A tagfüggvény hívására akkor kerül sor, amikor a parancsértelmező például az `@extension.setter` utasításhoz ér. A `setter` paraméter helyén a díszítendő függvény áll. A tagfüggvény elraktározza az átadott beállítófüggvényt (mostantól a `__set__()` tagfüggvényben tudjuk használni), és visszaadja a `setter` paramétert – ugyanis a díszítők visszatérési értéke az általuk díszített függvény, illetve tagfüggvény.

Ezzel áttekintettük a leírók mindhárom, meglehetősen különböző felhasználási területét. A leírók olyan sokoldalú és rugalmas képességekkel gazdagítják a Python nyelvet, amelyek segítségével sok háttérmunka végezhető el, pedig az őket használó osztályok szempontjából csupán egyszerű jellemzőknek tűnnek.

Osztályszintű díszítők

Ahogy a függvények és tagfüggvények elláthatók díszítőkkel, egész osztályok díszítése is megoldható. Az osztályszintű díszítők paramétere egy osztályobjektum (a `class` kifejezés eredménye), visszatérési értékük pedig egy másik osztály – rendes esetben az általuk díszített osztály módosított változata. Ebben a részben két osztályszintű díszítőt tanulmányozunk, így ismerve meg az osztályszintű díszítők megvalósításának módját.

`SortedList` 268 ☞

A 6. fejezetben `SortedList` néven létrehoztunk egy egyedi gyűjteményes adattípust. Az osztály a `self.__list` nevű privát jellemzőben egy egyszerű listát tárolt. Az osztály tagfüggvényei közül nyolc egyszerűen ennek a privát jellemzőnek adta tovább az elvégzendő munkát. Lássuk például a `SortedList.clear()` és a `SortedList.pop()` tagfüggvény megvalósítását:

```
def clear(self):
    self.__list = []

def pop(self, index=-1):
    return self.__list.pop(index)
```

A `clear()` tagfüggvénnyel semmi teendőnk, mivel a `list` típus nem rendelkezik megfelelő tagfüggvénnyel, de a `pop()` és a másik hat, a `SortedList` osztály által átruházott tagfüggvény esetében egyszerűen meghívhatjuk a `list` osztály megfelelő tagfüggvényét. Ez a könyv `Util` moduljában található `@delegate` osztályszintű díszítő hívásával oldható meg. Íme a `SortedList` osztály új változatának eleje:

```
@Util.delegate("__list", ("pop", "__delitem__", "__getitem__",
    "__iter__", "__reversed__", "__len__", "__str__"))
class SortedList:
```

Az első paraméter annak a jellemzőnek a neve, amelyiknek a munkát át szeretnénk adni, a második pedig egy legalább egy tagfüggvényből álló sorozat. A sorozat azokat a tagfüggvényeket tartalmazza, amelyeket a `delegate()` függvénnyel meg szeretnénk valósíttatni – mi magunk nem kívánunk munkát végezni. A `SortedListDelegate.py` fájlban levő `SortedList` osztály is így közelít a problémához, ezért a listában szereplő

tagfüggvényeknek megfelelő kódot nem tartalmaz, még akkor sem, ha gondoskodik a teljes támogatásukról. Íme az az osztályszintű díszítő, amelyik gondoskodik helyettünk a tagfüggvények megvalósításáról:

```
def delegate(attribute_name, method_names):
    def decorator(cls):
        nonlocal attribute_name
        if attribute_name.startswith("__"):
            attribute_name = "_" + cls.__name__ + attribute_name
        for name in method_names:
            setattr(cls, name, eval("lambda self, *a, **kw: "
                                    "self.{0}.{1}(*a, **kw)".format(
                                    attribute_name, name)))
        return cls
    return decorator
```

Azért nem használhatunk sima díszítőt, mert a díszítőknek paramétereket szeretnénk átadni. Így aztán inkább létrehoztunk egy olyan függvényt, amely átveszi a paramétereket, és osztályszintű díszítőt ad vissza. Maga a díszítő egyetlen paramétert vár: egy osztályt – pontosan úgy, ahogy a függvénydíszítő egyetlen függvényt, illetve tagfüggvényt vár paraméterül.

Azért kell használnunk a `nonlocal` kulcsszót, hogy a beágyazott függvény a külső hatókörből származó `attribute_name` változót használhassa, és ne próbálkozzon a saját hatókörében lévő hasonló nevű változó használatával. Arra is figyelnünk kell, hogy a privát jellemzők névváltozásai miatt képesnek kell lennünk a jellemző nevének helyesbítésére. A díszítő viselkedése meglehetősen egyszerű: bejárja a `delegate()` függvénynek átadott valamennyi tagfüggvénynevet, és mindegyikhez új tagfüggvényt hoz létre. Az újonnan létrehozott tagfüggvényeket az osztály jellemzőiként állítja be, mégpedig a megadott tagfüggvény nevével.

Az `eval()` függvényt használtuk a szükséges tagfüggvények létrehozására, mivel ez egyetlen utasítás végrehajtására alkalmas, a `lambda` utasítás pedig tagfüggvény, illetve függvény előállítására való. A `pop()` függvényt előállító kód például az alábbi:

```
lambda self, *a, **kw: self._SortedList__list.pop(*a, **kw)
```

A paramétereket azért adjuk meg a * és a ** forma használatával, hogy minden paraméter számára lehetővé tegyük adott paraméterlista használatát – még azoknak a tagfüggvényeknek is, amelyekre éppen átruházzuk a munkát. Például a `list.pop()` tagfüggvény egyetlen sorszámot vár paraméterül (ha semmit nem kap, akkor alapértelmezés szerint az utolsó elemmel foglalkozik). Ez rendben is van, hiszen ha nem megfelelő számú vagy típusú paramétert adunk át, a munka elvégzéséhez meghívott `list` tagfüggvény majd kiváltja a megfelelő kivételt.

FuzzyBool
240, 247 ☞

Szintén a 6. fejezetben került elő a másodikként vizsgált díszítő. A `FuzzyBool` osztály megvalósításakor említettük, hogy csak az `__lt__()` és az `__eq__()` különleges szerepű tagfüggvényeket adtuk meg (a „<”és az „==”műveletet), minden más összehasonlító műveletet automatikusan állítottunk elő. Nem ismertettük viszont teljes egészében az osztály meghatározásának elejét:

```
@Util.complete_comparisons
class FuzzyBool:
```

A másik négy összehasonlító műveletet a `complete_comparisons()` osztályszintű díszítővel állítottuk elő. Ha adott egy osztály, amely csak a „<” (illetve a „<” és a „==”) műveleteket valósítja meg, a díszítő az alábbi logikai egyenletek alapján állítja elő a hiányzó műveleteket:

$x = y$ ▯ $\neg (x < y$ Ú $y < x)$
x ▯ y ▯ $\neg (x = y)$
$x > y$ ▯ $y < x$
$x \leq y$ ▯ $\neg (y < x)$
$x \geq y$ ▯ $\neg (x < y)$

Ha a díszítendő osztály rendelkezik a „<” és az „==” műveletekkel, a díszítő mindkettőt használni fogja, ha azonban csak a „<” megvalósított az osztályban, akkor annak alapján is képes a többi művelet előállítására. (Az igazság az, hogy a Python automatikusan előállítja a „>” műveletet, ha a „<” adott, és a „!=” műveletet is, ha az „==” művelet ismert, a „<=” művelet ismeretében pedig a „>=” művelet előállítására is képes. Mindez annyit jelent, hogy elegendő a „<”, a „<=” és az „==” műveleteket megadnunk, a Python kikövetkezteti a többit. A fent ismertetett osztályszintű díszítő az általunk megvalósítandó műveleteket azonban mindössze a „<” műveletre korlátozza. Azon felül, hogy a dolog kényelmes, azt is biztosítja, hogy az összehasonlító műveletek ugyanazt a logikát kövessék.)

```
def complete_comparisons(cls):
    assert cls.__lt__ is not object.__lt__, (
            "{0} must define < and ideally ==".format(cls.__name__))
    if cls.__eq__ is object.__eq__:
        cls.__eq__ = lambda self, other: (not
                (cls.__lt__(self, other) or cls.__lt__(other, self)))
    cls.__ne__ = lambda self, other: not cls.__eq__(self, other)
    cls.__gt__ = lambda self, other: cls.__lt__(other, self)
    cls.__le__ = lambda self, other: not cls.__lt__(other, self)
    cls.__ge__ = lambda self, other: not cls.__lt__(self, other)
    return cls
```

A díszítő a következő problémával szembesül: az `object` osztály, amelyből végső soron minden egyéb osztály származik, mind a hat összehasonlító műveletet megadja. Ha ezek közül bármelyiket használjuk, `TypeError` kivétel lép fel. Vagyis a díszítőnek meg kell

tudnia, hogy létezik-e felülírása a „<" és az „==" műveleteknek, magyarán: szabad-e használni őket. Ez könnyen megoldható, ha a díszített osztály megfelelő különleges szerepű tagfüggvényeit összehasonlítjuk az `object` objektumban találhatókkal.

Ha a díszítendő osztályban nem találunk egyedi „<" műveletet, a feltételezés megbukik, ez ugyanis a minimális kívánalom a díszítő működéséhez. Ha van egyedi „==" művelet, akkor használjuk, ha nem, akkor létrehozunk egyet. Ezt követően létrejön a többi művelet is, és visszakapjuk a díszített, immáron mind a hat művelettel rendelkező osztályt.

Az osztályok módosításának talán legegyszerűbb és legközvetlenebb módja a díszítők használata. A másik lehetőség a metaosztályok használata – ez a téma a fejezet későbbi részében kerül sorra.

Metaosztályok ☞ 397

Elvont alaposztályok

Az elvont alaposztály olyan osztály, amelyből nem lehet objektumokat példányosítani, hanem arra használjuk, hogy felületeket adjunk meg, vagyis lényegében felsoroljuk azokat a tagfüggvényeket és tulajdonságokat, amelyeket az alaposztályból származó osztályoknak meg kell valósítaniuk. Az elvont alaposztályokat egyfajta ígéretként használhatjuk – ígéretet teszünk rá, hogy a belőle származó osztályok megvalósítják mindazokat a tagfüggvényeket és tulajdonságokat, amelyeket az alaposztály megad.*

Az elvont alaposztályok legalább egy elvont tagfüggvénnyel, illetve tulajdonsággal bírnak. Az elvont alaposztályok megadhatók megvalósítás nélkül is (vagyis a kódblokkjuk a `pass` utasítás, illetve `raise NotImplementedError()`, ha a felülírást a leszármazott osztályokban ki akarjuk kényszeríteni), de konkrét megvalósítás is jelen lehet, amely a leszármazott osztályok objektumaiból azonnal meghívható. Ez utóbbi eljárásnak főleg akkor van értelme, ha a megvalósítás sok leszármazott esetében azonos. Az elvont alaposztályokban megadhatunk konkrét – nem elvont – tagfüggvényeket és tulajdonságokat is.

Az elvont alaposztályok leszármazottjaiban újra meg kell valósítanunk minden, az alaposztálytól örökölt elvont tagfüggvényt, illetve tulajdonságot – csak ennek végeztével válik példányosíthatóvá az osztály. Az olyan elvont tagfüggvények esetében, amelyek konkrét megvalósítással bírnak (még akkor, is, ha ez a megvalósítás a `pass` utasításra korlátozódik), az osztály a `super()` függvénnyel egyszerűen használatba veheti az elvont alaposztályban lévő változatot. A rendes tagfüggvények és tulajdonságok az öröklődés szabályainak figyelembe vétele mellett a megszokott módon használhatók. Minden elvont alaposztálynak rendelkeznie kell az `abc` modulban

Metaosztályok ☞ 397

* A Python elvont alaposztályainak leírását a `www.python.org/dev/peps/pep-3119` webhelyen lévő PEP 3119 dokumentum tartalmazza. A dokumentumot igazán érdemes elolvasnunk, már a benne lévő remek magyarázat miatt is.

található `abc.ABCMeta` osztályba vagy annak valamelyik leszármazottjába tartozó metaosztállyal (*ABC: Abstract Base Class*, elvont alaposztály). A metaosztályok tárgyalására nemsokára sort kerítünk.

8.3. táblázat *A numbers modul elvont alaposztályai*

Elvont alaposztály	Alaposztálya	Felület	Példák
`Number`	`object`		`complex`, `decimal.Decimal`, `float`, `fractions.Fraction`, `int`
`Complex`	`Number`	`==`, `!=`, `+`, `-`, `*`, `/`, `abs()`, `bool()`, `complex()`, `conjugate()`; illetve a `real` és az `imag` tulajdonság	`complex`, `decimal.Decimal`, `float`, `fractions.Fraction`, `int`
`Real`	`Complex`	`<`, `<=`, `==`, `!=`, `>=`, `>`, `+`, `-`, `*`, `/`, `//`, `%`, `abs()`, `bool()`, `complex()`, `conjugate()`, `divmod()`, `float()`, `math.ceil()`, `math.floor()`, `round()`, `trunc()`; illetve a `real` és az `imag` tulajdonság	`decimal.Decimal`, `float`, `fractions.Fraction`, `int`
`Rational`	`Real`	`<`, `<=`, `==`, `!=`, `>=`, `>`, `+`, `-`, `*`, `/`, `//`, `%`, `abs()`, `bool()`, `complex()`, `conjugate()`, `divmod()`, `float()`, `math.ceil()`, `math.floor()`, `round()`, `trunc()`; illetve a `real`, az `imag`, a `numerator` és a `denominator` tulajdonság	`fractions.Fraction`, `int`
`Integral`	`Rational`	`<`, `<=`, `==`, `!=`, `>=`, `>`, `+`, `-`, `*`, `/`, `//`, `%`, `<<`, `>>`, `~`, `&`, `^`, `|`, `abs()`, `bool()`, `complex()`, `conjugate()`, `divmod()`, `float()`, `math.ceil()`, `math.floor()`, `pow()`, `round()`, `trunc()`; illetve a `real`, az `imag` a `numerator` és a `denominator` tulajdonság	`int`

A Pythonban két csoportját találjuk meg az elvont alaposztályoknak: az egyiket a `collections` modulban, a másikat a `numbers` modulban. Segítségükkel egy objektumra vonatkozó kérdéseket tehetünk fel. Például ha adott az `x` változó, az `isinstance(x, collections.MutableSequence)` kifejezéssel eldönthető, hogy egy sorozatról van-e szó,

az `isinstance(x, numbers.Integral)` kifejezés pedig arra ad választ, hogy egész-e a szám. Ez főként a Python nyelv dinamikus típuskezelését tekintve hasznos, mivel ilyenkor nem feltétlenül ismerjük, vagy nem feltétlenül fontos egy objektum típusa, de esetleg érdekel bennünket, hogy vajon támogatja-e az objektum a használni kívánt műveletet. A két említett modul elvont alaposztályai a 8.3. és a 8.4. táblázatban kaptak helyet. Az `io.IOBase` szintén jelentős alaposztály: belőle származik valamennyi fájl- és folyamkezelő osztály.

Ahhoz, hogy a saját gyűjteményes, illetve szám típusú szerkezeteket megvalósító osztályaink tökéletesen illeszkedjenek a képbe, ki kell alakítanunk a kapcsolatukat a szabványos elvont alaposztályokkal. Például a `SortedList` osztály egy sorozat, de az `isinstance(L, collections.Sequence)` kifejezés értéke `False` (az `L` egy `SortedList` objektum). Ez könnyen rendbehozható, ha a származtatás a megfelelő elvont alaposztályból történik:

```
class SortedList(collections.Sequence):
```

Minthogy alaposztályként a `collections.Sequence` osztályt jelöltük meg, az `isinstance()`-vizsgálat immár `True` értéket ad vissza. Ezen felül mostantól meg kell valósítanunk az `__init__()` (vagy a `__new__()`), a `__getitem__()` és a `__len__()` tagfüggvényeket. Ezt meg is tesszük. A `collections.Sequence` elvont alaposztály rendelkezik rendes (vagyis nem elvont) megvalósításokkal is, mégpedig a `__contains__()`, az `__iter__()`, a `__reversed__()`, a `count()` és az `index()` tagfüggvényeket illetőleg. Ami a `SortedList` osztályt illeti, ezek mindegyikét újból megvalósítjuk, de ha kedvünk tartaná, használhatnánk az elvont alaposztályból származó változatokat is – ehhez mindössze annyit kellene tennünk, hogy nem írjuk meg az új változatokat. A `SortedList` osztály nem lehet a `collections.MutableSequence` alaposztály leszármazottja, hiába változtatható. Ennek az az oka, hogy a `SortedList` osztály nem rendelkezik mindazokkal a tagfüggvényekkel, amelyekkel a `collections.MutableSequence` alaposztály leszármazottjainak rendelkezniük kell – ilyen például a `__setitem__()` és az `append()` tagfüggvény. (A `SortedList` osztály kódja a `SortedListAbc.py` fájlban található.) A metaosztályokról szóló részben újabb olyan módszert ismerhetünk meg, amellyel a `SortedList` osztály a `collections.Sequence` elvont alaposztály leszármazottjává tehető.

Metaosztályok ☞ 397

Most, hogy láttuk a saját osztályaink és a szabványos elvont alaposztályok közötti kapcsolat kialakításának módját, az elvont alaposztályok újabb felhasználási lehetőségével ismerkedünk meg: a felületről adunk előzetes képet saját készítésű osztályaink számára. Három meglehetősen különböző példát tekintünk át, az elvont alaposztályok létrehozásának és használatának három oldalát megismerve.

Kezdetnek vegyünk egy igen egyszerű példát, amely az írható-olvasható tulajdonságok kezelését mutatja be. Az osztály háztartási készülékek ábrázolására használható. Minden létrehozott készüléknek rendelkeznie kell egy csak olvasható karakterlánc-tulajdonsággal (a gyártmány jelölésére), illetve egy írható-olvasható árral. Azt is biztosítani szeretnénk,

hogy az elvont alaposztály `__init__()` tagfüggvényét újra meg kelljen valósítani. Íme az `Appliance.py` fájlban lévő elvont alaposztály. Az `abstractmethod()` és az `abstractproperty()` tagfüggvény használatához szükséges `import abc` utasítás nem látható – a függvények díszítőként használatosak.

```
class Appliance(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def __init__(self, model, price):
        self.__model = model
        self.price = price

    def get_price(self):
        return self.__price
    def set_price(self, price):
        self.__price = price
    price = abc.abstractproperty(get_price, set_price)

    @property
    def model(self):
        return self.__model
```

8.4. táblázat *A collections modul főbb elvont alaposztályai*

Elvont alaposztály	Alaposztálya	Felület	Példák
`Callable`	`object`	`()`	Valamennyi függvény, tagfüggvény és a `lambda`-függvények
`Container`	`object`	`in`	`bytearray`, `bytes`, `dict`, `frozenset`, `list`, `set`, `str`, `tuple`
`Hashable`	`object`	`hash()`	`bytes`, `frozenset`, `str`, `tuple`
`Iterable`	`object`	`iter()`	`bytearray`, `bytes`, `collections.deque`, `dict`, `frozenset`, `list`, `set`, `str`, `tuple`
`Iterator`	`Iterable`	`iter()`, `next()`	
`Sized`	`object`	`len()`	`bytearray`, `bytes`, `collections.deque`, `dict`, `frozenset`, `list`, `set`, `str`, `tuple`
`Mapping`	`Container`, `Iterable`, `Sized`	`==`, `!=`, `[]`, `len()`, `iter()`, `in`, `get()`, `items()`, `keys()`, `values()`	`dict`

8.4. táblázat *A collections modul főbb elvont alaposztályai* *(folytatás)*

Elvont alaposztály	Alaposztálya	Felület	Példák
Mutable-Mapping	Mapping	==, !=, [], del, len(), iter(), in, clear(), get(), items(), keys(), pop(), popitem(), setdefault(), update(), values()	dict
Sequence	Container, Iterable, Sized	[], len(), iter(), reversed(), in, count(), index()	bytearray, bytes, list, str, tuple
Mutable-Sequence	Container, Iterable, Sized	[], +=, del, len(), iter(), reversed(), in, append(), count(), extend(), index(), insert(), pop(), remove(), reverse()	bytearray, list
Set	Container, Iterable, Sized	<, <=, ==, !=, =>, >, &, |, ^, len(), iter(), in, isdisjoint()	frozenset, set
MutableSet	Set	<, <=, ==, !=, =>, >, &, |, ^, &=, |=, ^=, -=, len(), iter(), in, add(), clear(), discard(), isdisjoint(), pop(), remove()	set

A fenti kóddal az osztály metaosztályává az `abc.ABCMeta` osztályt tettük – azért, mert az elvont alaposztályok esetében ez kívánalom; természetesen az `abc.ABCMeta` osztály bármely leszármazottja megfelelt volna. Az `__init__()` tagfüggvényből elvont tagfüggvényt csináltunk, ezzel téve szükségessé az újbóli megvalósítását. Elkészítettünk egy olyan megvalósítást, amelyet a várakozásaink szerint a leszármazottak hívni fognak – ezt azonban nem tudjuk kötelezővé tenni. Elvont írható-olvasható osztályok létrehozásakor nem használhatjuk a díszítő utasításformát, és a kiolvasó-, illetve beállítófüggvények sem kaptak privát nevet: ha így tettünk volna, az az alosztályoknál okozna kényelmetlenséget. A `model` tulajdonság sem elvont, így az alosztályokban nem kell újból megvalósítani. Az `Appliance` osztály nem példányosítható, ugyanis az osztályban elvont jellemzőket találunk. Íme egy példaként szolgáló alosztály:

```
class Cooker(Appliance):
    def __init__(self, model, price, fuel):
        super().__init__(model, price)
        self.fuel = fuel
        price = property(lambda self: super().price,
                         lambda self, price: super().set_price(price))
```

A `Cooker` osztályban újra meg kell valósítanunk az `__init__()` tagfüggvényt és a `price` tulajdonságot. Ami a tulajdonságot illeti, minden munkát átadtunk az alaposztálynak. A `model` csak olvasható tulajdonságot örökölte az osztály. Az `Appliance` („készülék") osztály alapján sok egyéb osztály létrehozható, mondjuk a `Fridge` („hűtő"), a `Toaster` („pirító"), és így tovább.

A következőként áttekintett elvont alaposztály még rövidebb. A `TextFilter.py` fájlban lévő, szöveg szűrésére használható elvont alaposztályról van szó:

```
class TextFilter(metaclass=abc.ABCMeta):
    @abc.abstractproperty
    def is_transformer(self):
        raise NotImplementedError()
    @abc.abstractmethod
    def __call__(self):
        raise NotImplementedError()
```

A `TextFilter` elvont alaposztály semmiféle működésre nem képes, létének egyetlen oka, hogy egy felületet ad meg – az `is_transformer` csak olvasható tulajdonságot és a `__call__()` tagfüggvényt, amelyeket minden leszármazott osztályban újra meg kell valósítani. Minthogy az elvont tulajdonság és az elvont tagfüggvény megvalósítása egyaránt hiányzik, nem szeretnénk, ha az alosztályok meghívnák őket – egy ártalmatlanul megbúvó `pass` utasítás helyett kivételt váltunk ki, ha valami használná őket (például a `super()` függvény meghívásával). Íme egy egyszerű alosztály:

```
class CharCounter(TextFilter):
    @property
    def is_transformer(self):
        return False
    def __call__(self, text, chars):
        count = 0
        for c in text:
            if c in chars:
                count += 1
        return count
```

Ez a szövegszűrő nem végez átalakítást – az átadott szövegben a megadott karaktereket számolja meg – vagyis az `is_transformer()` tagfüggvény `False` értéket ad vissza. Íme a használatát bemutató példa:

```
vowel_counter = CharCounter()
vowel_counter("dog fish and cat fish", "aeiou") # visszatérési érték: 5
```

Két másik szövegszűrő osztályt is megadunk – a `RunLengthEncode` (futamhossz-kódoló) és a `RunLengthDecode` (futamhossz-visszafejtő) osztályokat –; mindkettő átalakítást végez. Így használjuk őket:

```
rle_encoder = RunLengthEncode()
rle_text = rle_encoder(text)
...
rle_decoder = RunLengthDecode()
original_text = rle_decoder(rle_text)
```

A futamhossz-kódoló egy karakterláncot alakít UTF-8 kódolású bájtsorozattá; a `0x00` bájtokat a `0x00`, `0x01`, `0x00` sorozatra cseréli, illetve bármely, 3-255 karakter hosszúságú ismétlődő bájtot a `0x00`, *darabszám*, *bájt* sorozatra cserél. Ha a karakterláncban sok a legalább négy egymást követő azonos bájtból álló „futam", az eljárással a sima UTF-8 kódolású karakterláncnál rövidebbet kapunk. A futamhossz-visszafejtő paramétere egy az előzőekben ismertetett eljárással kódolt karakterlánc, visszatérési értéke pedig az eredeti karakterlánc. Íme a `RunLengthDecode` osztály eleje:

```
class RunLengthDecode(TextFilter):
    @property
    def is_transformer(self):
    return True
def __call__(self, rle_bytes):
    ...
```

A `__call__()` tagfüggvény belső részét elhagytuk, de a könyv mellékleteként érkező forráskódok között megtalálható. A `RunLengthDecode` osztály szerkezete pontosan ugyanilyen.

Az utolsó áttekintett elvont alaposztály egy visszavonási művelet alkalmazásprogramozási felületét (Application Programming Interface, API) adja meg, az alapértelmezett megvalósítással együtt. Íme az `Abstract.py` fájlban lévő teljes elvont alaposztály:

```
class Undo(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def __init__(self):
        self.__undos = []
    @abc.abstractproperty
    def can_undo(self):
        return bool(self.__undos)
    @abc.abstractmethod
    def undo(self):
        assert self.__undos, "nothing left to undo"
        self.__undos.pop()(self)
    def add_undo(self, undo):
        self.__undos.append(undo)
```

Minthogy az `__init__()` és az `undo()` tagfüggvények egyaránt elvontak, újbóli megvalósításuk kötelező. Ugyanez áll a csak olvasható `can_undo` tulajdonságra is. Az alosztályokban nem kötelező az `add_undo()` tagfüggvény újbóli megvalósítása, bár nem is tiltja semmi.

Az `undo()` tagfüggvény rejt némi érdekességet magában. A `self.__undos` listának tagfüggvényekre mutató objektumhivatkozásokat kell tárolnia. Minden tagfüggvénynek az a feladata, hogy a meghívásakor visszavonja a megfelelő műveletet – ez remélhetőleg jobb megvilágításba kerül, ha áttekintettük az `Undo` alosztályt. A visszavonás úgy végezhető el, hogy a `self.__undos` lista végéről a `pop()` tagfüggvénnyel kivesszük az utolsó visszavonást végző tagfüggvényt, amelyet aztán függvényként meghívunk. A függvény paramétere a `self`. (Azért van szükség a `self` átadására, mert függvényként és nem tagfüggvényként hívjuk meg a tagfüggvényt.)

Lássuk a `Stack` osztály elejét! Az osztály az `Undo` osztály leszármazottja, így a rajta végzett műveletek a `Stack.undo()` tagfüggvény paraméterek nélküli meghívásával visszavonhatók.

```
class Stack(Undo):
    def __init__(self):
        super().__init__()
        self.__stack = []

    @property
    def can_undo(self):
        return super().can_undo

    def undo(self):
        super().undo()

    def push(self, item):
        self.__stack.append(item)
        self.add_undo(lambda self: self.__stack.pop())

    def pop(self):
        item = self.__stack.pop()
        self.add_undo(lambda self: self.__stack.append(item))
        return item
```

A `Stack.top()` és a `Stack.__str__()` tagfüggvényt elhagytuk, mivel egyikük sem szolgál semmiféle újdonsággal, és az `Undo` alaposztállyal sincsenek kapcsolatban. Ami a `can_undo` tulajdonságot, illetve az `undo()` tagfüggvényt illeti, a munkát egyszerűen átadjuk az alaposztálynak. Ha a tulajdonság és a tagfüggvény nem lenne elvont, nem kellene újra megvalósítani őket, a hatásuk pedig ugyanaz maradna. Mi azonban ez esetben kényszeríteni akartuk a leszármazott osztályokat az újbóli megvalósításra, hogy a visszavonás lehetőségét mindenképpen figyelembe vegyék. A `push()` és a `pop()` tagfüggvények esetében elvégezzük, majd a visszavonható műveletek között elhelyezzük a műveletet – így válik a frissen elvégzett művelet visszavonhatóvá.

Az elvont alaposztályok a nagy méretű programok, könyvtárak és alkalmazás-keretrendszerek esetében a leghasznosabbak – segítségükkel megoldható, hogy bárki is a program írója, bármilyen is a megvalósítás, a program egyes részei képesek együttműködni egymással, mert az elvont alaposztályaik ugyanazt az alkalmazásprogramozási felületet tartalmazzák.

Többszörös öröklés

Többszörös öröklésnek azokat az eseteket tekintjük, amikor egy osztály legalább két osztálytól örököl. Bár a Python (és például a C++) tökéletesen támogatja a többszörös öröklést, vannak nyelvek – a legfigyelemreméltóbb köztük a Java –, amelyek nem engedik meg a használatát. A többszörös örökléssel kapcsolatos egyik probléma, hogy kialakulhat az a helyzet, amikor egy osztályból is többszörösen örököl az osztályunk (például két alaposztálya is ugyanannak az osztálynak a leszármazottja). Mindez annyit tesz, hogy ha olyan tagfüggvényt hívunk meg, amely nem az osztályban, hanem annak legalább két alaposztályában szerepel (vagy akár az ő alaposztályaikban, és így tovább), akkor az, hogy a tagfüggvénynek pontosan melyik változatát hívjuk, a tagfüggvények megtalálásának sorrendjétől függ – így viszont a többszörös öröklést használó osztályaink némileg törékennyé válnak.

A többszörös öröklés általában elkerülhető, ha egyszeres öröklés (egyetlen alaposztály) használata mellett egy metaosztály megadásával oldjuk meg a további alkalmazásprogramozási felületek támogatását – ugyanis, ahogy az a következőkből kiderül, a metaosztályokkal anélkül adhatunk meg felületeket, hogy egyetlen tagfüggvény, illetve adatjellemző tényleges öröklése megvalósulna. Dönthetünk úgy is, hogy a többszörös öröklést egyetlen konkrét osztállyal valósítjuk meg, és a további felületekhez akár több elvont alaposztályt használunk. További választási lehetőség, hogy egyszeres öröklés mellett objektum-összetétellel dolgozunk.

Mindazonáltal vannak olyan esetek, amelyekben a többszörös öröklés jelenti a kényelmes megoldást. Tételezzük fel például, hogy a korábbi `Stack` osztály új változatát szeretnénk létrehozni, de úgy, hogy az osztály támogassa a befőttek betöltését és mentését. Könnyen elképzelhető, hogy a befőttek kezelésének képességét számos osztályban meg szeretnénk valósítani, így saját, önálló osztályban valósítjuk meg.

```
class LoadSave:
    def __init__(self, filename, *attribute_names):
        self.filename = filename
    self.__attribute_names = []
    for name in attribute_names:
        if name.startswith("__"):
            name = "_" + self.__class__.__name__ + name
        self.__attribute_names.append(name)
    def save(self):
        with open(self.filename, "wb") as fh:
            data = []
            for name in self.__attribute_names:
                data.append(getattr(self, name))
            pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)
```

```
    def load(self):
        with open(self.filename, "rb") as fh:
            data = pickle.load(fh)
            for name, value in zip(self.__attribute_names, data):
                setattr(self, name, value)
```

Az osztálynak két jellemzője van: az egyik a `filename`, amely nyilvános, és bármikor megváltoztatható, a másik az `__attribute_names`, amely rögzített, és csak a példány létrejöttekor kaphat értéket. A `save()` tagfüggvény minden jellemzőnevet bejár, és azt a `data` nevű listát hozza létre, amely minden mentendő jellemző értékét tárolja. Ennek végeztével a lista adatai a befőttbe kerülnek. A `with` utasítás biztosítja, hogy amennyiben sikeresen megnyitottuk a fájlt, akkor be is zárjuk, illetve azt, hogy bármely, a fájllal, illetve a befőttel kapcsolatos kivétel eljusson a hívóhoz. A `load()` tagfüggvény a jellemzők neveit és a megfelelő betöltött adatelemeket járja be, és minden jellemző értékéül a betöltött értéket adja meg.

Az alábbiakban ismertetjük a `FileStack` osztályt, amely az előző részben megismert `Undo` és az itt szereplő `LoadSave` osztálytól egyaránt örököl.

```
class FileStack(Undo, LoadSave):

    def __init__(self, filename):
        Undo.__init__(self)
        LoadSave.__init__(self, filename, "__stack")
        self.__stack = []

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

Az osztály további része pontosan megegyezik a `Stack` osztállyal, így az itteni közlésétől eltekintettünk. Az `__init__()` tagfüggvényben nem használjuk a `super()` függvényt; helyette – minthogy a `super()` függvény nem képes a szándékaink kitalálására – nekünk kell megadnunk az alaposztályokat. Ami a `LoadSave` osztály előkészítését illeti, átadjuk a használandó fájlnevet, csakúgy, mint azoknak a jellemzőknek a nevét, amelyeket menteni szeretnénk – ebben az esetben csak a `__stack` nevű privát jellemzőről van szó. (Az `__undos` jellemzőt nem akarjuk menteni, sőt ebben az esetben nem is tudnánk, mert egy tagfüggvénylistát tárolunk benne, így ez a lista nem alakítható befőtté.)

A `FileStack` osztály rendelkezik az `Undo` osztály összes tagfüggvényével, valamint a `LoadSave` osztály `save()` és `load()` tagfüggvényeivel is. A `save()` tagfüggvény újbóli megvalósításától eltekintettünk, hiszen a mostani is remekül működik. Ami azonban a `load()` tagfüggvényt illeti, betöltés után gondoskodnunk kell a visszavonásokat tároló változó kitakarításáról. Ez azért szükséges, mert esetleg egy mentés után teszünk egyetmást, amit akár betöltés is követhet. A betöltés eredményeképpen törlődik minden, ami addig történt, ezen műveletek visszavonásának tehát a továbbiakban nincs értelme. Az eredeti `Undo` osztály nem rendelkezik `clear()` tagfüggvénnyel, vagyis azt most kell létrehozni:

```
    def clear(self):                # Az Undo osztályban
        self.__undos = []
```

A `Stack.load()` tagfüggvényben a `LoadSave.load()` tagfüggvény hívására azért használtuk a `super()` függvényt, mert nem létezik olyan `Undo.load()` tagfüggvény, amely kétséget okozhatna. Ha mindkét alaposztályban lenne `load()` tagfüggvény, az, hogy melyiket hívnánk, azon múlna, hogy a Python milyen sorrendben találja meg őket. A `super()` függvény használatát igyekszünk azokra az esetekre korlátozni, amikor nem merülhet fel kétség. Más esetekben megadjuk a megfelelő alaposztály nevét is, és soha nem hagyatkozunk a megtalálási sorrendre. Ami a `self.clear()` hívást illeti, megint csak nem merül fel kétség, hiszen `clear()` tagfüggvény csak az `Undo` osztályban található. A `super()` függvény használatára azért nincs szükség, mert – a `load()` tagfüggvénnyel ellentétben – a `Filestack` osztály nem rendelkezik `clear()` tagfüggvénnyel.

Mi történik, ha a későbbiekben a `FileStack` osztály is kiegészül egy `clear()` nevű tagfüggvénnyel? Nos, ez tönkretenné a `load()` tagfüggvény működését. A probléma egyik megoldását az jelentheti, ha a `load()` tagfüggvény belsejében a sima `self.clear()` hívás helyett a `super().clear()` hívással dolgoznánk. Ennek eredménye az lenne, hogy az elsőként megtalált alaposztály `clear()` tagfüggvényét hívnánk meg. Az efféle problémák elkerülése végett érdemes alapelvként elfogadnunk, hogy a többszörös öröklésű osztályok esetében az alaposztályok nevét beledrótozzuk az osztályunkba – a fenti példában az `Undo.clear(self)` hívást használjuk. Illetve megpróbálkozhatunk azzal, hogy egyáltalán nem alkalmazzuk a többszörös öröklést, hanem összetétellel (aggregation) oldjuk meg az ilyen eseteket – például az `Undo` osztályból öröklünk, a `LoadSave` osztályt pedig már a létrehozásakor felkészítjük az összetétel alkalmazására.

A többszörös öröklés eredményeként két meglehetősen különböző osztály keverékét hoztuk létre, anélkül, hogy akár a visszavonást, akár a betöltést, akár a mentést meg kellett volna valósítanunk – helyette az alaposztályok által biztosított képességekre hagyatkoztunk. Ez igen kényelmes lehet, és különösen akkor használható remekül, amikor a szülőosztályok felületében nincs átfedés.

Metaosztályok

A metaosztály úgy viszonyul az osztályhoz, mint az osztály a példányhoz, vagyis a metaosztályokat osztályok létrehozására használjuk, pontosan úgy, ahogy az osztályokat használjuk példányok létrehozására. Ahogy az `isinstance()` függvénnyel kideríthető, hogy egy példány egy adott osztály leszármazottja-e, az `issubclass()` függvény megmondja, hogy egy osztály – `dict`, `int`, `SortedList` – leszármazottja-e egy másik osztálynak.

A metaosztályok használatának legegyszerűbb esete, amikor velük illesztjük az egyéni osztályainkat a Python elvont alaposztályainak szabványos hierarchiájába. Például ha azt akarjuk, hogy a `SortedList` osztály a `collections` osztály alá tartozzon, nem öröklést állítunk be – ezt korábban már bemutattuk –, hanem egyszerűen bejegyezzük a `SortedList` osztályt a `collections.Sequence` osztály alá. Így:

```
class SortedList:
    ...
collections.Sequence.register(SortedList)
```

Az osztály szokásos megadását követően bejegyezzük az osztályt a `collections.Sequence` osztály alá. A bejegyzés az osztályt *virtuális alosztállyá** teszi. A virtuális alosztályok az `isinstance()`, illetve az `issubclass()` függvény kérdésére válaszolva az adott alaposztály alosztályának mondják magukat, de sem a tagfüggvényeket, sem az adatokat nem öröklik.

Az osztályok efféle bejegyzése elvárássá teszi, hogy az osztályunk támogassa annak az osztálynak a felületét, amelyik alá bejegyeztük – biztosítékot azonban nem nyújt minderre. A metaosztályok használatának egyik oka, hogy az elvárás mellett a biztosíték is megjelenjék. A másik használati lehetőség, hogy egy osztályt módosítunk a segítségével – éppúgy, mint az osztályszintű díszítőkkel. Természetesen egy metaosztály egyidejűleg szolgálhatja mindkét cél elérését.

Tegyük fel, hogy osztályok egy olyan csoportját szeretnénk létrehozni, ahol minden osztály rendelkezik `load()` és `save()` tagfüggvényekkel. Ezt úgy tehetjük meg, hogy létrehozunk egy osztályt, amely, ha metaosztályként használjuk, ellenőrzi, hogy az említett tagfüggvények jelen vannak-e.

```
class LoadableSaveable(type):
    def __init__(cls, classname, bases, dictionary):
        super().__init__(classname, bases, dictionary)
        assert hasattr(cls, "load") and \
               isinstance(getattr(cls, "load"),
                          collections.Callable), ("class '" +
               classname + "' must provide a load() method")
        assert hasattr(cls, "save") and \
               isinstance(getattr(cls, "save"),
                          collections.Callable), ("class '" +
               classname + "' must provide a save() method")
```

Azoknak az osztályoknak, amelyeket metaosztályként használunk, mindenképpen a hierarchia tetején álló `type` metaosztálytól vagy annak egy alosztályától kell származniuk. Figyeljük meg, hogy a fent megadott osztály meghívására akkor kerül sor, amikor az őt használó *osztályokat* példányosítjuk, tehát minden valószínűség szerint nem túl gyakran, vagyis a futásidejű költség meglehetősen alacsony. Figyeljük meg azt is, hogy az ellenőrzéseket az osztály létrehozása után kell foganatosítanunk (a `super()` függvényt meghívva), mivel az osztály jellemzői magán az osztályon belül csak ekkor válnak elérhetővé. (A jellemzők benne vannak a szótárban, de jobban szeretjük az ellenőrzéseket magán az előkészített osztályon elvégezni.)

---

* A Python szóhasználatában a *virtuális* nem ugyanazt jelenti, mint a C++ esetében.

A `hasattr()` függvénnyel is ellenőrizhettük volna, hogy a `load` és a `save` jellemzők hívhatók-e – vagyis van-e `__call__` jellemzőjük –, de inkább azt néztük meg, hogy a `collections.Callable` példányairól van-e szó. A `collections.Callable` elvont osztály megkívánja – de nem biztosítja – hogy az alosztályai, illetve a virtuális alosztályai hívhatók legyenek.

Ha az osztály – a `type.__new__()` tagfüggvény vagy egy újbóli megvalósítása segítségével – létrejött, az `__init__()` tagfüggvényének meghívásával a metaosztályt is előkészítettük. Az `__init__()` tagfüggvény paraméterei az éppen létrehozott osztály; az osztály neve (ami az *osztály*`.__`*név*`__` tulajdonságból is kiolvasható), az osztály alaposztályainak listája (kivéve az `object` osztályt, így akár üres is lehet), és a szótár, amely azokat a jellemzőket tárolja, amelyek akkor váltak osztályszintű jellemzőkké, amikor az osztály létrejött, hacsak a metaosztály `__new__()` tagfüggvényének egy esetleges újbóli megvalósításakor másképp nem rendelkeztünk.

Nézzünk pár interaktív példa annak szemléltetésére, hogy mi történik, ha a `LoadableSaveable` metaosztály segítségével hozunk létre osztályokat:

```
>>> class Bad(metaclass=Meta.LoadableSaveable):
...     def some_method(self): pass
Traceback (most recent call last):
...
AssertionError: class 'Bad' must provide a load() method
```

A metaosztályban megadottak miatt az őt használó osztályoknak bizonyos tagfüggvényeket meg kell valósítaniuk, ha pedig nem tennék – ahogy most sem –, akkor `AssertionError` kivétel lép fel.

```
>>> class Good(metaclass=Meta.LoadableSaveable):
...     def load(self): pass
...     def save(self): pass
>>> g = Good()
```

A `Good` osztály eleget tesz a metaosztály felületre vonatkozó kívánalmainak, bár a bennünk kialakult, a viselkedésével kapcsolatos elvárásoknak nem felel meg.

A metaosztályok az őket használó osztályok megváltoztatására is használhatók. Ha a változtatás az éppen létrehozott osztály nevét, az alaposztályát, vagy a szótárát (a `__slots__` -ra gondolunk) érinti, akkor a metaosztály `__new__()` tagfüggvényét kell újból megvalósítanunk, de más változtatások esetében – új tagfüggvénnyel vagy adatjellemzővel történő kiegészítés – elegendő az `__init__()` tagfüggvény újbóli megvalósítása – bár ez a `__new__()` tagfüggvényben is elvégezhető. A következőkben egy olyan metaosztályt tekintünk át, amely az őt használó osztályok módosítását pusztán a `__new__()` tagfüggvényével végzi.

A `@property` és a `@name.setter` díszítők használatán felüli lehetőségként létrehozhatunk olyan osztályokat is, ahol egyszerű elnevezési irányelveket léptetünk életbe a tulajdonságok azonosítására. Például ha az osztály rendelkezik `get_`*`név`*`()` és `set_`*`név`*`()` tagfüggvénnyel, akkor várhatóan találunk egy privát `__`*`név`* nevű tulajdonságot az osztályban, amelyet kiolvasáskor és beállításkor a *`példány`*`.`*`név`* alakban érünk el. Mindez megoldható egy metaosztály használatával. Íme egy a fenti irányelveket követő példaosztály:

```
class Product(metaclass=AutoSlotProperties):
    def __init__(self, barcode, description):
        self.__barcode = barcode
        self.description = description
    def get_barcode(self):
        return self.__barcode
    def get_description(self):
        return self.__description
    def set_description(self, description):
        if description is None or len(description) < 3:
            self.__description = "<Invalid Description>"
        else:
            self.__description = description
```

A `__barcode` („vonalkód") privát tulajdonságnak az előkészítés során kell értéket adnunk, mivel nem tartozik hozzá beállítófüggvény – a dolog másik folyománya, hogy így a tulajdonság csak olvashatóvá válik. A `description` viszont írható-olvasható tulajdonság. Íme néhány interaktív példa:

```
>>> product = Product("101110110", "8mm Stapler")
>>> product.barcode, product.description
('101110110', '8mm Stapler')
>>> product.description = "8mm Stapler (long)"
>>> product.barcode, product.description
('101110110', '8mm Stapler (long)')
```

Ha a `__barcode` tulajdonságnak próbálunk értéket adni, `AttributeError` kivétel lép fel, és megjelenik a „can't set attribute" (a jellemző értékét nem lehet beállítani) hibaüzenet.

Ha egy pillantást vetünk a `Product` osztály jellemzőire – például a `dir()` függvénnyel –, a nyilvános jellemzők közül csak a `barcode` és a `description` nevűeket találjuk meg. A `get_`*`név`*`()` és `set_`*`név`*`()` tagfüggvény már nincs ott – a *`név`* tulajdonság került a helyükre. A vonalkódot és a leírást tároló változók szintén privát változók (`__barcode` és `__description`), és memóriarekeszként kaptak helyet, hogy csökkenjen az osztály memóriaigénye. Mindezt az `AutoSlotProperties` metaosztály végzi el, amely egyetlen tagfüggvényből áll:

```
class AutoSlotProperties(type):
    def __new__(mcl, classname, bases, dictionary):
        slots = list(dictionary.get("__slots__", []))
        for getter_name in [key for key in dictionary
                            if key.startswith("get_")]:
            if isinstance(dictionary[getter_name],
                          collections.Callable):
                name = getter_name[4:]
                slots.append("__" + name)
                getter = dictionary.pop(getter_name)
                setter_name = "set_" + name
                setter = dictionary.get(setter_name, None)
                if (setter is not None and
                    isinstance(setter, collections.Callable)):
                    del dictionary[setter_name]
                dictionary[name] = property(getter, setter)
        dictionary["__slots__"] = tuple(slots)
        return super().__new__(mcl, classname, bases, dictionary)
```

A metaosztályok `__new__()` osztályszintű tagfüggvényének a metaosztály és a létrehozandó osztály neve, alaposztályai és szótára a paraméterei. Azért kell az `__init__()` tagfüggvény helyett a `__new__()` tagfüggvény újbóli megvalósítását használnunk, mert az osztály létrehozása előtt szeretnénk megváltoztatni a szótárat.

Először a `__slots__` gyűjteményt másoljuk át, ha pedig nem rendelkezünk ilyennel, létrehozunk egy üreset. Figyelünk, hogy lista legyen, nem pedig `tuple`, különben nem tudnánk módosítani. A szótárban lévő jellemzők közül kiválasztjuk azokat, amelyeknek a neve „`get_`”-tel kezdődik és hívhatók, vagyis a kiolvasófüggvényeket. A `__slots__` gyűjteménybe a megfelelő adatok tárolására minden kiolvasófüggvényhez egy-egy privát név kerül, a `get_`*`név`*`()` tagfüggvény esetében például a `__`*`név`*. Ennek végeztével vesszük a kiolvasófüggvényre mutató hivatkozást, és az eredeti nevére vonatkozó bejegyzést töröljük a szótárból (ezt egy menetben végezzük, a `pop()` függvénnyel). Ha van beállítófüggvény, akkor ugyanezt tesszük vele is, majd új elemet helyezünk el a szótárban. A kulcs a kívánt tulajdonságnév lesz, például ha a kiolvasófüggvény neve `get_`*`név`*`()`, akkor a tulajdonság neve *`név`*. Az elem értékéül egy tulajdonságot adunk meg, a szótárban fellelt és onnan eltávolított kiolvasófüggvénnyel és beállítófüggvénnyel (ami a `None` objektum is lehet).

Amikor ezzel is megvagyunk, az eredeti `__slots__` gyűjteményt lecseréljük a módosítottra, amelyben minden elhelyezett tulajdonság privát helyet kap, majd folytatjuk a hívást az alaposztálynál, amely elvégzi az osztály tényleges létrehozását, de már a módosított szótárral. Gondoljuk arra, hogy ez esetben a `super()` függvény meghívásakor pontosan meg kell adnunk a metaosztályt – a `__new__()` tagfüggvény hívásakor mindig ez a helyzet, ugyanis nem példányszintű, hanem osztályszintű tagfüggvényről van szó.

A fenti példában azért nem írtunk `__init__()` tagfüggvényt, mert minden munkát a `__new__()` tagfüggvényben végeztünk el. Mindazonáltal semmi sem gátolja mind a `__new__()`, mind az `__init__()` tagfüggvény újbóli megvalósítását, külön-külön feladatot adva nekik.

Ha párhuzamot vonunk az összetétel és az öröklés, illetve a kézifúrók, valamint a díszítők és a leírók, illetve az elektromos ütvefúrók között, akkor a metaosztályok valahol a lézersugárral egyenértékűek – ha a használhatóságot és sokoldalúságot tekintjük. A metaosztályok használata nem az első, amihez a fejlesztő fordul, hanem sokkal inkább az utolsó – kivéve azt az esetet, amikor alkalmazás-keretrendszerek fejlesztéséről van szó, és nem akarjuk, hogy a felhasználóink szenvedjenek, amíg felismerik a kínált előnyöket.

Függvényközpontú programozás

A függvényközpontú programozás a programozás olyan megközelítése, amelyben a számítások alapegységei olyan függvények, amelyek nem módosítják a saját paramétereiket, nem hivatkoznak a program állapotjelzőire, és nem is változtatják meg azokat, az eredményüket pedig visszatérési érték formájában adják meg. Az efféle programozási megközelítés mellett szóló érvek közül az egyik legerősebb az, hogy – legalábbis elméletben – sokkal egyszerűbb elszigetelt függvényeket fejleszteni, és így az ilyen programokban a hibakeresés is könnyebbé válik. Ezt az is elősegíti, hogy az ilyen programoknak nincsenek állapotváltozásaik, tehát a működésük matematikai eszközökkel is kifejezhető.

A függvényközpontú programozáshoz szorosan kapcsolódik a *megfeleltetés* (mapping), a *szűrés* (filtering) és az *elhagyás* (reducing) fogalma. A megfeleltetés lényege, hogy egy adott függvénnyel és egy bejárható típusú objektummal (iterable) olyan új bejárható objektumot (vagy listát) állítunk elő, ahol az egyes elemeket az eredeti bejárható objektum megfelelő elemeit a függvénynek átadva visszatérési értékként kapjuk meg. Ennek kivitelezésére a beépített `map()` függvény szolgál. Például:

```
list(map(lambda x: x ** 2, [1, 2, 3, 4]))     # visszatérési érték:
                                                [1, 4, 9, 16]
```

A `map()` függvény paramétere egy függvény és egy bejárható objektum, a visszatérési értéke pedig a hatékonyság okán nem lista, hanem szintén egy bejárható objektum. Ezúttal az eredmény érthetőbbé tétele miatt listát kértünk:

```
[x ** 2 for x in [1, 2, 3, 4]]   # visszatérési érték: [1, 4, 9, 16]
```

A `map()` függvény helyén gyakran használhatunk generátorkifejezést. Ezúttal a `list()` használatának elkerülése végett listaértelmezést használtunk – a kifejezés úgy alakítható generátorrá, hogy a külső szögletes zárójelek helyett rendes zárójelet használunk.

A szűrés lényege, hogy egy függvény és egy bejárható objektum segítségével új bejárható objektumot állítunk elő. A régi objektum elemei akkor kerülnek át az újba, ha a függvény `True` értékkel tér vissza. A `filter()` beépített függvény ezt valósítja meg:

```
list(filter(lambda x: x > 0, [1, -2, 3, -4]))      # visszatérési érték:
                                                     [1, 3]
```

A `filter()` függvény paramétere egy függvény és egy bejárható objektum, a visszatérési értéke pedig egy másik bejárható objektum.

