk olyan tagfüggvényt, amelynek a neve két aláhúzással kezdődik és végződik – hacsak nem valamilyen előre meghatározott tagfüggvényről van szó, amely az adott osztályban alkalmazható. Ez elejét veszi annak, hogy a tagfüggvényeink nevei ütközzenek a Python későbbi változataiban esetleg megjelenő új különleges tagfüggvényekével.

Az objektumok többnyire rendelkeznek jellemzőkkel (attribute) – a tagfüggvények meghívható jellemzők, a többiek pedig az adatokat tárolják. Egy `complex` objektum például rendelkezik a `real` és az `imag` jellemzővel, valamint rengeteg tagfüggvénnyel – találhatunk köztük különleges tagfüggvényeket, mint az `__add__()` vagy a `__sub__()` (ezek teszik lehetővé a `+` és a `-` műveletek használatát), továbbá egyszerű tagfüggvényeket, mint a `conjugate()`. Az adatjellemzőket (amelyekre gyakran egyszerűen jellemzőként hivatkozunk) többnyire *példányváltozókkal* valósítjuk meg, amelyeknek a tartalma objektumonként eltérő lehet. A használatukra számos példát láthatunk majd, mint ahogy megtanuljuk azt is, hogy miként tegyük elérhetővé az adatjellemzőket tulajdonságként (property), amelynek az értékét a példányváltozókhoz hasonlóan olvashatjuk ki és állíthatjuk be, de az elérést a háttérben tagfüggvények biztosítják. Amint azt a későbbiekben láthatjuk majd, a tulajdonságok nagy szolgálatot tesznek az adatok érvényességének biztosításában.

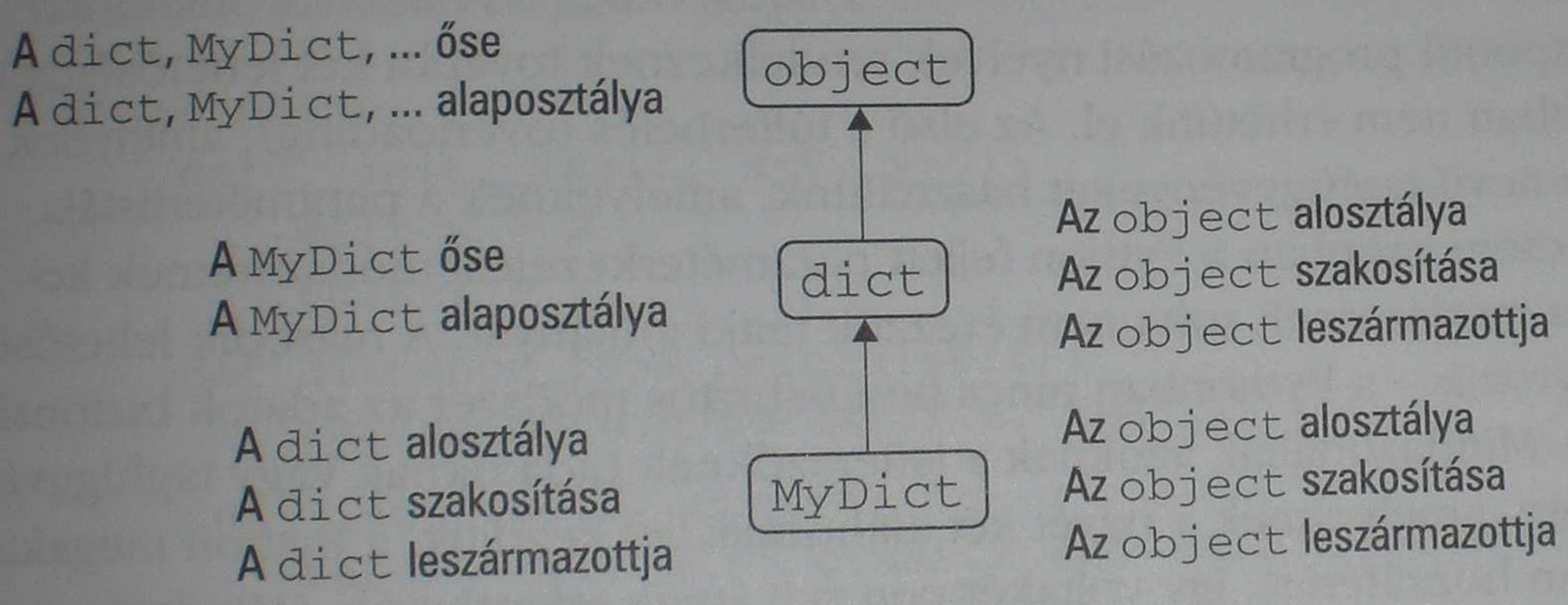
A tagfüggvényen (ami voltaképpen egy egyszerű függvény, azzal a kitétellel, hogy első paraméterként azt az objektumot várja, amelyen műveletet kell végeznie) belül számos, különböző típusú változót érhetünk el. Az objektum példányváltozóihoz úgy juthatunk hozzá, ha a nevüket magával a példánnyal minősítjük. A helyi változókat a tagfüggvényen belül hozhatjuk létre – ezeknél nincs szükség minősítésre. Az osztályváltozókat (amelyeket olykor statikus változóknak is neveznek) az osztálynévvel minősítve érhetjük el, míg a globális, vagyis modulszintű változókhoz minősítés nélkül juthatunk hozzá.

A Pythonnal foglalkozó irodalomban ismert a *névtér* fogalma is, amely a neveket objektumoknak felelteti meg. A modulok voltaképpen névterek – így az `import math` utasítást követően elérhetjük a `math` modul objektumait, ha a névterük nevével minősítjük őket (például: `math.pi` vagy `math.sin()`). Az osztályok és az objektumok szintén névterek: így a `z = complex(1, 2)` utasítással meghatározott objektum névterében két jellemzőt érhetünk el (ezek a `z.real` és a `z.imag`).

Az objektumközpontú megközelítés egyik legvonzóbb lehetősége, hogy ha rendelkezünk egy osztállyal (class), létrehozhatjuk egy *alosztályát* (subclass) – ezt *szakosításnak* vagy *specializációnak* is hívják –, vagyis egy új osztályt, amely örökli az eredeti osztály összes jellemzőjét (adatait és tagfüggvényeit), az eredményben pedig új tagfüggvényeket helyezhetünk el, egyesekkel esetleg régebbieket felváltva, illetve újabb példányváltozókat vehe-

tünk használatba. Ez a lehetőség rendelkezésre áll minden Python-osztályban – legyen az beépített, az alapkönyvtárból származó vagy egyedi osztály*. Az alosztályok használatának lehetősége az objektumközpontú megközelítés egyik legnagyobb vívmánya, hiszen így egy jól bevált régi osztályra építve új adatjellemzők és tagfüggvények hozzáadásával egyszerűen és gyorsan létrehozhatjuk a saját változatunkat. Ráadásul az új osztályunk objektumait nyugodt lélekkel átadhatjuk olyan függvényeknek és tagfüggvényeknek, amelyek az eredeti osztály példányait igénylik.

Azt az osztályt, amelyből az új osztályunkat – közvetlenül vagy közvetve – származtatjuk, az új osztály *ősének* vagy *alaposztályának* nevezzük, maga az új osztály pedig a *származtatott osztály* vagy más néven *alosztály*. A Pythonban minden beépített, illetve könyvtári osztály, továbbá minden egyedi osztály közvetve vagy közvetlenül az `object` nevű ősosztálytól származik. Az öröklés fogalmait áttekinthetjük a 6.1. ábrán.



6.1. ábra
Az objektumok öröklődésének fogalomrendszere

Bármely tagfüggvény felülírható egy adott alosztályban, vagyis elkészíthetjük egy új megvalósítását – ez hasonló a Javában látottakhoz (eltekintve a „final" tagfüggvényektől).** Ha rendelkezünk egy a `MyDict` osztályba tartozó objektummal (amely a `dict` leszármazottja), majd meghívunk egy olyan tagfüggvényt, amelyet meghatároztak mind a `dict`, mind a `MyDict` osztályban, a Python okosan a `MyDict`-változat használata mellett dönt – ez a *dinamikus tagfüggvénykötés* vagy más néven *többalakúság*. Amennyiben az ős tagfüggvényére van szükségünk az átírt tagfüggvény belsejében, a beépített `super()` függvény segítségével ezt is elérhetjük.

A Python a típusokat *fekete dobozként* kezeli – vagyis ha bizonyos tagfüggvényeket meg szeretnénk hívni egy objektumra, a rendszert nem érdekli, hogy milyen osztályba tartozik, csak azt vizsgálja, hogy rendelkezik-e a kívánt tagfüggvényekkel. Az előző fejezetben láthattuk, hogy amikor egy fájlobjektumra volt szükségünk, hozzájuthattunk egyhez

---

* Egyes C nyelven megvalósított osztályokból nem származtathatunk alosztályt – erre a korlátozásra nézve az adott osztály leírása ad tájékoztatást.

** A C++ szóhasználata szerint a Python összes tagfüggvénye virtuális.

a beépített `open()` függvénnyel – de létrehozhattunk helyette egy `io.StringIO` objektumot is, hiszen ezek ugyanazzal az API-val (alkalmazásprogramozási felülettel), vagyis ugyanolyan tagfüggvényekkel rendelkeznek, mint azok a fájlobjektumok, amelyeket az `open()` szöveges módban visszaad.

Az örökléssel „megfelel" („is-a") típusú kapcsolatokat ábrázolhatunk, ahol egy adott osztály objektumai lényegében megfelelnek egy másik osztály példányainak, mindössze néhány további adattal és tagfüggvénnyel rendelkeznek. Az objektumok között „birtokol" típusú kapcsolatok is megjelenhetnek – erre szolgál az *összetétel* (aggregation vagy composition), amelyben az egyik osztály példányváltozóként tartalmazza más osztályok objektumait. A Pythonban minden osztály részt vesz az öröklésben, lévén az `object` leszármazottja, és a legtöbbjük az összetételt is alkalmazza, hiszen az osztályok általában különféle típusú példányváltozókkal rendelkeznek.

Egyes objektumközpontú programozási nyelvek rendelkeznek további két lehetőséggel, amelyeket a Pythonban nem érhetünk el. Az első a túlterhelés (overloading), amelynek során olyan azonos nevű tagfüggvényeket használunk, amelyeknek a paraméterlistája különbözik. Szerencsére azonban a Python fejlett paraméterkezelési módszereinek köszönhetően ennek a módszernek soha nem érezzük majd a hiányát. A második lehetőség a *hozzáférés-szabályozás* – a Pythonban nincs bombabiztos módszer az adatok biztonságának megőrzésére. Mindazonáltal, azoknak a jellemzőknek (adatoknak vagy tagfüggvényeknek) az esetében, amelyeknek a nevét két aláhúzásjellel kezdjük, a Python megakadályozza az akaratlan hozzáférést, így voltaképpen privátnak tekinthetők. (Mindezt a névkeverés segítségével éri el, amelyre majd a 8. fejezetben láthatunk példát.)

Az egyedi osztályok elnevezésénél az egyedi modulokhoz hasonlóan járunk el: nagybetűvel kezdjük a nevüket. Annyi osztályt hozhatunk létre, amennyit csak akarunk, akár közvetlenül a programban, akár modulokban – az osztályok nevei nem kell, hogy egyezzenek a modulokéival, emellett egy modulban tetszőleges számú osztálymeghatározást elhelyezhetünk.

Láthattunk néhány problémát, amelyekre az osztályok megoldást adhatnak, megismerkedtünk az ebben a témakörben használatos elnevezésekkel, sőt némi háttértudásra is szert tettünk – nincs más hátra, mint hogy létrehozzunk néhány egyedi osztályt!

Egyedi osztályok

Egyedi osztályokat már a korábbi fejezetekben is hoztunk létre: ezek voltak az egyedi kivételek. Lássuk most az osztályok létrehozásának két új alakját:

```
class osztályNév:
    blokk

class osztályNév(alaposztályok):
    blokk
```

Mivel a kivételek alosztályainak létrehozásakor nem adtunk hozzájuk új jellemzőket (sem adatokat, sem tagfüggvényeket), a `blokk` helyére a `pass` kulcsszót írtuk (ami azt jelenti, hogy nincs semmi újdonság), és mivel a blokk egyetlen utasításból állt, a `class` utasítással egy sorba írhattuk. Érdemes megjegyezni, hogy a `def`-hez hasonlóan a `class` is utasítás, így osztályokat szükség esetén dinamikusan is létrehozhatunk. Az osztály tagfüggvényeit az osztály blokkján belül hozzuk létre `def` utasítások segítségével. Ahhoz pedig, hogy példányokat hozzunk létre az osztályból, magát az osztályt kell meghívnunk a megfelelő paraméterekkel. Így az `x = complex(4, 8)` egy komplex számot hoz létre, és így `x` egy erre a számra hivatkozó objektum lesz.

Jellemzők és tagfüggvények

Kezdjük egy igen egyszerű osztállyal – ez a `Point`, amely az `(x, y)` koordinátapárt tárolja. Az osztály a `Shape.py` fájlban található, teljes megvalósítása pedig (a dokumentációs karakterlánc kivételével) alább látható:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

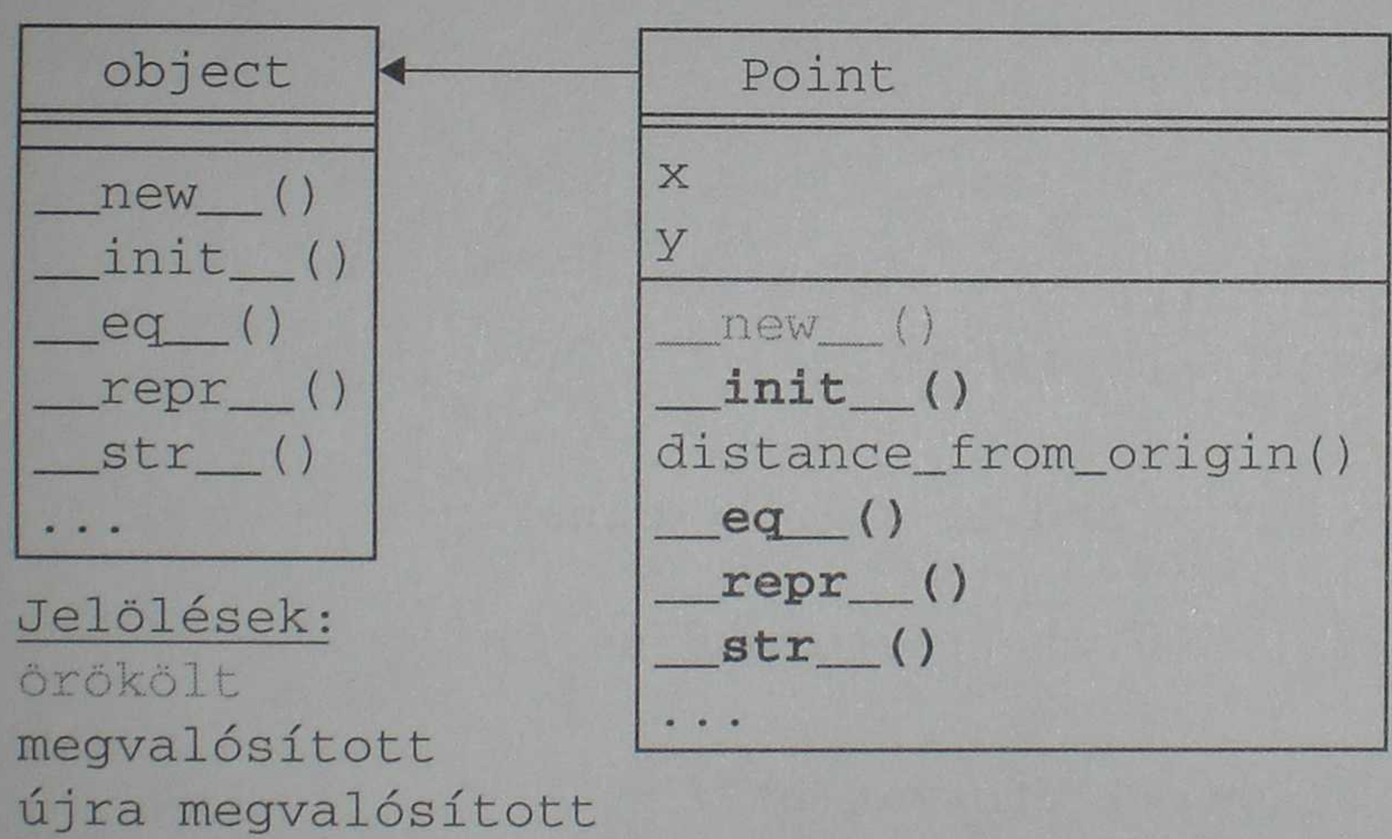
Mivel nem adtunk meg alaposztályt, a `Point` az `object` közvetlen leszármazottja, vagyis meghatározhattuk volna `class Point(object)` alakban is. Mielőtt rátérnénk a tagfüggvényei bemutatására, lássunk néhány példát a használatára:

```
import Shape
a = Shape.Point()
repr(a)                     # az eredmény: 'Point(0, 0)'
b = Shape.Point(3, 4)
str(b)                      # az eredmény: '(3, 4)'
b.distance_from_origin()    # az eredmény: 5.0
b.x = -19
str(b)                      # az eredmény: '(-19, 4)'
a == b, a != b              # az eredmény: (False, True)
```

A `Point` osztály két adatjellemzővel – `self.x` és `self.y` –, valamint öt tagfüggvénnyel rendelkezik (nem számítva az örökölteket), amelyek közül négy különleges tagfüggvény (lásd a 6.2. ábrát). Miután beemeltük a `Shape` modult, a `Point` osztály más társaihoz hasonlóan használatba vehető. Az adatjellemzőket közvetlenül elérhetjük (vagyis `y = a.y` alakban), és

az osztály nagyszerűen igazodik a nyelv más osztályaihoz, hiszen támogatja az egyenlőségvizsgálatot (==), továbbá ábrázoló és egyszerű karakterláncot is biztosít az objektumok leírására. A nem egyenlő (!=) műveletet már maga a Python adja, az egyenlőség műveletre építve. (Lehetőségünk van arra is, hogy mindkét műveletet magunk határozzuk meg, amennyiben nem szeretnénk, ha pontosan egymás ellentettjei lennének.)

A Python automatikusan megadja a tagfüggvények hívásának első paraméterét – ez egy objektumhivatkozás magára az objektumra (a C++-ban és a Javában ez a *this*). Mindazonáltal a tagfüggvények meghatározásakor ezt a paramétert is fel kell tüntetnünk a paraméterlistában – a neve közmegegyezés szerint `self`. Az objektum minden jellemzőjét (legyen szó adatról vagy tagfüggvényről) a `self`-fel kell minősítenünk. Ez némiképp több gépelést jelent, mint egyéb nyelvekben, de annyi előnye mindenképpen van, hogy így tökéletesen tisztán láthatunk: amennyiben egy jellemzőt a `self` kulcsszóval minősítünk, mindig tudhatjuk, hogy az aktuális objektum jellemzőjéről van szó.



6.2. ábra

A Point osztály öröklési hierarchiája

Ahhoz, hogy objektumot hozzunk létre, két lépésre van szükség. Először létre kell hoznunk egy „nyers", vagyis kezdőértékekkel nem rendelkező objektumot, majd biztosítanunk kell a számára a kezdőértékeket. Egyes programozási nyelvek (mint a C++ és a Java) egyesítik ezeket a lépéseket, a Python azonban elkülöníti őket. Amikor létrehozunk egy objektumot (például a `p = Shape.Point()` értékadással), a rendszer először a `__new__()` nevű különleges tagfüggvényt hívja meg az objektum létrehozásához, majd az `__init__()` segítségével ellátja kezdőértékekkel.

Szinte akármilyen Python-osztályt hozunk is létre, kizárólag az `__init__()` tagfüggvényt kell újra megvalósítanunk, ugyanis az `object.__new__()` az esetek túlnyomó többségében tökéletesen megteszi, és a rendszer automatikusan ezt hívja meg, amennyiben nem adunk meg mást helyette.

Alternatíva: FuzzyBool ☞ 247

(A fejezet egy későbbi részében bemutatunk egy igen ritka esetet, amikor valóban szükség van arra, hogy újra megvalósítsuk a `__new__()` tagfüggvényt.) Az objektumközpontú programozás egyik fontos vívmánya ez: ha az alaposztály tagfüggvénye alkalmas egy feladat ellátására, a leszármazottban nem kell újra megvalósítanunk. Mindezt az teszi lehetővé, hogy ha meghívunk egy objektumra egy tagfüggvényt, amelynek a megvalósítását a rendszer nem találja az objektum osztályában, automatikusan sorra veszi ennek őseit, majd az ősök őseit, és így tovább, amíg rá nem akad a kívánt tagfüggvényre – ha pedig végképp nem jár sikerrel, `AttributeError` kivételt vált ki.

Ha például elkezdjük végrehajtani a `p = Shape.Point()` utasítást, a Python először a `Point.__new__()` tagfüggvényt keresi meg. Mivel ezt nem valósítottuk meg újra, a `Point` alaposztályaiban folytatja a keresést. Esetünkben a `Point` egyetlen őse az `object`, így a rendszer meghívja az `object.__new__()` tagfüggvényt, amely egy nyers, kezdőértékek nélküli objektumot hoz létre. A Python ez után az `__init__()` tagfüggvény keresésébe kezd, amelyet magunk valósítottunk meg, így a `Point.__init__()` hívással célt ér. Végezetül már csak arra van szükség, hogy a `p` változóban az újonnan létrehozott és kezdőértékekkel ellátott `Point` objektum hivatkozását tárolja.

Rövidségük és a felejtéshez éppen elegendő pár oldalnyi távolság miatt az egyes tagfüggvényeket újfent bemutatjuk majd a részletes tárgyalásuk előtt.

```
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
```

A `self.x` és a `self.y` példányváltozókat az `__init__()` tagfüggvényben hozzuk létre, az `x` és az `y` paramétereket rendelve hozzájuk. Mivel a Python ezt a tagfüggvényt találja meg az új `Point` objektumok létrehozásakor, az `object.__init__()` hívására nem kerül sor – ha a rendszer megtalálta, amit keresett, nem folytatja tovább a kutakodást.

Az objektumközpontú programozás igazi megszállottai a tagfüggvény kezdetén minden bizonnyal meghívnák az alaposztály `__init__()` tagfüggvényét a `super().__init__()` utasítással. Azoknak az osztályoknak az esetében, amelyek az `object` közvetlen leszármazottai, erre semmi szükség, és könyvünkben is csak akkor hivatkozunk az alaposztály tagfüggvényeire, ha erre feltétlenül szükség van – például olyan osztályok létrehozásánál, amelyeket kifejezetten alosztályok szülőinek szánunk, illetve olyanoknál, amelyek nem közvetlen leszármazottai az `object` osztálynak. Bizonyos fokig persze mindez csak programozási stílus kérdése – tökéletesen megérthető az is, ha valaki az egyedi osztályai `__init__()` tagfüggvényeinek elején mindig elhelyezi a `super().__init__()` hívást.

```
def distance_from_origin(self):
    return math.hypot(self.x, self.y)
```

Ez egy hagyományos tagfüggvény, amely némi számítást végez az objektum példányváltozóiból kiindulva. A tagfüggvények esetében gyakori, hogy igen rövidek, és csak azt az objektumot várják paraméterként, amelyre meghívták őket, hiszen a működésükhöz szükséges adatok többnyire elérhetők az objektumon belül.

```
def __eq__(self, other):
    return self.x == other.x and self.y == other.y
```

A tagfüggvények neve általában nem kezdődhet és végződhet két aláhúzással – hacsak nem előre meghatározott különleges tagfüggvényről van szó. A Python minden összehasonlító művelethez biztosít ilyen különleges tagfüggvényeket (lásd a 6.1. táblázatban).

Az egyedi osztályok példányai alapértelmezés szerint támogatják az `==` műveletet, és az összehasonlítás mindig `False` eredményt ad. Ezt a viselkedést felülírhatjuk, ha újra megvalósítjuk az `__eq__()` tagfüggvényt – ezt tettük a fentiekben. Ha megvalósítjuk az `__eq__()` tagfüggvényt, a Python automatikusan biztosítja számunkra a `__ne__()` nevezetű ellentettpárját (vagyis a „nem egyenlő" műveletet), amennyiben mi magunk nem gondoskodunk a megvalósításáról.

Az egyedi osztályok példányai alapértelmezés szerint kivonatolhatók (hashable), így meghívhatjuk rájuk a `hash()` függvényt, továbbá szerepelhetnek szótárakban és halmazokban. A kivonatolhatóság végét jelenti azonban, ha újra megvalósítjuk az `__eq__()` tagfüggvényt – erre a helyzetre ad megoldást a később tárgyalt `FuzzyBool` osztály.

`FuzzyBool` ☞ 247

Ennek a különleges tagfüggvénynek a megvalósításával immár összehasonlíthatunk két `Point` objektumot, ha azonban egy `Point` objektumot egy másik típusú – például `int` – objektummal próbálunk összevetni, `AttributeError` kivételhez jutunk (hiszen az `int` értékek nem rendelkeznek `x` jellemzővel). Másrészről – köszönhetően a Python feketedoboz-szemléletének – a `Point` objektumokat összehasonlíthatjuk egyéb objektumokkal, amelyek rendelkeznek `x` jellemzővel, jóllehet az esetek többségében meglepő eredményt kapunk.

6.1. táblázat *Az összehasonlítás különleges tagfüggvényei*

Különleges tagfüggvény	Használat	Leírás
`__lt__(self, other)`	`x < y`	Az eredmény `True`, ha `x` kisebb, mint `y`.
`__le__(self, other)`	`x <= y`	Az eredmény `True`, ha `x` kisebb, mint `y`, vagy egyenlő azzal.
`__eq__(self, other)`	`x == y`	Az eredmény `True`, ha `x` egyenlő `y`-nal.
`__ne__(self, other)`	`x != y`	Az eredmény `True`, ha `x` nem egyenlő `y`-nal.
`__ge__(self, other)`	`x >= y`	Az eredmény `True`, ha `x` nagyobb, mint `y`, vagy egyenlő azzal.
`__gt__(self, other)`	`x > y`	Az eredmény `True`, ha `x` nagyobb, mint `y`.

Az értelmetlen összehasonlítások elkerülésére tehetünk néhány óvintézkedést. Alkalmazhatunk egy feltételezést – az `assert isinstance(other, Point)` éppen megteszi. Kiválthatunk egy `TypeError` kivételt is, jelezve, hogy az adott két típus összehasonlítását nem támogatjuk: `if not isinstance(other, Point): raise TypeError()`. Végezetül, a harmadik (és a leginkább „pythonos") módszer a következő kódrészlet használata: `if not isinstance(other, Point): return NotImplemented`. Ez esetben a `NotImplemented` visszatérési érték megjelenése esetén a Python megkísérli az `other.__eq__(self)` hívását – hátha a másik típus támogatja az összehasonlítást. Ha pedig nincs ilyen tagfüggvény, illetve itt is a `NotImplemented` értéket kapjuk vissza, a rendszer feladja, és kiváltja a `TypeError` kivételt. (Fontos megjegyeznünk, hogy a 6.1. táblázatban felsorolt különleges tagfüggvények esetében kizárólag az újra megvalósított változatok adhatják vissza a `NotImplemented` értéket.)

A beépített `isinstance()` függvény egy objektumot és egy osztályt (vagy osztályok rendezett sorozatát) várja, és akkor ad vissza `True` értéket, ha ez az objektum adott osztályból (illetve a sorozat egyik osztályából) való, illetve az osztály (vagy a sorozat egyik osztálya) ősének példánya.

```
def __repr__(self):
    return "Point({0.x!r}, {0.y!r})".format(self)
```

str.format() ☞ 75

A beépített `repr()` függvény meghívja az adott objektum `__repr__()` nevű különleges tagfüggvényét, és visszaadja az eredményt. A kapott karakterlánc kétféle típusba eshet. Az első esetén az eredményre a beépített `eval()` függvényt alkalmazva visszakaphatjuk a `repr()` számára megadott objektumot, míg a második típusnál ez nem járható út (a későbbiekben példát is láthatunk erre). Lássuk, hogyan kaphatunk egy `Point` objektumból karakterláncot, majd ebből ismét `Point` objektumot:

```
p = Shape.Point(3, 9)
repr(p)                                    # az eredmény: 'Point(3, 9)'
q = eval(p.__module__ + "." + repr(p))
repr(q)                                    # az eredmény: 'Point(3, 9)'
```

import ☞ 32, 192

Az `eval()` használata során meg kell adnunk a modul nevét, amennyiben a korábbiakban az `import Shape` utasítást alkalmaztuk. (Minderre nincs szükség, ha a betöltést másképp, például a `from Shape import Point` alakban végezzük el.) A Python minden objektumhoz biztosít néhány privát jellemzőt – ezek egyike a `__module__`, amely egy karakterlánccal megadja az objektumhoz tartozó modul nevét. Esetünkben ez a `"Shape"`.

Dinamikus kódvégrehajtás ☞ 349

A kódrészlet végén két `Point` objektum – a `p` és a `q` – áll előttünk, amelyek jellemzői megegyeznek, így az összehasonlítás során egyenlőnek bizonyulnak. Az `eval()` a paraméterként megadott karakterlánc végrehajtásának eredményét adja vissza – fontos tehát, hogy ez a karakterlánc érvényes Python-utasítás legyen.

```
    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

A beépített `str()` függvény működése hasonló a `repr()` függvénynél látottakhoz, azzal a különbséggel, hogy az objektum `__str__()` különleges tagfüggvényét hívja meg. Az eredmény a fejlesztők szándékai szerint emberi értelmezésre alkalmas alak – ugyanakkor éppen ezért nincs értelme az `eval()` függvénynek átadnunk. Előző példánk változóit véve ismét alapul, az `str(p)` (és hasonlóképpen az `str(q)`) eredményként a `'(3, 9)'` karakterláncot adja vissza.

Részletesen bemutattuk az aprócska `Point` osztályt – emellett pedig megismerkedtünk számos olyan háttérfolyamattal, amelyekről jó, ha tudunk, de egyébiránt ritkán kerülnek a szemünk elé. A `Point` osztály egy `(x, y)` koordinátapár tárolására alkalmas. Ez az adategység fontos részét képezi a körök leírásának, amint azt a fejezet elején láthattuk. A következőkben megfigyelhetjük, hogyan hozhatjuk létre egyedi `Circle` osztályunkat a `Point` leszármazottjaként, így nem újra megírnunk az `x` és `y` jellemzők, valamint a `distance_from_origin()` tagfüggvény kódját.

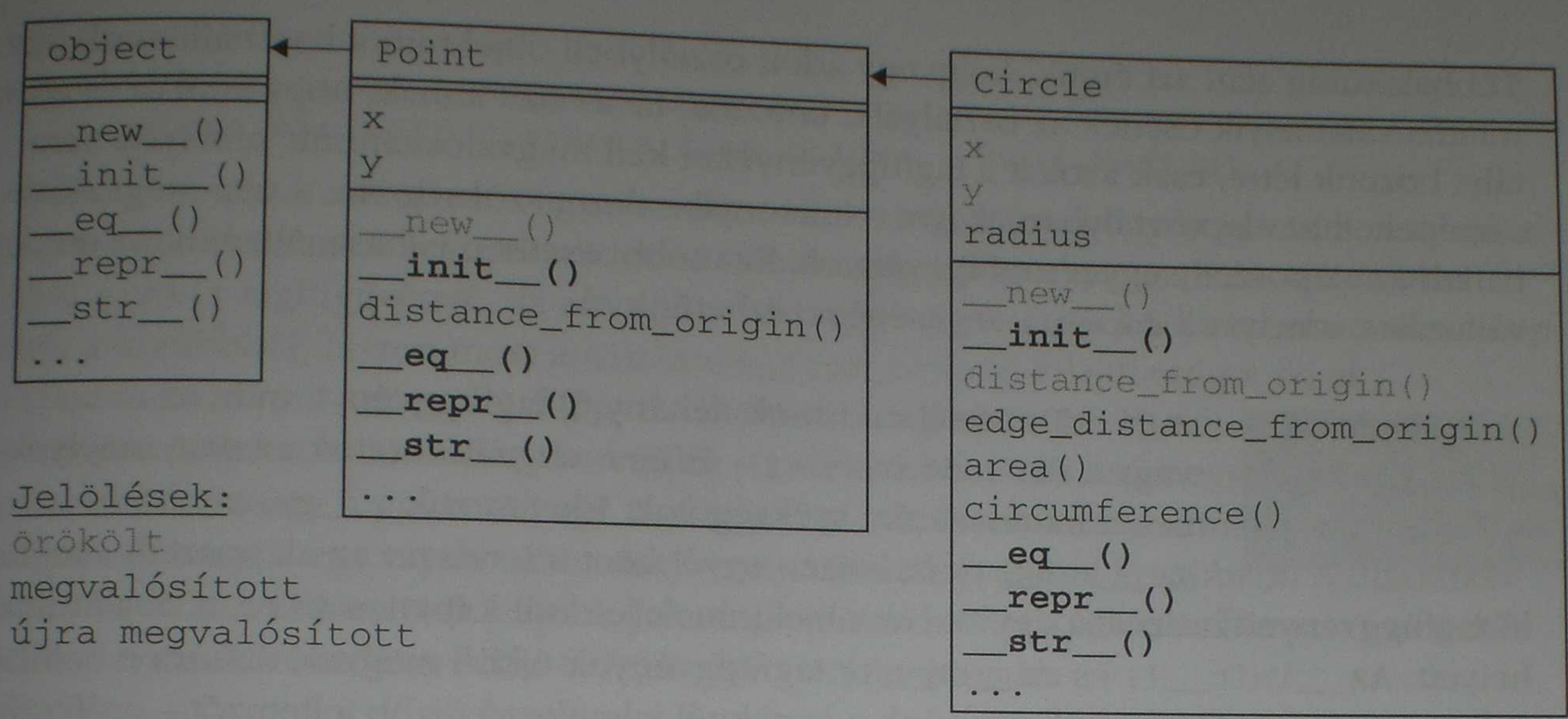
Öröklés és többalakúság

A `Circle` a `Point` osztályra épül, méghozzá az öröklés révén. Ősét egyetlen adatjellemzővel (`radius`) és három tagfüggvénnyel egészíti ki, emellett újra megvalósítja a `Point` néhány tagfüggvényét. Íme az osztály teljes meghatározása:

```
class Circle(Point):
    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius
    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin() - self.radius)
    def area(self):
        return math.pi * (self.radius ** 2)
    def circumference(self):
        return 2 * math.pi * self.radius
    def __eq__(self, other):
        return self.radius == other.radius and super().__eq__(other)
    def __repr__(self):
        return "Circle({0.radius!r}, {0.x!r}, {0.y!r})".format(self)
    def __str__(self):
        return repr(self)
```

Az örökléshez nem kell mást tennünk, mint felsorolni a kívánt őst (vagy ősöket) a `class` utasítás sorában.* Esetünkben a `Point` volt az alaposztály – a `Circle` öröklési hierarchiáját a 6.3. ábrán figyelhetjük meg.

* A többszörös öröklés, az elvont alaptípusok és más haladó objektumközpontú módszerek használatának tárgyalására a 8. fejezetben kerül sor.



6.3. ábra

A Circle osztály öröklési hierarchiája

Az `__init__()` tagfüggvény törzsében a `super()` segítségével meghívjuk az alaposztály `__init__()` tagfüggvényét – ez létrehozza és kezdőértékkel látja el a `self.x` és a `self.y` jellemzőket. Az osztály felhasználói nyugodtan megadhatnak érvénytelen sugárértékeket (mint például a `-2`) – a későbbiekben megmutatjuk, hogyan lehetünk úrrá ezen a helyzeten, ha a jellemzők sérülékenységét csökkentjük a tulajdonságok révén.

Az `area()` és a `circumference()` tagfüggvények kódja magától értetődő.
Az `edge_distance_from_origin()` számítása részeként meghívja a `distance_from_origin()` tagfüggvényt. Mivel ehhez a `Circle` osztály nem szolgál megvalósítással, a Python megtalálja a `Point` osztályban megadott alakot, és ezt használja fel. Az `__eq__()` tagfüggvényt ugyanakkor újra megvalósítjuk a `Circle` osztályban. A kódja összehasonlítja a két kör sugarát, és amennyiben ezek megegyeznek, a `super()` segítségével meghívja a szülőosztály `__eq__()` tagfüggvényét. Ha nem használtuk volna a `super()` tagfüggvényt, végtelen önhivatkozásba kerültünk volna, hiszen a `Circle.__eq__()` saját magát hívná meg. Érdemes észrevennünk, hogy a `super()` hívásánál nem kellett megadnunk a `self` paramétert, ugyanis a Python automatikusan megteszi ezt helyettünk.

Lássunk pár példát az osztály használatára:

```
p = Shape.Point(28, 45)
c = Shape.Circle(5, 28, 45)
p.distance_from_origin()          # az eredmény: 53.0
c.distance_from_origin()          # az eredmény: 53.0
```

A `distance_from_origin()` tagfüggvényt egyaránt meghívhatjuk `Point` és `Circle` objektumokra, hiszen az utóbbiak szerepelhetnek az előbbiek helyén.

Többalakúság alatt azt értjük, hogy egy adott osztálybeli objektumot használhatunk úgy, mintha valamelyik ősének az osztályába tartozna. Ez az oka annak, hogy amikor új alosztályt hozunk létre, csak azokat a tagfüggvényeket kell megvalósítanunk, amelyek nem szerepeltek az alaposztályban, illetve – amennyiben erre szükség van – újra megvalósíthatjuk az alaposztály egyes tagfüggvényeit. Ez utóbbi esetben is használhatjuk az eredeti változatot, amelyet a `super()` segítségével érhetünk el.

Sekély és mély másolás 143 ☞

A `Circle` esetében írtunk néhány új tagfüggvényt, mint az `area()` vagy a `circumference()`, és újra megvalósítottuk azokat, amelyeknek a módosítására szükség volt. Ide tartozik a `__repr__()` és az `__str__()` is, hiszen egyébként a rendszer az alaposztály megfelelő tagfüggvényeit használja, így `Point` objektumok leírását kapjuk a `Circle` objektumok helyett. Az `__init__()` és az `__eq__()` tagfüggvények újbóli megvalósítására is szükség volt, ugyanis számításba kell vennünk a köröknél jelentkező újabb jellemzőt – erőfeszítéseink mérséklésére azonban segítségül hívtuk az alaposztály megfelelő tagfüggvényeit is.

A `Point` és a `Circle` osztályok lehetőségei kedvünk szerint bővíthetők. Írhatunk például különleges tagfüggvényeket az összehasonlításhoz, hogy rendezni tudjuk a `Point` és `Circle` objektumokat. Érdemes lehet a megfelelő tagfüggvénnyel lehetővé tennünk a `Point`, illetve `Circle` objektumok másolását, ugyanis a legtöbb Python-osztály nem tesz elérhetővé `copy()` tagfüggvényt (kivételként megemlítenénk a `dict.copy()` és a `set.copy()` példáját). Ha másolási lehetőségekre vágyunk, könnyen célt érhetünk, csak töltsük be a `copy` modult, és máris hozzájutunk a `copy.copy()` függvényhez. (A `copy.deepcopy()` használatára ez esetben nincs szükség, ugyanis a `Point` és a `Circle` objektumok kizárólag állandó értékű példányváltozókat tartalmaznak.)

Jellemzők elérésének szabályozása tulajdonságokkal

A korábbiakban láthattuk, hogy a `Point` osztály tartalmaz egy `distance_from_origin()` nevű tagfüggvényt, a `Circle` osztályban pedig elérhető az `area()`, a `circumference()`, valamint az `edge_distance_from_origin()` tagfüggvény is. Mindegyikük egy-egy `float` értéket ad vissza, így a felhasználók szempontjából lehetnének akár (csak olvasható) adatjellemzők is. A `ShapeAlt.py` programban létrehoztuk az osztályok eredetitől eltérő változatait, amelyekben a fenti tagfüggvényeket tulajdonságok formájában valósítottuk meg. Mindez lehetőséget ad az alábbi kód végrehajtására:

```
circle = Shape.Circle(5, 28, 45) # feltételezi: import ShapeAlt as Shape
circle.radius                    # az eredmény: 5
circle.edge_distance_from_origin # az eredmény: 48.0
```

Íme a `ShapeAlt.Circle` osztály area és `edge_distance_from_origin` tulajdonságainak kiolvasó tagfüggvényei:

```
@property
def area(self):
    return math.pi * (self.radius ** 2)
```

```
@property
def edge_distance_from_origin(self):
    return abs(self.distance_from_origin - self.radius)
```

Amennyiben csak a kiolvasó függvényeket adjuk meg, mint ahogy itt is tettük, a tulajdonságok csak olvashatók lesznek. Az `area` tulajdonság kódja ugyanaz, mint a korábban látott `area()` tagfüggvényé, az `edge_distance_from_origin` kódja azonban némiképp eltér a korábbitól, hiszen most a `distance_from_origin` tulajdonságot éri el a `distance_from_origin()` függvény helyett. A legfeltűnőbb eltérés mindkét esetben a *tulajdonságdíszítő* használatában rejlik – a díszítő nem más, mint egy függvény, amely paraméterként egy függvényt vagy tagfüggvényt vár, és visszaadja annak „díszített" változatát, vagyis egy olyan változatot, amely némiképp eltér a korábbitól. A díszítőket a nevük előtt feltüntetett @ jel jelzi. Jelenleg tekintsük őket a kódolás formai részének – a létrehozásuk lehetőségeiről a 8. fejezetben tanulunk majd.

A `property()` beépített díszítőfüggvény, amely négy paramétert vár: egy kiolvasó függvényt, egy beállító függvényt, egy törlő függvényt, valamint egy dokumentációs karakterláncot. Az `@property` használatával ugyanazt a hatást érjük el, mintha a `property()` függvényt egyetlen paraméterrel, a beolvasó függvénnyel hívtuk volna meg. Az `area` tulajdonságot így is létrehozhattuk volna:

```
def area(self):
    return math.pi * (self.radius ** 2)
area = property(area)
```

Ezt az alakot azonban ritkán használjuk, ugyanis a díszítővel rövidebb és tisztább kódot kapunk.

Az előzőekben megjegyeztük, hogy a `Circle` objektum `radius` jellemzőjén semmiféle érvényességvizsgálatot nem végzünk – ezen a helyzeten most változtathatunk, ha a jellemzőből tulajdonságot hozunk létre. Mindehhez nem kell megváltoztatnunk a `Circle.__init__()` tagfüggvényt, illetve semmilyen kódot, amely eléri a `Circle.radius` jellemzőt – minden változatlanul működik, azzal a különbséggel, hogy a rendszer ellenőrzi a `radius` értékének érvényességét, mihelyt beállítottuk.

A Python-programozók rendszerint tulajdonságokat alkalmaznak, nem pedig kifejezett kiolvasó és beállító függvényeket – mint például `getRadius()` vagy `setRadius()` –, mint más objektumközpontú nyelvekben. Mindennek az az oka, hogy itt igen egyszerűen tulajdonsággá alakíthatunk egy adatjellemzőt anélkül, hogy az osztály használatát bármilyen módon befolyásolnánk.

Ahhoz, hogy egy jellemzőt írható/olvasható tulajdonsággá alakítsunk, létre kell hoznunk egy privát jellemzőt, amelyben helyet kap maga az adat, valamint meg kell írnunk a megfelelő beállító és kiolvasó tagfüggvényeket. Íme a sugár kiolvasó és beállító tagfüggvényei a maguk teljes valójában, dokumentációs karakterláncokkal együtt:

```
    @property
    def radius(self):
        """The circle's radius

        >>> circle = Circle(-2)
        Traceback (most recent call last):
        ...
        AssertionError: radius must be nonzero and non-negative
        >>> circle = Circle(4)
        >>> circle.radius = -1
        Traceback (most recent call last):
        ...
        AssertionError: radius must be nonzero and non-negative
        >>> circle.radius = 6
        """
        return self.__radius

    @radius.setter
    def radius(self, radius):
        assert radius > 0, "radius must be nonzero and non-negative"
        self.__radius = radius
```

Az `assert` utasítás segítségével biztosítjuk, hogy a sugár ne lehessen nulla vagy negatív, az értékét pedig a `self.__radius` privát jellemzőben tároljuk. Figyeljük meg, hogy a kiolvasó és a beállító függvény (valamint a törlő függvény, amennyiben ilyenre is szükség van) neve megegyezik – a díszítők tesznek különbséget közöttük: ezek adnak nekik új neveket, hogy ne ütközzenek egymással.

A beállító függvény díszítője első pillantásra furcsának tűnhet. Minden általunk létrehozott tulajdonság rendelkezik `getter`, `setter` és `deleter` jellemzővel, így, mihelyt létrehoztuk a `radius` tulajdonságot az `@property` utasítással, a `radius.getter`, a `radius.setter` és a `radius.deleter` egy csapásra elérhetővé válik. A `radius.getter` jellemzőhöz a `@property` díszítő rendeli hozzá a kiolvasó függvényt, míg a másik kettőt a Python határozza meg. Az utóbbiak így nem tesznek semmit (vagyis nem írhatjuk vagy törölhetjük a tulajdonság tartalmát), hacsak nem alkalmazzuk őket díszítőként – ilyenkor önmagukat helyettesítik azzal a tagfüggvénnyel, amelyet díszítünk velük.

A `Circle` kezdeti értékadó tagfüggvénye, a `Circle.__init__()` tartalmazza a `self.radius = radius;` utasítást, amely meghívja a `radius` tulajdonság beállító tagfüggvényét, így ha az objektum létrehozásakor érvénytelen sugárértéket adunk meg, `AssertionError` kivételt kapunk. Hasonlóképpen, ha egy `Circle` objektum `radius` tulajdonságának érvénytelen értéket adunk meg, ismét a beállító függvény állja az utunkat, így kivételt kapunk. A dokumentációs karakterláncban megtalálhatjuk a megfelelő teszteket annak ellenőrzésére, hogy a kivétel az elvárt esetekben keletkezik-e.

Mind a `Point`, mind a `Circle` olyan adattípusok, amelyek elegendő lehetőséggel rendelkeznek ahhoz, hogy jó hasznukat vegyük. A legtöbb adattípus, amelyet a későbbiekben magunk hozunk létre, ehhez hasonló elvek mentén épül fel, de adódhatnak olyan esetek, amikor valóban mindenféle értelemben teljes adattípust kell készítenünk. Erre láthatunk példákat a következőkben.

Teljes és tökéletesen beágyazott adattípusok létrehozása

Teljes adattípus létrehozásánál alapvetően két lehetőség áll előttünk. Létrehozhatjuk a nulláról indulva – ilyenkor – jóllehet az osztály (hasonlóan a Python összes osztályához) az `object` leszármazottja – minden adatjellemzőt és tagfüggvényt nekünk kell meghatároznunk az egyetlen `__new__()` kivételével. A másik lehetőség, hogy az osztályt egy már létező adattípusból származtatjuk, amely hasonlít az elvárt típushoz. Ilyenkor a munkánk jobbára abból áll, hogy újra megvalósítjuk azokat a tagfüggvényeket, amelyeknek a működését meg szeretnénk változtatni, továbbá megszabadulunk azoktól, amelyekre nincs szükségünk a továbbiakban.

A következőkben először a nulláról indulva létrehozzuk a `FuzzyBool` adattípust, majd megfigyeljük azt is, hogy miként valósíthatjuk meg úgy, hogy az öröklés lehetőségét alkalmazzuk az erőfeszítéseink mérséklésére. A beépített `bool` típus két értéket (`True` és `False`) vehet fel, de a mesterséges intelligencia alkalmazásának bizonyos területein többállapotú logikai értékekre („fuzzy boolean") van szükség, vagyis az „igaz" és „hamis" között is találunk igazságállapotokat. A mi megvalósításainkban az igazságértékek ábrázolására lebegőpontos számokat használunk, ahol a 0.0 a `False`, az 1.0 pedig a `True` értéket jelöli. Ebben a rendszerben a 0.5 azt jelenti, hogy a logikai érték 50%-ban igaz, a 0.25 pedig 25%-os igazságtartalmat takar. Az alábbiakban bemutatunk néhány példát a típus használatára (ezek ugyanazt az eredményt adják mindkét megvalósítás esetében):

```
a = FuzzyBool.FuzzyBool(.875)
b = FuzzyBool.FuzzyBool(.25)
a >= b                                  # az eredmény: True
bool(a), bool(b)                        # az eredmény: (True, False)
~a                                      # az eredmény: FuzzyBool(0.125)
a & b                                   # az eredmény: FuzzyBool(0.25)
b |= FuzzyBool.FuzzyBool(.5)            # a b értéke most: FuzzyBool(0.5)
"a={0:.1%} b={1:.0%}".format(a, b)      # az eredmény: 'a=87.5% b=50%'
```

Szeretnénk, ha a `FuzzyBool` típus támogatna minden összehasonlító műveletet (<, <=, ==, !=, >=, >), valamint a három alapvető logikai műveletet, vagyis a `not` (~) az `and` (&) és az `or` (|) operátorokat. Mindemellett más logikai tagfüggvényeket is biztosítani szeretnénk, mint a `conjunction()` és a `disjunction()`, amelyek képesek tetszőleges számú `FuzzyBool` paraméter fogadására, és az eredményt egy `FuzzyBool` objektum alakjában adják vissza. Mindemellett az adattípus teljessé tételéhez lehetővé tesszük az átalakítást `bool`, `int`, `str` és `float` típusokká, és készítünk egy ábrázolási alakot is, amely az `eval()` segítségével kiértékelhető. Végezetül pedig elérjük azt is, hogy a `FuzzyBool` támogassa az `str.format()` formátumleírásait, a `FuzzyBool` objektumok használhatók legyenek

szótárak kulcsaiként és halmazok elemeiként, és állandó értékű típussá tesszük a `FuzzyBool`-t – ugyanakkor gondoskodunk róla, hogy az egyszerűbb használat érdekében a bővített értékadó műveletek (`&=` és `|=`) rendelkezésre álljanak.

A 6.1. táblázatban láthattuk az összehasonlítás különleges tagfüggvényeit, a 6.2. táblázat az alapvető különleges tagfüggvényeket sorolja fel, a 6.3. táblázat pedig a számműveletek különleges tagfüggvényeit mutatja be – ez utóbbiak közé tartoznak a bitenkénti műveletek (`~`, `&` és `|`), amelyeket a `FuzzyBool` objektumok logikai műveletként alkalmaznak, valamint a számtani műveletek, mint a `+` és a `-`, amelyeket a `FuzzyBool` osztály nem valósít meg, hiszen a használatuk értelmetlen lenne.

Adattípusok létrehozása nulláról

Ahhoz, hogy a nulláról indulva hozzuk létre a `FuzzyBool` típust, létre kell hoznunk egy adatjellemzőt, amely tárolja az objektum értékét, valamint az ennek kezeléséhez szükséges tagfüggvényeket. Íme a `class` utasítás sora, valamint az osztály kezdeti értékadó tagfüggvénye, ahogy a `FuzzyBool.py` fájlban látható:

```
class FuzzyBool:
    def __init__(self, value=0.0):
        self.__value = value if 0.0 <= value <= 1.0 else 0.0
```

A tulajdonságjellemzőt privát értékként határoztuk meg, hiszen állandó értékű `FuzzyBool` objektumokra van szükségünk, vagyis nem lenne jó, ha megengednénk a hozzáférést ehhez a jellemzőhöz. Ha pedig valamilyen, a megengedett tartományon kívül eső értéket adnánk meg, az objektum egy biztosan érvényes értéket (0.0, vagyis hamis) alkalmaz helyette. Ha visszaemlékszünk, az előzőekben a `ShapeAlt.Circle` esetében szigorúbb eljárást választottunk, és ha a `Circle` objektum a kezdeti értékadásnál érvénytelen sugárértékkel találkozott, kivételt váltott ki. A `FuzzyBool` családfáját a 6.4. ábrán tekinthetjük meg.

A legegyszerűbb logikai művelet a `not`, amelyet a bitenkénti ellentettműveletel (`~`) jelenítünk meg:

```
def __invert__(self):
    return FuzzyBool(1.0 - self.__value)
```

6.2. táblázat *Alapvető különleges tagfüggvények*

Különleges tagfüggvény	Használat	Leírás
`__bool__(self)`	`bool(x)`	Ha megadjuk, az `x` igazságértékét adja vissza; hasznos az `if x: ...` kifejezésekben.
`__format__(self, format_spec)`	`"{0}".format(x)`	Az `str.format()` támogatását biztosítja egyedi osztályoknál.
`__hash__(self)`	`hash(x)`	Ha megadjuk, az `x` alkalmazható szótárkulcsként vagy halmazok elemeként.

6.2. táblázat *Alapvető különleges tagfüggvények* *(folytatás)*

Különleges tagfüggvény	Használat	Leírás
`__init__(self, paraméterek)`	`x = X(paraméterek)`	A rendszer az objektum kezdeti értékadásánál hívja meg.
`__new__(cls, paraméterek)`	`x = X(paraméterek)`	A rendszer az objektum létrehozásánál hívja meg. `__new__()` ☞ 256, 399
`__repr__(self)`	`repr(x)`	Az `x` karakterlánc-ábrázolását adja vissza; ahol ez lehetséges, `eval(repr(x)) == x`
`__repr__(self)` `str.format()` 75 ☞	`ascii(x)`	Az `x` csak ASCII-karakterekből álló ábrázolását adja vissza.
`__str__(self)`	`str(x)`	Az `x` emberi értelmezésre alkalmas karakterlánc-ábrázolását adja vissza.

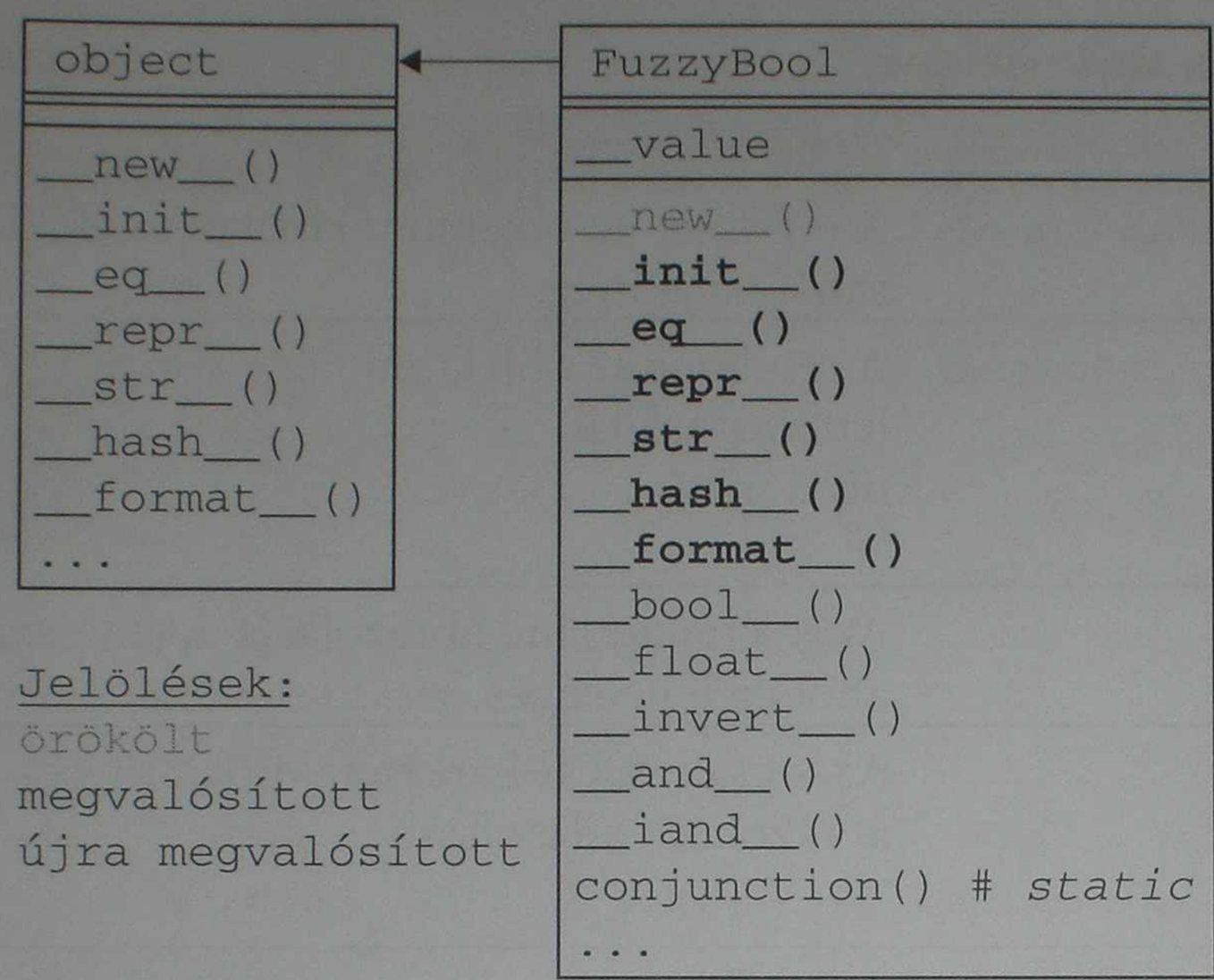
A __del__(self) különleges tagfüggvény

Figyelem! *A* `__del__(self)` *különleges tagfüggvényt akkor hívja meg a rendszer, amikor egy objektumot megsemmisít – legalábbis elméletben. A gyakorlatban azonban előfordulhat, hogy a tagfüggvény hívása soha nem kerül sorra, még a program befejezésekor sem. Ráadásul, ha a* `del x` *utasítással élünk, a rendszer mindössze az* `x` *nevű objektumhivatkozást törli, és eggyel csökkenti annak a számlálónak az értékét, amelyik a hivatkozott objektum hivatkozásainak számát tartja nyilván. A* `__del__()` *hívására csak akkor számíthatunk, ha ez a számláló eléri a 0 értéket – jóllehet a Python még ezt sem garantálja. Mindennek ismeretében érthető, hogy ritkán találkozunk a* `__del__()` *újra megvalósított változatával (könyvünk példáiban egyszer sem), továbbá éppen ezért nem érdemes az erőforrások felszabadítására használnunk, vagyis nem alkalmas fájlok bezárására, illetve hálózati vagy adatbázis-kapcsolatok bontására.*

A Python kétféle módot ad arra, hogy biztonságosan felszabadítsuk a lefoglalt erőforrásokat. Az egyik a már megismert `try...finally` *blokkok használata, amelyekkel ismét foglalkozunk a 7. fejezetben, a másik pedig egy környezeti objektum és a* `with` *utasítás együttes alkalmazása – róluk a 8. fejezetben ejtünk szót.*

A bitenkénti logikai `and` művelet (`&`) megvalósításáról az `__and__()` különleges tagfüggvény gondoskodik, az értékadó változatot (`&=`) pedig az `__iand__()` biztosítja:

```
def __and__(self, other):
    return FuzzyBool(min(self.__value, other.__value))
```



6.4. ábra
A FuzzyBool osztály öröklési hierarchiája

```
def __iand__(self, other):
    self.__value = min(self.__value, other.__value)
    return self
```

A bitenkénti `and` művelet egy új `FuzzyBool` objektumot ad vissza az aktuális, valamint egy másik objektumot alapul véve, míg az értékadó változat a privát értéket módosítja. Szigorúan véve ez nem felel meg az állandó értékű típusok viselkedésének, de mégis hasonlít több állandó értékű Python-típushoz – ilyen például az `int`, ahol a `+=` használatakor úgy tetszik, mintha a bal oldali tényezőt módosítanánk, holott a rendszer egy új `int` objektumhoz rendeli hozzá az összeadás eredményét. Esetünkben azonban erre az újrakötésre nincs is szükség, hiszen magát a `FuzzyBool` objektumot változtatjuk meg. Végül pedig, azért a `self` értéket adjuk vissza, hogy lehetővé tegyük a műveletek láncba rendezését.

Megvalósíthatjuk továbbá a `__rand__()` különleges tagfüggvényt is, amelyet a rendszer akkor hív meg, ha a `self` és az `other` különböző típusúak, és erre a párra nem valósítottuk meg az `__and__()` tagfüggvényt. `FuzzyBool` osztályunkhoz erre nincs szükség. A bináris műveletek különleges tagfüggvényei többnyire egyaránt rendelkeznek „i” és „r” változattal – az előbbi az értékadó alakhoz tartozik, míg az utóbbi megcseréli a tényezőket.

Nem mutatjuk be a bitenkénti `or` művelethez (`|`) tartozó `__or__()`, illetve értékadó változatához tartozó `__ior__()` különleges tagfüggvényeket, hiszen a kódjuk pontosan úgy fest, mint az `and` esetében, azzal a különbséggel, hogy ezúttal a `self` és az `other` értékeinek maximumát vesszük a minimum helyett.

