LEKCE 5

Moduly

V této lekci:

- Moduly a balíčky
- Přehled standardní knihovny Pythonu

Zatímco pomocí funkcí můžeme rozdělit části kódu tak, aby je bylo možné opětovně použít v rámci daného programu, moduly poskytují prostředky pro shromažďování skupin funkcí (a jak uvidíme v následující lekci také vlastních datových typů) tak, aby je bylo možné použít v libovolném počtu programů. Python dále nabízí možnost vytvářet balíčky, což jsou skupiny modulů seskupené dohromady, obvykle z toho důvodu, že tyto moduly poskytují související funkční prvky, nebo proto, že na sobě vzájemně závisejí.

První část této lekce popisuje syntaxe pro importování funkčních prvků z modulů a balíčků, ať už se standardní knihovny nebo z našich vlastních modulů a balíčků. Ve druhé části se posuneme dále a podíváme se, jak vytvářet vlastní balíčky a moduly. Ukážeme si dva vlastní moduly, z nichž první bude seznamovací a druhý bude demonstrovat, jak vyřešit řadu praktických problémů, jako je nezávislost na platformě a testování.

Webová dokumentace ➤ 171

Druhá část nabízí stručný přehled standardní knihovny Pythonu. Je důležité mít na paměti, co tato knihovna obsahuje, protože používání předem definovaných funkčních prvků značně zrychluje programování. Mnohé moduly standardní knihovny je široce rozšířené, dobře otestované a robustní. Kromě přehledu si pomocí malých příkladů ukážeme několik běžných případů užití. U modulů probíraných v jiných lekcích budou navíc uvedeny křížové odkazy.

Moduly a balíčky

Modul jazyka Python je jednoduše řešeno soubor s příponou `.py`. Modul může obsahovat libovolný kód jazyka Python. Všechny programy, které jsme dosud napsali, byly obsažené v jediném souboru `.py`, a proto se jedná o moduly i programy. Podstatným rozdílem je to, že programy jsou navržené pro spouštění, kdežto moduly jsou navržené pro importování a použití v programech.

Ne všechny moduly mají přidružené soubory `.py`. Například modul `sys` je vestavěný do Pythonu a některé moduly jsou napsané v jiných jazycích (povětšinou v jazyku C). Nicméně větší část knihovny Pythonu je napsána v jazyku Python. Pokud tedy například napíšeme `import collections`, můžeme voláním `collections.namedtuple()` vytvářet pojmenované n-tice a funkčnost, k níž přistupujeme, se nachází v souboru modulu `collections.py`. Z hlediska našeho programu je nepodstatné, ve kterém jazyku je daný modul napsán, protože všechny moduly se importují a používají stejným způsobem.

Pro importování lze použít několik syntaxí:

```
import importovatelný_prvek
import importovatelný_prvek1, importovatelný_prvek2, ..., importovatelný_prvekN
import importovatelný_prvek as preferovaný_název
```

Balíčky ➤ 197

Zde je *importovatelný_prvek* obvykle modulem, jako je `collections`, může však jít také o balíček nebo modul v balíčku, přičemž se každá část odděluje tečkou (`.`), například `os.path`. První dvě syntaxe používáme v rámci této knihy. Jsou nejjednodušší a také nejbezpečnější, protože nehrozí možnost konfliktu názvů, což je dáno tím, že musíme vždy používat plně kvalifikované názvy.

Třetí syntaxe nám umožňuje dát importovanému balíčku nebo modulu název dle vlastní volby. To sice může teoreticky vést ke kolizi názvů, ovšem v praxi se tato syntaxe používá právě proto, aby k ní nedocházelo. Přejmenování je užitečné zejména tehdy, když experimentujeme s různými implementacemi nějakého modulu. Pokud máme například dva moduly `MyModuleA` a `MyModuleB`, které mají stejné rozhraní API (Application Programming Interface – aplikační programovací rozhraní), mohli bychom v programu napsat `import MyModuleA as MyModule` a později hladce přejít k druhému modulu příkazem `import MyModuleB as MyModule`.

Kde bychom měli příkazy `import` umisťovat? Běžně se všechny příkazy `import` umisťují na začátek souborů `.py`, za řádek shebang a před dokumentaci modulu. Jak jsme si již řekli v lekci 1, doporučujeme nejdříve importovat moduly standardní knihovny, poté moduly knihoven třetích stran a nakonec naše vlastní moduly.

Zde je několik dalších syntaxí příkazu `import`:

```
from importovatelný_prvek import objekt as preferovaný_název
from importovatelný_prvek import objekt1, objekt2, ..., objektN
from importovatelný_prvek import (objekt1, objekt2, objekt3, objekt4, objekt5,
                                  objekt6, ..., objektN)
from importovatelný_prvek import *
```

Tyto syntaxe mohou způsobovat konflikty názvů, poněvadž činí importované objekty (proměnné, funkce, datové typy nebo moduly) přímo přístupné. Pokud chceme použít syntaxi `from ... import` pro importování velkého množství objektů, můžeme použít více řádků, a to buď tak, že každý nový řádek kromě posledního potlačíme nebo názvy objektů uzavřeme do závorek, jak ukazuje třetí syntaxe.

Hvězdička (`*`) v poslední syntaxi znamená „importuj vše, co není soukromé“, což v praxi znamená buď to, že se importuje každý objekt v modulu kromě těch, jejichž název začíná podtržítkem, nebo že se importují všechny objekty, jejichž názvy jsou obsaženy v seznamu `__all__` definovaném jako globální proměnná daného modulu.

`__all__` ➤ 198

Zde je několik příkladů použití příkazu `import`:

```
import os
print(os.path.basename(filename)) # bezpečný, plně kvalifikovaný přístup

import os.path as path
print(path.basename(filename))    # riziko kolize názvů s path

from os import path
print(path.basename(filename))    # riziko kolize názvů s path

from os.path import basename
print(basename(filename))         # riziko kolize názvů s basename

from os.path import *
print(basename(filename))         # riziko kolize spousty názvů
```

Syntaxe `from importovatelný_prvek import *` importuje všechny objekty z daného modulu (nebo všechny moduly z daného balíčku), což může znamenat stovky názvů. V případě příkazu `from os.path import *` se importuje téměř 40 názvů, mezi něž patří i `dirname`, `exists` a `split`, což mohou být názvy, které bychom raději použili pro naše vlastní proměnné nebo funkce.

Pokud například napíšeme příkaz `from os.path import dirhame`, pak můžeme přímo zavolat `dirname()` bez kvalifikace. Pokud poté ale ve svém kódu napíšeme `dirname = "."`, bude nyní odkaz na objekt `dirname` místo funkce `dirname()` svázán s řetězcem ".", takže když se pokusíme zavolat `dirname()`, obdržíme výjimku `TypeError`, protože `dirname` nyní ukazuje na řetězec a řetězce nejsou volatelné.

Z hlediska případných kolizí názvů vytvářených syntaxí import * některé vývojářské týmy specifikují ve svých směrnicích, že se může používat pouze syntaxe `import importovatelný_prvek`. Nicméně určité rozsáhlé balíčky, zvláště pak knihovny GUI, se často importují tímto způsobem, protože obsahují obrovské množství funkcí a tříd (vlastních datových typů), jejichž ruční vypisování by bylo velice pracné.