```
[x for x in [1, -2, 3, -4] if x > 0]      # visszatérési érték: [1, 3]
```

A `filter()` függvény mindig helyettesíthető egy generátorkifejezéssel vagy egy listaértelmezéssel.

Az elhagyás lényege, hogy egy függvény és egy bejárható objektum segítségével egyetlen visszatérési értéket állítunk elő. A visszatérési érték előállítása úgy történik, hogy a függvényt az első két értékkel paraméterezve hívjuk meg, majd a paraméterek az előző hívás visszatérési értéke és a harmadik érték lesznek, és így tovább, amíg valamennyi érték sorra nem került. A `functools` modul `functools.reduce()` függvénye ezt valósítja meg. Íme két kódsor, amelyek ugyanazt a számítást végzik el:

```
functools.reduce(lambda x, y: x * y, [1, 2, 3, 4])  # visszatérési
                                                     # érték: 24
functools.reduce(operator.mul, [1, 2, 3, 4])  # visszatérési érték: 24
```

Az `operator` modulnak minden, a Python nyelvben használatos művelethez van megfelelő függvénye, mégpedig elsősorban a függvényközpontú programozás megkönnyítése végett. A fenti példa második sorában ahelyett, hogy az első sorhoz hasonlóan a `lambda` utasítással hoztuk volna létre a szorzás műveletét, inkább az `operator.mul()` függvényt használtuk.

A Python beépített elhagyófüggvényekkel is rendelkezik. A bejárható objektummal paraméterezett `all()` függvény akkor ad vissza `True` értéket, ha a `bool()` függvényt az objektum egyes elemeivel meghívva a visszatérési érték mindig `True`. Az `any()` függvény visszatérési értéke `True`, ha bármelyik elem kiértékelése `True` értéket ad. A `max()` függvény a bejárható objektum legnagyobb elemét adja vissza, a `min()` függvény pedig a legkisebbet. A `sum()` függvény a bejárható objektum elemeinek összegével tér vissza.

Az alapelvek tisztázása után tekintsünk meg néhány további példát. Először több módszerrel is kiszámoljuk a `files` listában lévő fájlok együttes méretét.

```
functools.reduce(operator.add, (os.path.getsize(x) for x in files))
functools.reduce(operator.add, map(os.path.getsize, files))
```

A `map()` függvény használata gyakran eredményez a listaértelmezésnél, illetve a generátorkifejezésnél rövidebb megoldást, kivéve, ha feltétel is van. A hozzáadás függvénye ezúttal nem a `lambda x, y: x + y`, hanem az `operator.add()` volt.

Ha csak a .py kiterjesztésű fájlok méretére vagyunk kíváncsiak, a más kiterjesztésű fájlokat kiszűrhetjük. A feladat az alábbi három módszerrel oldható meg:

```
functools.reduce(operator.add, map(os.path.getsize,
                 .filter(lambda x: x.endswith(".py"), files)))
functools.reduce(operator.add, map(os.path.getsize,
                (x for x in files if x.endswith(".py"))))
functools.reduce(operator.add, (os.path.getsize(x)
                for x in files if x.endswith(".py")))
```

Könnyen belátható, hogy a második és harmadik módszer jobb az elsőnél, ugyanis ezekben nem kell megírnunk a lambdafüggvényt. A generátorkifejezés (illetve a listaértelmezés), valamint a map() és a filter() függvény használata közötti döntés azonban a legtöbb esetben pusztán személyes megfontolásokon alapul.

A map(), a filter() és a functools.reduce() függvény használata gyakran oda vezet, hogy szükségtelenné válnak a ciklusok – a fenti példák is ezt szemléltetik. Ezek a függvények olyankor hasznosak, amikor függvényközpontú nyelven írt programokat alakítunk át, de a Python nyelvben a map() függvény helyett rendszerint alkalmazhatunk listaértelmezést, a filter() függvény helyett pedig feltételt tartalmazó listaértelmezést.
A functools.reduce() függvény pedig sok esetben kiküszöbölhető, ha a Python beépített függvényei közül az all(), az any(), a max(), a min() vagy a sum() valamelyikét használjuk. Például:

```
sum(os.path.getsize(x) for x in files if x.endswith(".py"))
```

A fenti kifejezés eredménye megegyezik az előző háróméval, de ez lényegesen tömörebb.

operator.attrgetter() függvény 375 ☜

Az operator modul, azon felül, hogy a Python műveleteinek megfelelő függvényeket találunk benne, az operator.attrgetter() és az operator.itemgetter() függvényeket is tartalmazza. Az előzőről a fejezet korábbi részében futólag már szóltunk. Mindkét függvény olyan függvényeket ad vissza, amelyeket a megadott jellemzők, illetve elemek kinyerésére használhatunk.

Amíg szeleteléssel egy lista egy része kapható meg, szeleteléssel együtt alkalmazott egyeléssel pedig részek egy sorozata (például az L[::3] kifejezéssel a lista minden harmadik eleme), addig az operator.itemgetter() függvénnyel egy lista tetszőleges elemeit kaphatjuk meg (például: operator.itemgetter(4, 5, 6, 11, 18)(L)).
Az operator.itemgetter() függvénytől visszakapott függvényt nem kötelező azonnal meghívni, sem pedig elvetni, mint ahogy az előző példában tettük – ha megtartjuk, paraméterül adható a map(), a filter() vagy a functools.reduce() függvények bármelyikének, használhatjuk szótárban, listában, vagy halmazértelmezésben.

Ha adatok rendezése a cél, megadhatunk egy kulcsfüggvényt. Ez a függvény bármilyen függvény lehet: lambdafüggvény, beépített függvény, tagfüggvény (például az `str.lower()`), vagy az `operator.attrgetter()` függvény visszatérési értékeként kapott függvény. Ha például feltételezzük, hogy az `L` listában `priority` jellemzővel bíró objektumok vannak, a listát rendezhetjük a prioritás szerint, mégpedig így: `L.sort(key=operator.attrgetter("priority"))`.

A már említett `functools` és `operator` modulokon felül a függvényközpontú programozás során az `itertools` modul is jó szolgálatot tehet. Több lista bejárása például nem csak a listák egymáshoz fűzésével lehetséges, hanem az `itertools.chain()` függvény használatával is:

```
for value in itertools.chain(data_list1, data_list2, data_list3):
    total += value
```

Az `itertools.chain()` függvény visszatérési értéke egy olyan bejárható objektum, amelyben először az első lista egymásra következő értékei vannak, aztán a másodiké, és így tovább, amíg az összes lista összes értéke sorra nem kerül. Az `itertools` modulban sok egyéb függvény található, a dokumentációja pedig számos rövid, ámde hasznos és elolvasásra érdemes példával szolgál.

Részleges függvények alkalmazása

A részleges függvények (parciális függvények) alkalmazása azt jelenti, hogy egy már létező függvényből és annak pár paraméteréből új függvényt alakítunk ki. Az új függvény ugyanazt teszi, mint az eredeti, de néhány paraméter rögzítettsége mellett – amelyeket így a hívónak nem kell átadnia. Íme egy nagyon egyszerű példa:

```
enumerate1 = functools.partial(enumerate, start=1)
for lino, line in enumerate1(lines):
    process_line(i, line)
```

Az első sorban `enumerate1()` néven új függvényt hozunk létre, amely az átadott `enumerate()` függvény burkolója. A megadott kulcsszóparaméter (a `start=1`) miatt az `enumerate1()` az eredeti függvényt – esetünkben a `lines` objektumot – egy rögzített paraméterrel hívja meg; és természetesen átadja a híváskor kapott további paramétereket. A fenti példában az `enumerate1()` függvényt arra használjuk, hogy a sorok számozását a hagyományos módon, 1-től kezdve végezzük el.

A részleges függvények segítségével egyszerűsíthetjük a kódot, különösen akkor, amikor újra meg újra ugyanazokkal a paraméterekkel kell meghívnunk egy függvényt. Például amikor UTF-8 kódolású fájl feldolgozására van szükség, ahelyett, hogy minden alkalommal megadnánk a kódolásra, illetve a megnyitásra vonatkozó paramétereket, az említett paraméterek rögzítésével létrehozhatunk egy függvénypárt:

```
reader = functools.partial(open, mode="rt", encoding="utf8")
writer = functools.partial(open, mode="wt", encoding="utf8")
```

Innentől kezdve a `reader(fájlnév)` hívással olvasásra, a `writer(fájlnév)` hívással pedig írásra nyithatunk meg egy fájlt.

A részleges függvényeket a grafikus felhasználói felületek programozásában (a témakört a 17. fejezet tárgyalja) igen elterjedten használják. Gyakran kényelmes, ha pár gomb közül egyet lenyomva ugyanazt a függvényt hívhatjuk meg. Például:

```
loadButton = tkinter.Button(frame, text="Load",
                            command=functools.partial(doAction, "load"))
saveButton = tkinter.Button(frame, text="Save",
                            command=functools.partial(doAction, "save"))
```

A fenti példa azt a `tkinter` nevű, grafikus felhasználói felület programozására alkalmas könyvtárat használja, amely alapértelmezés szerint együtt települ a Pythonnal.
A `tkinter.Button` osztállyal gombok hozhatók létre – ezúttal kettőt készítettünk, mindkettőt egyazon kereten belül. Mindkettőre ráírtuk, hogy mire való. Mindkét gomb `command` jellemzőjének értéke az a függvény, amelyet a gomb lenyomásakor a `tkinter` objektumnak meg kell hívnia – esetünkben a `doAction()` függvényről van szó. Részleges függvény használatával biztosítottuk, hogy a `doAction()` függvénynek átadott első paraméter egy olyan karakterlánc legyen, amely arról tájékoztatja a függvényt, hogy melyik gombot nyomtuk le. Így a `doAction()` függvény el tudja dönteni, hogy mit kell tennie.

Példa: Valid.py

Leírók
☞ 379

Ebben a részben a leírókat az osztályszintű díszítőkkel együttesen arra fogjuk használni, hogy egy érvényesített jellemzők létrehozására szolgáló, jól használható eljárást alakítsunk ki.

Osztályszintű díszítők
☞ 384

Egészen eddig, ha biztosak akartunk benne lenni, hogy egy jellemző érvényes értéket kap, csak a tulajdonságokra, vagy a beállító és kiolvasó tagfüggvények használatára támaszkodhattunk. Az efféle megközelítés hátránya abban rejlik, hogy érvényességvizsgáló kódot kell elhelyeznünk minden olyan osztályban és minden olyan jellemzőnél, ahol ennek szükségét érezzük.
Kényelmesebb és egyszerűbben módosítható eljárás lenne, ha az osztályok olyan jellemzőket kapnának, amelyekbe a szükséges ellenőrzőkód be van építve. Íme a használni kívánt utasításforma példája:

```
@valid_string("name", empty_allowed=False)
@valid_string("productid", empty_allowed=False,
              regex=re.compile(r"[A-Z]{3}\d{4}"))
@valid_string("category", empty_allowed=False, acceptable=
          frozenset(["Consumables", "Hardware", "Software", "Media"]))
```

```
@valid_number("price", minimum=0, maximum=1e6)
@valid_number("quantity", minimum=1, maximum=1000)
class StockItem:

    def __init__(self, name, productid, category, price, quantity):
        self.name = name
        self.productid = productid
        self.category = category
        self.price = price
        self.quantity = quantity
```

A `StockItem` osztály minden jellemzőjét érvényesítettük.

Szabályos kifejezések ☞ 463

A `productid` jellemző például csak olyan karakterláncot kaphat értékül, ami nem üres, három nagybetűvel kezdődik, és négy számra végződik, a `category` jellemző pedig csak olyat, ami nem üres, és a megadott értékek valamelyikével megegyezik. A `quantity` jellemző egy szám, amely nem kisebb egynél, és nem nagyobb ezernél. Ha érvénytelen érték beállításával próbálkozunk, kivétel lép fel.

Osztályszintű díszítők 384 ☞

Az érvényesítést osztályszintű díszítők és leírók együttes használatával oldottuk meg. Ahogy korábban már jeleztük, az osztályszintű díszítők egyetlen paramétert kaphatnak – a díszítendő osztályt. Így aztán ezúttal is ahhoz a módszerhez folyamodtunk, amelyet az osztályszintű díszítőkkel együtt ismertünk meg. A `valid_string()` és a `valid_number()` függvény az általunk megadott összes paramétert elfogadja, majd egy díszítőt ad vissza, amely viszont egy osztályt vár paraméterül, és az osztály módosított változatával tér vissza.

Az alábbiakban a `valid_string()` függvény működését tekintjük át.

```
def valid_string(attr_name, empty_allowed=True, regex=None,
                 acceptable=None):
    def decorator(cls):
        name = "__" + attr_name
        def getter(self):
            return getattr(self, name)
        def setter(self, value):
            assert isinstance(value, str), (attr_name +
                                            " must be a string")
            if not empty_allowed and not value:
                raise ValueError("{0} may not be empty".format(
                                 attr_name))
            if ((acceptable is not None and value not in acceptable) or
                (regex is not None and not regex.match(value))):
                raise ValueError("{0} cannot be set to {1}".format(
                                 attr_name, value))
            setattr(self, name, value)
        setattr(cls, attr_name, GenericDescriptor(getter, setter))
        return cls
    return decorator
```

A függvény először egy osztályszintű díszítőfüggvényt hoz létre, amelynek egyetlen paramétere egy osztály. A díszítő a dísztett osztályt két jellemzővel egészíti ki: egy privát adatjellemzővel és egy leíróval. Például ha a `valid_string()` függvényt a „productid" névvel hívjuk meg, a `StockItem` osztályba bekerül a termékazonosítót tároló `__productid` jellemző és a leíró `productid` jellemző, amely az érték elérésére szolgál. Ha például az `item = StockItem("TV", "TVA4312", "Electrical", 500, 1)` utasítással hozunk létre egy elemet, a termékazonosító az `item.productid` utasítással olvasható ki, a beállítására pedig mondjuk az `item.productid = "TVB2100"` utasítás használható.

A díszítő által létrehozott kiolvasófüggvény a privát adatjellemző értékének visszaadására egyszerűen a globális `getattr()` függvényt használja. A beállítófüggvény elvégzi az érvényességvizsgálatot, majd a privát adatjellemző értékéül az új – és érvényes – értéket a `setattr()` függvénnyel állítja be. Valójában a privát adatjellemző csak akkor jön létre, amikor értéket kap.

Amikor a kiolvasó- és beállítófüggvény elkészült, ismét a `setattr()` függvényt használjuk, ezúttal azért, hogy a megadott név (például a `productid`) alapján új, osztályszintű jellemzőt hozzunk létre, amelynek az értéke egy `GenericDescriptor` típusú leíró lesz. Ha ez is megvan, a díszítő a módosított osztályt adja vissza, a `valid_string()` függvény pedig a díszítőfüggvényt.

Felépítését tekintve a `valid_number()` és a `valid_string()` függvény egyforma; az egyetlen különbség a paraméterezésükben és a beállítófüggvény érvényesítést végző kódjában rejlik – így az ismertetésétől ezúttal eltekintünk. (A teljes forráskód megtalálható a `Valid.py` modulban.)

Az utolsó dolog, amit tárgyalnunk kell, a `GenericDescriptor` típus, ami történetesen a legegyszerűbb rész:

```
class GenericDescriptor:
    def __init__(self, getter, setter):
        self.getter = getter
        self.setter = setter
    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.getter(instance)
    def __set__(self, instance, value):
        return self.setter(instance, value)
```

A leírót az egyes jellemzők kiolvasó- és beállítófüggvényeinek tárolására használjuk. A kiolvasással és a beállítással kapcsolatos munkát egyszerűen átadja a megfelelő függvényeknek.

Összefoglalás

A fejezet során lényegesen bővültek a Python eljárásközpontú és objektumközpontú programozásáról szerzett ismereteink, és belekóstoltunk a függvényközpontú programozásba is. Az első részben megismertük a generátorkifejezések létrehozásának módját, és mélyebben tárgyaltuk a generátorkifejezéseket. Azt is megtanultuk, hogy miként lehet dinamikusan modulokat betölteni, miként használhatjuk az ilyen modulok képességeit, és miként történik a dinamikus kódvégrehajtás. Megismertük a `nonlocal` utasítás használatát, és az önhívó függvényeket, és azt is megtanultuk, hogy miként írhatjuk meg saját függvény- és tagfüggvénydíszítőinket, illetve miként írhatunk és használhatunk jegyzetelt függvényeket.

A fejezet második részében az objektumközpontú programozás néhány bonyolultabb területét tanulmányoztuk. Először a jellemzők elérését, illetve a `__getattr__()` különleges szerepű tagfüggvény használatát ismertük meg jobban, majd a funktorokról tanultunk, valamint arról, hogy a funktorok miként használhatók a függvények állapottal való kiegészítésére – ez olyasmi, amit úgy is megoldhatunk, hogy a függvényeket tulajdonságokkal ruházzuk fel, vagy záradékokat használunk. A fejezetben ezekről is volt szó. Megtanultuk, hogy a `with` utasítás miként használatos a környezetkezelők vonatkozásában, és azt is, hogy mi a módja az egyéni környezetkezelők létrehozásának. Minthogy a Python fájlobjektumai egyben környezetkezelők is, a fájlkezelést mostantól a `try with...except` szerkezettel végezzük, biztosítva, hogy a megnyitott fájlokat a `finally` blokkok használata nélkül is lezárjuk.

A második részt további, az objektumközpontú programozás elmélyült ismeretét igénylő képességek ismertetésével folytattuk. Először a leírókkal ismerkedtünk meg, amelyek sokféle módon használhatók, és amelyek a Python alapvető díszítőinek – többek között a `@property` és a `@classmethod` díszítőkre gondolunk – használhatóságát nagyban emelik. Megismertük az egyéni leírók létrehozásának módját, és három, meglehetősen különböző példán keresztül barátkoztunk meg a használatukkal. Ezt követően az osztály-szintű díszítőket tanulmányoztuk, és láttuk, hogy miként módosítható velük egy osztály szinte ugyanúgy, ahogy a függvénydíszítők módosítanak egy függvényt.

A szakasz utolsó három részében a Python elvont alaposztályairól, a többszörös öröklésről és a metaosztályokról tanultunk. Megtudtuk, hogy a saját osztályaink miként egyezethetők össze a Python elvont alaposztályaival, és azt is, hogy miként hozhatók létre egyéni elvont alaposztályok. Azt is láttuk, hogy a többszörös örökléssel különböző osztályok képességei egyetlen osztályban egyesíthetők. A metaosztályok tárgyalása során fény derült arra is, hogy egy osztály – nem egy objektumpéldány – létrejöttekor és előkészítésekor is elvégeztethetünk a programunkkal egyet-mást.

Az utolsó előtti részben néhány olyan függvényt és modult ismertünk meg, amelyekkel a Python a függvényközpontú programozást segíti. Megtanultuk a megfeleltetés, a szűrés és az elhagyás, vagyis a függvényközpontú programozás három legközönségesebb műveletének használatát. A részleges függvények létrehozásának módját is megismertük.

Végül az utolsó részben megtanultunk olyan eljárást kialakítani, amely – az osztályszintű díszítők és a leírók együttes használatával – jól használható és rugalmas megoldást kínál a jellemzőértékek érvényességének ellenőrzésére.

A fejezettel lezárult magának a Python nyelvnek a tárgyalása. A mostani, illetve az előző fejezetben nem minden képességével ismerkedtünk meg a nyelvnek, de csak a homályos és ritkán használt területeket nem érintettük. A következő fejezetek egyike sem ismertet nyelvi újdonságokat, bár az alapkönyvtár moduljai közül mindegyikben találkozunk eleddig ismeretlennel, és néhányban továbblépünk a már megismert módszerek használatát illetőleg is. Mindezen felül a következő fejezetekben bemutatásra váró programoknál nem alkalmazzuk az előzőekben használt megszorításokat – eddig ugyanis vigyáztunk, hogy csak a már megismert nyelvi eszközökkel dolgozzunk fel egy problémát –, vagyis ezekben a fejezetekben találjuk meg a nyelv sajátságait leginkább kiaknázó példákat.

Gyakorlatok

Az alábbi három gyakorlat egyike sem jár hosszú kódrészletek megírásával – és egyikükről sem állítjuk, hogy egyszerű.

1. Készítsünk másolatot a `magic-numbers.py` programról, és töröljük belőle a `get_function()` függvényeit, valamint egy kivételével a `load_modules()` függvényeit is. Vegyünk fel egy új `GetFunction` funktorosztályt, amely két gyorsítótárral bír. Az egyikben a megtalált, a másikban a fellelhetetlen függvényeket tároljuk, így kerülve el, hogy egy olyan modulban keressünk újra meg újra egy adott függvényt, amelyikben az nincs meg. A `main()` függvényben mindössze annyi módosítást kell elvégeznünk, hogy a ciklus előtt elhelyezünk egy `get_function = GetFunction()` utasítást, és egy `with` utasítást használunk a `finally` blokk helyett. Azt is ellenőrizzük, hogy a modul függvényei meghívhatók-e, mégpedig a `hasattr()` függvény helyett a `collections.Callable` osztállyal. Az osztály nagyjából húsz sorban megírható. A megoldást a `magic-numbers_ans.py` fájlban találjuk.
2. Hozzunk létre egy új modult, és adjunk meg benne három függvényt. Az `is_ascii()` függvény `True` értékkel tér vissza, ha a neki átadott karakterlánc minden karakterének kódja 127-nél kisebb. Az `is_ascii_punctuation()` függvény akkor tér vissza `True` értékkel, ha valamennyi karakter megtalálható a `string.punctuation` karakterláncban. Az `is_ascii_printable()` függvény visszatérési értéke pedig akkor `True`, ha minden karakter benne van

a `string.printable` karakterláncban. Szerkezetét tekintve az utolsó két függvény egyforma. Minden függvényt a `lambda` utasítás használatával hozzunk létre; ez függvényközpontú megközelítéssel két sorban megoldható. Mindenképp tartozzon mindegyikhez docstring és doctest. A dokumentációs teszteket a modul futassa is le. A függvények egyenként három-öt sor hosszúak, így a modul egésze – a dokumentációs tesztekkel együtt – a huszonöt sort sem éri el. A megoldást az `Ascii.py` fájlban találjuk.

3. Hozzunk létre egy új modult, és határozzuk meg benne az `Atomic` környezetkezelő osztályt. Az osztály működése mindenben egyezzen meg a fejezetben bemutatott `AtomicList` osztályéval, de nem csak listával, hanem bármilyen változtatható gyűjteményes adattípussal működnie kell. A tároló megfelelőségét az `__init__()` tagfüggvénynek kell ellenőriznie, és nem azt kell jeleznie, hogy sekély vagy mély másolás szükséges, hanem ennek fényében a `self.copy` jellemzőnek kell megfelelő függvényt értékül adnia. A másolást végző függvényt az `__enter__()` tagfüggvényben hívjuk meg. Az `__exit__()` is bővül, mivel a halmazok és a szótárak tartalmának cseréje nem úgy történik, mint a listáké – és értékadást sem használhatunk, mert az az eredeti tárolón nem változtatna. Az osztály nagyjából harminc sorban megírható, bár dokumentációs tesztekre is szükség lenne. A megoldást – dokumentációs tesztekkel együtt – a nagyjából százötven soros `Atomic.py` fájlban találjuk.

9

Folyamatok és szálak

- Feladatok végrehajtása folyamatokkal
- Feladatok végrehajtása szálakkal

Amióta a többmagos processzorok nemhogy kivételt nem jelentenek, de egyenesen alapvetőnek számítanak, a programozót még inkább kísérti az az immáron gyakorlati érvekkel is könnyebben alátámasztható igény, hogy az adatok feldolgozásának terhét megossza, a többmagos gépeknél így érve el a lehető legnagyobb teljesítményt. A munkateher megosztásának alapvetően két módja kínálkozik. Az egyik a több folyamat, a másik a több szál használata. A fejezet mindkét megközelítési mód használatát bemutatja.

A több folyamat használatának, azaz különálló programok futtatásának megvan az az előnye, hogy minden folyamat egymástól függetlenül fut. Így az erőforrásokért való versengés kezelésével járó minden terhet az operációs rendszerre hagyunk. A hátrányos oldal, hogy a kommunikáció és az adatmegosztás könnyen válik kényelmetlenné a főprogram, illetve az általa létrehozott különálló folyamat között. Unix rendszereken az exec-fork megosztás megoldást jelenthet, de a rendszerfüggetlen programok esetében más megoldást kell találnunk. A legegyszerűbb és az itt bemutatott egyetlen eljárás az, ha a főprogram látja el az általa létrehozott folyamatokat adatokkal, de a folyamatok a kimenetet önállóan állítják elő. Rugalmasabb, a kétirányú kommunikációt lényegesen leegyszerűsítő megközelítés a hálózat használata. Természetesen számos esetben nincs szükség efféle kommunikációra, mindössze arra, hogy egy-két másik programot futtassunk a folyamatok vezénylését végző főprogram segítségével.

Ha nem akarjuk a munkát független folyamatok között elosztani, készíthetünk olyan, többszálú programot, amely a feladatokat több önálló végrehajtású szál között osztja szét. Ennek az eljárásnak az az előnye, hogy a kommunikáció pusztán az adatok megosztásával is megvalósul – feltételezve, hogy az adathoz egy adott pillanatban egyetlen szál fér hozzá –, de az erőforrások párhuzamos használatának kezelésével járó teher a programozó nyakában marad. A Python remekül támogatja a többszálú programok készítését – az elvégzendő munka a lehető legkevesebb. A többszálú programok mindazonáltal lényegükből fakadóan összetettebbek egyszálú társaiknál, és mind a program kialakításánál, mind a karbantartás során lényegesen nagyobb odafigyelést kívánnak meg.

A fejezet első részében két rövid programot fogunk megírni. Az első programot a felhasználó indítja el, a másodikat pedig az első program. A második program annyiszor indul el, ahányszor szükség van erre. A második rész elején megismerkedünk a többszálú programok írásának alapjaival, ezt követően pedig megírunk egy olyan többszálú programot, amely egyenértékű az előzőleg megírt két program együttesével. Tesszük mindezt úgy, hogy rávilágítunk a több folyamat és a több szál használata közötti főbb különbségekre. Végezetül áttekintjük egy másik, az előzőnél kifinomultabb többszálú program működését, amely kiosztja az elvégzendő feladatokat, és az eredményeket is összegyűjti.

Feladatok végrehajtása folyamatokkal

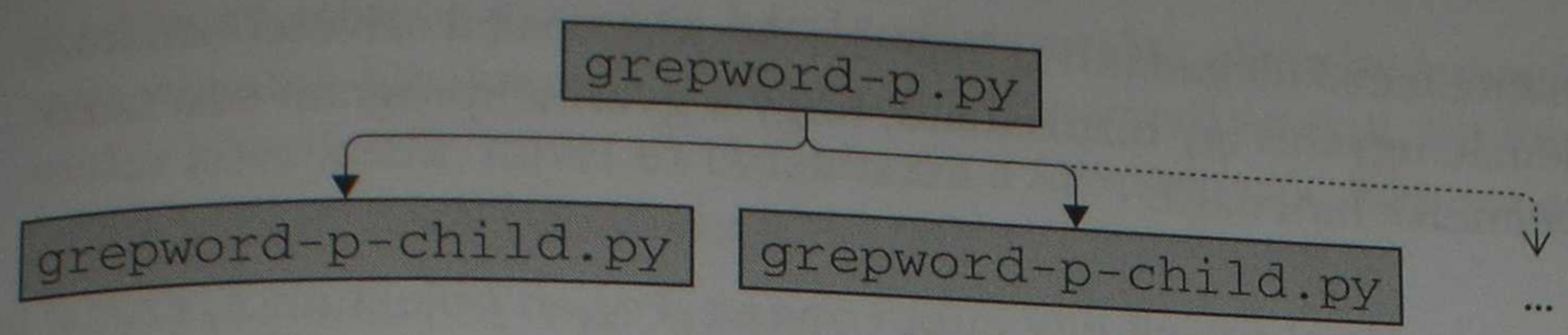
Vannak helyzetek, amikor rendelkezésünkre állnak az elvégzendő feladatot ellátni képes programok, de automatizálni szeretnénk a futtatásukat. Ez megoldható a Python `subprocess` moduljával, amelyben megtaláljuk a más programok futtatásához és a megfelelőnek talált parancssori paraméterek átadásához szükséges eszközöket, sőt azokat is, amelyek csővezetékek használatával biztosítják a folyamatok közötti kommunikációt. Már láttunk egy igen egyszerű példát, mégpedig az 5. fejezetben, amikor a `subprocess.call()` függvényt használtuk a konzolablak operációs rendszertől függő letakarítására. Ugyanezek az eszközök használhatók „szülő-gyermek" programpárok megalkotására is. Ilyen esetekben a felhasználó futtatja a szülőprogramot, amely annyi példányban indít különböző feladatokat ellátó gyermekprogramokat, ahányszor azt szükségesnek ítéli. A fenti megközelítési lehetőséget tárgyaljuk ebben a részben.

A 3. fejezetben bemutattuk a `grepword.py` nevű, igen egyszerű programot, amely a parancssorba írt szót keresi meg a parancssorban a szó után álló fájlokban. Ebben a részben a program egy kifinomultabb változatát írjuk meg, amely alkönyvtárakban is képes a beolvasandó fájlok megtalálására, és a munkát tetszőleges számú önálló gyermekfolyamat között képes szétosztani. A kimenet egy elérési útvonalakat is tartalmazó fájlnévlista, a megadott szót tartalmazó fájlok neveivel.

A szülőprogram neve `grepword-p.py`, a gyermekprogramé pedig `grepword-p-child.py`. A programok futása alatt fennálló kapcsolatokat a 9.1. ábra szemlélteti.

A `grepword-p.py` program szíve a `main()` függvényben rejlik, amelyet három részben tekintünk át:

```
def main():
    child = os.path.join(os.path.dirname(__file__),
                         "grepword-p-child.py")
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    files_per_process = len(filelist) // opts.count
    start, end = 0, files_per_process + (len(filelist) % opts.count)
    number = 1
```



9.1. ábra

Szülőprogram és gyermekprogramok

A munkát a gyermekprogram nevének kiolvasásával kezdjük, majd a felhasználó által megadott parancssori paraméterek feldolgozása következik. A `parse_options()` függvény az `optparse` modult használja; visszatérési értéke pedig az `opts` nevű nevesített rendezett sorozat, amely jelzi, hogy a programnak vizsgálnia kell-e az alkönyvtárakat, és tudatja a használandó folyamatok számát – hét az alapértelmezett, és a program legfeljebb húszat indíthat. A visszaadott értékben megtaláljuk a keresett szót, illetve a parancssorban megadott könyvtár- és fájlnevek listáját is. A `get_files()` függvény visszatérési értéke a beolvasandó fájlok listája.

Ha megvan minden, a feladat elvégzéséhez szükséges információ, kiszámítjuk, hogy az egyes folyamatoknak hány fájlt adjunk ki feldolgozásra. A `start` és az `end` változó jelzi a `filelist` listában annak a résznek az elejét és végét, amelyet a következő gyerekfolyamatnak adunk oda. A fájlok száma rendszerint nem lesz pontos többszöröse a folyamatok számának, a maradékot tehát az első folyamat kapja. A `number` változó egyetlen szerepe a hibakeresés segítése; ennek segítségével láthatjuk, hogy a kimenet egyes sorai melyik folyamattól származnak.

```
pipes = []
while start < len(filelist):
    command = [sys.executable, child]
    if opts.debug:
        command.append(str(number))
    pipe = subprocess.Popen(command, stdin=subprocess.PIPE)
    pipes.append(pipe)
    pipe.stdin.write(word.encode("utf8") + b"\n")
    for filename in filelist[start:end]:
        pipe.stdin.write(filename.encode("utf8") + b"\n")
    pipe.stdin.close()
    number += 1
    start, end = end, end + files_per_process
```

A `filelist` minden `start:end` közötti részéhez létrehozunk egy parancslistát, amely a Python parancsértelmezőből (a `sys.executable` változóban találjuk), abból a gyermekprogramból, amelyet a Pythonnal akarunk elindíttatni és a parancssori paraméterekből – esetünkben, hibakereséskor csak a gyermekfolyamatok számából – áll. Ha a gyermekprogram megfelelő shebang-sorral vagy fájltársítással rendelkezik, lehetne ő az első a listában, és

nem kellene a Python-értelmező megadásával bajlódni. Mindazonáltal szívesebben közelítjük meg így a problémát, ugyanis így biztosítható, hogy a gyermekprogram és a szülő ugyanazt a Python-értelmezőt használja.

Amikor elkészült a parancs, létrehozunk egy `subprocess.Popen` objektumot, ezúttal úgy, hogy tudjunk írni a folyamat szabványos bemenetére. (Hasonló kulcsszó-paraméter megadásával a folyamat szabványos kimenetének olvasására is van lehetőség.) Ezután leírjuk a keresett szót, majd egy újsor karakter, végül a fájllista megfelelő részében lévő fájlnevek következnek. A `subprocess` modul karakterláncok helyett bájtokat ír és olvas, vagyis a leírt karakterláncokat kódolnunk kell (a beolvasott bájtokat pedig visszafejtenünk), mégpedig megfelelő kódolás használatával. Ezúttal az UTF-8 kódolást választottuk. Ha a gyermekfolyamatnak átadtuk a fájlok listáját, lezárjuk a hozzá tartozó szabványos bemenetet, és továbblépünk.

Nem feltétlenül kötelező minden folyamat objektumhivatkozását megtartanunk – a `pipe` változó a ciklus minden ismétlődésekor az új `subprocess.Popen` osztályú objektumra hivatkozik –, hiszen mindegyik önállóan fut. Mégis mindegyiket elhelyezzük egy listában, így ugyanis módunkban áll megszakítani egy-egy folyamatot. Az eredményeket sem gyűjtjük be, inkább minden folyamattal a megfelelő időben kiíratjuk az eredményt a folyamat saját konzoljára. Ez azzal jár, hogy a különböző folyamatok kimenete átfedésben lehet. (A gyakorlatoknál megismerjük az átfedések elkerülésének módját.)

```
while pipes:
    pipe = pipes.pop()
    pipe.wait()
```

Amikor minden folyamat elindult, megvárjuk, amíg az egyes folyamatok befejeződnek. Ez nem feltétlenül szükséges, de a Unix-szerű rendszereken így lehetünk biztosak abban, hogy az összes folyamat lefutása után visszakapjuk a konzol parancsjelét – egyéb esetben meg kell nyomnunk az ENTER billentyűt a folyamatok lefutása után. A várakozás további előnye, hogy ha megszakítjuk a főprogram futását – mondjuk a CTRL+C billentyűk lenyomásával –, azok a folyamatok is megszakadnak, amelyek ekkor még futnak. A leállás egy el nem kapott `KeyboardInterrupt` kivételt eredményez. Ha nem várakoznánk, a főprogram befejeződne – vagyis nem lenne megszakítható –, a gyermekfolyamatok pedig tovább futnának, egészen addig, amíg a `kill` programmal vagy a feladatkezelővel véget nem vetnénk a futásuknak.

A megjegyzések és a modulok betöltését végző sorok kivételével következzék a teljes `grepword-p-child.py` program, két részletben.

```
BLOCK_SIZE = 8000

number = "{0}: ".format(sys.argv[1]) if len(sys.argv) == 2 else ""
word = sys.stdin.readline().rstrip()
```

A program először a number karakterlánc értékét állítja be a megfelelő számra, illetve ha nem végzünk hibakeresést, üres karakterláncot ad értékül. Ezt követően az első sor beolvasása következik, mivel ez tartalmazza a keresett szót. Ez a sor és az ezt követők karakterlánccként lesznek beolvasva.

```
for filename in sys.stdin:
    filename = filename.rstrip()
    previous = ""
    try:
        with open(filename, "rb") as fh:
            while True:
                current = fh.read(BLOCK_SIZE)
                if not current:
                    break
                current = current.decode("utf8", "ignore")
                if (word in current or
                    word in previous[-len(word):] +
                            current[:len(word)]):
                    print("{0}{1}".format(number, filename))
                    break
                if len(current) != BLOCK_SIZE:
                    break
                previous = current
    except EnvironmentError as err:
        print("{0}{1}".format(number, err))
```

Az első sort a fájlnevek követik, az elérési utakkal együtt. Minden sornál megnyitjuk a megfelelő fájlt, beolvassuk, és ha tartalmazza a keresett szót, kiírjuk a nevét. Lehetséges, hogy pár fájl igen méretes, ami problémákhoz is vezethet, főleg ha húsz gyermekfolyamat fut egyszerre, és mindegyik nagy fájlok beolvasását végzi. A problémán úgy igyekszünk úrrá lenni, hogy a fájlok beolvasását blokkokban végezzük, az előzőleg beolvasott blokkot megtartva – így vigyázunk arra, hogy az olyan esetek se sikkadjanak el, amikor a szó egyetlen előfordulása blokkhatárra esik. A blokkonkénti beolvasás további előnye, hogy amennyiben a keresett szót már a fájl eleje felé megtaláljuk, nem kell végigolvasnunk a fájlt – hiszen csak annyi érdekel bennünket, hogy a szó benne van-e az adott állományban, vagy sem.

Karakterkódolás 86

A fájlok beolvasása bináris módban történik, vagyis mielőtt elolvasnánk, minden blokkot karakterlánccá kell alakítanunk – a keresett szó ugyanis karakterlánc típusú. Feltételezzük, hogy minden állomány az UTF-8 karakterkódolást használja, bár minden valószínűség szerint lesznek ettől eltérő esetek. Egy kifinomultabb program megpróbálkozna a használt karakterkódolás meghatározásával, majd bezárná, és újra megnyitná a fájlt, ezúttal már a helyes karakterkódolást használva. Ahogyan a 2. fejezetben már említettük, a Python Package Index webhelyén (`pypi.python.org/pypi`) legalább két olyan Python-csomag található, amely a fájlok karakterkódolásának automatikus megállapítását célozza. (Kísértést érezhetünk arra, hogy

a keresett szót `bytes` típusú objektummá alakítsuk, majd bájtsorozatok között végezzük el az összehasonlítást, de ez nem megbízható módszer, vannak ugyanis olyan karakterek, amelyeknek több érvényes UTF-8 kódolású ábrázolásuk van.)

A `subprocess` modul lényegesen több képességgel rendelkezik annál, mint amit itt felhasználtunk. Képes például a parancshéj fordított aposztrófjának (`) és csővezetékének, illetve az `os.system()` függvénynek megfelelő működés ellátására, valamint függvények előállítására.

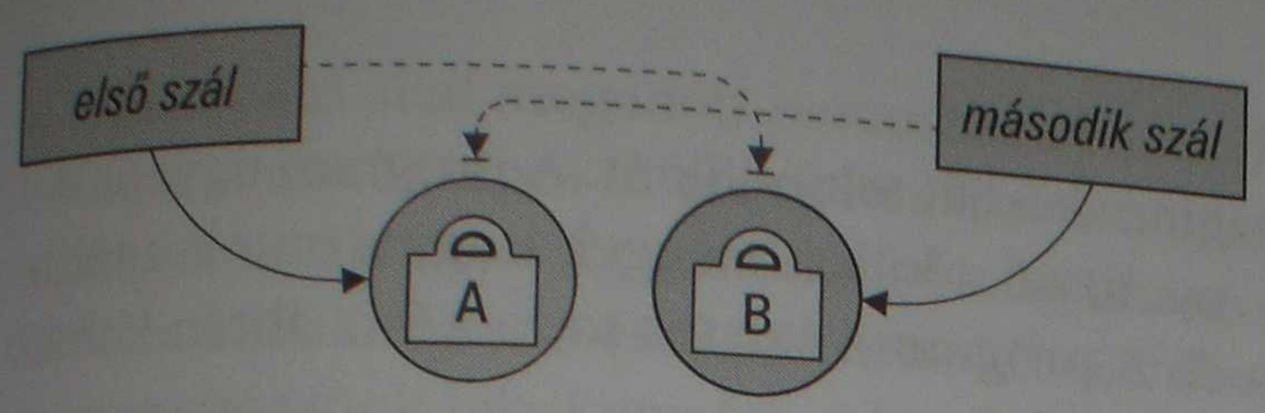
A következő részben megismerkedünk a `grepword-p.py` program többszálú változatával, és lesz módunk összehasonlítani a szülő-gyermek folyamatokon alapulóval. Áttekintjük egy olyan, kifinomultabb többszálú program működését is, amelyik kiosztja a feladatokat, majd összeszedi az eredményeket, így szabályozva még pontosabban a kimenetet.

Feladatok végrehajtása szálakkal

A Pythonban meglehetősen egyszerű egy kettő vagy akár több végrehajtási szálat tartalmazó program írása. A dolog onnantól válik bonyolulttá, ha azt is szeretnénk, hogy az egyes szálak osztozzanak az adatokon. Képzeljük el, hogy két szál osztozik egy listán. Az egyik szál a lista bejárásához fog a `for x in L` ciklussal, és amikor a közepe felé jár, egy másik szál töröl pár elemet a listából. Ez legjobb esetben is a program gyanús elhalálozásához vezet – a legrosszabb eset pedig az, hogy hamis eredményt kapunk.

Az egyik legáltalánosabban elterjedt megoldás valamiféle zárolási rendszer használata. Például, az első szál zárolja a listát, és csak *ez után* kezdi el bejárni. Ilyenkor a zárolás miatt egyetlen más szál sem fér a listához. A valóságban a helyzet messze nem ennyire egyszerű. Az adat és az azt zároló zár közötti kapcsolat ugyanis kizárólag a mi képzeletünk szülötte. Ha egy szál zárol valamit, és egy másik szál megpróbálja ugyanazt a dolgot zárolni, a második szál egészen addig kénytelen várni, amíg az első szál meg nem szünteti a zárolást. Ha az adatok hozzáférhetőségének szabályozásakor a zárolási eljárást is figyelembe vesszük, biztosítható, hogy a megosztott adathoz egy adott időben csak egy adott szál férjen hozzá – még akkor is, ha a védelem csak közvetett.

A zárolással kapcsolatos a holtpont kialakulásának kockázata. Tegyük fel, hogy az *első szál* egy a megosztott adathoz való hozzáférés okán létrehozza az *A* zárat, és amikor az létrejön, megpróbál a *b* megosztott adathoz is hozzáférni. A *B* zár létrehozására azonban képtelen, ugyanis azt időközben egy *második szál* már létrehozta, hogy hozzáférhessen a *b* megosztott adathoz. A második szál most éppen az *A* zár létrehozásával próbálkozik, hogy hozzáférjen az a megosztott adathoz. Vagyis az *első szál* birtokolja az *A* zárat, és próbálkozik a *B* zár kialakításával, miközben a második szál birtokolja a *B* zárat, és próbálkozik az *A* zár kialakításával. Végül mindkét szál várakozásra kényszerül, a program pedig holtpontra jut – ahogy a 9.2. ábra is mutatja.



9.2. ábra

Holtpont: legalább két szál várakozik arra, hogy a másik által zárolt adathoz hozzáférhessen

Bár a holtpont lerajzolása és elképzelése egyszerű, a gyakorlatban a holtpontok komoly gondokat okozhatnak – ugyanis a létük nem mindig nyilvánvaló. Néhány többszálúságot kezelő programkönyvtár figyelmeztetni próbálja a programozót a lehetséges holtpontokra, de a holtpontok elkerülése emberi figyelmet és beavatkozást igényel.

A holtpontok elkerülésének egyik egyszerű, ámde hatékony módja, ha felállítunk egy rendet, amely megadja a zárolások sorrendjét. Például, ha a rend azt mondja ki, hogy az *A* zárolásnak mindig meg kell előznie a *B* zárolást, és éppen meg akarjuk valósítani a *B* zárolást, a rend miatt előbb az *A* zárolást kell megvalósítanunk. Ezzel a módszerrel biztosítható, hogy a fent leírt holtpont sohasem alakul ki, mivel mindkét szál az *A* zárolást akarja elvégezni, és amelyiknek sikerül, csak az léphet tovább a *B* zároláshoz. Mindez remekül működik, amíg valaki el nem felejtkezik a rend betartásáról.

A zárolás további problémája, hogy ha több szál várakozik egy zárolásnál, akkor csak állnak, és semmi hasznosat nem végeznek. Ennek hatása jelentősen csökkenthető, ha a kódolási szokásainkat úgy változtatjuk meg, hogy csak minimális munkát végeztessünk a programmal, miközben valamit zárolva tartunk.

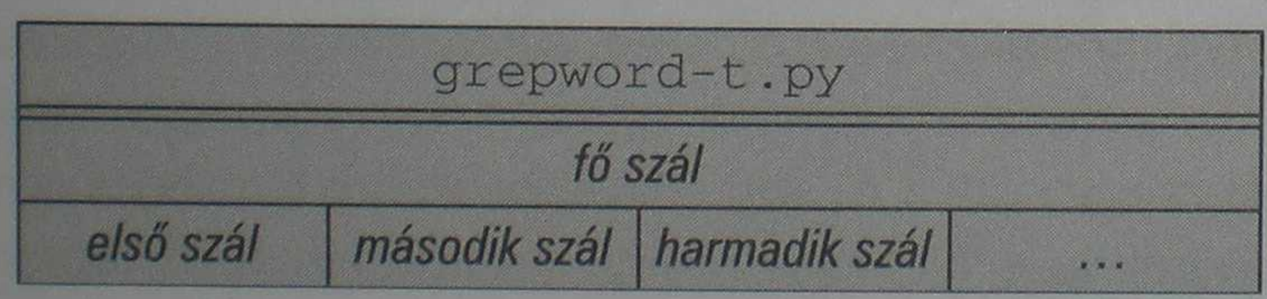
Minden Python-programban van legalább egy szál, a fő szál. További szálak létrehozása előtt be kell töltenünk a `threading` modult, amellyel aztán annyi szálat hozhatunk létre, amennyit csak akarunk. Szálak létrehozására két mód kínálkozik. Meghívhatjuk a `threading.Thread()` tagfüggvényt, valamilyen hívható objektumot adva át neki, illetve alosztályt alakíthatunk ki a `threading.Thread` osztály alapján – a fejezetben mindkét megközelítést bemutatjuk. Az alosztály létrehozása a rugalmasabb és egyszerűbb módszer. Az alosztályok lecserélhetik az `__init()__` tagfüggvényt (ebben az esetben az alaposztály-beli megvalósítást *kell* meghívniuk), és le *kell* cserélniük a `run()` tagfüggvényt – ez a tagfüggvény felelős a szál munkájának elvégzéséért. A kódunk *soha* nem hívhatja meg a `run()` tagfüggvényt – a szál indítása a `start()` tagfüggvény meghívásával történik, és ha készen áll, a `start()` tagfüggvény hívja majd meg a `run()` tagfüggvényt. A `threading.Thread` osztály semelyik másik tagfüggvénye nem cserélhető le, ugyanakkor lehetőségünk van új tagfüggvények megadására.

Példa: Többszálú szókereső program

Ebben a részben a `grepword-t.py` program kódját tekintjük át. A program ugyanazt a feladatot látja el, mint a `grepword-p.py`, azzal a különbséggel, hogy a munkát több folyamat helyett több szál között osztja el. A program vázlatos rajza a 9.3. ábrán látható.

A program különösen érdekes tulajdonsága, hogy láthatólag egyáltalán nem használ zárolást. Ez azért lehetséges, mert a megosztott adat egy fájllista, amelynek a tárolására egy `queue.Queue` osztályú objektumot használunk, a `queue.Queue` osztály különlegessége pedig az, hogy mindenféle zárolást belül végez el, vagyis minden olyan alkalommal, amikor új elemek elhelyezését vagy meglévők törlését akarjuk elvégezni, az objektum maga fogja *sorba állítani* az eléréseket. A többszálúság szempontjából az elérések sorba állítása azzal jár, hogy egyidejűleg biztosan nem fogja több szál is elérni az adatot. A `queue.Queue` osztály használatának további előnye, hogy nem nekünk kell megoldanunk a munka elosztását – egyszerűen elhelyezzük a sorban a feladat egyes részeit, és a munkát végző szálak majd jelentkeznek érte, ha az előzővel elkészültek.

A `queue.Queue` osztály „elsőnek be – elsőnek ki" (first in, first out, FIFO) alapon dolgozik. A `queue` modulban találjuk a `queue.LifoQueue` osztályt, amely „utolsónak be – elsőnek ki" (last in, first out, LIFO) módszerrel végzi a feladatot, illetve a `queue.PriorityQueue` (prioritási sor) osztályt is, mely objektumainak kételemű (prioritás, elem) rendezett sorozatokat adunk át, és a feldolgozás a legalacsonyabb prioritású elemmel kezdődik. Minden sorféleség létrehozásakor megadható a maximális méret – ha ezt elérjük, egészen addig nem helyezhető új elem a sorba, amíg néhányat el nem távolítunk.



9.3. ábra
Többszálú program

A `grepword-t.py` program áttekintését három részletben végezzük el. Elsőként a teljes `main()` függvény következik:

```
def main():
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    work_queue = queue.Queue()
    for i in range(opts.count):
        number = "{0}: ".format(i + 1) if opts.debug else ""
        worker = Worker(work_queue, word, number)
        worker.daemon = True
        worker.start()
    for filename in filelist:
        work_queue.put(filename)
    work_queue.join()
```

A felhasználó által megadott paraméterek és a fájllista bekérése ugyanúgy történik, mint eddig. Ha a szükséges információ a birtokunkban van, létrehozunk egy `queue.Queue` objektumot, a ciklus pedig annyiszor fut le, ahány szálat szeretnénk: alapértelmezés szerint hétszer. Hibakeresési céllal minden szál esetében előkészítjük a `number` karakterláncot – ha nem végzünk hibakeresést, a karakterlánc üres –, majd létrehozunk egy `Worker` objektumpéldányt (a `Worker` osztály alaposztálya a `threading.Thread` osztály), amelynek `daemon` tulajdonságához rövidesen visszatérünk. Ennek végeztével elindítjuk a szálat, bár ebben a pillanatban még nincs feladata, hiszen a munkalista még üres. Így aztán a szál azonnal meg is áll, és nem is indul tovább, amíg feladatot nem kap.

Most, hogy minden szál munkára készen áll, bejárjuk a fájlokat, és elhelyezzük őket a munkasorban. Amint az első fájl bekerül a sorba, valamelyik szál lecsap rá, és dolgozni kezd rajta – és ez így folytatódik, amíg van feladat nélkül lévő szál. Ahogy egy szál befejezi a munkát egy fájlon, másikat vesz magának a sorból, egészen addig, amíg a feldolgozandó fájlok el nem fogynak.

Figyeljük meg, hogy a programunk ebben a tekintetben különbözik a `grepword-p.py` programtól, ahol nekünk kellett átadnunk a fájllista szeleteit az egyes gyermekfolyamatoknak, amelyeknek az indulása – és így a listák átvétele is – egymás után történt. A szálak használata ekképpen a nagyobb hatékonyság lehetőségét rejti magában. Példának okáért, ha az első öt fájl nagyon nagy, a többi pedig kicsi, akkor – minthogy minden szál egyszerre csak egy feladatot kap meg – a nagy fájlok különböző szálakhoz kerülnek, és így megvalósul a munka egyenlő elosztása. A többfolyamatos megközelítés alkalmazásával – amit a `grepword-p.py` program esetében is tettünk – minden nagy fájl az első folyamathoz kerülne, a kicsik meg a többihez, és végül az első folyamat végezné a munka túlnyomó részét, a többiek meg gyorsan végeznének, és alig-alig dolgoznának.

Amíg van futó szál, a programunk nem áll le. Ez azért baj, mert a szálak technikai értelemben tovább futnak akkor is, ha már befejezték a feladatukat. A megoldás az, ha a szálakat démonokká alakítjuk, ami arra jó, hogy a programunk le tud állni, ha már csak olyan szálai vannak, amelyek nem démonok. A fő szál nem démon, vagyis amikor a fő szál befejezi a futását, a program szépen megszakítja a démonszálak futását, és végül saját maga is leáll. Természetesen ez esetleg az előzőekben vázolt problémával ellentétesen rossz helyzethez vezethet. Így ha a munkát végző szálak elindultak, vigyáznunk kell arra, hogy a fő szál is tovább fusson – egészen addig, amíg van elvégzendő feladat. Ezt a problémát a `queue.Queue.join()` tagfüggvény meghívásával oldhatjuk meg: a tagfüggvény addig várakozik, amíg ki nem ürül a sor. Íme a `Worker` alosztály eleje:

```
class Worker(threading.Thread):
    def __init__(self, work_queue, word, number):
        super().__init__()
        self.work_queue = work_queue
        self.word = word
        self.number = number
```

```
    def run(self):
        while True:
            try:
                filename = self.work_queue.get()
                self.process(filename)
            finally:
                self.work_queue.task_done()
```

Az `__init__()` tagfüggvénynek az ősosztály `__init__()` tagfüggvényét kell meghívnia. A munkasor egyetlen `queue.Queue` objektum; ezen osztozik valamennyi szál.

A `run()` tagfüggvényt végtelen cikluskénti írtuk meg. Ez a démonszálak esetében általánosan elterjedt módszer, és azért járunk el így, mert nem tudjuk, hogy a szálnak hány fájlt kell feldolgoznia. Minden ismétlődés alkalmával a `queue.Queue.get()` tagfüggvénnyel kapjuk meg a következő feldolgozandó fájlt. Ha a sor üres, a hívásnak várakoznia kell, és azért nem kell zárolással ügyelnünk rá, mert a `queue.Queue` objektumok ezt automatikusan elvégzik helyettünk. Ha van feldolgozható fájlunk, fel is dolgozzuk, de utána meg kell mondanunk a sornak, hogy az adott feladatot befejeztük – a `queue.Queue.join()` tagfüggvény helyes működésének elengedhetetlen feltétele, hogy a `queue.Queue.task_done()` tagfüggvény meghívása megtörténjen.

A `process()` függvényt nem mutattuk be, ugyanis a `previous = ""` sortól a kód végéig megegyezik a `grepword-p-child.py` fájlban lévővel – a `def` utasítást tartalmazó sortól eltekintve.

Megjegyeznénk még egy utolsó dolgot: a könyv példái között megtaláljuk a `grepword-m.py` nevű programot, amely majdnem teljesen megegyezik a `grepword-t.py` nevű, az imént áttekintett társával, azzal a különbséggel, hogy a `threading` modul helyett a `multiprocessing` modult használja. A kódban mindössze három helyen van eltérés. Először is a `queue` és a `threading` modul helyett a `multiprocessing` modult töltjük be. Másodszor, a `Worker` osztályt a `threading.Thread` osztály helyett a `multiprocessing.Process` osztályból származtatjuk. Harmadszor pedig, a munkasor nem `queue.Queue`, hanem `processing.JoinableQueue` objektumként jön létre.

A `multiprocessing` modul a szálak használatához hasonló viselkedést elágazással valósítja meg az elágazást támogató (Unix) rendszereken, és gyermekfolyamatokkal az elágazást nem támogató (Windows) rendszereken. Így nem kell mindig zárolást használnunk, a folyamataink pedig annyi processzormagon fognak futni, ahány az operációs rendszer rendelkezésére áll. A csomag a folyamatok közti adatátadásnak számos módját kínálja – többek között olyan sor is található benne, amely ahhoz hasonlóan osztja szét a feladatokat a folyamatok között, ahogy a `queue.Queue` objektumok a szálakat látják el tennivalóval.

A `multiprocessing` modulon alapuló változat legfőbb előnye, hogy a szálakkal dolgozó változatnál a többmagos gépek esetében elvileg gyorsabb, ugyanis a folyamatait képes az összes elérhető magon futtatni. Tekintsük mindezt annak a fényében, hogy a szabványos Python-értelmező (C nyelven írták, így CPython néven is ismert) egy GIL (Global Interpreter Lock) nevű zárolási eljárást használ, ami annyit tesz, hogy egyszerre csak egy szál hajthat végre Python-kódot. Ez a megszorítás egy megvalósítási részletkérdés, és más Python-értelmezők esetében – amilyen például a Jython* – nem szükségszerűen igaz.

Példa: A többször előforduló fájlokat kereső program többszálú változata

Első pillantásra a többszálúságot bemutató második példaprogram felépítése az előzőhöz hasonló, maga a program ugyanakkor sok szempontból lényegesen kifinomultabb. Két sort használ, az egyik a munkasor, a másikban pedig az eredményeket tárolja. Önálló eredményfeldolgozó szállal bír, így oldva meg azt, hogy az eredményeket az elérhetővé válásukat követően azonnal megjeleníthesse. A program egyaránt igénybe veszi a `threading.Thread` ősosztályból származó alosztályt, és a `threading.Thread()` tagfüggvény hívását, illetve zárolást használ a megosztott adathoz (egy `dict` típusú objektumhoz) történő hozzáférések sorba állítására.

A `findduplicates-t.py` program az 5. fejezetben megismert `finddup.py` program továbbfejlesztett változata. Bejárja az aktuális könyvtár – illetve a megadott elérési útvonal – fájljait, illetve az alkönyvtárakat is. A `finddup.py` programmal megegyező módon összehasonlítja egymással az egyező nevű fájlok hosszát. Az egyező nevű és méretű állományoknál MD5 (Message Digest) algoritmussal megállapítja, hogy a fájlok tényleg megegyeznek-e, és az egyezéseket a képernyőre írja.

Először a `main()` függvényt tekintjük át, mégpedig négy részletben.

```
def main():
    opts, path = parse_options()
    data = collections.defaultdict(list)
    for root, dirs, files in os.walk(path):
        for filename in files:
            fullname = os.path.join(root, filename)
            try:
                key = (os.path.getsize(fullname), filename)
            except EnvironmentError:
                continue
            if key[0] == 0:
                continue
            data[key].append(fullname)
```

* Annak rövid magyarázatát, hogy a CPython miért a GIL eljárást használja, az alábbi helyeken találjuk meg: www.python.org/doc/faq/library/#can-t-we-get-rid-of-the-global-interpreter-lock és docs.python.org/api/threads.html.

A `data` alapértelmező szótár minden kulcsa egy kételemű (méret, fájlnév) rendezett sorozat, amelyben a fájlnév nem tartalmazza az elérési utat. Az értékek pedig fájlnévlisták, mégpedig az elérési utakat tartalmazó fájlnevekkel. Minden olyan elemről, amelynek az értéklistája egynél több nevet tartalmaz, elképzelhető, hogy többször előforduló fájlt jelent. A szótár feltöltésekor egy adott elérési útvonalon található fájlokat járjuk be, kihagyva azokat, amelyeknek nem tudjuk lekérni a méretét – hiányzó engedélyek miatt, vagy esetleg azért, mert nem rendes állományokról van szó –, illetve azokat, amelyeknek a mérete 0, hiszen az ilyenek ugyebár bizonyosan egyezőek lennének.

```
work_queue = queue.PriorityQueue()
results_queue = queue.Queue()
md5_from_filename = {}
for i in range(opts.count):
    number = "{0}: ".format(i + 1) if opts.debug else ""
    worker = Worker(work_queue, md5_from_filename, results_queue,
                    number)
    worker.daemon = True
    worker.start()
```

Amikor minden adat a helyére került, készen állunk a munkát végző szálak kialakítására. Először a munkasort és az eredménysort hozzuk létre. A munkasor prioritási sor, vagyis mindig a legalacsonyabb prioritású elemet adja elsőként vissza. A mi esetünkben ez a legkisebb méretű fájlt jelenti. Létrehozunk egy szótárat is, amelynek a kulcsai elérési útvonalat tartalmazó fájlnevek, az értékei pedig a fájlok MD5-ellenőrzőösszegei. A szótár arra jó, hogy ne kelljen egynél többször kiszámolni ugyanannak a fájlnak az ellenőrzőösszegét – ez ugyanis meglehetősen erőforrás-igényes feladat.

Most, hogy a megosztott adatgyűjtemények a helyükre kerültek, annyiszor futtatjuk a ciklusmagot, ahány szálat létre szeretnénk hozni – alapértelmezés szerint hétszer. A `Worker` alosztály hasonló ahhoz, mint amit az előzőekben alkottunk, azzal a különbséggel, hogy ezúttal a sorokat és az MD5-összegeket tartalmazó szótárat egyaránt átadjuk neki. Ahogy korábban is tettük, minden feldolgozó szálat azonnal elindítunk, és majd várakoznak, amíg feladatot tudunk adni nekik.

```
results_thread = threading.Thread(
                    target=lambda: print_results(results_queue))
results_thread.daemon = True
results_thread.start()
```

Ahelyett, hogy a `threading.Thread` osztályon alapuló alosztályt hoznánk létre az eredmények feldolgozására, létrehoztunk egy függvényt, amelyet átadunk a `threading.Thread()` tagfüggvénynek. A visszatérési érték olyan egyedi szál, amely a szál elindulása után az átadott függvényt hívja meg. Az eredménysort (ami természetesen üres) adjuk át neki, így a szál azonnal várakozásra kényszerül.

Mostanra elkészült a munkát elvégző összes szál, sőt az eredményfeldolgozó szál is, és mindannyian arra várnak, hogy feladatot kapjanak.

```
for size, filename in sorted(data):
    names = data[size, filename]
    if len(names) > 1:
        work_queue.put((size, names))
work_queue.join()
results_queue.join()
```

A fenti kóddal bejárjuk a `data` objektumot. Minden olyan (méret, fájlnév) kételemű rendezett sorozat esetében, amelyben legalább két olyan fájlt találunk, amelyekről elképzelhető, hogy egymás másolatai, a méretet és az elérési utakat is tartalmazó fájlneveket új feladatként elhelyezzük a munkasorban. Minthogy a sor osztálya a `queue` modulból származik, a zárolással nem kell foglalkoznunk.

Végül a `work_queue.join()` és a `results_queue.join()` tagfüggvény hívásával megoldjuk, hogy a sorok kiürüléséig várakozni kelljen. Így biztosítható, hogy a program addig fusson, amíg minden feladatát el nem végzi, és minden eredmény a képernyőre nem kerül. Ha ezzel elkészült, a futása tisztán ér véget.

```
def print_results(results_queue):
    while True:
        try:
            results = results_queue.get()
            if results:
                print(results)
        finally:
            results_queue.task_done()
```

A fenti függvényt a `threading.Thread()` tagfüggvénynek adjuk át paraméterül, és akkor fut le, amikor a szál, amelynek átadtuk, elindul. Azért találunk benne végtelen ciklust, mert démonszálként fogjuk használni. A feladata abban rejlik, hogy megkapja az eredményeket – egy többsoros karakterláncról van szó –, és ha a kapott karakterlánc nem üres, a képernyőre írja a tartalmát – amíg csak van mit kiírni.

A `Worker` osztály eleje hasonló a korábban megismerthez:

```
class Worker(threading.Thread):

    Md5_lock = threading.Lock()

    def __init__(self, work_queue, md5_from_filename, results_queue,
                 number):
        super().__init__()
        self.work_queue = work_queue
        self.md5_from_filename = md5_from_filename
        self.results_queue = results_queue
        self.number = number
```

```
    def run(self):
        while True:
            try:
                size, names = self.work_queue.get()
                self.process(size, names)
            finally:
                self.work_queue.task_done()
```

A különbséget az jelenti, hogy több olyan megosztott adatunk van, amelyet nyomon kell követnünk, valamint az, hogy az általunk készített `process()` függvényt más paraméterekkel hívjuk meg. A sorok miatt nem kell aggódnunk, mivel azok majd megoldják az elérések sorba állítását, de más adatelemek esetében – gondolunk itt az `md5_from_filename` szótárra – ezt a sorba állítást egy zárolás kialakításával magunknak kell megoldanunk. A zárolást az osztály egy jellemzőjeként valósítottuk meg, ugyanis azt szeretnénk, ha a `Worker` osztály minden példánya ugyanazt a zárolást használná. Így aztán, ha valamelyik példány érvénybe juttatja a zárolást, a többi példánynak addig kell várakoznia, amíg ugyanaz a példány fel is oldja a zárat.

A `process()` függvényt két részletben tekintjük át.

```
def process(self, size, filenames):
    md5s = collections.defaultdict(set)
    for filename in filenames:
        with self.Md5_lock:
            md5 = self.md5_from_filename.get(filename, None)
        if md5 is not None:
            md5s[md5].add(filename)
        else:
            try:
                md5 = hashlib.md5()
                with open(filename, "rb") as fh:
                    md5.update(fh.read())
                    md5 = md5.digest()
                    md5s[md5].add(filename)
                    with self.Md5_lock:
                        self.md5_from_filename[filename] = md5
                except EnvironmentError:
                    continue
```

Kezdetben még üres az az alapértelmező szótár, amelynek később az MD5-ellenőrzőösszegek lesznek a kulcsai, az értékek pedig halmazok lesznek: azoknak a fájlneveknek a halmazai, amelyeknek az MD5-ellenőrzőösszegei megegyeznek a kulcsban tárolttal. A fájlok bejárásakor kiolvassuk az ellenőrzőösszeget, ha már kiszámoltuk, és kiszámoljuk, ha még nem tettük meg.

Környezetkezelők 375

Az `md5_from_filename` szótárat akkor is zároljuk, ha olvasás céljából nyúlunk bele, és akkor is, ha beleírunk valamit. A `threading.Lock` osztály példányai belépéskor zárolást hajtanak végre, kilépéskor pedig feloldják a zárolást. A `with` utasítások várakoznak, amíg egy másik szál birtokában van az `Md5_lock` zár, egészen addig, amíg a zárolás meg nem szűnik. Az első `with` utasítással zárolást hajtunk végre, hogy a szótárból ki tudjuk olvasni az MD5-összeget – ha a szótár még üres, a `None` különleges szerepű objektumot olvassuk ki. Ha a kiolvasott MD5-összeg értéke a `None` objektum, akkor az összeget nekünk kell kiszámolnunk. Ilyenkor az összeget elhelyezzük az `md5_from_filename` szótárban, így kerülve el a többszöri számítást egyetlen fájl esetében.

Figyeljük meg, hogy a várakozás időtartamát úgy próbáljuk a lehető legrövidebbé tenni, hogy minden lehetséges esetben a zárolás időtartama alatt végzett munka mennyiségének csökkentésére törekszünk. Esetünkben minden egyes zároláskor éppen csak egy szótárműveletet hajtunk végre.

GIL 423

Szigorúan véve, ha a CPython értelmezőt használjuk, nem szükséges zárolnunk, a GIL ugyanis elvégzi a szótárműveletek felügyeletét helyettünk. Mindazonáltal úgy döntöttünk, hogy a program írása során nem támaszkodunk a GIL létére, és inkább mi magunk végezzük a zárolásokat.

```
for filenames in md5s.values():
    if len(filenames) == 1:
        continue
    self.results_queue.put("{0}Duplicate files ({1:n} bytes):"
                           "\n\t{2}".format(self.number, size,
                                            "\n\t".join(sorted(filenames))))
```

Végül egy ciklussal végignézzük az `md5s` nevű helyi alapértelmező szótárat, és minden olyan halmaz esetében, amely legalább két fájlnevet tartalmaz, egy többsoros karakterláncot fűzünk az eredményeket tároló sorhoz. A karakterlánc tartalmazza a szál számát (alapértelmezés szerint egy üres karakterláncot), a fájl bájtban kifejezett méretét, és a többször előforduló fájl neveit. Amikor az eredményeket tároló sorral végzünk műveleteket, nem kell zárolást használnunk. A sor ugyanis `queue.Queue` objektum, és mint ilyen, a kulisszák mögött automatikusan ellátja a zárolással kapcsolatos teendőket.

A `queue` modul osztályai nagymértékben leegyszerűsítik a többszálú alkalmazások készítését, és amikor mégis magunknak kell elvégeznünk a zárolást, a `threading` modul siet a segítségünkre. Példánkban a lehetőségek közül a legegyszerűbbet, a `threading.Lock` osztályt használtuk. Vannak további módszerek is, például a `threading.RLock` osztály (olyan zárolás, amelyet a zárolást már birtokló szál újra és újra végrehajthat), a `threading.Semaphore` osztály (olyan zárolás, amely adott számú forrás védelmére alkalmas) és a `threading.Condition` osztály használata, amely a várakozás feltételét adja meg.

GIL
423 ☞

Többszálú programok írásával gyakran elegánsabban oldhatjuk meg a felmerülő problémákat, mint ha a `subprocess` modult használnánk, de sajnos ha a több folyamatot használó programokkal hasonlítjuk össze őket, a többszálú Python-programok nem szükségszerűen nyújtják a lehető legjobb teljesítményt. Ahogy korábban már jeleztük, ez a probléma a Python alapmegvalósításában rejlik, a CPython értelmező ugyanis egyszerre csak egy processzormagon tudja futtatni a kódot, akkor is, ha a kód több szálat használ.

A problémát megoldani hivatott csomagok egyike a `multiprocessing` modul. Ahogy már korábban is jeleztük, a `grepword-m.py` program valójában a `grepword-t.py` programnak a `multiprocessing` modulra épülő változata, és a két program mindössze három sorban különbözik egymástól. Az előzőekben áttekintett `findduplicates-t.py` program esetében is lehetséges lenne hasonló átalakítás, de a gyakorlatban ez nem tanácsos. Bár a `multiprocessing` modulban lévő API igen-igen hasonlít a `threading` modul API-jára – ezzel is könnyítve a programok átalakítását –, a két API mégsem azonos, és mindkettőnek megvannak a maga bogarai. Ezen felül elmondható, hogy bár egy olyan kicsi és egyszerű programot, mint a `grepword-t.py`, bizonyosan egyszerű mechanikusan a `threading` modul használatáról a `multiprocessing` modul használatára átírni, a megközelítés túlságosan durva lenne a `findduplicates-t.py` program esetében. A legjobb eredmény általában akkor várható, ha a tervezés megkezdésétől fogva szem előtt tartjuk a `multiprocessing` modul használatát. (A `findduplicates-m.py` program megtalálható a program példái között. Ugyanazt a feladatot látja el, mint a `findduplicates-t.py` program, de tőle igencsak eltérően működik, és a működését a `multiprocessing` modul használatára alapozza.)

A CPython értelmező egy a többszálú programok írását jobban támogató változatának már folyik a fejlesztése. A munka állapotáról a legfrissebb hírek a `http://code.google.com/p/python-safethread/` címen olvashatók.

Összefoglalás

A fejezet az olyan programok készítésének módját mutatta be, amelyek az alapkönyvtár `subprocess` moduljának használatával más programok elindítására képesek.
A `subprocess` modul használatával futtatott programok kaphatnak parancssori paramétereket, kaphatnak adatokat a szabványos bemenetükre, a szabványos kimenetük és hibacsatornájuk pedig kiolvasható. A gyermekfolyamatok használata lehetővé teszi, hogy a többmagos processzorok minden előnyét kihasználjuk, a párhuzamos futásból eredő erőforrás-megosztási problémák kezelését pedig az operációs rendszerre hagyhatjuk.
A módszer hátránya, hogy ha adatokat kell megosztanunk, illetve a folyamatok futását kell egymáshoz igazítanunk, ki kell agyalnunk valamilyen kommunikációs eljárást – például meg kell osztanunk a memória egy területét (ez többek között az `mmap` modullal oldható meg), megosztott fájlokat kell kialakítanunk, vagy a hálózatot kell igénybe vennünk. Ez pedig nem mindig egyszerű.

A fejezet azt is bemutatta, hogy miként készíthetők többszálú programok. Az ilyen programok sajnos nem képesek élni a többmagos processzorok minden előnyével (mármint, ha az alap CPython értelmezőt használjuk), vagyis amikor a teljesítmény fontos szempont, és a Pythonnal dolgozunk, a több folyamat használata gyakran előnyösebb. Mindezek mellett láthattuk, hogy a `queue` modul használata és a Python zárolási módszerei annyira egyszerűen használhatók, amennyire ez egyáltalán elképzelhető. Tanúi lehettünk annak is, hogy az olyan egyszerű programok esetében, amelyeknek a `queue.Queue`, illetve a `queue.PriorityQueue` objektumoknál bonyolultabbra nincs szükségük az adattárolláshoz, a zárolással egyáltalán nem kell magunknak foglalkoznunk.

Bár a többszálú programoknak kétségkívül vannak előnyeik, a megírásuk és karbantartásuk, illetve a bennük végzett hibakeresés az egyszálú programokhoz képest sokszor jelentős többletterhet ró a programozóra. A többszálú programoknál ugyanakkor lényegesen egyszerűbb a szálak közötti kapcsolattartás – már amennyiben zárolást vagy valamilyen `queue` objektumot használunk –, illetve az összehangolás – például az eredmények összegyűjtése – , mint a gyermekfolyamatokat használóknál. A többszálúság az olyan grafikus felületű programok esetében is igen hasznos lehet, amelyeknek hosszú ideig tartó számításokat kell végezniük, és közben folyamatosan reagálniuk kell a felhasználó felől érkező parancsokra – például a futó feladat megszakítását célzókra. Ha azonban megosztott memórián, több oldalról is elérhető sor adatszerkezeten, vagy hasonlóan hatékony csatornán át kommunikálhatnak, a több szálon alapuló megoldásokkal szemben érdemes lehet a több folyamaton alapuló megoldások mellett dönteni.

A következő fejezetben ismét egy többszálú programmal ismerkedünk meg: egy olyan kiszolgálóval, amely a hozzá intézett kéréseket külön szálon kezeli, a megosztott adat védelméről pedig zárolással gondoskodik.

Gyakorlatok

1. Másoljuk le a `grepword-p.py` programot, és módosítsuk úgy, hogy ne a gyermekfolyamatok írják a kimenetet a képernyőre, hanem gyűjtse össze mindet a főprogram, és miután minden gyermekfolyamat futása véget ért, a főprogram rendezze, majd írja a képernyőre az eredményeket. A feladat elvégzéséhez csak a `main()` függvényt kell módosítanunk: három során változtatnunk kell, és három új sort kell hozzáírnunk. A gyakorlat elvégzése igényel némi gondolkodást és odafigyelést, valamint el kell olvasnunk a `subprocess` modul dokumentációját. A megoldást a `grepword-p_ans.py` fájlban találjuk.
2. Írjunk olyan többszálú programot, amely beolvassa a parancssorban megadott fájlokat, illetve a parancssorban megadott könyvtárak állományait is. Minden XML-fájlt (vagyis az „`<?xml`" karaktersorozattal kezdődőket) olvassunk be és dolgozzuk fel valamelyik XML-feldolgozóval. Állítsuk elő az egyedi címkék listáját, ha pedig a feldolgozás hibába ütközik, jelenítsünk meg hibaüzenetet. Példaként íme a program egy adott futásának kimenete:

```
./data/dvds.xml is an XML file that uses the following tags:
    dvd
    dvds
./data/bad.aix is an XML file that has the following error:
    mismatched tag: line 7889, column 2
./data/incidents.aix is an XML file that uses the following tags:
    airport
    incident
    incidents
    narrative
```

A program megírásának legegyszerűbb módja, ha a `findduplicates-t.py` program másolatát írjuk át, bár természetesen lehetőségünk van a program írását teljesen elölről elkezdeni. Kisebb mértékben módosítanunk kell a `Worker` osztály `__init__()` és `run()` tagfüggvényét, a `process()` tagfüggvényt pedig teljesen újra kell írnunk (de ez mindössze nagyjából húsz sort jelent). A program `main()` függvényét számos helyen egyszerűsítenünk kell, ami a `print_results()` függvény egyik sorára is igaz. A használati utasítást is át kell alakítanunk, az alábbiaknak megfelelően:

```
Usage: xmlsummary.py [options] [path]
outputs a summary of the XML files in path; path defaults to .

Options:
  -h, --help    show this help message and exit
  -t COUNT, --threads=COUNT
                the number of threads to use (1..20) [default 7]
  -v, --verbose
  -d, --debug
```

Mindenképpen próbáljuk meg a `--debug` kapcsolóval futtatni a programot, így ugyanis ellenőrizhetjük, hogy a szálak rendben elindultak, és mindegyik szál elvégzi a munka rá eső részét. A megoldást a száz sornál alig hosszabb, közvetlen zárolást nem végző `xmlsummary.py` fájlban találjuk.

10

Hálózatkezelés

- TCP-ügyfél létrehozása
- TCP-kiszolgáló létrehozása

A hálózatok lehetővé teszik a számítógépprogramok számára, hogy kapcsolatba lépjenek egymással, még akkor is, ha különböző gépeken futnak. Egyes programoknak (ide tartoznak például a webböngészők) ez a kapcsolatteremtés a lényegi feladatuk, másoknál viszont a hálózati lehetőségek csak kiegészítik az eredeti működést, például a távoli működtetés, illetve naplózás biztosításával, vagy éppen azzal, hogy a hálózatok révén adatokat tárolhatunk egy másik rendszeren, illetve betölthetjük őket onnan. A legtöbb hálózati program egyenrangú partnerek között (ilyenkor ugyanaz a program fut különböző gépeken), illetve még gyakrabban ügyfél-kiszolgáló alapokon működik, ahol az ügyfélprogram kérelmeket küld a kiszolgálónak.

Fejezetünkben elkészítünk egy igen egyszerű ügyfél-kiszolgáló alkalmazást. Az ilyen típusú alkalmazásokat rendszerint két külön programban valósítják meg – a kiszolgáló fogadja a kérelmeket és válaszol rájuk, míg egy vagy több ügyfél kérelmeket küld a kiszolgálónak, és várakozik a kiszolgáló válaszára. Mindehhez az ügyfeleknek tudniuk kell, hol találják a kiszolgálót, vagyis ismerniük kell az IP-címét, valamint a kapuszámát.* Ráadásul az ügyfélnek és a kiszolgálónak meg kell egyeznie az adatátviteli protokollban, és olyan formátumban kell elküldeniük az adataikat, amelyet kölcsönösen megértenek.

A Python alacsonyszintű `socket` modulja (erre épül a nyelv összes magasabb szintű hálózatkezelő modulja) támogatja mind az IPv4, mind az IPv6 formátumú címek használatát. Mindemellett ismeri a leggyakrabban használt hálózati protokollokat, mint az UDP (User Datagram Protocol – ez egy könnyűsúlyú, de megbízhatatlan, kapcsolat nélküli protokoll, amelyben az adatok csomagok alakjában utaznak, de semmi sem garantálja, hogy célba is érkeznek) és a TCP (Transmission Control Protocol – ez egy megbízható, adatfolyamköz-

* A gépek kiépíthetik a kapcsolatot a szolgáltatás feltárása révén is, például a bonjour API használatával – a megfelelő modulokat megtalálhatjuk a Python Package Indexen, a `pypi.python.org/pypi` címen.

pontú protokoll). A TCP-vel tetszőleges adatmennyiség átvihető – a csatoló feladata, hogy hálózati átvitelre alkalmas darabokra bontsa az adatokat, majd a megérkezésük után újra összeállítsa a darabokból az eredeti alakot.

Befőttek 293 ☜

Az UDP-t gyakran használják olyan műszerek adatainak továbbítására, amelyek folyamatosan szolgáltatnak eredményt, viszont nem számít, ha olykor-olykor kimarad egy adat – hasonlóképpen, a protokoll alkalmazható hang- és videófolyamok átvitelére is abban az esetben, amennyiben nem gond, ha egy-egy képkocka elvész az út során. Mind az FTP, mind a HTTP a TCP-re épül, hasonlóan a legtöbb ügyfél-kiszolgáló alkalmazáshoz, amelyeknek szükségük van a TCP kapcsolatközpontú működésére és megbízhatóságára. Ebben a fejezetben egy ügyfél-kiszolgáló programot készítünk, következésképpen mi is a TCP lehetőségeit vesszük igénybe.

Döntenünk kell emellett arról is, hogy az adatainkat szöveges sorok vagy bináris blokkok alakjában szeretnénk küldeni és fogadni, továbbá – amennyiben az utóbbit választottuk – a formátumot is meg kell határoznunk. Fejezetünkben olyan bináris blokkokat használunk, amelyekben az első négy bájt az utánuk következő adatcsomag hosszát adja meg (egy előjel nélküli egész alakjában, amelyet a `struct` modullal állítunk elő), ez az adatcsomag pedig nem más, mint egy befőtt. Ennek a megközelítésnek nagy előnye, hogy ugyanazt a küldő és fogadó kódot használhatjuk bármely alkalmazás esetében, hiszen szinte tetszőleges adatokat tárolhatunk befőtt formájában. Hátránya ugyanakkor, hogy a befőttek használatához mind az ügyfelet, mind a kiszolgálót Python nyelven kell elkészítenünk, illetve mindenképpen lehetőséget kell biztosítanunk a Python elérésére (erre alkalmas a Jython könyvtár a Javában, illetve a Boost.Python a C++-ban). Mindemellett a befőttekkel kapcsolatos biztonsági meggondolások, amelyekről már szót ejtettünk, itt is érvényesek.

A hálózatok lehetőségeit egy autóregisztrációs program példáján mutatjuk be. A kiszolgáló tárolja a regisztrációs adatokat (a rendszámot, az ülések számát, a megtett kilométereket és a tulajdonos nevét), az ügyféllel pedig kiolvashatjuk az autó adatait, módosíthatjuk a kilométerek számát és a tulajdonos nevét, illetve létrehozhatunk egy új bejegyzést. Tetszőleges számú ügyfelet munkába állíthatunk, és ezek nem zavarják egymás működését, még akkor sem, ha ketten egyszerre próbálják elérni a kiszolgálót. Mindez úgy lehetséges, hogy a kiszolgáló minden ügyfél kérelmét külön szállal kezeli. (Látjuk majd, hogy ugyanilyen könnyedén különböző folyamatokat is használhatnánk.)

A példában a könnyebb megvalósítás érdekében az ügyfeleket és a kiszolgálót egyazon gépen futtatjuk, vagyis IP-címként használhatjuk a „localhost" kulcsszót (ha a kiszolgáló nem ugyanazon a gépen fut, az ügyfeleknek átadhatjuk az IP-címét a parancssorban, és a rendszer gond nélkül működik, ha csak egy tűzfal az útjába nem áll). Meghatároztuk a kapuszámot is – esetünkben a 9653 bizonyult szimpatikusnak. A választásban azonban igen tág lehetőségeink vannak – a kapuszámok általában az 5001–32767 tartományból kerülnek ki, de elméletben 1024 és 65 535 között bármelyik számot kiválaszthatjuk.

A kiszolgáló a következő öt típusú kérelmet képes fogadni: GET_CAR_DETAILS, CHANGE_MILEAGE, CHANGE_OWNER, NEW_REGISTRATION és SHUTDOWN – ezek mindegyikére megfelelő válasszal is rendelkezik, ami a kért adatokat, a kért művelet visszaigazolását, illetve egy hibajelzést tartalmazhat.

TCP-ügyfél létrehozása

Ügyfélprogramunk a car_registration.py fájlban kap helyet. Lássuk most, hogyan is folyik a „beszélgetés" a kiszolgálóval (a kiszolgáló már fut, a menüt pedig egy kicsit átszerkesztettük, hogy elférjen az oldalon):

```
(C)ar  (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License: 024 hyr
License: 024 HYR
Seats:   2
Mileage: 97543
Owner:   Jack Lemon
(C)ar  (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]: m
License [024 HYR]:
Mileage [97543]: 103491
Mileage successfully changed
```

A beírt adatokat félkövérrel jelöltük – ahol nem látszik a bemenet, ott a felhasználó egyszerűen ENTER-t nyomott, és ezzel elfogadta az alapértelmezést. Jelen esetben a felhasználó egy autó regisztrációs adatait nézte meg, majd módosította a megtett kilométerek számát.

Annyi ügyfelet futtathatunk, amennyit csak akarunk, és ha a felhasználó bezárja az egyik ügyfelet, ez semmilyen hatással nincs a kiszolgálóra. Ha azonban a kiszolgálót leállítjuk, az az ügyfél, amelyikben az erre vonatkozó parancsot kiadtuk, leáll, a többiek pedig „Connection refused" (A kapcsolat elutasítva) hibaüzenetet kapnak, és a működésük megszakad, amint legközelebb kapcsolódni próbálnak a kiszolgálóhoz. Alkalmazásunk kifinomultabb változatában elérhetnénk, hogy a leállításra csak bizonyos felhasználóknak legyen lehetősége, esetleg csak egyes gépeken, de most egyszerűen az ügyfélprogramba építettük be ezt a lehetőséget, hogy lássuk, miként is élhetünk vele.

A következőkben áttekintjük a kódot – a main() függvénnyel és a felhasználói felület kezelésével kezdjük, végére hagyva magát a hálózatkezelési kódot.

```
def main():
    if len(sys.argv) > 1:
        Address[0] = sys.argv[1]
    call = dict(c=get_car_details, m=change_mileage, o=change_owner,
                n=new_registration, s=stop_server, q=quit)
    menu = ("(C)ar Edit (M)ileage Edit (O)wner (N)ew car "
            "(S)top server (Q)uit")
    valid = frozenset("cmonsq")
```

```
    previous_license = None
    while True:
        action = Console.get_menu_choice(menu, valid, "c", True)
        previous_license = call[action](previous_license)
```

Az `Address` egy globális változó, amely kételemű listában tárolja az IP-címet és a kapuszámot, `["localhost", 9653]` alakban – az IP-cím azonban a parancssorból felülírható. A `call` szótár a menüpontokat függvényeknek felelteti meg.

A `Console` modul, amelyhez ezzel a könyvvel juthatunk hozzá, hasznos függvényeket tartalmaz, amelyekkel a konzolon keresztül értékeket olvashatunk be a felhasználóktól – ilyenek például a `Console.get_string()` és a `Console.get_integer()`. Ezek a függvények hasonlóak a korábbi fejezetekben elkészítettekhez; a könnyebb elérhetőség kedvéért helyeztük el őket egy modulban.

A felhasználók kényelmét szolgálja, hogy a program megjegyzi a legutóbb kért rendszámot, amelyet alapértelmezettként használhatunk, hiszen a legtöbb parancs első kérdése éppen erre az adatra irányul. Ha a felhasználó meghozta a döntését, meghívjuk a megfelelő függvényt, átadva az előző rendszámot, és elvárjuk, hogy a függvény működése végén az általa használt rendszámot adja vissza. Mivel a parancsokat kezelő ciklus végtelen, a programot egy függvényből kell végül befejeznünk – ezt majd láthatjuk a későbbiekben.

```
def get_car_details(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is not None:
        print("License: {0}\nSeats: {1[0]}\nMileage: {1[1]}\n"
              "Owner:   {1[2]}".format(license, car))
    return license
```

Ezzel a függvénnyel az egyes autókról szerezhetünk adatokat. Mivel a legtöbb függvénynek szüksége van egy rendszámra a felhasználótól, valamint gyakran az autó egyéb adataira is, amelyekkel műveleteket végezhet, ezek megszerzését egy `retrieve_car_details()` nevű függvényre bíztuk. Ez egy rendezett párt ad vissza, amelyben szerepel a felhasználó által megadott rendszám, valamint egy névvel ellátott rendezett sorozat, a `CarTuple`, amely tartalmazza az autó üléseinek számát, a megtett kilométereket, valamint a tulajdonos nevét (ismeretlen rendszám esetén az eredményként kapott rendezett párban az előző rendszám és a `None` szerepel). Itt azonban mindössze kiírjuk a kiolvasott adatokat, és visszaadjuk azt a rendszámot, amelyet a program alapértelmezésként használ a következő olyan függvény hívásakor, amelynek szüksége van rendszámra.

```
def retrieve_car_details(previous_license):
    license = Console.get_string("License", "license",
                                 previous_license)
    if not license:
        return previous_license, None
```

```
    license = license.upper()
    ok, *data = handle_request("GET_CAR_DETAILS", license)
    if not ok:
        print(data[0])
        return previous_license, None
    return license, CarTuple(*data)
```

Ez az első olyan függvényünk, amely kihasználja a hálózatkezelési lehetőségeket. Mindebből pillanatnyilag csak annyit látunk, hogy a kód meghívja a `handle_request()` függvényt, amelyet a későbbiekben veszünk szemügyre közelebbről. A működése röviden annyiból áll, hogy elküldi a számára átadott adatokat a kiszolgálónak, és visszaadja a kapott választ. Nem törődik azzal, hogy milyen adatok haladnak át rajta: tisztán a hálózati forgalmat biztosítja.

Az autóregisztrációk esetében a protokoll egyszerűen abban áll, hogy első paraméterként átküldjük a kiszolgálónak az elvégzendő művelet kódját, másodikként pedig a szükséges adatokat – esetünkben mindössze a rendszámra lesz szükség. A válasz protokollja szerint a kiszolgáló mindig egy rendezett párt küld vissza, amelynek az első eleme egy logikai siker/kudarc jelző. Amennyiben ennek értéke `False`, egy rendezett párt kaptunk a kiszolgálótól, amelynek a második eleme egy hibaüzenet. Ha az érték `True`, két eset lehetséges. Kaphattunk egy rendezett párt, amelynek a második eleme egy visszaigazoló üzenet, vagy egy rendezett n-est, amelyben a második és az azt követő elemek a kért adatokat tartalmazzák.

Itt tehát, ha a rendszám ismeretlen, az `ok` értéke `False`, így kiírjuk a `data[0]`-ban található hibaüzenetet, és változatlan alakban visszaadjuk az előző rendszámot. Egyébként viszont visszaadjuk az új rendszámot (ami ezt követően az „előző" lesz), valamint egy `CarTuple` értéket, amelyet a `data` listából (ülések száma, kilométerszám, tulajdonos) nyertünk ki.

```
def change_mileage(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is None:
        return previous_license
    mileage = Console.get_integer("Mileage", "mileage",
                                  car.mileage, 0)
    if mileage == 0:
        return license
    ok, *data = handle_request("CHANGE_MILEAGE", license, mileage)
    if not ok:
        print(data[0])
    else:
        print("Mileage successfully changed")
    return license
```

A függvény működése alapjaiban a `get_car_details()` esetében látott mintát követi, azzal a különbséggel, hogy ha rendelkezésre állnak az adatok, az egyiküket módosítjuk is. A kódban valójában két hálózati művelet is szerepel, hiszen a `retrieve_car_details()`

meghívja a `handle_request()` függvényt, hogy hozzájusson az autó adataihoz – erre szükségünk van egyrészt azért, hogy igazoljuk a rendszám érvényességét, másrészt hozzá kell jutnunk a kilométerek számához, hogy ezt használhassuk alapértelmezésként. A válasz itt minden esetben egy rendezett pár, amelynek a második tagja egy hibaüzenet vagy a `None` érték.

A `change_owner()` függvény bemutatására nem térünk ki, hiszen a szerkezete azonos a `change_mileage()` kódjában látottal, de nem foglalkozunk a `new_registration()` függvénnyel sem, ugyanis ez csak annyiban tér el az előzőektől, hogy itt nem töltjük be kezdetben az autó adatait (hiszen új autót veszünk fel a rendszerbe), és minden adatot elkérünk a felhasználótól, nem csak egyet-egyet módosítunk. Mindez semmiféle új hálózati ismeretet nem ad, így inkább tovább is lépünk egy érdekesebb függvényhez.