```
def __repr__(self):
    return ("{0}({1})".format(self.__class__.__name__,
```

Létrehoztuk az objektum ábrázolási alakját, amelyet az `eval()` segítségével kiértékelhetünk. Így az `f = FuzzyBool.FuzzyBool(.75);` utasítást követően a `repr(f)` eredményeként a `'FuzzyBool(0.75)'` karakterláncot kapjuk.

Minden objektum rendelkezik néhány különleges jellemzővel, amelyeket a Python biztosít a számára – egyikük a `__class__`, amely az objektum osztályát adja meg. Minden osztálynál elérhető a privát `__name__` jellemző, amelyhez szintén automatikusan hozzájutunk. Ezeket a jellemzőket az ábrázolási alaknál az osztály nevének előállítására használtuk. Mindez azt jelenti, hogy ha a `FuzzyBool` osztályból alosztályokat származtatunk, hogy újabb tagfüggvényeket vehessünk használatba, az örökölt `__repr__()` tagfüggvényt nem kell újra megvalósítanunk, hiszen automatikusan az alosztály nevét olvassa ki.

```
def __str__(self):
    return str(self.__value)
```

Az egyszerű karakterlánc alakban az objektum lebegőpontos értékét adjuk vissza kartakterláncként formázva. A végtelen ciklus elkerülésére nem szükséges a `super()` hívást alkalmaznunk, mivel az `str()` tagfüggvényt a `self.__value` jellemzőre, nem pedig magára az objektumra hívjuk meg.

```
def __bool__(self):
    return self.__value > 0.5
def __int__(self):
    return round(self.__value)
def __float__(self):
    return self.__value
```

A `__bool__()` különleges tagfüggvény az objektumot logikai értékké alakítja, vagyis minden esetben `True` vagy `False` értéket ad vissza. Hasonlóképpen, az `__int__()` különleges tagfüggvény egésszé alakítja az objektumot. Ez utóbbi megvalósításához a beépített `round()` függvényt alkalmaztuk, mivel az `int()` egyszerűen csonkolja a megadott értéket (vagyis az 1.0 kivételével minden `FuzzyBool` objektum esetében 0-t adna vissza). A lebegőpontos számmá alakítás egyszerű feladat, hiszen az objektumban tárolt érték eleve lebegőpontos.

```
def __lt__(self, other):
    return self.__value < other.__value
def __eq__(self, other):
    return self.__value == other.__value
```

Ahhoz, hogy biztosítsuk az összehasonlító műveletek (`<`, `<=`, `==`, `!=`, `>=`, `>`) mindegyikét, legalább hármat meg kell valósítanunk közülük – a `<`, a `<=` és a `==` alapján azután a Python képes létrehozni a párjaikat, vagyis a `>`, a `>=` és a `!=` műveleteket. Közülük is csak kettőt mutattunk be, hiszen általában igencsak hasonlóak.*

* Valójában csak a fent bemutatott `__lt__()` és `__eq__()` tagfüggvényeket valósítottuk meg – a többiről a rendszer automatikusan gondoskodott. Ennek mikéntjéről a 8. fejezetben ejtünk szót.

6.3. táblázat *Számműveleti és bitenkénti különleges tagfüggvények*

Különleges tagfüggvény	Használat	Különleges tagfüggvény	Használat
`__abs__(self)`	`abs(x)`	`__complex__(self)`	`complex(x)`
`__float__(self)`	`float(x)`	`__int__(self)`	`int(x)`
`__index__(self)`	`bin(x) oct(x) hex(x)`	`__round__(self, *számjegyek*)`	`round(x, *számjegyek*)`
`__pos__(self)`	`+x`	`__neg__(self)`	`-x`
`__add__(self, other)`	`x + y`	`__sub__(self, other)`	`x - y`
`__iadd__(self, other)`	`x += y`	`__isub__(self, other)`	`x -= y`
`__radd__(self, other)`	`y + x`	`__rsub__(self, other)`	`y - x`
`__mul__(self, other)`	`x * y`	`__mod__(self, other)`	`x % y`
`__imul__(self, other)`	`x *= y`	`__imod__(self, other)`	`x %= y`
`__rmul__(self, other)`	`y * x`	`__rmod__(self, other)`	`y % x`
`__floordiv__(self, other)`	`x // y`	`__truediv__(self, other)`	`x / y`
`__ifloordiv__(self, other)`	`x //= y`	`__itruediv__(self, other)`	`x /= y`
`__rfloordiv__(self, other)`	`y // x`	`__rtruediv__(self, other)`	`y / x`
`__divmod__(self, other)`	`divmod(x, y)`	`__rdivmod__(self, other)`	`divmod(y, x)`
`__pow__(self, other)`	`x ** y`	`__and__(self, other)`	`x & y`
`__ipow__(self, other)`	`x **= y`	`__iand__(self, other)`	`x &= y`
`__rpow__(self, other)`	`y ** x`	`__rand__(self, other)`	`y & x`
`__xor__(self, other)`	`x ^ y`	`__or__(self, other)`	`x | y`
`__ixor__(self, other)`	`x ^= y`	`__ior__(self, other)`	`x |= y`
`__rxor__(self, other)`	`y ^ x`	`__ror__(self, other)`	`y | x`
`__lshift__(self, other)`	`x << y`	`__rshift__(self, other)`	`x >> y`
`__ilshift__(self, other)`	`x <<= y`	`__irshift__(self, other)`	`x >>= y`
`__rlshift__(self, other)`	`y << x`	`__rrshift__(self, other)`	`y >> x`
		`__invert__(self)`	`~x`

```
def __hash__(self):
    return hash(id(self))
```

Alapértelmezés szeint az egyedi osztályok példányai támogatják az `==` műveletet (amely minden esetben a `False` értékkel tér vissza), továbbá kivonatolhatók (vagyis alkalmazhatók szótárak kulcsaiként, és elhelyezhetők halmazokban). Ha azonban újra megvalósítjuk az `__eq__()` különleges tagfüggvényt az egyenlőség valóban megfelelő vizsgálatához, az objektumok a továbbiakban nem lesznek kivonatolhatók. A gondokat orvosolhatjuk egy `__hash__()` különleges tagfüggvénnyel – éppen ezt tettük itt is.

A Python kivonatoló függvényeket biztosít karakterláncok, számok, fagyasztott halmazok és más osztályok számára. Esetünkben egyszerűen a beépített `hash()` függvényt használtuk (amely képes kezelni bármilyen típust, amely rendelkezik a `__hash__()` különleges tagfüggvénnyel), átadva neki az objektum egyedi azonosítóját a kivonat (hash) kiszámításához. (Nem használhattuk a `self.__value` privát változót, hiszen ennek értéke módosulhat egy bővített értékadó műveletben, míg az objektum kivonatértéke sosem változhat.)

A beépített `id()` függvény visszaadja a számára paraméterként megadott objektumhoz tartozó egész értékű azonosítót. Ez leggyakrabban az objektum memóriabeli címe, de biztosak csak abban lehetünk, hogy minden objektum azonosítója különböző. A színfalak mögött az `is` művelet is az `id()` függvényt alkalmazza annak megállapítására, hogy két objektumhivatkozás vajon ugyanarra az objektumra utal-e.

```
def __format__(self, format_spec):
    return format(self.__value, format_spec)
```

A beépített `format()` függvényre igazán csak az osztálymeghatározásoknál van szükség., A függvény egyetlen objektumot, valamint esetleg egy formátumleírást vár, és az objektum megfelelően formázott alakjával tér vissza.

Minden beépített osztály rendelkezik megfelelő `__format__()` tagfüggvénnyel – ez esetben a `float.__format__()` tagfüggvényt alkalmazzuk, átadva neki az objektum lebegőpontos értékét és a formátumleírást. Ugyanezt az eredményt elérhettük volna az alábbi kóddal is:

```
def __format__(self, format_spec):
    return self.__value.__format__(format_spec)
```

A `format()` függvény használata egy kicsit kevesebb gépelést igényel, ráadásul az eredmény is átláthatóbb. Mindazonáltal semmi sem kötelez arra, hogy ezt a függvényt használjuk – létrehozhatjuk a saját formátumleíró nyelvünket is, és megvalósíthatjuk a `__format__()` tagfüggvényben. Mindössze arra kell ügyelnünk, hogy végül karakterláncot adjunk vissza.

```
@staticmethod
def conjunction(*fuzzies):
    return FuzzyBool(min([float(x) for x in fuzzies]))
```

A beépített `staticmethod()` függvény rendeltetése, hogy díszítőként használjuk – éppen ezt tettük a fentiekben. A statikus tagfüggvények mindössze annyiban térnek el korábban megismert társaiktól, hogy a Python *nem* adja át nekik automatikusan a `self` értéket vagy bármilyen más paramétert.

Az `&` műveletet láncban is használhatjuk, így ha adottak az `f`, `g` és `h` `FuzzyBool`-objektumok, konjunkciójukat megkaphatjuk az `f & g &` h alakban. Ez nagyszerűen működik, ha egy maroknyi objektumról van szó, ha azonban már egy tucatnyival dolgozunk, kérdések merülnek fel a művelet hatékonyságával kapcsolatban, hiszen minden `&` jel egy függvényhívást jelent. A fenti tagfüggvényt alkalmazva ugyanezt egyetlen függvényhívással elérhetjük, méghozzá így: `FuzzyBool.FuzzyBool.conjunction(f, g, h)`. Mindezt kicsit rövidebb alakban is elvégezhetjük egy `FuzzyBool`-példány megadásával, de mivel a statikus tagfüggvények nem jutnak hozzá a `self` paraméterhez, ezt magunknak kell átadnunk, mint az `f.conjunction(f, g, h)` esetében.

A megfelelő `disjunction()` függvényt nem mutatjuk be, hiszen társától csak nevében, illetve abban tér el, hogy a `min()` helyett a `max()` függvényt alkalmazza.

Egyes Python-programozók úgy tartják, hogy a statikus tagfüggvények használata nem elég „pythonos", és csak akkor folyamodnak hozzájuk, ha más nyelvekből (mint a C++ vagy a Java) kapott kódokat kell átalakítaniuk, illetve ha olyan tagfüggvényt írnak, amely nem használja a `self` paramétert. A Pythonban a statikus tagfüggvények helyett jobb, ha modulfüggvényeket írunk (lásd a következőkben), vagy osztálytagfüggvényeket hozunk létre (róluk a fejezet vége felé esik szó).

Hasonlóan, ha létrehozunk egy változót az osztály meghatározásán belül, de a tagfüggvényeken kívül, statikus változóhoz (osztályváltozóhoz) jutunk. Az állandók esetében kényelmesebb modulszintű globális változókat alkalmaznunk, de az osztályváltozók jó szolgálatot tehetnek, ha az osztály példányai között szeretnénk adatokat megosztani.

Végére értünk hát a `FuzzyBool` osztály megvalósításának, amelyet valóban a „nulláról" kezdtünk. Összesen 15 tagfüggvényt kellett újra megvalósítanunk (illetve 17-et, ha a mind a négy alapvető összehasonlító műveletet beleszámoljuk), továbbá két statikus tagfüggvényt. A következőkben az osztály egy újabb változatát hozzuk létre, ezúttal a `float` típusból származtatva. Itt mindössze nyolc tagfüggvény újbóli megvalósítására lesz szükség, továbbá két modulfüggvény megírására – emellett pedig 32 tagfüggvényt ki kell iktatnunk.

A legtöbb objektumközpontú nyelvben az öröklés során olyan új osztályokat hozunk létre az ősökből, amelyek rendelkeznek a szülő minden tagfüggvényével és jellemzőjével, és ezek mellé jönnek azok az új tagfüggvények és jellemzők, amelyeket újdonságként kívánunk bevezetni. A Python mindezt teljes mértékben támogatja, így írhatunk új tagfüggvényeket, és újra megvalósíthatjuk a régieket, módosítva a viselkedésüket. Mindemellett

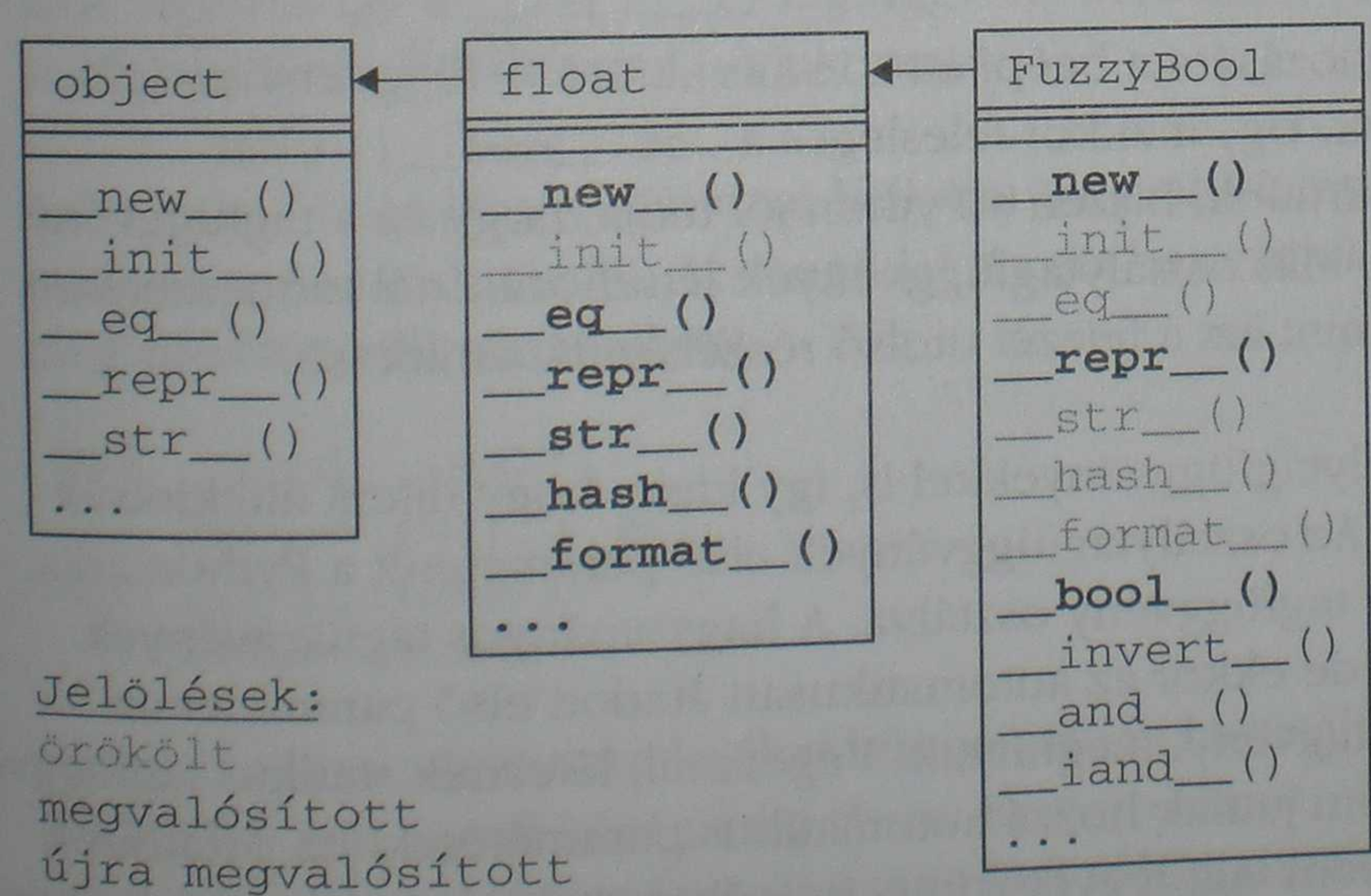
azonban lehetőségünk van egyes tagfüggvények kiiktatására is, így elérhetjük, hogy az új osztály úgy viselkedjen, mintha nem rendelkezne bizonyos örökölt tagfüggvényekkel. Az objektumközpontú programozás igazi elkötelezettjei persze idegenkedhetnek ettől, hiszen így elvész a többalakúság – mindazonáltal a Pythonban néha igen jól jöhet ez a lehetőség.

Adattípusok létrehozása más típusokból kiindulva

A `FuzzyBool` osztály következőkben bemutatott megvalósítását a `FuzzyBoolAlt.py` fájlban találhatjuk meg. Nyomban láthatjuk, hogy különbözik az előző változattól, hiszen a `conjunction()` és a `disjunction()` korábban statikus tagfüggvények voltak, most azonban modulfüggvények alakjában tesszük őket elérhetővé, valahogy így:

```
def conjunction(*fuzzies):
    return FuzzyBool(min(fuzzies))
```

A mindezt megvalósító kód számottevően egyszerűbb, mint a korábbiakban látott, ugyanis a `FuzzyBoolAlt.FuzzyBool` objektumok a `float` alosztályai, így közvetlenül használhatók `float` értékek helyett mindenféle átalakítás nélkül. (Az öröklési fát a 6.5. ábrán láthatjuk). A függvények elérése is egyszerűbb, mint előzőleg – korábban a modult és az osztályt kellett megadnunk (vagy egy példányt használnunk), az `import FuzzyBoolAlt` utasítást követően viszont ezúttal az egyszerűbb `FuzzyBoolAlt.conjunction()` alakot használhatjuk.



6.5. ábra

A FuzzyBool osztály újabb változatának öröklési hierarchiája

Íme a `FuzzyBool` osztály `class` sora és `__new__()` tagfüggvénye:

```
class FuzzyBool(float):
    def __new__(cls, value=0.0):
        return super().__new__(cls,
            value if 0.0 <= value <= 1.0 else 0.0)
```

Ha új osztályt hozunk létre, az többnyire változtatható, és a nyers, kezdőértékek nélküli objektum létrehozásához az `object.__new__()` tagfüggvényre támaszkodik. Az állandó értékű objektumok esetében azonban a létrehozást és a kezdeti értékadást egy lépésben kell elvégeznünk, hiszen ezek az objektumok a születésük után nem módosulhatnak.

A `__new__()` tagfüggvényt a rendszer még azelőtt meghívja, hogy egyetlen objektumot is létrehozott volna (hiszen éppen az objektumok létrehozása a `__new__()` feladata), így a `self` objektumot nem adhatjuk át neki, mivel az még nem is létezik. A `__new__()` valójában az osztálytagfüggvények közé tartozik, amelyek annyiban térnek el „rendes" társaiktól, hogy magára az osztályra hívjuk meg őket, nem pedig annak egy példányára, és a Python első paraméterként ezt az osztályt adja át a számukra. Az osztályt azonosító `cls` változónév mindössze a közmegegyezés miatt alakult így, hasonlóan az objektumokat azonosító `self` névhez.

Így ha az `f = FuzzyBool(0.7)` utasítást alkalmazzuk, a háttérben a Python a `FuzzyBool.__new__(FuzzyBool, 0.7)` hívással létrehoz egy új objektumot (nevezzük mondjuk `fuzzy`-nak), majd a `fuzzy.__init__()` hívásával elvégzi a további kezdeti értékadásokat, végül pedig a `fuzzy` egy objektumhivatkozását adja vissza – ez lesz az `f` értéke. A `__new__()` tagfüggvény feladatainak nagy részét az alaposztálybeli megvalósítása, vagyis az `object.__new__()` végzi el, nekünk csak azt kell biztosítanunk, hogy az érték a megfelelő tartományba essen.

Az osztálytagfüggvények létrehozására a beépített `classmethod()` függvény szolgál, amelyet díszítőként használunk. Ugyanakkor felesleges a `def __new__()` előtt az `@classmethod` kulcsszót kiírnunk, hiszen a Python jól tudja, hogy ez a tagfüggvény csak osztálytagfüggvény lehet. Más osztálytagfüggvények létrehozásánál természetesen szükségünk van a díszítőre, amint azt a fejezet utolsó részében láthatjuk is.

Megismerkedtünk hát az osztálytagfüggvényekkel is, így ideje, hogy újfent áttekintsük a Python tagfüggvénytípusait. Az osztálytagfüggvények első paraméterét a Python adja, ennek értéke nem más, mint a tagfüggvény osztálya. A hagyományos tagfüggvények esetében is hasonló a helyzet, de ekkor az automatikusan átadott első paraméter az az objektum, amelyikre a tagfüggvényt meghívtuk. Végezetül, léteznek statikus tagfüggvények is, amelyek egyáltalán nem jutnak hozzá automatikus paraméterekhez. Mindezek mellett természetesen mindhárom tagfüggvénytípus fogadhat általunk átadott paramétereket is (az előbbi két esetben a második, az utóbbiban pedig az első paramétertől kezdve).

```
def __invert__(self):
    return FuzzyBool(1.0 - float(self))
```

Ezzel a tagfüggvénnyel támogatjuk a bitenkénti `not` műveletet (~), éppúgy, mint a korábbiakban. Vegyük észre, hogy a `FuzzyBool` értékét tartalmazó privát változó helyett közvetlenül a `self` objektumot érjük el. Mindez annak köszönhető, hogy az osztályunkat

a `float` típusból származtattuk, így a `FuzzyBool` objektumok bárhol használhatók, ahol a `float` értékek – már amennyiben nem próbáljuk meghívni a `FuzzyBool` kiiktatott tagfüggvényeit.

```
def __and__(self, other):
    return FuzzyBool(min(self, other))
def __iand__(self, other):
    return FuzzyBool(min(self, other))
```

A fenti függvények szerkezete megegyezik a korábbiakban látottakkal (jóllehet a kód kissé eltér), és az `__invert__()` tagfüggvényhez hasonlóan a `self` és az `other` objektumokat úgy használhatjuk, mintha csak `float` típusúak lennének. Az `or` változatait nem mutatjuk be, hiszen csak a nevükben (`__or__()` és `__ior__()`), valamint abban térnek el, hogy a `max()`, nem pedig a `min()` függvényt alkalmazzák.

```
def __repr__(self):
    return ("{0}({1})".format(self.__class__.__name__,
                              super().__repr__()))
```

A `__repr__()` tagfüggvényt újra meg kell valósítanunk, hiszen alaposztálybeli változata, a `float.__repr__()` egyszerűen karakterláncként adja vissza az objektumban tárolt számot, míg nekünk az osztály nevére is szükségünk van, hogy kiértékelhessük az `eval()` segítségével. Az `str.format()` második paramétereként nem adhatjuk át a `self` objektumot, ugyanis így a `__repr__()` tagfüggvény hívásának végtelen ciklusába kerülnénk, ezért az alaposztálybeli változatot hívjuk meg.

Az `__str__()` tagfüggvényt nem kell újra megvalósítanunk, mivel az alaposztálybeli változat, a `float.__str__()` megfelel, és ezt használjuk a `FuzzyBool.__str__()` újra megvalósított változata helyett.

```
def __bool__(self):
    return self > 0.5
def __int__(self):
    return round(self)
```

Ha egy `float` objektumot logikai környezetben használunk, az értéke `False` lesz, ha eredetileg 0.0 volt, egyébként pedig `True`. Ez `FuzzyBool` objektumok esetén nem alkalmas viselkedés, így a tagfüggvényt újra meg kell valósítanunk. Hasonló a helyzet az `int()` függvénnyel is, hiszen az `int(self)` egyszerűen csonkolja a megadott értéket, így az 1.0 kivételével 0 eredményt kapunk. Ahhoz, hogy a megoldás megfeleljen az elvárásainknak, a `round()` függvényt vesszük használatba, amely 0.5 alatt a 0, felette pedig az 1 értéket adja vissza.

A `__hash__()`, a `__format__()`, valamint az összehasonlító műveleteket támogató tagfüggvényeket nem kellett újra megvalósítanunk, hiszen a `float` alaposztály által biztosított viselkedés tökéletesen megfelel a `FuzzyBool` objektumok esetében is.

Az újra megvalósított tagfüggvényekből összeáll a `FuzzyBool` osztály – és így sokkal kevesebb kódra van szükség, mint az előzőekben bemutatott változatnál. Van itt azonban egy aprócska gond: a `FuzzyBool` több mint 30 olyan tagfüggvényt örökölt, amelyeknek a használata nem egyeztethető össze a `FuzzyBool` objektum elvárt viselkedésével. Így például az egyszerű szám- vagy bitenkénti műveletek (+, -, *, /, <<, >> stb.) egyike sem alkalmazható értelmesen ebben a környezetben. Az alábbiakban láthatjuk, hogyan kezdhetünk hozzá az összeadás kiiktatásához:

```
def __add__(self, other):
    raise NotImplementedError()
```

Hasonló kódot kell írnunk az `__iadd__()` és az `__radd__()` tagfüggvények számára is, hogy az összeadás minden formáját megakadályozzuk. (Fontos tudnunk, hogy a `NotImplementedError` egy szabványos kivétel, amely nem tévesztendő össze a `NotImplemented` objektummal.) A `NotImplementedError` kivétel kiváltása helyett – különösen, ha utánozni szeretnénk a Python beépített osztályainak viselkedését – választhatjuk a `TypeError`-t is. Az alábbiakban bemutatjuk, miként érhetjük el, hogy a `FuzzyBool.__add__()` pontosan úgy viselkedjen, mint a beépített osztályok, amikor érvénytelen művelettel találkoznak:

```
def __add__(self, other):
    raise TypeError("unsupported operand type(s) for +: "
                    "'{0}' and '{1}'".format(
                self.__class__.__name__, other.__class__.__name__))
```

Az egytényezős műveletek esetében, amennyiben a kiiktatást a beépített típusok viselkedését utánozva szeretnénk megvalósítani, kicsit egyszerűbb dolgunk van:

```
def __neg__(self):
    raise TypeError("bad operand type for unary -: '{0}'".format(
                    self.__class__.__name__))
```

Az összehasonlító műveletek esetében a helyzet jelentősen egyszerűsödik. Az == kiiktatására például az alábbiakat írhatjuk:

```
def __eq__(self, other):
    return NotImplemented
```

Ha egy tagfüggvény, amely összehasonlító műveletet (<, <=, ==, !=, >=, >) valósít meg, a beépített `NotImplemented` objektumot adja vissza, amikor kísérletet teszünk a használatára, a Python először megfordítja az összehasonlítást, felcserélve a tényezőket (hátha a másik objektum rendelkezik a megfelelő összehasonlító tagfüggvénnyel, ha már a `self` nem), ha pedig ez sem vezet eredményre, a rendszer `TypeError` kivételt vált ki egy üzenettel, amely arról tájékoztat, hogy a műveletet az adott típusú tényezők nem támogatják. Az olyan nem összehasonlító tagfüggvények esetében viszont, amelyeket ki szeretnénk iktatni, `NotImplementedError`, illetve `TypeError` kivételt kell kiváltanunk, ahogy azt korábban az `__add__()` és a `__neg__()` tagfüggvényekkel tettük.

Minden olyan tagfüggvényt egyenként kiiktatni, amelyre nincs szükségünk, igencsak fáradságos munka, noha végül a kívánt eredményre vezet. Az alábbiakban egy ennél hatékonyabb módszert mutatunk be a tagfüggvények kiiktatására (ezt alkalmazzuk a `FuzzyBoolAlt` modulban is), de jobb, ha a következő részre ugrunk, és csak akkor térünk vissza ide, amikor a kiiktatásra a gyakorlatban is szükség lesz.

```
for name, operator in (("__neg__", "-"),
                       ("__index__", "index()")):
    message = ("bad operand type for unary {0}: '{{self}}'"
               .format(operator))
    exec("def {0}(self): raise TypeError(\"{1}\".format("
         "self=self.__class__.__name__))".format(name, message))
```

A beépített `exec()` függvény dinamikusan végrehajtja a paraméterként megadott objektumtól kapott kódot. Esetünkben egy karakterláncot adtunk át a számára, jóllehet más típusú objektumok is szerepelhetnek paraméterként. A rendszer alapértelmezés szerint az utasítást tartalmazó hatókör környezetében futtatja a kódot – esetünkben a `FuzzyBool` osztály meghatározásában –, így a végrehajtott `def` utasítások `FuzzyBool` tagfüggvényeket hoznak létre, ami éppen megfelel az elvárásainknak. A Python csak egyszer hajtja végre a kódot, mégpedig a `FuzzyBoolAlt` modul betöltésekor. Íme az első értékpár, vagyis a `("__neg__", "-")` esetében előállított kód:

```
def __neg__(self):
    raise TypeError("bad operand type for unary -: '{self}'"
                    .format(self=self.__class__.__name__))
```

A kivételt és a hibaüzenetet úgy választottuk meg, hogy megfeleljenek a Python saját típusainál kapottaknak. A kéttényezős tagfüggvények és az n-tényezős függvények (mint a `pow()`) hasonló kódot kapnak, mindössze a hibaüzenetben jelentkezik eltérés. A teljesség kedvéért bemutatjuk az itt alkalmazott kódot:

```
for name, operator in (("__xor__", "^"), ("__ixor__", "^="),
        ("__add__", "+"), ("__iadd__", "+="), ("__radd__", "+"),
        ("__sub__", "-"), ("__isub__", "-="), ("__rsub__", "-"),
        ("__mul__", "*"), ("__imul__", "*="), ("__rmul__", "*"),
        ("__pow__", "**"), ("__ipow__", "**="),
        ("__rpow__", "**"), ("__floordiv__", "//"),
        ("__ifloordiv__", "//="), ("__rfloordiv__", "//"),
        ("__truediv__", "/"), ("__itruediv__", "/="),
        ("__rtruediv__", "/"), ("__divmod__", "divmod()"),
        ("__rdivmod__", "divmod()"), ("__mod__", "%"),
        ("__imod__", "%="), ("__rmod__", "%"),
        ("__lshift__", "<<"), ("__ilshift__", "<<="),
        ("__rlshift__", "<<"), ("__rshift__", ">>"),
        ("__irshift__", ">>="), ("__rrshift__", ">>")):
```

```
    message = ("unsupported operand type(s) for {0}: "
               "'{{self}}'{{join}} {{args}}".format(operator))
    exec("def {0}(self, *args):\n"
         "    types = [\"'\" + arg.__class__.__name__ + \"'\" "
         "for arg in args]\n"
         "    raise TypeError(\"{1}\".format("
         "self=self.__class__.__name__, "
         "join=(\" and\" if len(args) == 1 else \",\"),"
         "args=\", \".join(types)))".format(name, message))
```

A kód némiképp összetettebb az előzőekben látottaknál, ugyanis a kéttényezős műveleteknél jelentkező hibaüzenetekben a két típus felsorolásánál a `type1` and `type2` alakot kell használnunk, míg ha három vagy több típus szerepel, a `type1`, `type2`, `type3` felsorolt alakot kell választanunk, ha utánozni szeretnénk a beépített típusok viselkedését. Íme az első értékpár, a `("__xor__", "^")` esetében előállított kód:

```
def __xor__(self, *args):
    types = ["'" + arg.__class__.__name__ + "'" for arg in args]
    raise TypeError("unsupported operand type(s) for ^: "
                    "'{self}'{join} {args}".format(
                    self=self.__class__.__name__,
                    join=(" and" if len(args) == 1 else ","),
                    args=", ".join(types)))
```

A két `for...in` ciklust egyszerűen kivághatjuk és átmásolhatjuk, majd eldönthetjük, hogy mely egy-, két-, illetve n-tényezős műveleteket szeretnénk meghagyni, illetve kiiktatni – eszerint kell eltávolítanunk vagy hozzáadnunk az elemeket az egyes ciklusokhoz.

Ha ezek a kódrészletek is a helyükre kerültek, az `f` és `g` `FuzzyBool` objektumokkal az `f + g` műveletet végezve egy `TypeError` kivételt kapunk az „unsupported operand type(s) for +: 'FuzzyBool' and 'FuzzyBool'" hibaüzenettel – ez éppen a kívánt viselkedés.

Az osztályok létrehozásánál a leggyakrabban úgy járunk el, mint a `FuzzyBool` első megvalósításánál – ez a módszer az esetek túlnyomó többségében tökéletesen megfelel. Ha azonban állandó értékű osztályra van szükségünk, akkor valamelyik beépített állandó értékű típusból érdemes kiindulnunk (`float`, `int`, `str` vagy `tuple`), és újra meg kell valósítanunk az `object.__new__()` tagfüggvényt, valamint meg kell írnunk a kívánt viselkedéshez szükséges további tagfüggényeket. Ennek a módszernek az a hátránya, hogy egyes tagfüggvényeket ki kell iktatnunk – ez tönkreteszi a többalakúságot, így a legtöbb esetben az összetétel módszere, amelyet a `FuzzyBool` első megvalósításánál használtunk, kedvezőbb megoldást ad.

Egyedi gyűjteményosztályok

A következőkben olyan osztályokkal foglalkozunk, amelyek nagy adatmennyiségek tárolására alkalmasak. Az elsőként bemutatott `Image` nevű osztály képek adatait képes megőrizni, és jól példázza az egyedi adattároló osztályok működését, hiszen amellett, hogy az adatait elérhetővé teszi a memóriában, rendelkezik olyan tagfüggvényekkel is, amelyek biztosítják a lemezre mentés és a betöltés lehetőségét. Ezt követően megismerkedünk a `SortedList` és a `SortedDict` osztályokkal, amelyekkel a Python alapkönyvtárának meglepő hiányosságát pótoljuk – lehetővé tesszük ugyanis beépített rendezéssel bíró gyűjteményes adattípusok használatát.

Gyűjteményeket egyesítő osztályok létrehozása

A síkbeli színes képek megjelenítésére kínálkozik egy egyszerű módszer, ahol egy kétdimenziós tömböt használunk, amelynek minden eleme egy szín. Így ha egy 100x100 képpontos képet szeretnénk ábrázolni, 10 000 szín tárolására van szükség. Az `Image` osztályban (amelyet az `Image.py` fájlban találhatunk meg) ennél hatékonyabb módszert alkalmazunk – az osztály ugyanis tárol egy háttérszínt, valamint azokat a pontokat, amelyeknek a színe eltér ettől a háttérszíntől. Mindehhez egy szótárt használunk, egyfajta ritka tömbként, amelyben a kulcsok az `(x, y)` koordinátapárok, a hozzájuk tartozó értékek pedig a megfelelő színek. Így ha egy 100x100-as képet szeretnénk tárolni, amelyben a pontok felerészben háttérszínűek, mindössze 5000+1 színt kell tárolnunk, ami jelentős memóriamegtakarítást jelent.

Az `Image.py` modul szerkezete ismerős módon épül fel: az elején egy shebang-sor áll, ezt követik a szerzői jogi adatok megjegyzések formájában, majd a modul dokumentációs karakterlánca néhány teszttel, ezután pedig betöltjük az `os` és a `pickle` modulokat. A képek betöltésének és mentésének tárgyalásakor ez utóbbiról ejtünk majd pár szót. A betöltés végeztével létrehozunk néhány egyedi kivételosztályt:

Karakterkódolások
☞ 297, 317

```
class ImageError(Exception): pass
class CoordinateError(ImageError): pass
```

Mindössze az első két kivételosztályt mutattuk be; a többi (`LoadError`, `SaveError`, `ExportError` és `NoFilenameError`) létrehozása hasonló módon történik, és mindegyikük az `ImageError` leszármazottja. Az `Image` osztály felhasználói e kivételek bármelyikét vizsgálhatják, de megelégedhetnek az `ImageError` alaposztály ellenőrzésével is.

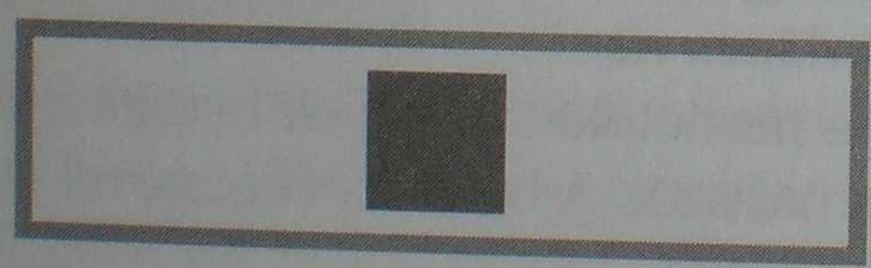
A modul további részében az `Image` osztály kódja található, valamint három szabványos sor a fájl végén a modul tesztjeinek futtatására. Mielőtt az osztály szerkezetének boncolgatásába fognánk, lássuk, miként vehetjük hasznát:

```
border_color = "#FF0000"     # vörös
square_color = "#0000FF"     # kék
width, height = 240, 60
midx, midy = width // 2, height // 2
```

```
image = Image.Image(width, height, "square_eye.img")
for x in range(width):
    for y in range(height):
        if x < 5 or x >= width - 5 or y < 5 or y >= height - 5:
            image[x, y] = border_color
        elif midx - 20 < x < midx + 20 and midy - 20 < y < midy + 20:
            image[x, y] = square_color
image.save()
image.export("square_eye.xpm")
```

Vegyük észre, hogy a kép színeinek beállítására használhatjuk az elemelérési műveletet (`[]`). A szögletes zárójeleket alkalmazhatjuk a pontok kiolvasására és törlésére is (ez utóbbi esetben háttérszínűre változtatjuk a kívánt pontot). A koordinátákat egyetlen rendezett sorozat objektumként adhatjuk át (hála a vesszőnek) – az eredmény ugyanaz, mintha az `image[(x, y)]` alakot használnánk. A használati alakok ilyen egységét a Pythonban könnyen elérhetjük – mindössze a megfelelő különleges tagfüggvényeket kell megvalósítanunk. Az elemelérési művelet esetében itt a `__getitem__()`, a `__setitem__()` és a `__delitem__()` tagfüggvényekről van szó.

Az `Image` osztály HTML-stílusú hexadecimális karakterláncokkal írja le a színeket. A háttérszínt az objektum létrehozásánál kell megadnunk, egyébként az alapértelmezett fehér színt veszi fel. Az osztály saját formátumában menti és tölti be a képeket, de lehetőséget kapunk a képek `.xpm` formátumú kivitelére is, amelyet számos képfeldolgozó alkalmazás ismer. A fenti kódrészletben előállított `.xpm` képet a 6.6. ábrán láthatjuk.



6.6. ábra

A square_eye.xpm kép

A következőkben megismerkedünk az `Image` osztály tagfüggvényeivel – elsőként nézzük a `class` sort és az előkészítést:

```
class Image:
    def __init__(self, width, height, filename="",
                 background="#FFFFFF"):
        self.filename = filename
        self.__background = background
        self.__data = {}
        self.__width = width
        self.__height = height
        self.__colors = {self.__background}
```

Ha létrehozunk egy `Image` objektumot, meg kell határoznunk a szélességét és magasságát, a fájlnevet és a háttérszínt azonban nem feltétlenül, hiszen az osztály itt alapértelmezett értékeket biztosít. A `self.__data` osztály kulcsai `(x, y)` koordináták, értékei pedig színkarakterláncok. A `self.__colors` halmazba kezdetben a háttérszín kerül, a továbbiakban pedig ez tartja nyilván, hogy mely színek szerepelnek a képen.

A fájlnév kivételével az összes adatjellemző privát, így valamilyen formában elérhetővé kell tennünk őket a felhasználók számára. Éppen erre szolgálnak a tulajdonságok.*

```
@property
def background(self):
    return self.__background

@property
def width(self):
    return self.__width

@property
def height(self):
    return self.__height

@property
def colors(self):
    return set(self.__colors)
```

Gyűjtemények másolása 132, 142 ☞

Ha egy adatjellemzőt adunk vissza egy objektumból, fontos tisztában lennünk azzal, hogy változtatható vagy állandó értékű jellemzőről van-e szó. Az utóbbi esetében az érték visszaadása mindig biztonságos, hiszen nem változtatható meg, a változtatható jellemzőknél azonban el kell gondolkodnunk, hogy mit is tegyünk. Ha közvetlenül a változtatható jellemző egy hivatkozását adjuk vissza, igen gyors és hatékony eredményt érhetünk el, hiszen nincs szükség másolásra, de ezzel a hívó hozzáfér az objektum belső állapotához, és ennek módosításával akár érvénytelenné is teheti az objektumot. Érdemesebb a változtatható adattípusok esetében csak egy másolatot visszaadnunk, hacsak a vizsgálatok ki nem mutatják, hogy ez rossz hatással van a program teljesítményére. (Esetünkben az egyedi színek tárolása helyett egy szín kiolvasásakor visszaadhatnánk a `set(self.__data.values()) | {self.__background}` értéket is.)

```
def __getitem__(self, coordinate):
    assert len(coordinate) == 2, "coordinate should be a 2-tuple"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    return self.__data.get(tuple(coordinate), self.__background)
```

* A 8. fejezetben a jellemzők elérésének egy teljesen eltérő módszerét mutatjuk majd be, amelyben a különleges tagfüggvények, mint a `__getattr__()` és a `__setattr__()` kapnak szerepet – bizonyos körülmények között ez a megoldás a célravezető.

Ez a tagfüggvény az elemelérési művelet ([]) segítségével olvassa ki az adott koordinátákkal meghatározott pont színét. Az elemelérési művelet, valamint néhány, a gyűjteményeknél használatos különleges tagfüggvény leírását megtalálhatjuk a 6.4. táblázatban.

Az elemek elérésénél két szabályt alkalmaztunk. Az első egy előfeltétel, amely szerint a koordináták megadásánál kételemű sorozatokat (jobbára rendezett párokat) kell használnunk – ezt egy feltételezéssel biztosítjuk. A második szabály alapján tetszőleges koordinátaértéket elfogadunk, de ha bármelyik koordináta kívül esik a kép tartományán, egyedi kivételt váltunk ki.

Egy adott koordinátával meghatározott képpont színének kiolvasásához a `dict.get()` tagfüggvényt használjuk, amelynek a háttérszínt adjuk meg alapértelmezett értékként. Ezzel biztosítjuk, hogy ha nem állítottuk be a pont színét, akkor visszakapjuk a háttérszínt – az egyébként várható `KeyError` kivétel helyett.

```
def __setitem__(self, coordinate, color):
assert len(coordinate) == 2, "coordinate should be a 2-tuple"
```

6.4. táblázat *A gyűjtemények különleges tagfüggvényei*

Különleges tagfüggvény	Használat	Leírás
`__contains__(self, x)`	`x in y`	Az eredmény `True`, ha az `x` szerepel az `y` sorozatban, vagy kulcs az `y` megfeleltetésben.
`__delitem__(self, k)`	`del y[k]`	Törli az `y` sorozat `k`-adik elemét, illetve az `y` megfeleltetés `k` kulcsához tartozó elemét.
`__getitem__(self, k)`	`y[k]`	Visszaadja az `y` sorozat `k`-adik elemét, illetve az `y` megfeleltetés `k` kulcsához tartozó elemét.
`__iter__(self)`	`for x in y: pass`	Visszaad egy bejárót az `y` sorozat elemeihez, illetve az `y` megfeleltetés kulcsaihoz.
`__len__(self)`	`len(y)`	Visszaadja az `y` elemeinek számát.
`__reversed__(self)`	`reversed(y)`	Visszaad egy hátrafelé haladó bejárót az `y` sorozat elemeihez, illetve az `y` megfeleltetés kulcsaihoz.
`__setitem__(self, k, v)`	`y[k] = v`	Az `y` sorozat `k`-adik elemét, illetve az `y` megfeleltetés `k` kulcsához tartozó elemét `v` értékre állítja.

```
if (not (0 <= coordinate[0] < self.width) or
    not (0 <= coordinate[1] < self.height)):
    raise CoordinateError(str(coordinate))
```

```
if color == self.__background:
    self.__data.pop(tuple(coordinate), None)
else:
    self.__data[tuple(coordinate)] = color
    self.__colors.add(color)
```

Ha a felhasználó egy koordináta értékét a háttérszínre állítja, egyszerűen törölhetjük a szótár megfelelő elemét, hiszen ha egy koordináta nincs a szótárban, az osztály a megfelelő pontot automatikusan háttérszínűnek tekinti. A `dict.pop()` tagfüggvényt alkalmaztuk egy szabadon választott második paraméterrel, ha ugyanis a `del`-t használnánk, a rendszer `KeyError` kivételt váltana ki, ha a kulcs (vagyis a koordináta) nem szerepel a szótárban.

Ha a szín eltér a háttérszíntől, tároljuk a megfelelő koordinátán, továbbá elhelyezzük a felhasznált színek halmazában.