Přirozeně zde vyvstává otázka, jak Python ví, kde importované moduly a balíčky hledat? Vestavěný modul `sys` obsahuje seznam `sys.path`, který uchovává seznam adresářů, které představují *cestu Pythonu* (Python path). První adresář je adresářem, který obsahuje samotný program, a to i tehdy, pokud byl program vyvolán z jiného adresáře. Je-li nastavena proměnná prostředí PYTHONPATH, jsou v ní specifikované další cesty v seznamu. Poslední cesty jsou ty, které jsou nezbytné pro přístup k standardní knihovně Pythonu – nastavují se při instalaci Pythonu.

Upozornění: Při prvním importu vestavěného modulu se Python po tomto modulu podívá postupně v každé cestě uvedené v seznamu `sys.path`. Jedním z důsledků tohoto postupu je to, že pokud vytvoříme modul nebo program se stejným názvem, jako má jeden z knihovních modulů Pythonu, najde se ten náš jako první, což nevyhnutelně způsoby problémy. Aby k tomuto nedošlo, nikdy nevytvářejte program či modul se stejný názvem, jako má některý z knihovních adresářů či modulů Pythonu nejvyšší úrovně. Výjimkou je samozřejmě situace, kdy vytváříte svou vlastní implementaci takovéhoto modulu a záměrně jej přepisujete. (Modul nejvyšší úrovně je modul, jehož soubor `.py` je umístěn v jednom z adresářů v cestě Pythonu, a ne v jednom z jejich podadresářů.) Například v systému Windows obsahuje cesta Pythonu obvykle adresář s názvem `C:\Python31\Lib`, takže na této platformě bychom neměli vytvářet modul s názvem `Lib.py` ani modul se stejným názvem, jako má kterýkoliv modul v adresáři `C:\Python31\Lib`.

Pro rychlý způsob kontroly, zda se nějaký název modulu již používá, stačí vyzkoušet příkaz `import` s tímto názvem. To lze provést v konzole zavoláním interpretu s volbou příkazového řádku `-c` („execute code" – proveď kód), za níž umístíme příslušný příkaz `import`. Pokud například chceme zjistit, zda existuje modul s názvem `Music.py` (nebo adresář nejvyšší úrovně v cestě Pythonu s názvem `Music`), pak můžeme do konzoly napsat následující příkaz:

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

Pokud obdržíme výjimku `ImportError`, víme, že se žádný modul nebo adresář nejvyšší úrovně tohoto jména nepoužívá. Jakýkoliv jiný výstup (nebo žádný) znamená, že toto jméno je již zabráno. Tento postup ale naneštěstí nezaručuje, že daný název bude vždy bez problémů, poněvadž můžeme později nainstalovat balíček či modul Pythonu od třetí strany, který obsahuje konfliktní název, i když v praxi se jedná o velmi ojedinělý problém.

Pokud bychom například vytvořili soubor modulu s názvem `os.py`, došlo by ke konfliktu s knihovním modulem `os`. Pokud ale vytvoříme modul s názvem `path.py`, bude to v pořádku, protože bude importován jako modul `path`, kdežto knihovní modul se importuje jako `os.path`. V této knize je v názvech souborů našich vlastních modulů první písmeno vždy velké, čímž se vyhneme konfliktům názvů (alespoň na Unixu), protože názvy souborů modulů standardní knihovny obsahují jen malá písmena.

Program může importovat některé moduly, které následně importují své vlastní moduly, včetně některých, které již byly importovány. To však nezpůsobí žádné problémy. Kdykoliv je totiž nějaký modul importován, tak Python nejdříve zkontroluje, zda již nebyl importován dříve. Pokud nebyl, tak Python spustí zkompilovaný bajt-kód modulu, čímž vytvoří příslušné proměnné, funkce a další objekty daného modulu a interně si poznačí, že tento modul již byl importován. Při každém dalším importu tohoto modulu Python detekuje, že již byl importován, a neudělá nic.

Když Python potřebuje zkompilovaný bajt-kód modulu, automaticky jej vygeneruje. Tím se liší třeba od Javy, kde se kompilace do bajt-kódu musí provést explicitním způsobem. Python se nejdříve podívá po souboru se stejným názvem jako má soubor `.py` daného modulu, ale s příponou `.pyo`, což je optimalizovaná verze se zkompilovaným bajt-kódem modulu. Pokud žádný soubor `.pyo` neexistuje (nebo pokud je starší než soubor `.py`, což znamená, že není aktuální), Python se podívá po souboru s příponou `.pyc`, což je neoptimalizovaná verze se zkompilovaným bajt-kódem modulu. Pokud Python najde aktuální verzi se zkompilovaným bajt-kódem modulu, tak ji načte. V opačném případě načte soubor `.py` a zkompiluje verzi se zkompilovaným bajt-kódem. Tak jako tak bude mít nakonec Python modul v paměti ve formě zkompilovaného bajt-kódu.

Pokud by Python musel zkompilovat soubor `.py`, uloží jej do souboru `.pyc` (nebo `.pyo`, pokud je na příkazovém řádku uvedena volba `-O` nebo pokud je nastavena proměnná prostředí `PYTHONOPTIMIZE`), ovšem za předpokladu, že do příslušného adresáře lze zapisovat. Ukládání bajt-kódu lze zamezit volbou příkazového řádku `-B` nebo nastavením proměnné prostředí `PYTHONDONTWRITEBYTECODE`.

Používání souborů se zkompilovaným bajt-kódem vede k rychlejšímu spouštění, protože interpret nemusí kód načítat, kompilovat, ukládat (je-li to možné) a spouštět, ale stačí mu jen kód načíst a spustit. Je však třeba poznamenat, že na samotný běh kódu to žádný vliv nemá. Při instalaci Pythonu se moduly standardní knihovny obvykle zkompilují do bajt-kódu v rámci instalačního procesu.

Balíčky

Balíček je obyčejným adresářem, který obsahuje skupinu modulů a soubor s názvem `__init__.py`. Představte si, že máme smyšlenou skupinu souborů modulů pro čtení a zapisování nejrůznějších formátů grafických souborů, jako jsou například moduly `Bmp.py`, `Jpeg.py`, `Png.py`, `Tiff.py` a `Xpm.py`, z nichž každý nabízí funkce `load()`, `save()` a tak podobně.* Moduly bychom mohli nechat ve stejném adresáři, v němž je náš program, ale u větších programů, které používají množství vlastních modulů, by se grafické moduly úplně ztratily. Jejich umístěním do samostatného podadresáře (např. `Graphics`) je budeme mít pěkně pohromadě. A pokud k nim do adresáře `Graphics` přidáme prázdný soubor `__init__.py`, stane se z tohoto adresáře balíček:

* Rozsáhlou podporu pro práci s grafickými soubory poskytuje spousta modulů třetích stran, z nichž je nejpozoruhodnější knihovna Python Imaging Library (*www.pythonware.com/products/pil*).

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

Dokud bude adresář `Graphics` podadresářem uvnitř adresáře našeho programu nebo bude uveden v cestě Pythonu, můžeme libovolný z těchto modulů importovat a používat. Musíme ale myslet na to, aby název zastřešujícího modulu (`Graphics`) nebyl stejný jako kterýkoliv z názvů nejvyšší úrovně ve standardní knihovně, jinak by došlo ke konfliktu názvů. (Na systému Unix stačí, když budou mít názvy našich modulů první písmeno velké, neboť všechny moduly standardní knihovny mají názvy s malými písmeny.) Náš modul můžeme importovat a používat takto:

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

U krátkých programů můžeme používat kratší názvy, což lze v Pythonu provést dvěma malinko odlišnými způsoby.