```
def quit(*ignore):
    sys.exit()
def stop_server(*ignore):
    handle_request("SHUTDOWN", wait_for_reply=False)
    sys.exit()
```

Ha a felhasználó maga dönt úgy, hogy kilép a programból, a `sys.exit()` hívásával „tisztán" fejezzük be a program futtatását. Minden menüponthoz tartozó függvényt az előző rendszámmal hívunk meg, de ebben az esetben nem foglalkozunk a paraméterrel. Mindazonáltal a `def quit():` meghatározás hibás működéshez vezetne, ugyanis ez esetben a függvény nem várna paramétert – így az előző rendszámot a számára átadva `TypeError` kivételt kapnánk, ami azt jelzi, hogy a várt nulla helyett egy paramétert adtunk át a függvénynek. Ezért hát a `*ignore` paramétert használjuk, így tetszőleges számú hely szerinti paramétert megadhatunk. Az `ignore` név mindössze arra utal, hogy figyelmen kívül hagyjuk az átadott paramétereket, a Python számára nem bír semmilyen jelentőséggel.

Ha a felhasználó úgy dönt, hogy leállítja a kiszolgálót, ezt az utasítást a `handle_request()` függvénnyel adjuk át – ez esetben nem várunk választ. Ha tehát elküldtük az adatokat, a `handle_request()` visszatér anélkül, hogy a válaszra várna, és végrehajthatunk egy tiszta leállást a `sys.exit()` segítségével.

```
def handle_request(*items, wait_for_reply=True):
    SizeStruct = struct.Struct("!I")
    data = pickle.dumps(items, 3)
    try:
        with SocketManager(tuple(Address)) as sock:
            sock.sendall(SizeStruct.pack(len(data)))
            sock.sendall(data)
            if not wait_for_reply:
                return
            size_data = sock.recv(SizeStruct.size)
            size = SizeStruct.unpack(size_data)[0]
            result = bytearray()
```

```
        while True:
            data = sock.recv(4000)
            if not data:
                break
            result.extend(data)
            if len(result) >= size:
                break
    return pickle.loads(result)
except socket.error as err:
    print("{0}: is the server running?".format(err))
    sys.exit(1)
```

A fenti függvény felel az ügyfélprogram hálózati működéséért. A kódban először létrehozunk egy `struct.Struct` változót, amely egy előjel nélküli egészet tartalmaz hálózati bájtsorrendben, majd pedig létrehozunk egy befőttet, amelyben a paraméterként kapott adatokat tároljuk. A függvény nem tudja, miféle adatokhoz jutott hozzá, de nem is foglalkozik vele. Figyeljük meg, hogy a 3-as befőttprotokollt állítottuk be – tettük ezt azért, hogy a kiszolgáló és az ügyfelek biztosan ugyanazt a változatot használják, még akkor is, ha időközben valamelyiküknél a Python újabb változatát vették használatba.

Ha a protokollt jobban fel szeretnénk készíteni a jövőre, változatkezelést alkalmazhatunk (hasonlóan a bináris lemezformátumokhoz), amit megtehetünk a hálózat vagy az adatok szintjén. Amennyiben az előbbit választjuk, a korábbi egy helyett adjunk át két előjel nélküli egészet, amelyek közül a második a protokoll változatszámát tartalmazza. Az adatok szintjén ragaszkodhatunk ahhoz, hogy a befőttbe mindig egy lista (vagy mindig egy szótár) kerüljön, amelynek az első („változat") eleme a tartalmazza a változatszámot. (A változatkezelés alkalmazására lesz még módunk a gyakorlatok során.)

A `SocketManager` egy egyedi környezetkezelő, amely biztosítja számunkra az adatátvitelhez szükséges csatolót – hamarosan közelebbről is megismerkedhetünk vele.
A `socket.socket.sendall()` tagfüggvény minden adatot elküld, amit csak megadunk neki, esetleg több `socket.socket.send()` hívást végezve a háttérben. Minden esetben két adatelemet küldünk át: a befőtt hosszát, valamint magát a befőttet. Amennyiben a `wait_for_reply` paraméter értéke `False`, nem várunk a válaszra, hanem azonnal visszatérünk a függvényből – a környezetkezelő biztosítja, hogy lezárjuk a csatolót, mielőtt a visszatérés valóban megtörténne.

Az adatok elküldése után a `socket.socket.recv()` tagfüggvénnyel hozzájutunk a válaszhoz (amennyiben szükségünk van rá). A tagfüggvény blokkolja a programot, amíg hozzá nem jut a várt adatokhoz. Az első hívással négy bájtot kérünk – ennyi ugyanis annak az egésznek a mérete, amely az utána álló befőtt hosszát tárolja. Ha hozzájutottunk, a `struct.Struct` segítségével elkészítjük belőle a `size` nevű egész számot. Ezt követően létrehozunk egy üres `bytearray` típusú változót, és megkíséreljük betölteni a befőttet legfeljebb 4000 bájtos blokkokban. Ha beolvastunk `size` mennyiségű bájtot (illetve ha

az adatfolyam még ez előtt véget ért), kiszállunk a ciklusból, kicsomagoljuk az adatokat a `pickle.loads()` függvénnyel (amely `bytes` vagy `bytearray` objektumot képes fogadni), és visszaadjuk az eredményt. Ez esetben tökéletesen tisztában vagyunk azzal, hogy egy rendezett sorozatot kapunk, hiszen ezt rögzíti az autóregisztrációs kiszolgálóval közös protokoll, de maga a `handle_request()` nem tudja, hogy milyen adatokról van szó, és ez nem is érdekli.

Ha valami gond adódik a hálózati kapcsolattal, például leáll a kiszolgáló, vagy megszakad a kapcsolat, `socket.error` kivételhez jutunk. Ilyenkor a programunk elfogja a kivételt, kiír egy hibaüzenetet, és leáll.

```
class SocketManager:
    def __init__(self, address):
        self.address = address
    def __enter__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect(self.address)
        return self.sock
    def __exit__(self, *ignore):
        self.sock.close()
```

Az `address` objektum egy rendezett pár (IP-cím, kapuszám), amelyet akkor állítunk be, amikor a környezetkezelőt létrehozzuk. Mihelyt a környezetkezelőt egy `with` utasításban alkalmazzuk, létrehoz egy csatolót, és megkísérli a kapcsolat kiépítését – mindaddig blokkol, amíg a kapcsolat létre nem jön, vagy nem kap csatolókivételt. A `socket.socket()` kezdeti értékadó tagfüggvény első paramétere a címcsalád – esetünkben a `socket.AF_INET` (IPv4) beállítást használtuk, de mást is választhatunk, például a `socket.AF_INET6` (IPv6), a `socket.AF_UNIX` vagy a `socket.AF_NETLINK` beállítást. A második paraméter általában az itt is használt `socket.SOCK_STREAM` (TCP) vagy a `socket.SOCK_DGRAM` (UDP).

Ha a programvezérlés elhagyja a `with` utasítás hatókörét, a rendszer meghívja a környezeti objektum `__exit__()` tagfüggvényét. Ez esetben nem törődünk azzal, hogy bekövetkezett-e valamilyen kivétel (így a kivételparamétereket figyelmen kívül hagyjuk), egyszerűen csak bezárjuk a csatolót. Mivel a tagfüggvény visszatérési értéke `None` (logikai környezetben `False`), minden kivétel felfelé vándorol a hívásveremben – ez nagyszerűen beválik, hiszen a csatolókivételek kezelésére elhelyeztünk egy alkalmas `except` blokkot a `handle_request()` kódjában.

TCP-kiszolgáló létrehozása

Mivel a kiszolgálók létrehozásának kódja gyakran igen hasonló szerkezetű, az alacsonyszintű `socket` modul helyett használhatjuk a magasszintű `socketserver` modult, amely elvégzi a háttérmunkát helyettünk. Nekünk mindössze az a feladatunk, hogy egy kérelemkezelő osztályt hozzunk létre egy `handle()` nevű tagfüggvénnyel, amelyet a kérelmek kiadására és

a válaszok fogadására használhatunk. A `socketserver` modul kezeli az adatcserét, kiszolgálja a kapcsolati kérelmeket, akár sorban egymás után, akár minden kérelmet külön szálnak vagy folyamatnak adva át – és mindezt áttetszően teszi, vagyis nem kell foglalkoznunk az alacsonyszintű részletekkel.

Alkalmazásunk kódját a `car_registration_server.py` fájl tartalmazza.* A programban egy igen egyszerű, `Car` nevű osztályt találhatunk, amely tulajdonságok alakjában tárolja az ülések számát, a megtett kilométereket és a tulajdonos adatait (az ülések száma természetesen csak olvasható). A rendszámokat azonban ne itt keressük – az autókat a program egy szótárban tárolja, amelyben a kulcsok a rendszámok.

Először a `main()` függvényt vizsgáljuk meg, majd röviden áttekintjük, hogyan tölti be a rendszer a kiszolgáló adatait, ezt követően pedig létrehozzuk az egyedi kiszolgálóosztályunkat, végül pedig létrehozunk egy osztályt az ügyfelek kérelmeinek kezelésére.

```
def main():
    filename = os.path.join(os.path.dirname(__file__),
                            "car_registrations.dat")
    cars = load(filename)
    print("Loaded {0} car registrations".format(len(cars)))
    RequestHandler.Cars = cars
    server = None
    try:
        server = CarRegistrationServer(("", 9653), RequestHandler)
        server.serve_forever()
    except Exception as err:
        print("ERROR", err)
    finally:
        if server is not None:
            server.shutdown()
            save(filename, cars)
            print("Saved {0} car registrations".format(len(cars)))
```

Az autóregisztrációs adatokat ugyanabban a könyvtárban tároltuk, mint magát a programot. A `cars` objektumban egy szótárat helyezünk el, amelynek a kulcsai a rendszámok karakterláncai, az értékei pedig `Car` objektumok. A kiszolgálók általában nem írnak ki semmit a képernyőre, hiszen az indításuk és a leállításuk rendszerint automatikus, és a háttérben működnek – így az állapotukról naplókban tudósítanak (például a `logging` modul használatával). Most azonban a tesztek és a kísérletezés megkönnyítésére kiírunk egy üzenetet a kiszolgáló indításakor és leállításakor.

* Windows rendszereken a kiszolgáló első futtatása alkalmával kaphatunk egy párbeszédablakot a tűzfaltól, amely figyelmeztet, hogy letiltotta a hálózatelérést a Python számára – a kiszolgáló helyes működése érdekében kattintsunk az Unblock (Tiltás feloldása) gombra.

A kérelemkezelő osztálynak el kell érnie a `cars` szótárat, de ezt nem adhatjuk át egy példánynak, mivel ezeket a példányokat a kiszolgáló hozza létre a számunkra – minden kérelem kezeléséhez egyet. Ezért a szótárat a `RequestHandler.Cars` osztályváltozóban tároljuk, ahol minden példány számára elérhető lesz.

Létrehozzuk a kiszolgáló egy példányát, és átadjuk neki azt a címet és kaput, ahol működni fog, továbbá a `RequestHandler` osztályobjektumot – nem pedig ennek egy példányát. A cím helyén egy üres karakterláncot adunk át, jelezve, hogy bármely elérhető IPv4-cím megfelel (ide tartozik a helyi gép címe, vagyis a `localhost` is). Ez után arra utasítjuk a kiszolgálót, hogy fogadja a kérelmeket örökkön örökké. Amikor a kiszolgáló leáll (arról, hogy ezt miként teszi, hamarosan szólunk), lemezre mentjük a `cars` szótárat, hiszen ezt az ügyfelek időközben megváltoztathatták.

```
def load(filename):
    try:
        with contextlib.closing(gzip.open(filename, "rb")) as fh:
            return pickle.load(fh)
    except (EnvironmentError, pickle.UnpicklingError) as err:
        print("server cannot load data: {1}".format(err))
        sys.exit(1)
```

A betöltés kódja egyszerű, mivel az alapkönyvtár `contextlib` moduljának környezetkezelőjét használtuk annak biztosítására, hogy a rendszer mindenképpen bezárja a fájlt akkor is, ha időközben kivétel lép fel. Ugyanezt az eredményt saját környezetkezelővel is elérhetjük, amelynek a kódja valahogy így alakul:

```
class GzipManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
    def __enter__(self):
        self.fh = gzip.open(self.filename, self.mode)
        return self.fh
    def __exit__(self, *ignore):
        self.fh.close()
```

Egyedi `GzipManager` osztályunkkal a `with` utasítás alakja így fest:

```
with GzipManager(filename, "rb") as fh:
```

A `save()` függvény (amelyet nem mutatunk be) szerkezetileg megegyezik a `load()` függvénnyel, az eltérés mindössze annyi, hogy a fájlt bináris módban nyitjuk meg, az adatokat a `pickle.dump()` függvénnyel mentjük lemezre, és nem adunk vissza semmit.

```
class CarRegistrationServer(socketserver.ThreadingMixIn,
                            socketserver.TCPServer): pass
```

Többszörös öröklés 395

Bármily meglepő, ez az egyéni kiszolgálóosztályunk teljes kódja. Ha olyan kiszolgálót szerettünk volna készíteni, amely nem szálakat, hanem folyamatokat alkalmaz, mindössze annyiban kellene eltérnünk a fenti kódtól, hogy az osztályunkat a `socketserver.ThreadingMixIn` helyett a `socketserver.ForkingMixIn` osztályból származtatjuk. A *mixin* szócska jobbára olyan osztályra utal, amelyet eleve többszörös öröklésre szántak. A `socketserver` modul osztályai segítségével különféle egyedi kiszolgálókat készíthetünk a megfelelő alaposztályok párjaiból leszármaztatva – így létrehozhatunk UDP-kiszolgálókat, de unixos TCP- és UDP-kiszolgálókat is készíthetünk.

Fontos megjegyeznünk, hogy az általunk használt `socketserver` bekeveredő (mixin) osztálynak az öröklési sorban mindig elöl kell állnia. Ezzel biztosíthatjuk, hogy a rendszer a bekeveredő osztály tagfüggvényeit használja a második osztálybeli társaik helyett, amennyiben névegyezés fordul elő. Ennek az az oka, hogy egy tagfüggvény hívásakor a Python a megadásuk sorrendjében halad végig az alaposztályokon, és azt a tagfüggvényt használja, amelyik elsőként megfelel a hívottnak.

A csatolókiszolgáló az általunk megadott osztály révén kérelemkezelőt hoz létre minden egyes kérelem kezelésére. Egyedi `RequestHandler` osztályunk a kérelmek minden kezelhető típusához külön tagfüggvényt biztosít, emellett meg kell valósítania a `handle()` tagfüggvényt is – hiszen ez az egyetlen olyan tagfüggvény, amelyet a csatolókiszolgáló használ. Mielőtt azonban közelebbről is megismerkednénk a tagfüggvényekkel, tekintsük át az osztály meghatározását és osztályváltozóit.

```
class RequestHandler(socketserver.StreamRequestHandler):
    CarsLock = threading.Lock()
    CallLock = threading.Lock()
    Call = dict(
            GET_CAR_DETAILS=(
                    lambda self, *args: self.get_car_details(*args)),
            CHANGE_MILEAGE=(
                    lambda self, *args: self.change_mileage(*args)),
            CHANGE_OWNER=(
                    lambda self, *args: self.change_owner(*args)),
            NEW_REGISTRATION=(
                    lambda self, *args: self.new_registration(*args)),
            SHUTDOWN=lambda self, *args: self.shutdown(*args))
```

Létrehoztunk egy `socketserver.StreamRequestHandler` nevezetű alosztályt, hiszen egy TCP-adatfolyamkiszolgálót használunk. Az UDP-kiszolgálók esetében hasonló lehetőségeket biztosít a `socketserver.DatagramRequestHandler`, de származtathatjuk az alosztályunkat a `socketserver.BaseRequestHandler` osztályból is, amennyiben alacsonyabb szintű elérésre van szükségünk.

A `RequestHandler.Cars` szótár egy osztályváltozó, amelyet a `main()` függvényben illesztettünk be – ez tartalmazza a regisztrációs adatokat. A Python lehetővé teszi, hogy mindenféle bonyolult kódolás nélkül további jellemzőkkel egészítsünk ki objektumokat (így osztályokat és példányokat) az osztályon kívülről (amennyiben az objektum rendelkezik a `__dict__` jellemzővel), ami igen kényelmes lehetőség. Mivel ismert, hogy az osztály működése ezen a változón áll vagy bukik, egyes programozók szükségesnek érezhetik, hogy a `Cars = None` sorral jelezzék az osztályváltozó jelenlétét az osztályban.

Szinte minden kérelemkezelő tagfüggvénynek szüksége van a `Cars` adataira, de valahogy biztosítanunk kell, hogy egyszerre két tagfüggvény (két különböző szálból) ne érhesse el – ha ugyanis erről nem gondoskodunk, a szótár tartalma károsodhat, és a program akár össze is omolhat. Mindennek elkerülésére bevezettünk egy zárolási osztályváltozót, amellyel biztosíthatjuk, hogy a `Cars` szótárhoz egyszerre csak egy szál férjen hozzá.* (A szálak és a zárolás használatáról a 9. fejezetben olvashattunk bővebben.)

A `Call` szótár szintén osztályváltozó, amelyben a kulcsok a kiszolgáló által végezhető műveletek, az értékek pedig a műveletekhez tartozó függvények. Az utóbbiakat nem használhatjuk közvetlenül, mint az ügyfél menüjének szótárában, ugyanis az osztály szintjén nem érhető el a `self` változó. Ezért burkolófüggvényeket készítettünk, amelyek a hívásuk idején hozzájutnak a `self` objektumhoz, majd meghívják a megfelelő tagfüggvényt, átadva neki a `self` objektumot és a többi szükséges paramétert. Megoldás lehetne az is, ha a `Call` szótárat a tagfüggvények után hoznánk létre. Így már alkalmazhatnánk a `GET_CAR_DETAILS=get_car_details` és ehhez hasonló értékadásokat, hiszen a Python megtalálja a `get_car_details()` tagfüggvényt, amelyet így már létrehoztunk. Azért döntöttünk az előbbi megoldás mellett, mert az alkalmazásának a célja világosabban látszik, és így a kód nem függ a szótár és a tagfüggvények létrehozásának sorrendjétől.

GIL 423 ☜

Jóllehet a `Call` szótár olvasására csak az után kerül sor, hogy az osztályt létrehoztuk, változtathatósága miatt mégis különösen ügyeltünk a biztonságára, és egy zárat hoztunk létre, amely meggátolja, hogy egyszerre két szál férjen hozzá. (A GIL jelenléte miatt a CPythonban erre a zárolásra nincs igazán szükség.)

```
def handle(self):
    SizeStruct = struct.Struct("!I")
    size_data = self.rfile.read(SizeStruct.size)
    size = SizeStruct.unpack(size_data)[0]
    data = pickle.loads(self.rfile.read(size))
    try:
        with self.CallLock:
            function = self.Call[data[0]]
        reply = function(self, *data[1:])
```

GIL 423 ☜

* A GIL (Global Interpreter Lock) képes biztosítani, hogy a `Cars` szótár elérései összhangban legyenek, de – amint azt a korábbiakban is említettük – nem használjuk ki ezt a lehetőséget, ugyanis ez a CPython megvalósításhoz kötődik.

```
except Finish:
    return
data = pickle.dumps(reply, 3)
self.wfile.write(SizeStruct.pack(len(data)))
self.wfile.write(data)
```

Ahányszor csak egy ügyfél kérelmet ad ki, létrehozunk egy új szálat a `RequestHandler` osztály új példányával, majd meghívjuk ennek a példánynak a `handle()` tagfüggvényét. A tagfüggvény belsejében az ügyféltől érkező adatokat a `self.rfile` objektum révén érhetjük el, a válasz adatait pedig a `self.wfile` objektumon keresztül küldhetjük el – mindkét objektumot a `socketserver` modultól kapjuk, és ilyenkor már nyitva állnak, készen a használatra.

A `struct.Struct` egy egész értékű bájtszámláló, amelyre a „hossz + befőtt" formátumhoz szükségünk van – ilyen alakban cserélünk adatokat az ügyfél és a kiszolgáló között.

Először beolvasunk négy bájtot, és az eredményt a `size` egész változóba csomagoljuk ki, így már tudjuk az átküldött befőtt hosszát. Ezt követően beolvasunk `size` darab bájtot, és tároljuk őket a `data` változóban. Az olvasás alatt blokkoljuk a többi hálózati műveletet. Esetünkben tudjuk, hogy az adatok egy rendezett sorozatot adnak ki, amelynek az első eleme a művelet, az utána következők pedig annak paraméterei – ebben a protokollban állapodtunk ugyanis meg az ügyfelekkel.

A `try` blokk belsejében hozzájutunk a kért műveletnek megfelelő lambdafüggvényhez. A zárolás segítségével letiltjuk a `Call` szótár elérését, jóllehet sokan ezt túlzott óvatosságnak tekinthetik. Szokásainkhoz híven a zárolás hatókörén belül a műveleteinket a lehető legszükségesebbekre korlátozzuk, ami esetünkben egy szótárbeli keresést jelent, amellyel hozzájutunk a kívánt függvény hivatkozásához. Ha megkaptuk, meg is hívjuk, első paraméterként átadva számára a `self` objektumot, ezt követően pedig a `data` sorozat további elemeit. Fontos megértenünk, hogy itt egy egyszerű függvényhívásról van szó, ezért a Python nem adja át automatikusan a `self` paramétert. Persze ez nem jelent gondot, hiszen itt, a lambdafüggvényen belül ezt egyszerű paraméterként használjuk a híváshoz. A végeredmény a `self.`*`tagfüggvény`*`(*data[1:])` hívás, ahol a *`tagfüggvény`* megfelel a `data[0]`-ban átadott műveletnek.

Amennyiben a művelet a leállítás, a `Finish` egyéni kivételt váltjuk ki a `shutdown()` tagfüggvényben – ilyenkor tudjuk, hogy az ügyfél nem várhat semmiféle választ, így egyszerűen visszatérünk a `return` utasítással. A többi művelet esetében azonban befőzzük a megfelelő tagfüggvény hívásának eredményét (a 3-as befőttprotokollal), majd kiírjuk a befőtt méreteit és magát a befőzött adatcsomagot.

```
def get_car_details(self, license):
    with self.CarsLock:
        car = copy.copy(self.Cars.get(license, None))
```

```
        if car is not None:
            return (True, car.seats, car.mileage, car.owner)
        return (False, "This license is not registered")
```

A fenti tagfüggvény először megpróbál hozzájutni az autók adatainak zárolásához – amíg ezt meg nem szerzi, magánál tartja a vezérlést. A megadott rendszámhoz tartozó autó adatait a `dict.get()` tagfüggvénnyel olvassa ki, második paraméterként a `None` értéket adva át – ha nincs ilyen autó, az eredmény `None`. A program azonnal átmásolja az autó adatait, és a `with` utasítás véget is ér, ezzel a zárolás a lehető legrövidebb ideig van érvényben. Jóllehet az olvasás nem módosítja a kiolvasott adatokat, mégis csak egy módosuló gyűjteménnyel dolgozunk, amelyet egy másik tagfüggvény egy másik szálban esetleg meg szeretne változtatni éppen az olvasás közben – a zárolással ennek elejét vesszük. A zárolás hatókörén kívül ebben a pillanatban rendelkezünk az autó adatainak másolatával (illetve, ha nincs meg az autó, a `None` értékkel), amelyeket nyugodtan kezelhetünk anélkül, hogy bármely szálat feltartanánk.

Az autóregisztráció műveletkezelő tagfüggvényeihez hasonlóan egy rendezett sorozatot adunk vissza, amelynek az első eleme egy logikai siker/kudarc jelző, a többi elem pedig változó lehet az adott művelettől függően. E tagfüggvények egyikének sem kell aggódnia amiatt, hogy milyen adatokat kap, sőt egyáltalán nem kell foglalkoznia vele azon túl, hogy amit kap, az „egy rendezett sorozat, amelynek az első eleme logikai érték" – a hálózati műveleteket ugyanis a `handle()` tagfüggvény végzi.

```
    def change_mileage(self, license, mileage):
        if mileage < 0:
            return (False, "Cannot set a negative mileage")
        with self.CarsLock:
            car = self.Cars.get(license, None)
            if car is not None:
                if car.mileage < mileage:
                    car.mileage = mileage
                        return (True, None)
                        return (False, "Cannot wind the odometer back")
                return (False, "This license is not registered")
```

Ebben a tagfüggvényben az első vizsgálatot zárolás nélkül is elvégezhetjük – ha azonban a kilométerszám nem negatív, szükségünk lesz a zárolásra, majd hozzá kell jutnunk a megfelelő autóhoz, és ha ez megvan (vagyis ismert rendszámot adtunk meg), még a zárolás hatókörében maradva megváltoztatjuk az óraállást, vagy visszaadunk egy hibaüzenetet. Ha nincs a rendszámhoz tartozó autó a rendszerben (vagyis a `car` értéke `None`), kilépünk a `with` hatóköréből, és egy hibaüzenetet adunk vissza.

Úgy tűnhet, hogy ha az adatok ellenőrzését az ügyfélnél végeznénk el, némi hálózati forgalmat megtakaríthatnánk, így például az ügyfél maga adhatna hibaüzenetet, ha negatív óraállással találkozik (vagy egyáltalán nem tenné lehetővé ilyen adat beírását). Jóllehet

az ügyfél megteheti ezt, továbbra is el kell végeznünk az ellenőrzést a kiszolgálón, hiszen sosem lehetünk biztosak az ügyfelek hibamentes működésében. Ráadásul, még ha tudjuk is, hogy az ügyfél az autó eredeti óraállását használja alapértelmezésként, akkor sem tekinthetjük a felhasználó által beírt adatot érvényesnek (hiába nagyobb az eredeti értéknél), hiszen időközben egy másik ügyfél módosíthatta. Következésképpen szükség van az ellenőrzésre a kiszolgálón, mégpedig a zárolás hatókörén belül.

A `change_owner()` tagfüggvény kódja hasonló, így nem foglalkozunk vele részletesebben.

```
def new_registration(self, license, seats, mileage, owner):
    if not license:
        return (False, "Cannot set an empty license")
    if seats not in {2, 4, 5, 6, 7, 8, 9}:
        return (False, "Cannot register car with invalid seats")
    if mileage < 0:
        return (False, "Cannot set a negative mileage")
    if not owner:
        return (False, "Cannot set an empty owner")
    with self.CarsLock:
        if license not in self.Cars:
            self.Cars[license] = Car(seats, mileage, owner)
            return (True, None)
    return (False, "Cannot register duplicate license")
```

Ezúttal is lehetőségünk van arra, hogy több hibaellenőrzési lépést elvégezzünk a zárolás előtt, de ha minden adat érvényes, a zárolás nem kerülhető el. Ha a rendszám nem található a `RequestHandler.Cars` szótárban (márpedig ha minden rendben folyik, ez a helyzet, hiszen itt új rendszámról van szó), létrehozunk egy új `Car` objektumot, és tároljuk a szótárban. Mindezt egyazon zárolás hatókörében kell megtennünk, ugyanis nem szabad megengednünk, hogy más ügyfél is bejegyezzen egy autót ugyanezzel a rendszámmal a rendszám ellenőrzése és az új autó bejegyzése közti holtidőben.

```
def shutdown(self, *ignore):
    self.server.shutdown()
    raise Finish()
```

Amennyiben a kért művelet a leállítás, meghívjuk a kiszolgáló `shutdown()` tagfüggvényét – ez meggátolja a további kérelmek fogadását, jóllehet a kiszolgáló mindaddig nem áll le, amíg még kezeli a korábban beérkezett kérelmeket. Ezt követően kiváltunk egy egyedi kivételt, amellyel jelezzük a `handler()` számára, hogy végeztünk – ennek hatására a `handler()` függvény futása véget ér anélkül, hogy választ adna az ügyfélnek.

Összefoglalás

Ebben a fejezetben meggyőződhettünk arról, hogy a hálózati ügyfelek és kiszolgálók létrehozása nem különösebben nehéz feladat a Pythonban, köszönhetően az alapkönyvtár hálózatkezelési moduljainak, de sokat segít a `struct` és a `pickle` modul is.

Az első részben egy olyan ügyfélprogramot írtunk, amely egyetlen, `handle_request()` nevű függvénnyel éri el a hálózatot – ez intézi a befőzhető adatok küldését és fogadását a kiszolgáló felé az általános „hossz + befőtt" adatformátumban. A második részben egy kiszolgáló-alaposztályt hoztunk létre a `socketserver` modul osztályainak leszármazottjaként, és megtanultuk, hogy miként valósítsunk meg egy kérelemkezelő osztályt, amely képes az ügyfelektől érkező kérelmek feldolgozására. A hálózati működés lebonyolítása egyetlen, `handle()` nevezetű tagfüggvényre hárul, amely képes az ügyféllel kapcsolatba lépve tetszőleges befőzhető adatok küldésére és fogadására.

A `socket`, a `socketserver`, valamint más, a témakörben érintett modulok (mint az `asyncore`, az `asynchat` vagy az `ssl`) a bemutatottaknál jelentősen tágabb lehetőségekkel rendelkeznek. Ha pedig az alapkönyvtár hálózati képességei nem bizonyulnak elegendőnek, vagy nem elég magas szintűek, érdemes egy pillantást vetnünk a külső fejlesztésű Twisted hálózati környezetre (`www.twistedmatrix.com`).

Gyakorlatok

A következő gyakorlatokban a korábbiakban elkészített ügyfél- és kiszolgálóprogramokat kell módosítanunk. Mindehhez kevesebb gépelésre, de kicsit több furfangra lesz szükség.

1. Készítsünk másolatot a `car_registration_server.py` és a `car_registration.py` fájlokról, majd módosítsuk őket úgy, hogy az adatcseréhez használt protokoll esetében hálózati szintű változatkezelést alkalmazzanak. Ezt megtehetjük például úgy, hogy két egész számot (hossz, protokollváltozat) adunk át a struktúrában egy helyett.
   A feladat végrehajtásához nagyjából tíz sor beillesztésére, illetve módosítására lesz szükség az ügyfélprogram `handle_request()` függvényében, valamint körülbelül tizenhat sor beillesztésére, illetve módosítására a kiszolgáló `handle()` tagfüggvényében. Mindebbe beleértjük azt a kódrészletet is, amely gondoskodik arról az esetről, amikor a beolvasott protokollváltozat nem egyezik a várttal.
   Ennek és az ezt követő feladatoknak egy lehetséges megoldását megtalálhatjuk a `car_registration_ans.py` és a `car_registration_server_ans.py` fájlokban.
2. Készítsünk másolatot a `car_registration_server.py` programról (vagy használjuk az 1. feladatban elkészült változatot), és tegyünk benne elérhetővé egy új, `GET_LICENSES_STARTING_WITH` nevű műveletet, amely egyetlen paramétert, egy karakterláncot vár, és megkeres minden olyan rendszámot, amely ezzel a karakterlánccal kezdődik. A műveletet megvalósító tagfüggvénynek mindig egy (`True`,

rendszámok listája) rendezett párt kell visszaadnia – ezúttal nem következhet be hiba (`False`), ugyanis ha nincs találat, az nem hibát jelent: ilyenkor az eredmény egyszerűen a `True` és egy üres lista.
A rendszámok (vagyis a `RequestHandler.Cars` szótár kulcsai) kiolvasását zárolás hatálya alatt végezzük, a többi műveletet azonban a zárolás hatókörén kívül helyezzük el, a blokkolás időtartamának leszorítása érdekében. A megfelelő rendszámok megkeresésének egyik igen hatékony módszere, ha rendezzük a kulcsokat, majd a `bisect` modullal megkeressük az első egyezést, és innen haladunk tovább. Választhatjuk azt a megközelítést is, amelynek során egymás után végighaladunk a rendszámokon, és tároljuk azokat, amelyek az adott karakterlánccal kezdődnek, például listaértelmezés használatával.
Az új `import` műveleten túl a `Call` osztályban szükségünk lesz pár új sorra a művelet megvalósításához. A műveletet levezénylő tagfüggvény tíznél kevesebb sorban megírható – nem különösebben nehéz feladat, de azért oda kell figyelnünk. Egy lehetséges megoldást (amely a `bisect` modult alkalmazza) találhatunk a `car_registration_server_ans.py` fájlban.

3. Készítsünk másolatot a `car_registration.py` programról (vagy használjuk az 1. feladatban elkészült változatot), és módosítsuk úgy, hogy kihasználja az új kiszolgáló (`car_registration_server_ans.py`) lehetőségeit. Mindehhez meg kell változtatnunk a `retrieve_car_details()` függvényt úgy, hogy ha a felhasználó ismeretlen rendszámot ír be, lehetővé tegyük a számára, hogy egy a rendszám kezdetének beírása után kapott listából válasszon. Íme egy lehetséges párbeszéd a felhasználó és az ügyfél között. (Feltételezzük, hogy a kiszolgáló már fut, továbbá némiképp átszerkesztettük a menüt, hogy elférjen az oldalon, a felhasználó válaszait pedig **félkövérrel** emeltük ki.)

```
(C)ar  (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License: da 4020
License: DA 4020
Seats:   2
Mileage: 97181
Owner:   Jonathan Lynn
(C)ar  (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License [DA 4020]: z
This license is not registered
Start of license: z
No licence starts with Z
Start of license: a
(1) A04 4HE
(2) A37 4791
(3) ABK3035
Enter choice (0 to cancel): 3
License: ABK3035
Seats:   5
Mileage: 17719
Owner:   Anthony Jay
```

A módosításhoz egy sor törlésére, és nagyjából húsz új sor beillesztésére van szükség. A feladat nehézségét az adja, hogy lehetővé kell tennünk, hogy a felhasználó bármely szinten kiléphessen, vagy folytathassa a keresést. Fontos, hogy az új lehetőséget minden elképzelhető esetben ellenőrizzük (nincs olyan rendszám, amely a megadott karakterlánccal kezdődik; egy ilyen van; illetve kettő vagy több találat adódik).
Egy lehetséges megoldást a `car_registration_ans.py` fájlban találhatunk.

11

Adatbázis-programozás

- DBM-adatbázisok
- SQL-adatbázisok

A legtöbb programfejlesztő számára az *adatbázis* kifejezés valójában relációsadatbázis-kezelő rendszert (RDBMS, Relational Database Management System) jelent, amelyben táblák (vagyis a táblázatokhoz hasonló rácsok) szerepelnek, amelyek sorai a bejegyzéseknek, oszlopai pedig a mezőknek felelnek meg. A táblákat és az azokat megtöltő adatokat az SQL segítségével hozhatjuk létre és kezelhetjük. A Python biztosít egy API-t (alkalmazásprogramozási felületet) az SQL-adatbázisok kezeléséhez, emellett a nyelv szokásos programcsomagjában hozzájutunk az SQLite 3 adatbázishoz is.

Az adatbázisok másik típusa a *DBM* (Database Manager, adatbázis-kezelő), amely tetszőleges számú kulcs-érték párt tartalmazhat. A Python alapkönyvtára felületet biztosít számos DBM-hez, amelyek között kifejezetten Unix rendszereken működőket is találhatunk.
A DBM-ek működése hasonló a Pythonban megismert szótárakéhoz, azzal a különbséggel, hogy a memória helyett jellemzően a merevlemezen érhetjük el őket, kulcsaik és értékeik pedig minden esetben `bytes` objektumok, amelyek hosszára olykor korlátozások állhatnak fenn. A fejezetünk első részében bemutatott `shelve` modul kényelmes DBM-felületet ad, amely lehetővé teszi, hogy kulcsok gyanánt karakterláncokat, értékekként pedig (befőzhető) objektumokat szerepeltessünk.

Ha a rendelkezésre álló adatbázis-kezelők és az SQLite adatbázis nem bizonyulnak elegendőnek, a Python Package Indexen (`pypi.python.org/pypi`) számos adatbázisokhoz kapcsolódó programcsomagot találhatunk, például a bsddb-t („Berkeley-adatbázis"), továbbá olyan objektumrelációs leképezőket, mint az SQLAlchemy (`www.sqlalchemy.org`), de kaphatunk felületeket az ismertebb ügyfél-kiszolgáló adatbázisokhoz is, mint a DB2, az Informix, az Ingres, a MySQL, az ODBC, valamint a PostgreSQL.

Fejezetünkben egy DVD-ket nyilvántartó adatbázis két változatát készítjük el, amely a lemezek listájában tárolja a DVD-k címét, kiadási évét, percben mért hosszát, valamint a film rendezőjét. Az első változatban egy DBM-et használunk (a `shelve` modul révén)

az adatok tárolására, míg a másodikban az SQLite adatbázis szolgáltatásait vesszük igénybe. Mindkét program lehetőséget ad arra, hogy az adatainkat egyszerű XML formátumban tároljuk és töltsük be, így a DVD-k adatait könnyedén átvihetjük egyik programból a másikba. Az SQL alapú változat lehetőségei némiképp tágabbak, mint a DBM-re épülő társánál elérhetők, és az adatok szerkezete is valamelyest átláthatóbb.

DBM-adatbázisok

A `shelve` modul burkolót biztosít egy DBM köré, amely lehetővé teszi, hogy úgy érintkezzünk vele, mintha egy szótárral tennénk – amennyiben kizárólag karakterláncokat használunk kulcsok és befőzhető objektumokat értékek gyanánt. A háttérben a modul a kulcsokat és értékeket `bytes` objektumokká alakítja, és viszont.

Mivel a `shelve` modul az elérhető legjobb háttér-DBM-et használja, előfordulhat, hogy az egyik gépen mentett DBM-fájlt egy másik gép programja nem képes olvasni, ugyanis az nem rendelkezik ugyanazzal a DBM-mel, mint a másik gépen futó társa. Gyakori megoldás erre, hogy lehetővé teszik az XML-fájlok ki- és bevitelét, amelyeknek a formátuma minden rendszeren értelmezhető. Jelen esetben, a `dvds-dbm.py` programban éppen ezt tettük.

A kulcsok szerepét a DVD-k címei játsszák, az értékek pedig a rendezőt, a megjelenés évét, valamint a film hosszát tartalmazó rendezett sorozatok. A `shelve` modulnak köszönhetően nem kell az adatok átalakításával foglalkoznunk – a DBM objektumot pontosan úgy kezelhetjük, mint egy szótárat.

Mivel a program szerkezete hasonló a korábban látott interaktív, menüvezérelt programokéhoz, most csak azokra a részeire összpontosítunk, amelyek részt vesznek az adatbázis-kezelésben. Az alábbiakban ezért csak egy részletet mutatunk be a `main()` függvényből, amelyből kihagytuk a menükezelés kódját:

```
db = None
try:
    db = shelve.open(filename, protocol=pickle.HIGHEST_PROTOCOL)
    ...
finally:
    if db is not None:
        db.close()
```

Itt megnyitjuk (illetve, ha még nem létezik, létrehozzuk) a megadott DBM-fájlt írásra és olvasásra. Az egyes elemek értékeit befőttként tároljuk a megadott befőttprotokollal – a már meglevő elemek akkor is beolvashatók, ha korábbi protokollal mentettük őket, ugyanis olvasás esetén a Python képes önállóan kitalálni az alkalmazott protokollt. Ha mindennel elkészültünk, bezárjuk a DBM-et – vagyis kiürítjük a DBM belső átmeneti tárát, meggyőződünk róla, hogy a lemezen tárolt fájlba bekerültek a módosítások, majd bezárjuk a fájlt.

A program lehetővé teszi, hogy beillesszük, szerkesszük, felsoroljuk, töröljük, betöltsük és mentsük a DVD-k adatait. Az XML formátumú be- és kivitel leírásától most eltekintünk, hiszen hasonló feladattal már találkoztunk a 7. fejezetben. Mindemellett a beillesztésen kívül elhagyjuk a felület kódjának többségét, ugyanis ezeket is láthattuk már más környezetben.

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director = Console.get_string("Director", "director")
    if not director:
        return
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   minimum=0, maximum=60*48)
    db[title] = (director, year, duration)
    db.sync()
```

Ez a függvény, hasonlóan a menüből meghívott társaihoz, egyedüli paraméterként egy `db` nevű DBM-objektumot vár. Kódjának nagy részében a DVD adatainak beolvasásával foglalkozik, az utolsó előtti sor pedig tárolja a kapott kulcs-érték elemet a DBM-fájlban, ahol a DVD címe szerepel kulcsként, az érték pedig a rendező, a megjelenés éve és a hossz hármasa egy a `shelve` által előállított befőttben.

A Python szokásos egységesítési lehetőségeinek folyományaként a DBM-ek ugyanazt az API-t adják, mint a szótárak, így nem találkozhatunk sok újdonsággal a korábban már látott `shelve.open()` függvényen, valamint a `shelve.Shelf.sync()` tagfüggvényen kívül, amellyel kiürítjük a `shelve` belső átmeneti tárát, és összhangba hozzuk a lemezen tárolt adatokat a módosításokkal – ez esetben egy új elem beillesztéséről van szó.

```
def edit_dvd(db):
    old_title = find_dvd(db, "edit")
    if old_title is None:
        return
    title = Console.get_string("Title", "title", old_title)
    if not title:
        return
    director, year, duration = db[old_title]
    ...
    db[title] = (director, year, duration)
    if title != old_title:
        del db[old_title]
    db.sync()
```

Ahhoz, hogy a felhasználó szerkeszthessen egy DVD-t, először ki kell választania, hogy melyikhez szeretne hozzáférni. Itt mindössze a címét kell megadnia, hiszen a címek jelentik a kulcsokat, míg a hozzájuk tartozó értékek adják meg a további adatokat. Mivel az itt alkalmazandó kódra máshol is szükség van (nevezetesen a DVD-k eltávolításánál), ezt egy `find_dvd()` nevű függvényben különítettük el, amelyet az alábbiakban mutatunk be.
Ha megtaláltuk a DVD-t, beolvassuk a felhasználó módosításait, a már létező adatokat alkalmazva alapértelmezésként, hogy némiképpen felgyorsítsuk a párbeszédet. (A felhasználói felület kezelésének kódját nagyrészt elhagytuk, ugyanis ez szinte megegyezik a DVD beillesztésénél látottakkal.) Ha mindennel végeztünk, tároljuk az adatokat, mint a DVD beillesztésénél. Amennyiben a cím változatlan, felülírjuk a hozzá tartozó értéket, ha pedig megváltozott, új kulcs-érték párt hozunk létre, és töröljük az eredetit.

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    while True:
        matches = []
        start = Console.get_string(message, "title")
        if not start:
            return None
        for title in db:
            if title.lower().startswith(start.lower()):
                matches.append(title)
        if len(matches) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(matches) == 1:
            return matches[0]
        elif len(matches) > DISPLAY_LIMIT:
            print("Too many dvds start with {0}; try entering "
                  "more of the title".format(len(matches)))
            continue
        else:
            for i, match in enumerate(sorted(matches, key=str.lower)):
                print("{0}: {1}".format(i + 1, match))
            which = Console.get_integer("Number (or 0 to cancel)",
                                        "number", minimum=1, maximum=len(matches))
            return matches[which - 1] if which != 0 else None
```

A DVD-k gyors és egyszerű felkutatásának érdekében elkérjük a felhasználótól a keresett cím első néhány karakterét, majd végighaladunk az adatbázison, egyezéseket keresve. Ha egyetlen találat van, egyszerűen visszaadjuk. Ha több egyezést is találunk (de kevesebbet, mint a `DISPLAY_LIMIT` értéke, amit másutt adunk meg a programban), kiírjuk őket a képernyőre a betűállást figyelmen kívül hagyó sorrendben, mindegyiküket egy számmal azonosítva – a felhasználó így egy szám beírásával választhatja ki a megfelelő címet. (A `Console.get_integer()` függvény elfogadja a 0 értéket még akkor is, ha a minimum nullánál nagyobb, így a 0-t használhatjuk kilépési érték gyanánt. Ezt a viselkedést

kikapcsolhatjuk az `allow_zero=False` átadásával. Az ENTER lenyomásával – ha nem írunk be semmit – nem érhetjük el ugyanezt a hatást, hiszen ez eleve azt jelenti, hogy elfogadjuk az alapértelmezést.)

```
def list_dvds(db):
    start = ""
    if len(db) > DISPLAY_LIMIT:
        start = Console.get_string("List those starting with "
                                   "[Enter=all]", "start")
    print()
    for title in sorted(db, key=str.lower):
        if not start or title.lower().startswith(start.lower()):
            director, year, duration = db[title]
            print("{0} ({1}) {2} minute{3}, by {4}".format(
                title, year, duration, Util.s(duration), director))
```

Az összes DVD felsorolása (illetve azoké, amelyek egy adott karakterlánccal kezdődnek) egyszerűen abból áll, hogy sorra vesszük az adatbázis elemeit.

A `Util(s)` függvény kódja egyszerűen `s = lambda x: "" if x == 1 else "s"`; így a visszatérési érték az „`s`", amennyiben az `x` értéke nem éppen egy perc.