```
def __delitem__(self, coordinate):
    assert len(coordinate) == 2, "coordinate should be a 2-tuple"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    self.__data.pop(tuple(coordinate), None)
```

Amennyiben törölni szeretnénk egy koordináta színét, valójában háttérszínűvé kell alakítanunk. Ezúttal is a `dict.pop()` tagfüggvényt használjuk, hiszen ez mindig a megfelelő eredményt adja, akár a szótárban van a megadott koordináta, akár nem.

A `__len__()` számára nem készítettünk megvalósítást, hiszen e tagfüggvény használatának nem lenne sok értelme egy kétdimenziós objektumnál. Emellett ábrázolási alakot sem adhatunk meg, hiszen egy teljes `Image` objektumot nem lehet létrehozni az `Image()` hívással, így hát nincs szükség a `__repr__()` (vagy az `__str__()`) tagfüggvényre. Ha egy felhasználó mégiscsak meghívja a `repr()` vagy az `str()` függvényt egy `Image` objektumra, az alaposztály `object.__repr__()` tagfüggvénye visszaadja a megfelelő karakterláncot, a következő alakban: `'<Image.Image object at 0x9c794ac>'`. Ez a formátum szabványos az `eval()` függvénnyel nem kiértékelhető objektumok körében. A megadott hexadecimális szám az objektum azonosítója – ez egyedi (többnyire az objektum címe a memóriában), de változhat.

Szeretnénk az `Image` osztály felhasználóinak lehetővé tenni, hogy lemezre menthessék, valamint betölthessék az adataikat – ennek érdekében készítettük el a `save()` és a `load()` tagfüggvényeket.

A tárolás megvalósítására az adatok *befőzése* (pickling) mellett döntöttünk. A Python nyelvezetében a befőzés a sorosítás (serialization) bizonyos fajtájára utal (amelynek során az adatokat egyszerű bájtsorozattá vagy karakterlánccá alakítjuk). A módszer jelentősége

abban rejlik, hogy a befőzött objektum származhat gyűjteményes adattípusból, így lehet lista vagy szótár, és tartalmazhat más objektumokat is (más gyűjteményeket, gyűjtemények gyűjteményeit, és így tovább) – ráadásul a rendszer arra is ügyel, hogy a többször előforduló objektumokat csak egyszer tárolja.

A befőttet (pickle) egyszerűen beolvashatjuk egy Python-változóba – nem kell értelmeznünk vagy átalakítanunk a kapott adatokat. Így a módszer különösen jó szolgálatot tesz ötletszerűen létrehozott adatgyűjtemények tárolására, különösen kisebb, illetve személyes használatra szánt programokban.

Figyelem! *Mindazonáltal a befőttek nem rendelkeznek semmiféle biztonsági eljárással (titkosítással vagy digitális aláírással), így a megbízhatatlan forrásból származó befőttek felbontása bizony veszélyekkel járhat. Ennek ismeretében a nem kifejezetten személyes használatra szánt programok esetében jobb, ha kifejezetten a programhoz készített egyedi fájlformátumot alkalmazunk. A 7. fejezetben megtanuljuk, miként hozhatunk létre saját bináris, szöveges és XML-fájlformátumokat.*

```
def save(self, filename=None):
    if filename is not None:
        self.filename = filename
    if not self.filename:
        raise NoFilenameError()
    fh = None
    try:
        data = [self.width, self.height, self.__background,
                self.__data]
        fh = open(self.filename, "wb")
        pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)
    except (EnvironmentError, pickle.PicklingError) as err:
        raise SaveError(str(err))
    finally:
        if fh is not None:
            fh.close()
```

A függvény első része teljes egészében a fájlnévvel foglalkozik. Amennyiben az `Image` objektumot fájlnév megadása nélkül hoztuk létre, és azóta sem határoztunk meg ilyet, a `save()` tagfüggvénynek mindenképpen át kell adnunk egy fájlnevet (ez esetben a tagfüggvény a „mentés másként" lehetőséget adja, ahol meg kell adnunk a programban a továbbiakban használt fájlnevet). Ha nem adunk át fájlnevet, a program az aktuális fájlnevet használja – ha nincs ilyen, és nem is adjuk meg, kivételt kapunk.

Létrehozunk egy `data` nevű listát – ez tárolja majd azokat az objektumokat, amelyeket menteni szeretnénk. Ide tartozik a koordinátákhoz tartozó színeket tartalmazó `self.__data` szótár, a felhasznált színek halmaza azonban kimarad, hiszen ez előállítható a szótár alapján. Ezek után már csak meg kell nyitnunk a fájlt a bináris módú íráshoz, és a `pickle.dump()` függvénnyel tárolnunk az adatobjektumot. Ennyi az egész!

A `pickle` modul különböző formátumokban képes sorosítani az adatainkat (ezeket a formátumokat a Python leírása *protokolloknak* nevezi) – közülük a `pickle.dump()` harmadik paraméterének megadásával választhatunk. A 0-s protokoll ASCII-adatokat tárol, és a hibakeresésben vehetjük nagy hasznát. Esetünkben a 3-as protokoll (`pickle.HIGHEST_PROTOCOL`) mellett döntöttünk, amely tömör, bináris formátumot biztosít – ez indokolja, hogy miért kellett bináris módban megnyitnunk a fájlt. A befőttek olvasásánál nem kell nyilatkoznunk a protokollról, ugyanis a `pickle.load()` függvény képes kikövetkeztetni, hogy melyiket használtuk.

```
def load(self, filename=None):
    if filename is not None:
        self.filename = filename
    if not self.filename:
        raise NoFilenameError()
    fh = None
    try:
        fh = open(self.filename, "rb")
        data = pickle.load(fh)
        (self.__width, self.__height, self.__background,
         self.__data) = data
        self.__colors = (set(self.__data.values()) |
                         {self.__background})
    except (EnvironmentError, pickle.UnpicklingError) as err:
        raise LoadError(str(err))
    finally:
        if fh is not None:
            fh.close()
```

A fenti függvény ugyanúgy kezdődik, mint a `save()`, így hozzájut a betöltendő fájl nevéhez. A fájlt bináris olvasási módban kell megnyitnunk, az adatokat pedig egyetlen utasítással olvassuk be, méghozzá ilyen egyszerűen: `data = pickle.load(fh)`. Az adatobjektum a mentett adatok pontos mását adja vissza, így egy listát kapunk, amelyben szerepel a szélesség és a magasság egész értékek formájában, a háttérszín karakterlánca, valamint a szótár a koordináta–szín elemekkel. A rendezett sorozatok kicsomagoló műveletével a lista elemeit hozzárendeljük a megfelelő változókhoz, így a bennük tárolt korábbi képadatok (érthető módon) elvesznek.

A felhasznált színek halmazát visszaállítjuk a szótár tartalma alapján, kiegészítve a háttérszínnel.

```
def export(self, filename):
    if filename.lower().endswith(".xpm"):
        self.__export_xpm(filename)
    else:
        raise ExportError("unsupported export format: " +
                          os.path.splitext(filename)[1])
```

Egyetlen általános kiviteli tagfüggvényt készítettünk, amely a fájl kiterjesztése alapján választja ki, hogy melyik privát tagfüggvény lehetőségeit vegye igénybe – ha pedig az adott fájlformátum esetében nem képes a kivitelre, kivételt vált ki. Esetünkben csak az `.xpm` fájlok kivitelét támogatjuk (azt is csak akkor, ha a kép 8390-nél kevesebb színt használ). Az `__export_xpm()` függvény kódját nem közöltük, mivel nem kapcsolódik szorosan fejezetünk anyagához, de a mellékelt forráskódban természetesen megtalálhatjuk.

Ezzel végére értünk az `Image` osztály megvalósításának, amellyel számos jellemzően előforduló feladatot oldottunk meg – így tároltunk a programra jellemző adatokat, elérhetővé tettük annak egyes elemeit, gondoskodtunk a lemezre mentés és a betöltés lehetőségéről, mindemellett a kódban kizárólag a feltétlenül szükséges tagfüggvények megvalósítására szorítkoztunk. A következőkben két általános egyedi gyűjteményes adattípust mutatunk be, amelyek teljes API-t tesznek elérhetővé a felhasználóik számára.

Gyűjteményosztályok létrehozása összetétellel

A következőkben összeállítunk egy `SortedList` nevezetű teljes egyedi gyűjteményosztályt, amely elemek rendezett listájának tárolására képes. A rendezéshez a < műveletet használjuk, amelyet az `__lt__()` különleges tagfüggvény valósít meg, illetve egy kulcsfüggvény, amennyiben megadunk ilyet. Az osztállyal kísérletet teszünk arra, hogy megfeleljünk a `list` osztály API-jának a könnyebb megértés és használat érdekében, de egyes tagfüggvényeket kénytelenek leszünk kihagyni – így például az összefűző műveletet (+), amelynek a használata hibás sorrendet eredményezhetne.

Az egyedi osztályok létrehozásánál megszokott módon választhatunk, hogy az osztályt egy másik hasonlóból származtatjuk, vagy teljesen új osztályt hozunk létre, és a működéséhez szükséges további osztályok példányait a belsejében helyezzük el. A most bemutatott `SortedList` osztály esetében tisztán ez utóbbi módszert alkalmazzuk, míg a `SortedDict` felépítésében egyszerre vesz részt az öröklés és az összetétel.

A 8. fejezetben láthatjuk majd, hogy az osztályok ígéreteket tesznek az általuk biztosított API-kkal kapcsolatban. Így például a `list` osztály a `MutableSequence` API-t biztosítja, így tudhatjuk, hogy támogatja az `in` műveletet, az `iter()` és a `len()` beépített függvényeket, az elemelérési műveletet (`[]`) az elemek kiolvasásához, beállításához és törléséhez, valamint az `insert()` tagfüggvényt. `SortedList` osztályunk nem teszi lehetővé az elemek beállítását, és nem rendelkezik `insert()` tagfüggvénnyel, így nem támogatja a `MutableSequence` API-t. Ha a `SortedList` osztályt a `list` leszármazottjaként hoznánk létre, az eredményként kapott osztály változtatható sorozatként mutatná magát, mindemellett azonban nem valósítaná meg a teljes API-t. Esetünkben a `SortedList` nem a `list` leszármazottja, és így semmiféle ígéretet nem tesz az API-jával kapcsolatban. Másrészről azonban, a későbbiekben bemutatott `SortedDict` osztály a `dict` osztályban elérhető teljes `MutableMapping` API-t megvalósítja, így létrehozhatjuk a `dict` alosztályaként.

Lássunk most néhány példát a `SortedList` használatára:

```
letters = SortedList.SortedList(("H", "c", "B", "G", "e"), str.lower)
# str(letters) == "['B', 'c', 'e', 'G', 'H']"
letters.add("G")
letters.add("f")
letters.add("A")
# str(letters) == "['A', 'B', 'c', 'e', 'f', 'G', 'G', 'H']"
letters[2] # returns: 'c'
```

A `SortedList` *összetétel* (aggregate) két privát jellemzőt tartalmaz (vagyis ezekből épül fel) – egy `self.__key()` nevezetű függvényt (amelyet a `self.__key` objektumhivatkozás alakjában tárol), valamint egy `self.__list` nevű listát.

Lambdafüggvények 167, 178 ☜

A kulcsfüggvényt a második paraméterben adjuk át (illetve a `key` kulcsszó-paraméterrel, ha nem adtunk meg kezdeti sorozatot). Amennyiben nem határozunk meg kulcsfüggvényt, az alábbi privát modulfüggvényt használjuk:

```
_identity = lambda x: x
```

Ez maga az identitásfüggvény: Egyszerűen változatlanul visszaadja a paraméterét, így ha a `SortedList` kulcsfüggvényeként alkalmazzuk, a lista objektumainak rendezési kulcsai önmaguk lesznek.

A `SortedList` típus nem teszi lehetővé az elemelérési műveletnek (`[]`), hogy megváltoztassa az elemeket (tehát nem valósítja meg a `__setitem__()` különleges tagfüggvényt), továbbá nem támogatja az `append()` és az `extend()` tagfüggvényeket, hiszen ezek érvényteleníthetnék a sorrendet. Elemek elhelyezésére kizárólag a `SortedList` objektum létrehozásakor átadott sorozattal, valamint a `SortedList.add()` tagfüggvénnyel van lehetőségünk. Másrészről az elemelérési műveletet nyugodtan használhatjuk elemek kiollvasására és törlésére, hiszen ez nem befolyásolja a sorrendet – megvalósítjuk tehát a `__getitem__()` és a `__delitem__()` különleges tagfüggvényeket.

A következőkben tagfüggvényenként elemezzük az osztály szerkezetét; elsőként szokás szerint a `class` sort és a kezdeti értékadást mutatjuk be:

```
class SortedList:
    def __init__(self, sequence=None, key=None):
        self.__key = key or _identity
        assert hasattr(self.__key, "__call__")
        if sequence is None:
            self.__list = []
        elif (isinstance(sequence, SortedList) and
              sequence.key == self.__key):
            self.__list = sequence.__list[:]
        else:
            self.__list = sorted(list(sequence), key=self.__key)
```

Mivel a függvény neve valójában egy objektumhivatkozás (amely magára a függvényre mutat), így a függvényeket ugyanúgy tárolhatjuk változókban, mint bármely más objektumhivatkozást. Esetünkben a `self.__key` változó egy hivatkozást tárol az átadott kulcsfüggvényre, illetve az identitásfüggvényre. A tagfüggvény első utasítása arra épít, hogy az `or` művelet első tényezőjét adja vissza, amennyiben annak értéke logikai környezetben `True` (márpedig, ha a kulcsfüggvény nem `None`, ez lesz az eredmény), egyébként pedig a második tényezővel tér vissza. Ugyanezt az eredményt kapjuk a kissé hosszabb, ám könnyebben érthető `self.__key = key if key is not None else _identity` utasítással is.

Ha rendelkezünk kulcsfüggvénnyel, egy feltételezéssel biztosítjuk, hogy hívható legyen. A beépített `hasattr()` függvény a `True` értéket adja vissza, ha az első paraméterében átadott objektum rendelkezik a második paraméterben meghatározott nevű jellemzővel. Érdemes tudnunk az ide tartozó `setattr()` és `delattr()` függvényekről is, amelyek tárgyalására a 8. fejezetben kerül sor. Minden hívható objektum – így a függvények és a tagfüggvények is – rendelkezik a `__call__` jellemzővel.

Ahhoz, hogy a `SortedList` objektumok létrehozását a `list` objektumokéhoz hasonlóvá tegyük, bevezetünk egy nem kötelezően megadandó `sequence` nevű paramétert, amely megfelel a `list()` egyetlen, nem kötelező paraméterének. A `SortedList` egy `list` gyűjteményt tárol a `self.__list` privát változóban, és ennek rendezését a megadott kulcsfüggvénnyel tartja fenn.

Az `elif` utasításban típusvizsgálattal állapítjuk meg, hogy az adott sorozat a `SortedList` egy példánya-e, és ha igen, vajon ugyanazzal a kulcsfüggvénnyel rendelkezik-e, mint az aktuális rendezett lista. Ha mindkét feltétel teljesül, egyszerűen sekély másolatot készítünk a sorozat listájából – rendezésre nincs szükség. Amennyiben a legtöbb kulcsfüggvényt helyben, a `lambda` utasítással hozzuk létre, még akkor is eltérőnek mutatkoznak, ha a kódjuk megegyezik, így a várt hatékonyságnövekedés elmaradhat.

```
@property
def key(self):
    return self.__key
```

Ha elkészült a rendezett lista, a kulcsfüggvényét rögzítjük, így privát változóként tároljuk, hogy megakadályozzuk a módosítását a felhasználók számára. Egyeseknek azonban szükségük lehet a kulcsfüggvény hivatkozására (amint azt hamarosan láthatjuk), így elérhetővé tettük a csak olvasható `key` tulajdonság révén.

```
def add(self, value):
    index = self.__bisect_left(value)
    if index == len(self.__list):
        self.__list.append(value)
    else:
        self.__list.insert(index, value)
```

Amennyiben a felhasználó ezt a tagfüggvényt hívja meg, a megadott értéket a `self.__list` privát lista megfelelő helyére kell beillesztenünk, hogy megtartsuk a sorrendet. A kívánt sorszámot a `SortedList.__bisect_left()` tagfüggvény adja meg, amint azt hamarosan láthatjuk. Ha az új érték nagyobb, mint a lista összes eleme, a lista végére kerül, így a sorszám megegyezik a lista hosszával (a sorszámok `0`-tól `len(L) - 1`-ig terjednek) – ha így áll a helyzet, egyszerűen hozzáfűzzük a listához az új értéket. Egyébként pedig a megadott sorszámon illesztjük be az értéket – ami 0 lesz, ha az új érték kisebb a lista összes eleménél.

```
def __bisect_left(self, value):
    key = self.__key(value)
    left, right = 0, len(self.__list)
    while left < right:
        middle = (left + right) // 2
        if self.__key(self.__list[middle]) < key:
            left = middle + 1
        else:
            right = middle
    return left
```

A fenti privát tagfüggvény megadja azt a sorszámot, amely a listában az adott értékhez tartozik, kiszámít egy összehasonlító kulcsot az adott érték számára a rendezett lista kulcsfüggvényének használatával, és összehasonlítja ezt a tagfüggvény által vizsgált elemek számított összehasonlítási kulcsaival. Az algoritmus *bináris keresést* alkalmaz, amely igen nagy listák esetében is hatékonyan működik – olyannyira, hogy ha például egy 1 000 000 elemből álló listában keressük egy érték helyét, mindössze 21 összehasonlítást kell végeznünk.* Érdemes mindezt összevetnünk az egyszerű, rendezetlen `list` objektum esetével, amely lineáris keresést alkalmaz, és ugyanehhez a feladathoz átlagosan 500 000, legrosszabb esetben pedig 1 000 000 összehasonlításra van szüksége.

```
def remove(self, value):
    index = self.__bisect_left(value)
    if index < len(self.__list) and self.__list[index] == value:
        del self.__list[index]
    else:
        raise ValueError("{0}.remove(x): x not in list".format(
                          self.__class__.__name__))
```

Ezzel a tagfüggvénnyel az adott érték első előfordulását távolítjuk el.
A `SortedList.__bisect_left()` tagfüggvénnyel megkeressük azt a sorszámot, ahová az érték tartozik, majd megvizsgáljuk, hogy ez a sorszám létezik-e a listában, és az e sorszámon álló érték megegyezik-e a megadottal. Ha az elem megfelel a feltételeknek, eltávolítjuk, egyébként pedig `ValueError` kivételt váltunk ki (a `list.remove()` viselkedése ilyen körülmények között pontosan így alakul).

* A Python `bisect` moduljában elérhetjük a `bisect.bisect_left()` függvényt és egyéb társait, de könyvünk írásának idején a modul egyetlen függvénye sem volt képes kulcsfüggvényeket kezelni.

```
    def remove_every(self, value):
        count = 0
        index = self.__bisect_left(value)
        while (index < len(self.__list) and
                self.__list[index] == value):
            del self.__list[index]
            count += 1
        return count
```

A fenti tagfüggvény hasonlít a `SortedList.remove()`-hoz – valójában a listák API-jának bővítése. Először is megkeresi azt a sorszámot, ahol az érték először megjelenik a listában, majd addig halad a lista elemein, amíg azok értéke megegyezik a megadottal. A kódban van egy apró csavar, ugyanis a ciklus minden végrehajtásakor törli a megfelelő elemet, így az aktuális sorszám mindig a következő elemre utal.

```
    def count(self, value):
        count = 0
        index = self.__bisect_left(value)
        while (index < len(self.__list) and
                self.__list[index] == value):
            index += 1
            count += 1
        return count
```

A tagfüggvény visszatérési értéke az átadott érték listabeli előfordulásainak száma (ez lehet 0 is). Működése a `SortedList.remove_every()` tagfüggvényben látott cikluson alapul, de ezúttal nem végzünk törlést, és minden ciklusban növeljük az aktuális sorszámot.

```
    def index(self, value):
        index = self.__bisect_left(value)
        if index < len(self.__list) and self.__list[index] == value:
            return index
        raise ValueError("{0}.index(x): x not in list".format(
                        self.__class__.__name__))
```

Mivel a `SortedList` rendezett listát tárol, az érték után gyors bináris kereséssel kutathatunk.

```
    def __delitem__(self, index):
        del self.__list[index]
```

A `__delitem__()` különleges tagfüggvény lehetővé teszi a `del L[n]` alak használatát, ahol *L* egy rendezett lista, *n* pedig egy egész értékű sorszám. A tartományon kívül eső sorszámok esetét nem vizsgáljuk, hiszen ilyen esetben a `self.__list[index]` hívás `IndexError` kivételt vált ki, és éppen ez az, amit vártunk.

```
    def __getitem__(self, index):
        return self.__list[index]
```

Ez a tagfüggvény támogatja az `x = L[n]` alak használatát, ahol *L* egy rendezett lista, *n* pedig egy egész értékű sorszám.

```
def __setitem__(self, index, value):
    raise TypeError("use add() to insert a value and rely on "
                    "the list to put it in the right place")
```

Nem szeretnénk, ha a felhasználó megváltoztathatná egy adott sorszámú elem értékét (vagyis kizárjuk az `L[n] = x` alakot), hiszen így a lista rendezettsége megszűnhetne. A nem támogatott műveleteket a `TypeError` kivétellel jelezhetjük.

```
def __iter__(self):
    return iter(self.__list)
```

A tagfüggvény megvalósítása egyszerű, hiszen a privát lista számára könnyedén visszaadhatunk egy bejárót a beépített `iter()` függvénnyel. Ez a tagfüggvény támogatja az *érték* `in` *bejárható_objektum* alak használatát.

Fontos tudnunk, hogy ha egy sorozatra van szükség, az osztály ezt a tagfüggvényt alkalmazza. Ha tehát egy `L` nevű `SortedList` objektumot listává szeretnénk alakítani, a `list(L)` hívást használjuk, amelynek során a Python a háttérben a `SortedList.__iter__(L)` hívással állítja elő azt a sorozatot, amelyre a `list()` függvénynek szüksége van.

```
def __reversed__(self):
    return reversed(self.__list)
```

Mindezzel a beépített `reversed()` függvényt támogatjuk, így alkalmazhatjuk az *érték* `in` `reversed(`*bejárható_objektum*`)` alakot.

```
def __contains__(self, value):
    index = self.__bisect_left(value)
    return (index < len(self.__list) and
            self.__list[index] == value)
```

A `__contains__()` tagfüggvény az `in` művelet működését teszi lehetővé. Itt ismét alkalmazhatjuk a gyors bináris keresést, szemben az egyszerű `list` típusnál használt, jóval lassabb lineáris változattal.

```
def clear(self):
    self.__list = []
def pop(self, index=-1):
    return self.__list.pop(index)
def __len__(self):
    return len(self.__list)
def __str__(self):
    return str(self.__list)
```

A `SortedList.clear()` tagfüggvény elveti a már meglevő listát, és egy új, üres példánnyal helyettesíti. A `SortedList.pop()` tagfüggvény eltávolítja és visszaadja az adott sorszámon található elemet, illetve, ha a sorszám a tartományon kívül esik, kiváltja az `IndexError` kivételt. A `pop()`, a `__len__()`, valamint az `__str__()` tagfüggvények esetében a munkát egyszerűen áthárítjuk az összetétellel létrehozott `self.__list` objektumra.

A `__repr__()` különleges tagfüggvényt nem valósítjuk meg újra, így az alaposztály `object.__repr__()` tagfüggvénye lép működésbe, ha a felhasználó a `repr(L)` utasítást alkalmazza egy `L` nevű `SortedList` objektumra. Az eredmény így egy `'<SortedList.SortedList object at 0x97e7cec>'` alakú karakterlánc lesz, amelyben a hexadecimális azonosító természetesen változhat. A `__repr__()` számára képtelenek vagyunk értelmes megvalósítást készíteni, hiszen ehhez szükség lenne a kulcsfüggvényre, és egy függvényobjektum hivatkozását nem tudjuk az `eval()`-lal kiértékelhető karakterlánc alakjában ábrázolni.

Az `insert()`, a `reverse()` és a `sort()` tagfüggvényeket nem valósítottuk meg, mivel a használatuknak nem lenne értelme e típus esetén. Bármelyiküket is hívjuk meg, `AttributeError` kivételhez jutunk.

Ha egy rendezett listát az `L[:]` alakkal másolunk, egyszerű `list` objektumhoz jutunk a `SortedList` helyett. Ha valóban másolni szeretnénk, a legegyszerűbb, ha betöltjük a `copy` modult, és a `copy.copy()` függvényt alkalmazzuk – ez képes a rendezett listánkat (és általában a legtöbb egyedi osztályt) bármiféle külső segítség nélkül lemásolni. Mindennek ellenére úgy döntöttünk, hogy saját `copy()` tagfüggvényt írunk:

```
def copy(self):
    return SortedList(self, self.__key)
```

Az első paraméterként átadott `self` objektummal biztosítjuk, hogy a rendszer sekély másolatot készítsen a `self.__list` objektumból, és ne rendezze a másolás után. (Mindez az `__init__()` tagfüggvény `elif` utasításánál található típusvizsgálatnak köszönhető.) Ez a módszer elméletben hatékonyságnövekedést ígér, de ez jelen pillanatban nem érhető el a `copy.copy()` függvény számára – szerencsére a helyzeten egyetlen sorral változtathatunk:

```
__copy__ = copy
```

A `copy.copy()` hívásakor a függvény megpróbálja használatba venni az objektum `__copy__()` különleges tagfüggvényét, ha pedig nem talál ilyet, a saját kódját alkalmazza. A fenti sor használata esetén a függvény a `SortedList.copy()` tagfüggvényt veszi igénybe a rendezett listák másolásánál. (Készíthetünk egy `__deepcopy__()` nevű különleges tagfüggvényt is, de ez már több erőfeszítést igényel – a részletekről a `copy` modul leírásából tájékozódhatunk.)

Ezzel a `SortedList` osztály megvalósításának végére értünk. A következőkben az itt elért eket arra használjuk, hogy a `SortedDict` osztály kulcsait egy rendezett listában tároljuk.

Gyűjteményosztályok létrehozása örökléssel

A következőkben bemutatott `SortedDict` osztály a `dict` viselkedését kísérli meg minél élethűbben utánozni. A lényegi különbség a két típus között abban áll, hogy a `SortedDict` objektumok kulcsai mindig rendezettek egy megadott kulcsfüggvény vagy az identitásfüggvény szerint. A `SortedDict` ugyanazt az API-t teszi elérhetővé, mint a `dict` (eltekintve a `repr()` függvénytől, amely az `eval()`-lal kiértékelve nem adja vissza az objektumot), két tagfüggvénnyel kiegészítve, amelyek használatának csak rendezett gyűjtemények esetében van értelme.*

Lássunk most néhány példát, hogy megismerkedjünk a `SortedDict` felhasználási lehetőségeivel:

```
d = SortedDict.SortedDict(dict(s=1, A=2, y=6), str.lower)
d["z"] = 4
d["T"] = 5
del d["y"]
d["n"] = 3
d["A"] = 17
str(d) # returns: "{'A': 17, 'n': 3, 's': 1, 'T': 5, 'z': 4}"
```

`SortedDict` osztályunk megvalósításánál egyaránt felhasználjuk az összetétel és az öröklés lehetőségeit. A kulcsok rendezett listáját egy példányváltozó formájában ágyazzuk be, ugyanakkor a `SortedDict` osztály a `dict` leszármazottja. A kód áttekintését a `class` sorral és a kezdeti értékadással nyitjuk, majd sorra vesszük az osztály tagfüggvényeit.

```
class SortedDict(dict):
    def __init__(self, dictionary=None, key=None, **kwargs):
        dictionary = dictionary or {}
        super().__init__(dictionary)
        if kwargs:
        super().update(kwargs)
    self.__keys = SortedList.SortedList(super().keys(), key)
```

A `class` sorban megadjuk a `dict` alaposztályt, majd a kezdeti értékadásnál a `dict()` függvényhez hasonlóan járunk el, de a kezdeti értékadó tagfüggvény egy további paramétert is vár a kulcsfüggvény beolvasására. `SortedDict` objektumunk kezdeti beállításait az alaposztály megfelelő függvényével végezzük el a `super().__init__()` hívással. Hasonlóan, ha kulcsszó-paramétereket alkalmazunk, az alaposztály `dict.update()` tagfüggvényével helyezzük el őket a szótárban. (Fontos megjegyeznünk, hogy a kulcsszó-paramétereknek csak egy előfordulása lehetséges, így a `kwargs` kulcsszó-paramétereiben nem fordulhat elő a `"dictionary"` vagy a `"key"` kulcsszó.)

* Az itt bemutatott `SortedDict` osztály eltér a szerző *Rapid GUI Programming with Python and Qt* című könyvében (ISBN 0132354187) találhatótól, valamint a Python Package Index hasonló nevű osztályától.

A szótár kulcsaiból másolatokat tárolunk egy rendezett listában, amely a `self.__keys` változóban kap helyet. A rendezett lista kezdeti beállításához átadjuk a szótár kulcsait az alaposztály `dict.keys()` tagfüggvényének – a `SortedList.keys()` tagfüggvényt nem használhatjuk, hiszen ez a `self.__keys` változóra épít, ami csak az után jön létre, hogy létrehoztuk ezt a rendezett listát.

```
def update(self, dictionary=None, **kwargs):
    if dictionary is None:
        pass
    elif isinstance(dictionary, dict):
        super().update(dictionary)
    else:
        for key, value in dictionary.items():
            super().__setitem__(key, value)
    if kwargs:
        super().update(kwargs)
    self.__keys = SortedList.SortedList(super().keys(),
                                        self.__keys.key)
```

Ez a tagfüggvény teszi lehetővé, hogy egy szótár elemeit egy másikéival, kulcsszó-paraméterekkel, illetve mindkét módon frissítsük. Azok az elemek, amelyek csak a másik szótárban léteznek, bekerülnek ebbe, míg ott, ahol a kulcs megegyezik, az új érték felülírja a régit. Az eredeti viselkedést némiképp bővítenünk kellett, így megtartjuk az eredeti szótár kulcsfüggvényét, még akkor is, ha a másik szótár a `SortedDict` egy példánya.

A frissítést két lépésben végezzük el. Először a szótár elemeit frissítjük. Amennyiben az adott szótár a `dict` alosztálya (ide tartozik a `SortedDict` is), az alaposztály `dict.update()` tagfüggvényét alkalmazzuk – így elkerüljük, hogy a `SortedDict.update()` tagfüggvény hívásával végtelen önhivatkozáshoz jussunk. Ha a szótár nem `dict` objektum, sorra vesszük az elemeit, és egyenként állítjuk be a kulcs-érték párokat. (Amennyiben a szótár nem `dict` objektum, ráadásul nem rendelkezik az `items()` tagfüggvénnyel, `AttributeError` kivételt váltunk ki.) Ha kulcsszó-paramétereket alkalmaztunk, a beágyazásukhoz ismét az alaposztály `update()` tagfüggvényét hívjuk meg.

A frissítés következtében a `self.__keys` lista elavulttá válik, így helyettesítjük egy új `SortedList` objektummal, amely tartalmazza a szótár kulcsait (ezeket az alaposztálytól kell beszereznünk, ugyanis a `SortedDict.keys()` tagfüggvény éppen azt a `self.__keys` listát használja, amelyet frissítünk), valamint az eredeti rendezett lista kulcsfüggvényét.

```
@classmethod
def fromkeys(cls, iterable, value=None, key=None):
    return cls({k: value for k in iterable}, key)
```

A `dict` osztály API-ja tartalmazza a `dict.fromkeys()` tagfüggvényt, amellyel új szótárt hozhatunk létre egy bejárható objektumból kiindulva. A bejárható objektum elemeiből lesznek a kulcsok, amelyek a `None`, illetve a megadott értéket kapják.

Osztálytagfüggvényről lévén szó, az első paramétert a Python szolgáltatja, értéke pedig maga az osztály. A `dict` osztály esetében ez a paraméter a `dict`, a `SortedDict` esetében pedig a `SortedDict`. Visszatérési értékként megkapjuk az osztályhoz tartozó szótárt, valahogy így:

```
class MyDict(SortedDict.SortedDict): pass
d = MyDict.fromkeys("VEINS", 3)
str(d)          # az eredmény: "{'E': 3, 'I': 3, 'N': 3, 'S': 3, 'V': 3}"
d.__class__.__name__          # az eredmény: 'MyDict'
```

Ha tehát az örökölt osztálytagfüggvényeket hívjuk meg, a `cls` változójukba a megfelelő osztály kerül – éppúgy, mint az egyszerű tagfüggvényeknél, ahol a `self` változó mindig az aktuális objektumot tartalmazza. Ezért ez jobb megoldás, mint a statikus tagfüggvények használata, hiszen azok adott osztályhoz kötődnek, függetlenül attól, hogy az eredeti osztály vagy egy alosztályának környezetében futtatjuk.

```
def __setitem__(self, key, value):
    if key not in self:
        self.__keys.add(key)
    return super().__setitem__(key, value)
```

Ezzel a tagfüggvénnyel valósítjuk meg a `d[key] = value` alakot. Ha a `key` nem szerepel a szótárban, hozzáadjuk a kulcsok listájához, a `SortedList` segítségével megtalálva a helyét a sorban. Ez után meghívjuk az alaposztály megfelelő tagfüggvényét, az eredményt pedig visszaadjuk a hívónak, hogy lehetővé tegyük a műveletek láncba rendezését, valahogy így: `x = d[key] = value`.

Vegyük észre, hogy az `if` utasításban a `not in self` kóddal vizsgáljuk meg, hogy a kulcs szerepel-e már a `SortedDict` objektumban. Mivel a `SortedDict` a `dict` leszármazottja, a példányai bárhol szerepelhetnek, ahol a `dict` objektumok, már pedig ez esetben a `self` `SortedDict` típusú. Amikor a `dict` tagfüggvényeit újra megvalósítjuk a `SortedDict` osztályban, ügyeljünk arra, hogy a szülőosztálybeli megvalósításokat a `super()` segítségével hívjuk meg – így tettünk a fenti tagfüggvény utolsó sorában is. Ezzel meggátoljuk, hogy az újra megvalósított tagfüggvény saját magát hívja meg, és így végtelen önhivatkozásba kerüljön.

Generátorfüggvények

A *generátorfüggvények*, illetve *generátor-tagfüggvények* jellemzője, hogy tartalmazzák a `yield` kifejezést, eredményként pedig bejárókat adnak vissza, amelyeknek az értékeit egyenként olvashatjuk ki a `__next__()` tagfüggvényükkel. A `__next__()` tagfüggvény minden hívásakor hozzájutunk a generátorfüggvény `yield` kifejezésének értékéhez (ha nincs ilyen, akkor a kapott érték a `None`). Ha a generátorfüggvény futása befejeződik, illetve végrehajt egy `return` utasítást, `StopIteration` kivétel lép fel.

A gyakorlatban ritkán hívjuk meg a `__next__()` tagfüggvényt, illetve fogjuk el a `StopIteration` kivételt – ehelyett úgy használjuk a generátorfüggvényeket, mint bármely más bejárható objektumot. Az alábbiakban két, majdhogynem egyenértékű függvényt mutatunk be – a bal oldali egy listát ad vissza, a jobb oldali pedig egy generátort.

```
# Létrehoz egy listát és visszaadja
def letter_range(a, z):
    result = []
    while ord(a) < ord(z):
        result.append(a)
        a = chr(ord(a) + 1)
    return result
```

```
# Kérésre szolgáltatja az egyes
# értékeket
def letter_range(a, z):
    while ord(a) < ord(z):
        yield a
        a = chr(ord(a) + 1)
```

Az eredmény elemeit mindkét esetben sorra vehetjük egy `for` ciklussal, valahogy így: `for letter in letter_range("m", "v"):`. Ha azonban a kapott betűk listájára vagyunk kíváncsiak, a `letter_range("m", "v")` csak a jobb oldali esetben elegendő, míg a bal oldali generátorfüggvénynél a `list(letter_range("m", "v"))` alakot kell használnunk.

A generátorfüggvényeket és -tagfüggvényeket (valamint a generátorkifejezéseket) bővebben a 8. fejezetben tárgyaljuk.

A `__getitem__()` tagfüggvényt nem kell újra megvalósítanunk, hiszen az alaposztálybeli változat nagyszerűen működik, és nincs hatással a kulcsok sorrendjére.

```
def __delitem__(self, key):
    try:
        self.__keys.remove(key)
    except ValueError:
        raise KeyError(key)
    return super().__delitem__(key)
```

Ez a tagfüggvény teszi lehetővé a `del d[key]` alak használatát. Ha a `key` nincs jelen, a `SortedList.remove()` hívás eredményeképpen `ValueError` kivételt kapunk. Ha ez bekövetkezik, elfogjuk, és `KeyError` kivételt váltunk ki helyette, hogy megfeleljünk a `dict` osztály API-jának. Egyébként pedig azt az eredményt adjuk vissza, amit a tagfüggvény alaposztálybeli változatától a kulcshoz tartozó elem törlésével kapunk.

```
def setdefault(self, key, value=None):
    if key not in self:
        self.__keys.add(key)
    return super().setdefault(key, value)
```

A fenti tagfüggvény visszaadja az adott kulcshoz tartozó értéket, amennyiben ez a kulcs megtalálható a szótárban, egyébként pedig létrehoz egy új elemet a meghatározott kulccsal és értékkel, majd visszaadja ez utóbbit. A `SortedDict` esetében biztosítanunk kell, hogy a kulcsot elhelyezzük a kulcsok listájában, amennyiben még nem szerepel a szótárban.

Ha a megadott kulcs jelen van a szótárban, ez a tagfüggvény visszaadja a hozzá tartozó értéket, majd törli a szótárból ezt a kulcs-érték párt. A kulcsot szintén el kell távolítanunk a kulcsok listájából.

A megvalósítás meglehetősen körülményes, mivel a `pop()` tagfüggvénynek két eltérő viselkedést kell mutatnia, hogy megfeleljen a `dict.pop()` működésének. Az első a `d.pop(k);` alakban érhető el – ilyenkor visszakapjuk a `k` kulcshoz tartozó értéket, ha pedig a `k` kulcs nem található, `KeyError` kivételt kapunk. A második használati alak a `d.pop(k, value);`, amelynél szintén a `k` kulcs értékéhez jutunk, ha viszont a kulcs nem elérhető, a `value` értéket kapjuk vissza (ami lehet `None` is). Bárhogy is legyen, ha a `k` kulcs létezik, a tagfüggvény eltávolítja a hozzá tartozó elemet.

```
def popitem(self):
    item = super().popitem()
    self.__keys.remove(item[0])
    return item
```

A `dict.popitem()` tagfüggvény eltávolít egy véletlenszerűen kiválasztott kulcs-érték párt az osztályból, amelyet eredményként visszaad. Először az alaposztálybeli változatot kell meghívnunk, hiszen nem tudjuk előre, hogy melyik elemet fogjuk eltávolítani. Töröljük az elemhez tartozó kulcsot a kulcsok listájából, és visszaadjuk az elemet.

```
def clear(self):
    super().clear()
    self.__keys.clear()
```

Itt töröljük a szótár, valamint a kulcsok listájának összes elemét.

```
def values(self):
    for key in self.__keys:
        yield self[key]
def items(self):
    for key in self.__keys:
        yield (key, self[key])
def __iter__(self):
    return iter(self.__keys)
keys = __iter__
```

A szótárak négy olyan tagfüggvénnyel rendelkeznek, amelyek bejárókat adnak vissza – a `dict.values()` a szótár értékeit, a `dict.items()` a kulcs-érték elemeket, a `dict.keys()` a kulcsokat, az `__iter__()` különleges tagfüggvény pedig kulcsokat vár, és az `iter(d)` alakot támogatja. (Valójában a tagfüggvények alaposztálybeli változatai szótár-nézeteket adnak vissza, de a legtöbb esetben az itt megadott bejárók viselkedése azonos.)

Mivel az `__iter__()` és a `keys()` tagfüggvények viselkedése megegyezik, a `keys()` megvalósítása helyett egyszerűen létrehozhatunk egy `keys` nevű objektumhivatkozást, amellyel az `__iter__()` tagfüggvényre hivatkozunk. Így a `SortedDict` felhasználói a `d.keys()` vagy az `iter(d)` hívásával a szótár kulcsait felsoroló bejárót kaphatnak – hasonlóan ahhoz, ahogy a `d.values()` hívással kapott bejáró az értékeket veszi sorra.

A values() és az items() generátor-tagfüggvények – működésükről röviden az előző keretes írásban szóltunk. Mindkét esetben a kulcsok rendezett listáján haladnak végig, ami azt jelenti, hogy a kulcsok sorrendjét követik (a kulcsok sorrendje a kezdeti értékadásnál meghatározott kulcsfüggvénytől függ). Az items() és a values() tagfüggvények esetében az értékeket d[k] alakban olvassuk ki (ilyenkor a háttérben a dict.__getitem__() működik), hiszen a self objektumokat kezelhetjük dict típusúként.

Generátorok ☞ 346

```
def __repr__(self):
    return object.__repr__(self)
def __str__(self):
    return ("{" + ", ".join(["{0!r}: {1!r}".format(k, v)
                             for k, v in self.items()]) + "}")
```

A SortedDict számára nem adhatunk meg az eval() segítségével kiértékelhető ábrázolást, ugyanis ilyen a kulcsfüggvényhez sem áll rendelkezésre. A __repr__() megvalósításánál ezért átugorjuk a dict.__repr__() tagfüggvényt, és inkább az ősváltozatot (object.__repr__()) használjuk. Így olyasforma karakterláncot kapunk, mint általában az eval() révén ki nem értékelhető objektumoknál, vagyis valami ilyesmit: '<SortedDict.SortedDict object at 0xb71fff5c>'.

A SortedDict.__str__() tagfüggvényt magunk valósítottuk meg, mivel az elemeket a kulcsok szerint rendezett alakban szeretnénk kiírni. A tagfüggvényt így is megírhattuk volna:

```
items = []
for key, value in self.items():
    items.append("{0!r}: {1!r}".format(key, value))
return "{" + ", ".join(items) + "}"
```

Listaértelmezés használatával rövidebb alakot kapunk, és az items átmeneti változóra sincs szükség.

Az alaposztály dict.get(), dict.__getitem__() (a v = d[k] alakhoz), dict.__len__() (a len(d) használatához), valamint dict.__contains__() (az x in d alakhoz) tagfüggvényei tökéletesen megfelelnek a leszármazottban is, hiszen nem befolyásolják a kulcsok sorrendjét – ezért nem is kell őket újra megvalósítanunk.

A dict osztály utolsó tagfüggvénye, amelynek a megvalósítására szükségünk van, a copy().

```
def copy(self):
    d = SortedDict()
    super(SortedDict, d).update(self)
    d.__keys = self.__keys.copy()
    return d
```

Választhattuk volna az egyszerűbb, `def copy(self): return SortedDict(self)` alakot, de mégis a fenti mellett döntöttünk, ugyanis így elejét vehetjük a már rendezett kulcsok újrarendezésének. Létrehozunk egy üres rendezett szótárat, majd az alaposztály `dict.update()` tagfüggvénye révén frissítjük az eredeti rendezett szótár elemeivel, így elkerülhetjük a `SortedDict.update()` újbóli megvalósítását, és a szótár `self.__keys` `SortedList` objektumát az eredeti sekély másolatával helyettesítjük.

Ha a `super()` tagfüggvényt paraméterek nélkül hívjuk meg, az alaposztályon, illetve a `self` objektumon fejti ki a hatását. Ugyanakkor tetszőleges objektumra vagy osztályra is alkalmazhatjuk, ha átadjuk számára azt. Ilyenkor a tagfüggvény a megadott osztály közvetlen ősére hat, így esetünkben a kód egyenértékű ezzel: `dict.update(d, self)`.

Mivel a Python rendezési algoritmusa igen gyors, és a működése különösen alkalmas részlegesen rendezett listák kezelésére, az így nyert hatékonyság kicsi vagy elhanyagolható mértékű a hatalmas szótárak esetét kivéve. Mindazonáltal a fenti kód legalább elviekben mutatja, hogy az egyedi `copy()` tagfüggvények használata hatékonyabb lehet, mint a Python beépített típusai által támogatott `copy_of_x = ClassOfX(x)` alak. A `SortedList` esetéhez hasonlóan a `__copy__ = copy` beállítással éltünk, így a `copy.copy()` függvény a saját kódja helyett a mi egyedi másoló tagfüggvényünket használja:

```
def value_at(self, index):
    return self[self.__keys[index]]
def set_value_at(self, index, value):
    self[self.__keys[index]] = value
```

Ez a két utolsó tagfüggvény a `dict` API bővítése, amelyekre azért volt szükség, mert a `SortedDict` a `dict` osztálytól eltérően rendezett, vagyis a kulcsok sorszámai is értelmet nyernek. Így például a szótár első eleme a `0` sorszámmal határozható meg, az utolsó pedig `len(d)-1`-gyel. Mindkét tagfüggvény olyan szótárelemet ér el, amelynek a kulcsa a kulcslistában az `index`-edik elem. Az öröklésnek köszönhetően a `SortedDict` objektum értékeit elérhetjük az elemelérési művelettel (`[]`) közvetlenül a `self`-en keresztül, hiszen ez egyben `dict` típusú is. Ha tartományon kívüli sorszámot adunk meg, `IndexError` kivételt kapunk.

Ezzel a `SortedDict` osztály megvalósításának végére értünk. Ritka, hogy ehhez hasonló teljes általános gyűjteményosztályokat kell létrehoznunk, de ha mégis erre van szükség, a Python különleges tagfüggvényei lehetővé teszik, hogy az osztályt teljesen beépítsük a nyelv szerkezetébe, így a felhasználók úgy kezelhetik, mintha a beépített típusok közül származna.

Összefoglalás

Ebben a fejezetben körbejártuk a Python objektumközpontú programozási lehetőségeinek alapjait. Először megismerkedtünk a tisztán eljárásközpontú megközelítés bizonyos hátrányaival, amelyek az objektumközpontú módszerekkel elkerülhetők. Ezt követően betekintést nyerhettünk az objektumközpontú világ fogalmaiba – így találkozhattunk több szinonimával is, mint az *alaposztály* és az *ős*.

Megtanultuk, hogyan hozhatunk létre egyszerű osztályokat adatjellemzőkkel és egyedi tagfüggvényekkel, és szó esett az öröklésről is, amelynek során már létező osztályokat egészíthetünk ki újabb adatjellemzőkkel és tagfüggvényekkel, továbbá bizonyos tagfüggvényeket kiiktathatunk. Ez utóbbira akkor lehet szükség, ha az osztályunkat egy másikból származtatjuk, de korlátozni szeretnénk az onnan kapott egyes tagfüggvények használatát. Ezt a módszert körültekintően kell alkalmaznunk, ugyanis ezzel elveszítjük a többalakúságot – vagyis az új osztályt nem mindenütt használhatjuk a régi helyében.

Az egyedi osztályokat tökéletesen beépíthetjük a Python szerkezetébe, így pontosan olyan alakban használhatók, mint a nyelv beépített és könyvtári osztályai. Mindezt a különleges tagfüggvények megvalósításával érhetjük el. Láthattuk, hogy a különleges tagfüggvényekkel miként támogathatjuk az összehasonlítást, hogyan biztosíthatunk karakterlánc- és ábrázolási alakot az objektumainknak, továbbá hogyan alakíthatjuk át az objektumokat más (például `int` vagy `float`) típusúvá, amennyiben ennek van értelme. Külön foglalkoztunk a `__hash__()` tagfüggvény megvalósításával, amely lehetőséget ad arra, hogy az egyedi osztályunk példányai szótárak kulcsaiként és halmazok elemeiként szerepelhessenek.

Az adatjellemzők önmagukban nem képesek biztosítani, hogy csak érvényes értékeket fogadjanak. Láthattuk, milyen egyszerűen helyettesíthetjük az adatjellemzőket tulajdonságokkal – így létrehozhatunk csak olvasható tulajdonságokat, az írhatóak esetében pedig könnyedén biztosíthatjuk az értékek ellenőrzését.

Az általunk létrehozott osztályok többsége „nem teljes", ugyanis jobbára csak azokat a tagfüggvényeket írjuk meg, amelyekre valóban szükség van. Ez persze minden további nélkül megtehető a Pythonban, de emellett teljes egyedi osztályokat is létrehozhatunk, amelyek minden elvárt tagfüggvényt elérhetővé tesznek. Először megtanultuk, hogyan hozzunk létre ilyen, egy értéket tároló osztályokat, mind a nulláról indulva, mind pedig az öröklés révén, majd pedig szó esett a gyűjteményosztályok megvalósításáról is. Az egyedi gyűjteményosztályok ugyanazokat a lehetőségeket képesek biztosítani, mint beépített társaik, így elérhetővé tehetik az `in` és az elemelérési (`[]`) műveletet, valamint a `len()`, az `iter()` és a `reversed()` függvényeket.

Megtudtuk, hogy az objektumok létrehozása és kezdeti értékadása két önálló lépés, amelyek szabályozását a Python lehetővé teszi, jóllehet jobbára csak a kezdeti értékadás testreszabásával foglalkozunk. Azt is megtanultuk, hogy az állandó értékű adatjellemzőket

nyugodtan visszaadhatjuk, míg a változtathatók esetében jobb, ha csak másolatot szolgáltatunk, ellenkező esetben ugyanis az objektum belső állapota nyilvánossá válhat, és esetleg véletlenül ezen keresztül érvényteleníthetjük az objektumot.

A Python egyszerű, statikus és osztálytagfüggvényeket, valamint modulfüggvényeket tesz elérhetővé a számunkra. Láttuk, hogy a tagfüggvényeink többsége az első csoportba tartozik, de olykor hasznát vesszük az osztálytagfüggvényeknek is. Ugyanakkor a statikus tagfüggvények lehetőségeit igen ritkán vesszük igénybe, mivel az osztálytagfüggvények és a modulfüggvények szinte minden esetben jobban használhatók helyettük.

A beépített `repr()` függvény egy objektum `__repr__()` különleges tagfüggvényét hívja meg. Ahol csak lehetséges, törekszünk arra, hogy az `eval(repr(x)) == x` azonosság fennálljon – láttuk, miként gondoskodhatunk erről. Ha mégsem tudunk ilyen ábrázoló karakterláncot létrehozni, az alaposztály `object.__repr__()` tagfüggvényéhez folyamodunk, ami szabványos eredményt ad – így azonban nem kapjuk vissza a kiértékeléssel az eredeti objektumot.

A beépített `isinstance()` függvénnyel végzett típusvizsgálat jelenthet némi hatékonyságnövekedést, de akik igazán komolyan veszik az objektumközpontú programozást, tartózkodnak a használatától. Az alaposztály tagfüggvényeit a beépített `super()` függvénnyel érhetjük el, amelynek a használata alapvető fontosságú, amikor egy tagfüggvény alosztálybeli újramegvalósításában az alaposztály megfelelő tagfüggvényét kell használnunk.

A generátorfüggvények és -tagfüggvények lépésről lépésre végzik a kiértékelést, vagyis az eredményt (a `yield` utasítás révén) értékenként, kérésre adják vissza, ha pedig ez nem lehetséges, `StopIteration` kivételt váltanak ki. A generátorokat mindenütt használhatjuk, ahol a kódban bejárókra (iterátorokra) van szükség, véges generátorok esetén pedig a lehetséges értékeiket listákban, illetve rendezett sorozatokban tárolhatjuk, ha a visszakapott bejárót átadjuk a `list()` vagy a `tuple()` függvénynek.

Az objektumközpontú programozás szinte minden esetben egyszerűbbé teszi a kódot a tisztán eljárásközpontú megközelítéshez képest. Az egyedi osztályokkal biztosíthatjuk, hogy csak a megfelelő műveletek legyenek elérhetők (hiszen csak ezeket a tagfüggvényeket valósítjuk meg), továbbá azt, hogy egyetlen művelet se tegye az objektum adatait érvénytelenné (erre a tulajdonságok adnak lehetőséget). Ahogy ráérzünk az objektumközpontú megközelítés ízére, úgy távolodik el a programozási stílusunk a globális adatszerkezetektől és az ezeket elérő globális függvényektől, és úgy használunk egyre inkább osztályokat és hozzájuk tartozó tagfüggvényeket. Az objektumközpontú programozás lehetővé teszi, hogy egységbe zárjuk az adatokat és azokat a tagfüggvényeket, amelyek alkalmasak ezen adatok kezelésére. Ezzel megelőzhetjük az adatok és a függvények keveredését, és programjaink könnyebben fenntarthatóvá válnak, hiszen az egyes működési egységek önálló osztályokban kapnak helyet.

Gyakorlatok

Az első két gyakorlatban a fejezetben bemutatott osztályok módosításával foglalkozunk, míg a másik kettőben teljesen új egyedi osztályokat kell létrehoznunk.

1. Módosítsuk a `Point` osztályt (lásd a `Shape.py` vagy a `ShapeAlt.py` fájlban) úgy, hogy támogassa az alábbi műveleteket, ahol `p`, `q` és `r` `Point` objektumok, `n` pedig egy szám:

```
p = q + r    # Point.__add__()
p += q       # Point.__iadd__()
p = q - r    # Point.__sub__()
p -= q       # Point.__isub__()
p = q * n    # Point.__mul__()
p *= n       # Point.__imul__()
p = q / n    # Point.__truediv__()
p /= n       # Point.__itruediv__()
p = q // n   # Point.__floordiv__()
p //= n      # Point.__ifloordiv__()
```

   A hozzárendelési tagfüggvények megvalósítása négy-négy sor hosszú, ha a `def` sorát is beleszámítjuk, míg társaik ugyanígy számítva két sorból állnak – emellett természetesen igen egyszerűek és hasonlítanak egymásra. Némi leírást és pár tesztet is mellékelve összesen nagyjából százharminc új sorra lesz szükség. Egy lehetséges megoldást mutat a `Shape_ans.py`, valamint a `ShapeAlt_ans.py` fájl.
2. Módosítsuk az `Image.py` osztályt úgy, hogy elérhetővé tegye a `resize(width, height)` tagfüggvényt. Ha az új szélesség vagy magasság kisebb, mint az aktuális érték, az új határokon kívül eső pontokat törölnünk kell. Amennyiben akár a szélesség, akár a magasság paramétereként a `None` értéket adjuk meg, használjuk az eredeti beállítást. Végül pedig gondoskodjunk a `self.__colors` halmaz frissítéséről, és egy logikai értéket visszaadva jelezzük, hogy történt-e módosítás. A tagfüggvényt kevesebb mint 20 sorban elkészíthetjük (illetve 35-ben, amennyiben egyszerű leírást és egy tesztet is mellékelünk), és az `Image_ans.py` fájlban megtekinthetünk egy lehetséges megoldást.
3. Készítsünk el egy `Transaction` nevezetű osztályt, amely egy összeget, egy dátumot, egy pénznemet (utóbbi esetben az alapértelmezés az amerikai dollár, `"USD"`), egy dollárátváltási arányt (alapértelmezés szerint `1`), valamint egy leírást (alapértelmezés: `None`) vár. Minden esetben alkalmazzunk privát adatjellemzőket. Hozzuk létre a következő csak olvasható tulajdonságokat: `amount`, `date`, `currency`, `usd_conversion_rate`, `description` és `usd` (amelyet az `amount * usd_conversion_rate` képlettel számítunk ki). Az osztály nagyjából hatvan sorban megvalósítható – beleértve ebbe néhány tesztet is. Az `Account.py` egy lehetséges megoldást ad erre a gyakorlatra (és a következőre).
4. Valósítsuk meg az `Account` osztályt, amely egy számla számát, nevét, valamint egy `Transaction` objektumokból álló listát tartalmaz. A számlaszám csak olvasható legyen, a név viszont írható és olvasható, de egy feltételezéssel biztosítsuk, hogy

a hossza elérje a négy karaktert. Az osztály támogassa a beépített `len()` függvényt (ez adja vissza a tranzakciók számát), továbbá hozzunk létre két csak olvasható számított tulajdonságot: a `balance` adja meg a számlaegyenleget amerikai dollárban, az `all_usd` értéke pedig legyen `True`, ha minden tranzakció dollárban megy végbe, egyébként pedig `False`. Szükség van három további tagfüggvényre: az `apply()` segítségével elvégezhetünk egy tranzakciót, a `save()` és a `load()` tagfüggvényekkel pedig adatokat tárolunk. Ehhez bináris befőttet készítünk, amelynek a fájlneve a számlaszám, a kiterjesztése pedig `.acc`. A `save()` és a `load()` feladata, hogy biztosítsa a számla számának, nevének és tranzakciós adatainak tárolását és betöltését. Az osztály mindösszesen – egyszerű tesztekkel, valamint a mentés és betöltés lehetőségeivel együtt – nagyjából 90 sorban elkészíthető. Használjunk a `name = os.path.join(tempfile.gettempdir(), account_name)` sorhoz hasonló kódrészletet egy ideiglenes fájlnév létrehozására, ha pedig elkészültünk a tesztekkel, töröljük ezt a fájlt. Egy lehetséges megoldást találhatunk az `Account.py` fájlban.

7

Fájlkezelés

- Bináris adatok írása és olvasása
- Szövegfájlok írása és feldolgozása
- XML-fájlok írása és olvasása
- Véletlen elérésű bináris fájlok

A legtöbb program a működése során fájlokból olvas és fájlokba ír: például adatot, állapotinformációt. A Python többféle módon támogatja a fenti műveletek megvalósítását. A szövegfájlok kezeléséről esett már néhány szó a 3. fejezetben, és érintőlegesen az előző fejezetben is előkerült a téma. Ebben a fejezetben lényegesen mélyebben tárgyaljuk a fájlkezelés témakörét.

A fejezetben bemutatott valamennyi megoldás független a használt rendszertől. Ez annyit tesz, hogy a példaprogramok által az egyik gép processzorával és operációs rendszerével mentett fájlok a másik gép processzorával és operációs rendszerével is beolvashatók. Mindez az Olvasó által készített programok esetében is igaz lehet, amennyiben a példaprogramokkal megegyező módszereket használja bennük.

A fejezet első három szakasza azt a gyakori esetet tárgyalja, amikor teljes adatgyűjtemények lemezre mentését, illetve lemezről történő beolvasását végezzük el. Az első szakasz azt mutatja be, hogy miként tehető ez meg bináris fájlformátumok használatával. Először „befőtteket" fogunk használni – akár tömörítetteket is –, utána pedig befőttek nélkül végezzük el a munkát. A második rész a szövegfájlok kezelését mutatja be. A szövegek írása egyszerű, a visszaolvasásuk azonban nem feltétlenül az, főleg ha nem csak szöveget, hanem dátumokat vagy számokat is vissza kell olvasnunk. A szövegfeldolgozásnak két megközelítését mutatjuk be: a kézi feldolgozás mellett megismerjük a szabályos kifejezéseken alapuló módszert is. A harmadik szakasz az XML-fájlok olvasását és írását mutatja be. Ebben a részben az írást és a feldolgozást elvégezzük elemfákkal, illetve a DOM (Document Object Model, dokumentum-objektummodell) használatával is. Az írást elvégezzük kézi módszerrel is, feldolgozáskor pedig a SAX (Simple API for XML) használatát is kipróbáljuk.

A negyedik szakasz a véletlen vagy tetszőleges elérésű bináris fájlok kezelését mutatja be. Ennek akkor vesszük hasznát, ha az egyes adatok azonos méretűek, és több adatelemünk van, mint amennyit el akarunk, illetve el tudunk helyezni a gép memóriájában.

Teljes gyűjtemények tárolására melyik fájlformátum a legmegfelelőbb? A bináris fájlok, a szövegfájlok, vagy az XML? Melyik az egyes formátumok kezelésére a legmegfelelőbb módszer? Ezek a kérdések túlzottan függnek az adott helyzettől ahhoz, hogy egyetlen helyes válasz adható legyen rájuk, már csak azért is, mert nem sorolhatók fel az egyes formátumok és kezelési módok előnyei, illetve hátrányai. Bemutatjuk az egyes kombinációkat, így segítve az Olvasót, hogy megalapozott döntést hozhasson a konkrét esetekben.

7.1. ábra *Repülőgép-balesetek tárolására szolgáló adatbázisrekordok*

Név	Adattípus	Megjegyzés
`report_id`	`str`	Legalább nyolc karakter, nincs benne szóköz
`date`	`datetime.date`	
`airport`	`str`	Nem lehet üres, nincs benne újsor karakter
`aircraft_id`	`str`	Nem lehet üres, nincs benne újsor karakter
`aircraft_type`	`str`	Nem lehet üres, nincs benne újsor karakter
`pilot_percent_hours_on_type`	`float`	Tartomány 0.0-tól 100.0-ig
`pilot_total_hours`	`int`	Pozitív egész szám
`midair`	`bool`	
`narrative`	`str`	Többsoros

A bináris formátumok mentése és betöltése rendszerint nagyon gyors, a fájlok mindemellett igen tömörek lehetnek. A bináris adatok esetében nincs szükség feldolgozásra, ugyanis ilyenkor az egyes adattípusokba tartozó változókat a memóriabeli ábrázolásuknak megfelelően tároljuk. A binárisan tárolt adatokat mi emberek nem tudjuk átalakítás nélkül elolvasni, illetve írni, a pontos formátum ismerete nélkül pedig lehetetlen a programunktól különálló, a bináris adatokat kezelni képes eszközt kialakítani.

A szövegfájlokat az ember azonnal képes olvasni és szerkeszteni, ezáltal könnyebbé válik a szövegfájlok programunktól különálló eszközzel való feldolgozása, szövegszerkesztővel való szerkesztése. A szöveges formátumú fájlok feldolgozása nehézségeket rejthet magában, ha pedig a formátum sérül – mondjuk a kellő odafigyelés nélkül végzett szerkesztés miatt –, sokszor gondot jelent a megfelelő hibaüzenet megjelenítése.

Az XML formátumú fájlok ember által olvashatók és szerkeszthetők, ugyanakkor gyakran válnak túlontúl részletessé, és gyakran eredményeznek túlméretezett fájlokat. A szöveges formátumokhoz hasonlóan az XML formátumok feldolgozása is megoldható a programunktól különálló eszközökkel. Az XML formátumú fájlok feldolgozása egyszerű – már amennyiben nem saját kezűleg óhajtjuk elvégezni a feladatot, hanem valamilyen XML-feldolgozót használunk –, és néhány XML-feldolgozó igen ügyes a hibajelzéseket illetőleg. Az XML-feldolgozók igen lassúak tudnak lenni, azaz a nagyon nagy XML-fájlok beolvasása lényegesen hosszabb időt vehet igénybe a megfelelő bináris, illetve szöveges fájlok beolvasásánál. Az XML formátumú fájlok metaadatokat is tartalmazhatnak – például a karakterkódolás is megadható –, így a hordozásuk egyszerűbb a szöveges fájlokénál.

A végfelhasználók számára rendszerint a szöveges állományok használata a legkényelmesebb, de néha a teljesítménnyel kapcsolatos megfontolásokból kifolyólag a bináris formátum az egyetlen szóba jöhető választás. Mindazonáltal minden esetben érdemes biztosítani az XML-fájlok betöltésének és mentésének lehetőségét, hiszen az ilyen formátumú fájlok feldolgozása úgy lehetséges külső eszközök felhasználásával, hogy közben nem kell lemondanunk az optimális szöveges, illetve bináris formátum használatáról a program normális működése közben.

7.2. ábra

A repülőgép-baleseteket tároló különböző formátumú fájlok írásának és olvasásának összehasonlítása

Formátum	Olvasó/Író	Olvasó + író kód sorainak száma		Kód sorainak száma összesen	Kimeneti állomány mérete (~KB)
Bináris	Befőtt (gzip tömörítés)	20 + 16	=	36	160
Bináris	Befőtt	20 + 16	=	36	416
Bináris	Kézi (gzip tömörítés)	60 + 34	=	94	132
Bináris	Kézi	60 + 34	=	94	356
Szöveges	Szabályos kifejezés olvasó/kézi író	39 + 28	=	67	436
Szöveges	Kézi	53 + 28	=	81	436
XML	Elemfa	37 + 27	=	64	460
XML	DOM	44 + 36	=	80	460
XML	SAX olvasó/kézi író	55 + 37	=	92	464

A fejezet első három szakaszában ugyanazt az adatgyűjteményt használjuk: repülőgép-baleseteket tároló adatbázisrekordok halmazát. A 7.1. ábra a repülőgép-baleseteket tároló adatbázisrekordok mezőinek nevét, adattípusát és érvényességi megszorításait mutatja be. A feldolgozott adat nem igazán számít, a fontos az, hogy megtanuljuk az alapvető adattípusok – karakterláncok, egész és lebegőpontos számok, logikai értékek és dátumok – feldolgozását, mégpedig azért, mert ha az előző típusokat fel tudjuk dolgozni, akkor bármilyen egyéb adattípussal is boldogulni fogunk.

Az, hogy a bináris, a szöveges és az XML formátum esetében is ugyanazt a repülőgép-balesetekről szóló adathalmazt dolgozzuk fel, lehetővé teszi a különféle formátumok és a kezelésükhöz szükséges kód összehasonlítását, illetve a közöttük lévő különbségek kiemelését. A 7.2. ábra bemutatja és összehasonlítja az egyes formátumok olvasásához és írásához szükséges kódsorok számát, illetve ezek összegét is.

A fájlméretek csak közelítőek, és az adott, 596 repülőgép-balesetet tartalmazó minta esetében érvényesek.* Ha ugyanazt az adatot más fájlnév használatával, valamilyen tömörített formátumban mentjük, az eredmény néhány bájtnyi eltérést mutathat, lévén a fájlnevek is bekerülnek a tömörített adatba, és a fájlnevek hossza eltérő. Ehhez hasonlóan az XML-fájlok mérete is változó lehet, hiszen néhány XML-író az idézőjeleket `&quot`, az aposztrófokat pedig `&apos` szöveggel helyettesíti a szöveges adatok belsejében, míg más programok ezt az átalakítást nem végzik el.

Az első három szakaszban lévő kódszövegek egyazon programból, a `convert-incidents.py` nevűből származnak. Ez a program képes a repülőgép-balesetek adatait az egyik formátumból beolvasni, majd egy másikba kiírni. Íme a program konzolon megjelenített súgószövege (a kimenetet kissé módosítottuk, hogy a könyv oldalszélességének megfeleljen):

```
Usage: convert-incidents.py [options] infile outfile
Reads aircraft incident data from infile and writes the data to
outfile. The data formats used depend on the file extensions:
.aix is XML, .ait is text (UTF-8 encoding), .aib is binary,
.aip is pickle, and .html is HTML (only allowed for the outfile).
All formats are platform-independent.

Options:
  -h, --help        show this help message and exit
  -f, --force       write the outfile even if it exists [default: off]
  -v, --verbose     report results [default: off]
  -r READER, --reader=READER
                    reader (XML): 'dom', 'd', 'etree', 'e', 'sax', 's'
                    reader (text): 'manual', 'm', 'regex', 'r'
                    [default: etree for XML, manual for text]
  -w WRITER, --writer=WRITER
                    writer (XML): 'dom', 'd', 'etree', 'e',
                    'manual', 'm' [default: manual]
  -z, --compress    compress .aib/.aip outfile [default: off]
  -t, --test        execute doctests and exit (use with -v for verbose)
```

A kapcsolók használata bonyolultabb annál, mint normális esetben lehetne, hiszen egy végfelhasználót vajmi kevéssé érdekel, hogy melyik olvasási és írási módszert használjuk egy adott formátum esetében. Ha a programot valós használatra szánnánk, az olvasási és

* A példában használt adatok valós repülőgép-balesetek adatain alapulnak. Az adatok forrása az FAA (az Egyesült Államok kormányának Federal Aviation Administration nevű intézménye, `www.faa.gov`).

az írási módszert beállító kapcsolók nem léteznének, és a módszerek közül formátumonként csak egyet-egyet valósítanánk meg. A `test` kapcsoló csak a kód tesztelésére szolgál, és ugyanígy nem volna jelen a program egy kiadott változatában.

A programban egyetlen saját kivételt adunk meg:

```
class IncidentError(Exception): pass
```

A repülőgép-baleseteket `Incident` osztályú objektumokként tartjuk nyilván. Íme a `class` utasítást tartalmazó sor és az osztály előkészítése:

```
class Incident:

def __init__(self, report_id, date, airport, aircraft_id,
             aircraft_type, pilot_percent_hours_on_type,
             pilot_total_hours, midair, narrative=""):
    assert len(report_id) >= 8 and len(report_id.split()) == 1, \
           "invalid report ID"
    self.__report_id = report_id
    self.date = date
    self.airport = airport
    self.aircraft_id = aircraft_id
    self.aircraft_type = aircraft_type
    self.pilot_percent_hours_on_type = pilot_percent_hours_on_type
    self.pilot_total_hours = pilot_total_hours
    self.midair = midair
    self.narrative = narrative
```

A jelentésazonosító érvényesítése akkor történik, amikor az osztályból származó objektumot példányosítjuk; elérése pedig a `report_id` nevű, csak olvasható tulajdonság olvasásával történik. Minden egyéb adattulajdonság írható és olvasható. Például íme a `date` (dátum) tulajdonság kódja:

```
@property
def date(self):
    return self.__date
@date.setter
def date(self, date):
    assert isinstance(date, datetime.date), "invalid date"
    self.__date = date
```

Minden további tulajdonság ezt a mintát követi, így az ismertetésüktől itt most eltekintünk, hiszen különbségeket mindössze az `assert` utasítást tartalmazó sorokban találunk.
Az `assert` utasítások használata miatt a program leáll, amennyiben érvénytelen adatok alapján szeretnénk létrehozni egy `Incident` osztályú objektumot, és olyankor is, ha egy már létező `Incident` osztályú objektum valamelyik tulajdonságának érvénytelen adatot próbálunk értékül adni. Azért döntöttünk ilyen szigorú elbírálás mellett az érvénytelen adatokat illetően, mert biztosak akarunk lenni benne, hogy a mentett és beolvasott adat mindig érvényes. Ha pedig mégsem volna az, jobb, ha a program futása némi panaszkodás mellett megszakad, mint ha a problémát elsunnyogva folytatná a futást.

A baleseteket tároló objektumokat egy `IncidentCollection` osztályú objektumban gyűjtjük. Az osztály a `dict` osztály leszármazottja, amelytől sok mindent örököl: például az elemelérési műveletet (`[]`), amelyet az egyes balesetek kiolvasására, beállítására és törlésére fogunk használni. Íme a `class` utasítást tartalmazó sor és az osztály néhány tagfüggvénye:

```
class IncidentCollection(dict):

    def values(self):
        for report_id in self.keys():
            yield self[report_id]

    def items(self):
        for report_id in self.keys():
            yield (report_id, self[report_id])

    def __iter__(self):
        for report_id in sorted(super().keys()):
            yield report_id

keys = __iter__
```

Az előkészítő újbóli megvalósítását nem kellett elvégeznünk, hiszen a `dict.__init__()` megfelelő. A kulcsok a jelentésazonosítók (`report_id`), az értékek pedig `Incident` osztályú objektumok. A `values()`, az `items()` és a `keys()` tagfüggvény újbóli megvalósítására azért volt szükség, hogy a bejárásuk a jelentésazonosítók sorrendjében történjen. A megoldás azért megfelelő, mert a `values()` és az `items()` tagfüggvény az `IncidentCollection.keys()` tagfüggvénytől visszakapott kulcsokat járja be, amely viszont a bejárást rendezett sorrendben, az ősosztály `dict.keys()` tagfüggvényétől visszakapott értékek alapján végzi. (Az `IncidentCollection.keys()` egyébiránt nem más, mint az `IncidentCollection.__iter__()` tagfüggvény másik neve.)

Az eddigieken felül az `IncidentCollection` osztály rendelkezik egy `export()` és egy `import_()` tagfüggvénnyel. (A név és a zárójel között álló aláhúzásjellel a beépített `import` utasítás nevétől szándékszunk eltérni.) Az `export()` tagfüggvénynek egy fájlnevet adunk át, elhagyható paraméterként pedig egy írási módszer megjelölését, illetve egy tömörítésre utasító jelzést. A tagfüggvény az írási módszer és a fájl kiterjesztése alapján a munkát egy pontosabban meghatározott tagfüggvénynek, például az `export_xml_dom()` vagy az `export_xml_etree()` nevűnek adja át. Az `import_()` tagfüggvény egy fájlnevet és elhagyhatóan egy olvasási módot kap paraméterül, egyebekben a párjához hasonlóan működik. Azok a betöltést végző tagfüggvények, amelyek bináris adatot olvasnak be, nem kapnak jelzést a bemeneti állomány tömörítettségére vonatkozólag – egyedül kell felmérniük a helyzetet, a munkát pedig ennek megfelelően kell elvégezniük.

Bináris adatok írása és olvasása

A bináris formátumban tárolt adatok rendszerint még tömörítés nélkül is a legkevesebb helyet foglalják; rendszerint az írásuk és az olvasásuk is a leggyorsabb. A legegyszerűbb, ha befőtteket használunk, bár a kézzel feldolgozott adatok elvileg kevesebb helyet foglalnak.

Befőttek tömörítéssel vagy anélkül

Figyelem! Python nyelvű programjainkban az adatok mentésére és betöltésére a befőttek (pickle) kínálják a legegyszerűbb módszert, de ahogy azt már az előző fejezetben is jeleztük, a befőttek semmiféle titkosítással nem rendelkeznek: nincs titkosító algoritmus, nincs digitális aláírás. Így az ismeretlen eredetű befőttek veszélyeket rejthetnek magukban. A veszély forrását az jelenti, hogy a befőttek képesek modulok betöltésére és függvények meghívására, vagyis kaphatunk olyan módon kezelt befőttet, amely miatt a parancsértelmező valamilyen káros kódot hajt végre a befőtt betöltésekor. Mindemellett a befőttek igen alkalmasak ideiglenes, egyszer használatos adatok kezelésére, különösen a személyes használatra készült programokban.

Fájlformátumok kialakításakor rendszerint egyszerűbb a betöltést végző kód megírása előtt megírni a mentést végző kódot. Így először azt tekintjük át, hogy miként menthetők a repülőgép-balesetek egy befőttbe.

```
def export_pickle(self, filename, compress=False):
    fh = None
    try:
        if compress:
            fh = gzip.open(filename, "wb")
        else:
            fh = open(filename, "wb")
            pickle.dump(self, fh, pickle.HIGHEST_PROTOCOL)
            return True
        except (EnvironmentError, pickle.PicklingError) as err:
            print("{0}: export error: {1}".format(
                  os.path.basename(sys.argv[0]), err))
            return False
        finally:
            if fh is not None:
                fh.close()
```

Ha a felhasználó kérte a fájl tömörítését, a `gzip` modul `gzip.open()` függvényének használatával nyitjuk meg a fájlt, egyébként maradunk a beépített `open()` függvénynél. Ha az adatok befőzése bináris formátumba történik, a `wb` („write binary", azaz bináris írás) módot kell választanunk. (A Python 3.0-ban a `pickle.HIGHEST_PROTOCOL` a 3-as protokoll, ami egy tömör bináris befőttformátum.*)

* A 3-as protokollt csak a Python 3-ban találjuk meg. Ha azt szeretnénk, hogy a befőttjeinket a Python 2 és a Python 3 egyaránt be tudja olvasni, a 2-es protokollt kell használnunk.

Hibakezelési stratégiául azt választottuk, hogy a hiba felmerülésekor azonnal jelzünk a felhasználónak, a hívónak pedig a sikert vagy hibát jelző logikai változót adunk át. A `finally` utasítással kezdődő blokkal bizonyosodunk meg róla, hogy a műveleteket követően a fájl lezárul – attól függetlenül, hogy volt-e hiba. A 8. fejezetben egy tömörebb formát ismerünk meg a fájlok lezárásának ellenőrzésére, olyat, ami szükségtelenné teszi a `finally` blokk használatát.

Környezetkezelők ☞ 375

A fenti kód igen hasonló az előző fejezetben megismerthez, de van egy aprócska, mégis figyelemreméltó különbség. A befőttben lévő adat szótár típusú. A szótár értékei azonban `Incident` osztályú objektumok, vagyis általunk létrehozott osztályból származnak. A befőttek kezelését végző `pickle` modul elég okos ahhoz, hogy a legtöbb saját készítésű osztály objektumait is anélkül tudja menteni, hogy nekünk bármit kellene tennünk az ügy érdekében.

A befőttek általában logikai, szám és karakterlánc típusú változókat tartalmazhatnak, valamint osztályokból példányosított objektumokat – akár egyedi osztályból származókat is –, amennyiben az adott osztály objektumainak privát `__dict__` változója megfelelő befőttek készítéséhez. Befőtt ezen felül bármely beépített gyűjteményes adattípus változóiból – rendezett sorozatokból, listákból, halmazokból, szótárakból – készíthető, amennyiben bennük csak befőzhető adattípusból származó objektumok vannak. Ez érvényes a bennük lévő gyűjteményes adattípusokra is, vagyis a rekurzív szerkezetek is lehetnek befőzhetők. Lehetőség van továbbá arra is, hogy olyan objektumokat (olyan egyedi osztályok példányait) is befőttbe tegyük, amelyek rendesen nem kerülhetnek befőttbe – például azért, mert valamelyik jellemzőjük megakadályozza. Ilyen esetekben vagy némi segítséget nyújtunk a `pickle` modulnak, vagy saját függvényeket valósítunk meg a befőttbe helyezésre és az abból való kivételre. A `pickle` modul leírásában minden ide vonatkozó részlet megtalálható.

__dict__ ☞ 369

Ha a befőttbe helyezett adatokat vissza szeretnénk olvasni, észre kell vennünk a különbséget a tömörített és tömörítetlen befőttek között. Minden olyan fájl, amelyet a `gzip` tömörítési eljárással tömörítettünk, egy bizonyos *mágikus számmal* kezdődik. A mágikus számok olyan, egy vagy több bájt hosszúságú jelsorozatok, amelyek az állományok elején találhatók, és az állomány típusát jelzik. A `gzip` tömörítésű fájlok esetében ez a mágikus szám a `0x1F 0x8B` bájtpár, amelyet egy `bytes` típusú változóban tárolunk:

```
GZIP_MAGIC = b"\x1F\x8B"
```

A `bytes` adattípusról részletesebb információkat kicsit később, *A bytes és a bytearray adattípus* című keretes szövegben, illetve a típusok tagfüggvényeit felsoroló 7.1., 7.2. és 7.3. táblázatokban találunk.

Íme a kód, amely az `Incident` objektumokból készült befőtt beolvasását végzi:

```
def import_pickle(self, filename):
    fh = None
    try:
        fh = open(filename, "rb")
        magic = fh.read(len(GZIP_MAGIC))
        if magic == GZIP_MAGIC:
            fh.close()
            fh = gzip.open(filename, "rb")
        else:
            fh.seek(0)
        self.clear()
        self.update(pickle.load(fh))
        return True
    except (EnvironmentError, pickle.UnpicklingError) as err:
        print("{0}: import error: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
    finally:
        if fh is not None:
            fh.close()
```

Nem tudjuk, hogy az átadott fájl tömörített-e. Mindkét lehetséges esetben a fájlt `rb` („read binary", azaz bináris olvasás) módban nyitjuk meg először, és beolvassuk az első két bájtot. Ha a két bájt megegyezik a `gzip` tömörítést jelölő két bájttal, lezárjuk a fájlt, és a `gzip.open()` függvénnyel új objektumot hozunk létre. Ha tömörítetlen fájllal állunk szemben, az `open()` függvénytől visszakapott fájlobjektumot használjuk, amelynek a `seek()` tagfüggvényét hívva visszaállítjuk a fájlmutatót az állomány elejére. Így a következő olvasás – amit a `pickle.load()` függvény végez majd – jó helyről indulhat.

A `self` objektumnak nem adhatunk értéket, mert azzal a használatban lévő `IncidentCollection` osztályú objektumot megszüntetnénk. Így inkább az `Incident` objektumok eltakarításával kiürítjük a szótárt, majd a `dict.update()` tagfüggvény segítségével a befőttből betöltött, `IncidentCollection` osztályú szótárban tárolt `Incident` objektumokkal töltjük fel ismét.

Megjegyeznénk, hogy semmi jelentősége sincs, hogy a processzor milyen bájtsorrendet alkalmaz: a mágikus számot bájtonként olvassuk be, az adatok esetében a bájtsorrenddel a `pickle` modul törődik helyettünk.

Nyers bináris adatok tömörítéssel és anélkül

A nyers bináris adatok kezelését végző saját kód megírása a fájlformátumunkat illetőleg teljes irányítást ad a kezünkbe. Akár biztonságosabb is lehet a befőttek használatán alapuló módszernél, hiszen a rossz szándékkal megváltoztatott adatokat ezúttal a saját kódunk kezeli, és nem a parancsértelmező hajtja végre.

Amikor saját bináris fájlformátumot hozunk létre, a fájltípus azonosítására bölcs dolog mágikus számot is létrehozni, a használt fájlformátum-változat azonosítása pedig a fájlban elhelyezett változatszámmal oldható meg. Íme a `convert-incidents.py` programban használt számok:

```
MAGIC = b"AIB\x00"
FORMAT_VERSION = b"\x00\x01"
```

Négy bájton tároljuk a mágikus számot, kettőn a fájlformátum változatszámát. Az, hogy milyen bájtsorrendet használunk, nem érdekes, mert az adatot bájtonként fogjuk írni, nem pedig egész számok bájtábrázolásaként – így minden processzor esetében azonos lesz a leírt adat.

Ha nyers bináris adatot szeretnénk írni és olvasni, valamiféle átalakítást kell elvégeznünk a Python-objektum, illetve az objektum megfelelő bináris ábrázolása között. Amire szükségünk van, annak a legnagyobb része megtalálható a `struct` modulban (a modul rövid leírását a kicsit később következő, *A struct modul* című keretes szövegben találjuk), illetve a `bytes` és a `bytearray` adattípus biztosítja (ezeket röviden *A bytes és a bytearray adattípus* című keretes szöveg ismerteti).

Sajnos a `struct` modul csak egyforma hosszúságú karakterláncok kezelésére képes, nekünk pedig változó hosszúságú karakterláncokra van szükségünk, mind a jelentésazonosító, mind a repülőgép-azonosító esetében, nem különben a repülőtér, a repülőgéptípus és a kísérőszöveg tárolására. Ennek az igénynek a kielégítése végett létrehoztuk a `pack_string()` függvényt. A függvény paramétere egy karakterlánc, a visszatérési értéke pedig egy `bytes` típusú objektum, amely két összetevőt tartalmaz. Az első egy hosszúságot jelölő egész szám, a második pedig egy az előző számnak megfelelő hosszúságú, UTF-8 kódolású szöveg, amely az átadott karakterláncban lévő szöveget ábrázolja.

Helyi függvények
☞ 357

Minthogy az egyetlen olyan hely, ahol a `pack_string()` függvényre szükség van, az `export_binary()` függvény belsejében található, a `pack_string()` függvény megadása az `export_binary()` függvény belsejében történik. Ez annyit jelent, hogy a `pack_string()` függvény az `export_binary()` függvényen kívülről nem látható, és egyértelművé teszi, hogy csak egy helyi segédfüggvényről van szó. Íme az `export_binary()` függvény eleje, benne a teljes beágyazott `pack_string()` függvénnyel:

```
def export_binary(self, filename, compress=False):
    def pack_string(string):
        data = string.encode("utf8")
        format = "<H{0}s".format(len(data))
        return struct.pack(format, len(data), data)
```

A bytes és a bytearray adattípus

A Python két adattípussal segíti a nyers bájtok kezelését: az egyik az állandó értékű `bytes`, a másik a változtatható `bytearray` típus. Mindkét típus nulla vagy több, előjel nélküli, nyolc biten ábrázolható egész szám (azaz bájt) sorozatát tárolja. Minden bájt 0 és 255 közötti értéket vehet fel.

Mindkét típus nagyon hasonló a karakterláncokhoz, és sok a közös tagfüggvény, illetve lehetőség is – ilyen többek között a szeletelés. Ezen felül a `bytearray` típus rendelkezik néhány, a listákra jellemző tagfüggvénnyel. A típusok tagfüggvényeinek listáját a 7.1., 7.2. és 7.3. táblázatban találjuk.

Míg egy `bytes` vagy `bytearray` egy szelete a kiindulóval megegyező típusú objektumot ad vissza, az elemelérési művelet (`[]`) egy egész értéket, amely a megadott bájt értékét hordozza. Például:

```
word = b"Animal"
x = b"A"
word[0] == x      # returns: False  # word[0] == 65;     x == b"A"
word[:1] == x     # returns: True   # word[:1] == b"A"; x == b"A"
word[0] == x[0]   # returns: True   # word[0] == 65;     x[0] == 65
```

Íme néhány további `bytes`, illetve `bytearray` típusú adat:

```
data = b"5 Hills \x35\x20\x48\x69\x6C\x6C\x73"
data.upper() # az eredmény: b'5 HILLS 5 HILLS'
data.replace(b"ill", b"at")                  # az eredmény: b'5 Hats 5 Hats'
bytes.fromhex("35 20 48 69 6C 6C 73")        # az eredmény: b'5 Hills'
bytes.fromhex("352048696C6C73")              # az eredmény: b'5 Hills'
data = bytearray(data)                       # az adat immár bytearray
data.pop(10)                                 # az eredmény: 72 (ord("H"))
data.insert(10, ord("B"))                    # data == b'5 Hills 5 Bills'
```

Azok a tagfüggvények, amelyek csak karakterláncok esetében értelmezhetők – ilyen például a `bytes.upper()` –, feltételezik, hogy ASCII kódolású szöveget tárolunk a változóban. A `bytes.fromhex()` tagfüggvény figyelmen kívül hagyja a térközkaraktereket, és minden kétszámjegynyi karakterlánc-részletet tizenhatos számrendszerű számként értelmez. Vagyis a „35” egy `0x35` értékű bájt lesz, és így tovább.

Karakterkódolás 86

A `str.encode()` tagfüggvény egy `bytes` objektumot ad vissza, benne a megadott karakterkódolás szerint kódolt karakterlánccal. Az UTF-8 karakterkódolás igen kényelmes, ugyanis minden Unicode-karakter tárolására képes, az ASCII-karakterek tárolásakor pedig különösen tömör: mindegyiket egyetlen bájton ábrázolja. A `format` nevű változó értéke egy a `struct` modulban használ-

ható formátumot tárol, amelyet a karakterlánc hossza alapján határozunk meg. Például, ha az „en.wikipedia.org" karakterláncot tekintjük, a `format` változó értéke „`<H16s`" lesz („littleendian" bájtsorrend, kétbájtos előjel nélküli egész, 16 bájtos karakterlánc), a visszaadott objektum pedig a `b'\x10\x00en.wikipedia.org'` értéket fogja hordozni. Igen kényelmes dolog, hogy a Python a `bytes` típusú objektumokat tömör formában, ahol lehet, nyomtatható ASCII-karakterek használatával, ahol pedig nem, ott levédett tizenhatos számrendszerű értékekkel (és pár megkülönböztetett szerepű védőkarakterrel, mint a `\t` és a `\n`) mutatja meg.

A struct modul

A `struct` modulban találjuk többek között a `struct.pack()` és a `struct.unpack()` függvényt, és természetesen a `struct.Struct()` osztályt. A `struct.pack()` függvény paramétere egy a `struct` modulra jellemző, formátumleíró karakterlánc, illetve egy vagy több érték. Visszatérési értéke egy olyan `bytes` objektum, amely az átadott értékeket a megadott karakterlánc szerint formázva tartalmazza. A `struct.unpack()` függvény paramétere egy formátumleírás, és egy `bytes` vagy egy `bytearray` objektum, visszatérési értéke pedig egy olyan rendezett sorozat, amely azokat az értékeket tartalmazza, amelyeket a formátum használatával valamikor régebben becsomagoltunk.

Például:

```
data = struct.pack("<2h", 11, -9)  # data == b'\x0b\x00\xf7\xff'
items = struct.unpack("<2h", data) # items == (11, -9)
```

A formátumleíró karakterláncok egy vagy több karakterből állnak. A legtöbb karakter egy adott típusú értéket képvisel. Ha egy típusból több értékre is szükségünk van, akkor vagy annyiszor írjuk be a típusnak megfelelő karaktert, ahány értékünk van („`hh`"), vagy a karakter előtt álló számmal jelezzük a kívánt mennyiséget – ahogy az előző példában látható („`2h`").

A `struct` modul leírásában sok formátumleíró karaktert találunk. A „b" a nyolc bites előjeles egész számokat jelöli, a „B" a nyolc bites előjel nélküli egész számok jele, a „h" a tizenhat bites előjeles egészek jele (a fenti példákban is ezt használtuk), a „H" a tizenhat bites előjel nélküli egészeket jelöli. Az „i" jelöli a harminckét bites előjeles egészeket, az „I" pedig a harminckét bites előjel nélküli egészeket, a „q" jelöli a hatvannégy bites előjeles egészeket, a „Q" pedig a hatvannégy bites előjel nélküli egészeket. A harminckét bites lebegőpontos számok jele az „f", a hatvannégy biteseké – a Python nyelv `float` típusának megfelelően a „d". A kérdőjel a logikai értékeket jelöli, az „s" a `bytes`, illetve a `bytearray` típus jele, és még igen sok jel létezik.

Néhány adattípus esetében – például a több bájtos egészeknél – a processzor által használt kódolás hatással van a bájtsorrendre. Kikényszeríthetjük egy bizonyos kódolás használatát, ha a formátumleírót egy ennek megfelelő karakterrel kezdjük. Könyvünkben mindig

a „kisebb mint" jel, a „little-endian" kódolás jele szerepel. Azért éppen ez, mert a széles körben használt Intel és AMD processzorok is ezt használják. A „big-endian" kódolás – ha úgy tetszik: hálózati bájtsorrend – jele a „nagyobb mint" jel vagy a felkiáltójel. Ha a kódolást nem adjuk meg, a Python a számítógépnek megfelelő kódolást használja. Érdemes az endian-kódolást mindig megadnunk, hiszen így hordozható adatokat kapunk eredményül.

A `struct.calcsize()` függvény paramétere egy formátumleíró, visszatérési értéke pedig egy szám: ennyi bájtot foglal el az adott formátumleíró alapján formázott adatszerkezet.

A formátumleírók tárolása történhet `struct.Struct()` osztályú objektumokban, amelyeknek paraméterül a formátumleírót adjuk át. Az objektum méretét a `size` jellemző adja meg. Például:

```
TWO_SHORTS = struct.Struct("<2h")
data = TWO_SHORTS.pack(11, -9)      # data == b'\x0b\x00\xf7\xff'
items = TWO_SHORTS.unpack(data)     # items == (11, -9)
```

A `11` mindkét példában `0x000b`, de a little-endian bájtsorrendűvé alakítás miatt ez `0x0b 0x00` formában jelenik meg.

7.1. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (1)*

Utasításforma	Leírás
ba`.append(`*i*`)`	Az *i* 0 és 255 közötti egész számot a *ba* `bytearray` típusú változó végére fűzi.
b`.capitalize()`	A *b* `bytes/bytearray` típusú változó értékét adja vissza, az első karaktert nagybetűre cserélve (már amennyiben az egy ASCII kódolású betű).
b`.center(`*szélesség*`, `*bájt*`)`	A *b* változó szóközökkel vagy a *bájt* értékkel *szélesség* szélességűre kiegészített, középre igazított másolatát adja vissza.
b`.count(`*x*`, `*kezdet*`, `*vég*`)`	Az *x* `bytes/bytearray` típusú érték előfordulásainak számát adja vissza a *b* `bytes/bytearray` típusú változóban (vagy a *b* változó *kezdet*`:`*vég* részében).
b`.decode(`*kódolás*`, `*hiba*`)`	Olyan `str` osztályú objektumot ad vissza, amely a `bytes` típusú adat UTF-8, illetve a *kódolás* értékkel megadott kódolású ábrázolását tartalmazza. A hibák kezelése az elhagyható *hiba* paraméternek megfelelően történik. Karakterkódolás 86 ☞
b`.endswith(`*x*`, `*kezdet*`, `*vég*`)`	`True` értéket ad vissza, ha a *b* változó (vagy annak *kezdet*`:`*vég* része) az *x* `bytes/bytearray` típusú értékkel, vagy az *x* rendezett sorozatban lévő bármely `bytes/bytearray` típusú értékkel végződik; egyéb esetekben `False` a visszaadott érték.

7.1. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (1)* *(folytatás)*

Utasításforma	Leírás
`b.expandtabs(méret)`	A `b` `bytes/bytearray` típusú változó másolatát adja vissza, amelyben a tabulátorokat 8, illetve – ha megadtuk – a `méret` többszöröseinek megfelelő számú szóközre cseréli.
`ba.extend(sorozat)`	A `ba` `bytearray` típusú változót a `sorozat`-ban lévő egész számokkal egészíti ki; a számok 0 és 255 közöttiek lehetnek.
`b.find(x, kezdet, vég)`	A `b` változóban (vagy annak `kezdet:vég` részében) az `x` `bytes/bytearray` típusú érték bal szélső előfordulását adja vissza, illetve `-1`-et, ha `x` nem fordul elő `b`-ben. A jobb szélső előfordulás megtalálása az `rfind()` tagfüggvénnyel lehetséges.
`b.fromhex(h)`	A `h` karakterláncban lévő tizenhatos számrendszerű egész számoknak megfelelő `bytes` osztályú objektumot ad vissza.
`b.index(x, kezdet, vég)`	A `b` változóban (vagy annak `kezdet:vég` részében) az `x` érték bal szélső előfordulását adja vissza, illetve `ValueError` kivételt ad, ha `x` nem fordul elő `b`-ben. A jobb szélső előfordulás megtalálása az `rindex()` tagfüggvénnyel lehetséges.
`ba.insert(p, i)`	A `ba` változóba `p` helyen beszúrja az `i` egész számot, amelynek az értéke 0 és 255 közötti lehet.
`b.isalnum()`	`True` értéket ad vissza, ha a `b` `bytes/bytearray` típusú változó nem üres, és minden karaktere ASCII-szám vagy -betű.
`b.isalpha()`	`True` értéket ad vissza, ha a `b` `bytes/bytearray` típusú változó nem üres, és minden karaktere ASCII-betű.
`b.isdigit()`	`True` értéket ad vissza, ha a `b` `bytes/bytearray` típusú változó nem üres, és minden karaktere ASCII-számjegy.
`b.islower()`	`True` értéket ad vissza, ha a `b` `bytes/bytearray` típusú változó nem üres, van legalább egy ASCII-betű karaktere, és minden betűkaraktere kisbetű.
`b.isspace()`	`True` értéket ad vissza, ha a `b` `bytes/bytearray` típusú változó nem üres, és minden karaktere ASCII-szóközt jelölő karakter.

7.2. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (2)*

Utasításforma	Leírás
`b.istitle()`	`True` értéket ad vissza, ha a `b` változó nem üres, és az angol címeknek megfelelően nagy kezdőbetűs szavakból áll.
`b.isupper()`	`True` értéket ad vissza, ha a `b` változóban legalább egy ASCII-betű van, és minden betű karaktere nagybetű.
`b.join(sorozat)`	Összefűzve visszaadja a `sorozat` sorozatban lévő `bytes/bytearray` típusú változókat, az egyes változók értékei közé beékelve `b`-t, amely üres is lehet.

7.2. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (2)* *(folytatás)*

Utasításforma	Leírás
`b.ljust(szélesség, bájt)`	A `b` `bytes/bytearray` típusú változó szóközökkel vagy az elhagyható `bájt` értékkel `szélesség` szélességűre kiegészített, balra rendezett másolatát adja vissza. Jobbra rendezéshez az `rjust()` tagfüggvény használható.
`b.lower()`	A `b` `bytes/bytearray` típusú változó ASCII szerinti kisbetűs másolatát adja vissza.
`b.partition(elválasztó)`	Három `bytes` típusú változóból álló rendezett sorozatot ad vissza. Az első elem a `b` `bytes/bytearray` típusú változónak a változóban lévő bal szélső `elválasztó` előtti része, ezt követi maga az `elválasztó`, majd `b` `elválasztó` utáni része következik. Ha `b`-nek az `elválasztó` nem része, akkor `b`-t és két `bytes` típusú üres objektumot kapunk vissza. Ha az `elválasztó` jobb szélső előfordulásánál szeretnénk tagolni `b`-t, használjuk az `rpartition()` tagfüggvényt.
`ba.pop(p)`	A `p` helyen álló egész számot kiveszi a `ba` változóból, és visszaadja a hívónak.
`ba.remove(i)`	A `ba` `bytearray` típusú változóból eltávolítja az `i` egész szám első előfordulását.
`b.replace(x, y, n)`	A `b` változó másolatát adja vissza, benne az `x` `bytes/bytearray` típusú érték minden (ha `n` adott, akkor az első `n`) előfordulását `y`-ra cserélve.
`ba.reverse()`	A `ba` `bytearray` típusú változót helyben megfordítja.
`b.split(x, n)`	`x` előfordulásainál szeli `b`-t, legfeljebb `n` alkalommal. Ha `x` nem adott, akkor a szóközöknél szel, ha `n` nem adott, akkor minden lehetséges helyen. Visszatérési értéke egy `bytes` objektumokból álló lista. Jobbról kezdődő szeleteléshez az `rsplit()` tagfüggvényt kell használnunk.
`b.splitlines(f)`	A `b` változó sorvégeknél történt szeletelésével képzett sorokból álló listát adja vissza. A sor végének jelzését elhagyja, kivéve, ha `f` értéke `True`.
`b.startswith(x, kezdet, vég)`	`True` értékkel tér vissza, ha a `b` `bytes/bytearray` típusú változó (vagy annak `kezdet:vég` közötti része) az `x` `bytes/bytearray` típusú értékkel, vagy az `x` rendezett sorozatban lévő bármely `bytes/bytearray` típusú értékkel kezdődik, egyéb esetekben `False` a visszaadott érték.
`b.strip(x)`	`b` olyan másolatát adja vissza, amelynek az elejéről és végéről a szóközöket (illetve az `x` `bytes/bytearray` típusú értéket) eltávolította. Az `lstrip()` tagfüggvény csak a paraméter elejénél, az `rstrip()` tagfüggvény csak a paraméter végénél vág.

7.2. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (2)*

(folytatás)

Utasításforma	Leírás
`b.swapcase()`	*b* olyan másolatát adja vissza, amelyben a kisbetűs ASCII-karakterek nagybetűsek lesznek, és fordítva.
`b.title()`	*b* olyan másolatát adja vissza, amelyben minden egyes szó első ASCII-betűje nagybetűs, az összes többi pedig kisbetűs.
`b.translate(bt, d)`	*b* olyan másolatát adja vissza, amelyben nincs *d*-ben előforduló bájt, és amelyben miden egyéb bájt a `bt` `bytes` típusú érték „bájtadik" bájtjára van cserélve.

7.3. táblázat *A bytes és a bytearray adattípusok tagfüggvényei (3)*

Utasításforma	Leírás
`b.upper()`	A *b* `bytes/bytearray` típusú változó ASCII szerinti nagybetűs másolatát adja vissza.
`b.zfill(szélesség)`	*b* olyan másolatát adja vissza, amelynek az elejére annyi nullát (`0x30` kódú karaktert) írunk, hogy `szélesség` szélességűvé váljon.

A `pack_string()` tagfüggvény akár 65 535 UFT-8 kódolású karakter kezelésére képes. Ami a bájtok számát illeti, könnyedén választhatunk más egész típust – például a négybájtos előjeles egész számokkal („i" formátum) akár 2^{31}-1 (kétmilliárdnál is nagyobb) hosszúságú karakterláncokat is használhatunk.

A `struct` modulban találunk egy hasonló beépített formátumot, a „p"-t, amely egy bájtot karakterszámként tárol, és ezt 255 további karakter követheti. A becsomagolásnál a „p" formátumot használó kód lényegesen egyszerűbb megoldást jelent, mintha minden munkát magunknak kellene elvégeznünk. A „p" formátum azonban legfeljebb 255 UTF-8 kódolású karakter használatát teszi lehetővé, és a kicsomagolás során alig jelent valami előnyt. (Az összehasonlítás kedvéért a `convert-incidents.py` fájlban megtaláljuk a `pack_string()` és az `unpack_string()` függvény „p" formátumot használó változatát is.

Figyelmünket immáron a kód hátralévő részére, az `export_binary()` tagfüggvényre fordíthatjuk.

```
fh = None
try:
    if compress:
        fh = gzip.open(filename, "wb")
    else:
        fh = open(filename, "wb")
    fh.write(MAGIC)
    fh.write(FORMAT_VERSION)
```

```
    for incident in self.values():
        data = bytearray()
        data.extend(pack_string(incident.report_id))
        data.extend(pack_string(incident.airport))
        data.extend(pack_string(incident.aircraft_id))
        data.extend(pack_string(incident.aircraft_type))
        data.extend(pack_string(incident.narrative.strip()))
        data.extend(NumbersStruct.pack(
                            incident.date.toordinal(),
                            incident.pilot_percent_hours_on_type,
                            incident.pilot_total_hours,
                            incident.midair))
        fh.write(data)
    return True
```

Az `except` és a `finally` kódblokkokat elhagytuk, mert az `except` blokkban elkapott kivételektől eltekintve megegyeznek az előző szakaszban lévőkkel.

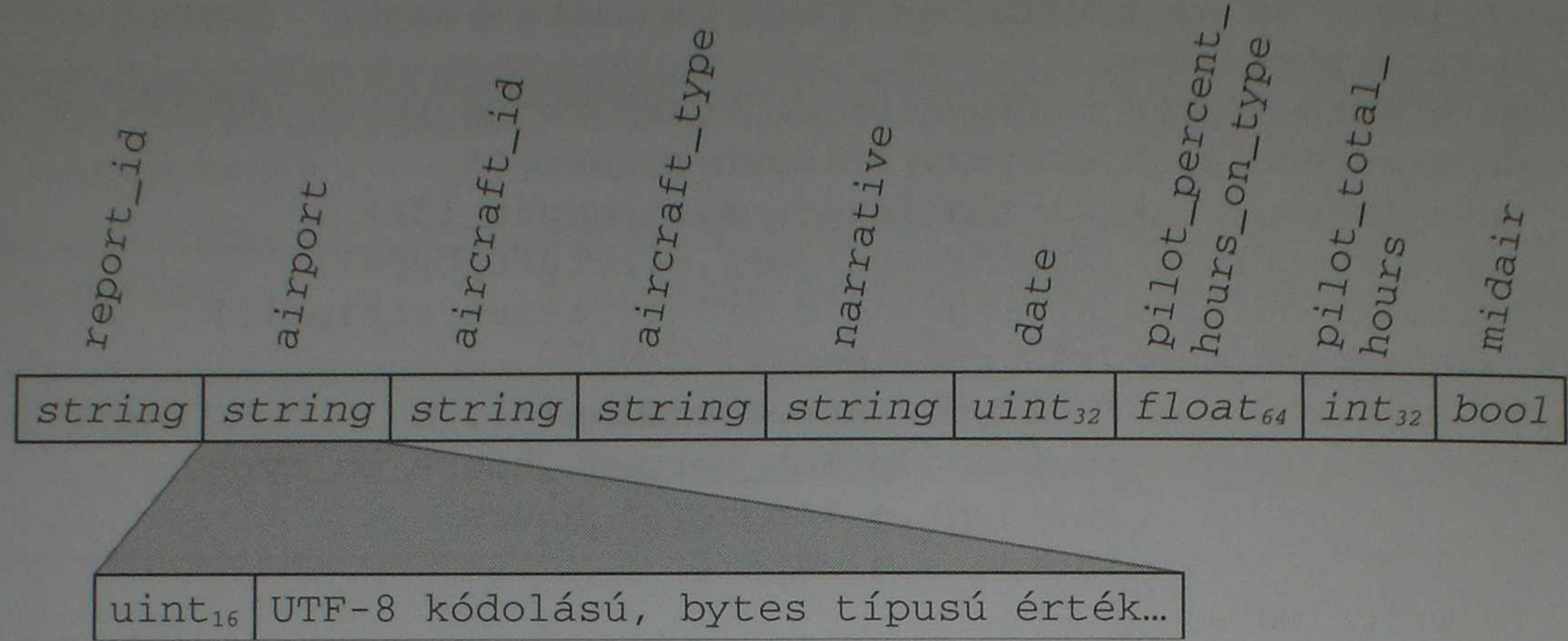
A fájlt a tagfüggvény elején `wb` („write binary", azaz bináris írás) módban nyitjuk meg, a `compress` változóban tárolt értéktől függően mint rendes vagy mint `gzip`-pel tömörített állományt. Ezek után leírjuk azt a négybájtos mágikus számot, amely – remélhetőleg – csak a mi programunkra jellemző, amit a kétbájtos változatszám követ.* A változatszám használata megkönnyíti a formátum megváltoztatását a jövőben – a változatszám beolvasásával meghatározható, hogy a beolvasáskor melyik kódot használjuk.

Az `Incident` objektumok bejárásával folytatjuk, és minden példány számára létrehozunk egy `bytearray` típusú változót. A változóban minden adatelemet változó hosszúságú karakterláncként helyezünk el. A `date.toordinal()` tagfüggvény egyetlen, a tárolt dátumot ábrázoló egész számot ad vissza, amelyet a `datetime.date.fromordinal()` tagfüggvény tud rendes dátummá visszaalakítani a számból. A `NumbersStruct` objektumot a program korábbi részében, az alábbi utasítással adtuk meg:

```
NumbersStruct = struct.Struct("<Idi?")
```

A formátumleíró little-endian bájtsorrendet ad meg, egy előjel nélküli harminckét bites egész számot (az egész számmá alakított dátum tárolására), egy hatvannégy bites lebegőpontos számot (a pilóta adott repülőgép-típuson repült idejére vonatkozó százalékos arány tárolására), egy harminckét bites egész számot (a teljes repült óraszám tárolására) és egy logikai értéket (ami azt mutatja, hogy a baleset a levegőben történt-e). A repülőgép-baleset tárolására szolgáló adatbázisrekord sematikus rajzát a 7.3. ábrán láthatjuk.

* A mágikus számoknak nincs olyan központi nyilvántartásuk, mint például az internetes tartományneveknek – így az egyediséget nem biztosítja semmi.



7.3. ábra

A repülőgép-baleset tárolására szolgáló bináris adatbázisrekord felépítése

Ha a `bytearray` típusú objektumba minden, a balesettel kapcsolatos adat bekerült, az objektumot lemezre írjuk. Amikor pedig az összes baleset a lemezen van, `True` értéket adunk vissza (már amennyiben nem volt hiba). A visszatérés előtt közvetlenül még a `finally` kódblokkal megbizonyosodunk arról, hogy a fájl le van-e zárva.

Az adatok visszaolvasása nem annyira egyszerű, mint a lemezre írásuk – legalábbis egy dolog miatt további hibaellenőrzést kell végeznünk. A változó hosszúságú karakterláncok visszaolvasása szintén rejt érdekességeket magában. Íme az `import_binary()` tagfüggvény eleje, benne a teljes beágyazott `unpack_string()` függvénnyel, amely a változó hosszúságú karakterláncok visszaolvasását végzi:

```
def import_binary(self, filename):

    def unpack_string(fh, eof_is_error=True):
        uint16 = struct.Struct("<H")
        length_data = fh.read(uint16.size)
        if not length_data:
            if eof_is_error:
                raise ValueError("missing or corrupt string size")
            return None
        length = uint16.unpack(length_data)[0]
        if length == 0:
            return ""
        data = fh.read(length)
        if not data or len(data) != length:
            raise ValueError("missing or corrupt string")
        format = "<{0}s".format(length)
        return struct.unpack(format, data)[0].decode("utf8")
```

Minthogy minden baleset rekordja a jelentésazonosítót tároló karakterlánccal kezdődik, amikor megpróbálkozunk a karakterlánc beolvasásával, és az sikerül, azonnal egy új rekord elejére kerülünk. Ha pedig problémába ütközünk, akkor elértük a fájl végét, és el is készül-

tünk. Amikor egy jelentésazonosító beolvasásával próbálkozunk, az `eof_is_error` (a fájl-vége-jel hibát jelent) értékét `False`-ra állítjuk, hiszen ha nem kapunk adatot, az csak annyit tesz, hogy befejezhetjük a beolvasást. Minden egyéb karakterlánc esetében elfogadjuk az alapértelmezett `True` beállítást, hiszen ha egy másik karakterláncban nincs adat, az baj. (Még az üres karakterláncok előtt is ott van a tizenhat bites előjel nélküli, a karakterlánc hosszát jelző szám.)

Elsőként a karakterlánc hosszát jelző számot próbáljuk meg beolvasni. Ha új balesetet szeretnénk beolvasni, és a hosszúság beolvasása sikertelen, a `None` értéket adjuk vissza, ezzel jelezve a fájl végét. Egyéb esetekben `VauleError` kivételt váltunk ki, ami most a hibás vagy hiányzó adat jele. A `struct.unpack()` tagfüggvény mindig egy rendezett sorozatot ad vissza, akkor is, ha csak egyetlen érték van benne. Kicsomagoljuk a hosszúságra vonatkozó adatot, és a benne ábrázolt számot a `length` változóban tároljuk. Mostanra tudjuk, hogy mennyi bájt beolvasása szükséges a karakterlánc kinyeréséhez. Ha a `length` változó értéke nulla, egyszerűen üres karakterláncot adunk vissza. Egyéb esetekben megkíséreljük a megadott számú bájt beolvasását. Ha nem kapunk adatot, vagy nem annyit, amennyire számítunk, `VauleError` kivételt váltunk ki.

Ha megvan a kívánt számú bájt, a `struct.unpack()` tagfüggvény számára létrehozunk egy megfelelő formátumleíró karakterláncot, majd visszaadjuk az adat kicsomagolásából, és a bájtok UTF-8 kódolású karakterekként történt értelmezéséből származó karakterláncot. (Elméletben az utolsó két sort lecserélhetnénk a `return data.decode("utf8")` utasításra, de jobbnak tartjuk, ha végig figyelemmel kísérjük a kicsomagolás folyamatát. Lehetséges ugyanis – bár nem túl valószínű –, hogy az „s" formátumleíró úgy alakítja át az adatainkat, hogy beolvasáskor visszaalakítást kell végeznünk.)

Most pedig megnézzük az `import_binary()` tagfüggvény fennmaradó részét – a magyarázat megkönnyítése végett mindezt két részletben tesszük.