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

Zde jsme importovali modul `Jpeg` z balíčku `Graphics` a řekli jsme Pythonu, že se na něj chceme místo plně kvalifikovaného jména `Graphics.Jpeg` odkazovat jen jako na `Jpeg`.

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

V tomto úryvku kódu importujeme modul `Png` přímo z balíčku `Graphics`. Při použití této syntaxe (`from ... import`) je modul `Png` přímo přístupný. Nikdo nás nenutí používat v našem kódu názvy používané v balíčku:

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

Zde používáme modul `Tiff`, který jsme ale ve svém programu přejmenovali na modul `picture`.

V některých situacích je výhodné načíst všechny moduly balíčku pomocí jediného příkazu. K tomu je nutné upravit soubor `__init__.py` daného balíčku tak, aby obsahoval příkaz, který stanoví, které moduly se mají načíst. Tento příkaz musí přiřadit seznam názvů modulů do speciální proměnné `__all__`. Zde je například požadovaný řádek pro soubor `Graphics/__init__.py`:

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

Nic víc není třeba, i když do souboru `__init__.py` samozřejmě můžeme umístit libovolný další kód. Nyní můžeme použít další typ příkazu `import`:

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

Syntaxe `from balíček import *` přímo importuje všechny moduly uvedené v seznamu `__all__`. Po provedení tohoto příkazu je přímo přístupný nejen modul `Xpm`, ale také všechny ostatní moduly.

Jak jsme si řekli již dříve, tuto syntaxi lze aplikovat také na modul (tj. `from modul import *`) při tom se importují všechny funkce, proměnné a další objekty definované v zadaném modulu (kromě těch, jejichž název začíná podtržítkem). Pokud chceme mít kontrolu nad tím, co přesně se má při použití syntaxe `from modul import *` importovat, můžeme definovat seznam `__all__` i v samotném modulu. V takovém případě importuje příkaz `from modul import *` pouze ty objekty, jejichž názvy jsou uvedeny v seznamu `__all__`.

Dosud jsme si ukázali pouze jednu úroveň zanoření, Python nám však umožňuje vnořovat balíčky tak hluboko, jak potřebujeme. Můžeme tak mít v adresáři `Graphics` třeba podadresář `Vector` a v něm soubory modulů, například `Eps.py` a `Svg.py`:

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
    Tiff.py
    Vector/
        __init__.py
        Eps.py
        Svg.py
    Xpm.py
```

Adresář `Vector` se stane balíčkem v okamžiku, kdy do něj umístíme soubor `__init__.py`, a jak jsme si již řekli, tento soubor může být prázdný nebo může obsahovat seznam `__all__` pro potřeby programátorů, kteří chtějí provést import pomocí příkazu `from Graphics.Vector import *`.

Pro přístup k vnořenému balíčku vycházíme ze syntaxe, kterou jsme již používali:

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

Plně kvalifikovaný název je poněkud delší, čemuž se snaží někteří programátoři zamezit tím, že se snaží udržovat hierarchie svých modulů docela ploché.

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

Pro modul můžeme vždy použít náš vlastní krátký název, jak jsme to provedli zde, i když se tím vystavujeme vyššímu riziku konfliktu názvů.

Všechny importy, které jsme dosud používali (a které budeme používat i ve zbývající části knihy), se označují jako *absolutní* importy, což znamená, že každý námi importovaný modul je v některém

z adresářů uvedených v proměnné `sys.path` (nebo podadresářů, pokud název v příkazu import obsahuje jednu či více teček, které slouží v podstatě jako oddělovače v cestě). Při vytváření rozsáhlých balíčků s více moduly a adresáři je často užitečné importovat další moduly, které jsou součástí stejného balíčku. Například v souboru `Eps.py` nebo `Svg.py` můžeme získat přístup k modulu `Png` pomocí konvenčního nebo *relativního* importu:

```
import Graphics.Png as Png
```

```
from ..Graphics import Png
```

Tyto dva úryvky kódu jsou ekvivalentní, protože oba přímo zpřístupňují modul `Png` uvnitř modulu, kde se používají. Všimněte si ale, že relativní importy, což jsou importy, které používají syntaxi `from` *`modul`* `import` s tečkami před názvem modulu (každá tečka představuje jeden krok směrem nahoru v hierarchii adresářů), můžeme použít pouze v modulech, které jsou uvnitř nějakého balíčku. Relativní importy usnadňují přejmenování zastřešujícího balíčku a uvnitř balíčků zamezují nechtěnému importování standardních modulů místo našich vlastních.

Vlastní moduly

Moduly jsou obyčejné soubory s příponou `.py`, a proto je lze vytvářet bez dalších formalit. V této části se podíváme na dva naše vlastní moduly. První modul `TextUtil` (v souboru `TextUtil.py`) obsahuje jen tři funkce: `is_balanced()`, která vrací hodnotu `True`, má-li zadaný řetězec vyvážené závorky nejrůznějších druhů, `shorten()` (viz dříve – strana 175) a `simplify()`, která dokáže z řetězce odříznout nežádoucí bílé místo a další znaky. Při probírání tohoto modulu se podíváme také na to, jak spouštět kód v dokumentačních řetězcích jako testy jednotek.

Druhý modul `CharGrid` (v souboru `CharGrid.py`) uchovává znakovou mřížku a umožňuje do této mřížky „kreslit“ čáry, obdélníky a text a celou mřížku vykreslit do konzoly. V tomto modulu si ukážeme několik technik, se kterými jsme se dosud nesetkali a které jsou typické pro rozsáhlejší a komplexnější moduly.

Modul TextUtil

Struktura tohoto modulu (a většiny dalších) se odlišuje od struktury programu. Prvním řádkem je řádek shebang a poté zde máme komentáře (obvykle copyright a informace o licenci). Dále je obvykle uváděn řetězec s trojitými uvozovkami, který poskytuje přehled obsahu modulu často doplněný ukázkovými příklady – jedná se o dokumentační řetězec modulu. Zde je začátek souboru `TextUtil.py` (ovšem bez řádků s informacemi o licenci):

```
#!/usr/bin/env python3
# Copyright (c) 2008-9 Qtrac Ltd. All rights reserved.
"""
Tento modul nabízí několik funkcí pro manipulaci s řetězci.

>>> is_balanced("(Python (není (jako (lisp))))")
True
>>> shorten("Velká křižovatka", 10)
'Velká k...'
```

```
>>> simplify(" nějaký    text    s  nadbytečnými  mezerami  ")
'nějaký text s nadbytečnými mezerami'
"""

import string
```

Dokumentační řetězec modulu je k dispozici programům (nebo dalším modulům), které tento modul importují, jako `TextUtil.__doc__`. Za dokumentačním řetězcem modulu následují importy a poté zbytek modulu.

Funkci `shorten()` jsme již viděli, a proto ji zde nebudeme opakovat. A protože se spíše než na funkce zaměřujeme na moduly, ukážeme si kromě funkce `simplify()`, kterou si ukážeme celou včetně dokumentačního řetězce, pouze kód funkce `is_balanced()`.

shorten() ➤ 175

Takto vypadá funkce `simplify()` rozdělená na dvě části:

```
def simplify(text, whitespace=string.whitespace, delete=""):
    r"""Vrátí text s vícenásobnými mezerami zredukovanými do jediné mezery

    Parametr whitespace je řetězec znaků, z nichž každý
    je považován za mezeru.
    Není-li parametr delete prázdný, měl by obsahovat řetězec, jehož
    znaky se vyhledají ve výsledném řetězci a odstraní.