```
def remove_dvd(db):
    title = find_dvd(db, "remove")
    if title is None:
        return
    ans = Console.get_bool("Remove {0}?".format(title), "no")
    if ans:
        del db[title]
        db.sync()
```

Egy adott DVD eltávolításához keressük meg, hogy melyiket választotta a felhasználó, kérjünk tőle megerősítést, majd ha valóban törölni akarja, távolítsuk el az elemet a DBM-ből.

Láttuk tehát, miként nyithatunk meg (vagy hozhatunk létre) egy DBM-fájlt a `shelve` modul segítségével, hogyan illesszünk be elemeket ebbe az adatbázisba, miként szerkesszük, vegyük sorra, illetve töröljük őket. Sajnos az adatszerkezetünkben rejlik egy apró hiba. A rendezők nevei több változatban is szerepelhetnek, ami könnyen ellentmondásokhoz vezethet – például előfordulhat, hogy Danny DeVito nevét egyszer „Danny De Vito", egy másik filmnél pedig „Danny deVito" alakban adják meg. Az egyik lehetséges megoldás, ha két DBM-fájlt hozunk létre: egy központi DVD-tárolót, amelyben a címek a kulcsok, és az (év, hossz, rendezőazonosító) hármasok az értékek, továbbá egy rendezőket tartalmazó adatbázist, amelyben a rendezők azonosítói (egész számok) a kulcsok, a neveik pedig az értékek. Ezeket a gondokat orvosoljuk programunk következőkben bemutatott, SQL-adatbázist alkalmazó változatában is, amelyben két tábla szerepel: egy a DVD-k, egy pedig a rendezők számára.

SQL-adatbázisok

A legtöbb népszerű SQL-adatbázishoz létezik külső fejlesztőktől származó modul, a Pythonnal pedig hozzájutunk az `sqlite3` modulhoz (és az SQLite 3 adatbázishoz), így hát, ha kedvünk tartja, minden különösebb előkészület nélkül nekikezdhetünk az adatbázis-programozásnak. Az SQLite-ról tudnunk kell, hogy könnyűsúlyú SQL-adatbáziskezelő, és mint ilyen, számos lehetőség hiányzik belőle, ami például a PostgreSQL-ben elérhető, de igen kényelmes a prototípus-készítéshez, és sok esetben tökéletesen elegendő a saját céljainkra.

Annak érdekében, hogy könnyen válthassunk a háttéradatbázisok között, a PEP 249 (Python Database API Specification v2.0) meghatároz egy a DB-API 2.0 nevű API-szabványt, amelyet minden adatbázis-felületnek követnie illik – az `sqlite3` modul megteszi ezt a szívességet, de a külső modulok nem feltétlenül. Az API két központi objektumot – nevezetesen egy kapcsolati és egy kurzorobjektumot – határoz meg, amelyeknek a 11.1. és a 11.2. táblázatban bemutatott felületeket kell támogatniuk. Az `sqlite3` esetében a két objektum a szabványban megköveteltek mellett számos egyéb jellemzőt és tagfüggvényt is a rendelkezésünkre bocsát.

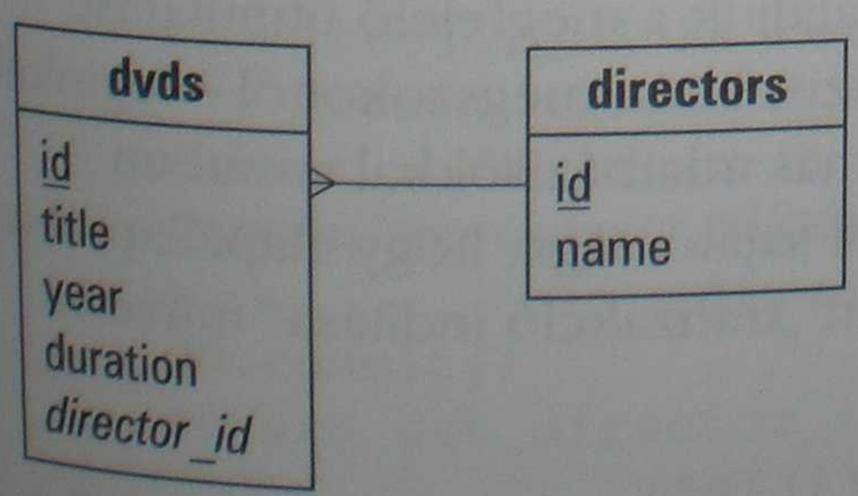
DVD-nyilvántartó programunk SQL-változata, amely a `dvds-sql.py` névre hallgat, a DVD-k adataitól elkülönítve tárolja a rendezőket a névváltozatok megjelenésének elkerülésére, és egy újabb menüponttal lehetővé teszi a rendezők listázását. A két táblát a 11.1. ábrán figyelhetjük meg. Programunk közel 300 soros, míg az előző részben bemutatott `dvds-dbm.py` kódja nem érte el a 200 sort. A különbség nagyrészt annak tudható be, hogy ezúttal a szótárműveletek helyett SQL-lekérdezéseket kell használnunk, és a program első futtatásakor létre kell hoznunk a táblákat.

11.1. táblázat *DB-API 2.0 kapcsolati objektumának tagfüggvényei*

Használati alak	Leírás
`db.close()`	Lezárja a kapcsolatot az adatbázissal (ezt a `db` objektum jeleníti meg, amelyhez a `connect()` függvény hívásával jutottunk hozzá).
`db.commit()`	Jóváhagyja az összes függő tranzakció eredményét az adatbázisban – abban az esetben, ha az adatbázis nem támogatja a tranzakciókat, a tagfüggvény nem végez semmilyen műveletet.
`db.cursor()`	Visszaad egy kurzorobjektumot az adatbázishoz, amelynek révén lekérdezéseket hajthatunk végre.
`db.rollback()`	Visszagörgeti az összes függőben levő tranzakciót a kezdeti állapotba – abban az esetben, ha az adatbázis nem támogatja a tranzakciókat, a tagfüggvény nem végez semmilyen műveletet.

11.2. táblázat *A DB-API 2.0 kurzorobjektumának jellemzői és tagfüggvényei*

Használati alak	Leírás
`c.arraysize`	A sorok (írható/olvasható) száma, amelyet a `fetchmany()` akkor ad vissza, ha nem határoztunk meg méretet.
`c.close()`	Bezárja a `c` kurzort. Ez automatikusan megtörténik, amikor a kurzor kikerül a hatókörből.
`c.description`	Hételemű rendezett sorozatok csak olvasható sora, amelynek elemei (`name`, `type_code`, `display_size`, `internal_size`, `precision`, `scale`, `null_ok`) alakban írják le a `c` kurzor egymást követő oszlopait.
`c.execute(sql,` *`paraméterek`*`)`	Végrehajtja az `sql` karakterláncban megadott SQL-lekérdezést, amelyben a helykitöltőket az átadott *`paraméterek`* sorozat vagy megfeleltetés elemeivel helyettesíti.
`c.executemany(sql,` *`paramétersorozat`*`)`	Végrehajtja az `sql` karakterláncban megadott SQL-lekérdezést sorozatok sorozatának minden elemével. Nem alkalmazhatjuk olyan műveletek esetében, amelyek eredményhalmazt adnak (ilyenek a `SELECT` utasítások).
`c.fetchall()`	Visszaadja azoknak a soroknak a sorozatát, amelyeket még nem olvastunk ki (itt szó lehet akár az összes sorról is).
`c.fetchmany(`*`méret`*`)`	Sorok sorozatát adja vissza (az egyes sorok önmagukban is sorozatok). A *`méret`* alapértelmezett értéke a `c.arraysize`.
`c.fetchone()`	Visszaadja a lekérdezés eredményhalmazának következő sorát, illetve a `None` értéket, ha az eredményhalmaz végére ért. Amennyiben nincs eredményhalmaz, kivételt vált ki.
`c.rowcount`	A legutóbbi műveletben (`SELECT`, `INSERT`, `UPDATE` vagy `DELETE`) visszakapott sorok számának csak olvasható értéke, illetve -1, amennyiben az érték nem érhető el, vagy nem meghatározható.



11.1. ábra
A DVD-nyilvántartó program adatbázisának felépítése

A `main()` függvény felépítése hasonló az előzőekben látotthoz, de ezúttal az egyedi `connect()` függvénnyel kapcsolódunk az adatbázishoz.

```
def connect(filename):
    create = not os.path.exists(filename)
    db = sqlite3.connect(filename)
    if create:
```

```
    cursor = db.cursor()
    cursor.execute("CREATE TABLE directors ("
        "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
        "name TEXT UNIQUE NOT NULL)")
    cursor.execute("CREATE TABLE dvds ("
        "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
        "title TEXT NOT NULL, "
        "year INTEGER NOT NULL, "
        "duration INTEGER NOT NULL, "
        "director_id INTEGER NOT NULL, "
        "FOREIGN KEY (director_id) REFERENCES directors)")
    db.commit()
return db
```

Az `sqlite3.connect()` függvény létrehoz egy adatbázisobjektumot, miután megnyitotta a számára átadott adatbázisfájlt, illetve, amennyiben az még nem létezett, létrehozta. Ezért, mielőtt meghívnánk ezt a tagfüggvényt, feljegyezzük, hogy az adatbázisunkat a kezdetektől építjük-e fel, ugyanis ez esetben létre kell hoznunk azokat a táblákat, amelyekre az adatbázis működése épül. A lekérdezéseket az adatbázis kurzorobjektumán végezhetjük el, amelyhez az adatbázisobjektum `cursor()` tagfüggvényével juthatunk hozzá.

Vegyük észre, hogy mindkét táblát olyan ID mezővel hoztuk létre, amelyre érvényes az `AUTOINCREMENT` megszorítás. Ez azt jelenti, hogy az SQLite automatikusan egyedi értékekkel tölti fel ezt az oszlopot, így új bejegyzések beillesztésénél ennek a mezőnek a kitöltését ráhagyhatjuk a rendszerre.

Az SQLite csak bizonyos adattípusokat támogat – lényegében csak logikai értékeket, számokat és karakterláncokat használhatunk –, de az elérhető típusok skálája bővíthető az úgynevezett „átalakítók" segítségével, legyen szó beépített átalakítókról, mint a dátum és a dátum-idő értékeknél, illetve egyediekről, amelyekkel olyan adattípust jeleníthetünk meg, amilyet csak akarunk. A DVD-nyilvántartó programhoz nincs szükség erre a lehetőségre, de ha élnénk vele, az `sqlite3` leírásában megtaláljuk a megfelelő útmutatást.
Az idegen kulcsok használati alakja eltér a más adatbázisokban megszokottól – leginkább a szándékainkat hivatottak jelezni, ugyanis az SQLite más adatbázisokkal szemben kikényszeríti a kapcsolati épséget. Az `sqlite3` további sajátossága, hogy alapállapotban támogatja a beleértett tranzakciókat, így nincs kifejezett „tranzakció indítása" művelet.

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director = Console.get_string("Director", "director")
    if not director:
        return
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
```

```
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor = db.cursor()
    cursor.execute("INSERT INTO dvds "
                   "(title, year, duration, director_id) "
                   "VALUES (?, ?, ?, ?)",
                   (title, year, duration, director_id))
    db.commit()
```

A függvény kódjának első sorai megegyeznek a `dvds-dbm.py` programbeli társánál látottakkal, de amint összegyűjtöttük az adatokat, a működése más irányt vesz. A felhasználó által megadott rendező szerepelhet a `directors` táblában, de megeshet az is, hogy nincs jelen – a `get_and_set_director()` függvény nagyszerűen kezeli a helyzetet, hiszen utóbbi esetben beilleszti a rendező nevét az adatbázisba, és bármi történjen is, visszaadja a rendező azonosítóját, amelyet azután elhelyezhetünk a `dvds` táblában. Ha minden adat a rendelkezésünkre áll, végrehajtunk egy `INSERT` SQL-utasítást. A bejegyzés azonosítóját nem nekünk kell megadnunk, hiszen az SQLite automatikusan szolgáltatja.

A lekérdezésben helykitöltőként kérdőjeleket használtunk, amelyek mindegyikének helyére az SQL-utasítás karakterlánca után álló sorozat megfelelő eleme kerül. Mindemellett névvel ellátott helykitöltőket is használhatunk, amire a bejegyzések szerkesztésénél láthatunk majd példát. Jóllehet módunk van arra is, hogy elkerüljük a helykitöltők használatát, és magunk formázzuk az SQL-karakterláncot a megfelelő adatokkal együtt, mégis ajánlatos helykitöltőket alkalmaznunk, ugyanis így az adatbázismodul leveszi a vállunkról az adatelemek kódolásának és levédésének terhét. A helykitöltők használatának további előnye a megnövekedett biztonság, hiszen ez meggátolja a rosszindulatú behatolókat abban, hogy tetszőleges SQL-kódot helyezzenek el a lekérdezéseinkben.

```
def get_and_set_director(db, director):
    director_id = get_director_id(db, director)
    if director_id is not None:
        return director_id
    cursor = db.cursor()
    cursor.execute("INSERT INTO directors (name) VALUES (?)",
                   (director,))
    db.commit()
    return get_director_id(db, director)
```

A fenti függvény visszaadja a paraméterként meghatározott rendező azonosítóját, szükség esetén létrehozva számára egy új bejegyzést az adatbázisban. Ha beillesztettünk egy bejegyzést, az azonosítóját a `get_director_id()` függvénnyel olvassuk ki – ahogy ezzel a függvény első sorában is megpróbálkoztunk.

```
def get_director_id(db, director):
    cursor = db.cursor()
    cursor.execute("SELECT id FROM directors WHERE name=?",
```

```
                    (director,))
    fields = cursor.fetchone()
    return fields[0] if fields is not None else None
```

A `get_director_id()` függvény a megadott rendező azonosítójával tér vissza, illetve ha a rendező nem szerepel az adatbázisban, a `None` értéket adja. A kiolvasáshoz a `fetchone()` tagfüggvényt használjuk, hiszen tudjuk, hogy legfeljebb egy találatot kapunk. (Egyetlen rendező neve sem szerepelhet többször, mivel a `directors` tábla `name` mezőjére a `UNIQUE` megszorítás érvényes, ráadásul új rendező beillesztése előtt mindig megvizsgáljuk, hogy szerepel-e már az adatbázisban.) A kiolvasó tagfüggvények mindig mezők sorozatát adják vissza (illetve a `None` értéket, ha az eredményhalmaz végére értek), még akkor is, ha egyébként egyetlen mezőt szeretnénk kiolvasni, mint a mi esetünkben.

```
def edit_dvd(db):
    title, identity = find_dvd(db, "edit")
    if title is None:
        return
    title = Console.get_string("Title", "title", title)
    if not title:
        return
    cursor = db.cursor()
    cursor.execute("SELECT dvds.year, dvds.duration, directors.name "
                   "FROM dvds, directors "
                   "WHERE dvds.director_id = directors.id AND "
                   "dvds.id=:id", dict(id=identity))
    year, duration, director = cursor.fetchone()
    director = Console.get_string("Director", "director", director)
    if not director:
        return
    year = Console.get_integer("Year", "year", year, 1896,
                               datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   duration, minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor.execute("UPDATE dvds SET title=:title, year=:year, "
                   "duration=:duration, director_id=:director_id "
                   "WHERE id=:id", locals())
    db.commit()
```

Ahhoz, hogy szerkesszük az adott DVD-hez tartozó bejegyzést, mindenekelőtt meg kell találnunk. Ha megvan, először lehetővé tesszük, hogy a felhasználó megváltoztassa a címét. Ezt követően kiolvassuk a többi mező tartalmát, így alapértelmezett értékekkel tehetjük kényelmesebbé a szerkesztést – ha a felhasználó leüti az ENTER-t, egyszerűen elfogadja az előző, alapértelmezett értéket. Ez esetben névvel ellátott helykitöltőket használtunk (a `:`*`név`* alakban), így a megfelelő értékeket egy megfeleltetés alakjában kell biztosítanunk. A `SELECT` utasításoknál egy frissen létrehozott szótárat alkalmazunk, az `UPDATE` esetében pedig ahhoz a szótárhoz folyamodunk, amelyet a `locals()` meghívásával kaptunk.

Használhattunk volna új szótárat mindkét esetben, de ekkor az UPDATE műveletnek a `locals()` eredménye helyett a `dict(title=title, year=year, duration=duration, director_id=director_id, id=identity)` értéket kellett volna átadnunk.

Ha megkaptuk minden mező értékét, és a felhasználó elvégezte a kívánt módosításokat, kiolvassuk a megfelelő rendezőazonosítót (illetve szükség esetén új bejegyzést hozunk létre a rendező számára), majd frissítjük az adatbázist az új adatokkal. A lehető legegyszerűbb megoldás mellett döntöttünk, így az összes mezőt frissítjük, nem csak azokat, amelyek megváltoztak.

A DBM-fájlra épülő megoldásban a DVD címét alkalmaztuk kulcsként, így ha ez megváltozott, létre kellett hoznunk egy új kulcs-érték párt, és törölnünk az eredetit. Ez esetben azonban minden DVD-bejegyzés egyedi azonosítóval rendelkezik, amelyet a beillesztésekor kapott, így minden további erőfeszítés nélkül megváltoztathatjuk bármelyik mező értékét.

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    cursor = db.cursor()
    while True:
        start = Console.get_string(message, "title")
        if not start:
            return (None, None)
        cursor.execute("SELECT title, id FROM dvds "
                       "WHERE title LIKE ? ORDER BY title",
                       (start + "%"))
        records = cursor.fetchall()
        if len(records) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(records) == 1:
            return records[0]
        elif len(records) > DISPLAY_LIMIT:
            print("Too many dvds ({0}) start with {1}; try entering "
                  "more of the title".format(len(records), start))
            continue
        else:
            for i, record in enumerate(records):
                print("{0}: {1}".format(i + 1, record[0]))
            which = Console.get_integer("Number (or 0 to cancel)",
                                        "number", minimum=1, maximum=len(records))
            return records[which - 1] if which != 0 else (None, None)
```

Ez a függvény ugyanazt a feladatot végzi el, mint a `dvds-dbm.py` program `find_dvd()` függvénye, eredményként pedig egy (cím, DVD azonosító) rendezett párt ad, ha a keresés sikeres volt, egyébként pedig a `(None, None)` párhoz jutunk. Ezúttal azonban használhatjuk az SQL % műveletét, így csak azokat a bejegyzéseket olvassuk ki, amelyekre valóban szükségünk van.

Mivel arra számítunk, hogy a találatok száma kicsi lesz, az összeset egyszerre olvassuk ki sorozatok sorozata alakjában. Amennyiben a megfelelő bejegyzések száma egynél több, viszont nem túlzottan sok a megjelenítéshez, mindegyiküket kiírjuk egy-egy számmal azonosítva, amelyeket a felhasználó kiválaszthat – hasonlóan a `dvds-dbm.py` programban látottakhoz.

```
def list_dvds(db):
    cursor = db.cursor()
    sql = ("SELECT dvds.title, dvds.year, dvds.duration, "
           "directors.name FROM dvds, directors "
           "WHERE dvds.director_id = directors.id")
    start = None
    if dvd_count(db) > DISPLAY_LIMIT:
        start = Console.get_string("List those starting with "
                                   "[Enter=all]", "start")
        sql += " AND dvds.title LIKE ?"
    sql += " ORDER BY dvds.title"
    print()
    if start is None:
        cursor.execute(sql)
    else:
        cursor.execute(sql, (start + "%",))
    for record in cursor:
        print("{0[0]} ({0[1]}) {0[2]} minutes, by {0[3]}".format(
              record))
```

Az egyes DVD-k adatainak kiírásához egy `SELECT` lekérdezést végzünk, amely egyesíti a két táblát, egy második elemet is elhelyezve a `WHERE` záradékban, amennyiben a bejegyzések száma (ezt a `dvd.count()` függvénnyel kaphatjuk meg) több, mint amennyi megjeleníthető. Ezt követően végrehajtunk egy lekérdezést, és végighaladunk az eredményeken. A kapott bejegyzések olyan sorozatok, amelyeknek a mezői egyezést adnak a `SELECT` lekérdezéssel.

```
def dvd_count(db):
    cursor = db.cursor()
    cursor.execute("SELECT COUNT(*) FROM dvds")
    return cursor.fetchone()[0]
```

Ezeket a kódsorokat egy önálló függvényben helyeztük el, mivel a program több pontján is szükségünk van rájuk.

A `list_directors()` függvény kódját nem mutatjuk be, mivel szerkezetében igen hasonló a `list_dvds()` függvényhez, mindössze egyszerűbb, hiszen csak egy mezőt (ez pedig a `name`) ír ki.

```
def remove_dvd(db):
    title, identity = find_dvd(db, "remove")
    if title is None:
```

```
    return
ans = Console.get_bool("Remove {0}?".format(title), "no")
if ans:
    cursor = db.cursor()
    cursor.execute("DELETE FROM dvds WHERE id=?", (identity,))
    db.commit()
```

Ezt a függvényt akkor hívjuk meg, ha a felhasználó törölni kíván egy bejegyzést. A szerkezete nagyban hasonlít a `dvds-dbm.py` programbeli megfelelőjéhez.

Ezzel végére értünk a `dvds-sql.py` program bemutatásának, amelynek során megtanultuk, hogy miként hozhatunk létre adatbázis-táblákat, hogyan választhatunk ki bejegyzéseket és vehetjük sorra őket, továbbá miként illesszük be, frissítsük vagy töröljük a bejegyzéseket. Az `execute()` tagfüggvénnyel tetszőleges SQL-utasítást végrehajthatunk, amelyet a háttérben működő adatbázis támogat.

Az SQLite az itt szükségesnél jóval többre képes, így alkalmazhatjuk az automatikus jóváhagyási módot (és más tranzakcióvezérlési lehetőségeket), továbbá olyan függvényeket is írhatunk, amelyeket az SQL-kódban futtathatunk. Létrehozhatunk továbbá gyártófüggvényeket is, amelyekkel meghatározhatjuk, hogy milyen alakban kapjuk vissza a kiolvasott bejegyzéseket (a mezők sorozata helyett szótárat vagy egyedi típust is meghatározhatunk így). Ráadásul lehetőséget kapunk arra is, hogy a memóriában tárolt SQLite adatbázisokat használjunk, csak adjuk át a `":memory:"` karakterláncot fájlnév gyanánt.

Összefoglalás

A 7. fejezetben számos módot megismertünk a lemezen tárolt fájlok adatainak mentésére és betöltésére, itt pedig azt is megtudtuk, hogyan kezeljük azokat az adattípusokat, amelyek a tartalmukat a memória helyett lemezen raktározzák.

A DBM-fájlok használatánál a `shelve` modul igen kényelmes megoldást ad, hiszen karakterlánc–objektum párok tárolására alkalmas. Amennyiben teljes mélységében át szeretnénk látni a rendszert, természetesen közvetlenül is hozzáférhetünk a háttérben működő bármely DBM-hez. A `shelve` modul és a DBM-ek nagyszerű tulajdonsága, hogy a szótárak API-ját használják, így az elemek kiolvasása, beillesztése, szerkesztése és törlése gyerekjáték, sőt így azokat a programokat, amelyek szótárakat használnak, könnyedén rávehetjük az adatbázisok elérésére. A DBM-ek használatának apró szépséghibája, hogy a relációs adatok esetében minden kulcs-érték táblához külön DBM-fájlt kell készítenünk, míg az SQLite ezeket az adatokat egyetlen fájlban tárolja.

Az SQL-adatbázisok esetében az SQLite igen hasznos lehet a prototípus-készítésnél, de gyakran önállóan is használható, mindemellett az az előnye mindenképp megvan, hogy megkapjuk a Python szokásos programcsomagjával. Láthattuk, hogyan juthatunk hozzá

az adatbázisobjektumhoz a `connect()` függvénnyel, és miként hajthatunk végre SQL-lekérdezéseket (mint a `CREATE TABLE`, a `SELECT`, az `INSERT`, az `UPDATE` vagy a `DELETE`) az adatbázis kurzorának `execute()` tagfüggvényével.

A Python az adattárolási lehetőségek széles skáláját biztosítja a számunkra, legyen szó memóriabeli vagy merevlemezes tárolásról. A bináris, szöveg- és XML-fájloktól a befőtteken át a DBM- és az SQL-adatbázisokig minden rendelkezésre áll, hogy az adott feladathoz a megfelelő megoldást válasszuk.

Gyakorlat

Készítsünk interaktív konzolprogramot könyvjelzők listájának kezelésére, amelyben minden elem két adatot – egy URL-t és egy nevet – tartalmaz. Lássuk, miként is működhet egy ilyen program:

```
Bookmarks (bookmarks.dbm)
(1) Programming in Python 3........ http://www.qtrac.eu/py3book.html
(2) PyQt........................... http://www.riverbankcomputing.com
(3) Python......................... http://www.python.org
(4) Qtrac Ltd...................... http://www.qtrac.eu
(5) Scientific Tools for Python.... http://www.scipy.org

(A)dd (E)dit (L)ist (R)emove (Q)uit [l]: e
Number of bookmark to edit: 2
URL [http://www.riverbankcomputing.com]:
Name [PyQt]: PyQt (Python bindings for GUI library)
```

Tegyük lehetővé a felhasználóknak, hogy új könyvjelzőt illesszenek be, továbbá adjunk lehetőséget a könyvjelzők szerkesztésére, felsorolására és törlésére. A szerkesztéshez, illetve az eltávolításhoz azonosítsuk a könyvjelzőket számokkal, így a felhasználó ezek valamelyikét választva végezheti el a kívánt műveletet. Tároljuk az adatokat egy DBM-fájlban a `shelve` modul segítségével, amelyben a kulcsok szerepét a nevek játsszák, az értékeknek pedig az URL-ek felelnek meg. A program a szerkezetét tekintve igencsak hasonlít majd a `dvds-dbm.py` fájlra, azzal a különbséggel, hogy a `find_dvd()` helyett ezúttal a `find_bookmark()` szerepel, ami jóval egyszerűbb a társánál, hiszen ezúttal mindössze egy egész számot kell beolvasnunk a felhasználótól, és ez alapján megkeresni a megfelelő könyvjelző nevét.

A felhasználók kényelme kedvéért a beillesztett vagy szerkesztett URL előtt helyezzük el a `http://` karakterláncot, amennyiben nem adnak meg semmilyen más protokollt.

A teljes program száznál kevesebb sort igényel (feltéve, hogy igénybe vesszük a `Console` modul szolgáltatásait, köztük a `Console.get_string()` függvényt). Egy lehetséges megoldást megtekinthetünk a `bookmarks.py` fájlban.

12

Szabályos kifejezések

- A Python szabályoskifejezés-nyelve
- A Python szabályoskifejezés-modulja

A szabályos kifejezések olyan tömör jelölések, amelyek karakterláncok egy csoportját írják le. A szabályos kifejezéseket az teszi olyan hatékonnyá, hogy egyetlen szabályos kifejezés korlátlan számú karakterláncot ábrázolhat – feltéve, hogy azok megfelelnek a szabályos kifejezés követelményeinek. A szabályos kifejezéseket (angolul regular expression – ezért általában csak „regex"-nek hívjuk őket) egy mininyelv segítségével határozzuk meg, ami teljesen eltér a Pythontól, a Python részét képező `re` modulon keresztül azonban zökkenőmentesen hozhatunk létre és használhatunk szabályos kifejezéseket.* Szabályos kifejezéseket általában a következő négy célra szoktunk használni:

- Érvényesség-ellenőrzés: annak vizsgálata, hogy egy adott szövegrész megfelel-e bizonyos feltételeknek, például tartalmazza-e egy pénznem jelét, amelyet számjegyek követnek.
- Keresés: olyan rész-karakterláncok keresése, amelyek részben különbözhetnek; például a „pet.png", a „pet.jpg", a „pet.jpeg" vagy a „pet.svg" keresése, de nem a „carpet.png" és hasonló karakterláncoké.
- Keresés és csere: minden olyan szövegrész lecserélése, amelyre a szabályos kifejezés illeszkedik, például hogy a „bicycle" és a „human powered vehicle" karakterláncokból egyaránt „bike" legyen.
- Karakterláncok felbontása: karakterlánc szétvágása minden olyan ponton, amely illeszkedik a szabályos kifejezésre, például minden „:" vagy „=" karakternél.

Legegyszerűbb formájában a szabályos kifejezés egy kifejezés (vagyis literális karakter), amelyet egy elhagyható mennyiségjelző követ. Az összetettebb szabályos kifejezések tetszőleges számú, mennyiségjelzővel ellátott kifejezésből állhatnak, és feltételezések (assertion, „állítás") is szerepelhetnek bennük, valamint jelzőkkel (flag) szabályozhatók.

---

* Az egyik jó könyv, amely a szabályos kifejezésekről szól, Jeffrey E. F. Friedl *Mastering Regular Expressions* című kötete (ISBN 0596528124). A mű nem foglalkozik kifejezetten a Pythonnal, de a Python `re` modulja nagyon hasonló szolgáltatásokat nyújt, mint a Perl nyelvnek a könyvben részletesen ismertetett szabályoskifejezés-motorja.

Fejezetünk első részében bemutatjuk és elmagyarázzuk a szabályos kifejezésekkel kapcsolatos összes fontos fogalmat. Tisztán a szabályos kifejezések nyelvtanával foglalkozunk – magára a Pythonra csak annyiban utalunk, amennyire feltétlenül szükséges. A fejezet második részében aztán a korábban tanultakra építve megmutatjuk, hogy miként használhatók a szabályos kifejezések a Python programozási környezetében. Azok, akik már ismerik a szabályos kifejezéseket, és csak azt szeretnék megtanulni, hogy miként működnek a Pythonban, rögtön a fejezet második felére ugorhatnak. A fejezet a `re` modul által kínált teljes szabályoskifejezés-nyelvet bemutatja, beleértve az összes feltételezést és jelzőt. A szövegben a szabályos kifejezéseket **`félkövérrel`** szedtük, az illeszkedéseket (találatokat) `aláhúzással`, az elfogott találatokat pedig `árnyékolással` jelöltük.

A Python szabályoskifejezés-nyelve

Ebben a részben négy témakörre bontva tárgyaljuk a szabályos kifejezések nyelvét. Az első szakaszban azt mutatjuk meg, hogy miként lehet egyes karakterekre (például a-ra vagy b-re) vagy karakterek egy csoportjára (például „*a vagy b*”-re) illeszteni. A második szakaszban az illesztések mennyiségjelzőivel foglalkozunk (például egyszeres illesztés, legalább egyszeres illesztés vagy a lehető legtöbbszöri illesztés). A harmadik szakasz az alkifejezések csoportosításának és az illeszkedő szövegek elfogásának módját mutatja be, a negyedik, utolsó szakasz pedig a nyelv feltételezéseinek és jelzőinek használatát, amelyekkel szabályozhatjuk a szabályos kifejezések működését.

Karakterek és karakterosztályok

A legegyszerűbb kifejezések csupán literális karakterek, mint az **`a`** vagy az **`5`**, és ha nem adunk meg kifejezetten mennyiségjelzőt, a nyelv egyszeri illesztésként értelmezi őket. A **`tune`** szabályos kifejezés például négy, beleértett módon egyszeri illesztést meghatározó kifejezésből áll: arra illeszkedik, ha egy *`t`*-t egy *`u`*, azt egy *`n`*, azt pedig egy *`e`* követi, így a `tune` és az `attuned` karakterláncokra is találatot ad.

Bár a legtöbb karakter használható literálként, vannak „különleges karakterek” is – olyan jelek, amelyek a szabályoskifejezés-nyelv sajátjai, ezért ha literális értelemben szeretnénk használni őket, le kell védenünk egy fordított perjellel (`\`). A különleges karakterek a következők: `\.^$?+*{}[]()|`. A Python szabványos védőkaraktereinek többsége – például az újsor karaktert jelző `\n` vagy a tabulátor karaktert jelző `\t` – ugyancsak használható a szabályos kifejezéseken belül, akárcsak az `\x`*`HH`*, az `\u`*`HHHH`* vagy az `\U`*`HHHHHHHH`* formát követő hexadecimális védőkarakterek.

Sok esetben nem egy bizonyos karaktert keresünk, hanem egy karakterhalmaz bármelyik tagját. Ezt egy *karakterosztály* használatával tehetjük meg, ami nem más, mint egy vagy több, szögletes zárójelek közé zárt karakter. (A karakterosztályoknak semmi közük a Python osztályaihoz, csupán így hívják a szabályos kifejezések nyelvében a karakterek halmazait.) A karakterosztály is kifejezés, és mint ilyen, rá is érvényes, hogy ha nem adunk

meg hozzá mennyiségjelzőt, akkor pontosan egy karakterre illeszkedik (ami a karakterosztályban szereplő karakterek bármelyike lehet). A `r[ea]d` szabályos kifejezés például mind a `red`, mind a `radar` szóra találatot ad, de a `read`-re nem. Ehhez hasonlóan, ha egyetlen számjegyre szeretnénk illeszteni, akkor a `[0123456789]` szabályos kifejezést használhatjuk. A kényelem kedvéért karakterek egy tartományát egy kötőjel segítségével is meghatározhatjuk, így a `[0-9]` szabályos kifejezés szintén egy számjegyre illeszkedik. Ha a nyitó zárójel után egy kalapjelet írunk, a karakterosztályt tagadhatjuk is, tehát a `[^0-9]` szabályos kifejezés minden olyan karakterre illeszkedni fog, ami *nem* számjegy.

Megjegyzendő, hogy a karakterosztályokon belül a \ kivételével a különleges karakterek elveszítik az eredeti (különleges) jelentésüket – bár a ^ esetében azt figyelhetjük meg, hogy a karakter új jelentést (tagadás) kap, ha a karakterosztály első karaktereként szerepel (minden más esetben egyszerűen ^ karakternek minősül). Emellett a – karaktertartományt jelöl, kivéve, ha ez az első karakter, mely esetben egyszerű kötőjelként értelmezendő.

Mivel egyes karakterhalmazokra gyakran szükség van, ezek gyorsírásos alakkal is rendelkeznek, amelyeket a 12.1. táblázatban láthatunk. Egy kivétellel valamennyi gyorsírásos alak a karakterhalmazokon belül használható, tehát a `[\dA-Fa-f]` szabályos kifejezés például bármely hexadecimális számjegyre ad találatot. A kivételt a . jelenti, amely a karakterosztályokon kívül minősül gyorsírásos alaknak, karakterosztályon belül azonban a literális pont (.) karakterre illeszkedik.

12.1. táblázat *Karakterosztályok gyorsírásos alakjai*

Jel	Jelentés
.	Az újsor kivételével bármely karakterre illeszkedik, illetve a `re.DOTALL` jelzővel megadva minden karakterre, karakterosztályon belül azonban a pont (.) karakterre.
\d	Az Unicode-számjegyekre illeszkedik, illetve a `re.ASCII` jelzővel megadva a `[0-9]` tartományra.
\D	A nem számjegy Unicode-karakterekre illeszkedik, illetve a `re.ASCII` jelzővel megadva a `[^0-9]` tartományra.
\s	Az Unicode-üreshelyekre illeszkedik, illetve a `re.ASCII` jelzővel megadva a `[ \t\n\r\f\v]` tartományra.
\S	A nem üreshely Unicode-karakterekre illeszkedik, illetve a `re.ASCII` jelzővel megadva a `[^ \t\n\r\f\v]` tartományra.
\w	Az Unicode-„szókarakterekre” illeszkedik, illetve a `re.ASCII` jelzővel megadva az `[a-zA-Z0-9]` tartományra.
\W	A nem „szókarakter” Unicode-karakterekre illeszkedik, illetve a `re.ASCII` jelzővel megadva a `[^a-zA-Z0-9]` tartományra.

Mennyiségjelzők

A mennyiségjelzők alakja `{m,n}`, ahol az *m* és az *n* azt adják meg, hogy legalább, illetve legfeljebb hány alkalommal kell illeszkednie annak a kifejezésnek, amelyikre a mennyiségjelző vonatkozik. Az **`e{1,1}e{1,1}`** és az **`e{2,2}`** szabályos kifejezések például egyaránt illeszkednek a `feel`-re, de a `felt`-re egyik sem.

Ha minden kifejezés után mennyiségjelzőt kellene írnunk, az hamar fárasztóvá válna, és a kód olvashatóságát is rontaná. Szerencsére a szabályos kifejezések nyelve többféle kényelmes gyorsírásos alakot is támogat. Ha a mennyiségjelzőben csak egy számot adunk meg, akkor úgy veszi, hogy ez a minimum és a maximum is egyben, tehát az **`e{2}`** ugyanazt jelenti, mint az **`e{2,2}`**. Ahogy pedig az előző részben már megjegyeztük, ha nem adunk meg kifejezetten mennyiségjelzőt, a nyelv egyszeri illesztést feltételez (tehát **`{1,1}`**-et vagy **`{1}`**-et) – az **`ee`** tehát ugyanazt jelenti, mint az **`e{1,1}e{1,1}`**, illetve az **`e{1} e{1}`**, ezért az **`e{2}`** és az **`ee`** egyaránt illeszkedik a `feel`-re, de a `felt`-re nem.

Gyakran hasznos más-más minimum- és maximumértéket megadni. Ahhoz például, hogy a `travelled` és a `traveled` szóra egyaránt illeszthessünk (mindkettő egyformán elfogadott írásmód), a **`travel{1,2}ed`** és a **`travell{0,1}ed`** alakot is használhatjuk. A **`{0,1}`** mennyiségjelzőt olyan gyakran használják, hogy saját gyorsírásos alakja van: a kérdőjel (`?`). Az előző szabályos kifejezést tehát így is írhatjuk (és a gyakorlatban általában így is szoktak tenni): **`travell?ed`**.

Még két gyorsírásos mennyiségjelző áll rendelkezésre: a pluszjel (`+`), amely az **`{1,`***n***`}`** („legalább egy") alakot helyettesíti, illetve a csillag (`*`), amelynek a jelentése **`{0,`***n***`}`** („tetszőleges számú"). Az *n* mindkét esetben az adott mennyiségjelző esetében lehetséges előfordulások maximális száma, ami általában legalább 32 767. A mennyiségjelzőket a 12.2. táblázatban láthatjuk.

A `+` mennyiségjelző nagyon hasznos. Az egész számok illesztéséhez például használhatjuk a **`\d+`** alakot is, hiszen ez egy vagy több számjegyre illeszkedik. A `4588.91` karakterláncban például ez a szabályos kifejezés két helyen is illeszkedik: `4588.91` és `4588.91`.

Az elírásoknak néha az az oka, hogy egy billentyűt túl sokáig tartunk lenyomva. A **`bevel+ed`** szabályos kifejezéssel kiszűrhetjük a helyes `beveled` és `bevelled` alakokat, míg a helytelen bevellled nem fog illeszkedni. Ha egységesen az egy *l*-es írásmódot szeretnénk használni, és csak azokat az előfordulásokat keressük, amelyekben kettő vagy több l található, a **`bevell+ed`** szabályos kifejezés lehet a segítségünkre.

A `*` mennyiségjelzőnek kevésbé vesszük hasznát, egyszerűen azért, mert gyakran nem várt eredményt ad. Tegyük fel például, hogy azokat a sorokat keressük néhány Python-fájlban, amelyek megjegyzéseket tartalmaznak. Erre a célra megpróbálkozhatunk a **`#*`** kifejezéssel, de ez a szabályos kifejezés minden sorra illeszkedni fog, beleértve az üres sorokat is, mert a jelentése „keress tetszőleges számú #-et", és a „tetszőleges szám"-ba

a nulla is beleértendő. Azoknak, akik csak most ismerkednek a szabályos kifejezésekkel, alapszabályként a `*` teljes kerülését javasoljuk, ha viszont mégis használnánk (akár a csillagot, akár a `?` regexet), akkor gondoskodjunk róla, hogy legalább még egy olyan kifejezés szerepeljen a szabályos kifejezésben, amelynek a mennyiségjelzője nem nulla, vagyis egy olyan mennyiségjelző, amely se nem `*`, se nem `?`, mivel ez a kettő a megadott kifejezés nulla előfordulását is keresi.

Ahol csillagot használnánk, ott gyakran használhatjuk a `+` mennyiségjelzőt is, és viszont. A legalább egy *l*-t tartalmazó „tasselled" szót például a `tassell*ed` és a `tassel+ed` kifejezéssel is megtaláljuk, míg a kettő vagy több *l*-t tartalmazó előfordulások illesztésére a `tasselll*ed` és a `tassell+ed` is használható.

A `\d+` szabályos kifejezés a `136` karakterláncra is illeszkedik. De miért az összes számjegyre, miért nem csak az elsőre? Nos, a mennyiségjelzők alapértelmezés szerint kivétel nélkül *mohók*, vagyis annyi karakterre illeszkednek, amennyire csak tudnak. A mennyiségjelzők azonban nem mohóvá (úgy is mondják, hogy „*minimálissá*") is tehetők, ha egy kérdőjelet írunk utánuk. (A kérdőjelnek két különböző jelentése van: önmagában a `{0,1}` mennyiségjelző gyorsírásos alakja, míg ha egy mennyiségjelzőt követ, akkor arra utasítja azt, hogy ne legyen mohó.) A `\d+?` szabályos kifejezés például három helyen illeszkedik a `136` karakterláncra: `136`, `136` és `136`. Íme még egy példa: a `\d??` nulla vagy egy számjegyre illeszkedik, de a nulla előfordulást részesíti előnyben, mert nem mohó – önmagában állva ugyanaz a gond vele, mint a csillaggal: a semmire is találatot jelez, vagyis „minden szövegre" illeszkedik.

12.2. táblázat *A szabályos kifejezések mennyiségjelzői*

Alak	Jelentés
`e?` vagy `e{0,1}`	Mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e??` vagy `e{0,1}?`	Nem mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e+` vagy `e{1,}`	Mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e+?` vagy `e{1,}?`	Nem mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e*` vagy `e{0,}`	Mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e*?` vagy `e{0,}?`	Nem mohón illeszkedik az *e* kifejezés nulla vagy egy előfordulására.
`e{m}`	Az *e* kifejezés pontosan *m* előfordulására illeszkedik.
`e{m,}`	Mohón illeszkedik az *e* kifejezés legalább *m* előfordulására.
`e{m,}?`	Nem mohón illeszkedik az *e* kifejezés legalább *m* előfordulására.
`e{,n}`	Mohón illeszkedik az *e* kifejezés legfeljebb *n* előfordulására.
`e{,n}?`	Nem mohón illeszkedik az *e* kifejezés legfeljebb *n* előfordulására.
`e{m,n}`	Mohón illeszkedik az *e* kifejezés legalább *m* és legfeljebb *n* előfordulására.
`e{m,n}?`	Nem mohón illeszkedik az *e* kifejezés legalább *m* és legfeljebb *n* előfordulására.

A nem mohó mennyiségjelzők akkor hasznosak, ha „gyors és piszkos" XML- vagy HTML-feldolgozásra van szükségünk. Például ha az összes képcímkét keressük, az **`<img.*>`** („keress egy „<", majd egy „i", egy „m", egy „g" és nulla vagy több tetszőleges karaktert az újsor karakter kivételével, végül pedig egy „>" karaktert") nem fog működni, mert a **`.*`** rész mohó, így mindenre illeszkedni fog, beleértve a címke záró > karakterét, és addig ad találatokat, amíg el nem érte a teljes szöveg utolsó > karakterét.

Három megoldás kínálkozik (egy megfelelő feldolgozó használatán kívül). Az első az **`<img[^>]*>`** (ez akkor illeszkedik, ha az `<img` karaktereket tetszőleges számú olyan karakter követi, ami nem a >, majd a címke záró > karaktere), a második az **`<img.*?>`** (ez akkor illeszkedik, ha az `<img` karaktereket tetszőleges karakter követi – nem mohón értelmezve, ezért a címke záró > karaktere előtt megáll –, majd a > karakter következik), a harmadik pedig az **`<img[^>]*?>`**, amely az előző kettőt ötvözi. Ugyanakkor egyik megoldás sem helyes, mert mind találatot ad az `<img>` karakterláncra is, ami nem érvényes címke. Mivel tudjuk, hogy minden képelemnek rendelkeznie kell az `src` jellemzővel, a következő szabályos kifejezés pontosabb lenne: **`<img\s+[^>]*?src=\w+[^>]*?>`**. Ez a kifejezés literálisan illeszkedik az `<img` karakterekre, majd egy vagy több üreshelyre, aztán nem mohón nulla vagy több tetszőleges karakterre a > kivételével (így átugrik minden más jellemzőt, például az `alt`-ot), ezt követően az `src` jellemzőre (vagyis a literálisan értelmezett `src=` karakterláncra, amelyet legalább egy „szókarakter" követ), aztán az egyéb jellemzők megtalálása érdekében bármely olyan karakterre, ami nem a > (a „semmi" is számít), végül pedig a záró > karakterre.

Csoportosítás és elfogás

A valós alkalmazásokban gyakran van szükség olyan szabályos kifejezésekre, amelyek két vagy több alternatíva közül bármelyikre illeszkednek, és a találatot vagy annak egy részét is gyakran kell elfognunk további feldolgozás céljából. Az is előfordul, hogy azt szeretnénk, ha egy mennyiségjelző egyszerre több kifejezésre vonatkozna. Ezek mindegyikét elérhetjük csoportosítással, amire a `()` jelek szolgálnak, az alternatívák meghatározására pedig a `|` („vagy") jelet használhatjuk.

A „vagy" művelet különösen akkor hasznos, ha azt szeretnénk, hogy a szabályos kifejezés több teljesen különböző lehetőség közül bármelyikre illeszkedjen.
Az **`aircraft|airplane|jet`** szabályos kifejezés például bármilyen szöveget visszaad, amely tartalmazza akár az „aircraft", akár az „airplane", akár a „jet" karaktereket. Ugyanez elérhető az **`air(craft|plane)|jet`** szabályos kifejezéssel is. Itt a kerek zárójelek a kifejezések csoportosítására szolgálnak, tehát két „külső" kifejezésünk van – az **`air(craft|plane)`** és a **`jet`** –, az elsőben pedig egy belső kifejezés (**`craft|plane`**) található. Mivel ezt a belső kifejezést az **`air`** előzi meg, az első külső kifejezés csak az „aircraft", illetve az „airplane" szavakra illeszkedik.

A (kerek) zárójelek két különböző célt szolgálnak: kifejezések csoportosítására, illetve egy kifejezésre illeszkedő szöveg elfogására használhatók. A *csoport* szót az elfogott és el nem fogott csoportosított kifejezésekre egyaránt használjuk, a *fogás* vagy *elfogási csoport* szót viszont csak az elfogni szándékozott csoportokra. Ha az `(aircraft|airplane|jet)` szabályos kifejezést használnánk, az nem csak illeszkedne a benne szereplő három szó bármelyikére, hanem el is fogná a megtalált kifejezést későbbi használatra. Hasonlítsuk ezt össze az `(air(craft|plane)|jet)` szabályos kifejezéssel, amely két fogást tartalmaz, ha az első kifejezés illeszkedik (az első fogás az „aircraft" vagy „airplane", a második pedig a „craft" vagy „plane"), és egy fogást („jet"), ha a második kifejezés ad találatot. Az elfogást kikapcsolhatjuk, ha a nyitó zárójel után a `?:` karaktereket írjuk, tehát az `(air(?:craft|plane)|jet)` szabályos kifejezésnek például csak egy fogása lesz, ha illeszkedik („aircraft" vagy „airplane" vagy „jet").

A csoportosított kifejezés is kifejezés, tehát ellátható mennyiségjelzővel. Mint minden más kifejezés esetében, a nyelv egyszeri illesztést feltételez, hacsak kifejezetten meg nem adjuk, hogy hányszor szeretnénk illeszteni. Például ha egy olyan szövegfájlt olvasunk be, amelyben *kulcs=érték* alakú sorok találhatók, ahol minden *kulcs* alfanumerikus, a `(\w+)=(.+)` szabályos kifejezés minden olyan sorra illeszkedni fog, amelyben sem a kulcs, sem az érték mező nem üres. (Emlékezzünk vissza, hogy a . mindenre illeszkedik, kivéve az újsor karaktereket.) Ezen felül minden illeszkedő sor esetében két fogásunk lesz: az első a kulcs, a második pedig az érték.

A *kulcs=érték* fenti szabályos kifejezése például a teljes `topic= physical geography` sorra illeszkedni fog (a két fogást az árnyékolás mutatja). Figyeljük meg, hogy a második fogásban egy szóköz (üreshely karakter) is szerepel, az egyenlőségjel előtt azonban nem lehet üreshely. A szabályos kifejezést a következőképpen írhatnánk át egy kicsit hosszabb formára, hogy rugalmasabb legyen az üreshelyek elfogadása tekintetében, és eltávolítsa a nemkívánatos szóközöket:

```
[ \t]*(\w+)[ \t]*=[ \t]*(.+)
```

Ez a kifejezés is illeszkedik ugyanarra a sorra, mint a korábbi, de azokra is, amelyekben az egyenlőségjel körül szóközök találhatók, csak éppen az első fogásba nem kerül bele a bevezető vagy záró szóköz, a második fogásba pedig a bevezető üreshely. Például: `topic= physical geography`.

Szabályoskifejezés-jelzők ☞ 479

Az üreshelyekre illeszkedő részeket gondosan az elfogó zárójeleken kívül helyeztük el, és arra is ügyeltünk, hogy azokat a sorokat is megtaláljuk, amelyekben egyáltalán nincs szóköz. Az üreshelyekre nem az `\s` kifejezéssel illesztettünk, mert ez az újsor karakterekre (`\n`) is illeszkedik, ami helytelenül több sort átfogó találatokat eredményezett volna (például ha a `re.MULTILINE` jelzőt használjuk). Az érték esetében pedig azért nem az `\s` kifejezést alkalmaztuk a nem üreshely karakterek illesztésére, mert azokat az értékeket (például

a mondatokat) is meg kívántuk engedni, amelyek szóközt is tartalmaznak. Ahhoz, hogy elkerüljük, hogy a második fogásban záró szóköz is lehessen, kifinomultabb szabályos kifejezésre van szükség – erre a következő részben láthatunk majd példát.

Az elfogott kifejezésekre *visszirányú hivatkozásokkal* hivatkozhatunk, vagyis úgy, hogy visszautalunk egy korábbi elfogási csoportra.* Magukon a szabályos kifejezéseken belül az egyik lehetséges hivatkozási forma az `\i`, ahol az `i` a fogás száma. A fogások egytől számozódnak, és a szám balról jobbra egyesével nő, ahogy az értelmező egy új (elfogó) bal zárójellel találkozik. A kétszer szereplő szavakat például leegyszerűsítve úgy találhatjuk meg, ha a `(\w+)\s+\1` szabályos kifejezést használjuk, amely egy „szóra" illeszkedik, amelyet legalább egy üreshely követ, majd ismét az elfogott szó. (A 0 elfogási szám automatikusan jön létre, anélkül, hogy szükség lenne a zárójelekre – a 0. fogás a teljes találatot tartalmazza, vagyis azt, amit aláhúzással jelölünk.) Az ismétlődő szavak illesztésére később egy kifinomultabb módszert is látni fogunk.

A hosszú vagy összetett szabályos kifejezésekben gyakran kényelmesebb nevekkel, mint számokkal azonosítani a fogásokat. Ez a kód karbantartását is megkönnyíti, mivel az elfogó zárójelek hozzáadása vagy eltávolítása a számozást megváltoztatja, a nevekre azonban nincs hatással. Ha nevet szeretnénk adni egy fogásnak, a nyitó zárójel után a `?P<`*név*`>` kifejezést kell írnunk. A `(?P<key>\w+)=(?P<value>.+)` szabályos kifejezésben például két fogás szerepel: a `"key"` és a `"value"`. A nevesített fogásokra mutató visszirányú hivatkozások formája a szabályos kifejezéseken belül `(?P=`*név*`)`. A `(?P<word>\w+)\s+(?P=word)` például a `"word"` nevű fogás segítségével az ismétlődő szavakra illeszkedik.

Feltételezések és jelzők

Az eddig látott szabályos kifejezések nagy részét érintő probléma, hogy több, sőt egészen eltérő szövegekre is illeszkedhetnek, mint amit szerettünk volna.

Az `aircraft|airplane|jet` szabályos kifejezés például a „jet" mellett a „waterjet" és a „jetski" szavakat is visszaadja. Ezt a fajta problémát feltételezések (assertion, „állítás") segítségével oldhatjuk meg. A feltételezés nem illeszt semmilyen szöveget, csupán állít valamit a szövegről azon a ponton, ahol a feltételezés szerepel.

Ilyen feltételezés például a `\b` (szóhatár), amely azt mondja ki, hogy az előtte álló karakternek „szókarakternek" (`\w`) kell lennie, az utána állónak pedig nem szókarakternek (`\W`), vagy fordítva. A `jet` szabályos kifejezés például a „the jet and jetski are noisy" szövegben kétszer illeszkedik – `the jet and jetski are noisy` –, a `\bjet\b` viszont csak egyszer: `the jet and jetski are noisy`. Az eredeti szabályos kifejezés összefüggésében tehát a `\baircraft\b|\bairplane\b|\bjet\b` alakot, vagy hogy átláthatóbb legyen, a `\b(?:aircraft|airplane|jet)\b` szabályos kifejezést írhatjuk (vagyis: szóhatár, nem elfogó kifejezés, szóhatár).

---

* Jegyezzük meg, hogy a karakterosztályokon belül (vagyis egy `[]` belsejében) nem használhatunk visszirányú hivatkozásokat.

Ahogy a 12.3. táblázat is mutatja, a nyelv számos egyéb feltételezést támogat. Ezeket felhasználhatjuk például egy *kulcs=érték* szabályos kifejezés világosabbá tételére, ha a regexet a `^(\w+)=([^\n]+)` alakra módosítjuk, és beállítjuk a `re.MULTILINE` jelzőt, hogy biztosítsuk, hogy minden *kulcs=érték* pár egyetlen sorból származzon, és ne legyen lehetőség több sor átfogására. (A jelzőket a 12.5. táblázatban találhatjuk, használati alakjukat pedig ennek a résznek a végén, illetve a következő részben ismertetjük.) Ha a bevezető és záró üreshelyeket is el szeretnénk távolítani, valamint nevesített fogásokat szeretnénk használni, a szabályos kifejezés teljes alakja ez lesz:

```
^[ \t]*(?P<key>\w+)[ \t]*=[ \t]*(?P<value>[^\n]+)(?<![ \t])
```

Szabályoskifejezés-jelzők
☞ 479

Bár ez a szabályos kifejezés meglehetősen egyszerű feladatot lát el, elég bonyolultnak tűnik. A későbbi módosítás megkönnyítésének egyik módja, ha megjegyzésekkel látjuk el. Ezt megtehetjük soron belüli megjegyzések hozzáadásával is, `(?#`*megjegyzés*`)` alakban, de a gyakorlatban az ilyen megjegyzések a szabályos kifejezést még nehezebben olvashatóvá tehetik. Ennél sokkal elegánsabb megoldás a `re.VERBOSE` jelző használata, amely lehetővé teszi, hogy a szabályos kifejezésekben szabadon használhassunk üreshelyeket és szokványos Python-megjegyzéseket, azzal az egy megkötéssel, hogy ha üreshely karakterre szeretnénk illeszteni, akkor vagy az `\s` kifejezést, vagy egy olyan karakterosztályt kell alkalmaznunk, mint a `[ ]`. Íme a *kulcs=érték* szabályos kifejezés megjegyzésekkel ellátott formában:

```
^[ \t]*                # a sor kezdete és az esetleges bevezető üreshely
(?P<key>\w+)           # a kulcs szövege
[ \t]*=[ \t]*          # az egyenlőségjel a körülötte levő esetleges
                       # üreshelyekkel
(?P<value>[^\n]+)      # az érték szövege
(?<![ \t])             # negatív visszatekintés a záró üreshelyek elkerülésére
```

Nyers karakterláncok
☞ 61

Egy Python-programban egy ehhez hasonló szabályos kifejezést általában egy nyers, háromszoros idézőjelek közé zárt karakterláncban helyeznénk el – azért nyers karakterláncban, hogy ne kelljen megkettőzni a fordított perjeleket, és azért háromszoros idézőjelek közé zártban, hogy több sorra tördelhessük.

Az eddig bemutatott feltételezéseken kívül továbbiak is léteznek, amelyek az állítás előtt (vagy után) álló szöveget vizsgálják, hogy megfelel-e (vagy éppen ellenkezőleg) egy általunk megadott kifejezésnek. A visszatekintő állításokban (utófeltételezésekben) szereplő kifejezéseknek rögzített hosszúságúaknak kell lenniük (tehát a `?`, `+` és `*` mennyiségjelzők nem használhatók, a számszerű mennyiségjelzőknek pedig adott értékűnek kell lenniük, például: `{3}`).

12.3. táblázat *A szabályos kifejezések feltételezései*

Jel	Jelentés
^	A szöveg kezdetére illeszkedik, illetve a `re.MULTILINE` jelző használata esetén minden újsor karakter után.
$	A szöveg végére illeszkedik, illetve a `re.MULTILINE` jelző használata esetén minden újsor karakter előtt.
\A	A szöveg elejére illeszkedik.
\b	A szóhatárokra illeszkedik, a `re.ASCII` jelzőnek megfelelően. Egy karakterosztályon belül a „backspace" karakter levédésére szolgál.
\B	Nem szóhatárra illeszkedik, a `re.ASCII` jelzőnek megfelelően.
\Z	A szöveg végére illeszkedik.
(?=*e*)	Akkor illeszkedik, ha az *e* kifejezés illeszkedik a feltételezésnél, de nem halad át rajta. *Előfeltételezésnek* vagy *pozitív előretekintésnek* is hívják.
(?!*e*)	Akkor illeszkedik, ha az *e* kifejezés nem illeszkedik a feltételezésnél, és nem halad át rajta. *Negatív előretekintésnek* is hívják.
(?<=*e*)	Akkor illeszkedik, ha az *e* kifejezés közvetlenül a feltételezés előtt illeszkedik. *Pozitív visszatekintésnek* is hívják.
(?<!*e*)	Akkor illeszkedik, ha az *e* kifejezés nem illeszkedik közvetlenül a feltételezés előtt. *Negatív visszatekintésnek* is hívják.

Szabályoskifejezés-jelzők ☞ 479

A *kulcs=érték* szabályos kifejezés esetében a negatív visszatekintés azt jelenti, hogy ahol a feltételezés szerepel, ott az *előző* karakter nem lehet szóköz vagy tabulátor. Ez biztosítja, hogy a `"value"` elfogási csoportba olvasott utolsó karakter ne lehessen szóköz vagy tabulátor (ugyanakkor azt nem akadályozza meg, hogy az elfogott szövegen belül szóköz vagy tabulátor álljon).

Nézzünk meg még egy példát! Tegyük fel, hogy egy többsoros szöveget olvasunk, amely a „Helen Patricia Sharman", „Jim Sharman", „Sharman Joshi", „Helen Kelly" stb. neveket tartalmazza, és a „Helen Patricia" nevet keressük, de csak akkor, ha „Helen Patricia Sharman"-ról van szó. Ennek a **`\b(Helen\s+Patricia)\s+Sharman\b`** szabályos kifejezés használata a legegyszerűbb módja, de ugyanezt az eredményt egy előretekintő állítással is elérhetjük, például így: **`\b(Helen\s+Patricia)(?=\s+Sharman\b)`**. Ez a kifejezés csak akkor illeszkedik a „Helen Patricia" névre, ha azt egy szóhatár előzi meg, utána pedig egy szóköz és a „Sharman" szó áll, amelynek a végén szóhatár található.

A szövegben szereplő keresztnév-változatok („Helen", „Helen P." vagy „Helen Patricia") elfogásához a szabályos kifejezésen kissé bonyolítanunk kell, például így: **`\b(Helen(?:\s+(?:P\.|Patricia))?)\s+(?=Sharman\b)`**. Ez a kifejezés egy szóhatárra illeszkedik, amelyet valamelyik keresztnév-változat követ, de csak akkor, ha a keresztnév után egy szóköz és a „Sharman" szó áll, amelynek a végén szóhatár található.

Jegyezzük meg, hogy az elfogásnak csak két formája lehetséges: az `(e)` és a `(?P<név>e)`. Az egyéb zárójeles alakok egyike sem hajt végre elfogást. Ez tökéletesen érthető az előmi és utófeltételezések esetében, hiszen ezek csak egy állítást fogalmaznak meg arról, hogy mi előzi meg, illetve követi őket – nem részei az illesztésnek, csupán azt befolyásolják, hogy kapunk-e egyezést. Ezen kívül a következőkben bemutatott két utolsó zárójeles alaknál is ésszerű.

Korábban láttuk, hogy miként hivatkozhatunk visszafelé egy fogásra egy szabályos kifejezés belsejében, akár számmal (például `\1`), akár név szerint (például `(?P=név)`). A feltételes illesztés azonban attól függően is lehetséges, hogy korábban kaptunk-e már egyezést. Ennek utasításformája a következő: `(?(azonosító)igen_kifejezés)` vagy `(?(azonosító)igen_kifejezés|nem_kifejezés)`. Az *azonosító* annak a korábbi fogásnak a sorszáma vagy neve, amelyikre hivatkozunk. Ha az elfogás sikerült, az *igen_kifejezés*-re illesztünk, ha viszont meghiúsult, a *nem_kifejezés*-re (amennyiben megadtunk ilyet).

Vegyünk egy példát: tegyük fel, hogy az `img` HTML-elemek `src` jellemzőjéből szeretnénk kinyerni az ott hivatkozott fájlneveket. Először csak az `src` jellemzőre próbálunk illeszteni, de a korábbi kísérletünktől eltérően figyelembe vesszük a jellemző értéke által felvehető három alakot (aposztrófos, idézőjeles, idézőjel nélküli). Íme az első változat: `src=(["'])([^"'>]+)\1`. A `([^"'>]+)` rész legalább egy nem idézőjel vagy > karakter mohó illesztését fogja el. Ez a szabályos kifejezés az idézőjeles fájlnevek esetében megfelelően működik, és az `\1`-nek köszönhetően csak akkor ad egyezést, ha a nyitó és záró idézőjelek egyformák. Az idézőjel nélküli fájlneveket azonban nem engedi meg. Ennek kijavításához a nyitó idézőjelet elhagyhatóvá kell tennünk, és csak akkor szabad illesztenünk rá, ha jelen van. Íme a javított szabályos kifejezés: `src=(["'])?([^"'>]+)(?(1)\1)`. A *nem_kifejezés*-t nem adtuk meg, mivel nincs mire illeszteni, ha nincs idézőjel. Most már készen állunk arra, hogy a szabályos kifejezést összefüggésében helyezzük el – íme az `img` elemekre írt teljes szabályos kifejezés, nevesített csoportok és megjegyzések használatával:

```
<img\s+                          # a címke kezdete
[^>]*?                           # az src-t megelőző esetleges jellemzők
src=                             # az src jellemző kezdete
(?P<quote>["'])?                 # elhagyható nyitó idézőjel
(?P<image>[^"'>]+)               # a kép fájlneve
(?(quote)(?P=quote))             # záró idézőjel (ha van nyitó idézőjel,
                                 # azzal egyező)
[^>]*?                           # az src-t követő esetleges jellemzők
>                                # a címke vége
```

Az elfogott fájlnév elfogási neve `"image"` (a fogás sorszáma pedig 2).

Természetesen van egyszerűbb és kifinomultabb megoldás is: `src=(["']?)([^"'>]+)\1`. Itt, ha van nyitó idézőjel karakter, akkor elfogjuk az 1-es elfogási csoportban, és a nem idézőjel karakterek után illesztjük. Ha nincs nyitó idézőjel, az 1-es csoport akkor is illeszkedni

fog – mégpedig egy üres karakterláncra, mivel nem kötelező (a mennyiségjelzője nulla vagy egy), mely esetben a visszirányú hivatkozás ugyancsak egy üres karakterláncra fog illeszkedni.

Szabályoskifejezés-jelzők
☞ 479

A Python szabályoskifejezés-motorja által kínált szabályos kifejezések nyelvtanában az utolsó rész, aminek még a helyére kell kerülnie, a jelzők (flag) beállításának módja. A jelzőket általában kiegészítő paraméterekként átadva állítjuk be, amikor meghívjuk a `re.compile()` függvényt, de néha kényelmesebb magának a szabályos kifejezésnek a részeként beállítani őket. Az utasításforma egyszerűen `(?`*`jelzők`*`)`, ahol a *jelzők* egy vagy több jelzőt jelent az `a` (ez ugyanazt jelenti, mint a `re.ASCII` átadása) az `i` (`re.IGNORECASE`), az `m` (`re.MULTILINE`), az `s` (`re.DOTALL`) és az `x` (`re.VERBOSE`) közül.* Ha a jelzőket így állítjuk be, a szabályos kifejezés elejére kell tennünk őket. Semmire sem illeszkednek, ezért a hatásuk csupán annyi a szabályos kifejezésre, hogy beállítják a jelzőket.

A Python szabályoskifejezés-modulja

A `re` modul kétféle módját teszi lehetővé a szabályos kifejezésekkel való munkának. Az egyik a 12.4. táblázatban felsorolt függvények használata, amikor is az adott függvénynek első paraméterként egy szabályos kifejezést adunk át. A függvény a szabályos kifejezést egy belső formátumra alakítja át – ezt hívják *fordításnak* (compiling) –, és ezt követően hajtja végre a feladatát. Ez egyszeri használat esetén nagyon kényelmes, de ha ugyanazt a szabályos kifejezést többször is használni szeretnénk, a minden használatkori fordítás költségét célszerű elkerülnünk: ezt teszi lehetővé a `re.compile()` függvény, amellyel csak egyszer kell lefordítanunk a szabályos kifejezést. Ezt követően annyiszor hívhatunk meg tagfüggvényeket a lefordított szabályos kifejezésre, ahányszor csak szeretnénk. A lefordított szabályos kifejezések tagfüggvényeit a 12.6. táblázatban láthatjuk.

```
match = re.search.(r"#[\dA-Fa-f]{6}\b", text)
```

A fenti kódrészlet egy `re`-modulfüggvény használatát mutatja be. A szabályos kifejezés a HTML-stílusban (például `#C0C0AB` formában) megadott színeket keresi. Ha egyezést talál, a `re.search()` függvény egy találati objektumot ad vissza, egyébként pedig `None`-t. A találati objektumok által biztosított tagfüggvényeket a 12.7. táblázatban tekinthetjük meg.

Ha ezt a szabályos kifejezést többször szeretnénk használni, akkor egyszer lefordítjuk, majd szükség esetén a lefordított szabályos kifejezést használjuk:

```
color_re = re.compile(r"#[\dA-Fa-f]{6}\b")
match = color_re.search(text)
```

* A jelzőkhöz használt betűk ugyanazok, mint amelyeket a Perl szabályoskifejezés-motorja használ, ezért az s a re.DOTALL, illetve az x a re.VERBOSE jele.

Ahogy korábban megjegyeztük, nyers karakterláncokat használunk, hogy elkerüljük a fordított perjelek levédésének szükségességét. A fenti szabályos kifejezést másképp is felírhatnánk a `[\dA-F]` karakterosztály használatával és a `re.IGNORECASE` jelzőt a `re.compile()` hívás utolsó paramétereként átadva, vagy használhatnánk a `(?i)#[\dA-F]{6}\b` szabályos kifejezést, amely a kis- és nagybetűk figyelmen kívül hagyására utasító jelzővel kezdődik.

Ha egynél több jelzőre van szükség, az `OR` („vagy") művelettel (`|`) kapcsolhatjuk össze őket, például így: `re.MULTILINE|re.DOTALL` (vagy `(?ms)`, ha magába a szabályos kifejezésbe ágyazzuk be).

Ezt a részt néhány példa áttekintésével zárjuk. Néhány olyan szabályos kifejezéssel kezdjük, amelyeket a fejezet korábbi részeiben láttunk, hogy szemléltessük a `re` modul által kínált lehetőségek közül a leggyakrabban használtakat. Elsőként lássuk a kétszer szereplő szavakat kiszűrő szabályos kifejezést:

```
double_word_re = re.compile(r"\b(?P<word>\w+)\s+(?P=word)(?!\w)",
                            re.IGNORECASE)
for match in double_word_re.finditer(text):
    print("{0} is duplicated".format(match.group("word")))
```

Ez a szabályos kifejezés némileg bonyolultabb, mint a korábban elkészített változata. A kifejezés egy szóhatárral kezdődik (hogy biztosítsuk, hogy minden illeszkedés egy szó elején kezdődjön), majd mohón illeszkedik egy vagy több „szókarakterre", egy vagy több üreshely karakterre, majd ismét ugyanarra a szóra – de csak akkor, ha a szó második előfordulását nem szókarakter követi.

Ha a bemeneten érkező szöveg „win in vain" volt, az első feltételezés *nélkül* egyetlen egyezést (találatot) és két fogást kapnánk: `win in vain`. A szóhatár-feltételezés használata biztosítja, hogy az első egyezésként kapott karakterlánc egy teljes szó legyen, így nem kapunk egyezést vagy fogást, hiszen nem egy szó ismétlődik. Ehhez hasonlóan, ha a bemeneten érkező szöveg „`one and and two let's say`" volt, az utolsó feltételezés *nélkül* két egyezésünk és két fogásunk lenne: `one and and two let's say`. Az előretekintő feltételezés használata azt jelenti, hogy a második egyezésként kapott karakterláncnak teljes szónak kell lennie, ezért egyetlen egyezést és egyetlen fogást kapunk: `one and and two let's say`.

A `for` ciklus végigmegy a `finditer()` tagfüggvény által visszaadott összes találati objektumon, a találati objektum `group()` tagfüggvényével pedig kinyerjük az elfogási csoport szövegét. Ugyanilyen könnyen használhattuk volna a `group(1)` kifejezést is (bár ez később nehezebben módosítható), mely esetben egyáltalán nem kellett volna nevet adnunk az elfogási csoportnak, és egyszerűen használhattuk volna a `(\w+)\s+\1(?!\w)` szabályos kifejezést. Azt is érdemes megjegyezni, hogy a kifejezés végén a `(?!\w)` helyett egy szóhatárt (`\b`) is alkalmazhattunk volna.

Egy másik korábbi példa egy olyan szabályos kifejezés volt, amely a fájlneveket kereste a HTML-képelemekben. Lássuk, hogyan fordíthatnánk le ezt a szabályos kifejezést, jelzőket hozzáadva, hogy a kis- és nagybetűket ne különböztesse meg, és lehetővé tegye megjegyzések beágyazását is:

```
image_re = re.compile(r"""
                <img\s+                     # a címke kezdete
                [^>]*?                      # nem-src jellemzők
                src=                        # az src jellemző kezdete
                (?P<quote>["'])?            # elhagyható nyitó idézőjel
                (?P<image>[^"'>]+)          # a kép fájlneve
                (?(quote)(?P=quote))        # záró idézőjel
                [^>]*?                      # nem-src jellemzők
                >                           # a címke vége
                """, re.IGNORECASE|re.VERBOSE)
image_files = []
for match in image_re.finditer(text):
    image_files.append(match.group("image"))
```

Itt ismét a `finditer()` tagfüggvényt használtuk a találatok kinyerésére, és a találati objektum `group()` tagfüggvényét az elfogott szövegek kiolvasására. Mivel a kis- és nagybetűkre való érzéketlenség csak az **`img`** és **`src`** elemekre vonatkozik, a `re.IGNORECASE` jelzőt el is hagyhatnánk, és használhatnánk helyette az **`[Ii][Mm][Gg]`**, illetve az **`[Ss][Rr][Cc]`** kifejezéseket. Ez a szabályos kifejezést némileg kevésbé átláthatóvá tenné, ugyanakkor gyorsítana rajta, mivel az illesztett szöveget nem kellene nagybetűssé (vagy kisbetűssé) alakítani – de ez valószínűleg csak akkor számít, ha a szabályos kifejezést igen nagy mennyiségű szövegre alkalmaznánk.

Gyakori feladat, hogy egy HTML-szövegből a címkéket kihagyva sima szöveges kimenetet kell készítenünk. Ezt természetesen elvégezhetnénk a Python valamelyik feldolgozójának segítségével is, de szabályos kifejezések használatával is elkészíthetünk egy egyszerű eszközt. Három műveletet kell végrehajtanunk: törölnünk kell az összes címkét, a karakteregyedeket az általuk jelölt tényleges karakterekre kell cserélnünk, valamint üres sorokat kell beszúrnunk a bekezdések elválasztására. Íme egy függvény (a `html2text.py` programból), amely elvégzi mindezt:

```
def html2text(html_text):
    def char_from_entity(match):
        code = html.entities.name2codepoint.get(match.group(1), 0xFFFD)
        return chr(code)

    text = re.sub(r"<!--(?:.|\n)*?-->", "", html_text)          #1
    text = re.sub(r"<[Pp][^>]*?(?!</)>", "\n\n", text)          #2
    text = re.sub(r"<[^>]*?>", "", text)                        #3
    text = re.sub(r"&#(\d+);", lambda m: chr(int(m.group(1))), text)
    text = re.sub(r"&([A-Za-z]+);", char_from_entity, text)     #5
    text = re.sub(r"\n(?:[ \xA0\t]+\n)+", "\n", text)           #6
    return re.sub(r"\n\n+", "\n\n", text.strip())               #7
```

Az első szabályos kifejezés, a `<!-(?:.|\n)*?->`, a HTML-megjegyzésekre illeszkedik, beleértve azokat is, amelyekbe más HTML-elemeket ágyaztak. A `re.sub()` függvény az összes találatot a cserekarakterekre cseréli, illetve törli őket, ha a csereszöveg egy üres karakterlánc, mint itt. (A találatok számának maximumát a kifejezés végén paraméterként átadott egész számmal határozhatjuk meg.)

Ügyeltünk rá, hogy nem mohó (minimális) illesztést végezzünk, hogy minden találat esetében csak egy megjegyzést töröljünk. Ha nem így tettünk volna, az első megjegyzés elejétől az utolsó megjegyzés végéig mindent törölnénk.

A `re.sub()` függvény paraméterként nem fogad el jelzőket, tehát a pont (`.`) azt jelenti, hogy „az újsor kivételével bármely karakter", ezért pontot vagy `\n`-t kell keresnünk, mégpedig nem karakterosztály, hanem „vagy" művelet segítségével, mivel a karakterosztályokon belül a . literális jelentésű, tehát egy pontot jelöl. Egy másik lehetőség, ha a szabályos kifejezést a beágyazott jelzővel kezdjük (például: `(?s)<!-.*?->`), vagy ha a `re.DOTALL` jelzővel lefordítunk egy szabályoskifejezés-objektumot, mely esetben a szabályos kifejezés egyszerűen ennyi lenne: `<!-.*?->`.

A második szabályos kifejezés, a `<[Pp][^>]*?(?!</)>`, a nyitó bekezdéscímkékre (például `<P>` vagy `<p align=center>`) illeszkedik. Megkeresi a nyitó `<p` (vagy `<P`) karaktereket, majd (nem mohó illesztéssel) a jellemzőket, végül pedig a záró `>` karaktert, feltéve, hogy azt nem `/` előzi meg (erről egy negatív visszatekintő feltételezés gondoskodik), mivel az záró bekezdéscímkét jelölne. A `re.sub()` függvény második hívása ezt a szabályos kifejezést használja arra, hogy a nyitó bekezdéscímkéket két újsor karakterre cserélje (ami a sima szöveges fájlokban a bekezdések elválasztásának szabványos módja).

A harmadik szabályos kifejezés, a `<[^>]*?>`, minden címkére illeszkedik, és a harmadik `re.sub()` függvényhívásban használjuk a fennmaradó címkék törlésére.

A HTML-egyedek (HTML-karakteregyedek) a nem ASCII karakterek ASCII-karakterekkel történő leírására szolgálnak. Kétféle alakjuk lehetséges: az egyik az `&`*`név`*`;` – ahol a *`név`* a karakter neve (például a © karakter esetében `©`), a másik pedig az `&#`*`számjegyek`*`;` – ahol a *`számjegyek`* az Unicode-kódpontot azonosító decimális számjegyek (a ¥ karakter esetében például `¥`). A `re.sub()` negyedik hívása az `&#(\d+);` szabályos kifejezést használja, amely a számjegyes alakra illeszkedik, és a számjegyeket az 1-es elfogási csoportba olvassa. Literális csereszöveg helyett egy lambdafüggvényt adunk át. Amikor a `re.sub()` egy függvényt kap, minden egyezés megtalálásakor meghívja azt, és egyetlen paraméterként átadja neki a találati objektumot. A `lambda` függvényben (karakterláncként) kinyerjük a számjegyeket, a beépített `int()` függvénnyel egész számmá alakítjuk őket, majd a szintén beépített `chr()` függvénnyel kiolvassuk az adott kódpont Unicode-karakterét. Csereszövegként a függvény visszatérési értékét (illetve a `lambda` kifejezések esetében a kifejezés eredményét) használjuk.

Az ötödik `re.sub()` hívás az `&([A-Za-z]+);` szabályos kifejezéssel a nevesített egyedeket fogja el. Az alapkönyvtár `html.entities` modulja tartalmazza az egyedek szótárait, köztük a `name2codepoint`-ot, amelynek a kulcsai egyednevek, az értékei pedig egész értékű kódpontok. A `re.sub()` függvény minden találat esetében a helyi `char_from_entity()` függvényt hívja meg. A `char_from_entity()` függvény a `dict.get()` függvényt használja, `0xFFFD` alapértelmezett paraméterrel (ez a szabványos Unicode-cserekarakter kódpontja, amelyet gyakran ❓ formában ábrázolnak). Ez biztosítja, hogy mindig kapjunk egy kódpontot, amelyet a `chr()` függvénynek átadva visszaadhatjuk a nevesített egyed lecserélésére alkalmas karaktert – illetve az Unicode-cserekaraktert, amennyiben az egyed neve nem érvényes.

A hatodik `re.sub()` hívás szabályos kifejezése, a `\n(?:[ \xA0\t]+\n)+`, a csak üreshelyeket tartalmazó sorok törlésére szolgál. A karakterosztályunk egy szóközt, egy nem törő szóközt (amelyre az előző szabályos kifejezésben az ` ` egyedeket cseréltük) és egy tabulátort tartalmaz. A szabályos kifejezés akkor illeszkedik, ha olyan újsor karaktert talál (egy sor végén, amelyet egy vagy több, csak üreshelyeket tartalmazó sor követ), amely után legalább egy csak üreshelyeket tartalmazó sor következik. Mivel a találatnak az újsor karakter is része, a csak üreshelyeket tartalmazó sorokat megelőző sorban a találatot egyetlen újsor karakterre kell cserélnünk, máskülönben nem csak a csak üreshelyeket tartalmazó sorokat törölnénk, hanem a megelező sor újsor karakterét is.

A hetedik és egyben utolsó `re.sub()` hívás eredményét visszaadjuk a hívónak. Ez a szabályos kifejezés, a `\n\n+`, arra szolgál, hogy két vagy több újsor karakter sorozatát pontosan két újsor karakterre cseréljük, vagyis hogy biztosítsuk, hogy minden bekezdést csak egyetlen üres sor válasszon el.

A HTML-es példában egyik csereszöveget sem közvetlenül a találatból vettük (bár HTML-egyedneveket és -számokat használtunk), egyes esetekben azonban a csereszövegben szerepelnie kell az illeszkedő szövegnek vagy annak egy részének is. Például ha van egy nevekből álló listánk, amelyben a nevek *Keresztnév Középsőnév1…KözépsőnévN Családnév* alakban állnak, és tetszőleges számú középső név (akár nulla is) lehet bennük, és a listából olyan új változatot szeretnénk készíteni, amelynek az elemei *Családnév,KeresztnévKözépsőnév1…KözépsőnévN* alakban szerepelnek, ezt egy szabályos kifejezés segítségével könnyen megtehetjük:

```
new_names = []
for name in names:
    name = re.sub(r"(\w+(?:\s+\w+)*)\s+(\w+)", r"\2, \1", name)
    new_names.append(name)
```

A szabályos kifejezés első része, a `(\w+(?:\s+\w+)*)`, a keresztnévre illeszkedik az első `\w+` kifejezéssel, és nulla vagy több középső névre a `(?:\s+\w+)*` kifejezéssel. Ez utóbbi nulla vagy több szóközt is megtalál, ha egy szó következik utána. A szabályos kifejezés második része, a `\s+(\w+)`, a keresztnevet (és a középső neveket) követő szóközre, illetve a családnévre illeszkedik.

Ha a szabályos kifejezés kissé zavarosnak tűnik, nevesített elfogási csoportokkal javíthatunk az olvashatóságán, és megkönnyíthetjük a későbbi módosítását:

```
name = re.sub(r"(?P<forenames>\w+(?:\s+\w+)*)"
              r"\s+(?P<surname>\w+)",
              r"\g<surname>, \g<forenames>"; name)
```

Az elfogott szövegre egy `sub()` vagy `subn()` függvényből vagy tagfüggvényből hivatkozhatunk, `\i` vagy `\g<azonosító>` formában, ahol az *i* az elfogási csoport sorszáma, az *azonosító* pedig az elfogási csoport neve vagy száma – az `\1` tehát ugyanaz, mint a `\g<1>`, a példánkban pedig a `\g<forenames>`-nek felel meg. Ugyanezt az alakot a találati objektumok `expand()` tagfüggvényének átadott karakterláncban is használhatjuk.

Miért nem a teljes nevet olvassa ki a szabályos kifejezés első része (hiszen mohó illesztést végez)? Nos, valójában a teljes nevet kiolvassa, de az illesztés meghiúsul, mert bár a középső nevek nulla vagy több alkalommal illeszkedhetnek, a családnévnek pontosan egyszer kell, a középső neveket illesztő rész azonban mohón mindent kiolvas. Miután nem járt sikerrel, a szabályoskifejezés-motor „visszateker", és elereszti az utolsó „középső nevet", így lehetővé válik a családnév illeszkedése. A mohó illesztés annyi egyezést keres, amennyit csak lehet, de megáll, ha a további találatok meghiúsítanák az illesztést.

12.4. táblázat *A szabályoskifejezés-modul függvényei*

Utasításforma	Leírás
`re.compile(r,f)`	Az *r* lefordított szabályos kifejezést adja vissza, a jelzőit *f*-re állítva, ha ezt a paramétert is megadjuk.
`re.escape(s)`	Az *s* karakterláncot adja vissza, minden nem alfanumerikus karaktert fordított perjellel levédve, ezért a visszaadott karakterláncban nem lesznek különleges szabályoskifejezés-karakterek.
`re.findall(r,s,f)`	Az *r* szabályos kifejezés minden egymást át nem fedő illeszkedését visszaadja az *s* karakterláncból (az *f* jelzőkkel módosítva, ha ezt a paramétert is megadjuk). Ha a szabályos kifejezés elfogást is tartalmaz, minden találatot fogások rendezett sorozataként kapunk vissza.
`re.finditer(r,s,f)`	Egy találati objektumot ad vissza az *r* szabályos kifejezés minden egymást át nem fedő illeszkedésére az *s* karakterláncban (az *f* jelzőkkel módosítva, ha ezt a paramétert is megadjuk).
`re.match(r,s,f)`	Egy találati objektumot ad vissza, ha az *r* szabályos kifejezés az *s* karakterlánc elejére illeszkedik (az *f* jelzőkkel módosítva, ha ezt a paramétert is megadjuk), máskülönben a visszatérési érték `None` lesz.
`re.search(r,s,f)`	Egy találati objektumot ad vissza, ha az *r* szabályos kifejezés bárhol illeszkedik az *s* karakterláncban (az *f* jelzőkkel módosítva, ha ezt a paramétert is megadjuk), máskülönben a visszatérési érték `None` lesz.

12.4. táblázat *A szabályoskifejezés-modul függvényei* *(folytatás)*

Utasításforma	Leírás
`re.split(r,s,m)`	Azoknak a karakterláncoknak a listáját adja vissza, amelyeket az *s* karakterláncnak az *r* szabályos kifejezés minden előfordulásánál történő szétvágásával kapunk (*m* számú vágásig, illetve ha ezt a paramétert nem adjuk meg, annyi darabig, amennyi csak lehetséges). Ha a szabályos kifejezés elfogást is tartalmaz, a fogások bekerülnek a listába az általuk szétvágott részek közé.
`re.sub(r,x,s,m)`	Az *s* karakterlánc másolatát adja vissza, az *r* szabályos kifejezés minden illeszkedését (vagy amennyit az *m* paraméterrel meghatározunk) *x*-re cserélve, ami egy karakterlánc vagy egy függvény lehet (lásd a szövegben).
`re.subn(r,x,s,m)`	Ugyanaz, mint a `re.sub()`, azzal a különbséggel, hogy az eredményként kapott karakterláncot és a végrehajtott cserék számát adja vissza, kételemű rendezett sorozatként (rendezett párként).

12.5. táblázat *A szabályoskifejezés-modul jelzői*

Jelző	Jelentés
`re.A` vagy `re.ASCII`	Hatására a `\b`, `\B`, `\s`, `\S`, `\w` és `\W` feltételezések ASCII-ként kezelik a karakterláncokat. Ezek a gyorsírásos karakterosztályok alapértelmezés szerint az Unicode-szabványra támaszkodnak.
`re.I` vagy `re.IGNORECASE`	Hatására a szabályos kifejezés a kis- és nagybetűk különbözőségét figyelmen kívül hagyva illeszkedik.
`re.M` vagy `re.MULTILINE`	Hatására a kalapjel (`^`) minden új sor elejére, illetve minden újsor karakter után illeszkedik, a dollárjel (`$`) pedig az újsor karakterek elé, illetve a sorok végére.
`re.S` vagy `re.DOTALL`	Hatására a pont (`.`) minden karakterre illeszkedik, az újsor karaktereket is beleértve.
`re.X` vagy `re.VERBOSE`	Lehetővé teszi üreshelyek és megjegyzések beágyazását.

12.6. táblázat *A szabályoskifejezés-objektumok tagfüggvényei*

Utasításforma	Leírás
`rx.findall(s,kezdet,vég)`	A szabályos kifejezés minden egymást át nem fedő illeszkedését visszaadja az *s* karakterláncból (illetve annak a *kezdet*:*vég* szeletéből). Ha a szabályos kifejezés elfogást is tartalmaz, minden találatot fogások rendezett sorozataként kapunk vissza.
`rx.finditer(s,kezdet,vég)`	Egy találati objektumot ad vissza a szabályos kifejezés minden egymást át nem fedő illeszkedésére az *s* karakterláncban (illetve annak a *kezdet*:*vég* szeletében).

12.6. táblázat *A szabályoskifejezés-objektumok tagfüggvényei* *(folytatás)*

Utasításforma	Leírás
`rx.flags`	Azok a jelzők, amelyeket a szabályos kifejezés lefordításakor beállítottunk.
`rx.groupindex`	Szótár, amelynek a kulcsai elfogási csoportok nevei, az értékei pedig a csoportok sorszámai. Ha nem használunk neveket, akkor üres.
`rx.match(`*`s,kezdet,vég`*`)`	Egy találati objektumot ad vissza, ha a szabályos kifejezés az *s* karakterlánc elejére (vagy az *s* `kezdet:vég` szeletének elejére) illeszkedik, máskülönben a visszatérési érték `None` lesz.
`rx.pattern`	A karakterlánc, amelyből a szabályos kifejezést lefordítottuk.
`rx.search(`*`s,kezdet,vég`*`)`	Egy találati objektumot ad vissza, ha a szabályos kifejezés bárhol illeszkedik az *s* karakterláncban (illetve annak a `kezdet:vég` szeletében), máskülönben a visszatérési érték `None` lesz.
`rx.split(`*`s,m`*`)`	Azoknak a karakterláncoknak a listáját adja vissza, amelyeket az *s* karakterláncnak a szabályos kifejezés minden előfordulásánál történő szétvágásával kapunk (*m* számú vágásig, illetve ha ezt a paramétert nem adjuk meg, annyi darabig, amennyi csak lehetséges). Ha a szabályos kifejezés elfogást is tartalmaz, a fogások bekerülnek a listába az általuk szétvágott részek közé.
`rx.sub(`*`x,s,m`*`)`	Az *s* karakterlánc másolatát adja vissza, a szabályos kifejezés minden illeszkedését (vagy amennyit az *m* paraméterrel meghatározunk) *x*-re cserélve, ami egy karakterlánc vagy egy függvény lehet (lásd a szövegben).
`rx.subn(`*`x,s,m`*`)`	Ugyanaz, mint a `re.sub()`, azzal a különbséggel, hogy az eredményként kapott karakterláncot és a végrehajtott cserék számát adja vissza, kételemű rendezett sorozatként (rendezett párként).

Ha a név például „James W. Loewen", a szabályos kifejezés először a teljes névre illeszkedik, vagyis így: `James W. Loewen`. Ez kielégíti a szabályos kifejezés első részét, de a családnevet kereső résznek nem hagy semmit, és mivel a családnév kötelező (a beleértett mennyiségjelzője 1), a szabályos kifejezés nem jár sikerrel. A középső neveket kiolvasó rész mennyiségjelzője a `*`, tehát nulla vagy több alkalommal illeszkedhet (esetünkben kétszer: „ W." és „ Loewen"), ezért a szabályoskifejezés-értelmező motor ráveheti, hogy adjon fel valamennyit a találataiból, anélkül, hogy ez meghiúsítaná a rész illeszkedését. A szabályos kifejezés tehát visszateker, feladja az utolsó `\s+\w+`-t (mármint a „ Loewen"-t), így a találat `James W. Loewen` lesz, ami kielégíti a teljes szabályos kifejezést, és a két találati csoport a megfelelő szövegeket fogja tartalmazni.

Ha a „vagy" műveletet (`|`) alkalmazzuk két vagy több alternatívát elfogva, nem tudjuk, hogy melyik illeszkedett, ezért azt sem tudjuk, hogy melyik elfogási csoportból kell kinyernünk a szöveget. Természetesen végigjárhatjuk az összes csoportot, hogy megke-

ressük azt, amelyik nem üres, de ilyen helyzetben gyakran a találati objektum `lastindex` jellemzője adhatja meg a kívánt csoport sorszámát. Ennek szemléltetésére, illetve hogy kicsit több gyakorlatot szerezzünk a szabályos kifejezések használatában, még egy utolsó példát szemügyre veszünk.

Tegyük fel, hogy azt szeretnénk kideríteni, hogy milyen kódolást használ egy HTML-, XML- vagy Python-fájl. Megnyithatnánk a fájlt bináris módban, és egy `bytes` objektumba olvashatnánk belőle mondjuk az első 1000 bájtot, majd a fájlt bezárva megnézhetnénk a kódolást a `bytes` objektumban, és szöveges módban a talált kódolással vagy egy alapértelmezettel (például az UTF-8 kódolással) újra megnyithatnánk az állományt. A szabályoskifejezés-motor karakterlánccként várja a szabályos kifejezéseket, a szöveg azonban, amelyre a szabályos kifejezést alkalmazzuk, `str`, `bytes` vagy `bytearray` objektum is lehet, és az utolsó kettő esetében minden függvény és tagfüggvény `bytes` objektumokat ad vissza karakterláncok helyett, a `re.ASCII` jelző pedig automatikusan bekapcsolódik.

A HTML-fájlok esetében a kódolást általában egy `<meta>` elemben határozzák meg (ha egyáltalán megadják), például így: `<meta http-equiv='Content-Type' content='text/html;charset=ISO-8859-1' />`. Az XML-fájlok alapértelmezés szerint UTF-8 kódolásúak, de ez felülbírálható, például így: `<?xml version="1.0" encoding="Shift_JIS"?>`. A Python 3-fájlok alapértelmezett kódolása ugyancsak az UTF-8, de ez ismét csak felülbírálható egy közvetlenül a shebang-sor után írt sorral, valahogy így: `#encoding: latin1` vagy `#-*-coding: latin1-*-`.

A kódolást a következőképpen deríthetjük ki, amennyiben a `binary` változó egy `bytes` objektum, amely egy HTML-, XML- vagy Python-fájl első 1000 bájtját tartalmazza:

```
match = re.search(r"""(?<![-\w])                          #1
                      (?:(?:en)?coding|charset)           #2
                      (?:=(["']))?([-\w]+)(?(1)\1)        #3
                      |:\s*([-\w]+))""".encode("utf8"),
                  binary, re.IGNORECASE|re.VERBOSE)
encoding = match.group(match.lastindex) if match else b"utf8"
```

Ahhoz, hogy egy `bytes` objektumban kereshessünk, meg kell adnunk egy mintát, ami ugyancsak egy `bytes` objektum. Ebben az esetben a kényelem kedvéért egy nyers karakterláncot szeretnénk használni, ezért azt is használunk, és a `re.search()` függvény első paramétereként átadva alakítjuk `bytes` objektummá.

Magának a szabályos kifejezésnek az első része egy utófeltételezés, ami azt mondja ki, hogy a találatot nem előzheti meg kötőjel vagy szókarakter. A kifejezés második része az „encoding", „coding" vagy „charset" szóra illeszkedik, és így is írhattuk volna: **`(?:encoding|coding|charset)`**. A harmadik részt két sorra tördeltük, hogy kihangsú-

lyozzuk, hogy két „vagy-vagy” részből áll – `=([„']?([-\w]+)(?(1)\1)` és `:\s*([-\w]+)` –, amelyek közül csak az egyik illeszkedhet. Az első egy egyenlőségjel, amelyet egy vagy több szókarakter vagy kötőjel követ (ezek megegyező idézőjelek között is állhatnak, amit egy feltételes illesztéssel vizsgálunk), a második pedig egy kettőspont, utána egy nem kötelező üreshely, majd egy vagy több szókarakter vagy kötőjel. (Emlékezzünk vissza, hogy a kötőjel a karakterosztályokon belül literális értelmű, amennyiben első karakterként szerepel, más esetben azonban egy olyan karaktertartományt jelöl, mint például a `[0-9]`).

12.7. táblázat *A találati objektumok jellemzői és tagfüggvényei*

Utasításforma	Leírás
`m.end(`*g*`)`	A találat végének helyét adja vissza a szövegen belül a *g* csoport számára (ha megadjuk a csoportot), illetve a 0. csoport esetében a teljes találatot. Ha a csoport nem vett részt az illesztésben, a visszatérési érték –1 lesz.
`m.endpos`	A keresés végének helye (a szöveg vége vagy a `match()`, illetve a `search()` függvénynek átadott *vég*).
`m.expand(`*s*`)`	Az *s* karakterláncot adja vissza, a fogásjelzőket (`\1`, `\2`, `\g<`*név*`>` és hasonlók) a megfelelő fogásokra cserélve.
`m.group(`*g*`,...)`	A *g* sorszámozott vagy nevesített elfogási csoportot adja vissza. Ha egynél több csoportot adunk meg, a megfelelő elfogási csoportok rendezett sorozatát kapjuk vissza (a teljes találat a 0. csoport).
`m.groupdict(`*alapértelmezés*`)`	Az összes nevesített elfogási csoport szótárát adja vissza, amelyben a nevek a kulcsok, a fogások pedig az értékek. Ha megadjuk az alapértelmezést, akkor az illesztésben részt nem vevő elfogási csoportok ezt az értéket kapják.
`m.groups(`*alapértelmezés*`)`	Az 1-estől az összes elfogási csoport rendezett sorozatát adja vissza. Ha megadjuk az alapértelmezést, akkor az illesztésben részt nem vevő elfogási csoportok ezt az értéket kapják.
`m.lastgroup`	A legmagasabb számú illeszkedő elfogási csoport neve, vagy `None`, ha nem volt ilyen, illetve ha nem használtunk neveket.
`m.lastindex`	A legmagasabb számú illeszkedő elfogási csoport sorszáma, vagy `None`, ha nem volt ilyen.
`m.pos`	A keresés kezdetének helye (a szövege eleje vagy a `match()`, illetve a `search()` függvénynek átadott *kezdet*).
`m.re`	Az a szabályos kifejezés, amely az adott találati objektumot adta.
`m.span(`*g*`)`	A találat kezdetének és végének helyét adja vissza a szövegen belül a *g* csoport számára (ha megadjuk a csoportot), illetve a 0. csoport esetében a teljes találatot. Ha a csoport nem vett részt az illesztésben, a visszatérési érték `(-1, -1)` lesz.

12.7. táblázat *A találati objektumok jellemzői és tagfüggvényei* *(folytatás)*

Utasításforma	Leírás
`m.start(g)`	A találat kezdetének helyét adja vissza a szövegen belül a *g* csoport számára (ha megadjuk a csoportot), illetve a 0. csoport esetében a teljes találatot. Ha a csoport nem vett részt az illesztésben, a visszatérési érték –1 lesz.
`m.string`	A `match()`, illetve a `search()` függvénynek átadott karakterlánc.

A `re.IGNORECASE` jelzőt használtuk, hogy ne azt kelljen írnunk, hogy **`(?:(?:[Ee][Nn])?[Cc][Oo][Dd][Ii][Nn][Gg]|[Cc][Hh][Aa][Rr][Ss][Ee][Tt])`**, illetve a `re.VERBOSE` jelzőt, hogy szépen elrendezhessük a szabályos kifejezést, és megjegyzéseket is fűzhessünk hozzá (ebben az esetben csak sorszámokat, hogy könnyebben hivatkozhassunk a kifejezés egyes részeire a szövegben).

Három elfogási csoportunk van; mindegyik a kifejezés harmadik részében: a **`(["'])?`**, ami a nem kötelező nyitó idézőjelet fogja el, a **`([-\w+])`**, ami egy egyenlőségjelet követő kódolást, és (a következő sorban) a második **`([-\w+])`**, ami egy kettőspontot követő kódolást. Minket csak a kódolás érdekel, ezért vagy a második, vagy a harmadik elfogási csoportot kell kinyernünk, amelyek közül csak az egyik illeszkedhet, mivel „vagy" kapcsolatban állnak egymással. Az utolsó *illeszkedő* elfogási csoport (ebben a példában találat esetén ez a 2. vagy 3. csoport) sorszámát a `lastindex` jellemző tárolja, amelyből kiolvassuk a megfelelőt, illetve ha nem volt illeszkedés, egy alapértelmezett kódolást alkalmazunk.

Most, hogy a `re` modul leggyakrabban használt szolgáltatásai közül az összeset láttuk működés közben, a rész lezárásaként csupán megemlítünk még egy utolsó függvényt. A `re.split()` függvény (vagy a szabályoskifejezés-objektum `split()` tagfüggvénye) karakterláncok felbontására képes egy szabályos kifejezés alapján. Gyakori, hogy egy szöveget a szóközöknél kell szétvágni, hogy megkapjuk a szavak listáját.
Ezt a `re.split(r"\s+", text)` hívással hajthatjuk végre, amely egy szólistát ad vissza (pontosabban egy karakterlánc-listát, amelyben minden karakterlánc illeszkedik az **`\S+`** kifejezésre). A szabályos kifejezések nagyon hatékonyak és hasznosak, és ha egyszer megtanultuk a használatukat, könnyen úgy tekinthetünk minden szövegfeldolgozási feladatra, mint aminek a megoldása egy szabályos kifejezést igényel. Néha azonban a karakterlánc-függvények használata is elegendő, sőt talán helyénvalóbb. A szóközöknél történő szétvágás például ugyanilyen könnyen megoldható a `text.split()` függvénnyel is, mivel a `str.split()` tagfüggvény (így, vagy első paraméterként a `None` értékkel) alapértelmezett viselkedése, hogy az `\s+`-nál vág.

Összefoglalás

A szabályos kifejezések hatékony módját kínálják annak, hogy olyan karakterláncokat keressünk szövegekben, amelyek egy bizonyos mintára illeszkednek, illetve hogy ezeket a karakterláncokat más karakterláncokra cseréljük, amelyek maguk is függhetnek attól, hogy milyen egyezést kerestünk.

Ebben a fejezetben láttuk, hogy a legtöbb karaktert literális értelemben illesztjük, alapértelmezés szerint egyszer. Azt is megtanultuk, hogyan határozhatunk meg karakterosztályokat – illesztendő karakterek egy halmazát –, és hogyan negálhatjuk ezeket a halmazokat, illetve adhatunk meg bennük karaktertartományokat anélkül, hogy minden karaktert külön-külön ki kellene írnunk.

Azt is megtudtuk, hogy miként láthatjuk el mennyiségjelzővel a kifejezéseket, hogy meghatározzuk az illesztések pontos számát, vagy hogy a kifejezés legalább és legfeljebb hányszor illeszkedhet, valamint megtanultuk a mohó és a nem mohó illesztés módját. Arról is tanultunk, hogy miként képezhetünk csoportot egy vagy több kifejezésből, hogy egy egységként adhassuk meg az illesztéseik számát, illetve foghassuk el őket.

A fejezetben azt is megmutattuk, hogy az illesztést hogyan szabályozhatjuk különféle feltételezésekkel, például pozitív vagy negatív előretekintéssel és visszatekintéssel, illetve különböző jelzőkkel, amelyekkel például befolyásolhatjuk a pont jelentését, vagy hogy az illesztés során különbözőnek tekintjük-e a kis- és nagybetűket.

A fejezet utolsó részében azt mutattuk meg, hogy miként használhatjuk a szabályos kifejezéseket a Python-programok környezetében. Megtanultuk, hogyan használhatjuk a `re` modul által biztosított függvényeket, illetve a lefordított szabályos kifejezésekből, valamint a találati objektumokból elérhető tagfüggvényeket. Arról is szót ejtettünk, hogy miként cserélhetjük le a találatokat literális karakterláncokkal, visszirányú hivatkozásokat tartalmazó literális karakterláncokkal, valamint függvényhívások vagy lambdakifejezések eredményével, és megtanultuk, hogyan könnyíthetjük meg a szabályos kifejezések későbbi módosítását nevesített fogások és megjegyzések segítségével.

Gyakorlatok

1. Sok esetben (például egyes webes űrlapokon) a felhasználóknak meg kell adniuk egy telefonszámot, és előfordul, hogy számukra bosszantó módon csak egy adott formátumot fogadnak el. Írjunk olyan programot, amely amerikai telefonszámokat olvas be, a három számjegyű körzetszámot és a hét számjegyből álló helyi számot a következő formátumokban elfogadva: tíz számjegyként; kötőjelekkel, illetve szóközökkel tagolva; valamint a körzetszámot nem kötelező módon zárójelbe téve. A következő alakok közül tehát mindegyik érvényes: 555-555-5555, (555)5555555,

(555) 555 5555 és 5555555555. A telefonszámokat a `sys.stdin`-ről olvassuk be, és mindegyiket írjuk ki az `echo` utasítással „(555) 555 5555" formában, az érvénytelen számok esetében pedig adjunk hibaüzenetet.

A fenti telefonszámokat illesztő szabályos kifejezés mintegy nyolc sorban megírható (bőbeszédű módban), és meglehetősen egyszerű. Egy megoldást a `phone.py` fájlban láthatunk, amely körülbelül huszonöt sor hosszú.

2. Írjunk egy olyan rövid programot, amely egy a parancssorban megadott XML- vagy HTML-fájlt olvas be, és minden jellemzővel rendelkező elem esetében kiírja az elem nevét, alatta a jellemzőivel. Példaként álljon itt egy részlet a program kimenetéből, amelyet akkor kapunk, ha a Python dokumentációjának valamelyik `index.html` fájlját olvassuk be:

```
html
    xmlns = http://www.w3.org/1999/xhtml
meta
    http-equiv = Content-Type
    content = text/html; charset=utf-8
li
    class = right
    style = margin-right: 10px
```

Az egyik megközelítés az lehet, ha két szabályos kifejezést használunk: egyet az elemek és jellemzőik elfogására, egy másikat pedig az egyes jellemzők nevének és értékének kinyerésére. A jellemzőértékek aposztrófok vagy (kettős) idézőjelek között is állhatnak (mely esetben üreshelyeket, valamint olyan idézőjeleket is tartalmazhatnak, amelyek nem egyeznek meg a közrezárásukra használttal), de lehetnek idézőjel nélküliek is (ebben az esetben azonban nem lehet bennük üreshely vagy idézőjel). Valószínűleg az a legegyszerűbb, ha egy olyan szabályos kifejezés megírásával kezdjük, amely külön kezeli az idézőjeles és az idézőjel nélküli értékeket, majd a két szabályos kifejezést egyetlen kifejezésben egyesítjük, hogy mindkét esetet lefedjük. Az a legjobb, ha nevesített csoportokat használunk, mert a szabályos kifejezés így olvashatóbb lesz. A feladat nem egyszerű, különösen mert karakterosztályokon belül nem használhatunk visszirányú hivatkozásokat.

Egy megoldást az `extract_tags.py` fájlban láthatunk, amely kevesebb mint 35 sor hosszú. Az elemeket és a jellemzőket kinyerő szabályos kifejezés mindössze egy sor, a jellemzőneveket és -értékeket kiolvasó kifejezés azonban féltucat sorból áll, és „vagy" művelet, (két egymásba ágyazott) feltételes illesztés, valamint mohó és nem mohó mennyiségjelzők is szerepelnek benne.

13

Bevezetés a grafikus felhasználói felület programozásába

- Párbeszédablakos programok
- Főablakos programok

A Python alapállapotban nem támogatja a grafikus felhasználói felület (GUI, Graphical User Interface) programozását, de ez nem jelent gondot, hiszen más nyelveken számos olyan GUI-programkönyvtárat írtak, amelyeket a Python-programozók felhasználhatnak. Ezt az teszi lehetővé, hogy sok GUI-könyvtárhoz létezik Python-*burkoló* vagy *-kötés*, vagyis olyan csomagok és modulok, amelyeket ugyanúgy tölthetünk be és használhatunk, mint bármely más Python-csomagot és modult, csak éppen olyan szolgáltatásokat tesznek elérhetővé, amelyek nem Python-könyvtárakban találhatók.

A Python alapkönyvtárának része a Tcl/Tk: a Tcl egy szinte nyelvtanmentes parancsnyelv, a Tk pedig egy GUI-könyvtár, amit Tcl-ben és C-ben írtak. A Python `tkinter` modulja a Tk GUI-könyvtárhoz biztosít Python-kötéseket. A Tk-nak három előnye van a Pythonból hozzáférhető más GUI-könytárakkal szemben. Először is, a Pythonnal együtt települ, tehát mindig elérhető. Másodszor, kis méretű (még a Tcl-lel együtt is). Harmadszor pedig, része az IDLE környezetnek, mely környezet nagyon hasznos, ha kísérletezni szeretnénk a Pythonnal, illetve Python-programokat szeretnénk szerkeszteni és hibakeresést végezni bennük.

Sajnos a Tk megjelenése a 8.5-ös változat előtt igencsak elavult volt, és a Tk igen korlátozott elemkészlettel („widget"-ekkel vagy „mütyürökkel", a Windows nyelvén „vezérlőkkel" vagy „tárolókkal") rendelkezett. A Tk-ban ugyan meglehetősen könnyű egyéni vezérlőket létrehozni más vezérlők egyesítésével, ám a Tk nem nyújt semmilyen közvetlen módot arra, hogy a semmiből hozzunk létre egyéni vezérlőket, és úgy rajzoljuk meg azokat, ahogy csak szeretnénk. További Tk-megfelelő vezérlőkhöz juthatunk a Tix könyvtár használatával, amely szintén része a Python alapkönyvtárának, de a Windowstól eltérő rendszereken nem mindig áll rendelkezésre – például az Ubuntun, amely a könyv írásakor csak nem támogatott bővítményként tette lehetővé a használatát. Ezen kívül mind a Tk, mind

a Tix híján van egy Python-központú leírásnak – a dokumentáció nagy részét Tcl/Tk-programozóknak szánták, ezért a Tcl-ben járatlan programozók számára nehezen megfejthető lehet.*

Ha olyan grafikus felületű programokat szeretnénk fejleszteni, amelyeknek minden asztali Python-platformon (vagyis Windowson, Mac OS X-en és Linuxon is) futniuk kell, és csak a Python alaptelepítése áll a rendelkezésünkre, kiegészítő könyvtárak nélkül, egyetlen lehetőségünk van: a Tk használata.

Ha külső könyvtárakat is használhatunk, a lehetőségek száma jelentősen megnő. Választhatjuk például a WCK (Widget Construction Kit, vagyis „vezérlőépítő készlet", `www.effbot.org/zone/wck.htm`) könyvtárat, amely további Tk-megfelelő szolgáltatásokat kínál, például olyan egyéni vezérlők létrehozásának lehetőségét, amelyeknek a tartalmát kóddal határozzuk meg.

A Tk-t mellőző lehetőségek két csoportra oszthatók: azokra, amelyek csak egy adott rendszeren működnek, illetve azokra, amelyek rendszerfüggetlenek. A rendszerfüggő GUI-könyvtárak elérhetővé teszik az adott rendszerre jellemző szolgáltatásokat, de ennek az az ára, hogy a rendszerhez kötnek minket. A három legszélesebb körben elterjedt, Python-kötésekkel rendelkező rendszerfüggetlen GUI-könyvtár a PyGtk (`www.pygtk.org`), a PyQt (`www.riverbankcomputing.com/software/pyqt`) és a wkPython (`www.wkpython.org`). Mindhárom lényegesen több vezérlőt kínál, mint a Tk, szebb megjelenésű grafikus felületet állít elő (bár a Tk 8.5 megjelenésével a különbség csökkent), és lehetővé teszi a kóddal megrajzolt egyéni vezérlők létrehozását is. Ezen felül mindegyiket könnyebb megtanulni és használni, mint a Tk-t; a leírásuk is bővebb és sokkal inkább Python-központú, mint a Tk-é; és általában véve a PyGtk-t, a PyQt-t vagy a wxPythont használó programok kevesebb kódot igényelnek, és jobb eredményt adnak, mint a Tk-val írtak.

Korlátai és bosszantó hiányosságai ellenére azonban a Tk segítségével is készíthetünk hasznos grafikus felületű programokat – a Python világában az IDLE a legismertebb. Továbbá könyvünk írásakor a Tk fejlesztése meglódult, és a témákat kínáló Tk 8.5 már olyan Tk-programok készítését tette lehetővé, amelyek sokkal jobban hasonlítanak az adott rendszer sajátjaira, valamint számos régen várt új vezérlő is belekerült.

Ebben a fejezetben csupán ízelítőt nyújthatunk a Tk-programozásból. Ha komoly grafikus felületeket szeretnénk építeni, a legjobb, ha átugorjuk ezt a fejezetet (mivel itt a grafikus felület programozásának csak a hagyományos Tk-megközelítését mutatjuk be), és valamelyik

---

*Az egyetlen Python/Tk-könyv, amiről a szerzőnek tudomása van, John Grayson *Python and Tkinter Programming* című, 2000-ben kiadott (és ezért egyes részeit tekintve elavult) műve (ISBN 1884777813). Tcl/Tk témában Brent Welch és Ken Jones *Practical Programming in Tcl and Tk* című kötetét (ISBN 0130385603) érdemes elolvasni. A Tcl/Tk teljes leírását a www.tcl.tk internetcímen érhetjük el.

külső könyvtárat vesszük igénybe. Ha azonban a Tk az egyetlen választási lehetőségünk, akkor legalább annyi tudással kell rendelkeznünk a Tcl nyelvről, amennyi elég ahhoz, hogy megértsük a Tk leírását.

A következőkben a Tk segítségével két grafikus felületű programot fogunk elkészíteni. Az első egy igen kis méretű párbeszédablakos program, amely kamatos kamatok kiszámítását végzi. A második, főablakot használó program már kidolgozottabb lesz, és könyvjelzők (nevek és URL-ek) listáját kezeli majd. Mivel az adatok egyszerűek lesznek, semmi nem fogja elterelni a figyelmünket a grafikus felület programozását érintő kérdésekről. A könyvjelző-kezelő programot bemutató részben látni fogjuk, hogyan hozhatunk létre egy egyéni párbeszédablakot, illetve egy menükkel és eszköztárakkal rendelkező főablakot, valamint azt is, hogy miként kombinálhatjuk a kettőt egy teljes értékű, működőképes program előállításához. Először azonban át kell tekintenünk a grafikus felület programozásának néhány alapkérdését, mivel a grafikus programok készítése némileg eltér a konzolos programokétól.

A Pythonban a konzolos programok és a modulfájlok kiterjesztése mindig `.py`, a grafikus felületű programok esetében azonban a `.pyw` kiterjesztést használjuk (kivéve a modulfájlokat, amelyek mindig `.py` kiterjesztésűek). Mind a `.py`, mind a `.pyw` megfelelően működik Linuxon is, Windowson viszont a `.pyw` kiterjesztés biztosítja, hogy a `python.exe` helyett a `pythonw.exe` legyen az értelmező, amely gondoskodik róla, hogy amikor grafikus felületű Python-programot hajtunk végre, ne jelenjen meg egy szükségtelen konzolablak. A Mac OS X a Windowshoz hasonlóan működik, és a GUI-programokhoz a `.pyw` kiterjesztést használja.

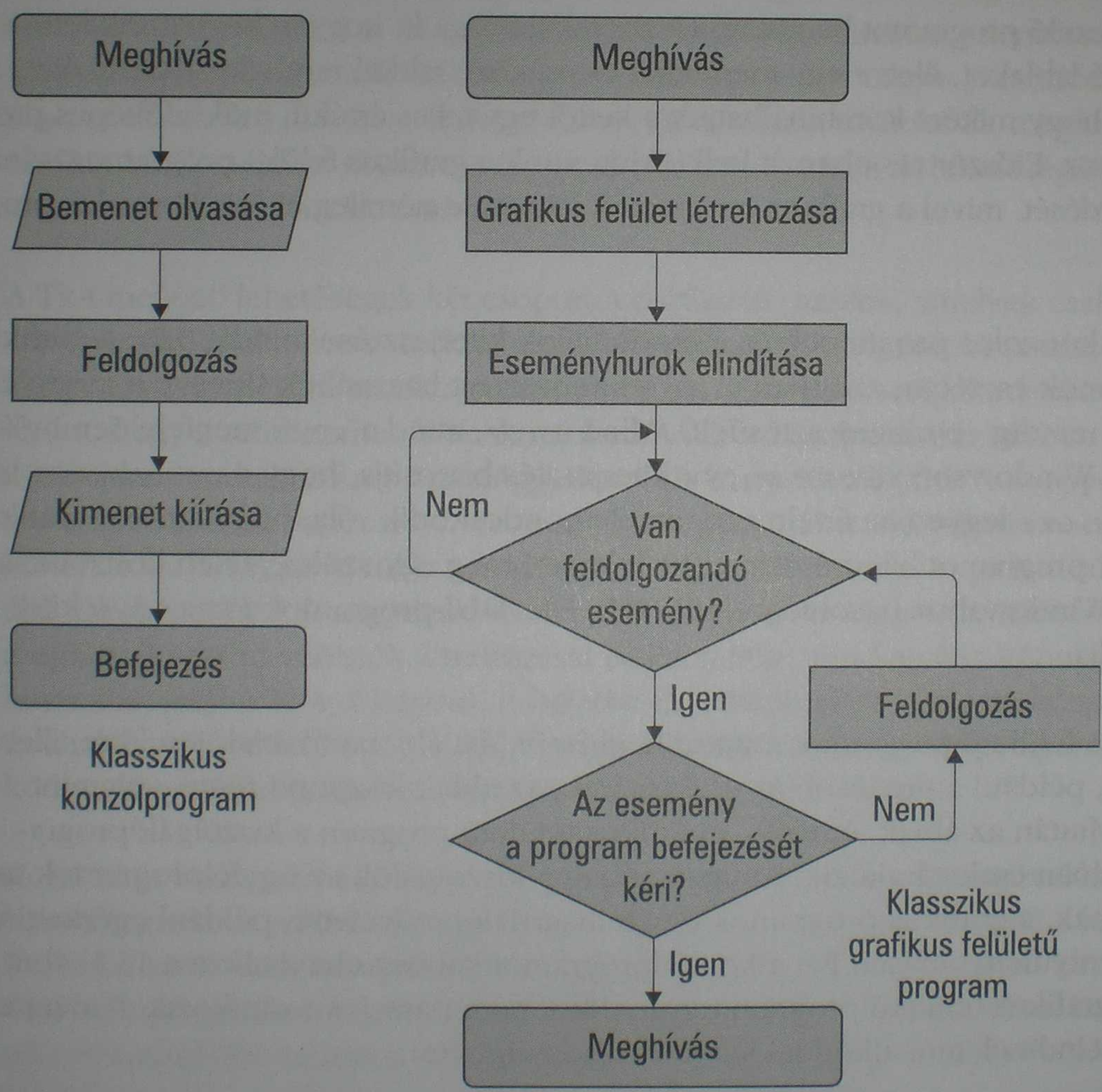
Amikor grafikus felületű programot futtatunk, először általában a főablak jön létre, illetve annak vezérlői, például a menüsor, az eszköztárak, az ablak központi része, valamint az állapotsor. Miután az ablak létrejött, a grafikus felületű program a kiszolgálóprogramokhoz hasonlóan csak várakozik. Amíg azonban a kiszolgálók az ügyfélprogramok csatlakozására várnak, a grafikus programok egy felhasználói műveletre, például egérkattintásra vagy billentyűlenyomásra. Ezt a konzolprogramokkal összehasonlítva a 13.1. ábra szemlélteti. A grafikus felületű program ugyanakkor nem passzívan várakozik, hanem egy *eseményhurkot* indít el, ami álkóddal leírva valahogy így fest:

```
while True:
    event = getNextEvent()
    if event:
        if event == Terminate:
            break
    processEvent(event)
```

Amikor a felhasználó valamilyen műveletet hajt végre a programban, vagy más esemény történik, például lejár egy időzítő, vagy a program ablaka aktívvá válik (például mert egy másik programot bezártak), egy esemény jön létre a GUI-könyvtárban, és bekerül

az eseménysorba. A program eseményhurka folyamatosan ellenőrzi, hogy van-e feldolgozandó esemény, és ha talál ilyet, feldolgozza azt (vagy feldolgozásra átadja az eseményhez társított függvénynek vagy tagfüggvénynek).

GUI-programozóként a GUI-könyvtárra bízhatjuk az eseményhurok létrehozását. A mi feladatunk az, hogy létrehozzuk azokat az osztályokat, amelyek a programunk által igényelt ablakokat és vezérlőket jelképezik, és ellássuk azokat a felhasználói műveletekre válaszolni képes tagfüggvényekkel.

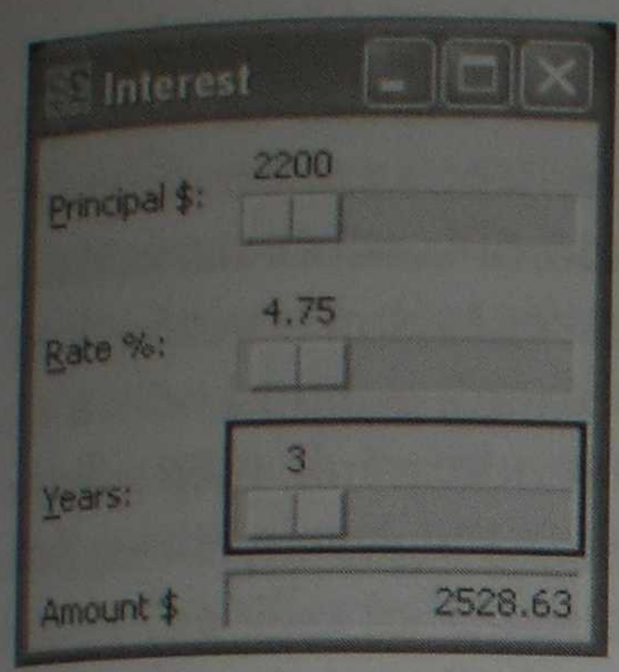


13.1. ábra

Konzolos programok és grafikus felületű programok

Párbeszédablakos programok

Az első program, amelyet megvizsgálunk, az Interest (Kamat) nevű kamatszámító program. Ez egy párbeszédablakos program (ami úgy értendő, hogy nincsenek menüi), amellyel a felhasználó kamatos kamatok kiszámítását hajthatja végre. A programot a 13.2. ábrán láthatjuk.



13.2. ábra

Az Interest program

A legtöbb objektumközpontú grafikus programban egy egyéni osztályt használnak, amely egyetlen főablakot vagy párbeszédablakot ábrázol, a rajta elhelyezett vezérlők többsége pedig szabványos vezérlők egy-egy példánya; ilyen például a gomb vagy a jelölőnégyzet, amelyet a könyvtár kínál. A legtöbb rendszerfüggetlen GUI-könyvtárhoz hasonlóan a Tk nem igazán tesz különbséget az ablakok és a vezérlők között, hanem az ablakot egyszerűen olyan vezérlőnek tekinti, amelynek nincs szülővezérlője (vagyis nem egy másik vezérlő belsejében található). A szülővezérlővel nem rendelkező vezérlők (vagyis ablakok) automatikusan keretet és díszítőelemeket (például címsort és bezáró gombot) kapnak, és általában más vezérlőket tartalmaznak.

A legtöbb vezérlő egy másik vezérlő gyermekeként jön létre (és a szülő belsejében tárolódik), míg az ablakok a `tkinter.Tk` objektum gyermekei. Ez egy olyan objektum, amely az alkalmazást jelképezi – hamarosan visszatérünk rá. A vezérlők és az ablakok (amelyeket felső szintű vezérlőknek is hívnak) megkülönböztetésén túl a szülő-gyermek viszonyok biztosítják, hogy a vezérlők a megfelelő sorrendben törlődjenek, illetve hogy a szülő törlésekor a gyermekvezérlői is automatikusan megsemmisüljenek.

A felhasználói felületet az előkészítő kód hozza létre (tehát ez végzi el a vezérlők hozzáadását és elrendezését, valamint az egér- és billentyűkötések meghatározását), míg a többi tagfüggvény a felhasználó műveleteire válaszol. A Tk lehetővé teszi egyéni vezérlők létrehozását is, vagy úgy, hogy egy kész vezérlőből, például a `tkinter.Frame`-ből alosztályt származtatunk, vagy úgy, hogy egy szokványos osztályt hozunk létre, és jellemzőként vezérlőket adunk hozzá. Mi most az alosztályok létrehozásának módszerét fogjuk alkalmazni, a következő példában azonban mindkét megközelítést bemutatjuk.

Mivel az Interest programnak csak egyetlen főablaka van, egyetlen osztályban fogjuk megvalósítani. Kezdjük az osztály előkészítő függvényének vizsgálatával, amelyet öt részre bontottunk, mivel elég hosszú:

```
class MainWindow(tkinter.Frame):
    def __init__(self, parent):
        super(MainWindow, self).__init__(parent)
        self.parent = parent
        self.grid(row=0, column=0)
```

Először az alaposztály előkészítését hajtjuk végre. (Ideális esetben a `super().__init__(parent)` kódot használhatnánk, de kötetünk írásakor ez nem működött, ezért a `super()`-nek kifejezetten át kellett adnunk paraméterként a szülőosztályt és a `self` objektumot.) Ezt követően elteszünk egy másolatot a szülőről későbbi használatra. Abszolút hely és méret megadása helyett elrendezéskezelők segítségével más vezérlőkön belül rendezzük el a vezérlőket. A `grid()` hívás a keret elhelyezéséről gondoskodik a rács-elrendezés-kezelő használatával. Minden látható vezérlőt el kell rendeznünk, még a legfelső szintűeket is. A Tk számos elrendezéskezelőt kínál, de a rács (grid) a legkönnyebben érthető és használható, bár az olyan legfelső szintű elrendzések esetében, ahol csak egy vezérlőt kell elhelyeznünk, a csomagoló (packer) elrendezéskezelőt is használhatnánk; ekkor a `grid(row=0,column=0)` helyett a `pack()` hívást kellene használnunk ugyanennek a hatásnak az eléréséhez.

```
self.principal = tkinter.DoubleVar()
self.principal.set(1000.0)
self.rate = tkinter.DoubleVar()
self.rate.set(5.0)
self.years = tkinter.IntVar()
self.amount = tkinter.StringVar()
```

A Tk lehetővé teszi, hogy vezérlőkhöz társított változókat határozzunk meg. Ha egy ilyen változó értékét a program kódja megváltoztatja, a változás tükröződni fog a hozzá társított vezérlőn, és ehhez hasonlóan az érték felhasználói módosítása a vezérlőben is megváltoztatja a változó értékét. Itt két „double" típusú változót hoztunk létre (a „double" típusú változók lebegőpontos – `float` – értékeket tárolnak), valamint egy egész értékűt és egy karakterlánc értékűt, és kettőnek a kezdőértékét is beállítottuk.

```
principalLabel = tkinter.Label(self, text="Principal $:",
                               anchor=tkinter.W, underline=0)
principalScale = tkinter.Scale(self, variable=self.principal,
        command=self.updateUi, from_=100, to=10000000,
        resolution=100, orient=tkinter.HORIZONTAL)
rateLabel = tkinter.Label(self, text="Rate %:", underline=0,
                          anchor=tkinter.W)
rateScale = tkinter.Scale(self, variable=self.rate,
        command=self.updateUi, from_=1, to=100,
        resolution=0.25, digits=5, orient=tkinter.HORIZONTAL)
yearsLabel = tkinter.Label(self, text="Years:", underline=0,
                           anchor=tkinter.W)
yearsScale = tkinter.Scale(self, variable=self.years,
        command=self.updateUi, from_=1, to=50,
        orient=tkinter.HORIZONTAL)
amountLabel = tkinter.Label(self, text="Amount $",
                            anchor=tkinter.W)
actualAmountLabel = tkinter.Label(self,
        textvariable=self.amount, relief=tkinter.SUNKEN,
        anchor=tkinter.E)
```

Az előkészítő kódnak ez az a része, ahol létrehozzuk a vezérlőket. A `tkinter.Label` vezérlő csak olvasható szöveget jelenít meg a felhasználó számára. Ugyanúgy, mint a többi vezérlőt, ezt is egy szülővel hozzuk létre (ebben az esetben – a szokásos módon – a szülő a tartalmazó vezérlő), a vezérlő megjelenésének és viselkedésének különböző jellemzőit pedig kulcsszó-paraméterekkel állítjuk be. Beállítottuk a `principalLabel` szövegét, az `anchor` tulajdonságának pedig a `tkinter.W` értéket adtuk, aminek a hatására a szöveg-címke szövege balra igazodik (W=west, vagyis „nyugat"). Az `underline` paramétert annak meghatározására használjuk, hogy a címkében melyik karakter legyen az aláhúzott, ami egy *gyorsbillentyűt* jelöl (például ALT+P). Később azt is látni fogjuk, hogy miként bírhatjuk működésre a gyorsbillentyűt. (A gyorsbillentyű olyan billentyűsorozat, amelynek az alakja ALT+BETŰ, ahol a BETŰ egy aláhúzott betűt jelent, amellyel a billentyűfókuszt a gyorsbillentyűhöz kapcsolt vezérlőre állíthatjuk, ami rendszerint a gyorsbillentyűvel rendelkező címke alatti vagy attól jobbra található vezérlő.)

A `tkinter.Scale` vezérlők szülőjeként a szokásos módon a `self` objektumot adjuk meg, és mindegyikhez egy változót társítunk. Ezen kívül parancsként hozzájuk kapcsoljuk egy függvény (illetve ebben az esetben egy tagfüggvény) objektumhivatkozását – ez a tagfüggvény fog automatikusan meghívódni, ha a csúszka (skála) értéke megváltozik –, beállítjuk a skála minimumát (`from_`, a végén aláhúzással, mert a `from` önmagában egy kulcsszó) és maximumát (`to`), valamint vízszintesen tájoljuk a csúszkákat. Egyes csúszkák esetében felbontást (resolution; itt: lépésköz) is megadunk, a `rateScale` számára pedig azt is meghatározzuk, hogy hány számjegyet kell tudnia megjeleníteni.

Az `actualAmountLabel`-hez ugyancsak társítunk egy változót, hogy később egyszerűen módosíthassuk a címke által megjelenített szöveget. Ezt a címkét besüllyesztjük (`relief`), hogy képileg jobban illeszkedjen az ablakra.

```
principalLabel.grid(row=0, column=0, padx=2, pady=2,
                    sticky=tkinter.W)
principalScale.grid(row=0, column=1, padx=2, pady=2,
                    sticky=tkinter.EW)
rateLabel.grid(row=1, column=0, padx=2, pady=2,
               sticky=tkinter.W)
rateScale.grid(row=1, column=1, padx=2, pady=2,
               sticky=tkinter.EW)
yearsLabel.grid(row=2, column=0, padx=2, pady=2,
                sticky=tkinter.W)
yearsScale.grid(row=2, column=1, padx=2, pady=2,
                sticky=tkinter.EW)
amountLabel.grid(row=3, column=0, padx=2, pady=2,
                 sticky=tkinter.W)
actualAmountLabel.grid(row=3, column=1, padx=2, pady=2,
                       sticky=tkinter.EW)
```

Miután létrehoztuk a vezérlőket, el kell rendeznünk őket. A használt rácselrendezést a 13.3. ábra mutatja.

principalLabel	principalScale
rateLabel	rateScale
yearsLabel	yearsScale
amountLabel	actualAmountLabel

13.3. ábra

Az Interest program elrendezése

Minden vezérlő ismeri a `grid()` tagfüggvényt (valamint más elrendezőfüggvényeket, például a `pack()`-et is). A `grid()` meghívásával a vezérlőt a szülőn belül rendezzük el, a megadott számú sorban és oszlopban. További kulcsszó-paraméterekkel (`rowspan` és `columnspan`) úgy is beállíthatjuk a vezérlőket, hogy több sort, illetve oszlopot fogjanak át, a `padx` és `pady` kulcsszó-paraméterekben egész képpontértékeket meghatározva pedig margót adhatunk köréjük (a `padx` a bal és jobb margót, a `pady` a felső és alsó margót állítja be). Ha egy vezérlőnek több helyet foglalunk, mint amennyire szüksége van, a `sticky` („ragadós") paraméterrel határozhatjuk meg, hogy mi történjen a fölös hellyel – ha nem adjuk meg, a vezérlő a számára lefoglalt terület közepére kerül. Az első oszlop minden vezérlőjét `sticky`-re állítottuk, mégpedig `tkinter.EW` (east-west, „kelet-nyugat") értékkel, aminek a hatására megnyúlnak, hogy a teljes rendelkezésükre álló szélességet kitöltsék.

Minden vezérlőt helyi változókban tárolunk, de szemétgyűjtésre nem kell kijelölnünk őket, mert a szülő–gyermek kapcsolatok gondoskodnak róla, hogy ne törlődjenek, amikor kikerülnek a hatókörből az előkészítő kód végén, hiszen mindegyiküknek a főablak a szülője. A vezérlőket néha példányváltozóként hozzuk létre, például akkor, ha az előkészítő függvényen kívül kell hivatkoznunk rájuk, de itt a vezérlőkhöz társított változókat (`self.principal`, `self.rate` és `self.years`) tettük példányváltozókká, mert ezeket fogjuk az előkészítő függvényen kívül használni.

```
principalScale.focus_set()
self.updateUi()
parent.bind("<Alt-p>", lambda *ignore: principalScale.focus_set())
parent.bind("<Alt-r>", lambda *ignore: rateScale.focus_set())
parent.bind("<Alt-y>", lambda *ignore: yearsScale.focus_set())
parent.bind("<Control-q>", self.quit)
parent.bind("<Escape>", self.quit)
```

Az előkészítő függvény végén a billentyűfókuszt a `principalScale` vezérlőre állítjuk, hogy amint a program elindult, a felhasználó beállíthassa a nyitó pénzösszeget. Ez után meghívjuk a `self.updateUi()` tagfüggvényt, hogy kiszámítsuk a kezdő végösszeget.

A következő lépés néhány billentyűkötés beállítása. (Sajnos a *kötés* szónak három különböző jelentése van. Az első a változókötés, amikor egy nevet, vagyis egy objektumhivatkozást kapcsolunk egy objektumhoz; a második a billentyűkötés, amikor egy billentyűműveletet, például egy billentyű lenyomását vagy felengedését társítjuk egy függvényhez vagy tagfügg-

vényhez, amelyet akkor kell meghívni, amikor a műveletet végrehajtják; a harmadik pedig a könyvtárkötés, ami azt a ragasztókódot jelenti, amely egy a Pythontól eltérő nyelven írt könyvtárat Python-modulokon keresztül elérhetővé tesz a Python-programozók számára.) A billentyűkötések létfontosságúak a fogyatékkal élők számára, akik nehezen vagy egyáltalán nem tudják használni az egeret, és azoknak a munkáját is sokkal kényelmesebbé teszik, akik nem akarják igénybe venni az egeret, mert gyors gépelők lévén az egér használata csak lelassítaná őket.

Az első három billentyűkötéssel a billentyűfókuszt állítjuk egy csúszkára. A `principalLabel` feliratként például a `Principal $:` szöveget állítottuk be, az aláhúzást pedig a 0. karakterre, a címke tehát `Principal $:` formában jelenik meg. Az első billentyűkötés beállításával a billentyűfókusz az ALT+P billentyűk lenyomásának hatására a `principalScale` csúszkára kerül. Ugyanígy működik a másik két billentyűkötés is. Megjegyzendő, hogy a `focus_set()` tagfüggvényt nem közvetlenül kötjük a vezérlőkhöz, mert amikor egy eseménykötés eredményeként függvények vagy tagfüggvények meghívására kerül sor, első paraméterként megkapják az őket hívó eseményt, nekünk viszont nincs szükségünk erre az eseményre. Ezért egy lambdafüggvényt használunk, amely elfogja az eseményt, és meghívja a tagfüggvényt, de a nemkívánatos paraméter nélkül.

Két *billentyűparancsot* is meghatároztunk – ezek olyan billentyűkombinációk, amelyek egy bizonyos műveletet indítanak el. Itt a CTRL+Q, illetve az Esc billentyűparancsokat adtuk meg, és mindkettőt a `self.quit()` tagfüggvényhez kötöttük, amely tisztán befejezi a program futtatását.

Billentyűkötéseket az egyes vezérlők számára is beállíthatunk, de itt mindet a szülőhöz (az alkalmazáshoz) adtuk meg, hogy attól függetlenül működjenek, hogy hol van éppen a billentyűfókusz.

A Tk `bind()` tagfüggvényét egérkattintások és billentyűleütések, sőt a programozó által meghatározott események kötésére is használhatjuk. A CTRL-hez és az Esc-hez hasonló különleges billentyűknek a Tk-ben adott nevük van (`Control` és `Escape`), míg a közönséges betűk önmagukat jelölik. Billentyűsorozatot úgy adhatunk meg, hogy a részeit szögletes zárójelek közé írjuk, és kötőjelekkel választjuk el őket.

Most, hogy létrehoztuk és elrendeztük a vezérlőket, és beállítottuk a billentyűkötéseket, a program megjelenése és alapműködése a helyére került. A program megvalósításának teljessé tételéhez a következőkben azokat a tagfüggvényeket tekintjük át, amelyek a felhasználói műveletekre reagálnak.

```
def updateUi(self, *ignore):
    amount = self.principal.get() * (
             (1 + (self.rate.get() / 100.0)) ** self.years.get())
    self.amount.set("{0:.2f}".format(amount))
```

Ez a tagfüggvény minden alkalommal meghívódik, amikor a felhasználó megváltoztatja a tőke összegét, a kamatlábat vagy a futamidőt, mivel mindegyik csúszkához ez a parancs társul. A függvény csupán annyit tesz, hogy kiolvassa a csúszkához tartozó változó értékét, végrehajtja a kamatos kamat kiszámítását, és az eredményt (karakterláncként) a végösszeget kiíró címkéhez társított változóba írja. Ennek eredményeképpen a végösszeget kiíró címke mindig naprakész összeget mutat.

```
def quit(self, event=None):
    self.parent.destroy()
```

Ha a felhasználó úgy dönt, hogy kilép a programból (a CTRL+Q vagy az ESC lenyomásával, vagy az ablak bezáró gombjára kattintva), ez a tagfüggvény hívódik meg. Mivel nincsenek mentendő adatok, csak utasítjuk a szülőt (az alkalmazás objektumát), hogy semmisítse meg önmagát. A gyermekek törléséről a szülő gondoskodik – ez az összes ablakot jelenti, amelyek pedig az összes vezérlőjüket pusztítják el –, így tisztán kiléphetünk a programból.

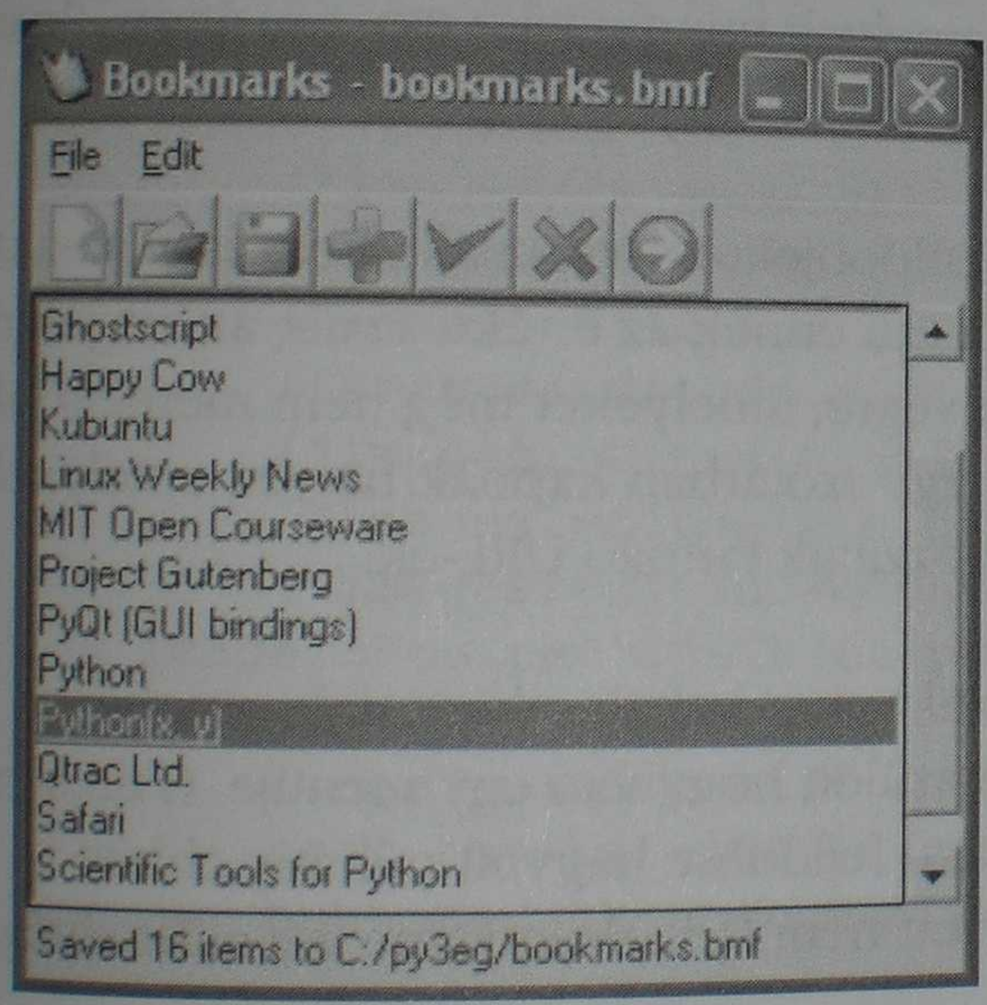
```
application = tkinter.Tk()
path = os.path.join(os.path.dirname(__file__), "images/")
if sys.platform.startswith("win"):
    icon = path + "interest.ico"
else:
    icon = "@" + path + "interest.xbm"
application.iconbitmap(icon)
application.title("Interest")
window = MainWindow(application)
application.protocol("WM_DELETE_WINDOW", window.quit)
application.mainloop()
```

Ha meghatároztuk a főablak (ami ebben az esetben az egyetlen ablak) osztályát, már rendelkezünk azzal a kóddal, ami elindítja a programot. Ez után először létrehozzuk az alkalmazás egészét jelképező objektumot. Windowson ahhoz, hogy egy programhoz ikont rendeljünk, egy `.ico` fájlra van szükségünk, amelynek a fájlnevét (teljes elérési úttal) átadjuk az `iconbitmap()` tagfüggvénynek. Unix rendszereken azonban egy bitképet (pontosabban egy monokróm képet) kell biztosítanunk. A Tk számos beépített bitképet kínál, amelyeket a fájlrendszerből származó képektől a nevük elé írt `@` jellel különböztethetünk meg. Ezt követően nevet (címet) adunk az alkalmazásnak (ez fog megjelenni az ablak címsorában), majd létrehozzuk a `MainWindow` osztályunk egy példányát, a szülőjeként az alkalmazásobjektumot megadva. Végül meghívjuk a `protocol()` tagfüggvényt, hogy megadjuk, hogy mi történjen, ha a felhasználó a bezáró gombra kattint – mi azt mondtuk, hogy ilyenkor a `MainWindow.quit()` tagfüggvényt kell meghívni. Végül elindítjuk az eseményhurkot – az ablak csak ekkor jelenik meg, és innentől fogva képes válaszolni a felhasználói műveletekre.

Főablakos programok

Bár a párbeszédablakos programok gyakran elegendőek az egyszerű feladatokhoz, ha egy program képességeinek köre nő, általában célszerű teljes, főablakos alkalmazást készíteni belőle, menükkel és eszköztárakkal. Az ilyen alkalmazásokat általában könnyebb bővíteni, mint a párbeszédablakos programokat, mivel anélkül adhatunk hozzájuk további menüket, menüpontokat vagy eszköztárgombokat, hogy ez befolyásolná a főablak elrendezését.

Ebben a részben a 13.4. ábrán látható `bookmarks-tk.pyw` programot vizsgáljuk meg. A program könyvjelzőket tárol (név, URL) formájú karakterláncok alakjában, és lehetőséget nyújt a felhasználónak a könyvjelzők felvételére, szerkesztésére és eltávolítására, valamint egy adott könyvjelző által hivatkozott weboldal megnyitására a webböngészőben.



13.4. ábra

A Bookmarks program

A programnak két ablaka van: a főablak a menüsorral, az eszköztárral, a könyvjelzők listájával és az állapotsorral, valamint egy párbeszédablak, amely a könyvjelzők felvételére és szerkesztésére szolgál.

Főablak létrehozása

A főablak annyiban hasonlít egy párbeszédablakhoz, hogy vezérlői vannak, amelyeket létre kell hozni és el kell rendezni. Ezen kívül hozzá kell adnunk a menüsort, a menüket, az eszköztárat és az állapotsort, valamint tagfüggvényeket, amelyek végrehajtják a felhasználó által kért műveleteket. A felhasználói felületet teljes egészében a főablak előkészítő függvényében alakítjuk ki, amelyet öt részletben tekintünk át, mert elég hosszú.

```
class MainWindow:
    def __init__(self, parent):
        self.parent = parent
        self.filename = None
        self.dirty = False
        self.data = {}
        menubar = tkinter.Menu(self.parent)
        self.parent["menu"] = menubar
```

Ehhez az ablakhoz – ellentétben az előző példával, ahol egy meglevő vezérlőtől örököltünk – egyszerűen egy szokványos Python-osztályt hozunk létre. Ha öröklést alkalmazunk, új megvalósítást adhatunk a szülőosztálytól örökölt tagfüggvényeknek, de ha erre nincs szükségünk, használhatunk egyszerűen összetételt, ahogy itt is tettük. Az ablak megjelenését vezérlő-példányváltozók létrehozásával alakítjuk ki, amelyek mindegyike egy `tkinter.Frame` objektumban kap helyet, ahogy hamarosan látni fogjuk.

Négyféle információt kell nyomon követnünk: a szülőobjektumot (az alkalmazást), az aktuális könyvjelzőfájl nevét, a „piszkos" (`dirty`) jelzőt (ha ennek az értéke `True`, az azt jelenti, hogy az adatokon olyan módosításokat hajtottunk végre, amelyeket még nem mentettünk lemezre), valamint magukat az adatokat, amelyek egy szótárban kapnak helyet, amelynek a kulcsai a könyvjelzők nevei, az értékei pedig a hozzájuk tartozó URL-ek.

A menüsor létrehozásához egy `tkinter.Menu` objektumot kell létrehoznunk, amelynek a szülője az ablak szülője, és tájékoztatnunk kell a szülőt, hogy van egy menüje. (Furcsának tűnhet, hogy a menüsor is egy menü, de a Tk hosszú fejlődése hagyott néhány ehhez hasonló furcsaságot.) Az így létrehozott menüsorokat nem kell elrendezni; ezt a Tk elvégzi helyettünk.