```
fh = None
try:
    fh = open(filename, "rb")
    magic = fh.read(len(GZIP_MAGIC))
    if magic == GZIP_MAGIC:
        fh.close()
        fh = gzip.open(filename, "rb")
    else:
        fh.seek(0)
    magic = fh.read(len(MAGIC))
    if magic != MAGIC:
        raise ValueError("invalid .aib file format")
    version = fh.read(len(FORMAT_VERSION))
    if version > FORMAT_VERSION:
        raise ValueError("unrecognized .aib file version")
    self.clear()
```

A fájl lehet tömörített és tömörítetlen is, így ugyanazzal a módszerrel döntjük el, hogy a `gzip.open()` vagy a beépített `open()` függvénnyel nyitjuk meg a fájlt, amit a befőttek beolvasásánál is használtunk.

Ha a fájl megnyitásával végeztünk, és az elején állunk, beolvassuk az első négy bájtot (`len(MAGIC)`). Ha ezek nem egyeznek meg a mágikus számunkkal, tudjuk, hogy nem bináris repülőgépbaleset-tároló adatfájllal van dolgunk, és `ValueError` kivételt váltunk ki. A műveletet a kétbájtos változatszám beolvasásával folytatjuk. Ezen a ponton kellene választanunk beolvasó kódot – a változatszám függvényében. Most azonban csak annyit ellenőrzünk, hogy a változat nem magasabb-e annál, amit a program beolvasni képes.

Ha a mágikus szám helyes, és a fájlváltozat is olyan, amit kezelni tudunk, akkor készen állunk az adatok beolvasására. A munkát a szótár kiürítésével, a már benne lévő balesetek törlésével kezdjük.

```
while True:
    report_id = unpack_string(fh, False)
    if report_id is None:
        break
    data = {}
    data["report_id"] = report_id
    for name in ("airport", "aircraft_id",
                 "aircraft_type", "narrative"):
        data[name] = unpack_string(fh)
    other_data = fh.read(NumbersStruct.size)
    numbers = NumbersStruct.unpack(other_data)
    data["date"] = datetime.date.fromordinal(numbers[0])
    data["pilot_percent_hours_on_type"] = numbers[1]
    data["pilot_total_hours"] = numbers[2]
    data["midair"] = numbers[3]
    incident = Incident(**data)
    self[incident.report_id] = incident
return True
```

A `while` ciklus addig ismétlődik, amíg ki nem fogyunk az adatokból. Először egy jelentés-azonosítót szeretnénk látni. Ha `None` értéket kapunk, az annyit tesz, hogy elértük a fájl végét – így kiléphetünk a ciklusból. Egyéb esetben létrehozunk egy `data` nevű szótárt, amely egy baleset adatainak tárolására fog szolgálni. Ezt követően megkíséreljük a baleset többi adatának beolvasását. A karakterlánc esetében az `unpack.string()` tagfüggvényt használjuk, a többi adatot pedig egy nekifutásra, a `NumbersStruct` nevű formátumleíró használatával olvassuk be. Minthogy az adatok tárolását átalakítás előzte meg, az adataink visszanyeréséhez most el kell végeznünk a visszaalakítást. A többi elem esetében azonnal használhatjuk a kicsomagolatlan adatokat – nincs szükség ellenőrzésre vagy átalakításra, mert annak idején a megfelelő adattípusok használatával vittük lemezre az adatainkat, most pedig ugyanazokat az adattípusokat olvassuk vissza a `NumbersStruct` nevű formátumleíróban tárolt formátum használatával.

Ha valamilyen hibába ütközünk, például nem sikerül az összes szám kicsomagolása, kivétel lép fel, amelyet majd az `except` kódblokkban kezelünk. (Az `except` és a `finally` kódblokk nem szerepel a könyvben, mivel a szerkezetüket tekintve megegyeznek az előző, `import_pickle()` tagfüggvényt bemutató szakaszban lévővel.)

Megfeleltetés-kicsomagoló művelet 175 ☞

A kód vége felé a kényelmes, megfeleltetés-kicsomagoló művelettel hozunk létre olyan `Incident` objektumokat, amelyek aztán bekerülnek a baleseteket tároló szótárba.

A változó hosszúságú karakterláncok kezelésétől eltekintve a `struct` modul igencsak megkönnyíti az adatok bináris formátumban való tárolását és visszatöltését. Ami pedig a változó hosszúságú karakterláncok kezelését illeti, az itt bemutatott `pack_string()` és `unpack_string()` tagfüggvény a legtöbb célra tökéletesen megfelelő.

Szövegfájlok írása és feldolgozása

A szövegek írása egyszerű, de a visszaolvasásuk problémás lehet, vagyis a felépítést úgy kell megválasztanunk, hogy ne legyen túlságosan bonyolult a szöveg feldolgozása. A 7.4. ábrán lévő példa a repülőgép-baleseteket tartalmazó adatbázisunk egy rekordját mutatja be abban a szöveges formátumban, amelyet használni fogunk. A baleseti adatbázis rekordjainak fájlba írásakor minden rekordot egy üres sor követ, bár a fájl feldolgozásakor azt is elfogadjuk, ha az egymás után álló baleseteket egyetlen sor sem választja el, és azt is, ha több elválasztó sor is van.

```
[20070927022009C]
date=2007-09-27
aircraft_id=1675B
aircraft_type=DHC-2-MK1
airport=MERLE K (MUDHOLE) SMITH
pilot_percent_hours_on_type=46.1538461538
pilot_total_hours=13000
midair=0
.NARRATIVE_START.
    ACCORDING TO THE PILOT, THE DRAG LINK FAILED DUE TO AN OVERSIZED
    TAIL WHEEL TIRE LANDING ON HARD SURFACE.
.NARRATIVE_END.
```

7.4. ábra

A repülőgép-baleseti adatbázis egy rekordjának szövege

Szövegírás

A baleseti adatbázis minden rekordja egy szögletes zárójelek ([]) közé zárt jelentésazonosítóval kezdődik. Ezt követik az adatelemek, amelyek mindegyike *kulcs=érték* alakú. A többsoros kísérőszöveg kezdetét a `.NARRATIVE_START.`, a végét pedig a `.NARRATIVE_END.` szöveg jelöli. A kezdő- és a végjel közötti szöveget behúzással is elkülönítjük, nehogy összekeverjük a jelzésekkel a szövegnek akár egyetlen sorát is.

Alább következik az `export_text()` függvény kódja, amelyből az `except` és a `finally` blokkot elhagytuk, ugyanis azok megegyeznek a korábban látott példákban lévőkkel, kivéve a kezelt kivételeket.

```
def export_text(self, filename):
    wrapper = textwrap.TextWrapper(initial_indent=" ",
                                   subsequent_indent=" ")
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        for incident in self.values():
            narrative = "\n".join(wrapper.wrap(
                                   incident.narrative.strip()))
            fh.write("[{0.report_id}]\n"
                     "date={0.date!s}\n"
                     "aircraft_id={0.aircraft_id}\n"
                     "aircraft_type={0.aircraft_type}\n"
                     "airport={airport}\n"
                     "pilot_percent_hours_on_type="
                     "{0.pilot_percent_hours_on_type}\n"
                     "pilot_total_hours={0.pilot_total_hours}\n"
                     "midair={0.midair:d}\n"
                     ".NARRATIVE_START.\n{narrative}\n"
                     ".NARRATIVE_END.\n\n".format(incident,
        airport=incident.airport.strip(),
        narrative=narrative))
return True
```

A kísérőszövegben lévő sortörések a mi szempontunkból nem jelentősek, azaz a szöveget úgy tördeljük, ahogy kedvünk tartja. Rendes esetben a `textwrap` modul `textwrap.wrap()` függvényét használnánk, de ezúttal behúzásra és sortörésre is szükség van. Így először létrehozunk egy `textwrap.TextWrap` objektumot, mégpedig a használni kívánt behúzással: négy szóközzel minden sor előtt. Az objektum alapértelmezés szerint hetven karakter szélesség elérésénél töri a sorokat, ami azonban más kulcsszó-paraméter átadásával megváltoztatható.

`datetime` modul 212

Használhattunk volna háromszoros idézőjelben lévő szöveget, de jobb szeretjük saját kezűleg elhelyezni a sortöréseket. A `textwrap.TextWrap` objektumnak van egy `wrap()` nevű tagfüggvénye, amely egy karakterláncot –

str.format()
75

__format__()
253

esetünkben a kísérőszöveget – vár bemenetként, visszatérési értéke pedig egy olyan lista, amelynek az elemei a megfelelő behúzással ellátott, az adott szélességet túl nem lépő karakterláncok. Ezt követően a listában lévő sorokat egyetlen karakterlánccá egyesítjük, összekötőként újsor karaktert használva. A baleset dátumát `datetime.date` osztályú objektumként tároljuk; amikor az `str.format()` tagfüggvénnyel a dátumot leírtuk, kikényszerítettük a karakterláncként való ábrázolást – ezzel a dátumot az igen kényelmesen kezelhető, ISO 8601 szabvány szerinti ÉÉÉÉ-HH-NN formátumban kaptuk meg.

Az `str.format()` függvényt arra is megkértük, hogy a `midair` nevű logikai változót egész szám formájában adja meg – ez igaz értékek esetén 1-et, hamis értékek esetén pedig 0-át jelent. Általában véve az `str.format()` tagfüggvény használata igen kényelmessé teszi a szöveges adatok írását, ugyanis a tagfüggvény a Python nyelv összes adattípusát automatikusan kezeli – sőt az egyedi adattípusokat is, amennyiben megvalósítjuk az `__str__()` vagy a `__format__()` különleges szerepű tagfüggvények valamelyikét.

Szövegfeldolgozás

Az a tagfüggvény, amely a baleseti adatbázis rekordjainak beolvasását és feldolgozását végzi, hosszabb, és több ponton számíthat az érdeklődésünkre, mint az írásra szolgáló. A fájl olvasása alatt a programunk számos állapota különíthető el. Lehet, hogy a kísérőszöveg sorainak beolvasását végezzük, állhatunk egy *kulcs=érték* alakú sor előtt vagy éppen egy új baleset elejénél, a jelentésazonosítónál. Az `import_text_manual()` tagfüggvényt öt részletben tekintjük át.

```
def import_text_manual(self, filename):
    fh = None
    try:
        fh = open(filename, encoding="utf8")
        self.clear()
        data = {}
        narrative = None
```

A tagfüggvény azzal kezdődik, hogy a fájlt szövegolvasó módban megnyitjuk. Ez után kitakarítjuk a baleseteket tároló szótárat, majd létrehozzuk azt a `data` szótárat, amely egyetlen baleset adatait tudja tárolni – éppen úgy, ahogyan a bináris fájl beolvasásakor tettük. A `narrative` változónak két feladata van: állapotjelző, és az aktuális baleset kísérőszövegét is tartalmazza. Ha a `narrative` változó értéke `None`, az annyit jelent, hogy az adott pillanatban nem folyik semmilyen kísérőszöveg beolvasása. Ha az érték valamilyen karakterlánc – lehet üres karakterlánc is –, az viszont azt jelenti, hogy éppen a kísérőszöveg sorainak beolvasását végezzük.

```
for lino, line in enumerate(fh, start=1):
    line = line.rstrip()
    if not line and narrative is None:
        continue
    if narrative is not None:
        if line == ".NARRATIVE_END.":
```

```
            data["narrative"] = textwrap.dedent(narrative).strip()
            if len(data) != 9:
                raise IncidentError("missing data on "
                                    "line {0}".format(lino))
            incident = Incident(**data)
            self[incident.report_id] = incident
            data = {}
            narrative = None
        else:
            narrative += line + "\n"
```

Minthogy a beolvasást soronként végezzük, mindig ismerjük az aktuális sor sorszámát, amelynek a segítségével a bináris állományok beolvasásakor lehetségesnél több információt nyújtó hibaüzenetet adhatunk. Először a sor végén lévő üreshely karaktereket vágjuk le, és ha ekkor üres sorunk marad – és nem éppen egy kísérőszöveg belsejében járunk –, egyszerűen továbblépünk a következő sorra. Ekként lehetséges az, hogy az egyes balesetek közötti üres sorok száma nem érdekes, ugyanakkor megőrizzük a kísérőszövegekben lévő üres sorokat.

Ha a `narrative` változó értéke nem `None`, hanem valami más, akkor tudjuk, hogy éppen egy kísérőszöveg belsejében járunk. Ha a sor a kísérőszöveg végét jelző karakterlánc, akkor tudjuk, hogy azon felül, hogy a kísérőszöveg feldolgozását befejeztük, befejeztük az aktuális baleset feldolgozását is. Ebben az esetben a kísérőszöveget elhelyezzük a `data` szótárban – persze csak miután a behúzásokat a `textwrap.dedent()` függvénnyel eltávolítottuk. Ha megvan mind a kilencféle szükséges adat, új objektumot példányosítunk az `Incidents` osztályból, és az elkészült objektumot elhelyezzük a szótárban. Ennek végeztével kiürítjük a `data` szótárat, és a `narrative` változó alaphelyzetbe állításával felkészülünk az újabb baleset fogadására. Ha a beolvasott sor nem a kísérőszöveg végét jelző karakterlánc, akkor hozzáfűzzük a `narrative` változó végéhez – azzal az újsor karakterrel együtt, amit az előbb levágtunk.

```
        elif (not data and line[0] == "["
                      and line[-1] == "]"):
            data["report_id"] = line[1:-1]
```

Ha a `narrative` változó értéke `None`, akkor vagy új jelentésazonosítóhoz érkeztünk, vagy valamilyen egyéb adat beolvasását végezzük. Új jelentésazonosítónál csak akkor lehettünk, ha a `data` szótár üres (üres, amikor a baleset beolvasását elkezdjük, és az egyes balesetek beolvasásának végeztével kiürítjük), a sor pedig nyitó szögletes zárójellel kezdődik, és záró szögletes zárójellel ér véget. Ilyenkor a jelentésazonosítót elhelyezzük a `data` szótárban. Mindez annyit tesz, hogy az `elif` ág feltétele nem válik igazzá egészen addig, amíg a `data` szótárat ismét ki nem ürítjük.

```
        elif "=" in line:
            key, value = line.split("=", 1)
            if key == "date":
                data[key] = datetime.datetime.strptime(value, "%Y-%m-%d").date()
```

```
    elif key == "pilot_percent_hours_on_type":
        data[key] = float(value)
    elif key == "pilot_total_hours":
        data[key] = int(value)
    elif key == "midair":
        data[key] = bool(int(value))
    else:
        data[key] = value
elif line == ".NARRATIVE_START.":
    narrative = ""
else:
    raise KeyError("parsing error on line {0}".format(lino))
```

Ha éppen nem járunk kísérőszöveg belsejében, és nem is egy jelentésazonosító beolvasását végezzük, akkor már csak három további helyzet lehetséges: vagy *kulcs=érték* párokat olvasunk be, vagy a kísérőszöveg kezdetét jelző karakterláncnál vagyunk, vagy valami nem egészen megfelelően alakult.

Ha *kulcs=érték* párokat olvasunk be, a sort az első egyenlőségjelnél vágjuk. Legfeljebb egy vágást engedélyezünk, hogy nyugodtan írhassunk egyenlőségjelet az *érték* belsejébe. Minden adatot Unicode-karakterláncként olvasunk be, vagyis dátumok, számok és logikai értékek esetén gondoskodnunk kell az érték megfelelő átalakításáról.

A dátumok esetében a `datetime.datetime.strptime()` függvényt használjuk (a `strptime` a „string parse time", vagyis a „karakterlánccal értelmezett adat" rövidítése), amelynek paraméterül egy formátumleíró karakterláncot adunk át, a visszatérési értéke pedig egy `datetime.datetime` osztályú objektum. Olyan formátumleírót használtunk, ami az ISO 8601 dátumformátumnak tesz eleget; arra pedig, hogy a visszakapott `datetime.datetime` objektumból `datetime.date` objektumot tudjunk kinyerni, a `datetime.datetime.date()` tagfüggvényt használjuk. Ez azért szükséges, mert a dátum és az időpont közül nekünk csak a dátum kell. A számátalakítások elvégzésekor a Python nyelv beépített típusfüggvényeire, a `float()` és az `int()` függvényekre támaszkodunk. Érdemes megjegyeznünk, hogy az `int("4.0")` függvényhívás `ValueError` kivételt eredményez, és ha tágabb határokat szeretnénk engedni az egész számokat illetően, használhatjuk az `int(float("4.0"))` függvényhívást, ha pedig a csonkolás helyett inkább kerekíteni óhajtunk, a `round(float("4.0"))` hívás a megfelelő. A logikai értékek kinyerése hangyányi ügyeskedéssel jár, mert például a `bool("0")` hívás igaz értéket ad vissza (minden olyan karakterlánc, amely nem üres, igaz értéket ad), vagyis a karakterláncot előbb egész számmá kell alakítanunk.

A hibás, hiányos, vagy a megengedett tartományon kívül eső értékek minden esetben kivételhez vezetnek. Ha valamelyik átalakítás zárul sikertelenül, `ValueError` kivételt kapunk. Ha bármely érték kívül esik a megengedett tartományon, `IncidentError` kivétel lép fel abban a pillanatban, amikor az érték használatával megpróbáljuk példányosítani a megfelelő `Incident` objektumot.

Ha egy sor nem tartalmaz egyenlőségjelet, ellenőrizzük, hogy nem éppen a kísérőszöveg kezdetét jelölő karakterláncot olvastuk-e be. Ha igen, a `narrative` változónak üres karakterláncot adunk értékül. Így az elágazás első feltétele a következő sorok esetében egészen addig igaz marad, amíg be nem olvassuk a kísérőszöveg végét jelölő karakterláncot.

Ha sem az `if`, sem az `elif` ág feltétele nem vált igazzá, akkor valamilyen hibába ütköztünk, ezért az elágazás végén lévő `else` záradékkal `KeyError` kivételt hozunk létre.

```
    return True
except (EnvironmentError, ValueError, KeyError,
        IncidentError) as err:
    print("{0}: import error: {1}".format(
          os.path.basename(sys.argv[0]), err))
    return False
finally:
    if fh is not None:
        fh.close()
```

Miután minden sort beolvastunk, `True` értéket adunk vissza a hívónak – hacsak nem keletkezett kivétel, mely esetben az `except` kódblokk elkapja a kivételt, hibaüzenetet ír a képernyőre a felhasználónak, és `False` értéket ad vissza a hívónak. Akármi történt is, ha a fájt megnyitottuk, be is zárjuk mindezek végeztével.

Szövegfeldolgozás szabályos kifejezések használatával

Azoknak az olvasóknak, akik nem mozognak otthonosan a szabályos kifejezések (regular expression vagy „regex"), világában, azt tanácsoljuk, hogy mielőtt a következő részbe belefognának, vagy olvassák el a 12. fejezetet, vagy folytassák az olvasást a következő szakasszal, ide pedig, ha úgy tartja kedvük, térjenek vissza később.

A szövegfájlok feldolgozásának szabályos kifejezések használatával történő megoldása gyakran rövidebb kódot eredményez, mint ha mindent saját kezűleg végeznénk – ahogy mi is tettük az előzőekben –, ugyanakkor pontos hibajelzéseket adni lényegesen nehezebb feladat lehet. Az `import_text_regex()` tagfüggvény áttekintését két lépésben ejtjük meg: először a szabályos kifejezéseket vizsgáljuk meg, azt követően pedig magát a feldolgozást. Az `except` és a `finally` kódrészleteket elhagyjuk, mivel már semmi újat nem tanulhatunk belőlük.

```
def import_text_regex(self, filename):
    incident_re = re.compile(
                    r"\[(?P<id>[^]]+)\](?P<keyvalues>.+?)"
                    r"^\.NARRATIVE_START\.$(?P<narrative>.*?)"
                    r"^\.NARRATIVE_END\.$",
                    re.DOTALL|re.MULTILINE)
    key_value_re = re.compile(r"^\s*(?P<key>[^=]+)\s*=\s*"
                              r"(?P<value>.+)\s*$", re.MULTILINE)
```

Nyers karakterláncok
61 ☞

A szabályos kifejezéseket nyers karakterláncként írjuk le. Ezzel megtakaríthatjuk azt a fáradságot, ami a fordított perjelek megkettőzésével (minden \ helyett \\ írása) járna – például a nyers karakterláncok használata nélkül a második szabályos kifejezést a következő alakban kellene leírnunk: `"^\\s*(?P<key>[^=]+)\\s*=\\s*(?P<value>.+)\\s*$"`. A könyvben a szabályos kifejezések írásakor mindig nyers karakterláncokat használunk.

Az első szabályos kifejezés, az `incident_re` egy teljes baleset feljegyzésének felel meg a szövegfájlban. A szabályos kifejezés használatának egyik következménye, hogy figyelmen kívül hagyunk minden olyan szöveget, ami az adatbázisrekordok *között* áll. Ez a szabályos kifejezés valójában két részből áll. Az első rész a `\[(?P<id>[^]]+)\](?P<keyvalues>.+?)`, amely megkeres egy nyitó szögletes zárójelet, megkeresi és beolvassa az utána álló akárhány karaktert, egészen addig, amíg záró szögletes zárójelet nem talál. Így megkapjuk a jelentésazonosítót. Ez után megkeres néhány – de legalább egy –, a `keyvalues` illeszkedési csoportnak megfelelő karaktert (a `re.DOTALL` jelzés miatt az újsor karakterek is számítanak). A `keyvalues` illeszkedési csoportnak megfelelő karakterek jelentik annak a minimálisan szükséges feltételnek a teljesülését, amellyel továbbléphetünk a szabályos kifejezés második részéhez.

Az első szabályos kifejezés második része a következő: `^\.NARRATIVE_START\.$(?P<narrative>.*?)^\.NARRATIVE_END\.$`, ami betű szerint illeszkedik a `.NARRATIVE_START.` karakterláncra, majd a `narrative` illeszkedési csoportba bekerült, lehető legkevesebb féle karakterre, végül a baleset végénél lévő `.NARRATIVE_END.` karakterláncra. A `re.MULTILINE` jelölő miatt ebben a szabályos kifejezésben a kalap (^) nem csak a karakterlánc elejére, hanem minden sor elejére illeszkedik, és ehhez hasonlóan a „dollár" ($) sem csak a karakterlánc végére, hanem minden sor végére illeszkedik – így a kísérőszöveg elejét és végét jelölő karakterláncokra is a sorok elején érvényes az illeszkedés.

A második szabályos kifejezés, a `key_value_re` a *kulcs=érték* párokat tartalmazó sorokat keresi. Az átadott szövegben csak a sorok elejénél illeszkedik, mégpedig a sor elején lévő akárhány üreshely karakterre (az sem baj, ha nem talál ilyet), amelyet egyenlőségjelet nem tartalmazó karakterek sorozata követ – ezekből lesz a `key` illeszkedési csoport. Utánuk következik az egyenlőségjel, amelyet aztán további karakterek követnek – ezekből az előttük és az utánuk álló szóköz jellegű karakterek elhagyásával a `value` illeszkedési csoportot képezzük.

A fájl feldolgozásának alapvető elve megegyezik azzal, amit az előző részben tárgyalt kézi szövegfeldolgozóban használtunk, azzal a különbséggel, hogy ezúttal nem sorról sorra olvassuk be a baleseteket tároló adatbázisrekordokat, hanem szabályos kifejezésekkel nyerjük ki őket a fájlból.

```
        fh = None
        try:
            fh = open(filename, encoding="utf8")
            self.clear()
            for incident_match in incident_re.finditer(fh.read()):
                data = {}
                data["report_id"] = incident_match.group("id")
                data["narrative"] = textwrap.dedent(
                                incident_match.group("narrative")).strip()
                keyvalues = incident_match.group("keyvalues")
                for match in key_value_re.finditer(keyvalues):
                    data[match.group("key")] = match.group("value")
                data["date"] = datetime.datetime.strptime(
                                        data["date"], "%Y-%m-%d").date()
                data["pilot_percent_hours_on_type"] = (
                        float(data["pilot_percent_hours_on_type"]))
                data["pilot_total_hours"] = int(
                    data["pilot_total_hours"])
                data["midair"] = bool(int(data["midair"]))
                if len(data) != 9:
                    raise IncidentError("missing data")
                incident = Incident(**data)
                self[incident.report_id] = incident
            return True
```

Az `re.finditer()` tagfüggvény olyan bejárható objektumot ad vissza, ami viszont az egymást át nem fedő illeszkedéseket állítja elő. Ahogy eddig is tettük, létrehozzuk az egy baleset adatait tároló `data` szótárat, de ezúttal a jelentésazonosítót és a kísérőszöveget az `incident_re` szabályos kifejezés egyes illeszkedéseiből kapjuk. Ennek végeztével a `keyvalues` illeszkedési csoporttal egyetlen menetben kinyerjük az összes *kulcs=érték* párt, és a `key_values_re` szabályos kifejezés `re.finditer()` tagfüggvényének alkalmazásával bejárjuk az egyes *kulcs=érték* sorokat. Minden megtalált (kulcs, érték) párt elhelyezünk a `data` szótárban – vagyis minden érték karakterlánccként kerül a szótárba. Utóbb azokat az értékeket, amelyek nem igazi karakterláncok, lecseréljük az előző, megfelelő típusú adatra. A karakterláncok átalakítását ugyanazokkal a függvényekkel végezzük, mint amelyeket előzőleg, a kézi feldolgozás során is használtunk.

A kódba bekerült egy ellenőrzés is, ami azért felelős, hogy a `data` szótárba kilenc elem kerüljön – ha ugyanis az adatbázis egy balesetet tartalmazó rekordja megsérül, a `key_value.finditer()` függvény esetleg túl sok vagy túl kevés *kulcs=érték* párt tartalmazó sort talál. A végeredmény ugyanaz, mint az előzőekben – létrehozunk egy új `Incident` objektumot, amelyet behelyezünk a baleseteket tartalmazó szótárba, majd `True` értéket adunk vissza. Ha bárhol hiba történne, az `except` kódrész kiírja a megfelelő hibaüzenetet, és `False` értéket ad vissza, majd a `finally` kódblokk lezárja a fájlt.

A Python nyelv kivételkezelése az egyik dolog, amely mind a kézi, mind a szabályos kifejezések használatán alapuló szövegfeldolgozót olyannyira rövidd é és magától értetődővé teszi. A feldolgozóknak nem kell ellenőrizniük egyetlen olyan átalakítást sem, ahol karakterláncból dátumot, számot, vagy logikai értéket képzünk, és nem kell elvégezniük egyetlen tartományellenőrzést sem, mert ez az `Incident` objektumok dolga. Ha az előzőekben megjelölt műveletek közül bármelyik sikertelen, kivétel jön létre, a kivételeket pedig igen takarosan, a kód végén, egyetlen helyen kezeljük. Annak, hogy a közvetlen ellenőrzés helyett a kivételkezelésre támaszkodunk, van még egy előnyös oldala: a kód jó méretezhetősége. A hibakezelést végző kód akkor sem lesz hosszabb, ha az adatbázisrekord formátumának megváltozásával további adatelemek ellenőrzése is szükségessé válik.

XML-fájlok írása és feldolgozása

Néhány program minden adat kezeléséhez az XML fájlformátumot használja, más programok csak a kényelmes adatbetöltéshez és -mentéshez. Az XML formátumú adatok betöltésére és mentésére (kivitelére) való képesség igen hasznos dolog, és akkor is érdemes lehet vele foglalkozni, ha a program fő formátuma egyébként valamilyen szöveges vagy bináris fájlformátum.

A Python három módszert kínál az XML-fájlok írására: megírhatjuk őket saját kezűleg; létrehozhatunk egy elemfát, és használhatjuk az elemfa `write()` tagfüggvényét, illetve létrehozhatunk egy DOM-ot, és használhatjuk a DOM `write()` tagfüggvényét. Ami az XML-fájlok olvasását és feldolgozását illeti, négy megközelítési lehetőség közül választhatunk: beolvashatjuk és értelmezhetjük kézi módszerrel az XML-fájlokat (a módszer ellenjavallt, és nem tárgyaljuk – a homályosabb vagy bonyolultabb részletek kezelése meglehetős nehézségeket okozhat), használhatunk elemfát, illetve DOM- vagy SAX-feldolgozót.

A repülőgép-balesetek tárolására a 7.5. ábrán látható formátumot használjuk. Ebben a részben bemutatjuk, miként lehetséges a formátum kézi módszerrel, elemfával és DOM-mal való írása, illetve elemfával, DOM-mal és SAX-szal való beolvasása és feldolgozása. Ha az Olvasó nem különösebben törődik azzal, hogy melyik módszert használja, akkor elég, ha elolvassa az *Elemfák* című részt, és rögtön továbblapozhat a fejezet utolsó, *Véletlen elérésű bináris fájlok* című részére.

Elemfák

Az adatok elemfa használatával történő írása két lépésben történik. Először létre kell hozni az adatot ábrázoló elemfát, majd a fát egy fájlba kell írni. Lehetnek olyan programok, amelyek az adataikat eleve elemfában tárolják; ebben az esetben az elemfa már meg is van, csak az adatok kiírásával kell törődni. Az `export_xml_etree()` tagfüggvényt két részletben tekintjük át.

```
def export_xml_etree(self, filename):
    root = xml.etree.ElementTree.Element("incidents")
    for incident in self.values():
```

```
<?xml version="1.0" encoding="UTF-8"?>
<incidents>
<incident report_id="20070222008099G" date="2007-02-22"
    aircraft_id="80342" aircraft_type="CE-172-M"
    pilot_percent_hours_on_type="9.09090909091"
    pilot_total_hours="440" midair="0">
<airport>BOWERMAN</airport>
<narrative>
ON A GO-AROUND FROM A NIGHT CROSSWIND LANDING ATTEMPT THE AIRCRAFT HIT
A RUNWAY EDGE LIGHT DAMAGING ONE PROPELLER.
</narrative>
</incident>
<incident>
    ...
</incident>
    :
</incidents>
```

7.5. ábra

A repülőgép-balesetet tároló XML formátumú adatbázisrekord és környezete

```
    element = xml.etree.ElementTree.Element("incident",
                  report_id=incident.report_id,
                  date=incident.date.isoformat(),
                  aircraft_id=incident.aircraft_id,
                  aircraft_type=incident.aircraft_type,
                  pilot_percent_hours_on_type=str(
                                  incident.pilot_percent_hours_on_type),
                  pilot_total_hours=str(incident.pilot_total_hours),
                  midair=str(int(incident.midair)))
    airport = xml.etree.ElementTree.SubElement(element, "airport")
    airport.text = incident.airport.strip()
    narrative = xml.etree.ElementTree.SubElement(element, "narrative")
    narrative.text = incident.narrative.strip()
    root.append(element)
tree = xml.etree.ElementTree.ElementTree(root)
```

Először az `<incidents>` gyökérelemet hozzuk létre, majd bejárjuk a baleseteket tároló adatbázisrekordokat. Minden balesetnél létrehozzuk a baleset adatait tároló `<incident>` elemet, és kulcsszó-paraméterek használatával adjuk meg a jellemzőket. Minden jellemzőt szövegként adunk meg, ennek megfelelően a dátumokat, számadatokat és logikai értékeket át kell alakítanunk. Az „&", a „<" és a „>" elemek, illetve a jellemzőkben szereplő idézőjelek levédése miatt ne fájjon a fejünk, ugyanis az elemfa (illetve a DOM és a SAX) modul az ilyen feladatokat automatikusan ellátja.

Minden `<incident>` elem két alelemmel bír: az egyik a repülőtér nevét, a másik a kísérőszöveget tárolja. Alelemek létrehozásakor meg kell adnunk a szülőelemet és a címke nevét. Az egyes elemek olvasható-írható `text` jellemzője használatos a szöveg tárolására.

Ha az `<incident>` elemet minden jellemzőjével, `<airport>` és `<narrative>` alelemeivel együtt létrehoztuk, elhelyezhetjük a hierarchia gyökérelemén, az `<incidents>` elemen belül. A feldolgozás végeztével olyan elemhierarchiát kapunk, amely a baleseteket tároló adatbázis minden rekordját tartalmazza. A hierarchia ekkorra minden további nélkül elemfává alakítható.

```
try:
    tree.write(filename, "UTF-8")
except EnvironmentError as err:
    print("{0}: import error: {1}".format(
        os.path.basename(sys.argv[0]), err))
    return False
return True
```

A teljes elemfa XML formátumú fájlba való kiírása tehát nem jelent többet, mint megmondani az elemfa-objektumnak, hogy írja ki magát egy fájlba, egy adott karakterkódolással. Eddig, valahányszor karakterkódolást adtunk meg, szinte mindig az „`utf8`" alakot használtuk. Ez remekül működik a Python nyelv beépített `open()` függvénye esetében, amely sok karakterkódolást, az egyes karakterkódolások esetében pedig sok elfogadható nevet ismer: az „UTF-8", az „UTF8", az „utf-8" és az „utf8" mind megfelelő. Az XML-fájlok esetében azonban a karakterkódolást csak a hivatalosan használható nevek valamelyikével adhatjuk meg, ezért használtuk az „UTF-8" formát.*

Az XML-fájlok elemfa segítségével történő beolvasása nem sokkal nehezebb, mint a kiírásuk. Az eljárás ismét két lépésből áll: először beolvassuk és feldolgozzuk az XML-fájlt, majd a művelet eredményeként létrejövő elemfa adatainak felhasználásával feltöltjük adatokkal az `incidents` szótárat. Most is rámutatunk, hogy a második lépés nem szükséges, ha magát az elemfát használjuk adatok tárolására a memóriában is. Íme az `import_xml_etree()` tagfüggvény, két részre bontva.

```
def import_xml_etree(self, filename):
    try:
        tree = xml.etree.ElementTree.parse(filename)
    except (EnvironmentError,
            xml.parsers.expat.ExpatError) as err:
        print("{0}: import error: {1}".format(
            os.path.basename(sys.argv[0]), err))
        return False
```

Alapértelmezés szerint az elemfa-feldolgozó az expat XML-feldolgozóra támaszkodik a háttérben – így aztán fel kell készülnünk az expat-tól származó kivételek elkapására is.

```
self.clear()
for element in tree.findall("incident"):
    try:
```

* Az XML-fájlok karakterkódolásáról a `www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl` és a `www.iana.org/assignments/character-sets` webhelyeken találunk információt.

```
            data = {}
            for attribute in ("report_id", "date", "aircraft_id",
                              "aircraft_type",
                              "pilot_percent_hours_on_type",
                              "pilot_total_hours", "midair"):
                data[attribute] = element.get(attribute)
            data["date"] = datetime.datetime.strptime(
                                        data["date"], "%Y-%m-%d").date()
            data["pilot_percent_hours_on_type"] = (
                              float(data["pilot_percent_hours_on_type"]))
            data["pilot_total_hours"] = int(data["pilot_total_hours"])
            data["midair"] = bool(int(data["midair"]))
            data["airport"] = element.find("airport").text.strip()
            narrative = element.find("narrative").text
            data["narrative"] = (narrative.strip()
                                 if narrative is not None else "")
            incident = Incident(**data)
            self[incident.report_id] = incident
        except (ValueError, LookupError, IncidentError) as err:
            print("{0}: import error: {1}".format(
                os.path.basename(sys.argv[0]), err))
            return False
    return True
```

Ha az elemfánk elkészült, az `xml.etree.ElementTree.findall()` tagfüggvénnyel bejárhatjuk az `<incidents>` elemeket. Az egyes `<incidents>` elemeket `xml.etree.Element` osztályú objektumként kapjuk vissza. Az elemek jellemzőinek kezelésére ugyanazt az eljárást használjuk, mint korábban, az `import_text_regex()` tagfüggvénynél: először minden adatot a `data` szótárban tárolunk, majd a dátumokat, számadatokat és logikai értékeket a megfelelő típusú adattá alakítjuk. Az `airport` és a `narrative` alelemek esetében az `xml.etree.Element.find()` tagfüggvényt használjuk a megkeresésükre és a hozzájuk kapcsolódó `text` jellemző beolvasására. Ha egy szöveges elemben nincs szöveg, a `text` jellemzője a `None` értéket veszi fel, amire számítanunk kell a kísérőszöveg beolvasásakor – ez ugyanis lehet üres. A tagfüggvényektől visszakapott szöveg soha nem tartalmaz XML-védőkaraktereket, mert a kezelésüket a Python automatikusan elvégzi.

Mint minden, a repülőgép-balesetek adatait feldolgozó XML-feldolgozó esetében, kivétel jön létre, amennyiben az `airport` vagy a `narrative` elem, illetve bármely jellemző hiányzik, ha valamelyik átalakítás sikertelen, vagy ha bármely számadat kívül eshet a lehetséges értékek tartományán. Így biztosak lehetünk benne, hogy érvénytelen adat esetén leáll a feldolgozás, és hibaüzenet jelenik meg. A tagfüggvény végén lévő, az `incident` objektumok létrehozására és tárolására, valamint a kivételek kezelésére szolgáló kódrészlet megegyezik azzal, amit már korábban megismertünk.

DOM (Document Object Model)

A DOM az XML-fájlok memóriában történő ábrázolását és kezelését végző szabványos alkalmazásprogramozási felület. A DOM létrehozására és fájlba írására szolgáló, illetve az XML-fájl DOM használatával történő értelmezésére szolgáló kód szerkezetét tekintve nagyon hasonló az elemfával dolgozó megfelelőjéhez, és csak kevéssel hosszabb annál.

Elsőként az `export_xml_dom()` tagfüggvényt tekintjük át, két részletben. A tagfüggvény két lépésben végzi a munkáját: először egy a balesetek adatainak megfelelő DOM készül el, majd a kész DOM-ot kiírjuk egy fájlba. Az elemfához hasonlóan a DOM esetében is létezhetnek olyan programok, amelyek magát a DOM-ot használják belső adatszerkezetként – ilyen esetekben a tárolt adat egyszerűen lemezre írható.

```
def export_xml_dom(self, filename):
    dom = xml.dom.minidom.getDOMImplementation()
    tree = dom.createDocument(None, "incidents", None)
    root = tree.documentElement
    for incident in self.values():
        element = tree.createElement("incident")
        for attribute, value in (
                ("report_id", incident.report_id),
                ("date", incident.date.isoformat()),
                ("aircraft_id", incident.aircraft_id),
                ("aircraft_type", incident.aircraft_type),
                ("pilot_percent_hours_on_type",
                 str(incident.pilot_percent_hours_on_type)),
                ("pilot_total_hours",
                 str(incident.pilot_total_hours)),
                ("midair", str(int(incident.midair)))):
            element.setAttribute(attribute, value)
        for name, text in (("airport", incident.airport),
                           ("narrative", incident.narrative)):
            text_element = tree.createTextNode(text)
            name_element = tree.createElement(name)
            name_element.appendChild(text_element)
            element.appendChild(name_element)
        root.appendChild(element)
```

A tagfüggvény egy DOM-megvalósítás használatba vételével kezdődik. Alapértelmezés szerint a megvalósítást az expat XML-értelmező szolgáltatja. Az `xml.dom.minidom` modul egyszerűbb és kisebb DOM-megvalósítást kínál annál, amit az `xml.dom` modulban találunk, bár az általa használt objektumok az xml.dom modulból származnak. Ha megvan a DOM-megvalósítás, létrehozhatunk egy dokumentumot.

Az `xml.dom.DOMImplementation.createDocument()` tagfüggvény első paramétere a névtérhez tartozó URI, amelyre nincs szükségünk, így aztán `None` értéket adunk át. A második paraméter egy minősített név, a gyökérelemhez tartozó címke. A harmadik paraméter a dokumentumtípus; itt megint `None` értéket adunk meg, ugyanis nem állítunk be dokumentumtípust. Most, hogy hozzájutottunk a dokumentumot ábrázoló fához, kinyerjük a gyökérelemet, és továbblépünk: az `incident` objektumok bejárása következik.

Minden balesethez létrehozunk egy `<incident>` elemet, és minden olyan jellemző esetében, amivel fel akarjuk ruházni az elemet, meghívjuk a `setAttribute()` függvényt a jellemző nevével és értékével paraméterezve. Csakúgy, mint az elemfa esetében, itt sem kell aggódnunk a „&", a „<" és a „>" elemek, illetve a jellemzők értékében szereplő idézőjelek levédése miatt. Az `airport` és a `narrative` szövegelem esetében előbb létre kell hoznunk a szöveg tárolását végző szövegelemet, majd egy normál elemet – megfelelő címkenévvel –, a szövegelem szülőelemét. Amikor létrejöttek, a normál elemet, illetve a benne lévő szövegelemet elhelyezzük az aktuális `<incident>` elemben. Amikor pedig az `<incident>` elem is létrejött, elhelyezzük a gyökérelemben.

```
fh = None
try:
    fh = open(filename, "w", encoding="utf8")
    tree.writexml(fh, encoding="UTF-8")
    return True
```

Az `except` és a `finally` kódblokkot elhagytuk, ugyanis megegyeznek a korábban megismertekkel. A fenti kódrészlet a beépített `open()` függvénynek átadott karakterkódolás és az XML-fájlokban használt karakterkódolás közötti – az előzőekben már megbeszélt – különbségre hivatott rávilágítani.

Egy XML-dokumentumot egy DOM-ba betölteni hasonló ahhoz, mint amikor elemfába töltünk be, de a kivitelhez hasonlóan hosszabb kódot eredményez. Az `import_xml_dom()` függvényt három részletben tekintjük át. A munkát a `def` utasítást tartalmazó sorral és a beágyazott `get_text()` függvénnyel kezdjük.

```
def import_xml_dom(self, filename):
    def get_text(node_list):
        text = []
        for node in node_list:
            if node.nodeType == node.TEXT_NODE:
                text.append(node.data)
        return "".join(text).strip()
```

A `get_text()` függvény csomópontok listáját járja be – mármint a csomópontok gyermekcsomópontjait –, és minden szövegcsomópontnál kinyeri a csomópont szövegét, amelyet elhelyez egy szövegeket tartalmazó listában. A művelet végén a függvény az összegyűjtött szöveget egyetlen karakterláncként adja vissza, az üreshelyeket a karakterlánc elejéről és végéről egyaránt eltávolítva.

```
try:
    dom = xml.dom.minidom.parse(filename)
except (EnvironmentError,
        xml.parsers.expat.ExpatError) as err:
    print("{0}: import error: {1}".format(
        os.path.basename(sys.argv[0]), err))
    return False
```

Egy XML-fájl DOM-má alakítása könnyű, mivel a modul elvégzi helyettünk a munka nehéz részét, ugyanakkor készen kell állnunk az expat-hibák kezelésére, mert az elemfákhoz hasonlóan a háttérben a DOM-osztályok alapértelmezett feldolgozója is az expat XML-feldolgozó.

```
self.clear()
for element in dom.getElementsByTagName("incident"):
    try:
        data = {}
        for attribute in ("report_id", "date", "aircraft_id",
                          "aircraft_type",
                          "pilot_percent_hours_on_type",
                          "pilot_total_hours", "midair"):
            data[attribute] = element.getAttribute(attribute)
        data["date"] = datetime.datetime.strptime(
                                  data["date"], "%Y-%m-%d").date()
        data["pilot_percent_hours_on_type"] = (
                          float(data["pilot_percent_hours_on_type"]))
        data["pilot_total_hours"] = int(data["pilot_total_hours"])
        data["midair"] = bool(int(data["midair"]))
        airport = element.getElementsByTagName("airport")[0]
        data["airport"] = get_text(airport.childNodes)
        narrative = element.getElementsByTagName("narrative")[0]
        data["narrative"] = get_text(narrative.childNodes)
        incident = Incident(**data)
        self[incident.report_id] = incident
    except (ValueError, LookupError, IncidentError) as err:
        print("{0}: import error: {1}".format(
            os.path.basename(sys.argv[0]), err))
        return False
return True
```

Helyi függvények
☞ 357

Ha a DOM létrejött, töröljük az aktuális baleset adatait, és bejárjuk az `<incident>` elemeket. Minden elemből kinyerjük a jellemzőket, és a dátumot, számot vagy logikai értéket tárolók esetében átalakítjuk azokat a megfelelő adattípusra – pontosan ugyanúgy, ahogy az elemfa használatakor tettük. Az egyetlen igazán jelentős különbség a DOM és az elemfa használata között a szövegcsomópontok kezelésében rejlik.

Az `xml.dom.Element.getElementsByTagName()` tagfüggvényt használjuk az adott névvel rendelkező gyermekelemek felismerésére. Az `<airport>` és a `<narrative>` gyermekelemek esetében tudjuk, hogy mindkettőből pontosan egy létezik, így aztán mindkét típusnál csak az első (és egyetlen) gyermekelemet vesszük figyelembe. Ezt követően a beágyazott `get_text()` függvény használatával bejárjuk a fenti elemek gyermekcsomópontjait, kinyerve a bennük lévő szövegeket.

Ha hiba történik, a tagfüggvény elfogja a megfelelő kivételt, hibaüzenetet ír a felhasználó konzolablakába, és `False` értékkel tér vissza – ahogy azt már megszoktuk.

A DOM és az elemfa használata közötti különbségek nem jelentősek, és minthogy mindkét eljárás esetében az igazi munkát az expat feldolgozó végzi, mindkettő meglehetősen gyorsan dolgozik.

XML-fájl írása saját kezűleg

Ha az elemfánk, illetve a DOM-unk már készen áll, egyetlen tagfüggvény hívásával lemezre írható belőlük az XML-dokumentum. Ha az adatokat azonban még nem rendeztük valamelyik formátumba, akkor először létre kell hoznunk az elemfát, illetve a DOM-ot. Ilyen esetekben lehet, hogy kényelmesebb egyszerűen az adat közvetlen kiírását elvégezni.

Az XML-fájlok írásakor figyelnünk kell arra, hogy a szöveget és a jellemzők értékeit megfelelően levédjük, és arra, hogy jólformált XML-dokumentumot írjunk. Íme a balesetek XML-formátumban való kiírását végző `export_xml_manual()` tagfüggvény:

```
def export_xml_manual(self, filename):
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        fh.write('<?xml version="1.0" encoding="UTF-8"?>\n')
        fh.write("<incidents>\n")
        for incident in self.values():
            fh.write('<incident report_id={report_id} '
                     'date="{0.date!s}" '
                     'aircraft_id={aircraft_id} '
                     'aircraft_type={aircraft_type} '
                     'pilot_percent_hours_on_type='
                     '"{0.pilot_percent_hours_on_type}" '
                     'pilot_total_hours="{0.pilot_total_hours}" '
                     'midair="{0.midair:d}">\n'
                     '<airport>{airport}</airport>\n'
                     '<narrative>\n{narrative}\n</narrative>\n'
                     '</incident>\n'.format(incident,
                 report_id=xml.sax.saxutils.quoteattr(
                                                incident.report_id),
                 aircraft_id=xml.sax.saxutils.quoteattr(
                                                incident.aircraft_id),
                 aircraft_type=xml.sax.saxutils.quoteattr(
                                                incident.aircraft_type),
                 airport=xml.sax.saxutils.escape(incident.airport),
                 narrative="\n".join(textwrap.wrap(
                         xml.sax.saxutils.escape(
                             incident.narrative.strip()), 70))))
        fh.write("</incidents>\n")
        return True
```

Mint ahogy a fejezetben már gyakran megtettük, ezúttal is elhagytuk az `except` és a `finally` kódblokkot.

A fájl írását az UTF-8 karakterkódolás használatával végezzük, és ezt jeleznünk kell a beépített `open()` függvénynek. Ha pontosan akarunk fogalmazni, akkor azt kell mondanunk, hogy a karakterkódolást nem kell megadnunk az `<?xml?>` bevezetésben, hiszen az UTF-8 az alapértelmezett karakterkódolás, mindazonáltal szeretünk pontosak lenni. Úgy döntöttünk, hogy a jellemzők értékeit idézőjelek közé tesszük, így viszont a kényelem azt diktálta, hogy az egyes baleseteket tároló karakterláncok kerüljenek aposztrófok közé: ezzel elkerülhető az idézőjelek levédése.

A `sax.saxutils.quoteattr()` függvény abban az értelemben hasonló az általunk az XML formátumú szövegek esetében használt `sax.saxutils.escape()` függvényhez, hogy helyesen védi le az „&", a „<" és a „>" karaktereket. Ezen felül – szükség esetén – levédi az idézőjelet, az aposztrófot, és az olyan karakterláncokat, amelyeket idézőjelek vagy aposztrófok közé zárva adtunk át, használatra készen adja vissza. Ebben rejlik a magyarázata annak, hogy nem tettük idézőjelbe a jelentésazonosítót, illetve más jellemzők karakterlánc típusú értékeit.

A beszúrt újsorok, illetve a kísérőszöveg tördelése pusztán kozmetikai célokat szolgál. Arra valók, hogy a fájlt az ember könnyebben tudja olvasni és szerkeszteni, de akár el is hagyhatók.

A HTML formátumú adatírás nem sokban különbözik az XML formátumban történőtől. A `convert-incidents.py` program ennek egyszerű példájaként tartalmazza az `export_html()` függvényt, amelyet ez alkalommal nem vizsgálunk meg: aligha találunk benne bármi újat.

XML-fájlok feldolgozása a SAX segítségével

Az elemfától és a DOM-tól – amelyek az egész XML-dokumentumot ábrázolják a memóriában – eltérően a SAX (Simple API for XML, azaz egyszerű XML-alkalmazásprogramozási felület) a bemenet feldolgozását fokozatosan végzi, vagyis akár lehet gyorsabb és kevésbé memóriaigényes is. Teljesítményjavulásra mindazonáltal nincs okunk számítani, lévén az elemfa és a DOM egyaránt a gyors expat feldolgozót használja.

A SAX a bemenet feldolgozását „feldolgozási események" bejelentésével végzi, valahányszor kezdő vagy lezáró címkéhez, illetve egyéb XML-elemhez ér. Ha kezelni szeretnénk a számunkra érdekes eseményeket, létre kell hoznunk egy megfelelő, a kezelést végző osztályt, benne néhány előre megadott tagfüggvénnyel, amelyeknek a meghívására akkor kerül sor, amikor a hozzájuk tartozó feldolgozási esemény bekövetkezik. A leggyakrabban megvalósított kezelést végző osztály a tartalomkezelő, bár ha pontosabb ellenőrzésre van szükség, lehetséges hibakezelők és egyéb kezelőosztályok létrehozása is.

Íme a teljes `import_xml_sax()` tagfüggvény. Nagyon rövid, hiszen a munka legnagyobb részét az általunk létrehozott `IncidentSaxHandler` osztály végzi.

```
    def import_xml_sax(self, filename):
        fh = None
        try:
            handler = IncidentSaxHandler(self)
            parser = xml.sax.make_parser()
            parser.setContentHandler(handler)
            parser.parse(filename)
            return True
        except (EnvironmentError, ValueError, IncidentError,
                xml.sax.SAXParseException) as err:
            print("{0}: import error: {1}".format(
                os.path.basename(sys.argv[0]), err))
            return False
```

Létrehozzuk azt az egyetlen kezelőt, amelyet használni szeretnénk, majd létrehozzuk a SAX feldolgozót, tartalomkezelőjéül az imént létrehozott kezelőt állítva be. Ezek után átadjuk a fájl nevét a feldolgozó `parse()` tagfüggvénynek, és `True` értéket adunk vissza, ha a feldolgozásba nem csúszott hiba.

A saját készítésű, `IncidentSaxHandler` osztály előkészítőjének `self`-et, azaz a `dict` osztály `IncidentCollection` nevű alosztályát adjuk át. A kezelő kitakarítja a régi `Incident` objektumokat, majd a fájl feldolgozása közben kiépíti az `Incident` objektumokból álló szótárat. Mire a feldolgozás véget ér, a szótárban megtalálható az összes beolvasott baleset.

```
class IncidentSaxHandler(xml.sax.handler.ContentHandler):
    def __init__(self, incidents):
        super().__init__()
        self.__data = {}
        self.__text = ""
        self.__incidents = incidents
        self.__incidents.clear()
```

Az egyedi SAX-kezelőosztályok mindig a megfelelő ősosztálytól származnak. Így biztosak lehetünk benne, hogy minden olyan tagfüggvény esetében, amelyet nem valósítunk meg újra (hiszen nem érdekelnek bennünket az általuk kezelt események), az ősosztályban található változat kapja meg a hívást. Az ősosztályban található változat pedig semmit sem fog tenni – ami nekünk tökéletesen megfelel.

Az alaposztály előkészítő függvényének meghívásával kezdjük. Ez általában minden alosztály esetében követendő gyakorlat, bár az `object` osztályból közvetlenül származó osztályok esetében erre nincs szükség – ha viszont mégis megtesszük, abból sem lesz baj. A `self.__data` szótárat használjuk egyetlen baleset adatainak tárolására, a `self.__text` karakterlánc pedig a repülőtér nevét, illetve a kísérőszöveget tárolja, attól függően, hogy melyik beolvasásánál tartunk. A `self.__incidents` szótár pedig egy az `IncidentCollection` szótárra mutató objektumhivatkozás. Az `IncidentCollection` szótár frissítését a kezelő közvetlenül végzi. (Más megközelítés esetén megvalósíthatnánk

egy a kezelő belsejében lévő, független szótárat, amelyet a művelet végén a `dict.clear()` és utána a `dict.update()` függvény használatával átmásolunk az `IncidentCollection` szótárba.)