    >>> simplify(" tohle    a\n také\t tamto")
    'tohle a také tamto'
    >>> simplify("  Vejce   a.s.\n")
    'Vejce a.s.'
    >>> simplify("  Vejce   a.s.\n", delete=",;:.")
    'Vejce as'
    >>> simplify(" nesamohláskový ", delete="aáeiouyý")
    'nsmhlskv'
    """
```

Po řádku s příkazem `def` následuje dokumentační řetězec funkce se známou strukturou: na jednom řádku popis, prázdný řádek, podrobnější popis a pak několik příkladů napsaných tak, jako by byly zadávány interaktivně. Řetězce v uvozovkách jsou v dokumentačním řetězci, a proto je nutné buď potlačit v nich uvedená zpětná lomítka, nebo dokumentační řetězec uzavřít do trojitých uvozovek, což jsme udělali my.

Holé řetězce ➤ 72

```
    result = []
    word = ""
    for char in text:
        if char in delete:
            continue
        elif char in whitespace:
            if word:
```

```
                result.append(word)
                word = ""
        else:
            word += char
    if word:
        result.append(word)
    return " ".join(result)
```

Seznam `result` používáme pro uchovávání „slov“, což jsou řetězce, jež nemají mezery nebo vyškrtnuté znaky (parametr `delete`). Zadaný text procházíme po jednotlivých znacích, přičemž přeskakujeme vyškrtnuté znaky. Narazíme-li na znak považovaný za bílé místo (parametr `whitespace`) a současně vytváříme slovo (proměnná `word`), přidáme toto slovo do seznamu `result` a nastavíme jej na prázdný řetězec. V opačném případě bílé místo přeskočíme. Jakýkoliv jiný znak přidáme do vytvářeného slova. Nakonec vrátíme jediný řetězec všech slov v seznamu `result` spojených jedinou mezerou.

Funkce `is_balanced()` je zapsána podobným způsobem: nejdříve je řádek s příkazem `def`, poté dokumentační řetězec s jednořádkovým popisem, prázdný řádek, podrobnější popis, několik příkladů a pak samotný kód. Zde je kód bez dokumentačního řetězce:

```
def is_balanced(text, brackets="()[]{}<>"):
    counts = {}
    left_for_right = {}
    for left, right in zip(brackets[::2], brackets[1::2]):
        assert left != right, "znaky závorek se musejí lišit"
        counts[left] = 0
        left_for_right[right] = left
    for c in text:
        if c in counts:
            counts[c] += 1
        elif c in left_for_right:
            left = left_for_right[c]
            if counts[left] == 0:
                return False
            counts[left] -= 1
    return not any(counts.values())
```

V této funkci sestavujeme dva slovníky. Klíči slovníku `counts` jsou otevírací znaky („(“, „[“, „{“ a „<“) a jeho hodnotami celá čísla. Klíči slovníku `left_for_right` jsou uzavírací znaky („)“, „]“, „}“ a „>“) a jeho hodnotami odpovídající otevírací znaky. Jakmile jsou slovníky připraveny, procházíme text znak po znaku. Jakmile narazíme na otevírací znak, zvýšíme odpovídající počítadlo (slovník `counts`). A podobně, jakmile narazíme na uzavírací znak, vyhledáme odpovídající uzavírací znak. Je-li počítadlo pro tento znak 0, znamená to, že jsme narazili na uzavírací znak bez odpovídajícího otevíracího znaku, a proto okamžitě vrátíme hodnotu `False`. V opačném případě snížíme příslušné počítadlo. Na konci by mělo mít každé počítadlo hodnotu 0, jsou-li všechny dvojice vyvážené, takže pokud je některé z nich nenulové, funkce vrátí hodnotu `False`. Jinak vrátí hodnotu `True`.

Až dosud bylo vše podobné jako v kterémkoliv jiném souboru `.py`. Pokud by byl soubor `TextUtil.py` programem, jistě by obsahoval několik dalších funkcí a na konci by bylo jediné volání některé z těchto funkcí, které by zahájilo zpracování. Avšak vzhledem k tomu, že se jedná o modul, který má být importován, jsou definované funkce dostačující. Nyní může libovolný program či modul importovat modul `TextUtil` a používat jej:

```
import TextUtil

text = " záhadný   hlavolam "
text = TextUtil.simplify(text)     # text == 'záhadný hlavolam'
```

Pokud chceme modul `TextUtil` zpřístupnit určitému programu, musíme soubor `TextUtil.py` umístit do stejného adresáře, ve kterém se nachází daný program. Pokud chceme modul `TextUtil` zpřístupnit všem svým programům, pak máme k dispozici několik možností. Jednou z nich je umístit modul do podadresáře s balíčky v rámci distribuce Pythonu. V systému Windows se obvykle jedná o adresář `C:\Python31\Lib\site-packages`, v případě systémů Mac OS X a Unix však může být jeho umístění různé. Tento adresář se nachází v cestě Pythonu, takže libovolný v něm umístěný modul bude vždy nalezen. Druhá možnost spočívá ve vytvoření adresáře speciálně pro náš vlastní modul, který chceme používat ve všech svých programech, a v nastavení proměnné prostředí `PYTHONPATH` na tento adresář. Třetí možností je umístit modul do *lokálního* podadresáře `site-packages`. Jedná se o adresář, který je umístěn v cestě Pythonu a který má v systému Windows podobu `%APPDATA%\Python\Python31\site-packages` a v systému Unix (včetně Mac OS X) podobu `~/.local/lib/python3.1/site-packages`. Druhá a třetí možnost má tu výhodu, že máme náš kód oddělený od oficiální instalace.

Mít modul `TextUtil` je sice skvělé, ale pokud jej bude používat spousta programů, pak si chceme být jisti, že funguje tak, jak je uvedeno. Jeden skutečně jednoduchý způsob spočívá v provedení příkladů v dokumentačním řetězci a v kontrole, zda obdržíme očekávané výsledky. To můžeme provést přidáním pouhých třech řádků na konec souboru `.py` tohoto modulu:

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

Kdykoliv je nějaký modul importován, Python pro něj vytvoří proměnnou s názvem `__name__` a uloží název modulu do této proměnné. Název modulu je prostý název souboru `.py` bez přípony. V tomto případě bude mít tedy proměnná `__name__` po importu modulu hodnotu `"TextUtil"`, takže podmínka `if` nebude splněna, a proto se poslední dva řádky neprovedou. To znamená, že tyto tři řádky nemají prakticky žádný význam, je-li modul importován.

Kdykoliv je nějaký soubor `.py` spuštěn, Python pro něj vytvoří proměnnou s názvem `__name__` a nastaví ji na řetězec `"__main__"`. Pokud tedy spustíme soubor `TextUtil.py` jako by se jednalo o program, Python nastaví proměnnou `__name__` na hodnotu `"__main__"` a podmínka `if` se vyhodnotí na `True`, takže se poslední dva řádky provedou.

Funkce `doctest.testmod()` použije introspektivní prvky Pythonu pro vyhledání všech funkcí v daném modulu a jejich dokumentačních řetězců a pokusí se provést všechny v nich umístěné úryvky kódu. Když modul takto spustíme, obdržíme výstup pouze v případě výskytu nějakých chyb. To

může být na první pohled znepokojující, protože to vypadá, že se nic nestalo. Pokud ale na příkazovém řádku předáme příznak `-v`, obdržíme výstup podobný následujícímu:

```
Trying:
    is_balanced("(Python (není (jako (lisp))))")
Expecting:
    True
ok
...
Trying:
    simplify(" nesamohláskový ", delete="aáeiouyý")
Expecting:
    'nsmhlskv'
ok
4 items passed all tests:
   3 tests in __main__
   5 tests in __main__.is_balanced
   3 tests in __main__.shorten
   4 tests in __main__.simplify
15 tests in 4 items.
15 passed and 0 failed.
Test passed.
```

Místo množství vynechaných řádků je uveden výpustek. Jsou-li v modulu funkce (nebo třídy či metody), které nemají testy, pak jsou při použití volby `-v` taktéž uvedeny. Všimněte si, že modul `doctest` nalezl testy v dokumentačním řetězci modulu i v dokumentačních řetězcích funkcí.

Příklady v dokumentačních řetězcích, které lze spouštět jako testy, se nazývají *dokumentační testy* (doctests). Je třeba poznamenat, že při psaní dokumentačních testů můžeme zavolat funkci `simplify()` a ostatní funkce bez nutnosti kvalifikace (k dokumentačním testům totiž dochází uvnitř samotného modulu). Vně tohoto modulu musíme za předpokladu, že jsme použili příkaz `import TextUtil`, používat kvalifikované názvy, například `TextUtil.is_balanced()`.

V následující podčásti si ukážeme, jak provádět více důkladných testů – konkrétně testovací případy, v nichž očekáváme selhání, například neplatná data způsobující výjimky. (Testování se budeme podrobně věnovat v lekci 9.) Kromě toho se podíváme na několik dalších problémů, k nimž dochází při vytváření modulu, jako je inicializace modulu, ohled na rozdíly mezi platformami a zajištění, aby se při použití syntaxe `from modul import *` do programu či modulu skutečně importovaly pouze ty objekty, které chceme zveřejnit.

Modul CharGrid

Modul `CharGrid` uchovává v paměti znakovou mřížku. Nabízí funkce pro „kreslení“ čar, obdélníků a textu do mřížky a pro vykreslení této mřížky do konzoly. Zde jsou dokumentační testy z dokumentačních řetězců:

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

Funkce `CharGrid.add_rectangle()` přijímá nejméně čtyři argumenty: řádek a sloupec levého horního rohu a řádek a sloupec pravého spodního rohu. Znak použitý pro nakreslení obrysu lze zadat jako pátý argument a jako šestý argument logickou hodnotu signalizující, zda má být obdélník vyplněn (stejným znakem jako obrys). Při prvním zavolání předáme třetí a čtvrté argumenty rozbalením n-tice se dvěma prvky (šířka, výška) vrácené funkcí `CharGrid.get_size()`.

Funkce `CharGrid.render()` ve výchozím nastavení vymaže obrazovku před vypsáním mřížky, čemuž lze zamezit předáním hodnoty `False`, jak jsme zde učinili také my. Níže je uvedena mřížka, která je výsledkem výše uvedených dokumentačních testů:

```
%%%%%*********************************************
%   %                          @@@@@@@@@@@@@@@  *
%   %                          @Pěkně odporný@  *
%   %     ++++++++++           @@@@@@@@@@@@@@@  *
%%%%%                                           *
*      ###################################      *
*      ###################################  ****   *
*      ##                               ##  ****   *
*      ## Tohle je náš modul CharGrid ##  ****   *
* !    ##                               ##  ****   *
* !  | ###################################  ****   *
* !  | ###################################      *
*    |                                           *
*********************************************
```

Modul začíná stejným způsobem jako modul `TextUtil`: řádkem shebang, komentářem s copyrightem a informacemi o licenci a dokumentačním řetězcem, který popisuje modul a obsahuje výše uvedené dokumentační testy. Řádný kód pak začíná dvěma importy, z nichž prvním je modul `sys` a druhý modul `subprocess`. Modulu `subprocess` se budeme věnovat v lekci 10.

Modul se řídí dvěma zásadami pro ošetření chyb. Několik funkcí definuje parametr `char`, jehož aktuálním argumentem musí být vždy řetězec obsahující přesně jeden znak. Porušení tohoto požadavku je považováno za fatální chybu programování, takže ke kontrole délky používáme příkazy `assert`.

Na druhou stranu předání čísla řádku či sloupce mimo rozsah je považováno za chybové, ale normální, a proto dojde k vyvolání vlastní výjimky.

Nyní prostudujeme několik ilustrativních a klíčových částí kódu tohoto modulu, přičemž začneme vlastními výjimkami:

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

Žádná z funkcí v modulu, která vyvolává výjimku, nikdy nevyvolá chybu `RangeError`, ale vždy vyvolá specifickou výjimku v závislosti na tom, zda hodnota zadaná mimo rozsah představuje řádek nebo sloupec. Avšak tím, že používáme hierarchii, dáváme uživatelům modulu možnost zachytit specifickou výjimku nebo zachytit kteroukoliv z nich zachycením bázové třídy `RangeError`. Všimněte si také, že uvnitř dokumentačních testů používáme názvy výjimek tak, jak jsou zde uvedeny, ale při importu modulu příkazem `import CharGrid` je samozřejmě nutné psát `CharGrid.RangeError`, `CharGrid.RowRangeError` a `CharGrid.ColumnRangeError`.

```
_CHAR_ASSERT_TEMPLATE = ("je nutné zadat jediný znak: '{0}' "
                         "je příliš dlouhý")
_max_rows = 25
_max_columns = 80
_grid = []
_background_char = " "
```

Zde definujeme několik soukromých dat pro interní použití v modulu. Identifikátory začínají podtržítkem, takže pokud se modul importuje příkazem `from CharGrid import *`, neimportuje se žádná z těchto proměnných. (Další možností by bylo zřízení seznamu `__all__`.) Řetězec `_CHAR_ASSERT_TEMPLATE` se používá v souvislosti s metodou `str.format()` pro vytvoření chybové zprávy v příkazech `assert`. K ostatním proměnným se vrátíme, jakmile se dostaneme k jejich použití v kódu.