```
        fileMenu = tkinter.Menu(menubar)
        for label, command, shortcut_text, shortcut in (
                ("New...", self.fileNew, "Ctrl+N", "<Control-n>"),
                ("Open...", self.fileOpen, "Ctrl+O", "<Control-o>"),
                ("Save", self.fileSave, "Ctrl+S", "<Control-s>"),
                (None, None, None, None),
                ("Quit", self.fileQuit, "Ctrl+Q", "<Control-q>")):
            if label is None:
                fileMenu.add_separator()
            else:
                fileMenu.add_command(label=label, underline=0,
                        command=command, accelerator=shortcut_text)
                self.parent.bind(shortcut, command)
        menubar.add_cascade(label="File", menu=fileMenu, underline=0)
```

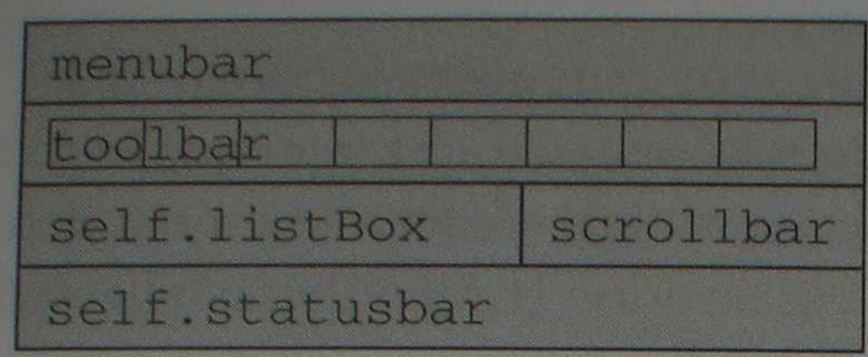
A menüsor minden menüjét ugyanúgy kell létrehozni. Először létrehozunk egy `tkinter.Menu` objektumot, amely a menüsor gyermeke, majd parancsokat (menüpontokat) és elválasztókat adunk a menühöz. (Megjegyzendő, hogy az „accelerator" – „gyorsító" –

a Tk nyelvezetében billentyűparancsot jelent, az `accelerator` paraméter azonban csak a billentyűparancs-hivatkozás szövegét állítja be, tényleges billentyűkötést nem hoz létre.) Az `underline` paraméter az aláhúzott karakter megadására szolgál – esetünkben minden menüpontnak az első betűjét húzzuk alá, és ez a betű lesz az adott parancs gyorsbillentyűje. A menüpontok (parancsok, command) hozzáadása mellett billentyűparancsokat is megadunk, mégpedig úgy, hogy egy billentyűsorozatot kötünk ahhoz a parancshoz, amelyik a kiválasztott menüponthoz tartozik. Végül a menüt a menüsorhoz adjuk az `add_cascade()` tagfüggvénnyel.

Az Edit (Szerkesztés) menüt kihagytuk, mert a szerkezetét tekintve megegyezik a File (Fájl) menü kódjával.

```
frame = tkinter.Frame(self.parent)
self.toolbar_images = []
toolbar = tkinter.Frame(frame)
for image, command in (
        ("images/filenew.gif", self.fileNew),
        ("images/fileopen.gif", self.fileOpen),
        ("images/filesave.gif", self.fileSave),
        ("images/editadd.gif", self.editAdd),
        ("images/editedit.gif", self.editEdit),
        ("images/editdelete.gif", self.editDelete),
        ("images/editshowwebpage.gif", self.editShowWebPage)):
    image = os.path.join(os.path.dirname(__file__), image)
    try:
        image = tkinter.PhotoImage(file=image)
        self.toolbar_images.append(image)
        button = tkinter.Button(toolbar, image=image,
                                command=command)
        button.grid(row=0, column=len(self.toolbar_images) -1)
    except tkinter.TclError as err:
        print(err)
toolbar.grid(row=0, column=0, columnspan=2, sticky=tkinter.NW)
```

Ez után első lépésként létrehozunk egy keretet, amely az ablak összes vezérlőjét tartalmazni fogja, majd egy másikat `toolbar` néven, amelyben olyan gombok vízszintes sora kap helyet, amelyeken szöveg helyett képek láthatók – ezek lesznek az eszköztár gombjai. Az eszköztárgombokat egymás után egy rácsban rendezzük el, amelynek egyetlen sora van, és annyi oszlopa, ahány gombbal rendelkezünk. Végül magának az eszköztárnak a keretét a főablak keretének első sorává tesszük, „északnyugatra" ragasztva, hogy mindig az ablak bal felső sarkához igazodjon. (A Tk automatikusan az ablakban található többi vezérlő felett helyezi el a menüsort.) Az elrendezést a 13.5. ábra szemlélteti, amelyen a Tk által elhelyezett menüsort fehér, míg az általunk elrendezett vezérlőket szürke háttérrel jeleztük.



13.5. ábra

A Bookmarks program főablakának elrendezése

Amikor egy képet adunk egy gombhoz, gyenge hivatkozás jön létre, ami azt jelenti, hogy ha a kép kikerül a hatókörből, akkor kijelölődik szemétgyűjtésre. Ezt el kell kerülnünk, mert azt szeretnénk, ha a gombokon megjelennének a képek, amikor az előkészítő befejezte a munkáját, ezért létrehozunk egy `self.toolbar_images` nevű példányváltozót, amelynek csak annyi a feladata, hogy tárolja a képekre mutató hivatkozásokat, hogy életben tartsa azokat a program teljes élete során.

Alapállapotában a Tk csupán néhány képfájlformátumot képes olvasni, ezért `.gif`* képeket kellett használnunk. Ha valamelyik kép nem található, `tkinter.Exception` kivétel keletkezik, ezért ügyelnünk kell rá, hogy elfogjuk ezt a kivételt, nehogy a program kilépjen, csak azért, mert nincs meg egy kép.

Figyeljük meg, hogy a menükben elérhető műveletek közül nem mindet tettük eszköztárgombként is hozzáférhetővé, ami elterjedt gyakorlat.

```
scrollbar = tkinter.Scrollbar(frame, orient=tkinter.VERTICAL)
self.listBox = tkinter.Listbox(frame, yscrollcommand=scrollbar.set)
self.listBox.grid(row=1, column=0, sticky=tkinter.NSEW)
self.listBox.focus_set()
scrollbar["command"] = self.listBox.yview
scrollbar.grid(row=1, column=1, sticky=tkinter.NS)

self.statusbar = tkinter.Label(frame, text="Ready...",
                               anchor=tkinter.W)
self.statusbar.after(5000, self.clearStatusBar)
self.statusbar.grid(row=2, column=0, columnspan=2, sticky=tkinter.EW)

frame.grid(row=0, column=0, sticky=tkinter.NSEW)
```

A főablak központi részét (az eszköztár és az állapotsor közötti területet) egy listamező és egy hozzá tartozó gördítősáv foglalja el. A listamezőt minden irányban ragadóssá tettük, a gördítősávot azonban csak északra és délre (vagyis függőlegesen). Mindkét vezérlőt az ablakkeret rácsához adtuk, egymás mellett.

---

* Ha telepítettük a Python Imaging Library Tk bővítményét, minden mai képformátum támogatottá válik. A részleteket lásd a `www.pythonware.com/products/pil/` címen.

Gondoskodnunk kell róla, hogy ha a felhasználó a LE és FEL nyílbillentyűkkel mozog a listában, vagy a gördítősávval görgeti a listát, a két vezérlő összhangban maradjon. Ezt úgy érjük el, hogy a listamező `yscrollcommand` tulajdonságát a gördítősáv `set()` tagfüggvényére állítjuk (így a listamezőben való mozgás a gördítősávot is elmozdítja, amennyiben szükséges), a gördítősáv `command` tulajdonságát pedig a listamező `yview()` tagfüggvényére (ennek eredményeképpen a gördítősáv mozgatása a lista megfelelő görgetését is maga után vonja).

Az állapotsor csupán egy szövegcímke. Az `after()` tagfüggvény egy egyszeri időzítő (vagyis egy olyan időzítő, amely egyszer jár le a megadott időtartam után), amelynek az első paramétere a lejárati időtartam ezredmásodpercben, a második paramétere pedig egy függvény vagy tagfüggvény, amelyet akkor kell meghívni, amikor az időtartam lejárt. Ez azt jelenti, hogy amikor a program elindul, az állapotsorban öt másodpercig a „Ready…" („Kész…") szöveg lesz látható, majd az állapotsor tartalma törlődik. Az állapotsort az ablak utolsó soraként helyezzük el, és nyugat–kelet irányban (vízszintesen) tesszük ragadóssá.

Végül elrendezzük magát az ablakkeretet. Már végeztünk a létrehozással és a főablak vezérlőinek elrendezésével, de jelen állapotban a vezérlők rögzített alapértelmezett méretet tételeznek fel, ezért ha az ablakot átméretezzük, a vezérlők nem nyúlnak meg vagy zsugorodnak össze, hogy igazodjanak a méretváltozáshoz. A következő kódrészlet megoldja ezt a problémát, és teljessé teszi az előkészítő függvényt:

```
frame.columnconfigure(0, weight=999)
frame.columnconfigure(1, weight=1)
frame.rowconfigure(0, weight=1)
frame.rowconfigure(1, weight=999)
frame.rowconfigure(2, weight=1)

window = self.parent.winfo_toplevel()
window.columnconfigure(0, weight=1)
window.rowconfigure(0, weight=1)

self.parent.geometry("{0}x{1}+{2}+{3}".format(400, 500, 0, 50))
self.parent.title("Bookmarks - Unnamed")
```

A `columnconfigure()` és `rowconfigure()` tagfüggvények lehetővé teszik a rács súlyozását. Az ablakkerettel kezdjük, ahol az összes „súlyt" az első oszlopnak és (a listamező által elfoglalt) második sornak adjuk, így ha a keretet átméretezzük, minden új területet a listamező fog megkapni. Önmagában azonban ez nem elég – a keretet tartalmazó legfelső szintű ablakot átméretezhetővé is kell tennünk. Ezt úgy érjük el, hogy a `wininfo_toplevel()` tagfüggvénnyel kinyerünk egy hivatkozást az ablakra, majd az ablakot úgy tesszük átméretezhetővé, hogy a sor- és oszlopsúlyozását 1-re állítjuk.

Az előkészítő függvény végén egy *`szélességxmagasság+x+y`* alakú karakterlánc segítségével beállítunk egy kezdeti ablakméretet és -helyzetet. (Ha csak a méretet szeretnénk meghatározni, a *`szélességxmagasság`* alakot használhatjuk helyette.) Végül megadjuk az ablak címét, amivel az ablak felhasználói felülete teljessé válik.

Ha a felhasználó egy eszköztárgombra kattint, vagy kiválaszt egy menüpontot, egy tagfüggvény hívódik meg, amely végrehajtja a kért műveletet, ezeknek a tagfüggvényeknek egy része pedig segédtagfüggvényekre támaszkodik. A következőkben valamennyi tagfüggvényt áttekintjük, azzal kezdve, amelyiknek a meghívására öt másodperccel a program elindulása után kerül sor:

```
def clearStatusBar(self):
    self.statusbar["text"] = ""
```

Az állapotsor egy egyszerű `tkinter.Label` objektum. A tartalmát törölhettük volna egy lambdakifejezéssel is az `after()` tagfüggvényhívásban, de mivel az állapotsor kiürítését több helyről is tudnunk kell elvégezni, külön tagfüggvényt hoztunk létre erre a célra:

```
def fileNew(self, *ignore):
    if not self.okayToContinue():
        return
    self.listBox.delete(0, tkinter.END)
    self.dirty = False
    self.filename = None
    self.data = {}
    self.parent.title("Bookmarks - Unnamed")
```

Ha a felhasználó új könyvjelzőfájlt akar létrehozni, először lehetőséget kell adnunk neki, hogy mentse a még nem mentett módosításokat a meglevő fájlba (ha van olyan). Ezt a lehetőséget a `MainWindow.okayToContinue()` tagfüggvényben teremtjük meg, mivel több különböző helyen is használni fogjuk. A tagfüggvény `True` értéket ad vissza, ha továbbléphetünk, más esetben pedig `False`-t. Ha folytathatjuk a műveletet, akkor kiürítjük a listamezőt az elsőtől az utolsóig az összes bejegyzés törlésével. A `tkinter.END` egy állandó, amellyel az utolsó elemet jelölhetjük egy olyan környezetben, ahol a vezérlő több elemet is tartalmazhat. Ezt követően töröljük a „piszkos" jelzőt, a fájlnevet és az adatokat, mivel a fájl új és módosítatlan, az ablak címét pedig átállítjuk, hogy tükrözze, hogy egy új, még mentetlen fájllal dolgozunk.

Az `ignore` („figyelmen kívül hagy") változó egy nulla vagy több helyparaméterből álló sorozatot tárol, amellyel nem törődünk. A menüpontok kiválasztásával vagy eszköztárgombok lenyomásával meghívott tagfüggvények esetében nincsenek figyelmen kívül hagyott paraméterek, de ha a felhasználó egy billentyűparancsot adott ki (például a CTRL+N-t), akkor a hívó esemény is átadódik, de mivel minket nem érdekel, hogy a felhasználó hogyan hívta meg a műveletet, figyelmen kívül hagyjuk a kérelmező eseményt.

```
def okayToContinue(self):
    if not self.dirty:
        return True
    reply = tkinter.messagebox.askyesnocancel(
                    "Bookmarks - Unsaved Changes",
                    "Save unsaved changes?", parent=self.parent)
```

```
    if reply is None:
        return False
    if reply:
        return self.fileSave()
    return True
```

Ha a felhasználó olyan műveletet szeretne végrehajtani, amely törli a listamező tartalmát (például létre szeretne hozni vagy meg szeretne nyitni egy új fájlt), lehetőséget kell adnunk neki, hogy mentse a még nem mentett módosításokat. Amennyiben a fájl nem „piszkos", nincsenek mentendő módosítások, ezért azonnal `True`-t adunk vissza, egyéb esetben pedig megjelenítünk egy szabványos üzenetablakot, amelyen `Yes` (Igen), `No` (Nem) és `Cancel` (Mégse) gombok találhatók. Ha a felhasználó visszavonja a parancsot (Mégse), a válasz (`reply`) `None` lesz; ezt úgy vesszük, hogy a felhasználó nem kívánja folytatni a megkezdett műveletet, és nem akar menteni sem, ezért egyszerűen `False` értéket adunk vissza. Amennyiben a felhasználó igennel válaszol, a válasz `True` lesz, így lehetőséget adunk a mentésre, és `True`-t adunk vissza, ha a mentés megtörtént, és `False`-t, ha nem. Ha pedig a felhasználó a Nem gombot nyomja meg, a válasz `False` lesz, ami arra utasítja a programot, hogy ne mentsen, de ettől függetlenül `True`-t adunk vissza, mert a felhasználó folytatni szeretné a megkezdett műveletet, csak éppen el kívánja vetni a még nem mentett módosításokat.

A Tk szabványos párbeszédablakait az `import tkinter` utasítás nem tölti be, ezért ezen a beemelésen felül ki kell adnunk az `import tkinter.messagebox`, illetve a következő tagfüggvényhez az `import tkinter.filedialog` utasítást is. Windowson és Mac OS X-en a rendszer beépített párbeszédablakainak a használatára kerül sor, míg más rendszereken a Tk-ra jellemző párbeszédablakok jelennek meg. A szabványos párbeszédablakoknak mindig átadjuk a szülőjüket, mert ez biztosítja, hogy megjelenésükkor automatikusan a szülőablak közepére kerüljenek.

Minden szabványos párbeszédablak *modális ablak*, ami azt jelenti, hogy ha megjelentek, a felhasználó csak ezen az egy ablakon végezhet műveleteket a programban, tehát ha hozzá szeretne férni a program többi részéhez, akkor előbb be kell zárnia a párbeszédablakot (az `OK`, `Open`, `Cancel` és más hasonló gombok egyikére kattintva). A programozók számára a modális párbeszédablakok használata a legkönnyebb, mert a felhasználó nem változtathatja meg mögöttük a program állapotát, és blokkolják a programot (maguknál tartják a vezérlését) egészen addig, amíg be nem zárják őket. A blokkolás azt jelenti, hogy amikor létrehozunk vagy meghívunk egy modális párbeszédablakot, az utána következő utasítás csak akkor hajtódik végre, amikor a párbeszédablakot bezárják.

```
def fileSave(self, *ignore):
    if self.filename is None:
        filename = tkinter.filedialog.asksaveasfilename(
                        title="Bookmarks - Save File",
                        initialdir=".",
                        filetypes=[("Bookmarks files", "*.bmf")],
```

```
                        defaultextension=".bmf",
                        parent=self.parent)
        if not filename:
            return False
        self.filename = filename
        if not self.filename.endswith(".bmf"):
            self.filename += ".bmf"
    try:
        with open(self.filename, "wb") as fh:
            pickle.dump(self.data, fh, pickle.HIGHEST_PROTOCOL)
        self.dirty = False
        self.setStatusBar("Saved {0} items to {1}".format(
                          len(self.data), self.filename))
        self.parent.title("Bookmarks - {0}".format(
                          os.path.basename(self.filename)))
    except (EnvironmentError, pickle.PickleError) as err:
        tkinter.messagebox.showwarning("Bookmarks - Error",
                "Failed to save {0}:\n{1}".format(
                self.filename, err), parent=self.parent)
    return True
```

Ha nincs fájl megnyitva, meg kell kérnünk a felhasználót, hogy válasszon egy fájlnevet. Ha a felhasználó a Mégse gombra kattint, `False`-t adunk vissza, hogy jelezzük, hogy a teljes műveletet vissza kell vonni, egyébként pedig meggyőződünk róla, hogy a megadott fájlnév kiterjesztése megfelelő-e. Ha igen, a meglevő vagy új fájlnév használatával a fájlba mentjük a befőzött `self.data` szótárt. A könyvjelzők mentését követően töröljük a „piszkos" jelzőt, hiszen most már nincsenek nem mentett módosítások, kiírunk egy üzenetet az állapotsorban (amely csak egy ideig látható, ahogy hamarosan látni fogjuk), és frissítjük az ablak címsorát, hogy a fájlnév is megjelenjen benne (az elérési út nélkül). Ha a fájl mentése nem sikerül, egy figyelmeztető üzenetablakot jelenítünk meg (ezen automatikusan helyet kap egy `OK` gomb), hogy tájékoztassuk a felhasználót.

```
def setStatusBar(self, text, timeout=5000):
    self.statusbar["text"] = text
    if timeout:
        self.statusbar.after(timeout, self.clearStatusBar)
```

A fenti tagfüggvény állítja be az állapotsor szövegét, és ha megadtunk egy lejárati időt (az alapértelmezés öt másodperc), a tagfüggvény beállít egy egyszeri időzítőt, amely törli az állapotsor tartalmát a lejárati idő letelte után.

```
def fileOpen(self, *ignore):
    if not self.okayToContinue():
        return
    dir = (os.path.dirname(self.filename)
           if self.filename is not None else ".")
    filename = tkinter.filedialog.askopenfilename(
               title="Bookmarks - Open File",
```

```
                initialdir=dir,
                filetypes=[("Bookmarks files", "*.bmf")],
                defaultextension=".bmf", parent=self.parent)
    if filename:
        self.loadFile(filename)
```

Ez a tagfüggvény ugyanúgy kezdődik, mint a `MainWindow.fileNew()` – lehetőséget ad a felhasználónak, hogy mentse a még nem mentett módosításokat, vagy visszavonja a fájlmegnyitási műveletet. Ha a felhasználó úgy dönt, hogy folytatja, egy ésszerű kezdőkönyvtárba kell irányítanunk, ezért az aktuális fájl (ha van ilyen) könyvtárát használjuk, illetve egyéb esetben a jelenlegi munkakönyvtárat. A `filetypes` paraméter (leírás, helyettesítő karakter) formájú rendezett párok listája, amelyet a fájlmegnyitási párbeszédablaknak meg kell jelenítenie. Ha a felhasználó kiválasztott egy fájlnevet, az aktuális fájlnevet a választottra állítjuk, és meghívjuk a `loadFile()` tagfüggvényt, amely végrehajtja a fájl tényleges beolvasását.

A `loadFile()` tagfüggvényt általában el szokták különíteni, hogy könnyebb legyen anélkül is betölteni egy fájlt, hogy a felhasználótól kérnénk egy fájlnevet. Egyes programok például induláskor betöltik az utoljára megnyitott fájlt, mások pedig a legutóbb megnyitott fájlokat felsorolják egy menüben, így ha a felhasználó kiválaszt egyet közülük, a `loadFile()` tagfüggvényt közvetlenül meg lehet hívni az adott menüponthoz társított fájlnévvel.

```
def loadFile(self, filename):
    self.filename = filename
    self.listBox.delete(0, tkinter.END)
    self.dirty = False
    try:
        with open(self.filename, "rb") as fh:
            self.data = pickle.load(fh)
        for name in sorted(self.data, key=str.lower):
            self.listBox.insert(tkinter.END, name)
        self.setStatusBar("Loaded {0} bookmarks from {1}".format(
                          self.listBox.size(), self.filename))
        self.parent.title("Bookmarks - {0}".format(
                          os.path.basename(self.filename)))
    except (EnvironmentError, pickle.PickleError) as err:
        tkinter.messagebox.showwarning("Bookmarks - Error",
                "Failed to load {0}:\n{1}".format(
                self.filename, err), parent=self.parent)
```

Ha a fenti tagfüggvény meghívódik, tudhatjuk, hogy minden módosítást mentettek vagy elvetettek, tehát nyugodtan törölhetjük a listamező tartalmát. Az aktuális fájlnevet az átadott fájlnévre állítjuk, kiürítjük a listát, töröljük a „piszkos" jelzőt, majd megkíséreljük megnyitni a fájlt és a befőttet kibontani a `self.data` szótárba. Ha megvannak az adatok, végigmegyünk a könyvjelzők nevein, és egymás után a listához fűzzük azokat. Végül tájékoztató

üzenetet jelenítünk meg az állapotsorban, és frissítjük az ablak címsorát. Ha a fájlt nem sikerült beolvasni, vagy a befőttet nem tudtuk kibontani, egy üzenetablakban figyelmeztetést jelenítünk meg, hogy értesítsük a felhasználót.

```
def fileQuit(self, event=None):
    if self.okayToContinue():
        self.parent.destroy()
```

Ez a File menü utolsó menüpontjának tagfüggvénye. Lehetőséget adunk a felhasználónak, hogy mentse a még nem mentett módosításokat – ha a `Cancel` gombra kattint, nem csinálunk semmit, és a program folytatja a futását, egyébként azonban utasítjuk a szülőt, hogy semmisítse meg önmagát, ami a program tiszta befejezéséhez vezet. Ha a felhasználó beállításait is menteni szeretnénk, azt is itt tennénk meg, közvetlenül a `destroy()` hívás előtt.

```
def editAdd(self, *ignore):
    form = AddEditForm(self.parent)
    if form.accepted and form.name:
        self.data[form.name] = form.url
        self.listBox.delete(0, tkinter.END)
        for name in sorted(self.data, key=str.lower):
            self.listBox.insert(tkinter.END, name)
        self.dirty = True
```

Ezt a tagfüggvényt akkor hívjuk meg, amikor a felhasználó új könyvjelzőt szeretne felvenni (az Edit, Add menüpontra vagy az eszköztár ➕ gombjára kattintva, esetleg a CTRL+A billentyűparancsot kiadva). Az `AddEditForm` egy egyéni párbeszédablak, amelyet a következő részben tárgyalunk – a használatához csak annyit kell tudnunk róla, hogy van egy `accepted` nevű jelzője, amelyet `True`-ra állítunk, ha a felhasználó az `OK` gombra kattintott, és `False`-ra, ha a `Cancel` gombot választotta, valamint két jellemzője, a `name` és az `url`, amelyek a felhasználó által felvett vagy szerkesztett könyvjelző nevét és URL-jét tárolják.

Létrehozunk egy új `AddEditForm`-példányt, amely azonnal megjelenik, mégpedig modális párbeszédablakként, vagyis blokkolja a programot – tehát az `if form.accepted...` utasítás a párbeszédablak bezárásáig nem hajtódik végre. Ha a felhasználó az `OK` gombra kattint az `AddEditForm` párbeszédablakban, és nevet ad a könyvjelzőnek, az új könyvjelző nevét és URL-jét hozzáaadjuk a `self.data` szótárhoz. Ez után töröljük a listamező tartalmát, és újra beszúrjuk az adatokat a megfelelő sorrendben. Persze hatékonyabb lenne egyszerűen az új könyvjelzőt beszúrni a megfelelő helyre, de egy mai számítógépen a különbség még könyvjelzők százainak esetén is alig lenne észrevehető. Végül beállítjuk a `dirty` jelzőt, mivel most van egy nem mentett módosításunk.

```
def editEdit(self, *ignore):
    indexes = self.listBox.curselection()
    if not indexes or len(indexes) > 1:
        return
    index = indexes[0]
    name = self.listBox.get(index)
```

```
form = AddEditForm(self.parent, name, self.data[name])
if form.accepted and form.name:
    self.data[form.name] = form.url
    if form.name != name:
        del self.data[name]
        self.listBox.delete(0, tkinter.END)
        for name in sorted(self.data, key=str.lower):
            self.listBox.insert(tkinter.END, name)
    self.dirty = True
```

A könyvjelzők szerkesztése valamivel bonyolultabb, mint a hozzáadásuk, mert először meg kell keresnünk azt a könyvjelzőt, amelyet a felhasználó szerkeszteni szeretne. A `curselection()` tagfüggvény egy (valószínűleg üres) listát ad vissza a kijelölt elemek sorszámaival. Ha a felhasználó egyetlen elemet jelölt ki, akkor kiolvassuk a szövegét, mivel ez a neve annak a könyvjelzőnek, amelyet a felhasználó módosítani szeretne (és egyben a `self.data` szótár kulcsa). Ezt követően létrehozunk egy új `AddEditForm`-ot, a szekeszteni kívánt könyvjelző nevét és URL-jét átadva.

Miután az ablakot bezárták, frissítjük a `self.data` szótárt – amennyiben a felhasználó az `OK` gombra kattintott, és nem „üres" nevet adott a könyvjelzőnek. Ha az új név megegyezik a régivel, akkor egyszerűen beállítjuk a „piszkos" jelzőt, és készen is vagyunk (ez esetben ugyanis a felhasználó feltehetőleg az URL-t módosította), ha viszont a könyvjelző neve megváltozott, akkor töröljük a szótárból azt az elemet, amelynek a kulcsa a régi név, kiürítjük a listamezőt, majd újra feltöltjük a listát a könyjelzőkkel, ugyanúgy, ahogy egy könyvjelző hozzáadásakor tettük.

```
def editDelete(self, *ignore):
    indexes = self.listBox.curselection()
    if not indexes or len(indexes) > 1:
        return
    index = indexes[0]
    name = self.listBox.get(index)
    if tkinter.messagebox.askyesno("Bookmarks - Delete",
                                   "Delete '{0}'?".format(name)):
        self.listBox.delete(index)
        self.listBox.focus_set()
        del self.data[name]
        self.dirty = True
```

Ha egy könyvjelzőt törölni szeretnénk, először ki kell derítenünk, hogy a felhasználó melyik könyvjelzőt választotta, ezért a fenti tagfüggvény ugyanazokkal a sorokkal kezdődik, mint a `MainWindow.editEdit()`. Ha a felhasználó egyetlen könyvjelzőt jelölt ki, megjelenítünk egy üzenetablakot, amelyben a felhasználót a törlés megerősítésére kérjük. Ha igennel válaszol, az üzenetablak függvénye `True`-t ad vissza, ezért töröljük a könyvjelzőt a listából, valamint a `self.data` szótárból, illetve beállítjuk a `dirty` jelzőt, és a billentyűfókuszt visszaadjuk a listamezőnek.

```
    def editShowWebPage(self, *ignore):
        indexes = self.listBox.curselection()
        if not indexes or len(indexes) > 1:
            return
        index = indexes[0]
        url = self.data[self.listBox.get(index)]
        webbrowser.open_new_tab(url)
```

Ha a felhasználó kiváltja ennek a tagfüggvénynek a meghívását, akkor megkeressük az általa kijelölt könyvjelzőt, és kinyerjük a megfelelő URL-t a `self.data` szótárból. Ez után a `webbrowser` modul `webbrowser.open_new_tab()` függvényével megnyitjuk az adott URL-hez tartozó oldalt a felhasználó böngészőjében. Ha a böngésző még nem fut, elindítjuk azt.

```
application = tkinter.Tk()
path = os.path.join(os.path.dirname(__file__), "images/")
if sys.platform.startswith("win"):
    icon = path + "bookmark.ico"
    application.iconbitmap(icon, default=icon)
else:
    application.iconbitmap("@" + path + "bookmark.xbm")
window = MainWindow(application)
application.protocol("WM_DELETE_WINDOW", window.fileQuit)
application.mainloop()
```

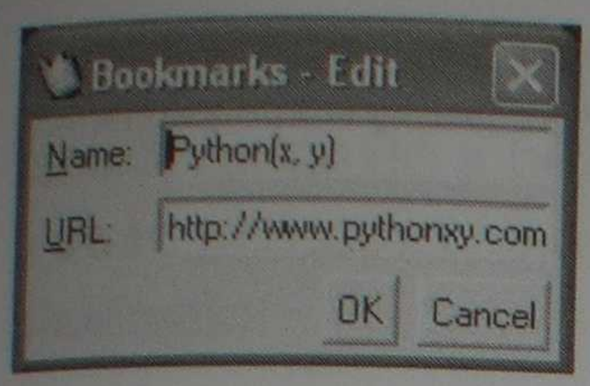
A program utolsó sorai megegyeznek a korábban látott `interest-tk.pyw` programban használtakkal, három eltéréssel. Az első az, hogy ha a felhasználó a programablak bezáró gombjára kattint, a Bookmarks programban más tagfüggvényt hívunk meg, mint az Interest programban. A második, hogy Windowson az `iconbitmap()` tagfüggvénynek van még egy paramétere, amelynek a segítségével alapértelmezett ikont határozhatunk meg a program összes ablaka számára – erre Unixon nincs szükség, mert ott ez önműködően történik. Az utolsó különbség pedig az, hogy az alkalmazás címsorát nem itt, hanem a `MainWindow` osztály tagfüggvényeiben állítjuk be. Az Interest programban a cím soha nem változik meg, ezért csak egyszer kell beállítanunk, a Bookmarks programban azonban a címet mindig annak megfelelően kell módosítanunk, hogy melyik könyvjelzőfájlon dolgozunk.

Most, hogy láttuk a főablak osztályának megvalósítását, valamint a programot előkészítő és az eseményhurkot elindító kódot, a figyelmünket az `AddEditForm` párbeszédablakra fordíthatjuk.

Egyéni párbeszédablak létrehozása

Az `AddEditForm` párbeszédablak nyújt lehetőséget a felhasználóknak a könyvjelzők neveinek és URL-jeinek a szerkesztésére, illetve hozzaádására. Az ablakot a 13.6. ábrán láthatjuk, ahol éppen egy meglevő könyvjelzőt szerkesztünk (ezért szerepel „Edit", vagyis

„Szerkesztés" a címsorban). Ugyanezt a párbeszédablakot könyvjelzők felvételére is használhatjuk. Először tekintsük át a párbeszédablak előkészítő kódját, mégpedig négy részre bontva.



13.6. ábra

A Bookmarks program Add/Edit párbeszédablaka

```
class AddEditForm(tkinter.Toplevel):
    def __init__(self, parent, name=None, url=None):
        super(AddEditForm, self).__init__(parent)
        self.parent = parent
        self.accepted = False
        self.transient(self.parent)
        self.title("Bookmarks - " + (
                    "Edit" if name is not None else "Add"))
        self.nameVar = tkinter.StringVar()
        if name is not None:
            self.nameVar.set(name)
        self.urlVar = tkinter.StringVar()
        self.urlVar.set(url if url is not None else "http://")
```

Úgy döntöttünk, hogy a `tkinter.TopLevel`-től öröklünk, amely egy olyan csupasz vezérlő, amely a legfelső szintű ablakként használt vezérlők alaposztályaként szolgál. (Ahogy korábban említettük, a `super().__init__(parent)` kódot szerettük volna írni, de mivel ez nem működött, a `super()`-t kellett meghívnunk, kifejezetten átadva neki paraméterként a szülőosztályt és a `self` objektumot.) Elraktározunk egy a szülőre mutató hivatkozást, létrehozzuk a `self.accepted` jellemzőt, és `False`-ra állítjuk azt. A `transient()` tagfüggvényt azért hívjuk meg, hogy tájékoztassuk a szülőablakot, hogy ennek az ablaknak mindig a szülő felett kell megjelennie. A címsor beállításával jelezzük, hogy éppen hozzáadunk vagy szerkesztünk – a beállítást attól függően végezzük el, hogy kaptunk-e nevet és URL-t. A könyvjelző nevét és URL-jét két `tkinter.StringVar` követi nyomon, amelyekbe az átadott értékek kerülnek, ha a párbeszédablakot szerkesztésre használják.

```
        frame = tkinter.Frame(self)
        nameLabel = tkinter.Label(frame, text="Name:", underline=0)
        nameEntry = tkinter.Entry(frame, textvariable=self.nameVar)
        nameEntry.focus_set()
        urlLabel = tkinter.Label(frame, text="URL:", underline=0)
        urlEntry = tkinter.Entry(frame, textvariable=self.urlVar)
        okButton = tkinter.Button(frame, text="OK", command=self.ok)
        cancelButton = tkinter.Button(frame, text="Cancel", command=self.close)
        nameLabel.grid(row=0, column=0, sticky=tkinter.W, pady=3, padx=3)
```

```
nameEntry.grid(row=0, column=1, columnspan=3,
               sticky=tkinter.EW, pady=3, padx=3)
urlLabel.grid(row=1, column=0, sticky=tkinter.W, pady=3, padx=3)
urlEntry.grid(row=1, column=1, columnspan=3,
              sticky=tkinter.EW, pady=3, padx=3)
okButton.grid(row=2, column=2, sticky=tkinter.EW, pady=3, padx=3)
cancelButton.grid(row=2, column=3, sticky=tkinter.EW, pady=3, padx=3)
```

nameLabel	nameEntry		
urlLabel	urlEntry		
		okButton	cancelButton

13.7. ábra

A Bookmarks program Add/Edit párbeszédablakának elrendezése

Ahogy a 13.7. ábrán láthatjuk, a vezérlőket egy rácsban hozzuk létre és rendezzük el. A név és az URL bevitelére szolgáló szövegmezők vezérlőit a megfelelő `tkinter.StringVar` változókhoz társítjuk, a két gomb pedig a később bemutatandó `self.ok()` és `self.close()` tagfüggvényeket fogja meghívni.

```
frame.grid(row=0, column=0, sticky=tkinter.NSEW)
frame.columnconfigure(1, weight=1)
window = self.winfo_toplevel()
window.columnconfigure(0, weight=1)
```

A párbeszédablakot csak vízszintesen van értelme átméretezni, ezért az ablakkeret második oszlopát vízszintesen méretezhetővé tesszük az oszlop súlyozását 1-re állítva – ez azt jelenti, hogy ha a keretet vízszintesen megnyújtják, az 1-es oszlopban levő vezérlők (a név és az URL bevitelére szolgáló szövegmezők) is megnőnek, hogy kitöltsék a többlethelyet. Ugyanígy az ablak oszlopát is vízszintesen átméretezhetővé tesszük a súlya 1-re állításával. Ha a felhasználó a párbeszédablak magasságát módosítja, a vezérlők megtartják az egymáshoz viszonyított helyzetüket, és az ablakon belül középre igazodnak, de ha a párbeszédablak szélessége változik meg, a név és az URL bevitelére szolgáló szövegmezők a vízszintesen rendelkezésre álló hely függvényében összezsugorodnak vagy megnyúlnak.

```
self.bind("<Alt-n>", lambda *ignore: nameEntry.focus_set())
self.bind("<Alt-u>", lambda *ignore: urlEntry.focus_set())
self.bind("<Return>", self.ok)
self.bind("<Escape>", self.close)
self.protocol("WM_DELETE_WINDOW", self.close)
self.grab_set()
self.wait_window(self)
```

Itt két szövegcímkét hozunk létre `Name:` és `URL:` szöveggel, amelyekre kattintva a billentyűfókusz a megfelelő szövegbeviteli vezérlőre kerül (az aláhúzások jelzik, hogy a két vezérlőhöz az ALT+N, illetve az ALT+U gyorsbillentyű tartozik). A szükséges billentyűkötéseket is

megadjuk, hogy mindez működjön. Lambdafüggvényeket használunk a `focus_set()` tagfüggvények közvetlen átadása helyett, hogy az eseményparamétert figyelmen kívül hagyhassuk. Ezen kívül az `OK` és `Cancel` gombok szabványos billentyűkötéseit (ENTER és Esc) is megadjuk.

A `protocol()` tagfüggvénnyel határozzuk meg, hogy melyik tagfüggvényt kell meghívni, ha a felhasználó a bezáró gombra kattintva bezárja a párbeszédablakot. A `grab_set()` és a `wait_window()` hívásokra azért van szükség, hogy az ablakot modális párbeszédablakká alakítsuk.

```
def ok(self, event=None):
    self.name = self.nameVar.get()
    self.url = self.urlVar.get()
    self.accepted = True
    self.close()
```

Ez a tagfüggvény hívódik meg, amikor a felhasználó az `OK` gombra kattint (vagy lenyomja az ENTER billentyűt). A függvény a `tkinter.StringVar` változókból a megfelelő példányváltozókba másolja a szövegeket (mely példányváltozókat csak most hozzuk létre), a `self.accepted` változót `True` értékre állítja, és meghívja a `self.close()` tagfüggvényt a párbeszédablak bezárásához.

```
def close(self, event=None):
    self.parent.focus_set()
    self.destroy()
```

Ezt a tagfüggvényt a `self.ok()` tagfüggvényből hívjuk meg, illetve akkor, ha a felhasználó az ablak bezáró gombjára vagy a `Cancel` gombra kattint (vagy lenyomja az Esc billentyűt). A függvény visszaadja a billentyűfókuszt a szülőnek, és a párbeszédablakot önmaga megsemmisítésére utasítja. Ebben az összefüggésben a megsemmisítés csak annyit jelent, hogy az ablak és annak vezérlői törlődnek – az `AddEditForm`-példány továbbra is létezni fog, mert a hívó még hivatkozik rá.

Miután a párbeszédablak bezáródott, a hívó megvizsgálja az `accepted` változót, és ha annak értéke `True`, akkor kiolvassa a felhasználó által megadott vagy módosított nevet és URL-t. Ezt követően, ha a `MainWindow.editAdd()` vagy a `MainWindow.editEdit()` tagfüggvény befejezte a működését, az `AddEditForm` objektum kikerül a hatókörből, és kijelölődik szemétgyűjtésre.

Összefoglalás

Ez a fejezet ízelítőt adott a grafikus felület programozásából a Tk GUI-könytár segítségével. A Tk nagy előnye, hogy része a Python alapcsomagjának, ugyanakkor számos hátránya is van – nem utolsósorban az, hogy régi könyvtár, amely némileg másképp működik, mint mai versenytársai többsége.

Ha még kezdők vagyunk a grafikus programozás terén, ne feledjük, hogy a Tk főbb rendszerfüggetlen versenytársai – a PyGtk, a PyQt és a wxPython – sokkal könnyebben megtanulhatók és használhatók, mint a Tk, és mindegyikkel jobb eredmény érhető el, ráadásul kevesebb kóddal. Ezek a Tk-riválisok továbbá kivétel nélkül bővebb és jobb Python-központú leírással, lényegesen több vezérlővel és szebb megjelenéssel rendelkeznek, és azt is lehetővé teszik, hogy úgy hozzunk létre a semmiből új vezérlőket, hogy közben a megjelenésüket és a viselkedésüket teljesen kézben tartjuk.

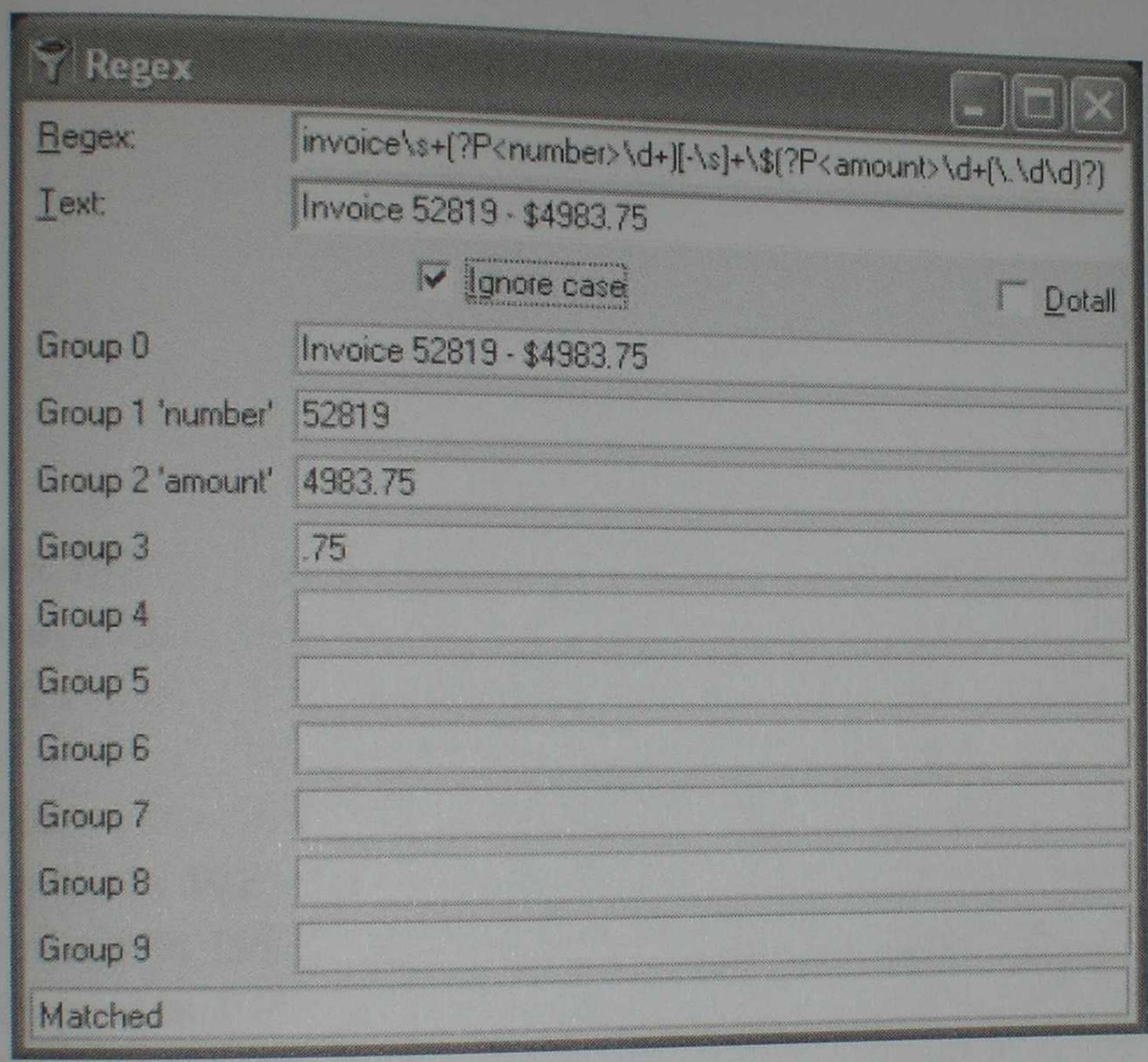
Bár a Tk-nak aprócska programok írásakor, illetve olyan helyzetekben, amikor csak a Python alapkönyvtára áll a rendelkezésünkre, hasznát vehetjük, minden más esetben bármely másik rendszerfüggetlen programkönyvtár sokkal jobb választás.

Gyakorlatok

Az első gyakorlatban az ebben a fejezetben bemutatott Bookmarks programot kell lemásolni és módosítani, míg a másodikban a semmiből kell elkészíteni egy grafikus felületű programot.

1. Másoljuk le a `bookmarks-tk.pyw` programot, és módosítsuk úgy, hogy képes legyen a (11. fejezetben gyakorlatként elkészített) `bookmarks.py` konzolprogram által használt DBM-fájlok betöltésére és mentésére. Ehhez adjunk két menüpontot – `Import` és `Export` – a File menühöz, és ne felejtsünk el billentyűparancsokat kapcsolni hozzájuk (ne feledjük, hogy a CTRL+E billentyűparancsot már használjuk az Edit, Edit menüponthoz). Ugyanígy hozzuk létre a két megfelelő eszköztárgombot is. Ehhez körülbelül öt sornyi kóddal kell kiegészítenünk a főablak előkészítő függvényét.
   Az említett lehetőségek megvalósításához két tagfüggvényre lesz szükség: a `fileImport()` és a `fileExport()` nevűekre, amelyek kevesebb mint 60 kódsorral megvalósíthatók, a hibakezelést is beleértve. Betöltéskor eldönthetjük, hogy a betöltött könyvjelzőket össze szeretnénk-e fésülni a meglevőkkel, vagy inkább lecseréljük a meglevő könyvjelzőket a betöltöttekre. A kód nem bonyolult, de a megírása némi körültekintést igényel. Egy megoldást (amely a betöltött könyvjelzőket egyesíti a meglevőkkel) a `bookmarks-tk_ans.py` fájlban találhatunk.
2. A 12. fejezetben láttuk, hogyan hozhatunk létre és használhatunk szabályos kifejezéseket szövegek illesztésére. Készítsünk olyan párbeszédablakos grafikus felületű programot, amelyet szabályos kifejezések beírására és tesztelésére használhatunk (lásd a 13.8. ábrát).
   A feladat megoldásához el kell olvasnunk a `re` modul leírását, mert a programnak akkor is helyesen kell működnie, ha érvénytelen szabályos kifejezéssel találkozik, vagy ha végig kell járnia az illesztési csoportokat, mivel a legtöbb esetben a szabályos kifejezés nem fog annyi illesztési csoportot tartalmazni, mint ahány szövegcímke rendelkezésre áll. Gondoskodjunk róla, hogy a program teljes támogatást nyújtson azoknak, akik a billentyűzetet használják: a szövegbeviteli vezérlők között lehessen

mozogni az ALT+R és ALT+T billentyűkkel, a jelölőnégyzeteket kezelni az ALT+I és ALT+D billentyűparancsok segítségével, ki lehessen lépni a programból a CTRL+Q és az ESC billentyűkkel, és végezzünk újraszámítást , ha a felhasználó lenyom és felenged egy billentyűt bármelyik szövegbeviteli vezérlőben, illetve ha bejelöl egy jelölőnégyzetet, vagy törli belőle a jelet.



13.8. ábra

A Regex program

A program megírása nem túl nehéz, bár a találatokat és a csoportszámokat (illetve neveket, ha meg vannak adva) megjelenítő kód egy icipicit trükkös. Egy megoldást a `regex-tk.pyw` fájlban találhatunk, amely körülbelül 140 sorból áll.