```
def startElement(self, name, attributes):
    if name == "incident":
        self.__data = {}
        for key, value in attributes.items():
            if key == "date":
                self.__data[key] = datetime.datetime.strptime(
                                   value, "%Y-%m-%d").date()
            elif key == "pilot_percent_hours_on_type":
                self.__data[key] = float(value)
            elif key == "pilot_total_hours":
                self.__data[key] = int(value)
            elif key == "midair":
                self.__data[key] = bool(int(value))
            else:
                self.__data[key] = value
    self.__text = ""
```

Minden alkalommal, amikor egy kezdőcímke, illetve a jellemzői beolvasására kerül sor, a címke nevével és a jellemzőkkel meghívjuk az `xml.sax.handler.ContentHandler.startElement()` tagfüggvényt. A repülőgép-baleseteket tároló XML-fájl esetében a kezdőcímkék a következők: az `<incidents>`, amelyet figyelmen kívül hagyunk, az `<incident>`, amelynek a jellemzőivel a `self.__data` szótár egy részét töltjük fel, illetve az `<airport>` és a `<narrative>`, amelyeket megint csak figyelmen kívül hagyunk. A `self.__text` karakterláncot minden kezdőcímke megtalálásakor ürítjük, ugyanis a repülőgép-balesetes XML-fájlban nem használunk beágyazott szöveges címkéket.

Az `IncidentSaxHandler` osztályon belül semmilyen kivételkezelést nem végzünk. Ha kivétel keletkezik, átadjuk a hívónak, esetünkben az `import_xml_sax()` tagfüggvénynek, ami majd elkapja, és kiírja a megfelelő hibaüzenetet.

```
def endElement(self, name):
    if name == "incident":
        if len(self.__data) != 9:
            raise IncidentError("missing data")
        incident = Incident(**self.__data)
        self.__incidents[incident.report_id] = incident
    elif name in frozenset({"airport", "narrative"}):
        self.__data[name] = self.__text.strip()
    self.__text = ""
```

Amikor egy lezáró címke beolvasása megtörténik, az `xml.sax.handler.ContentHandler.endElement()` tagfüggvényt hívjuk meg. Ha egy baleset végére értünk, elvileg megvan minden szükséges adatunk, így létrehozunk

egy új `<incident>` objektumot, amelyet elhelyezünk a baleseteket tároló szótárban. Ha egy szövegelem végére értünk, az elemet és az addig begyűjtött szöveget elhelyezzük a `self.__data` szótárban. Ha ez megtörtént, a `self.__text` karakterláncot kiürítjük, felkészítve a következő használatra. (Szigorú értelemben nem kell kiürítenünk, hiszen a kezdő címke beolvasásakor is kiürítjük. A kiürítés azonban néhány XML-formátum esetében jelentőségre tehet szert, például ott, ahol a címkék egymásba ágyazhatók.)

```
def characters(self, text):
    self.__text += text
```

Amikor a SAX feldolgozó szöveget olvas be, a `xml.sax.handler.ContentHandler.characters()` tagfüggvényt hívja meg. Nincs garancia arra vonatkozólag, hogy a teljes szöveg beolvasása alatt csak egyszer történik meg a tagfüggvény hívása: a szöveg ugyanis érkezhet darabokban is. Így aztán a tagfüggvényt egyszerűen csak arra használjuk, hogy gyűjtse össze a szöveget, és csak a megfelelő lezáró címkét elérve helyezzük el a szöveget a szótárban. (Hatékonyabb megoldást jelentene, ha a `self.__text` objektum egy lista lenne, a mostani tagfüggvény magja pedig a `self.__text.append(text)` tagfüggvény az objektumban; és persze a többi tagfüggvény átalakítását is el kellene végezni.)

A SAX API használata jelentősen különbözik az elemfa vagy a DOM használatától, de ugyanolyan hatékony módszert jelent. Létrehozhatunk további kezelőket is, illetve a tartalomkezelőben újra megvalósíthatunk egyes tagfüggvényeket, így érve el a folyamat általunk óhajtott ellenőrzési fokát. A SAX feldolgozó önmagában az XML-fájlból nyert adatok semmiféle ábrázolását nem tartalmazza – éppen ez teszi ideális eszközzé a fájl adatainak saját adatszerkezeteinkbe való beolvasására. Mindez azt is jelenti, hogy semmiféle „SAX-dokumentumunk" nincs, amit XML-formátumban kiírhatnánk, vagyis az XML-fájl megírásához a fejezet korábbi részében leírt valamelyik módszert kell igénybe vennünk.

Véletlen elérésű bináris fájlok

Az előző részekben abból indultunk ki, hogy a fájl beolvasása egy lépésben történik, ezt követően feldolgozzuk az adatokat, majd ismét egy lépésben írunk ki mindent. A mai számítógépekben annyi memória található, hogy az ilyen megközelítésnek még nagy adatmennyiség esetén is van létjogosultsága. Mindazonáltal vannak helyzetek, amikor jobb megoldás, ha az adatok a lemezen maradnak, csak a szükséges részeket olvassuk be, és csak a változtatásokat írjuk vissza. A lemezen tárolt adatok véletlen elérésének legegyszerűbb módja, ha valamilyen kulcs–érték alapú adatbázist, esetleg egy teljes értékű SQL-adatbázismotort használunk – a 11. fejezetben mindkettőt tárgyaljuk –, de a következő oldalakon azt ismertetjük, hogy miként lehetséges a véletlen elérésű fájlok kézi kezelése.

Elsőként a `BinaryRecordFile.BinaryRecordFile` osztályt mutatjuk be. Az osztályból példányosított objektum egy általános, olvasható és írható bináris fájlt ábrázol, rögzített hosszúságú rekordokként. Ha ezzel megvagyunk, a `BikeStock.BikeStock` osztályt

vesszük szemügyre, amely `BikeStock.Bike` objektumok gyűjteményét tárolja, mégpedig egy `BinaryRecordFile.BinaryRecordFile` objektum rekordjaiként. Tesszük mindezt azért, hogy megismerjük a véletlen elérésű bináris fájlok használatának módját.

Az általános BinaryRecordFile osztály

A `BinaryRecordFile.BinaryRecordFile` osztály alkalmazásprogramozási felülete egy olyan listához hasonlít, amelyben bármely adott sorszámú elemet olvashatjuk, írhatjuk és törölhetjük. Amikor egy rekordot törlünk, azzal spóroljuk meg a többi rekord előrébb mozgatását, hogy a rekordot egyszerűen csak töröltnek jelöljük. Ez azzal is együtt jár, hogy a törlés végeztével minden eredeti elemsorszám érvényes marad. A módszer további előnye, hogy a törlés a „törölt" jelzés eltávolításával egyszerűen visszavonható. Az előnyök ára, hogy a törléssel nem szabadul fel lemezterület. Ezen a problémán úgy segítünk, hogy módot adunk a fájl „tömörítésére", azaz a törölt rekordok eltávolítására – ami viszont az elemsorszámok érvénytelenné válásával jár együtt.

Mielőtt áttekintenénk a megvalósítást, tekintsük át a használatával kapcsolatos alapvető tudnivalókat:

```
Contact = struct.Struct("<15si")
contacts = BinaryRecordFile.BinaryRecordFile(filename, Contact.size)
```

A fenti példában egy olyan, little-endian bájtsorrendű, egy tizenöt bájtos karakterláncból és egy négybájtos előjeles egész számból álló adatszerkezetet hoztunk létre, amely az egyes rekordokat fogja ábrázolni. A példa második sorában létrehozunk egy `BinaryRecordFile.BinaryRecordFile` objektumpéldányt, fájlnevet és a használt adatszerkezetnek megfelelő rekordméretet adva meg. Ha a fájl létezik, a tartalmának megváltoztatása nélkül megnyitjuk, egyébként pedig létrehozzuk. Mindkét esetben bináris olvasás-írás módban nyitjuk meg.

```
contacts[4] = Contact.pack("Abe Baker".encode("utf8"), 762)
contacts[5] = Contact.pack("Cindy Dove".encode("utf8"), 987)
```

A fájllal úgy dolgozhatunk, mintha lista lenne: a listák elemelérési műveletét (`[]`) kell használnunk itt is. A fenti példában két bájtsorozatot (`bytes` objektumokat, mindkettőben egy a megjelölt karakterkódolással kódolt karakterlánccal és egy egész számmal) helyezünk el a fájl két, sorszámmal jelölt területén. A fenti két értékadás a már esetleg ott lévő adatokat felül fogja írni, ha pedig még nincs hat rekord a fájlban, akkor a hiányzó rekordok `0x00` értékkel létrejönnek.

```
contact_data = Contact.unpack(contacts[5])
contact_data[0].decode("utf8").rstrip(chr(0)) # az eredmény: 'Cindy
Dove'
```

Minthogy a „Cindy Dove" karakterlánc tizenöt UTF-8 karakternél rövidebb, a csomagolásakor a rekord végén fennmaradó helyet `0x00` értékű bájtokkal töltjük fel. Így aztán a rekord visszaolvasásakor a `contact_data` változóba egy kételemű rendezett sorozat

`(b'Cindy Dove\x00\x00\x00\x00\x00', 987)` kerül. A név visszanyeréséhez az UTF-8 kódolású karakterláncot Unicode kódolásúvá kell alakítanunk, és el kell távolítanunk a hátul lévő `0x00` bájtokat.

Most, hogy működés közben vetettünk már az osztályra egy pillantást, készen állunk a kód áttekintésére. A `BinaryRecordFile.BinaryRecordFile` osztály a `BinaryRecordFile.py` fájlban kapott helyett. A fájl a szokásos bevezetés után néhány privát, `bytes` típusú változó megadásával kezdődik.

```
_DELETED = b"\x01"
_OKAY = b"\x02"
```

Minden rekord egy állapotjelző bájttal kezdődik, ami vagy `_DELETED` (törölt), vagy `_OKAY` (oké), illetve az üres rekordok esetében `0x00`.

Íme a `class` utasítást tartalmazó sor és az előkészítést végző kódrész:

```
class BinaryRecordFile:
    def __init__(self, filename, record_size, auto_flush=True):
        self.__record_size = record_size + 1
        mode = "w+b" if not os.path.exists(filename) else "r+b"
        self.__fh = open(filename, mode)
        self.auto_flush = auto_flush
```

Két különböző rekordméret létezik. A `BinaryRecordFile.record_size` nevű rekordméret beállítását a felhasználó végzi, és a felhasználó szempontjából ekkora egy rekord. A privát, `BinaryRecordFile.__record_size` nevű változó a valós méretét adja meg a rekordnak, amibe beletartozik az állapotjelző bájt is.

Vigyázunk arra, hogy ha már létezik a fájl, akkor ki ne ürítsük – ezért ilyenkor „`r+b`" módban nyitjuk meg –, és arra is, hogy létrehozzuk, ha még nem létezik – ezért ilyenkor „`w+b`" módban nyitjuk meg. A mód megadásában lévő pluszjel jelenti az olvasásra *és* írásra való megnyitást. Ha a `BinaryRecordFile.auto_flush` változó értéke igaz, a fájl minden olvasás előtt és írás után a lemezre íródik a gyorsítótárból.

```
    @property
    def record_size(self):
        return self.__record_size - 1

    @property
    def name(self):
        return self.__fh.name

    def flush(self):
        self.__fh.flush()

    def close(self):
        self.__fh.close()
```

7.4. táblázat *Fájlobjektumok jellemzői és tagfüggvényei (1)*

Utasításforma	Leírás
`f.close()`	Az `f` fájlobjektumot lezárja, az `f.closed` jellemző értékét `True`-ra állítja.
`f.closed`	Ha `f` fájlobjektum lezárt, `True` értéket ad vissza.
`f.encoding`	A `bytes` és a `str` adattípusok közötti átalakítás során használt karakterkódolást tárolja.
`f.fileno()`	Az `f` fájlobjektumhoz tartozó leírót adja vissza. (Csak olyan fájloknál, amelyeknek van fájlleírójuk.)
`f.flush()`	A gyorsítótárból a háttértárra írja az `f` fájlobjektum tartalmát.
`f.isatty()`	`True` értéket ad vissza, ha a fájlobjektum egy konzolhoz van rendelve. (Csak igazi fájlokhoz tartozó fájlobjektumoknál használható.)
`f.mode`	Az `f` fájlobjektum megnyitásakor használt módot tárolja.
`f.name`	Az `f` fájlobjektum nevét tárolja (ha van).
`f.newlines`	Az `f` szövegfájl-objektumban tárolt újsor karakterek fajtáit tárolja.
`f.__next__()`	Az `f` fájlobjektum következő sorát adja vissza. A legtöbb esetben nem közvetlenül használjuk – például a `for sor in f` utasításban sem.
`f.peek(n)`	`n` bájtot ad vissza, a fájlmutató mozgatása nélkül.
`f.read(szám)`	Az `f` fájlobjektumból legfeljebb `szám` bájtot ad vissza. Ha a `szám` nincs megadva, akkor a fájlban lévő aktuális helyzettől a fájl végéig minden bájtot visszaad. Bináris módú olvasáskor `bytes`, szöveges módú olvasáskor `text` a visszatérési érték típusa. Ha nincs mit beolvasni (vége a fájlnak), üres `bytes`, illetve `str` típusú értéket kapunk vissza.
`f.readable()`	`True` értéket ad vissza, ha a fájlt olvasásra nyitották meg.
`f.readinto(ba)`	Legfeljebb `len(ba)` bájtot olvas be a `ba` `bytearray` típusú változóba, majd a beolvasott bájtok számát – a fájlok végéhez érve nullát – visszaadja. (Csak bináris módban használható.)
`f.readline(szám)`	Beolvassa a következő sort (ha a `szám` meg van adva, akkor legfeljebb az első `szám` bájtot, de csak akkor, ha van elég bájt az `\n` karakter előtt), az `\n` karakterrel együtt.
`f.readlines(méret)`	Beolvassa a fájl végéig hátralévő sorokat, és listaként adja vissza azokat. Ha a `méret` meg van adva, és a használt fájlrendszer támogatja, akkor körülbelül `méret` bájtot olvas be.
`f.seek(eltolás, innen)`	Ha az `innen` nincs megadva, vagy az értéke `os.SEEK_SET`, a fájlmutatót (vagyis ahol a következő írási vagy olvasási művelet kezdődik) az `eltolás`-nak megfelelően eltolja. Ha az `innen` értéke `os.SEEK_CUR`, akkor a jelenlegi helyzettől számítva, ha az `innen` értéke `os.SEEK_END`, akkor a fájl végétől számítva a fájlmutatót `eltolás` (az értéke lehet negatív is) hellyel elmozgatja. Az `a` („append", azaz hozzáfűzés) módban az írási műveletek a fájlmutató helyétől függetlenül mindig a fájl végén mennek végbe. Szöveges módban az `eltolás` értékéül csak a `tell()` függvény visszatérési értékeit érdemes használnunk.

7.5. táblázat *Fájlobjektumok jellemzői és tagfüggvényei (2)*

Utasításforma	Leírás
`f.seekable()`	`True` értéket ad vissza, ha az `f` fájlobjektum támogatja a véletlen elérést.
`f.tell()`	A fájlmutató értékét a fájl kezdetéhez viszonyítva adja meg.
`f.truncate(méret)`	A fájlt az aktuális pozíciónál (ha a `méret` meg van adva, akkor ott) csonkolja.
`f.writeable()`	`True` értéket ad vissza, ha a fájlt írásra nyitották meg.
`f.write(s)`	Ha a fájl bináris módban van megnyitva, akkor az `s` `bytes/bytearray` objektumot, ha pedig szöveges módban, akkor az `s` `string` objektumot a fájlba írja.
`f.writelines(sorozat)`	Objektumok sorozatát (szövegfájloknál karakterláncokét, bináris állományoknál `bytes/bytearray` objektumokét) írja a fájlba.

A rekord méretét és a fájlnevet csak olvasható tulajdonságként adtuk meg. A felhasználóval akkora rekordméretet közlünk, amekkorát kért, és amekkora megfelel a rekordoknak. A `flush` és a `close` eljárások egyszerűen továbbadják a feladatot a fájlobjektumnak.

```
def __setitem__(self, index, record):
    assert isinstance(record, (bytes, bytearray)), \
        "binary data required"
    assert len(record) == self.record_size, (
        "record must be exactly {0} bytes".format(
        self.record_size))
    self.__fh.seek(index * self.__record_size)
    self.__fh.write(_OKAY)
    self.__fh.write(record)
    if self.auto_flush:
        self.__fh.flush()
```

Ez a tagfüggvény a `brf[i] = adat` utasításformát használja, ahol a `brf` egy bináris rekordokból álló fájl, az `i` a rekord sorszáma, az `adat` pedig egy bájtsorozat. Figyeljük meg, hogy a rekordok méretének meg kell egyezniük azzal, amit a bináris fájl létrehozásakor megadtunk. Ha a paraméterek rendben vannak, a fájlmutatót a rekord első bájtjára állítjuk – vegyük észre, hogy ezúttal a valós rekordméretet használtuk, vagyis figyelembe vettük az állapotjelző bájtot. A `seek()` tagfüggvény alapértelmezés szerint a fájlmutatót egy abszolút sorszámú helyre állítja. A második paraméter megadásával tehetjük a helyzetet a jelenlegi helyzethez, illetve a fájl végéhez képest értelmezetté. (A fájlobjektumok jellemzői és tagfüggvényei a 7.4. és a 7.5. táblázatban olvashatók.)

Minthogy az az elem, amelyiknek a beállítását éppen most végezzük, nyilvánvalóan nincs törölve, leírjuk az `_OKAY` állapotjelző bájtot, ezt követően pedig a felhasználótól származó bináris rekordban lévő adatokat. A fájl szempontjából nem érdekes a használt rekordszerkezet, és nem is ismeri azt. Egyetlen dolog számít: a rekordok megfelelő mérete.

Nem ellenőrizzük, hogy a sorszám a megfelelő határok között van-e. A rekord akkor is a megadott helyre kerül, ha a sorszám a fájl végén túlra mutat, a fájl régi vége és a most leírt rekord közötti bájtok pedig automatikusan `b"\x00"` értéket kapnak. Az ilyen üres rekordok állapota se nem `_OKAY`, se nem `_DELETED`, így szükség esetén megkülönböztethetők.

```
def __getitem__(self, index):
    self.__seek_to_index(index)
    state = self.__fh.read(1)
    if state != _OKAY:
        return None
    return self.__fh.read(self.record_size)
```

A rekordok visszaolvasásakor négy különböző helyzet állhat elő, amire kell felkészülnünk. Az első, hogy a rekord nem létezik, vagyis a megadott sorszám a fájl végén túlra mutat. A második, hogy a rekord üres, a harmadik, hogy a rekord törölt, az utolsó pedig az, hogy a rekorddal minden rendben. Ha a rekord nem létezik, akkor a privát `__seek_to_index()` tagfüggvény `IndexError` kivételt vált ki. Egyébként bájtonként haladva a rekord elejére áll, és már olvashatjuk is az állapotjelző bájtot. Ha az állapot nem `_OKAY`, akkor vagy üres, vagy törölt a rekord; mindkét esetben `None` értéket adunk vissza, egyébként pedig magát a rekordot. (Az is járható út lenne, ha `None` érték visszaadása helyett valamilyen egyedi kivételt váltanánk ki az üres és a törölt rekordok esetében.)

```
def __seek_to_index(self, index):
    if self.auto_flush:
        self.__fh.flush()
    self.__fh.seek(0, os.SEEK_END)
    end = self.__fh.tell()
    offset = index * self.__record_size
    if offset >= end:
        raise IndexError("no record at index position {0}".format(
                                                            index))
    self.__fh.seek(offset)
```

A fenti kód egy segédtagfüggvény, amelyet más tagfüggvények használnak a fájlmutató adott sorszámú rekord elejére való állítására. Először megnézzük, hogy a megadott sorszám megfelelő határok közé esik-e. Ezt úgy tesszük meg, hogy a fájl végére megyünk (0 eltolás a fájl végétől), a `tell()` függvénnyel pedig megnézzük, hogy hová jutottunk. Ha a rekord eltolásértéke (a sorszám szorozva a valós rekordmérettel) nem kisebb a kapott értéknél, akkor a sorszám túl magas. Ilyenkor kiváltjuk a megfelelő kivételt. Ha a sorszám megfelelő, a megfelelő helyre állítjuk a fájlmutatót, és készen állunk a következő olvasási vagy írási műveletre.

```
def __delitem__(self, index):
    self.__seek_to_index(index)
    state = self.__fh.read(1)
    if state != _OKAY:
        return
```

```
        self.__fh.seek(index * self.__record_size)
        self.__fh.write(_DELETED)
        if self.auto_flush:
            self.__fh.flush()
```

Először a megfelelő helyre mozgatjuk a fájlmutatót. Ha a sorszáma megfelelő határok között van – vagyis ha nem kaptunk `IndexError` kivételt –, és a rekord nem üres, illetve nem törölt, akkor elvégezzük a rekord törlését. Az állapotjelző bájtot a `_DELETED` értékkel írjuk felül.

```
    def undelete(self, index):
        self.__seek_to_index(index)
        state = self.__fh.read(1)
        if state == _DELETED:
            self.__fh.seek(index * self.__record_size)
            self.__fh.write(_OKAY)
            if self.auto_flush:
                self.__fh.flush()
            return True
        return False
```

A fenti tagfüggvény először megkeresi a rekordot, és beolvassa a benne lévő állapotjelző bájtot. Ha a rekord törölt, az állapotjelző bájtot `_OKAY` értékkel írjuk felül, és a hívónak `True` értéket adunk vissza a siker jeleként. Egyéb esetekben (üres rekordoknál, illetve olyanoknál, amelyek nincsenek törölve) False értéket adunk vissza.

```
    def __len__(self):
        if self.auto_flush:
            self.__fh.flush()
        self.__fh.seek(0, os.SEEK_END)
        end = self.__fh.tell()
        return end // self.__record_size
```

Ez a tagfüggvény azt mondja meg, hogy hány rekord van egy adott bináris rekordokat tartalmazó fájlban. Úgy kapja meg a választ, hogy a fájlvég helyét – vagyis a fájlban lévő bájtok számát – elosztja a rekordmérettel.

Ezzel végeztünk a `BinaryRecordFile.BinaryRecordFile` osztály kínálta alapvető tagfüggvények tárgyalásával. Egyetlen dolgot kell még átgondolnunk: a fájl tömörítését, amivel az üres és a törölt rekordok eltávolítását végezzük. Lényegében két megközelítési lehetőségünk van. Az egyik az, hogy az üres és a törölt rekordokat felülírjuk a magasabb sorszámú helyen tárolt rekordokkal, és ha marad üres vagy törölt rekord a fájl végén, akkor csonkoljuk a fájlt. Az `inplace_compact()` tagfüggvény így dolgozik. A másik lehetőség, hogy azokat a rekordokat, amelyek se nem üresek, se nem töröltek, ideiglenes fájlba másoljuk, amit utóbb átnevezünk az eredeti fájl nevére. Az ideiglenes fájl használata főleg akkor kényelmes, ha meg szeretnénk tartani egy biztonsági másolatot. A `compact()` tagfüggvény ekképpen működik.

Először az `inplace_compact()` tagfüggvény működését tekintjük át, két részletben.

```
def inplace_compact(self):
    index = 0
    length = len(self)
    while index < length:
        self.__seek_to_index(index)
        state = self.__fh.read(1)
        if state != _OKAY:
            for next in range(index + 1, length):
                self.__seek_to_index(next)
                state = self.__fh.read(1)
                if state == _OKAY:
                    self[index] = self[next]
                    del self[next]
                    break
            else:
                break
        index += 1
```

Minden egyes rekordot bejárunk, és bejáráskor mindegyik rekord állapotát is beolvassuk. Ha üres vagy törölt rekordot találunk, megkeressük a fájlban a következő olyan rekordot, amely se nem üres, se nem törölt. Amikor ilyenhez érünk, az üres, illetve a törölt rekordot felülírjuk vele, majd ezt töröljük. Ha nem találunk ilyet, akkor kilépünk az egész `while` ciklusból, hiszen elfogytak az olyan rekordok, amelyek se nem üresek, se nem töröltek.

```
self.__seek_to_index(0)
state = self.__fh.read(1)
if state != _OKAY:
    self.__fh.truncate(0)
else:
    limit = None
    for index in range(len(self) - 1, 0, -1):
        self.__seek_to_index(index)
        state = self.__fh.read(1)
        if state != _OKAY:
            limit = index
        else:
            break
    if limit is not None:
        self.__fh.truncate(limit * self.__record_size)
self.__fh.flush()
```

Ha az első rekord üres vagy törölt, akkor mindegyik üres vagy törölt: az előző kódrészlet ugyanis minden olyan rekordot, ami nem volt üres vagy törölt, a fájl elejére, a többit pedig a fájl végére mozgatta. Ilyen esetben egyszerűen nulla bájt hosszúságúra csonkoljuk a fájlt.

Ha találunk legalább egy olyan rekordot, amely nem üres és nem törölt, az utolsó rekordtól visszafelé haladva bejárjuk a fájlt – tudjuk ugyanis, hogy az üres és a törölt rekordok a fájl végére kerültek. A `limit` változó az első üres vagy törölt rekordra mutat (illetve `None` marad, ha sem üres, sem törölt rekordot nem találunk), a fájlt pedig a megfelelő helyen csonkoljuk.

A helyben végzett tömörítés mellett a másik lehetőség, hogy más fájlba másoljuk az adatokat – ez akkor lehet hasznos módszer, ha meg akarunk tartani egy biztonsági másolatot – mint ahogy a most következő `compact()` tagfüggvény ismertetéséből is kiderül.

```
def compact(self, keep_backup=False):
    compactfile = self.__fh.name + ".$$$"
    backupfile = self.__fh.name + ".bak"
    self.__fh.flush()
    self.__fh.seek(0)
    fh = open(compactfile, "wb")
    while True:
        data = self.__fh.read(self.__record_size)
        if not data:
            break
        if data[:1] == _OKAY:
            fh.write(data)
    fh.close()
    self.__fh.close()

    os.rename(self.__fh.name, backupfile)
    os.rename(compactfile, self.__fh.name)
    if not keep_backup:
        os.remove(backupfile)
    self.__fh = open(self.__fh.name, "r+b")
```

A tagfüggvény két fájlt hoz létre, egy tömörítettet, illetve az eredeti fájl biztonsági másolatát. A tömörített fájl kezdetben ugyanazt a fájlnevet kapja, mint az eredeti, de a fájlnév végére `.$$$` kiterjesztés kerül. Ehhez hasonlóan a biztonsági másolat neve is megegyezik az eredetivel, de a végére a `.bak` kiterjesztés kerül. A létező fájlt rekordonként olvassuk be, és az olyan rekordokat, amelyek se nem üresek, se nem töröltek, átírjuk a tömörített fájlba. (Figyeljük meg, hogy minden esetben az igazi rekordot, vagyis a felhasználó által használható részt és az azt megelőző állapotjelző bájtot írjuk át.)

A `bytes` és a `bytearray` adattípus ☞ 297

Az `if data[:1] == _OKAY:` sor méltó a figyelmünkre. A `data` objektum és az `_OKAY` objektum egyaránt a `bytes` osztályba tartozik. A `data` objektum első bájtját az (egybájtos) `_OKAY` objektummal szeretnénk összehasonlítani. Ha egy `bytes` objektum szeletét vesszük, szintén `bytes` objektumot kapunk, ha azonban csak egyetlen bájtot veszünk belőle, mondjuk a `data[0]` utasítással, akkor `int` objektumot – a bájt értékét. Esetünkben a `data` objektumnak tehát egyetlen bájtnyi részsorozatát – az első, állapotjelző bájtot – hasonlítjuk

össze az egybájtos `_OKAY` objektummal. (Ennek a műveletnek az elvégzésére a másik módszer az lenne, ha a `data[0] == _OKAY[0]` összehasonlítást végeznénk el, két `int` osztályú értéket hasonlítva össze.)

Ha mindezzel megvagyunk, az eredeti fájlt átnevezzük biztonsági másolattá, a tömörített fájl pedig megkapja az eredeti fájl nevét. Utána eltávolítjuk a biztonsági másolatot, ha a `keep_backup` változó értéke `False` (ez az alapértelmezett). Végül megnyitjuk az olvasásra és írásra egyaránt készen álló tömörített fájlt – ami mostanra az eredeti nevét viseli.

A `BinaryRecordFile.BinaryRecordFile` osztály meglehetősen alacsonyszintű, de szolgálhat alapként olyan magasabb szintű osztályok számára, amelyeknek rögzített méretű rekordokból álló véletlen elérésű fájlokat kell kezelniük – ahogy ez a következő részből is kiderül.

Példa: A BikeStock modul osztályai

A `BikeStock` modul egy `BinaryRecordFile.BinaryRecordFile` objektumot használ egy egyszerű raktárkészlet-kezelő osztály megvalósításához. A raktárkészlet elemei kerékpárok; az egyes kerékpárokat `BikeStock.Bike` objektumpéldányok ábrázolják, a teljes raktárkészletet pedig egy `BikeStock.BikeStock` objektumpéldány.
A `BikeStock.BikeStock` osztály egy kerékpár-azonosító kulcsú szótárat képez le egy `BinaryRecordFile.BinaryRecordFile` osztályú fájlobjektumba. A szótár értékei a rekordok sorszámai. Íme a modul használatát bemutató rövid példa, amelyből ráérezhetünk az osztályok működésére:

```
bicycles = BikeStock.BikeStock(bike_file)
value = 0.0
for bike in bicycles:
    value += bike.value
bicycles.increase_stock("GEKKO", 2)
for bike in bicycles:
    if bike.identity.startswith("B4U"):
        if not bicycles.increase_stock(bike.identity, 1):
            print("stock movement failed for", bike.identity)
```

A fenti kódrészlet a kerékpár-raktárkészletet tartalmazó fájl megnyitása után bejárja a benne lévő rekordokat, és a kerékpárok értékeinek és számainak szorzatait összegezve deríti ki a fájlban lévő összes kerékpár értékét. Ha ezzel elkészült, a raktáron lévő „GEKKO" kerékpárok számát kettővel, és minden „B4U" kezdetű kerékpártípusét eggyel növeli. Az összes művelet a lemezen történik meg, vagyis minden olyan folyamat, amely a fájl beolvasásával nyer információt, mindig az aktuális adatokat kapja meg.

Bár a `BinaryRecordFile.BinaryRecordFile` fájlobjektum sorszámokkal dolgozik, a `BikeStock.BikeStock` objektum kerékpár-azonosítókat használ. A megfeleltetést maga a `BikeStock.BikeStock` objektum végzi, mégpedig egy olyan szótárral felszerelkezve, amely a kerékpár-azonosítókat a fájlbeli rekordok sorszámainak felelteti meg.

Vizsgálódásunkat a BikeStock.Bike osztály class utasítást tartalmazó soránál és az előkészítőnél kezdjük, majd áttekintünk néhány fontosabb tagfüggvényt a BikeStock.BikeStock osztályból, végül pedig a kódnak azt a részét nézzük meg, amelyik a BikeStock.Bike objektumokat köti össze azokkal a bináris rekordokkal, amelyek az objektumokat a BinaryRecordFile.BinaryRecordFile objektumban ábrázolják. (A kód egésze a BikeStock.py fájlban található.)

```
class Bike:

    def __init__(self, identity, name, quantity, price):
        assert len(identity) > 3, ("invalid bike identity '{0}'"
                                   .format(identity))
        self.__identity = identity
        self.name = name
        self.quantity = quantity
        self.price = price
```

A Bike objektumok minden jellemzője tulajdonságként érhető el: a kerékpár-azonosító (self.__identity) a csak olvasható Bike.identity tulajdonságként, az összes többi pedig olvasható-írható tulajdonságként – néhány érvényességellenőrzést követően. Ezen felül a Bike.values csak olvasható tulajdonság adja vissza a mennyiség és ár szorzatát. (A tulajdonságok megvalósítását nem közöltük, hiszen korábban már láttunk hasonló kódot.)

A BikeStock.BikeStock osztályban találunk a Bike objektumok kezelésére való tagfüggvényeket, mégpedig olyanokat, amelyek a Bike objektumok írható tulajdonságait használják.

```
class BikeStock:

    def __init__(self, filename):
        self.__file = BinaryRecordFile.BinaryRecordFile(filename,
                                                        _BIKE_STRUCT.size)
        self.__index_from_identity = {}
        for index in range(len(self.__file)):
            record = self.__file[index]
            if record is not None:
                bike = _bike_from_record(record)
                self.__index_from_identity[bike.identity] = index
```

A BikeStock.BikeStock osztály olyan egyedi gyűjteményes adattípust valósít meg, amely egyszerre kezel egy bináris rekordokból álló fájlt (self.__file) és egy szótárat (self.__index_from_identity), amelynek a kulcsai kerékpár-azonosítók, az értékei pedig rekordok sorszámai.

Ha a fájlt megnyitottuk (vagy létrehoztuk, ha korábban nem létezett), bejárjuk a tartalmát – már ha van benne valami. Kiolvassuk az egyes kerékpárokat, és bytes objektumból a privát _bike_from_record() függvény segítségével BikeStock.Bike objektummá alakítjuk őket. A kerékpárok azonosítói és sorszámai a self.__index_from_identity szótárba kerülnek.

```
    def append(self, bike):
        index = len(self.__file)
        self.__file[index] = _record_from_bike(bike)
        self.__index_from_identity[bike.identity] = index
```

Új kerékpár elhelyezése tulajdonképpen a megfelelő rekordsorszám megtalálását, illetve a kerékpár bináris ábrázolásának a sorszámhoz tartozó rekordba töltését jelenti. Arra is figyelünk, hogy a `self.__index_from_identity` szótárat frissítsük.

```
    def __delitem__(self, identity):
        del self.__file[self.__index_from_identity[identity]]
```

Egy kerékpárt tartalmazó rekord törlése egyszerű: az azonosító alapján kikeressük a hozzá tartozó rekord sorszámát, és a sorszámnak megfelelő rekordot töröljük.
A `BikeStock.BikeStock` osztályban nem használjuk fel
a `BinaryRecordFile.BinaryRecordFile` osztály törlés visszavonására való képességét.

```
    def __getitem__(self, identity):
        record = self.__file[self.__index_from_identity[identity]]
        return None if record is None else _bike_from_record(record)
```

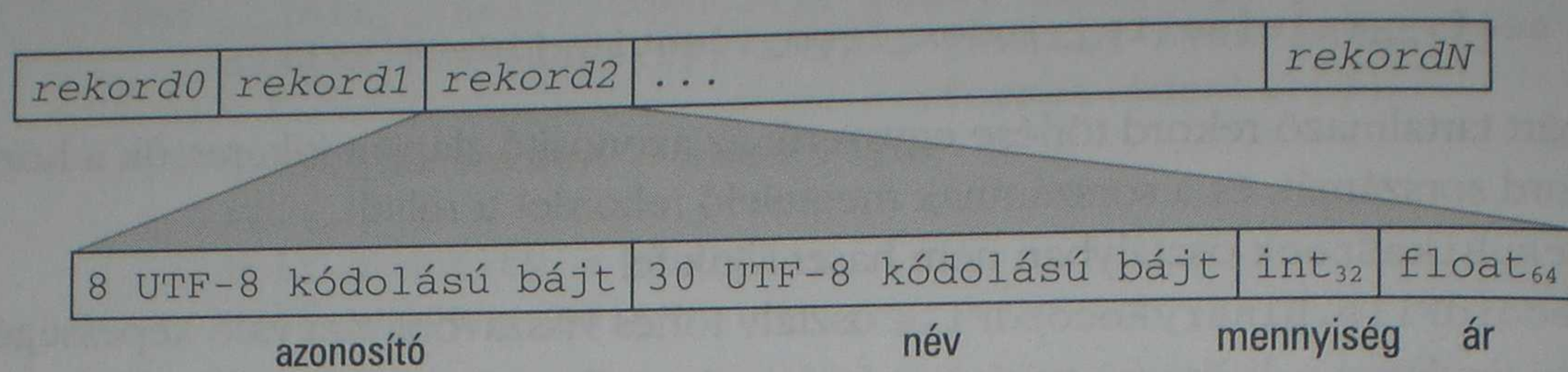
A kerékpárokhoz tartozó rekordok megtalálása kerékpár-azonosító alapján történik.
Ha a keresett azonosító nem létezik, a `self.__index_from_identity` szótáron lefutó keresés `KeyError` kivételt vált ki, ha pedig a rekord üres, illetve törölt,
a `BinaryRecordFile.BinaryRecordFile` objektum `None` értékkel tér vissza. Ha azonban megtaláljuk a rekordot, akkor azt adjuk vissza, `BikeStock.Bike` objektumként.

```
    def __change_stock(self, identity, amount):
        index = self.__index_from_identity[identity]
        record = self.__file[index]
        if record is None:
            return False
        bike = _bike_from_record(record)
        bike.quantity += amount
        self.__file[index] = _record_from_bike(bike)
        return True
    increase_stock = (lambda self, identity, amount:
                      self.__change_stock(identity, amount))
    decrease_stock = (lambda self, identity, amount:
                      self.__change_stock(identity, -amount))
```

A privát `__change_stock()` tagfüggvény az `increase_stock()` és a `decrease_stock()` tagfüggvény megvalósítását végzi. A kerékpár rekordjának sorszáma alapján nyers bináris rekordot olvas be, majd az adatot `BikeStock.Bike` objektummá alakítja, az objektumpéldányon elvégzi a szükséges módosítást, és a fájlban lévő rekordot felülírja a frissített `Bike` objektum bináris ábrázolásával. (Van egy `__change_bike()` tagfüggvény is, amely a `change_name()` és a `change_price()` tagfüggvényeket valósítja meg. A könyvben nem mutatjuk be őket, mivel az előzőleg tárgyalt tagfüggvényekhez igen hasonlók.)

```
    def __iter__(self):
        for index in range(len(self.__file)):
            record = self.__file[index]
            if record is not None:
                yield _bike_from_record(record)
```

A fenti tagfüggvénynek köszönhető, hogy a `BikeStock.BikeStock` objektumok a listákhoz hasonlóan bejárhatók. A bejárás egyes lépéseinél `BikeStock.Bike` objektumokat kapunk vissza, az üres és a törölt rekordokat átlépjük.



7.6. ábra

A kerékpárok tárolására szolgáló rekordokat tartalmazó fájl logikai szerkezete

A privát `_bike_from_record()` és `_record_from_bike()` függvény elrejti a `BikeStock.Bike` osztály bináris ábrázolását a kerékpárok gyűjteményét tároló `BikeStock.BikeStock` osztály elől. A kerékpárok tárolására szolgáló rekordokat tartalmazó fájl logikai szerkezete a 7.6. ábrán látható. A fizikai szerkezet kicsit eltér ettől: minden rekordot egy állapotjelző bájt előz meg benne.

```
_BIKE_STRUCT = struct.Struct("<8s30sid")

def _bike_from_record(record):
    ID, NAME, QUANTITY, PRICE = range(4)
    parts = list(_BIKE_STRUCT.unpack(record))
    parts[ID] = parts[ID].decode("utf8").rstrip("\x00")
    parts[NAME] = parts[NAME].decode("utf8").rstrip("\x00")
    return Bike(*parts)

def _record_from_bike(bike):
    return _BIKE_STRUCT.pack(bike.identity.encode("utf8"),
                             bike.name.encode("utf8"),
                             bike.quantity, bike.price)
```

Amikor egy bináris rekordot `BikeStock.Bike` objektummá alakítunk, először az `unpack()` függvénytől visszakapott rendezett sorozatot alakítjuk át egy listává. Így lehetségessé válik az elemek módosítása, esetünkben az UTF-8 kódolású bájtok olyan karakterlánccá való alakítása, ahonnan a `0x00` értékű bájtok már hiányoznak. Ezt követően a sorozatkicsomagoló művelet (`*`) használatával átadjuk a részeket a `BikeStock.Bike` objektum előkészítőjének. Az adatok becsomagolása lényegesen egyszerűbb, csak arra kell figyelnünk, hogy a karakterláncokat UTF-8 karakterkódolású bájtokká alakítsuk át.

A modern asztali számítógépekben a memória mennyiségének és a lemezek sebességének növekedésével a véletlen elérésű bináris állományok használatának szükségessége csökken. Ha mégis ilyesmire van szükségünk, gyakran az a legegyszerűbb, ha valamilyen adatbázisfájlt vagy egy SQL-adatbázist használunk. Ugyanakkor léteznek olyan rendszerek – például beágyazott vagy erőforrásszegény rendszerek –, amelyek esetében a fenti képességekkel bíró programok hasznosak lehetnek.

Összefoglalás

A fejezet a gyűjteményes adattípusok fájlba mentésére és onnan történő visszatöltésére szolgáló legelterjedtebben használt módszereket mutatta be. Láttuk, hogy milyen egyszerű a befőttek használata, és miként kezelhető mind a tömörített, mind a tömörítetlen fájlformátum anélkül, hogy előre tudnánk, hogy tömörített fájllal állunk-e szemben.

Láttuk, hogy a bináris adatok írása és olvasása nagy figyelmet igényel, és azt is láttuk, hogy amennyiben változó hosszúságú karakterláncokat kell kezelnünk, a kód igen hosszúra nyúlhat. Azt is megtanultuk azonban, hogy a bináris fájlok használata rendszerint a lehető legkisebb fájlméretet eredményezi, a leggyorsabb írás és olvasás mellett. Tudjuk azt is, hogy a fájltípus azonosítása végett fontos valamilyen mágikus szám használata, a formátum későbbi megváltoztatását pedig megkönnyíti, ha változatszámot is írunk a fájlba.

A fejezetben láttuk, hogy a felhasználók legkönnyebben a sima szöveges formátumú fájlokat olvassák, és ha ügyesen alakítottuk ki az adatfájl szerkezetét, akkor egyszerűen tudunk egyéb eszközöket is létrehozni az adatok kezelésére. Mindazonáltal a szöveges adatok értelmezése bonyolult lehet. Egyaránt láttunk a szöveges adatok kézi és szabályos kifejezéseket használó beolvasására is példát.

Az XML nagyon népszerű adatcsere-formátum, és általában legalább az adatok mentésére és betöltésére érdemes használni – még akkor is, ha a rendes formátum bináris vagy szöveges. Láttuk, hogy miként írható kézzel egy XML-fájl – láttuk a jellemzők értékeinek és a szöveges adatoknak a levédését is –, és azt is, hogy miként tehető meg mindez elemfa vagy DOM használatával. Megismertük az XML formátumnak a Python alapkönyvtárában lévő elemfával, a DOM-mal és a SAX-szal történő feldolgozását.

A fejezet utolsó részében megfigyelhettük egy általános célú, rögzített méretű rekordokat tartalmazó véletlen elérésű bináris fájlok kezelését végző osztály létrehozását. Megfigyelhettük az általános célú osztály egy adott helyzetben történő használatát is.

A fejezettel a Python-programozás alapjainak végére értünk. Ha most abbahagyjuk az olvasást, az eddig tanultak felhasználásával tökéletesen jó Python-programokat írhatunk. De nagy kár lenne éppen most abbahagyni az olvasást, hiszen olyan sok még a Pythonban rejlő lehetőség, a kód egyszerűsítésére és rövidítésére szolgáló apróságoktól

azokig az igazi agytornát jelentő lehetőségekig, amelyekről még akkor is érdemes legalább egy keveset tudnunk, ha csak ritkán fogjuk használni őket. A következő fejezetben ismét az eljárásközpontú és az objektumközpontú programozásban mélyedünk el, és megízleljük a függvényközpontú programozást is. A rákövetkező fejezetekben pedig olyan általánosabb érvényű témákkal foglalkozunk, mint a többszálúság, a hálózatkezelés, az adatbázisprogramozás, a szabályos kifejezések és a grafikus felhasználói felület programozása.

Gyakorlatok

Az első gyakorlat olyan egyszerű, bináris rekordokból álló fájlt kezelő modul megalkotása, amelyet a fejezet korábbi részében is bemutattunk – olyané, amelynek a rekordméretét a felhasználó adhatja meg. A második gyakorlat a `BikeStock` modul módosításával kapcsolatos – az új modul használatba vételét kell megoldanunk. A harmadik gyakorlatban teljesen új programot kell írnunk – a fájlkezelés meglehetősen magától értetődő, de a kimenet formázása némely tekintetben jelentős kihívásokat rejt magában.

1. Alkossuk meg a `BinaryRecordFile` modul új, egyszerűbb változatát – olyat, amelyik nem használ állapotjelző bájtot. Ebben a változatban a felhasználó a valós rekordméretet adja meg. Az új rekordok elhelyezése az új, `append()` nevű tagfüggvénnyel történik; a tagfüggvény a fájl végére mozgatja a fájlmutatót, és leírja az új rekordot. A `__setitem__()` tagfüggvény csak a már létező rekordok cseréjét engedheti meg; a feladat megoldásában nagy segítséget jelenthet a `__seek_to_index()` tagfüggvény használata. Most, hogy nincs állapotjelző bájt, a `__getitem__()` tagfüggvény mindössze három sorocskára zsugorodik. A `__delitem__()` tagfüggvényt teljesen újra kell írnunk, hiszen a keletkező rés feltöltésére minden rekordot előrébb kell hoznia. Mindez nagyjából fél tucat sorral megoldható, de kicsit el kell gondolkodni a megoldáson. Az `undelete()` tagfüggvényt el kell távolítani, lévén a művelet nem támogatott. A `compact()` és az `inplace_compact()` tagfüggvényt is el kell távolítani, de ezeket azért, mert a létük okafogyottá vált.
   Ahogy említettük, a változások az eredeti változathoz képest alig húsz új, és legalább hatvan törölt sort jelentenek – nem számítva a dokumentációs teszteket. A megoldást a `BinaryRecordFile_ans.py` fájlban találjuk.
2. Ha meggyőződtünk róla, hogy a `BinaryRecordFile` osztály egyszerűbb változata működik, készítsünk másolatot a `BikeStock.py` fájlról, és módosítsuk úgy, hogy az osztály új változatát használja. Ez alig pár sor megváltoztatását jelenti.
   A megoldást a `BikeStock_ans.py` fájlban találjuk.
3. A bináris formátumú fájlokban a hibakeresés néha bonyolult. Az egyik eszköz, ami a segítségünkre lehet, a bináris fájl tartalmát írja ki tizenhatos számrendszerben. Alkossunk olyan programot, amely a következő súgószöveget írja a képernyőre:

```
Usage: xdump.py [options] file1 [file2 [... fileN]]

Options:
  -h, --help            show this help message and exit
  -b BLOCKSIZE, --blocksize=BLOCKSIZE
                        block size (8..80) [default: 16]
-d, --decimal           decimal block numbers [default: hexadecimal]
-e ENCODING, --encoding=ENCODING
                        encoding (ASCII..UTF-32) [default: UTF-8]
```

Ha van egy olyan `BinaryRecordFile` formátumú fájlunk, amely a `"<i10s"` szerkezet (little-endian bájtsorrend, négybájtos előjeles egész, tízbájtnyi bájtsorozat) használatával tárolja a rekordokat, a blokkméretet a rekord méretéhez (állapotjelző bájttal együtt tizenöt bájt) igazítva tiszta képet kapunk a fájl tartalmáról. Például:

```
xdump.py -b15 test.dat
Block     Bytes                                   UTF-8 characters
--------  --------------------------------------  ----------------
00000000  02000000 00416C70 68610000 000000  .....Alpha.....
00000001  01140000 00427261 766F0000 000000  .....Bravo.....
00000002  02280000 00436861 726C6965 000000  .(...Charlie...
00000003  023C0000 0044656C 74610000 000000  .<...Delta.....
00000004  02500000 00456368 6F000000 000000  .P...Echo......
```

Minden bájt kétszámjegyű, tizenhatos számrendszerű számként jelenik meg, a négybájtos (vagyis nyolc tizenhatos számrendszerű számjegynyi) részek közötti szóközök csak az olvashatóságot könnyítik meg. Látható, hogy a második rekord – a „Bravo" – törölt, hiszen az állapotjelző bájtja `0x01`, a nem üres és nem törölt rekordokat jelző `0x02` helyett.

A parancssori paraméterek feldolgozására használjuk az `optparse` modult. (A paraméter „típusának" megadásával az `optparse` modul rávehető a karakterláncok egész számmá alakítására.) A fejlécek blokkméretének megfelelő kiírása meglehetős nehézséget jelenthet, csakúgy, mint az utolsó oszlopban lévő karakterek megfelelő elhelyezése. Mindenképpen hajtsunk végre teszteket különböző blokkméretekkel (például 8, 9, 10, …, 40). Azt se feledjük, hogy a változó hosszúságú fájlok esetében az utolsó blokk esetleg rövidebb a többinél. Ahogy a példa mutatja, használjunk pontokat a nem nyomtatható karakterek helyének kitöltésére.

A program két függvénnyel, hetven sornál rövidebben is megírható. A megoldást az `xdump.py` fájlban találjuk.

8

Haladó programozási eljárások

- Magasabb szintű eljárásközpontú programozás
- Magasabb szintű objektumközpontú programozás
- Függvényközpontú programozás

Ebben a fejezetben a különböző programozási módszerek széles skáláját tekintjük át, illetve a Python nyelv sok újabb – gyakran elmélyültebb tudásszintet képviselő – utasításformájával ismerkedünk meg. A fejezet anyagának bizonyos részei meglehetős kihívást jelentenek, mindazonáltal ne feledjük, hogy a legmagasabb szintű módszerekre ritkán van szükség.
Így ezúttal elegendő lehet, ha éppen csak futólag ismerkedünk meg a témával, bepillantást nyerünk a lehetőségekbe, és az alapos olvasás várhat, amíg szükségünk nem lesz valamelyik, a következőkben tárgyalt módszerre.

A fejezet első részében mélyebbre ássuk magunkat a Python eljárásközpontú (procedurális) lehetőségeibe. Először bemutatjuk, hogy újszerű megközelítésben miként használható a már elsajátított tudásunk, ezt követően pedig visszatérünk a generátoroknak a 6. fejezetben már érintett témájához. A rész a dinamikus programozás bemutatásával folytatódik – vagyis futásidőben fogjuk betöltetni a moduljainkat, miáltal lehetővé válik, hogy futásidőben döntsünk a futtatott kódról. Ezt követően visszakanyarodunk a helyi (beágyazott) függvényekhez, de tárgyaljuk a `nonlocal` kulcsszó és az önhívó függvények használatát is.
Az előzőekben már láttuk, hogy a Python előzetesen megadott díszítői miként használhatók. Ebben a részben azt is megtanuljuk, hogy miként hozhatunk létre saját díszítőket. A részt függvénymagyarázattal zárjuk.

A második rész az objektumközpontú (objektumorientált) programozással kapcsolatos, teljesen új anyagot tárgyal. Először bemutatjuk a `__slots__` osztályszintű változót, amely az egyes objektumok által felhasznált memória csökkentését segíti elő. Ez után azt tanuljuk meg, hogy miként lehet tulajdonságok használata nélkül hozzáférni jellemzőkhöz. Ebben a részben tárgyaljuk még a funktorokat (olyan objektumokról van szó, amelyeket a függvényekhez hasonlóan lehet meghívni), és a környezetkezelőket – ez utóbbiakat a `with` kulcsszóval összefüggésben használják, és sok esetben (például fájlkezeléskor) jók arra, hogy a `try...except...finally` szerkezet helyett az egyszerűbb `try...except` szerke-

zetet használhassuk. A szakasz azt is bemutatja, hogy miként hozhatók létre saját, egyedi környezetkezelőink, és bemutat néhány magasabb szintű objektumközpontú képességet, például az osztályszintű díszítőket, az elvont alaposztályokat, a többszörös öröklést és a metaosztályokat.

A harmadik rész megismertet bennünket a függvényközpontú (funkcionális) programozás néhány alapelvével, és bemutat néhány hasznos, a `functools`, az `itertools` és az `operator` modulban található függvényt. Azt is megtárgyaljuk, hogy miként használható a részleges függvényközpontú programozás a programkód egyszerűsítésére.

Az eddigi fejezetek összességében a „Python-alapismereteket" tartalmazták. Mindazt, amit eddig tárgyaltunk, ez a fejezet egyszeriben „luxusminőségű" ismeretté alakítja át. Minden eredeti módszerünk és utasításformánk megmarad, sőt kiegészül olyanokkal, amelyek a programozást könnyebbé, rövidebbé és hatékonyabbá teszik. Néhány eszköz felcserélhető egymással, vannak például olyan feladatok, amelyeket osztályszintű díszítőkkel és meta-osztályokkal egyaránt el lehet végezni. Más eszközök esetében – ilyenek például a leírók – több felhasználási mód létezik, és ezek a hatásukban is különböznek. A következőkben tárgyalt eszközök közül néhányat – például a környezetkezelőket – szinte egyfolytában használni fogjuk, mások viszont talonba kerülnek, és ott várják ki azokat a bizonyos helyzeteket, amelyekben ők jelentik a tökéletes megoldást.

Magasabb szintű eljárásközpontú programozás

Ennek a szakasznak a legnagyobb része az eljárásközpontú programozással kapcsolatos újabb eszközökkel foglalkozik. A legelső rész annyiban különbözik a többitől, hogy anélkül ismertet meg bennünket hasznos programozási módszerekkel, hogy egyetlen új utasításformát is meg kellene ismernünk – csak a már elsajátítottakra támaszkodunk.

Elágazások létrehozása szótárak segítségével

Ahogy már korábban is jeleztük, a Pythonban a függvények minden más elemhez hasonlóan objektumok, a függvény neve pedig a függvényre mutató objektumhivatkozás. Ha egy függvény nevét zárójelek nélkül írjuk le, a Python számára nyilvánvaló, hogy az objektumhivatkozásra gondolunk. Az ilyen objektumhivatkozások a többihez hasonlóan adhatók át a programokban. Ennek a ténynek az ismeretében a sok `elif` záradékot tartalmazó `if` utasítások egyetlen függvény hívásával helyettesíthetők. A 11. fejezetben át fogjuk tekinteni a `dvds-dbm.py` nevű interaktív konzolprogramot, amelynek a menüje az alábbi:

```
(A)dd (E)dit (L)ist (R)emove (I)mport e(X)port (Q)uit
```

A programban van egy függvény, amely a felhasználó választásának beolvasásáért felel, és amely csak érvényes választást ad vissza, esetünkben az „a", „e", „l", „r", „i", „x" és „q" karakterek valamelyikét. Az alábbiakban két egyenértékű kódrészletet ismertetünk; mindkettő a felhasználótól érkező bemenetnek megfelelő függvény hívását végzi el.