```
if sys.platform.startswith("win"):
    def clear_screen():
        subprocess.call(["cmd.exe", "/C", "cls"])
else:
    def clear_screen():
        subprocess.call(["clear"])
clear_screen.__doc__ = """Vymaže obrazovku pomocí příkazu pro \
vymazání obrazovky aktuálně používaného systému"""
```

Prostředek k vymazání obrazovky je nezávislý na platformě. V systému Windows musíme spustit program `cmd.exe` s příslušnými argumenty a na většině unixových systémů spustíme program `clear`. Funkce `subprocess.call()` modulu `subprocess` umožňuje spuštění externího programu, takže ji můžeme použít k vymazání obrazovky způsobem vhodným pro používanou platformu. Řetězec `sys.platform` uchovává název operačního systému, na kterém daný program běží (např. „win32“ nebo „linux2“). Jeden ze způsobů pro překlenutí rozdílů mezi platformami tedy spočívá v definování jediné funkce:

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

Nevýhodou tohoto postupu je, že i když víme, že se platforma za běhu programu nezmění, provádíme kontrolu platformy při každém volání funkce.

Abychom zamezili kontrole, na které platformě program běží, při každém zavolání funkce `clear_screen()`, vytvořili jsme funkci `clear_screen()` s ohledem na používanou platformu jedenkrát při importu modulu a od tohoto okamžiku ji vždy používáme. To je možné díky tomu, že příkaz `def` je příkazem jazyka Python úplně stejně jako kterýkoliv jiný. Jakmile interpret dorazí k příkazu `if`, provede buď první, nebo druhý příkaz `def`, čímž se dynamicky vytvoří první nebo druhá funkce `clear_screen()`. Vzhledem k tomu, že tato funkce není definována uvnitř jiné funkce (nebo uvnitř třídy, jak uvidíme v následující lekci), jedná se o globální funkci, která je přístupná stejně jako kterákoli jiná funkce v modulu.

Po vytvoření funkce explicitně nastaví její dokumentační řetězec. Díky tomu nemusíme psát stejný dokumentační řetězec na dvou místech a také vidíme, že dokumentační řetězec je jen jedním z atributů funkce. Mezi další atributy patří modul funkce a její název.

```
def resize(max_rows, max_columns, char=None):
    """Změní velikost mřížky, přičemž zahodí obsah a
    změní pozadí, nemá-li znak představující pozadí hodnotu None
    """
    assert max_rows > 0 and max_columns > 0, "příliš malé"
    global _grid, _max_rows, _max_columns, _background_char
    if char is not None:
        assert len(char) == 1, _CHAR_ASSERT_TEMPLATE.format(char)
        _background_char = char
    _max_rows = max_rows
    _max_columns = max_columns
    _grid = [[_background_char for column in range(_max_columns)]
             for row in range(_max_rows)]
```

Tato funkce používá příkaz `assert` pro vynucení zásady, že pokus o změnu velikosti mřížky na velikost menší než 1 x 1 představuje chybu programování. Je-li zadán znak pro pozadí, použije se příkaz `assert` pro zajištění, že se jedná o řetězec s přesně jedním znakem. Není-li tomu tak, bude chybovou zprávou tvrzení text konstanty `_CHAR_ASSERT_TEMPLATE` se zástupným symbolem `{0}` nahrazeným zadaným řetězcem `char`.

Naneštěstí musíme použít příkaz `global`, protože potřebujeme aktualizovat několik globálních proměnných uvnitř této funkce. Jak uvidíme v lekci 6, v tomto ohledu nám výrazně pomůže objektově orientovaný přístup.

Proměnnou `_grid` vytváříme pomocí seznamové komprehenze uvnitř seznamové komprehenze. Použití replikace seznamu (`[[char] * columns] *`) by zde nefungovalo, protože vnitřní seznam bude sdílený (mělce zkopírovaný). Mohli bychom ale použít vnořené cykly `for ... in`:

Seznamové komprehenze ➢ 120

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

Tento kód je pravděpodobně méně srozumitelný než seznamová komprehenze a je také mnohem delší.

Nyní se podíváme jen na jednu z kreslicích funkcí, abychom získali ponětí o tom, jak se takové kreslení provádí, protože naším hlavním zájmem je implementace modulu. Zde je funkce `add_horizontal_line()` rozdělená na dvě části:

```
def add_horizontal_line(row, column0, column1, char="-"):
    """Přidá do mřížky vodorovnou čáru s použití zadaného znaku

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

Dokumentační řetězec obsahuje dva testy, z nichž první by měl fungovat a druhý by měl vyvolat výjimku. Když v dokumentačních testech pracujeme s výjimkami, stačí uvést řádek „Traceback“, protože ten je vždy stejný a říká modulu `doctest`, že očekáváme výjimku, pak následuje výpustek znamenající další řádky (které se mění) a nakonec řádek s výjimkou, kterou čekáme, že obdržíme. Funkce `char_at()` patří mezi funkce tohoto modulu. Vrací znak na zadaném řádku a sloupci v mřížce.

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

Kód začíná stejnou kontrolou délky argumentu `char` jako ve funkci `resize()`. Místo explicitní kontroly argumentů `row` a `column` funguje funkce na základě předpokladu, že argumenty jsou platné. Pokud dojde k výjimce `IndexError` kvůli přístupu k neexistujícímu řádku nebo sloupci, výjimku zachytíme a vyvoláme místo ní příslušnou výjimku specifickou pro tento modul. Tento způsob programování se neformálně označuje jako „je snazší žádat o prominutí než o dovolení“ a ve srovnání se způsobem „dívej se, než skočíš“, kde se kontroly provádějí předem, je pro jazyk Python obecně považován za vhodnější. Spoléhání na vyvolání výjimky spíše než na předem provedenou kontrolu

je efektivnější v případech, kdy je výskyt výjimek vzácný. (Tvrzení se do oblasti „dívej se, než skočíš“ nepočítají, protože v kódu nasazeném do ostrého provozu by k nim nikdy nemělo dojít – proto jsou také často deaktivovány komentářem.)

Téměř na konci modulu je za definicemi všech funkcí jediné volání funkce `resize()`:

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

Toto volání inicializuje mřížku na výchozí velikost (25 x 80) a zajistí, aby kód, který importuje modul, mohl tento modul bezpečně a okamžitě používat. Bez tohoto volání by importující program či modul musel při každém importu tohoto modulu zavolat funkci `resize()` pro inicializaci mřížky, což by si programátoři museli pamatovat a mohlo by to navíc vést k vícenásobným inicializacím.

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

Poslední tři řádky modulu jsou standardní pro moduly, které používají modul `doctest` pro kontrolu svých dokumentačních testů. (Testování se budeme podrobně věnovat v lekci 9.)

Modul `CharGrid` má jeden podstatný nedostatek: podporuje pouze jediný mřížkový znak. To se dá vyřešit například uchováváním kolekce mřížek v modulu, což by ale znamenalo, že uživatelé modulu budou muset při volání každé funkce uvádět klíč nebo index označující, na kterou mřížku odkazují. V případech, kdy je vyžadováno více instancí nějakého objektu, je lepším řešením vytvořit modul, který definuje třídu (vlastní datový typ), protože můžeme vytvářet tolik instancí třídy (objektů daného datového typu), kolik jen chceme. Další výhoda vytvoření třídy spočívá v tom, že bychom nemuseli pro ukládání (statických) dat třídy používat příkaz `global`. Způsob vytváření tříd si ukážeme v další lekci.

Přehled standardní knihovny Pythonu

Standardní knihovna Pythonu je obecně popisována jako „včetně baterií“. K dispozici je skutečně pestrá paleta funkčních prvků rozprostřených do přibližně dvou stovek balíčků a modulů.

Ve skutečnosti bylo pro jazyk Python v průběhu let vyvinuto tolik vysoce kvalitních modulů, že pokud bychom je všechny začlenili do standardní knihovny, vzrostla by velikost distribučních balíčků Pythonu přinejmenším o jeden řád. Moduly umístěné do této knihovny jsou tedy spíše odrazem historického vývoje Pythonu a zájmů jeho hlavních vývojářů než jakkoli koordinovanou či systematickou snahou o vytvoření „vyvážené“ knihovny. Kromě toho se některé moduly ukázaly jako velmi obtížně udržovatelné v rámci knihovny – zvláště modul Berkeley DB – a proto byly z knihovny odebrány a nyní jsou udržovány nezávisle na ní. To znamená, že pro Python je k dispozici řada skvělých modulů třetích stran, které (bez ohledu na svoji kvalitu a užitečnost) nejsou součástí standardní knihovny. (Na dva takovéto moduly se později podíváme: moduly PyParsing a PLY použijeme v lekci 14 pro vytvoření analyzátorů.)

V této části se podíváme na obsáhlejší přehled toho, co vše máme dispozici. Výklad rozdělíme dle tematických okruhů a vynecháme ty balíčky a moduly, které jsou silně specializované, a také ty, které jsou určeny pro konkrétní platformu. V mnoha případech si ukážeme malý příklad, abychom si mohli

některé balíčky a moduly přímo „osahat“. U balíčků a modulů probíraných v jiných částech knihy jsou k dispozici křížové odkazy.

Práce s řetězci

Modul `string` nabízí několik užitečných konstant, mezi něž patří `string.ascii_letters` a `string.hexdigits`. Dále nabízí třídu `string.Formatter`, z níž můžeme odvodit třídu poskytující přizpůsobené formátování řetězců.* Modul `textwrap` lze použít k zalomení řádků textu na zadanou šířku a k minimalizaci odsazení.

typ bytes ➤ 286

Modul `struct` poskytuje funkce pro zabalení, resp. rozbalení, čísel, logických hodnot a řetězců do (resp. z) objektů typu `bytes` s použitím jejich binární reprezentace. To může být užitečné při práci s daty, která se mají odeslat nebo přijmout z nízkoúrovňových knihoven napsaných v jazyku C. Moduly `struct` a `textwrap` používá program `convert-incidents.py`, na který se podíváme v lekci 7.

modul struct ➤ 288

Modul `difflib` nabízí třídy a metody pro porovnávání posloupností, jako jsou kupříkladu řetězce. Tento modul je schopen vytvořit výstup ve standardních formátech „diff“ i v jazyku HTML.

Nejvýkonnějším modulem Pythonu pro práci s řetězci je modul `re` (regulární výrazy), který si podrobně prostudujeme v lekci 13.

Třída `io.StringIO` nabízí objekty podobné řetězcům, které se chovají jako textový soubor umístěný v paměti. To může být užitečné, chceme-li pro zápis do řetězce použít stejný kód, který zapisuje do souboru.

Příklad: Třída io.StringIO

Python nabízí dva odlišné způsoby zápisu textu do souborů. První spočívá v použití metody `write()` objektu souboru a druhý v použití funkce `print()` s klíčovým argumentem `file` nastaveným na objekt souboru, který je otevřený pro zápis:

```
print("Chybová zpráva", file=sys.stdout)
sys.stdout.write("Chybová zpráva\n")
```

Oba řádky textu se vypíšou do objektu `sys.stdout`, což je objekt souboru, který představuje „standardní výstupní proud“. Tento proud je obvykle směrován do konzoly a od objektu `sys.stderr` („chybový výstupní proud“) se liší pouze v tom, že je ukládán do mezipaměti. (Python automaticky vytváří a otevírá proudy `sys.stdin`, `sys.stdout` a `sys.stderr` při spuštění programu.) Funkce `print()` přidává standardně nový řádek, což můžeme potlačit zadáním klíčového argumentu `end`, který nastavíme na prázdný řetězec.

V některých situacích je užitečné mít možnost zachytit do řetězce výstup, který má jít do nějakého souboru. To lze realizovat pomocí třídy `io.StringIO`, jež poskytuje objekt, který je možné používat stejně jako objekt souboru, přičemž se všechna zapsaná data uchovají v řetězci. Objektu typu `io.StringIO` můžeme předat počáteční řetězec, takže z něj lze také číst jako ze souboru.

* Výraz *odvození třídy* (nebo též *specializace*) se používá tehdy, když vytváříme vlastní datový typ (tj. třídu) založený na jiné třídě. Tomuto tématu se budeme plně věnovat v lekci 6.

Ke třídě `io.StringIO` můžeme přistupovat po importu modulu `io` a můžeme ji použít k zachycení výstupu určeného pro objekt souboru, jako je například `sys.stdout`:

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

Pokud tento řádek umístíme na začátek programu za příkazy import, ale před jakékoliv použití proudu `sys.stdout`, odešle se veškerý text posílaný do proudu `sys.stdout` do objektu `io.StringIO`, který jsme vytvořili na tomto řádku a který nahradil standardní objekt souboru `sys.stdout`. Když se nyní provedou výše uvedené řádky `print()` a `sys.stdout.write()`, nepůjde jejich výstup do konzoly, ale do objektu `io.StringIO`. (Původní proud `sys.stdout` můžeme kdykoliv obnovit příkazem `sys.stdout = sys.__stdout__`.)

Všechny řetězce, které byly zapsány do objektu `io.StringIO`, můžeme získat pomocí funkce `io.StringIO.getvalue()`, což v našem případě znamená voláním funkce `sys.stdout.getvalue()`. Její návratovou hodnotou je řetězec obsahující všechny řádky, které byly dosud zapsány. Tento řetězec můžeme vypsat nebo uložit do souboru protokolu nebo poslat přes síťové připojení jako jakýkoliv jiný řetězec. S dalším příkladem použití třídy `io.StringIO` se setkáme v pozdější části této lekce (strana 233).

Programování na příkazovém řádku

Pokud potřebujeme, aby náš program byl schopen zpracovat text, který mohl být přesměrován z konzoly nebo který mohl být v souborech uvedených na příkazovém řádku, pak můžeme použít funkci `fileinput.input()` modulu `fileinput`. Tato funkce prochází všechny řádky přesměrované z konzoly (jsou-li nějaké) a všechny řádky v souborech uvedených na příkazovém řádku jako spojitou posloupnost řádků. Modul umí prostřednictvím funkcí `fileinput.filename()` a `fileinput.lineno()` ohlásit aktuální název souboru a číslo řádku a dále dokáže pracovat s některými typy komprimovaných souborů.

Pro práci s volbami příkazového řádku jsou k dispozici dva samostatné moduly. Modul `getopt` je populární, protože jej lze jednoduše používat a je již delší dobu součástí knihovny. Modul `optparse` je novější a výkonnější.