```
if action == "a":
    add_dvd(db)
elif action == "e":
    edit_dvd(db)
elif action == "l":
    list_dvds(db)
elif action == "r":
    remove_dvd(db)
elif action == "i":
    import_(db)
elif action == "x":
    export(db)
elif action == "q":
    quit(db)
```

```
functions = dict(a=add_dvd, e=edit_dvd,
l=list_dvds, r=remove_dvd,
i=import_, x=export, q=quit)
functions[action](db)
```

A bemenetet az `action` nevű változóban egyetlen karakterből álló karakterláncként tároljuk, a használandó adatbázis pedig a `db` nevű változóba kerül. Az `import_()` függvény nevében azért van aláhúzás, mert így különböztetjük meg a beépített `import` utasítástól.

A jobb oldalon levő kódrészletben létrehozunk egy szótárat, amelynek a kulcsai a felhasználó által megadható érvényes választási lehetőségek, az értékei pedig objektumhivatkozások. A második utasításban beillesztjük az adott műveletnek megfelelő objektumhivatkozást, és a függvényhívást végző műveleti jellel, vagyis a zárójellel meghívjuk a hivatkozott függvényt. Ebben a példában a `db` paramétert adjuk át.

A jobb oldalon lévő kódrészlet nemcsak lényegesen rövidebb, mint a bal oldali, de könnyen megbirkózik a méretezésből eredő problémákkal is (akár sokkal több elemű szótárat is használhatunk), és teszi mindezt anélkül, hogy a teljesítménye megváltozna. Nem így a bal oldali kódrészlet, amelynek a sebessége attól függ, hogy hány elágazást kell kiértékelni, amíg megtaláljuk azt a függvényt, amelyet meg kell hívnunk.

Az előző fejezetben megismert `convert-incidents.py` program is a fent ismertetett módszert használja az `import_()` tagfüggvényében, amint az a tagfüggvény alábbi részletéből is kiderül:

```
call = {(".aix", "dom"): self.import_xml_dom,
        (".aix", "etree"): self.import_xml_etree,
        (".aix", "sax"): self.import_xml_sax,
        (".ait", "manual"): self.import_text_manual,
        (".ait", "regex"): self.import_text_regex,
        (".aib", None): self.import_binary,
        (".aip", None): self.import_pickle}
result = call[extension, reader](filename)
```

A teljes tagfüggvény mindössze tizenhárom sorból áll, az `extension` paramétert magában a tagfüggvényben számítjuk ki, a `reader` paramétert pedig a hívó adja át. A szótár kulcsai kételemű rendezett sorozatok, az értékek pedig tagfüggvények. Ha `if` utasításokat használtunk volna, a kód huszonkét sornyi lenne, a méretezés pedig mindig gondot okozna.

Generátorkifejezések és -függvények

Generátorfüggvények 277 ☞

A generátorfüggvényekkel és -tagfüggvényekkel még a 6. fejezetben ismerkedtünk meg. Lehetőség van generátorkifejezések létrehozására is, amelyek nyelvtanilag szinte teljesen megegyeznek a listaértelmezésekkel, a különbség mindössze annyi, hogy szögletes zárójelek helyett egyszerű zárójelek közé kell írni őket. Az alábbi utasításformák használhatók:

```
(kifejezés for elem in bejárható_objektum)
(kifejezés for elem in bejárható_objektum if feltétel)
```

Az előző fejezetben `yield` kifejezések használatával hoztunk létre olyan tagfüggvényeket, amelyek adatszerkezetek bejárására szolgálnak. Az alábbiakban következik két egyenértékű kódrészlet, amelyekkel azt mutatjuk be, hogy egy `yield` kifejezés miként használható generátorként, ha `for...in` ciklusba helyezzük:

```
def items_in_key_order(d):
for key in sorted(d):
yield key, d[key]
```

```
def items_in_key_order(d):
    return ((key, d[key])
            for key in sorted(d))
```

Mindkét függvény olyan generátort ad vissza, amely egy olyan listát állít elő, amelynek az elemei az adott szótár kulcs-érték párjai. Ha minden elemre egyszerre van szükségünk, akkor a függvényektől visszakapott generátort átadhatjuk a `list()` vagy a `tuple()` függvénynek. Egyéb esetekben a generátor bejárásával kapjuk meg a szükséges elemeket.

A generátorok a „lusta" kiértékelés eszközei lehetnek, ugyanis csak azokat az értékeket számítják ki, amelyekre ténylegesen szükség van. Ez bizonyos körülmények között hatékonyabb, mintha mondjuk például egy lista összes értékét egy menetben számítanánk ki. Néhány generátor annyi értéket állít elő, amennyit kérünk tőle – felső határ nincs. Például:

```
def quarters(next_quarter=0.0):
    while True:
        yield next_quarter
        next_quarter += 0.25
```

A függvény a 0.0, 0.25, 0.5 stb. értékeket állítja elő örökkön örökké. A probléma megoldására az alábbi módon használhatunk generátort:

```
result = []
for x in quarters():
result.append(x)
if x >= 1.0:
break
```

A `break` utasítás létfontosságú: nélküle sohasem fejeződne be a `for...in` ciklus. Az eredménylista: `[0.0, 0.25, 0.5, 0.75, 1.0]`.

Minden alkalommal, amikor a `quarters()` függvényt meghívjuk, egy olyan generátort kapunk, amely 0.0-tól indul, és 0.25-tel emelkedik. Mi történik akkor, ha a generátor aktuális értékét vissza szeretnénk állítani a kezdőértékre? Lehetőség van arra, hogy a generátornak értéket adjunk át, mint ahogy azt a generátort megvalósító függvény alábbi, új változata is tanúsítja:

```
def quarters(next_quarter=0.0):
    while True:
    received = (yield next_quarter)
    if received is None:
        next_quarter += 0.25
    else:
        next_quarter = received
```

A `yield` kifejezés az egyes értékeket a hívónak adja vissza. Ezen felül, amikor a hívó a generátor `send()` tagfüggvényét hívja, az elküldött értéket a generátorfüggvény a `yield` kifejezés eredményeként kapja meg. Az alábbi példa az új generátorfüggvény használatát mutatja be:

```
result = []
generator = quarters()
while len(result) < 5:
    x = next(generator)
    if abs(x - 0.5) < sys.float_info.epsilon:
        x = generator.send(1.0)
    result.append(x)
```

Létrehozunk egy a generátorra hivatkozó változót, majd a beépített `next()` függvényt hívjuk meg, amely a neki átadott generátorból kinyeri a következő elemet. (Ugyanezt a hatást érhetjük el, ha a generátor `__next__()` nevű, különleges szerepű tagfüggvényét hívjuk meg. Ebben az esetben az utasítás formája `x = generator.__next__()`.) Ha az érték 0.5-del egyezik meg, akkor az 1.0 értéket küldjük át a generátornak (amely azonnal ugyanezt az értéket állítja elő). Ezúttal az eredménylista: `[0.0, 0.25, 1.0, 1.25, 1.5]`.

A következőkben a `magic-numbers.py` nevű programot fogjuk áttekinteni, amely a parancssorban megadott fájlokat dolgozza fel. Sajnos a Windows héjprogramja (a `cmd.exe`) nem kezeli tökéletesen a helyettesítő karaktereket (vagyis hiányzik belőle az úgynevezett *file globbing* nevű képesség), így aztán, ha egy programot Windowson a *.* paraméterrel futtatunk, a `sys.argv` lista – az aktuális könyvtárban lévő összes fájl helyett – betű szerint a „*.*" karakterlistát kapja meg. Ezt a problémát úgy oldjuk meg, hogy kétféle `get_files()` függvényt hozunk létre – az egyiket a Windows, a másikat a Unix számára. Mindkét függvény generátort használ. Íme a kód:

```
if sys.platform.startswith("win"):
    def get_files(names):
        for name in names:
            if os.path.isfile(name):
                yield name
            else:
                for file in glob.iglob(name):
                    if not os.path.isfile(file):
                        continue
                    yield file
else:
    def get_files(names):
        return (file for file in names if os.path.isfile(file))
```

A függvény paramétere mindkét esetben egy fájlneveket tartalmazó lista, például: `sys.argv[1:]`.

Windowson a függvény bejárja a listában szereplő neveket, és minden egyes fájlnév esetében előállítja magának a fájlnevet. A nem fájlokat takaró nevek esetében (ezek rendszerint könyvtárak) a `glob` modul `glob.iglob()` függvényét használjuk arra, hogy a helyettesítő karakterek értelmezése során előálló fájlnévlistát tartalmazó bejárható objektumot adjunk vissza. Rendes nevek, például az `autoexec.bat` esetében egyetlen elemből (a névből) álló bejárható objektumot adunk vissza. Azoknak a neveknek az esetében, amelyekben helyettesítő karakterek vannak – ilyen például a `*.txt` –, olyan bejárható objektumot adunk vissza, amely tartalmazza a helyettesítő karakterekkel megadott paraméterre illeszkedő fájlneveket (esetünkben azokat, amelyeknek `.txt` a kiterjesztése). (Létezik a `glob.glob()` tagfüggvény is, amely bejárható objektum helyett listát ad vissza.)

Unixon a parancshéj elvégzi számunkra a helyettesítő karakterek illesztését, így aztán csak annyi a dolgunk, hogy visszaadjuk az átadott nevek alapján létrehozott generátort.*

A generátorfüggvények *társrutinok* (coroutine) létrehozására is használhatók. A társrutinok olyan függvények, amelyek több be- és kilépési ponttal bírnak (ezek a `yield` utasítást tartalmazó kifejezések), és amelyek bizonyos pontoknál (szintén `yield` utasítást tartalmazó kifejezésekről van szó) felfüggeszthetők, illetve folytathatók. A társrutinokat gyakran használják a többszálúság helyett; egyszerűbb és alacsonyabb terheléssel járó megoldást jelenthetnek. A Python Package Index webhelyéről (`pypi.python.org/pypi`) számos, a társrutinokkal foglalkozó modul tölthető le.

---

* A `glob.glob()` függvények a képességeiket tekintve nem lehetnek méltó társai mondjuk a Unix bash parancshéjának, ugyanis a `*`, a `?` és a `[]` műveletek mellett hiányzik belőlük a `{}` művelet támogatása is.

Dinamikus kódvvégrehajtás és dinamikus betöltés

Vannak olyan helyzetek, amikor egyszerűbb, ha a kód közvetlen megírása helyett azt a kódot írjuk meg, amelyik előállítja a számunkra szükséges kódot. Vannak olyan helyzetek is, amelyekben kifejezetten hasznos, ha a kódot a felhasználó írja meg (gondolunk itt a táblázatkezelők függvényeire), amelyet majd a Python lefuttat. Így megtakarítható a beírt kód értelmezését végző programrészek megírása – ám azt ne felejtsük el, hogy az ilyen típusú közvetlen kódvégrehajtás természetesen biztonsági kockázatot rejt magában.
A dinamikus kódvégrehajtás másik felhasználási területe olyan bővítmények (beépülő modulok) használata, amelyek egy adott program képességeit hivatottak kiterjeszteni.
A bővítmények használatának megvan az a hátránya, hogy a programunk nem tartalmazza beépítetten az összes szükséges képességet – ez a program telepítésekor jelenthet nehézségeket, hiszen magában rejti a bővítmények elvesztésének lehetőségét. Ugyanakkor a módszernek megvan az az előnye, hogy a bővítmények újabb változatai egymástól függetlenül telepíthetők, illetve a modulok telepítése a főprogramtól függetlenül is történhet – akár olyan esetekben is, amikor a programunkat előre nem tervezett képességekkel kell felruháznunk.

Dinamikus kódvégrehajtás

Egy kifejezés kiértékeltetésének legegyszerűbb módja, ha a 6. fejezetben megismert, beépített `eval()` függvényt használjuk. Például:

```
x = eval("(2 ** 31) - 1")     # x == 2147483647
```

Ez a módszer megfelelő, ha a felhasználó által beírt kifejezések futtatásáról van szó. De mi a helyzet akkor, ha a program futása közben kell létrehoznunk egy függvényt? Ilyen esetekben használhatjuk a beépített `exec()` függvényt. Például elképzelhető, hogy a felhasználó a $4\square r^2$ képletet írja be, azonosítóként pedig a „gömb felszíne" nevet adja meg. Ebből szeretnénk függvényt készíteni. Feltételezve, hogy a $\square$ helyére a `math.pi` állandó kerül, a függvény az alábbiak szerint hozható létre:

```
import math
code = '''
def area_of_sphere(r):
    return 4 * math.pi * r ** 2
'''

context = {}
context["math"] = math
exec(code, context)
```

Megfelelő behúzást kell alkalmaznunk, hiszen az idézőjelek között szereplő kód szabványos Python-kód. (Bár ebben az esetben írhattunk volna mindent egyetlen sorba, hiszen maga a képlet egyetlen sorból áll.)

Ha az `exec()` függvény paramétere mindössze némi kód, akkor a kód végrehajtása során létrejövő függvények, illetve változók elérésére nincs mód. Ezen felül az `exec()` függvény képtelen elérni a betöltött modulokat, illetve bármely, a hívás helyének hatókörében lévő

változót, függvényt vagy egyéb objektumot. A fent említett problémák egyaránt megoldódnak, ha az `exec()` függvénynek második paraméterként egy szótárat adunk át. A szótár biztosítja az objektumhivatkozások tárolására szolgáló helyet. Ezek az objektumhivatkozások az `exec()` függvény végrehajtása után is elérhetők maradnak. Például a `context` nevű szótár használata arra szolgál, hogy az `exec()` függvénnyel létrehozott `area_of_sphere()` függvényre mutató objektumhivatkozást tárolni tudjuk. Ebben a példában az `exec()` függvényre azért volt szükség, hogy elérhessük a `math` modult, ezért a `context` nevű szótárban olyan elemet helyeztünk el, amelynek a kulcsa a modul neve, az értéke pedig a megfelelő modulobjektumra mutató objektumhivatkozás. Így az `exec()` függvény hívása alatt is lehetővé válik a `math.pi` állandó elérése.

Néhány esetben kényelmes lehet, ha az `exec()` függvény számára minden globálisan elérhető objektumot elérhetővé teszünk. Ez úgy oldható meg, hogy a `globals()` függvénytől visszakapott szótárat adjuk át az `exec()` függvénynek. A megközelítés egyik hátránya, hogy minden olyan objektum, amely az `exec()` függvény hívása alatt jön létre, bekerül ebbe a szótárba. Megoldást az jelent, ha a globálisan elérhető objektumokat bemásoljuk egy szótárba, például a `context = globals().copy()` utasítással. Így az `exec()` függvény még mindig hozzáfér a betöltött modulokhoz, illetve a hívás helyének hatókörében lévő változókhoz és egyéb objektumokhoz; minthogy azonban másolatot hoztunk létre, az `exec()` függvény futása alatt felmerülő változások a `context` nevű szótárban maradnak, és nem válnak a globális környezet részévé. (Arra gondolhatnánk, hogy biztonságosabb, ha inkább a `copy.deepcopy()` függvényt használjuk, de ha biztonsági megfontolások kerülnek szóba, akkor az a legokosabb, ha teljes egészében tartózkodunk az `exec()` függvény használatától.) A helyi hatókörben lévő változók és objektumok is átadhatók az `exec()` függvénynek – például úgy, hogy harmadik paraméterként a `locals()` függvényt adjuk meg.

Az `exec()` függvény hívása után a `context` szótár tartalmaz egy „`area_of_sphere`" nevű kulcsot, amelynek az értéke az `area_of_sphere()` függvény. Íme a függvény elérése és hívása:

```
area_of_sphere = context["area_of_sphere"]
area = area_of_sphere(5)     # area == 314.15926535897933
```

Az `area_of_sphere` objektum a dinamikusan létrehozott függvényre mutató objektumhivatkozás, és pontosan ugyanúgy használható, mint az összes többi függvény. És ámbár ezúttal az `exec()` függvény hívása során csak egyetlen függvényt hoztunk létre, eltérően az `eval()` függvénytől, amely csak egyetlen kifejezés kezelésére képes, az `exec()` függvénnyel annyi Python-nyelvű kifejezést kezelhetünk, amennyi csak jól esik. Ezalatt érthetünk egész modulokat is, ahogy az a következő részből kiderül.

Modulok dinamikus betöltése

A Pythonban három egyszerűen használható módszer áll a rendelkezésünkre bővítmények létrehozására. Mindhárom módszer használata során futásidőben, név szerint kell modulokat betöltenünk. A modulok betöltésének végeztével a Python vizsgálófüggvényeit használhatjuk arra, hogy megbizonyosodjunk a használni kívánt függvények létéről és elérhetőségéről.

A következőkben a `magic-numbers.py` programot fogjuk áttekinteni. A program a parancssorban megadott fájlokból beolvassa az első ezer bájtot, és megjeleníti az egyes fájlok típusát – illetve az „Unknown" (ismeretlen) szöveget – és a fájl nevét. Íme egy lehetséges parancssor, illetve a program futtatása során keletkezett kimenet részlete:

```
C:\Python30\python.exe magic-numbers.py c:\windows\*.*
...
XML.................c:\windows\WindowsShell.Manifest
Unknown.............c:\windows\WindowsUpdate.log
Windows Executable..c:\windows\winhelp.exe
Windows Executable..c:\windows\winhlp32.exe
Windows BMP Image...c:\windows\winnt.bmp
...
```

A program minden olyan modul betöltésével megpróbálkozik, amely vele azonos könyvtárban található, és a neve tartalmazza a „magic" karaktersorozatot. Az ilyen moduloktól azt várjuk, hogy egyetlen nyilvános függvényt tartalmaznak, amelynek a neve: `get_file_type()`. A könyv példái között szerepel két nagyon egyszerű példamodul, a `StandardMagicNumbers.py` és a `WindowsMagicNumbers.py`. Mindkettő rendelkezik `get_file_type()` nevű függvénnyel. A program `main()` függvényét két részletben tekintjük át.

```
def main():
    modules = load_modules()
    get_file_type_functions = []
    for module in modules:
    get_file_type = get_function(module, "get_file_type")
    if get_file_type is not None:
        get_file_type_functions.append(get_file_type)
```

Hamarosan megismerkedünk a `load_modules()` függvény három különböző megvalósításával. A függvény a modulobjektumok (esetleg üres) listáját adja vissza. Megismerkedünk a `get_function()` függvénnyel is. Minden megtalált modulnál megpróbálkozunk a `get_file_type()` függvény létének ellenőrzésével, és ha találunk ilyet, akkor elhelyezzük a függvényeket tároló listánkban.

```
for file in get_files(sys.argv[1:]):
    fh = None
    try:
        fh = open(file, "rb")
        magic = fh.read(1000)
```

```
        for get_file_type in get_file_type_functions:
            filetype = get_file_type(magic,
                                     os.path.splitext(file)[1])
            if filetype is not None:
                print("{0:.<20}{1}".format(filetype, file))
                break
        else:
            print("{0:.<20}{1}".format("Unknown", file))
    except EnvironmentError as err:
        print(err)
    finally:
        if fh is not None:
            fh.close()
```

A fenti ciklus bejárja a parancssorban megadott fájlokat, és beolvassa az egyes fájlok első ezer bájtját. Ennek végeztével kipróbál minden egyes `get_file_type()` függvényt, és megpróbálja megállapítani az aktuális fájl típusát. Ha a fájltípus megállapítása sikeres, a fájl adatait a képernyőre írjuk, és kilépünk a belső ciklusból, a következő fájlnál folytatva a feldolgozást. Ha a fájl típusának megállapítására egyetlen függvény sem volt képes – illetve, ha nem találtunk egyetlen `get_file_type()` függvényt sem –, az „Unknown" szót írjuk a képernyőre.

A modulok dinamikus betöltésének az alábbiakban három különböző, ámde egyenértékű módszerét tekintjük át. Először a leghosszabb és legbonyolultabb megközelítési módot tekintjük, mivel ez minden egyes lépést bemutat:

```
def load_modules():
    modules = []
    for name in os.listdir(os.path.dirname(__file__) or "."):
        if name.endswith(".py") and "magic" in name.lower():
            filename = name
            name = os.path.splitext(name)[0]
            if name.isidentifier() and name not in sys.modules:
            fh = None
            try:
                fh = open(filename, "r", encoding="utf8")
                code = fh.read()
                module = type(sys)(name)
                sys.modules[name] = module
                exec(code, module.__dict__)
                modules.append(module)
            except (EnvironmentError, SyntaxError) as err:
                sys.modules.pop(name, None)
                print(err)
            finally:
                if fh is not None:
                    fh.close()
    return modules
```

Első lépésként a programban lévő fájlokat járjuk be. Ha az aktuális könyvtárról van szó, az `os.path.dirname(__file__)` függvény üres karakterláncot ad vissza, aminek folyományaként az `os.listdir()` függvény kivételt vált ki. Szükség esetén inkább átadunk egy pontot (`".."`). Minden alkalmasnak látszó fájl esetében (`.py` kiterjesztése van, és a nevében szerepel a „magic" szó) úgy kapjuk meg a modul nevét, hogy a fájl kiterjesztésénél csonkolunk. Ha az így létrejött név valós azonosító – tehát használható modulnévként –, és nincs még a `sys.modules` szótárban tárolt globális modulnévlistában, akkor megpróbálkozunk a betöltésével.

A fájl szövegét a `code` karakterláncba olvassuk be. A következő sor, a `module = type(sys)(name)`, meglehetősen érdekes. Amikor a `type()` függvényt meghívjuk, a neki átadott objektum típusának megfelelő objektumot kapunk vissza. Tehát például a `type(1)` hívás eredménye egy `int` objektum lenne. Ha a típust ábrázoló objektumot le kell írni, akkor valamilyen ember által olvasható kimenetet kapunk (például „int"). Ha azonban a típusobjektumot függvényként hívjuk meg, akkor az adott típusba tartozó objektum lesz a visszatérési érték. Például ha azt írjuk, hogy `x = 5`, akkor az `x` változóban az `5` egész számot kapjuk eredményül. Ugyanez az eredmény, ha az alábbi utasítások egyikét adjuk ki: `x = int(5); x = type(0)(5); int_type = type(0); x = int_type(5)`. Esetünkben a kiadott utasítás: `type(sys)`, a `sys` pedig egy modul. Így aztán modul típusú objektumot kapunk vissza (valójában olyan, mintha osztály típusú objektumot kapnánk), amit arra használhatunk, hogy az adott név alapján új modult hozzunk létre. Az előző, `int` objektumról szóló példához hasonlóan, ahol nem számított, hogy melyik egész számot használtuk az `int` típusú objektum előállítására, most sem számít, hogy melyik modult használjuk (már amennyiben létező modult használunk, vagyis olyat, amelyet sikeresen betöltöttünk) a modul típusú objektum előállításához.

Ha az új – egyelőre üres – modul létrejött, elhelyezzük a modulokat tartalmazó globális listában, mégpedig azért, hogy nehogy véletlenül újból betöltsük. Ez még az `exec()` függvény hívása előtt történik meg, azért, hogy minél inkább megközelítsük az `import` utasítás működését. Ha készen vagyunk, az `exec()` függvényt meghívva végrehajtjuk a beolvasott kódot – és a modul könyvtárát használjuk a kód hatókörébe tartozó objektumok láthatóvá tételére. Ha ezzel is elkészültünk, elhelyezzük a modult abban a listában, amelyet visszaadunk. Ha viszont problémába ütközünk, töröljük a modult a modulokat tartalmazó globális szótárból – már amennyiben belekerült: ha ugyanis hiba történik művelet közben, akkor a modul nem kerül bele a listába. Tartsuk észben, hogy az `exec()` függvény bármekkora mennyiségű kód kezelésére képes (ugyanakkor az `eval()` függvény egyetlen kifejezés kiértékelését tudja elvégezni – lásd a 8.1. táblázatot), ha pedig nyelvtani hibába ütközne, `SyntaxError` kivételt vált ki.

8.1. táblázat *A dinamikus programozást és az objektumok vizsgálatát segítő függvények*

Utasításforma	Leírás
`__import__(...)`	Modult tölt be név alapján; lásd a szöveget.
`compile(`*`forrás, fájl, mód`*`)`	A *`forrás`* szöveg fordításából származó kódobjektummal tér vissza, a *`fájl`* a fájl neve vagy `"<string>"`; a *`mód`* lehetséges értékei: „single", „eval" vagy „exec".
`delattr(`*`obj, név`*`)`	Az *`obj`* objektumból törli a *`név`* jellemzőt.
`dir(`*`obj`*`)`	Az aktuális hatókör – illetve, ha az *`obj`* szerepel, akkor az *`obj`* objektum – jellemzőinek és tagfüggvényeinek neveiből álló listát ad vissza.
`eval(`*`forrás, globális_változók, helyi_változók`*`)`	A *`forrás`*-ban lévő egyetlen kifejezés kiértékeléséből származó eredménnyel tér vissza. Ha megadtuk, akkor a *`globális_változók`*-ban a globális, a *`helyi_változók`*-ban a helyi környezet található – szótárak formájában.
`exec(`*`forrás, globális_változók, helyi_változók`*`)`	A *`forrás`* (ami egy karakterlánc vagy a *`compile()`* függvény eredményeként előállt kódobjektum lehet) kiértékelése után `None` eredménnyel tér vissza. Ha megadtuk, akkor a *`globális_változók`*-ban a globális, a *`helyi_változók`*-ban a helyi környezet található.
`getattr(`*`obj, név, érték`*`)`	Az *`obj`* objektum *`név`* jellemzőjének értékével tér vissza, illetve az *`érték`*-kel, ha azt megadtuk, és a megadott jellemző nem létezik.
`globals()`	Az aktuális globális környezetből képzett szótárat adja vissza.
`hasattr(`*`obj, név`*`)`	`True` értékkel tér vissza, ha az *`obj`* objektum rendelkezik *`név`* nevű jellemzővel.
`locals()`	Az aktuális helyi környezetből képzett szótárat adja vissza.
`setattr(`*`obj, név, érték`*`)`	Az *`obj`* objektum *`név`* jellemzőjének értékét *`érték`*-re állítja; a jellemzőt szükség esetén létrehozza.
`type(`*`obj`*`)`	Az *`obj`* objektum típusobjektumával tér vissza.
`vars(`*`obj`*`)`	Az *`obj`* objektum környezetét adja vissza szótár formájában; ha az *`obj`* paramétert elhagyjuk, akkor a helyi környezetet.

Íme a második módszer, amellyel egy modul dinamikusan, futásidő alatt tölthető be – az alábbi kódrészlet az első megközelítés `try...except` kódblokkját helyettesíti:

```
try:
    exec("import " + name)
    modules.append(sys.modules[name])
except SyntaxError as err:
    print(err)
```

A fenti megközelítéssel az az elméleti probléma adódik, hogy biztonsági kockázatokat rejt magában. A `name` változó elején szerepelhet a `sys` szó, amelyet akár rossz szándékú kód is követhet.

És itt következik a harmadik megközelítés is, amely megint csak az első módszerhez tartozó kódrészletben lévő `try...except` kódblokkot cseréli le:

```
try:
    module = __import__(name)
    modules.append(module)
except (ImportError, SyntaxError) as err:
    print(err)
```

A fenti megközelítés kínálja a legegyszerűbb módot modulok dinamikus betöltésére. A módszer lényegesen biztonságosabb az `exec()` függvény használatánál, bár egyáltalán nem tekinthető biztonságosnak, mint ahogy a dinamikus betöltési módszerek egyike sem: soha nem lehetünk biztosak abban, hogy egy modul betöltésekor miféle kódot futtatunk.

Az előzőekben bemutatott módszerek egyike sem képes olyan csomagok, illetve modulok kezelésére, amelyek nem az aktuális könyvtárban vannak; és nem jelent nehézséget úgy megváltoztatni a kódot, hogy az ilyen helyzetek kezelésére is alkalmassá váljon. Mindazonáltal, amennyiben kifinomultabb működés a cél, érdemes elolvasni az internetes dokumentációt, különösen az `__import__()` függvény esetében.

Most, hogy a modul betöltésével végeztünk, használatba kell vennünk a benne megvalósított képességeket. Ez úgy oldható meg, hogy kiaknázzuk a Python nyelv beépített vizsgálati képességeit, azaz a `getattr()` és a `hasattr()` függvényeket. Az alábbi kódrészlet bemutatja, hogy miként használtuk ezeket a függvényeket a `get_function()` függvény megvalósításakor.

```
def get_function(module, function_name):
    function = get_function.cache.get((module, function_name), None)
    if function is None:
        try:
            function = getattr(module, function_name)
            if not hasattr(function, "__call__"):
                raise AttributeError()
            get_function.cache[module, function_name] = function
        except AttributeError:
            function = None
    return function
get_function.cache = {}
```

`collections.Callable` osztály
☞ 399

A gyorsítótárral kapcsolatos kódrészletet egy pillanatra hagyjuk figyelmen kívül. A függvény annyit tesz, hogy a modulobjektumra meghívja a `getattr()` függvényt, mégpedig az általunk kívánt névvel. Ha a megjelölt jellemző nem létezik, `AttributeError` kivétel lép fel. Ha azonban megtaláljuk a keresett jellemzőt, akkor a `hasattr()` függvénnyel ellenőrizzük, hogy a jellemző maga rendelkezik-e a `__call__`

nevű jellemzővel. Ez a jellemző olyasmi, amellyel minden hívható objektum (függvény és tagfüggvény) rendelkezik. (A továbbiakban ismertetni fogunk egy elegánsabb módszert annak eldöntésére, hogy egy bizonyos jellemző meghívható-e vagy sem.) Amennyiben a jellemző létezik és hívható, visszaadhatjuk a hívónak, egyéb esetekben pedig `None` objektumot adunk vissza, ezzel jelezve, hogy a függvény nem megfelelő.

Ha fájlok százainak feldolgozását kell elvégezni (például azért, mert a `C:\windows` könyvtárban a `*.*` paraméterrel futtattuk a programot), nem szeretnénk elvégezni a keresést minden fájl és minden modul esetében. Így aztán mihelyt megadjuk a `get_function()` függvényt, el is helyezünk benne egy jellemzőt, méghozzá egy `cache` nevű szótárat. (A Pythonban szinte mindig lehetőségünk nyílik a saját készítésű objektumokat nekünk tetsző nevű jellemzőkkel felruházni.) Amikor a `get_function()` függvényt először hívjuk meg, a `cache` szótár még üres, így a `dict.get()` függvény hívása `None` objektumot ad vissza. Minden olyan alkalommal azonban, amikor megfelelő függvényre találtunk, a függvény bekerül a szótárba, mégpedig kételemű rendezett sorozatként: a modul- és a függvénynév lesz a kulcs, a függvény maga pedig az érték. Így aztán a második esettől kezdve minden olyan alkalommal, amikor egy adott függvényre van szükség, a függvényt a gyorsítótárból olvassuk ki, és semmiféle keresés nem történik a jellemzőkön.*

Azt a módszert, amelyet a `get_function()` függvény egy adott paraméterhalmazra vonatkozó visszatérési értékeinek gyorsítótárazására használtunk, *jegyzetelésnek* (memoizing) nevezzük. Minden olyan függvény esetében használható, amelynek nincsenek mellékhatásai (nem változtat a globális változókon), és amelyek ugyanannak a (nem módosítható) paraméternek az átadásakor mindig ugyanazt az eredményt adják vissza. Mivel a jegyzetelt függvények gyorsítótárainak létrehozásához és kezeléséhez szükséges kód ugyanaz, ideális jelöltnek mutatkozik arra, hogy függvénydíszítőt alakítsunk ki belőle. A Python Cookbook webhelyén (`code.activestate.com/recipes/langs/python/`) sok, ezzel kapcsolatos díszítőreceptet találunk. Minthogy azonban a modulobjektumok módosíthatók, bonyolultabb a dolgunk annál, semhogy egyszerűen leakasztunk a szögről magunknak valamiféle jegyzetelő díszítőt. Az ilyen függvények a mi `get_function()` függvényünkkel nem fognak azonnal működni. Egyszerű megoldásnak kínálkozik, hogy a modul helyett csak a modul `__name__` karakterláncát használjuk a kulcs rendezett sorozatának első elemeként.

Figyelem! A dinamikus modulbetöltés egyszerű, csakúgy, mint az `exec()` függvénnyel történő, tetszőleges Python-kód futtatása. Mindez néhány esetben meglehetősen kényelmes, például lehetőségünk van adatbázisban tárolni kódot. Ne feledjük azonban, hogy nem tudjuk ellenőrizni, hogy a betöltött vagy az `exec()` függvény segítségével végrehajtott kód mit csinál. Jusson eszünkbe, hogy a modulok nem csak változókat, függvényeket és osztályokat tartalmazhatnak, hanem olyan kódot is, amely

* A `magic-numbers.py` programban a fent bemutatott változaton kívül megtaláljuk a `get_function()` függvény egy lényegesen kifinomultabb változatát is, amely ügyesebben kezeli az olyan modulokat, amelyekben nincs meg a szükséges függvény.

lefut a modul betöltésének során. Ha a kód ellenőrizetlen forrásból érkezik, esetleg kellemetlenségeink adódhatnak. Az, hogy ezzel a problémával miként küzdünk meg, az adott helyzettől függ. Természetesen mindez nem feltétlenül okoz gondot – így a személyes projektek esetében sem.

Helyi függvények és önhívó függvények

Gyakran lehet hasznos, ha egy függvény belsejében megírunk egy-két segédfüggvényt. A Pythonban az ilyen esetekben nem kell semmilyen, az eddigiektől eltérő utasításformát használnunk – egyszerűen annyit teszünk, hogy a már létező függvény belsejében megadjuk a szükséges függvényeket. Az ilyen függvényeket gyakran nevezzük *beágyazott függvényeknek* vagy *helyi függvényeknek*. Helyi függvényekkel a 7. fejezetben már találkoztunk.

A helyi függvények igénybe vételének egyik legelterjedtebb esete az önhívás (rekurzió) használata. Ilyenkor egy külső függvényt hívunk meg, amely beállít egy-két dolgot, majd első alkalommal meghív egy helyi önhívó függvényt. Az önhívó függvények (vagy tagfüggvények) olyan függvények, amelyek saját magukat hívják meg. Szerkezetét tekintve minden önhívó függvény két esetet különböztet meg: az *alapesetet* és az *önhívó esetet*. Az alapeset szerepe az önhívás megállítása.

Az önhívó függvények számításigénye általában magas, mivel minden önhívás során új memóriavermet foglalunk, de van néhány olyan algoritmus, amelyeknek a legtermészetesebb megvalósítási formája önhívást kíván. A legtöbb Python-megvalósításban meghatározott az önhívások maximális száma. A maximális szám kiolvasása a `sys.getrecursionlimit()` függvénnyel, beállítása pedig a `sys.setrecursionlimit()` függvénnyel történik. Megjegyeznénk azonban, hogy a maximális érték növelése általában annak a jele, hogy a használt algoritmus nem megfelelő, vagy a megvalósítás hibás.

Az önhívó függvények klasszikus példája faktoriálisok* kiszámítását végzi. Például a `factorial(5)` az 5! kifejezés kiszámítását végzi, visszatérési értéke pedig 120, azaz 1 x 2 x 3 x 4 x 5:

```
def factorial(x):
    if x <= 1:
        return 1
    return x * factorial(x - 1)
```

A megoldás nem hatékony, de bemutatja az önhívó függvények két alapvető jellemzőjét. Ha a megadott szám (`x`) értéke 1 vagy annál kevesebb, a visszaadott érték is 1, önhívás pedig nem történik. Ez az alapeset. Ha viszont az `x` nagyobb, mint 1, akkor a visszaadott érték: `x * factorial(x - 1)`. Ez az önhívó eset, hiszen a `factorial` függvény itt saját magát hívja meg. A függvény befejeződése biztosított, hiszen ha az `x` kezdőértéke nem

* A Python `math` moduljában megtalálható a sokkal hatékonyabb `math.factorial()` függvény.

nagyobb 1-nél, akkor az alapesetet használjuk, és a függvény azonnal véget ér; ha pedig az x nagyobb 1-nél, akkor minden önhívás paramétere az előzőnél eggyel kisebb szám, vagyis előbb-utóbb ismét elérjük az alapesetet.

Ha a helyi, illetve az önhívó függvényeket valós környezetben akarjuk tanulmányozni, megfigyelhetjük az `IndentedList.py` modulban lévő `indented_list_sort()` függvényt. A függvény egyik paramétere egy karakterláncokból álló lista, mégpedig olyan, amelyben behúzásokkal jelezzük a hierarchiát. A másik paraméter egy első szintű behúzást tartalmazó karakterlánc. A kimenet szintén lista, amelyben ugyanazok a karakterláncok találhatók meg, de a kis- és nagybetűket figyelmen kívül hagyó ábécésorrendbe rendezve, és a behúzást tartalmazó elemeket a szülőelem alatt szerepeltetve. A 8.1. ábra `before` és `after` listái ezt szemléltetik.

Az `after` lista a `before` lista alapján az alábbi függvényhívással áll elő: `after = IndentedList.indented_list_sort(before)`. Az alapértelmezett behúzás négy szóköz, ugyanaz, amit a `before` listában is használtunk – így ezt nem kellett külön beállítanunk.

```
before = ["Nonmetals",
          "    Hydrogen",
          "    Carbon",
          "    Nitrogen",
          "    Oxygen",
          "Inner Transitionals",
          "    Lanthanides",
          "        Cerium",
          "        Europium",
          "    Actinides",
          "        Uranium",
          "        Curium",
          "        Plutonium",
          "Alkali Metals",
          "        Lithium",
          "        Sodium",
          "        Potassium"]
```

```
after = ["Alkali Metals",
         "    Lithium",
         "    Potassium",
         "    Sodium",
         "Inner Transitionals",
         "    Actinides",
         "        Curium",
         "        Plutonium",
         "        Uranium",
         "    Lanthanides",
         "        Cerium",
         "        Europium",
         "Nonmetals",
         "    Carbon",
         "    Hydrogen",
         "    Nitrogen",
         "    Oxygen"]
```

8.1. ábra

Behúzásokat tartalmazó lista – rendezés előtt és után

Az `indented_list_sort()` függvényt először egészében, majd két helyi függvényként tekintjük át.

```
def indented_list_sort(indented_list, indent=" "):
    KEY, ITEM, CHILDREN = range(3)

    def add_entry(level, key, item, children):
        ...

    def update_indented_list(entry):
        ...
```

```
    entries = []
    for item in indented_list:
        level = 0
        i = 0
        while item.startswith(indent, i):
            i += len(indent)
            level += 1
        key = item.strip().lower()
        add_entry(level, key, item, entries)
    indented_list = []
    for entry in sorted(entries):
        update_indented_list(entry)
    return indented_list
```

A kód elején létrehozunk három állandót, amelyeket később arra használunk, hogy a helyi függvényekben használt indexhelyeknek nevet adjunk. Ezt követően megadjuk azt a két helyi függvényt, amelyet rövidesen áttekintünk. A rendező algoritmus működése két lépésre osztható. Az első lépésben létrehozzuk a bejegyzések listáját. Minden bejegyzés egy-egy háromelemű rendezett sorozatot jelent. Az első elem a „kulcs", amelyet a rendezéskor fogunk használni; a második az eredeti karakterlánc; a harmadik pedig egy lista, amely a karakterlánc gyermekeit tartalmazza. A kulcs nem egyéb, mint a karakterláncból létrehozott kisbetűs változat, amelynek mindkét végéről eltávolítottuk az üreshelyeket. A szint a behúzás szintjét jelöli – a felső szintű elemek esetében 0, ezek gyermekelemeinél 1, és így tovább. A második lépésben létrehozunk egy új, behúzásokat tartalmazó listát, amelyben elhelyezzük az előző listában rendezetten lévő elemeket, az egyes karakterláncok alatti szinten szereplő karakterláncokat, és így tovább. Eredményül egy behúzásokat tartalmazó, rendezett listát kapunk.

```
def add_entry(level, key, item, children):
    if level == 0:
        children.append((key, item, []))
    else:
        add_entry(level - 1, key, item, children[-1][CHILDREN])
```

A fenti függvényt a listában lévő minden egyes karakterlánc esetében meghívjuk. A függvény `children` paramétere az a lista, amelybe az új bejegyzéseket helyezni kell. Amikor a függvényt a külső – `indented_list_sort()` – függvényből hívjuk meg, a paraméter az `entries` lista. A művelet eredménye az lesz, hogy a karakterláncokból álló lista bejegyzésekből álló listává alakul; minden bejegyzés egy felső szintű (behúzás nélküli) karakterláncból, illetve egy a gyermekelemeket tartalmazó (esetleg üres) listából áll.

Ha a 0. szinten vagyunk (vagyis a felső szinten), új, háromelemű rendezett sorozatot helyezünk el az `entries` listában. Ebben tároljuk a kulcsot (a rendezéshez kell), az eredeti elemet (amely majd a művelet eredményeként létrejövő listába kerül), illetve a gyermekelemek üres listáját. Ez az alapeset, hiszen ilyenkor nem történik önhívás. Ha a szint 0-nál

magasabb, akkor az elem a `children` lista utolsó elemének gyermekeleme (ha tetszik, leszármazottja). Ebben az esetben az `add_entry()` függvényt ismét önhívással hívjuk meg, a szintet eggyel csökkentjük, és a gyermekeket tartalmazó lista utolsó elemének gyermeklistáját adjuk át bővítendő listaként. Ha a szint legalább 2, akkor több önhívás is történik, egészen addig, amíg a szint 0 nem lesz, és a bejegyzés közvetlenül a `children` listába kerül.

Például amikor az „Inner Transitionals" karakterlánchoz érünk, a külső függvény 0 szinttel hívja az `add_entry()` függvényt, a kulcs pedig az „inner transitionals" lesz, az elem maga az „Inner Transitionals", a gyermekelemeket tartalmazó lista pedig az `entries`. Mivel a 0. szinten vagyunk, új, egy kulcsot, magát az elemet és egy üres gyermekelemlistát magában foglaló elem kerül a gyermekelemeket tartalmazó lista (az `entries`) végére. A következő karakterlánc a „ Lanthanides". Van már benne behúzás, vagyis az „Inner Transitionals" karakterlánc gyermekének fogjuk tekinteni. Az `add_entry()` függvényhívás ezúttal első szintet kap, a kulcs a „lanthanides", az elem maga a „ Lanthanides", a gyermekelemeket tartalmazó lista pedig az `entries`. Mivel az első szinten vagyunk, az `add_entry()` függvény meghívja saját magát, ezúttal már a 0. szinttel (1-1=0), ugyanazzal a kulccsal és elemmel, de a gyermekelemeket tartalmazó lista az előző elem gyermekelemeket tartalmazó listája lesz, vagyis az „Inner Transitionals" elemhez tartozó gyermekelemlista. Íme az `entries` lista a karakterláncok elhelyezése után, de még a rendezés előtt:

```
[('nonmetals',
  'Nonmetals',
  [('hydrogen', '   Hydrogen', []),
   ('carbon', '   Carbon', []),
   ('nitrogen', '   Nitrogen', []),
   ('oxygen', '   Oxygen', [])]),
 ('inner transitionals',
  'Inner Transitionals',
  [('lanthanides',
    '    Lanthanides',
    [('cerium', '        Cerium', []),
     ('europium', '        Europium', [])]),
   ('actinides',
    ' Actinides',
    [('uranium', '        Uranium', []),
     ('curium', '        Curium', []),
     ('plutonium', '        Plutonium', [])])]),
 ('alkali metals',
  'Alkali Metals',
  [('lithium', '        Lithium', []),
   ('sodium', '        Sodium', []),
   ('potassium', '        Potassium', [])])]
```

A kimenet előállítására a `pprint` („pretty print") nevű modul `pprint.pprint()` függvényét használjuk. Figyeljük meg, hogy az `entries` listának csak három eleme van (mindhárom elem egy-egy háromelemű rendezett sorozat), és hogy minden háromelemű `tuple` utolsó eleme egy újabb háromelemű `tuple` (esetleg egy üres lista).

Az `add_entry()` függvény egyszerre helyi és önhívó függvény. Mint minden önhívó függvény, ez a függvény is rendelkezik egy az önhívásnak véget vetni képes *alapesettel* (esetünkben arról az esetről van szó, amikor a szint 0), illetve egy önhívó esettel.

A függvényt megírhattuk volna némileg eltérő formában is:

```
def add_entry(key, item, children):
    nonlocal level
    if level == 0:
        children.append((key, item, []))
    else:
        level -= 1
        add_entry(key, item, children[-1][CHILDREN])
```

A fenti esetben a `level` (szint) átadása nem paraméterként történik, hanem a `nonlocal` kulcsszóval jelezzük, hogy a változót egy eggyel magasabb szinten lévő hatókörből is el szeretnék érni. Ha a függvény belsejében nem szeretnénk megváltoztatni a `level` változó értékét, akkor nincs szükség a `nonlocal` kulcsszóra – ilyen helyzetekben a Python ugyanis nem találja meg a változót a helyi (belső függvényre érvényes) hatókörben, és az eggyel kintebbi hatókörben keresi tovább, ahol meg is találja. Az `add_entry()` függvény fenti változatában azonban a `level` változó értékét meg kell változtatnunk. A `global` kulcsszóval jelezzük a Pythonnak, hogy egy globális változó értékén szeretnénk változtatni – így oldható meg, hogy a globális változó értékének frissítése helyett ne új, helyi változó jöjjön létre. Ugyanez vonatkozik minden olyan változóra, amelynek az értékét úgy akarjuk megváltoztatni, hogy maga a változó az aktuális hatókörön kívül helyezkedik el. Ámbár gyakran az a legokosabb, ha a `global` kulcsszó használatát egy az egyben elkerüljük, és szintén alaposan oda kell figyelnünk a `nonlocal` kulcsszó használatára.

```
def update_indented_list(entry):
    indented_list.append(entry[ITEM])
    for subentry in sorted(entry[CHILDREN]):
        update_indented_list(subentry)
```

Az algoritmus első lépésében feltöltjük a bejegyzések háromelemű sorozatokból (kulcs, maga az elem, a gyermekelemek listája) álló listáját – a sorrend nem változik az eredeti listához képest. Az algoritmus második lépésében új, üres, behúzásokat tartalmazó listából indulunk ki, bejárjuk a rendezett bejegyzéseket, és a behúzásokat tartalmazó listát úgy alakítjuk ki, hogy a bejáráskor minden elemnél az `update_indented_list()` függvényt hívjuk meg. Az `update_indented_list()` függvény önhívó. Minden felső szintű bejegyzésnél elhelyez egy elemet az `indented_list` nevű listában, majd a listában lévő egyes elemek gyermekelemeinél saját magát hívja, és így tovább. Az alapeset (amikor az önhívás leáll) akkor következik be, amikor egy elem vagy egy gyermekelem, vagy egy gyermekelem gyermekeleme (és így tovább) nem rendelkezik további saját gyermekelemekkel.

A Python az `indented_list` nevű objektumot a helyi (a belső függvényhez kapcsolódó) hatókörben keresi, de nem találja meg. Így aztán eggyel magasabb szintű hatókörben keresi tovább, ahol aztán meg is találja. Figyeljük meg azonban, hogy elemeket helyezünk el a függvény belsejében – az `indented_list` listában –, holott nem használtuk a `nonlocal` kulcsszót. Ez azért lehetséges, mert a `nonlocal` (valamint a `global`) kulcsszó objektumhivatkozásokkal dolgozik, nem pedig az objektumhivatkozások által hivatkozott objektumokkal. Az `add_entry()` függvény második változatában azért használtuk a `level` változóval kapcsolatban a `nonlocal` kulcsszót, mert ha a `+=` művelet tényezője egy szám, akkor a művelet az objektumhivatkozás célpontjául egy új objektumot ad meg – valójában a `level = level + 1` műveletet hajtja végre a Python – így a `level` változó mostantól egy új, egész számot tartalmazó objektumra hivatkozik. Amikor azonban a `list.append()` függvényt az `indented_list` nevű listával használjuk, a függvény magát a listát módosítja, vagyis nincs szó arról, hogy az objektumhivatkozás új célpontot kapna. Így a `nonlocal` kulcsszó használata szükségtelen. (Olyan esetekben, amikor egy szótárral, listával vagy egyéb más gyűjteményes adattípusú szerkezettel dolgozunk, az elemek elhelyezése a szerkezetben – vagy a kivételük belőle – ugyanilyen okból kifolyólag a `global` kifejezés használata nélkül is lehetséges.)

Függvény- és tagfüggvénydíszítők

A díszítők olyan függvények, amelyeknek egyetlen paramétere egy másik függvény, illetve tagfüggvény, a visszatérési értékük pedig egy új, az átadott függvényt tartalmazó, de azt néhány további képességgel kiegészítő függvény, illetve tagfüggvény. Használtunk már néhány előzetesen megadott díszítőt, például a `@property` és a `@classmethod` nevűeket. Ebben a részben megtanuljuk, hogy miként hozhatunk létre saját függvénydíszítőket, a fejezet későbbi részében pedig az osztálydíszítők létrehozásával is megismerkedünk.

Osztályszintű díszítők ☞ 384

A díszítőkkel kapcsolatos első példánkban tételezzük fel, hogy sok olyan függvényünk van, ami valamilyen számítást végez, és ezek között van néhány olyan, amelynek mindig pozitív eredményt kell adnia. Megtehetnénk, hogy külön-külön végzünk bennük ellenőrzést, de a díszítő használata egyszerűbb és tisztább megoldás. Íme egy függvény, amelyet a `@positive_result` díszítővel díszítettünk. A díszítőt mindjárt létrehozzuk:

```
@positive_result
def discriminant(a, b, c):
    return (b ** 2) - (4 * a * c)
```

A díszítőnek köszönhetően minden olyan esetben, amikor az eredmény nullánál kisebb, `AssertionError` kivétel lép fel, és a program futása megszakad. A díszítőt természetesen annyi függvényen használjuk, amennyin csak jólesik. Íme a díszítő megvalósítása:

```
def positive_result(function):
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
```

```
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result
    wrapper.__name__ = function.__name__
    wrapper.__doc__ = function.__doc__
    return wrapper
```

A díszítő olyan helyi függvényt ad meg, amely az eredeti függvényt hívja meg. A fenti példában a helyi függvény neve `wrapper()`, vagyis burkoló. Ez az eredeti függvényt hívja meg, tárolja az eredményt, majd figyel, hogy az eredmény mindig pozitív legyen – ha nem lesz az, a program futása megszakad. A burkolófüggvény a benne lévő függvény által kiszámított értéket adja vissza. A burkolófüggvény létrehozását követően a nevét és a dokumentációs karakterláncát az eredeti függvénytől kölcsönözzük. Ez a hibakeresést segíti: azt szeretnénk, ha a hibaüzenetek az eredeti függvény nevét írnák ki, nem pedig a burkolóét. Az eredeti függvény helyett mostantól a burkolófüggvényt használjuk.

```
def positive_result(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result
    return wrapper
```

A `@positive_result` díszítőnek ez egy némiképp letisztult változata. A burkolót magát is beburkoljuk, mégpedig a `functools` modul `@functools.wraps` díszítőjével. Ez biztosítja, hogy a `wrapper()` függvény megkapja az eredeti függvény nevét és dokumentációs karakterláncát.

Vannak esetek, amikor hasznos lehet, ha a díszítőt paraméterezni tudjuk. Első látásra ez nem tűnik lehetségesnek, hiszen a díszítő csak egyetlen paramétert vár, ami vagy egy függvény, vagy egy tagfüggvény. De a problémának van egy elegáns megoldása. Meghívhatunk egy olyan függvényt, amelynek átadjuk a kívánt paramétereket, és amely egy olyan függvényt ad vissza, amely képes az őt követő függvény díszítésére. Például:

```
@bounded(0, 100)
def percent(amount, total):
    return (amount / total) * 100
```

A fenti esetben a `bounded()` függvénynek két paramétert adunk át, és olyan díszítőt kapunk vissza, amely aztán a `percent()` függvényt díszíti. Ebben az esetben a díszítő szerepe az, hogy garantálja, hogy a visszaadott érték mindig a 0…100 nyílt intervallumba esik. Íme a `bounded()` függvény megvalósítása:

```
def bounded(minimum, maximum):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            result = function(*args, **kwargs)
```

```
            if result < minimum:
                return minimum
            elif result > maximum:
                return maximum
            return result
        return wrapper
    return decorator
```

A függvény egy díszítőfüggvényt hoz létre, amely viszont egy burkolófüggvényt állít elő. A burkolófüggvény végrehajtja a számítást, majd visszaadja a megfelelő tartományba eső eredményt. A `decorator()` díszítőfüggvény visszatérési értéke a `wrapper()` függvény, a `bounded()` függvényé pedig a díszítőfüggvény.

Érdemes megfigyelnünk azt is, hogy minden olyan alkalommal, amikor a `bounded()` függvény belsejében létrejön egy burkolófüggvény, az éppen létrejött burkolófüggvény a `bounded()` függvénynek átadott minimum- és maximumértékeket használja.

Elérkeztünk ennek a résznek az utolsó díszítőfüggvényéhez, amely azonban az előzőeknél kissé összetettebb. Egy naplózó függvényről van szó, amely feljegyzi az általa díszített függvények nevét, paramétereit és eredményét. Például:

```
@logged
def discounted_price(price, percentage, make_integer=False):
    result = price * ((100 - percentage) / 100)
    if not (0 < result <= price):
        raise ValueError("invalid price")
    return result if not make_integer else int(round(result))
```

Ha a Pythont hibakereső módban futtatjuk (ez az alapeset), minden alkalommal, amikor meghívjuk a `discounted_price()` függvényt, az ideiglenes fájlok tárolására szolgáló helyi könyvtárban lévő `logged.log` fájl végére újabb bejegyzés kerül. Az alábbi fájlrészlet ezt szemlélteti:

```
called: discounted_price(100, 10) -> 90.0
called: discounted_price(210, 5) -> 199.5
called: discounted_price(210, 5, make_integer=True) -> 200
called: discounted_price(210, 14, True) -> 181
called: discounted_price(210, -8) <type 'ValueError'>: invalid price
```

Ha a Pythont optimalizált módban (a `-O` parancssori paramétert használva, illetve a `PYTHONOPTIMIZE` környezeti változóban a `-O` értéket tárolva) futtatjuk, nem történik naplózás. Íme a naplózást megvalósító függvény, illetve a díszítőfüggvény:

```
if __debug__:
    logger = logging.getLogger("Logger")
    logger.setLevel(logging.DEBUG)
    handler = logging.FileHandler(os.path.join(
                                  tempfile.gettempdir(), "logged.log"))
    logger.addHandler(handler)
```

```
def logged(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        log = "called: " + function.__name__ + "("
        log += ", ".join(["{0!r}".format(a) for a in args] +
                         ["{0!s}={1!r}".format(k, v)
                          for k, v in kwargs.items()])
        result = exception = None
        try:
            result = function(*args, **kwargs)
            return result
        except Exception as err:
            exception = err
        finally:
            log += (("") -> " + str(result)) if exception is None
                    else ") {0}: {1}".format(type(exception),
                                             exception))
            logger.debug(log)
            if exception is not None:
                raise exception
    return wrapper
else:
    def logged(function):
        return function
```

Hibakereső módban a `__debug__` globális változó értéke `True`. Ilyenkor a `logging` modul segítségével elindítjuk a naplózást, majd létrehozzuk a `@logged` díszítőt. A naplózó modul igen sokoldalú és rugalmas – képes fájlokba, forgatott fájlokba, email-be, hálózatra, HTTP-kiszolgálóra és egyéb helyekre naplózni. Ezúttal csak az alapvető eszközeit használtuk: létrehoztunk egy naplózó objektumot, beállítottuk a naplózás szintjét – számos szint támogatott –, és megjelöltük a kimenet tárolására szolgáló fájlt.

Szótárértelmezések 122, 130 ☞

A burkolófüggvény kódjának elején a naplóba írandó karakterláncot állítjuk elő, benne a függvény nevével és paramétereivel. Ez után megpróbálkozunk a függvény meghívásával, illetve az eredmény tárolásával. Ha kivétel lépne fel, azt is tároljuk. Minden esetben sor kerül a `finally` kódblokk végrehajtására. Itt helyezzük el a visszatérési értéket (vagy a kivételt) a naplóba kerülő karakterláncban, majd a karakterláncot a naplóba írjuk. Ha nem lépett fel kivétel, az eredményt adjuk vissza, egyéb esetben pedig újra kiváltjuk a kivételt, hogy az eredeti függvény viselkedését minél pontosabban utánozzuk.

Amikor a Python optimalizált módban fut, a `__debug__` változó értéke `False`. Ilyen esetben úgy adjuk meg a `logged()` függvényt, hogy egyszerűen a neki átadott függvényt adja vissza – vagyis eltekintve attól a kis terhelésnövekedéstől, amit a függvény létrehozásakor tapasztalhatunk, a program futása egyáltalán nem jár többletterheléssel.

Jegyezzük meg, hogy az alapkönyvtár `trace` és `profile` modulja programok futtatására és elemzésére szolgál; velük különféle nyomkövetési és összefoglaló jelentések állíthatók elő. Mindkettő közvetlenül a kódba tekint bele, azaz az előzőekben használt `logged()` függvénytől eltérően sem a `trace`, sem a `profile` modul használatakor nincs szükség a forráskód módosítására.

Függvénymagyarázatok

A függvények és a tagfüggvények magyarázatokkal (annotation) is megadhatók – a magyarázatok olyan kifejezések, amelyeket a függvény bevezetésében használhatunk. Íme az általános utasításforma:

```
def függvényNév(rész1 : kif1, rész2 : kif2, ..., részN : kifN) -> vkif:
    kódblokk
```

Minden kettősponttal elválasztott *rész* (`:` *kifX*) egy elhagyható magyarázatot jelképez, csakúgy, mint a nyilat követő „visszaadott kifejezés" rész (`->` *vkif*). Az utolsó – esetleg egyetlen – helyzeti paraméter (már amennyiben van ilyen) `*`*paraméterek* formában is megadható, magyarázattal vagy anélkül. Hasonlóan, az utolsó – esetleg egyetlen – kulcsszóparaméter (már amennyiben van ilyen) `*`*kulcsszó-paraméterek* formában is megadható, megint csak magyarázattal vagy anélkül.

Ha vannak magyarázatok, akkor a függvény `__annotations__` szótárába kerülnek. Ha nincsenek magyarázatok, akkor ez a szótár üres. A szótár kulcsai a paraméterek nevei, az értékek pedig a megfelelő kifejezések. Az utasításforma lehetővé teszi csak egy, csak néhány, vagy akár az összes paraméter magyarázattal való ellátását, illetve azt is, hogy egyáltalán ne adjunk meg magyarázatot. A visszatérési érték is ellátható magyarázattal, vagy hagyható magyarázat nélkül. A magyarázatoknak a Pythonban nincs megkülönböztetett jelentőségük. A Python egyetlen dolgot művel a magyarázatokkal, nevezetesen elhelyezi őket az `__annotations__` szótárában; minden más rajtunk áll. Íme egy magyarázatokkal ellátott, az `Util` modulból származó függvényt bemutató példa:

```
def is_unicode_punctuation(s : str) -> bool:
    for c in s:
        if unicodedata.category(c)[0] != "P":
            return False
    return True
```

Minden Unicode-karakter egy adott kategóriába tartozik, és minden kategóriát egy kétkarakteres azonosító azonosít. Minden olyan kategória, amelynek a neve *P*-vel kezdődik, írásjelkaraktereket tartalmaz.

A fenti példában a Python adattípusait használjuk a magyarázatok kifejezéseiként. Mindazonáltal a Python számára ez nem bír semmilyen megkülönböztetett jelentéssel, ahogy ennek az alábbi hívásokból ki kell derülnie:

```
Util.is_unicode_punctuation("zebr\a")          # az eredmény: False
Util.is_unicode_punctuation(s="!@#?")          # az eredmény: True
Util.is_unicode_punctuation(("!", "@"))        # az eredmény: True
```

Az első hívás egy helyzeti paramétert használ, a második pedig egy kulcsszóparamétert; mindezt csak azért, hogy megmutassuk, hogy mindkét paraméterfajta az elvárásainknak megfelelően működik. Az utolsó hívás karakterlánc helyett egy rendezett sorozatot ad át, amit a függvény elfogad, hiszen a Python semmi egyebet nem tesz, mint feljegyzi a magyarázatokat az `__annotations__` szótárába.

Ha a magyarázatokat értelmessé szeretnénk tenni, például típusellenőrzést akarunk végezni, akkor az egyik lehetőség az, hogy az ellenőrizendő függvényeket megfelelő díszítővel látjuk el. Íme egy meglehetősen alapszintű, típusellenőrzést végző díszítő:

```
def strictly_typed(function):
    annotations = function.__annotations__
    arg_spec = inspect.getfullargspec(function)

    assert "return" in annotations, "missing type for return value"
    for arg in arg_spec.args + arg_spec.kwonlyargs:
        assert arg in annotations, ("missing type for parameter '" +
                                    arg + "'")
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        for name, arg in (list(zip(arg_spec.args, args)) +
                          list(kwargs.items())):
            assert isinstance(arg, annotations[name]), (
                    "expected argument '{0}' of {1} got {2}".format(
                    name, annotations[name], type(arg)))
        result = function(*args, **kwargs)
        assert isinstance(result, annotations["return"]), (
                    "expected return of {0} got {1}".format(
                    annotations["return"], type(result)))
        return result
    return wrapper
```

Ez a díszítő akkor használható, ha a függvény minden paramétere és a visszatérési értéke is el van látva a megfelelő típust jelölő magyarázattal. Az átadott függvény a létrehozásakor ellenőrzi, hogy a függvény paramétereit és a visszatérési értéket ellátták-e a típusokat megjelölő magyarázattal, a program futtatásakor pedig azt, hogy az aktuális paraméterek megfelelnek-e a kívánalmaknak.

Az `inspect` modul sokoldalú, az objektumoknál használható vizsgálati szolgáltatást nyújt. Ezúttal csak egy kis részét használtuk az általa visszaadott paraméterleíró objektumnak: ennek segítségével nyertük ki a helyzeti és a kulcsszóparaméterek nevét – a helyzeti paraméterek esetében a pontos sorrenddel együtt. A neveket aztán a magyarázatokat tartalmazó szótárral együtt használva megbizonyosodunk arról, hogy minden paraméter és a visszatérési érték is kapott magyarázatot.

A díszítő belsejében létrehozott burkolófüggvény először bejárja a megadott helyzeti és kulcsszóparaméterekhez tartozó név-paraméter párokat. Minthogy a `zip()` függvény visszatérési értéke egy bejárható objektum, a `dictionary.items()` függvényé pedig egy szótár, nem oldható meg az, hogy azonnal egymáshoz fűzzük őket; először mindkettőt listává kell alakítanunk. Ha bármely, futáskor kapott paraméter típusa eltér a megfelelő magyarázatban tárolttól, az ellenőrzés hibát fog jelezni; egyéb esetben pedig meghívjuk a függvényt, ellenőrizzük a visszatérési érték típusát, és amennyiben megfelelőnek találjuk, visszaadjuk az értéket a hívónak. A `strictly_typed()` függvény végén a szokásos módon visszaadjuk a burkolt függvényt. Figyeljük meg, hogy az ellenőrzés csak hibakereső módban történik meg (ami a Python alapértelmezett módja, és a `-O` parancssori paraméterrel, illetve a `PYTHONOPTIMIZE` környezeti változóval állítható be).

Ha az `is_unicode_punctuation()` függvényt a `@strictly_typed` díszítővel díszítjük, és megpróbáljuk a függvényt az előző, a díszítetlen változat futtatása során használt példákkal futtatni, megfigyelhetjük a magyarázatok okozta változást:

```
is_unicode_punctuation("zebr\a")      # az eredmény: False
is_unicode_punctuation(s="!@#?")      # az eredmény: True
is_unicode_punctuation(("!", "@"))    # AssertionError kivételt vált ki
```

Ezúttal a paraméterek típusát ellenőrizzük, így az utolsó esetben `AssertionError` kivétel lép fel, hiszen a rendezett sorozat sem a karakterlánc típusnak, sem a `str` típus valamelyik alosztályának nem feleltethető meg.

A következőben a magyarázatok egy teljesen különböző használati módját fogjuk megismerni. Íme egy rövid függvény, amely a működését tekintve megegyezik a `range()` függvénnyel, a különbség mindössze annyi, hogy a mi függvényünk mindig lebegőpontos számokat ad vissza:

```
def range_of_floats(*args) -> "author=Reginald Perrin":
    return (float(x) for x in range(*args))
```

A függvény maga nem használja a magyarázatot, de könnyen elképzelhetünk egy olyan eszközt, amely a projekthez tartozó modulok betöltését követően listát készít a függvények és a függvények szerzőinek neveiről, mindezt az `__annotations__` szótár `"return"` elemét felhasználva.

A magyarázatok a Pythonban újdonságnak számítanak, és minthogy jelentésük a Python nyelvben nincs előírva, csak a képzeletünk szabhat határt a felhasználás lehetséges módjainak. A `www.python.org/dev/peps/pep-3107` webhelyen elérhető PEP 3107 (*Function Annotations*) dokumentumban további ötletekre lelhetünk a felhasználás lehetséges módjait illetőleg; illetve találhatunk itt néhány hasznos hivatkozást is.

Magasabb szintű objektumközpontú programozás

Ebben a részben a Python nyelv objektumközpontú programozást támogató oldalát ismerjük meg mélyebben, olyan módszereket tanulmányozva, amelyek a megírandó kód mennyiségét csökkentik, illetve tovább tágítják az általunk használható programozási eszközök lehetőségeit. Először azonban lássunk egy kicsi és egyszerű újdonságot. Íme a `Point` osztályt meghatározó kódrészlet eleje; az osztály viselkedése tökéletesen megegyezik a 6. fejezetben elkészített változatokéval:

```
class Point:
__slots__ = ("x", "y")
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
```

Amikor az osztályt a `__slots__` osztályszintű jellemző használata nélkül hozzuk létre, a Python a színfalak mögött minden új példány számára létrehoz egy privát, `__dict__` nevű szótárat; és ez a szótár tárolja a példány adatjellemzőit. Így válik lehetségessé jellemzők elhelyezése és eltávolítása az egyes objektumokból. (Például a `get_function()` függvényben a fejezet korábbi részében így helyeztünk el egy `cache` nevű jellemzőt.)

Ha csak olyan objektumokra van szükségünk, amelyeknek az eredeti jellemzőit kell elérnünk, és nem merül fel igény jellemzők elhelyezésére és eltávolítására, létrehozhatunk olyan osztályokat is, amelyek nem rendelkeznek `__dict__` szótárral. Ez úgy valósítható meg, hogy megadunk egy `__slots__` nevű osztályszintű jellemzőt, amelynek az értéke egy a jellemzők neveit tartalmazó rendezett sorozat. Minden, ebbe az osztályba tartozó objektum rendelkezni fog a megadott nevű jellemzőkkel, de nem lesz `__dict__` nevű szótára; ekképpen az ilyen osztályokban nem lehetséges új jellemzők elhelyezése, sem a már meglévők eltávolítása. Az ilyen objektumok a hagyományos objektumokhoz képest kevesebb memóriát használnak, bár ez feltehetőleg nem vezet jelentésbeli különbségekhez, hacsak nem használunk igen nagy számú ilyen objektumot.

A jellemzők elérésének szabályozása

Előfordulnak olyan helyzetek, amikor kényelmes lehet egy olyan osztállyal rendelkezni, amelyben a jellemzők értékét nem tároljuk, hanem szükség esetén kiszámítjuk. Íme egy ilyen osztály teljes megvalósítása:

```
class Ord:
def __getattr__(self, char):
        return ord(char)
```

Most, hogy létrehoztuk az `Ord` osztályt, példányosíthatjuk is: `ord = Ord()`. Ezzel létrehoztuk a beépített `ord()` függvény működését megvalósító objektumot, amely minden olyan karakter esetében működik, amely használható azonosítóként.

Például az `ord.a` visszatérési értéke 97, az `ord.Z`-é 90, az `ord.å`-é pedig 229. (De az `ord.!` és társai nyelvtanilag hibásak.)

Gondoljunk arra, hogy ha az `Ord` osztályt az IDLE fejlesztői környezetben adtuk volna meg, gondjaink támadtak volna az `ord = Ord()` utasítás kiadásakor. Ennek oka abban keresendő, hogy az objektumpéldány neve megegyezett volna a beépített `ord()` függvény nevével, amelyet az `Ord` osztály is használ. Így az `ord()` hívás valójában az `ord` nevű objektumpéldányt hívta volna meg, és `TypeError` kivételt eredményezett volna. A probléma nem jelentkezne, ha az `Ord` osztály egy betöltött modulban kapna helyet, ugyanis a fejlesztői környezetben közvetlenül megadott `ord` objektum, illetve a beépített `ord()` függvény – amelyet az `Ord` osztály is használ – két különböző modulban kapna helyet, így nem írnák egymást felül. Ha valaha is tényleg szükségünk van arra, hogy egy osztályt közvetlenül a fejlesztői környezetben adjunk meg, illetve egy beépített objektum nevét használjuk saját objektumunk nevéül, akkor ezt úgy tudjuk megtenni, hogy figyelünk arra, hogy az osztály a beépített objektumot hívja – esetünkben be kell töltenünk a `builtins` modult, amely egyértelművé teszi a beépített függvények hívását, mégpedig úgy, hogy a sima `ord()` hívás helyett a `builtins.ord()` hívást hajtja végre.

Íme egy újabb aprócska, ámde teljes osztály. Ez az osztály „állandók" létrehozását teszi lehetővé. Az osztály megkerülésével nem jelenthet gondot az értékek megváltoztatása, de a legkönnyebben elkövethető hibák kivédésénél jó szolgálatot tehet.

```
class Const:
    def __setattr__(self, name, value):
        if name in self.__dict__:
            raise ValueError("cannot change a const attribute")
        self.__dict__[name] = value

    def __delattr__(self, name):
        if name in self.__dict__:
            raise ValueError("cannot delete a const attribute")
        raise AttributeError("'{0}' object has no attribute '{1}'"
                             .format(self.__class__.__name__, name))
```

Az osztály segítségével állandó objektumokat hozhatunk létre, például így: `const = Const()`; és lehetőség nyílik a jellemzők beállítására is, például: `const.limit = 591`. Ha egyszer beállítottuk egy jellemző értékét, annyiszor olvassuk ki, ahányszor úri kedvünk diktálja, ugyanakkor minden, az érték megváltoztatására, illetve törlésére irányuló kísérlet eredménye `ValueError` kivétel lesz. Nem valósítottuk meg újra a `__getattr__()` tagfüggvényt, hiszen az alaposztály `object.__getattr__()` tagfüggvénye épp azt teszi, amire szükségünk van – visszaadja az átadott jellemző értékét, vagy a megfelelő jellemző hiányában `AttributeError` kivételt vált ki. A `__delattr__()` tagfüggvény megvalósításakor a `__getattr__()` tagfüggvény nem létező jellemzők esetén adott hibaüzenetét utánozzuk. Ezt úgy tudjuk megtenni, hogy kinyerjük annak az osztálynak a nevét, amelyikben éppen

tartózkodunk, csakúgy, mint a nem létező jellemző nevét. Az osztály működik, hiszen az osztály `__dict__` nevű szótárát használjuk, és ezt használja az alaposztály `__getattr__()`, `__setattr__()` és `__delattr__()` tagfüggvénye is. Mindazonáltal megjegyeznénk, hogy mi csak az alaposztály `__getattr__()` tagfüggvényét használtuk. A jellemzők elérését megvalósító különleges szerepű tagfüggvények listája a 8.2. táblázatban található.

8.2. táblázat *A jellemzők elérésére szolgáló különleges szerepű tagfüggvények*

Tagfüggvény	Használat	Leírás
`__delattr__(self, name)`	`del x.n`	Törli az `x` objektum `n` jellemzőjét.
`__dir__(self)`	`dir(x)`	Az `x` objektum jellemzőinek neveiből álló listával tér vissza.
`__getattr__(self, name)`	`v = x.n`	Az `x` objektum `n` jellemzőjének értékét adja vissza, ha a közvetlen keresés sikertelen.
`__getattribute__(self, name)`	`v = x.n`	Az `x` objektum `n` jellemzőjének értékét adja vissza; lásd a szöveget.
`__setattr__(self, name, value)`	`x.n = v`	Az `x` objektum `n` jellemzőjének `v`-t adja értékül.

Az állandók megvalósításának másik módja is kínálkozik: nevesített rendezett sorozatokat használhatunk. Íme néhány példa:

```
Const = collections.namedtuple("_", "min max")(191, 591)
Const.min, Const.max                                  # az eredmény: (191, 591)
Offset = collections.namedtuple("_", "id name description")(*range(3))
Offset.id, Offset.name, Offset.description   # az eredmény: (0, 1, 2)
```

A nevesített rendezett sorozatokat mindkét esetben csak egy egyszer használatos névvel jelöltük, mivel egyszerre csak egy nevesített rendezett sorozatra van szükségünk, nem pedig a `tuple` típus egy alosztályára, amelyből nevesített `tuple`-példányokat hozhatunk létre. Bár a Python nem támogatja a felsoroló (enum) adattípust, hasonló hatást érhetünk el nevesített rendezett sorozatok használatával, mint ahogy azt fent is tettük.

Image 262

A jellemzők elérését megvalósító különleges szerepű tagfüggvényekre vetett utolsó pillantásunkkal visszatérünk egy olyan példához, amellyel először a 6. fejezetben találkoztunk. Abban a fejezetben hoztuk létre az `Image` osztályt, amelynek a szélessége, magassága és háttérszíne a példányosítás során később meg nem változtatható értéket kap (bár, ha betöltünk egy képet, az értékek megváltoznak). Az értékek elérését csak olvasható tulajdonságokkal valósítottuk meg. Például így:

```
@property
def width(self):
    return self.__width
```

A kódolás elvégzése egyszerű, ám ha sok csak olvasható tulajdonsággal dolgozunk, könnyen válhat unalmassá. Íme egy eltérő megoldás, amely az `Image` csak olvasható tulajdonságait egyetlen tagfüggvényben kezeli:

```
def __getattr__(self, name):
    if name == "colors":
        return set(self.__colors)
    classname = self.__class__.__name__
    if name in frozenset({"background", "width", "height"}):
        return self.__dict__["_{0}__{1}".format(classname, name)]
    raise AttributeError("'{0}' object has no attribute '{1}'"
                         .format(classname, name))
```

Ha megpróbáljuk elérni egy objektum jellemzőjét, de a jellemzőt nem találjuk, a Python a `__getattr__()` tagfüggvényt hívja meg (amennyiben a tagfüggvény létezik, és nem végeztük el a `__getattribute__()` tagfüggvény újbóli megvalósítását), mégpedig a jellemzőt adva át paraméterként. A `__getattr__()` tagfüggvény megvalósításaival szemben elvárás, hogy amennyiben az átadott jellemzőt nem képesek kezelni, `AttributeError` kivételt kell kiváltaniuk.

Például ha az `image.colors` utasítást tekintjük, a Python keresni fogja a `colors` jellemzőt, és miután nem találja meg, kiadja az `Image.__getattr__(image,"colors")` hívást. Esetünkben a `__getattr__()` tagfüggvény kezeli a `colors` jellemzőnevet, és a gép által használt színeket tartalmazó halmaz másolatát adja vissza.

A többi jellemző nem módosítható, így aztán biztonságos őket közvetlenül visszaadni a hívónak. Írhattunk volna minden további jellemzőhöz önálló `elif`-ágat, például így:

```
elif name == "background":
    return self.__background
```

Ehelyett egy tömörebb megközelítés mellett döntöttünk. Minthogy tudjuk, hogy valójában minden olyan jellemző, amely nem bír különleges szereppel az objektumban, a `self.__dict__` szótárba kerül, úgy döntöttünk, hogy közvetlenül fogunk hozzájuk férni. A privát jellemzők esetében (azokról van szó, amelyeknek a neve két aláhúzással kezdődik) a nevek úgy alakulnak át, hogy *`_osztályNév__jellemzőNév`* alakot kapjanak. Amikor az objektum privát szótárából előhívjuk a jellemző értékét, erre figyelnünk kell.

Ahhoz tehát, hogy a privát jellemzőket meg tudjuk keresni, illetve elő tudjuk állítani a szokásos `AttributeError` hibaüzenet szövegét, névátalakítás szükséges. Ha ezt meg szeretnénk oldani, tudnunk kell annak az osztálynak a nevét, amelyben tartózkodunk. (Ez esetleg nem az `Image` osztály lesz, ugyanis a kérdéses objektum esetleg az `Image` osztály egy alosztályának a példánya.) Minden objektum rendelkezik egy `__class__` nevű különleges szerepű jellemzővel, vagyis a tagfüggvények belsejében mindig használhatjuk a `self.__class__` kifejezést, a nemkívánatos önhívás pedig a `__getattr__()` függvény használatával kerülhető el.

Figyeljünk fel arra a parányi különbségre, hogy a `__getattr__()` és a `self.__class__` használatakor a jellemzőt az objektumpéldány osztályában érjük el (ez az osztály lehet alosztály is), ugyanakkor a jellemző közvetlen elérésekor azt az osztályt használjuk, amelyikben a jellemző megadása történt.

Eddig nem tárgyaltuk a `__getattribute__()` különleges szerepű tagfüggvényt. Míg a `__getattr__()` tagfüggvényt olyankor hívjuk meg, amikor egy (különleges szereppel nem bíró) jellemzőt keresünk, a `__getattribute__()` jellemzőt minden olyan alkalommal elsőként hívjuk, amikor hozzá kell férnünk egy osztályhoz. Bár vannak esetek, amikor hasznos vagy akár létfontosságú lehet a `__getattribute__()` tagfüggvény hívása, újbóli megvalósítása néhány buktatót rejt magában. Az ilyen megvalósítások esetében figyelnünk kell arra, hogy nehogy saját magát hívja meg a tagfüggvény – ilyen esetekben gyakori a `super().__getattribute__()` és az `object.__getattribute__()` hívás használata. És minthogy a `__getattribute__()` tagfüggvényt minden elem elérésekor meghívjuk, az újbóli megvalósítás könnyen vezethet teljesítménycsökkenéshez közvetlen elérés esetén. A könyvben bemutatott egyetlen osztály sem valósítja meg újra a tagfüggvényt.

Funktorok

Figyelem!

A Python nyelvben a *függvényobjektumok* olyan objektumhivatkozások, amelyek egy hívható objektumra (függvény, lambdafüggvény, tagfüggvény) mutatnak. A meghatározásba belefférnek az osztályok is, hiszen egy osztályra mutató objektumhivatkozás is hívható objektumra mutat: ha az osztályokat hívjuk, a visszatérési érték egy az adott osztályba tartozó objektum lesz. Ilyen hívás például az `x = int(5)`. A számítástudomány a *functor* névvel az olyan objektumokat illeti, amelyek függvényként hívhatók. A Pythonban tehát a funktor nem egyéb, mint egy újabb függvényobjektumféleség. Minden olyan osztály funktor, amely rendelkezik a `__call__()` nevű különleges szerepű tagfüggvénnyel. A funktorok legfontosabb előnye, hogy képesek néhány, az állapotukra vonatkozó információ kezelésére. Például létrehozhatunk olyan funktort, amely karakterláncok végéről távolít el írásjeleket. Az alábbi példa szemlélteti a létrehozását és a használatát:

```
strip_punctuation = Strip(",;:.!?")
strip_punctuation("Land ahoy!")          # az eredmény: 'Land ahoy'
```

A fenti példában a `Strip` funktort példányosítjuk, majd az `",;:.!?"` kezdőértéket adjuk neki. Amikor a példányt meghívjuk, a neki átadott karakterláncot adja vissza, de előbb eltávolítja a végéről az írásjeleket. Íme a `Strip` osztály teljes megvalósítása:

```
class Strip:
    def __init__(self, characters):
        self.characters = characters
    def __call__(self, string):
        return string.strip(self.characters)
```

Ugyanez a hatás elérhető lett volna egy sima függvénnyel vagy egy lambdafüggvénnyel is, de ha kicsivel több állapotinformáció tárolására van szükség, illetve bonyolultabb az elvégzendő feladat, gyakran a funktorok használata jelenti a helyes megoldást.

Az, hogy a funktorok egy osztály használatával képesek állapotok rögzítésére, igen nagy sokoldalúságot és érdekes lehetőségeket rejt magában, de előfordul, hogy nincs szükségünk ilyen sok mindenre. Az állapot rögzítésének másik módszerét a *záradékok* használata jelenti. A záradék olyan függvény, illetve tagfüggvény, amely valamilyen külső állapot rögzítésére képes. Például:

```
def make_strip_function(characters):
    def strip_function(string):
        return string.strip(characters)
    return strip_function

strip_punctuation = make_strip_function(",;:.!?")
strip_punctuation("Land ahoy!")          # az eredmény: 'Land ahoy'
```

A `make_strip_function()` függvény egyetlen paraméterét a levágandó karakterek jelentik, a visszatérési értéke pedig egy függvény, a `strip_function()` nevű. Ez utóbbi paramétere egy karakterlánc, feladata pedig a záradék létrehozásakor megadott karakterek törlése. Vagyis, hasonlóan ahhoz, ahogy a `Strip` osztályból annyi példányt hozunk létre, amennyi nekünk tetszik, méghozzá mindegyiket a saját levágandó karaktereivel, a karakterek eltávolítását végző függvényből is sokat hozhatunk létre, és mindegyik saját levágandó karaktereket kaphat.

A funktorok klasszikus felhasználási területét jelentik a rendező algoritmusok kulcsfüggvényei. Íme a `SortKey.py` fájlban található általános célú `SortKey` funktorosztály:

```
class SortKey:

    def __init__(self, *attribute_names):
        self.attribute_names = attribute_names

    def __call__(self, instance):
        values = []
        for attribute_name in self.attribute_names:
            values.append(getattr(instance, attribute_name))
        return values
```

Amikor létrejön egy a `SortKey` osztályba tartozó objektumpéldány, egy rendezett sorozatban tárolja azokat a jellemzőneveket, amelyeket az előkészítés során kapott meg. Amikor az objektum hívása megtörténik, létrehoz egy listát a neki átadott példány jellemzőinek értékeiből, abban a sorrendben, amit a `SortKey` osztály előkészítésekor megadtunk. Például képzeljük el, hogy van egy `Person` nevű osztályunk:

```
class Person:
    def __init__(self, forename, surname, email):
        self.forename = forename
        self.surname = surname
        self.email = email
```

Tegyük fel, hogy a `people` listában `Person` osztályú objektumok vannak. A listát vezetéknevek alapján a következőképpen rendezhetjük: `people.sort(key=SortKey("surname"))`. Ha sok ember van, biztosan lesznek egyező vezetéknevűek, így a vezetékneveken belül keresztnév szerint is rendezhetünk. A módszer a következő: `people.sort(key=SortKey("surname", "forename"))`. Ha sok, teljesen egyező nevű ember van, az e-mail cím jelenthet újabb rendezési szempontot, és természetesen megoldható az is, hogy a rendezés először történjen keresztnév, aztán vezetéknév alapján: ehhez a `SortKey` funktornak átadott jellemzők neveinek sorrendjét kell megváltoztatnunk.

Ugyanez az eredmény más módon is elérhető, anélkül, hogy akár egyetlen funktort is létre kellene hoznunk. A megoldást az `operator` modul `operator.attrgetter()` függvényének használata jelenti. Ekkor a vezetéknév alapján történő rendezés a következő módon történne: `people.sort(key=operator.attrgetter("surname"))`. Ehhez hasonlóan, a vezeték- és keresztnév alapján történő rendezés a következőképp történne: `people.sort(key=operator.attrgetter("surname", "forename"))`. Az `operator.attrgetter()` függvény visszatérési értéke egy függvény (egy záradék), amely egy objektumra meghívva az objektumnak a záradék létrehozásakor megadott jellemzőit adja vissza.

A funktorokat alighanem ritkábban használják a Pythonban, mint az egyéb olyan nyelvekben, amelyek támogatják a használatukat. Ennek oka feltehetőleg abban keresendő, hogy a Python más, hasonló feladatok ellátására alkalmas eszközöket is rendelkezésre bocsát a funktorok mellett – például a záradékokat, illetve az elemek és a jellemzők kiolvasását megvalósító függvényeket.

Környezetkezelők

A környezetkezelők a kód egyszerűsítését azáltal teszik lehetővé, hogy bizonyos műveleteket adott kódblokkok végrehajtása előtt és után mindig elvégeznek. Ez a viselkedés arra vezethető vissza, hogy a környezetkezelők megadnak két különleges szerepű tagfüggvényt – `__enter__()` és `__exit__()` –, amelyeket a Python a `with` utasítás hatókörében különlegesen kezel. Amikor a környezetkezelő a `with` utasításon belül jön létre, automatikusan meghívjuk az `__enter__()` tagfüggvényét, amikor pedig kiérünk a környezetkezelő hatóköréből, az `__exit__()` tagfüggvény automatikus hívására kerül sor.

Megírhatjuk a saját környezetkezelőinket, vagy használhatunk előre megadottakat – ahogy a későbbiekből kiderül, a beépített `open()` függvénytől visszakapott fájlobjektumok is környezetkezelők. A környezetkezelők az alábbi utasításformával használhatók:

```
with kifejezés as változó:
    kódblokk
```

A *kifejezés* vagy eleve egy környezetkezelő, vagy a kiértékelésének kell környezetkezelőt eredményeznie. Ha az elhagyható `as` *`változó`* részt megadjuk, a változó értéke a környezetkezelő `__enter__()` tagfüggvényétől visszakapott objektumra fog hivatkozni. A visszakapott objektum gyakran maga a környezetkezelő. Mivel a környezetkezelő biztosan végrehajtja a kilépési kódját (ebben a kivételek sem gátolják), a környezetkezelők sok helyzetben használhatók a `finally` blokkok megszüntetésére.

A Python típusai közül néhány tulajdonképpen környezetkezelő – például az `open()` függvénytől visszakapott valamennyi fájlobjektum –, így szükségtelenné válnak a fájlkezelés során használt `finally` blokkok. Az alábbi két egyenértékű kódrészlet is ezt érzékelteti (feltételezzük, hogy a `process()` egy más helyen megadott függvény):

```
fh = None
try:
fh = open(filename)
for line in fh:
process(line)
except EnvironmentError as err:
print(err)
finally:
if fh is not None:
fh.close()
```

```
try:
with open(filename) as fh:
for line in fh:
process(line)
except EnvironmentError as err:
print(err)
```

A fájlobjektumok olyan környezetkezelők, amelyeknek a kilépéskor végrehajtandó kódja minden esetben lezárja a megnyitott állományokat. A kilépési kód a kivételek esetleges létrejöttétől függetlenül végrehajtódik, bár az utóbbi esetben a kivételről tudomást szerzünk. Így biztosítható, hogy a fájl le lesz zárva, és módunkban áll a felmerülő hibák kezelése is – ezúttal egy hibaüzenetet írunk a felhasználó képernyőjére.

Az az igazság, hogy a környezetkezelőkkel nem kell feltétlenül továbbadatnunk a kivételeket, de ha így járunk el, akkor lényegében elrejtjük a kivételeket, ami szinte bizonyosan kódolási hiba. A beépített környezetkezelők, csakúgy, mint az alapkönyvtárban lévők, továbbadják a kivételeket.

Vannak esetek, amikor egyszerre több környezetkezelőt is használnunk kell. Például:

```
try:
    with open(source) as fin:
        with open(target, "w") as fout:
            for line in fin:
                fout.write(process(line))
except EnvironmentError as err:
    print(err)
```

A fenti példában a forrásfájl sorait olvassuk be, amelyeket feldolgozás után a célfájlba írunk.

Az egymásba ágyazott `with` utasítások használata gyorsan vezethet rengeteg behúzáshoz. Szerencsére az alapkönyvtár `contextlib` modulja további támogatással segíti a környezetkezelők használatát, többek között a `contextlib.nested()` függvény formájában, amely lehetővé teszi, hogy kettő vagy akár több környezetkezelőt alkalmazzunk ugyanazon a `with` utasításon belül – így elkerülhető az egymásba ágyazásuk. Íme a fent bemutatott kód átírt változata – elhagyva azokat a sorokat, amelyek nem változtak az előző formához képest.

```
try:
    with contextlib.nested(open(source), open(target, "w")) as (
            fin, fout):
        for line in fin:
```

Nem csak a fájlobjektumok környezetkezelők. Például számos, a többszálúsággal kapcsolatos és a zárolások során használt osztály is környezetkezelő. Környezetkezelőket használhatunk a `decimal.Decimal` típusú számok esetében – ez különösen akkor hasznos, ha adott beállítások mellett (mondjuk egy adott pontossággal) szeretnénk számításokat végezni.

Többszálúság ☞ 420

Ha saját környezetkezelőt szeretnénk, egy olyan osztályt kell létrehoznunk, amely két tagfüggvényt valósít meg – az `__enter__()` és az `__exit__()` tagfüggvényről van szó. Minden olyan alkalommal, amikor egy az osztályból példányosított objektumot egy `with` utasításban használunk, az `__enter__()` tagfüggvény hívására kerül sor, és ennek a visszatérési értéke kerül az `as` kulcsszó után megadott változóba (ha nem adtunk meg változót, akkor a visszatérési érték elvész). Amikor a vezérlés kikerül a `with` hatóköréből, az `__exit__()` tagfüggvény hívása történik meg – ha közben kivétel lép fel, akkor ennek az adatai lesznek a tagfüggvény paraméterei.

Tegyük fel, hogy műveletek sorát szeretnénk végrehajtani egy adott listán, mégpedig tranzakciószerűen – azaz azt szeretnénk, hogy vagy minden művelet legyen végrehajtva, vagy egyik sem, és ekképp az eredménylista legyen mindig ismert állapotú. Például ha egy egész számokból álló listánk van, amelyhez egy újabb egész számot szeretnénk adni, egyet törölni, néhányat pedig megváltoztatni, és mindezt egyetlen műveletként, akkor az alábbihoz hasonló kódot írhatunk:

```
try:
    with AtomicList(items) as atomic:
        atomic.append(58289)
        del atomic[3]
        atomic[8] = 81738
        atomic[index] = 38172
except (AttributeError, IndexError, ValueError) as err:
    print("no changes applied:", err)
```

Ha nem lép fel kivétel, akkor minden műveletet elvégzünk az eredeti listán – az `items` nevűn –, amennyiben azonban kivétel keletkezik, semmilyen módosítás nem történik. Íme az `AtomicList` környezetkezelő kódja:

```
class AtomicList:

    def __init__(self, alist, shallow_copy=True):
        self.original = alist
        self.shallow_copy = shallow_copy

    def __enter__(self):
        self.modified = (self.original[:] if self.shallow_copy
                         else copy.deepcopy(self.original))
        return self.modified

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is None:
            self.original[:] = self.modified
```

Sekély és mély másolás 143 ☞

Amikor egy `AtomicList` objektum létrejön, megtartjuk az eredeti listára mutató hivatkozást, és átgondoljuk, hogy sekély másolást kell-e használnunk. (A sekély másolás megfelelő módszer számokból vagy karakterláncokból álló lista esetén; de olyan listáknál nem, amelyek további listákat vagy egyéb gyűjteményes típusú szerkezeteket tartalmaznak.)

Később, amikor az `AtomicList` környezetkezelő objektumot a `with` utasítás belsejében használjuk, az objektum `__enter__()` tagfüggvényét hívjuk meg. Ezen a ponton az eredeti listát lemásoljuk, és a másolatot adjuk vissza, így minden változtatás a másolaton fog megtörténni.

Amikor elérjük a `with` hatókörének végét, az `__exit__()` tagfüggvény hívódik meg. Ha nem lépett fel kivétel, az `exc_type` („exception type", kivétel típusa) változó értéke `None` lesz, és tudjuk, hogy biztonságosan cserélhetjük le az eredeti lista elemeit a módosított listából származókra. (Nem adhatjuk ki a `self.original = self.modified` utasítást, ugyanis az csak az eredeti objektum objektumhivatkozását cseréli le, és az eredeti listán semmit nem változtatna.) Ha azonban kivétel történt, nem nyúlunk az eredeti listához, a módosítottat pedig elvetjük.

Az `__exit__()` tagfüggvény visszatérési értéke arra jó, hogy jelezze, hogy történt-e továbbadandó kivétel. A `True` érték azt jelenti, hogy minden kivételt kezeltünk, továbbadásra nincs szükség. Normális esetben mindig `False` értéket – vagy olyasmit, ami logikai értelemben `False` értékké értékelődik ki – adunk vissza, lehetővé téve, hogy az esetlegesen létrejött kivételek továbbadódjanak. Ha nem adunk meg konkrét visszatérési értéket, az `__exit__()` tagfüggvényünk `None` objektumot ad vissza, amelynek a kiértékelésekor `False` értéket kapunk, és ez – helyesen – ahhoz vezet, hogy a kivételek továbbadódnak.

A 10. fejezetben arra használjuk a saját környezetkezelőket, hogy megbizonyosodjunk róla, hogy a csatolókapcsolatokat, illetve a gzip tömörítésű fájlokat lezártuk. A 9. fejezetben a többszálúsággal kapcsolatos modulok pár környezetkezelőjét azért vesszük igénybe, hogy megbizonyosodjunk róla, hogy a kölcsönösen kizáró zárolások fel vannak-e oldva. Az Olvasó a fejezet gyakorlataiban lehetőséget kap egy az előzőnél általánosabb, tranzakciókkal dolgozó környezetkezelő megírására.

Leírók

A leírók (descriptor) olyan osztályok, amelyek a más osztályok jellemzőihez történő hozzáférést szabályozzák. Minden olyan osztály, amely a `__get__()`, a `__set__()` és a `__delete__()` különleges szerepű tagfüggvények közül legalább egyet megvalósít, leírónak tekinthető és akként használható.

A beépített `property()` és `classmethod()` függvények megvalósítása leírók használatával történt. A leírók megértésének kulcsa abban rejlik, hogy bár a leíró osztályból az objektumokat az osztályon belül osztályjellemzőként példányosítjuk, a Python a leírókat az osztályból létrehozott példányokon keresztül éri el.

Hogy világosabb legyen, tegyük fel, hogy van egy osztályunk, amelynek a példányai bizonyos karakterláncokat tárolnak. A megszokott módon szeretnénk hozzáférni a karakterláncokhoz, például tulajdonságként, ugyanakkor szeretnénk egy az XML-nek megfelelően levédett változatot is kapni a karakterláncokból, amikor csak kedvünk tartja. Az egyik egyszerű megoldás az lenne, hogy minden alkalommal, amikor értékül adunk egy karakterláncot, azonnal létrehozzuk az XML-ben használatos másolatát is. Amennyiben azonban több ezer karakterlánccal dolgozunk, és ezek közül csak néhánynak szeretnénk az XML-ben használatos változatát beolvasni, azokat is csak egyszer-egyszer, rengeteg felesleges számítást végeznénk, és rengeteg memóriát pocsékolnánk el. Így inkább létrehozunk egy olyan leírót, amely az XML-nek megfelelő, levédett változatokat csak szükség esetén állítja elő, és nem tároljuk azokat. Elsőként azt az osztályt írjuk meg, amelyik a leírót használni fogja:

```
class Product:
    __slots__ = ("__name", "__description", "__price")
    name_as_xml = XmlShadow("name")
    description_as_xml = XmlShadow("description")
    def __init__(self, name, description, price):
        self.__name = name
        self.description = description
        self.price = price
```

A fentiekben csak a tulajdonságokat megvalósító kódot nem közöltük. A `name` tulajdonság csak olvasható, a `description` és a `price` pedig írható-olvasható tulajdonság; mindegyiket a szokásos módszerrel állítottuk be. (A kód egésze az `XmlShadow.py` fájlban található.)

A __slots__ változóval biztosítottuk, hogy az osztály ne rendelkezzen __dict__ szótárral, és csak a három megadott privát jellemző tárolására legyen képes; mindez nem áll kapcsolatban a leírók használatával, és nem is szükséges hozzá. A name_as_xml és a description_as_xml osztályszintű jellemzőket úgy állítottuk be, hogy az XmlShadow leíróosztály példányait kapják értékül. Bár egyetlen Product objektum sem rendelkezik sem name_as_xml, sem description_as_xml nevű jellemzővel, a leírónak köszönhetően a kód az alábbi formát kaphatja (ezúttal a modul dokumentációs tesztjeiből idézve):

```
>>> product = Product("Chisel <3cm>", "Chisel & cap", 45.25)
>>> product.name, product.name_as_xml, product.description_as_xml
('Chisel <3cm>', 'Chisel &lt;3cm&gt;', 'Chisel &amp; cap')
```

Azért járhatunk el így, mert amikor például a name_as_xml tulajdonság elérésével próbálkozunk, a Python rájön, hogy a Product osztálynak van ilyen nevű leírója, és így a leíróval olvassa ki a jellemző értékét. Íme az XmlShadow leíróosztály teljes kódja:

```
class XmlShadow:

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name

    def __get__(self, instance, owner=None):
        return xml.sax.saxutils.escape(
                                getattr(instance, self.attribute_name))
```

Amikor a name_as_xml és a description_as_xml objektumok létrejönnek, a Product osztály megfelelő jellemzőjét az XmlShadow osztály előkészítőjének adjuk át, így aztán a leíró tudja, hogy melyik jellemzővel kell dolgoznia. Később, amikor a name_as_xml és a description_as_xml jellemzőket keressük, a Python a leíró __get__() tagfüggvényét hívja meg. A self paraméter a leíróosztály egy példánya, az instance a Product-é (a Product "self"-je), az owner paraméter pedig a leírót birtokló osztályt jelzi (esetünkben a Product-ról van szó). A getattr() függvényt használjuk arra, hogy a Product-ból a megfelelő jellemzőt (esetünkben a megfelelő tulajdonságot) kiolvassuk, és az XML-nek megfelelően levédett változatát adjuk vissza.

Ha a használat módja az lenne, hogy a Product objektumoknak csak egy kis hányadából olvassuk ki az XML-nek megfelelően levédett karakterláncokat, de ezek a karakterláncok gyakran hosszúak, és sokszor olvassuk ki ugyanazokat, érdemes lenne gyorsítótárat használni. Például így:

```
def __init__(self, attribute_name):
    self.attribute_name = attribute_name
    self.cache = {}

def __get__(self, instance, owner=None):
    xml_text = self.cache.get(id(instance))
    if xml_text is not None:
        return xml_text
```

```
        return self.cache.setdefault(id(instance),
                xml.sax.saxutils.escape(
                                getattr(instance, self.attribute_name)))
```

A szótár kulcsaként nem magukat az objektumpéldányokat, hanem csak az egyedi azonosítójukat tároljuk, mivel a szótár kulcsainak kivonatolhatóknak kell lenniük – az azonosítók pedig ilyenek –, ugyanezt a követelményt azonban nem szeretnénk a `CachedXmlShadow` leírót használó osztályokkal szemben is megfogalmazni. A kulcsra azért van szükség, mert a leírók nem példányonként, hanem osztályokként jönnek létre. (A `dict.setdefault()` tagfüggvény számunkra igen kényelmesen a neki átadott kulcs értékét adja vissza, amennyiben pedig ilyen kulcsú elem nem létezik, új elemet hoz létre az átadott kulccsal és értékkel, és ilyenkor az értéket kapjuk vissza.)

Most, hogy láttuk, hogy a leírók miként képesek olyan adatok létrehozására, amelyeket nem feltétlenül tárolnak, megvizsgálunk egy olyan leírót, amelyet egy objektumpéldány jellemzőiben lévő adatok tárolására használhatunk – magának az objektumpéldánynak semmit sem kell tárolnia. Ebben a példában ismét egy szótárat használunk, de valósághűbb környezetben az adatot fájlban vagy adatbázisban tárolnánk. Az alábbiakban ismertetjük a `Point` osztály módosított változatát, amely az `ExternalStorage.py` fájlban található leírót használja:

```
class Point:
    __slots__ = ()
    x = ExternalStorage("x")
    y = ExternalStorage("y")
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Azzal, hogy a `__slots__` változó értékéül egy üres rendezett sorozatot adunk meg, biztosítjuk, hogy az osztály egyáltalán ne legyen képes adatjellemzők tárolására. Amikor a `self.x` kap értéket, a Python rájön, hogy van „x" nevű leíró, és a leíró `__set__()` tagfüggvényét használja. Az osztály további részét nem közöljük, mert megegyezik a 6. fejezetben ismertetett eredeti `Point` osztályban lévőkkel. Íme a teljes `ExternalStorage` leíróosztály:

```
class ExternalStorage:
    __slots__ = ("attribute_name",)
    __storage = {}
    def __init__(self, attribute_name):
        self.attribute_name = attribute_name
    def __set__(self, instance, value):
        self.__storage[id(instance), self.attribute_name] = value
    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.__storage[id(instance), self.attribute_name]
```

Minden `ExternalStorage` objektum egyetlen adatjellemzővel, az `attribute_name` nevűvel rendelkezik. A jellemző az objektumpéldányt birtokló osztály adatjellemzőjét tárolja. Minden alkalommal, amikor egy jellemző értéket kap, az értéket a `__storage` nevű privát, osztályszintű szótárban tároljuk. Amikor egy adat kiolvasására kerül sor, azt is a `__storage` nevű szótárból olvassuk ki.

A leírók minden tagfüggvényéhez hasonlóan a `self` is a leíróosztály egy példánya, az `instance` pedig azon objektum `self` tagfüggvényének feleltethető meg, amely a leírót tartalmazza, azaz ebben a helyzetben a `self` egy `ExternalStorage` objektum, az `instance` pedig egy `Point` objektum.

Bár a `__storage` osztályszintű jellemző, a `self.__storage` utasítással érhetjük el (pontosan úgy, mint ahogy a tagfüggvények hívását is a `self.`*`tagfüggvény`*`()` utasításformával oldhatjuk meg), hiszen a Python egy objektumpéldány jellemzőjeként fogja keresni, és miután nem talált semmi, a keresést az osztályszintű jellemzők között folytatja. Az ilyen megközelítés egyetlen elméleti hátránya abban rejlik, hogy amennyiben egyforma nevű osztályszintű és objektumpéldány-szintű jellemzőnk van, az egyik el fogja takarni a másikat. (Ha ebbe a problémába a valóságban is beleütközünk, az osztályszintű jellemzőre az osztály nevével hivatkozhatunk, vagyis így: `ExternalStorage.__storage`.) Bár az osztály nevének bedrótozása az alosztályok kialakítása során általában rossz ötlet, a privát jellemzők esetében nem igazán van jelentősége: a Python így is, úgy is végrehajt bizonyos névváltoztatásokat.

A `__get__()` különleges szerepű tagfüggvény megvalósítása azért valamennyivel kifinomultabb az előzőeknél, ugyanis biztosítjuk annak lehetőségét, hogy maga az `ExternalStorage` objektum elérhető legyen. Például ha van egy `p = Point(3, 4)` objektumunk, az *x* koordinátát a `p.x` formában érhetjük el, és a `Point.x` utasítással férhetünk hozzá az `x` értékeket tároló `ExternalStorage` objektumhoz.

A leírók tárgyalását azzal tesszük teljessé, hogy létrehozzuk azt a `Property` leíróosztályt, amely a beépített `property()` függvény viselkedését utánozza, legalábbis ami a beállítófüggvényeket és a kiolvasófüggvényeket illeti. A kód a `Property.py` fájlban található. Íme a leíróosztályt használó `NameAndExtension` osztály teljes kódja:

```
class NameAndExtension:
    def __init__(self, name, extension):
        self.__name = name
        self.extension = extension
    @Property                # Az egyéni Property leírót használja
    def name(self):
        return self.__name
    @Property                # Az egyéni Property leírót használja
    def extension(self):
        return self.__extension
```

```
    @extension.setter
    def extension(self, extension):                # Az egyéni Property leírót használja
        self.__extension = extension
```

A használat megegyezik a beépített `@property` díszítőével, illetve a `@tulajdonságNév.setter` díszítőével. Íme a `Property` leíróosztály megvalósításának eleje:

```
class Property:
    def __init__(self, getter, setter=None):
        self.__getter = getter
        self.__setter = setter
        self.__name__ = getter.__name__
```

Az osztály előkészítő függvénye egy vagy két függvényt vár paraméterül. Ha díszítőként használjuk, akkor csak a díszítendő függvényt kéri, és belőle jön létre a kiolvasófüggvény; a beállítófüggvény értéke pedig a `None` objektum lesz. A kiolvasófüggvény nevét fogjuk a tulajdonság neveként használni, így aztán minden tulajdonság kap egy kiolvasófüggvényt, esetleg egy beállítófüggvényt is, illetve egy nevet.

```
def __get__(self, instance, owner=None):
    if instance is None:
        return self
    return self.__getter(instance)
```

Valahányszor valami hozzányúl egy tulajdonsághoz, a kiolvasófüggvény meghívásával adjuk vissza az eredményt. A kiolvasófüggvény paramétere az objektumpéldány. Első ránézésre a `self.__getter()` utasítás úgy néz ki, mint egy tagfüggvény hívása, de nem az. A valóságban a `self.__getter` egy jellemző, mégpedig olyan, amely történetesen az átadott tagfüggvényre mutató objektumhivatkozást tárolja. Lényegében tehát annyi történik, hogy kiolvassuk a jellemzőt (`self.__getter`), majd függvényként hívjuk meg `()`. Minthogy nem tagfüggvényként, hanem függvényként hívjuk meg, magunknak kell átadnunk a megfelelő `self` objektumot. Leíróról lévén szó, a leírót használó osztályból származó `self` objektum neve valójában `instance` – hiszen a `self` a leíróobjektum. Ugyanez érvényes a `__set__()` tagfüggvényre is.

```
def __set__(self, instance, value):
    if self.__setter is None:
        raise AttributeError("'{0}' is read-only".format(
                                    self.__name__))
    return self.__setter(instance, value)
```

Ha a beállítófüggvény nincs megadva, `AttributeError` kivételt váltunk ki, egyéb esetben pedig a példánnyal és az új értékkel hívjuk meg a beállítófüggvényt.

```
def setter(self, setter):
    self.__setter = setter
    return self.__setter
```