Příklad: Modul optparse

příklad `csv2html.py` ➤ 100

V lekci 2 jsme si popsali program `csv2html.py`. Ve cvičeních druhé lekce jsme navrhli rozšíření tohoto programu spočívající v přijímání argumentů z příkazového řádku („maxwidth“ jako celé číslo a „format“ jako řetězec). Modelové řešení (`csv2html2_ans.py`) má pro zpracování argumentů funkci o velikosti 26 řádků. Zde je začátek funkce `main()` programu `csv2html2_opt.py`, což je další verze programu, která pro zpracování argumentů příkazového řádku nepoužívá vlastní funkci, ale modul `optparse`:

```
def main():
    parser = optparse.OptionParser()
    parser.add_option("-w", "--maxwidth", dest="maxwidth", type="int",
            help=("maximální počet znaků, které lze "
                  "vypsat do řetězcových polí [výchozí: %default]"))
    parser.add_option("-f", "--format", dest="format",
```

```
        help=("formát pro výpis čísel "
              "[výchozí: %default]"))
parser.set_defaults(maxwidth=100, format=".0f")
opts, args = parser.parse_args()
```

Stačí pouze devět řádků kódu plus příkaz import `optparse`. Kromě toho nepotřebujeme explicitně řešit volby `-h` a `--help`, o něž se postará modul `optparse`, který vypíše vhodnou zprávu o použití programu pomocí textů z klíčových argumentů help, v nichž nahradí text „%default" výchozí hodnotou příslušné volby.

Dále si všimněte, že volby nyní používají standardní unixový styl krátkých a dlouhých názvů začínajících pomlčkou. Krátké názvy jsou vhodné pro interaktivní použití v konzole, zatímco dlouhé názvy jsou srozumitelnější při použití ve skriptech shellu. Například pro nastavení maximální šířky na 80 můžeme použít `-w80`, `-w 80`, `--maxwidth=80` nebo `--maxwidth 80`. Po analýze příkazového řádku jsou volby k dispozici prostřednictvím atributů s příslušnými názvy, například `opts.maxwidth` a `opts.format`. Všechny argumenty, které nebyly zpracovány (obvykle názvy souborů), jsou v seznamu `args`.

Pokud během analýzy příkazového řádku dojde k nějaké chybě, zavolá modul `optparse` funkci `sys.exit(2)`. To vede k čistému ukončení programu, při němž operační systém obdrží jako návratovou hodnotu programu číslo 2. Návratová hodnota 2 standardně označuje chybu při použití, 1 označuje jakýkoliv jiný druh chyby a 0 znamená úspěch. Pokud funkci `sys.exit()` zavoláme bez argumentů, vrátí operačnímu systému hodnotu 0.

Matematika a čísla

Kromě vestavěných čísel typu `int`, `float` a `complex` nabízí knihovna čísla typu `decimal.Decimal` a `fractions.Fraction`. K dispozici máme tři numerické knihovny: `math` pro standardní matematické funkce, `cmath` pro matematické funkce pracující s komplexními čísly a `random` poskytující řadu funkcí pro generování náhodných čísel. Tyto moduly jsme si představili v lekci 2.

Numerické abstraktní bázové třídy jazyka Python (tj. třídy, od nichž lze odvozovat další třídy, ale které nelze použít přímo) se nacházejí v modulu `numbers`. Tyto třídy jsou užitečné pro kontrolu, zda daný objekt `x` představuje libovolný druh čísla (např. `isinstance(x, numbers.Number)`) nebo zda je specifickým druhem čísla (např. `isinstance(x, numbers.Rational)` nebo `isinstance(x, numbers.Integral)`).

V oblasti vědeckého a inženýrského programování přijde vhod balíček třetí strany s názvem `NumPy`. Tento modul nabízí vysoce efektivní n-dimenzionální pole, základní funkce z oblasti lineární algebry a Fourierových transformací a nástroje pro integraci kódu napsaného v jazyku C, C++ a Fortran. Balíček `SciPy` obsahuje modul `NumPy`, který rozšiřuje o moduly pro statistické výpočty, zpracování signálu a obrázků, genetické algoritmy a pro řadu dalších oblastí. Oba balíčky jsou k dispozici zdarma na adrese *www.scipy.org*.

Datum a čas

Moduly `calendar` a `datetime` nabízejí funkce a třídy pro práci s datem a časem. Jsou ovšem založeny na idealizovaném gregoriánském kalendáři, takže nejsou vhodné pro práci s předgregoriánskými kalendářními daty. Práce s datem a časem je velice složité téma. Používané kalendáře se v jednotli-

vých místech a časech liší, den netrvá přesně 24 hodin, rok nemá přesně 365 dnů, odlišný je též letní čas i časová pásma. Třída `datetime.datetime` (ne však třída `datetime.date`) si sice dokáže poradit s časovými pásmy, ale ne sama od sebe. Tento nedostatek napravují moduly třetích stran, například `dateutil` z *www.labix.org/python-dateutil* a `mxDateTime` z *www.egenix.com/products/python/mx-Base/mxDateTime*.

Modul `time` pracuje s časovými známkami. Jedná se o prostá čísla, která uchovávají počet vteřin od začátku jisté epochy (1. 1. 1970 00:00:00 na Unixu). Tento modul lze použít pro získání časové známky aktuálního času na daném stroji v UTC (Coordinated Universal Time – koordinovaný světový čas) nebo jako místní čas, který bere v potaz letní čas. Dále jej lze použít pro vytváření nejrůznějším způsobem naformátovaných řetězců s datem, časem a datem i časem. Kromě toho dokáže analyzovat řetězce obsahující kalendářní data a časy.

Příklad: Moduly calendar, datetime a time

Objekty typu `datetime.datetime` se obvykle vytvářejí programově, kdežto objekty, jež uchovávají data a časy v UTC, se obvykle získávají z externích zdrojů, jako jsou časové známky souborů. Zde je několik příkladů:

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

Proměnná `moon_datetime_a` je typu `datetime.datetime` a uchovává datum a čas přistání vesmírné lodi Apollo 11 na Měsíci. Proměnná `moon_time` je typu in a uchovává počet vteřin od začátku epochy po přistání na Měsíci – toto číslo poskytuje funkce `calendar.timegm()`, která přijímá objekt typu `time_struct` získaný od funkce `datetime.datetime.utctimetuple()` a vrací počet vteřin reprezentovaných objektem typu `time_struct`. (K přistání na Měsíci došlo před unixovou epochou, a proto je výsledné číslo záporné.) Proměnná `moon_datetime_b` je typu `datetime.datetime` a vytváříme ji z celého čísla `moon_time`, abychom si ukázali převod z počtu vteřin od začátku epochy na objekt typu `datetime.datetime`.* Poslední tři řádky vracejí identické řetězce s datem a časem ve formátu ISO 8601.

Aktuální datum a čas v UTC je k dispozici jako objekt typu `datetime.datetime` prostřednictvím funkce `datetime.datetime.utcnow()` a jako počet vteřin od počátku epochy prostřednictvím funkce `time.time()`. Pro místní datum a čas použijte `datetime.datetime.now()` nebo `time.mktime(time.localtime())`.

* Pro uživatele systému Windows je nutné podotknout, že funkce `datetime.datetime.utcfromtimestamp()` neumí zpracovat záporné časové známky, což znamená časové známky pro data před 1. lednem 1970.

Algoritmy a datové kolekce představující kolekce

Modul `bisect` nabízí funkce pro prohledávání seřazených posloupností, jako jsou seřazené seznamy, a pro vkládání prvků nenarušující jejich seřazené pořadí. Funkce tohoto modulu používají algoritmus binárního hledání, takže jsou velice rychlé. Modul `heapq` nabízí funkce pro převod posloupnosti (např. seznamu) na haldu, což je datový typ představující kolekci, v němž je první prvek (na indexové pozici 0) vždy nejmenším prvkem. Dále nabízí funkce pro vkládání a odstraňování prvků při zachování posloupnosti ve stavu haldy.

Výchozí slovníky ➤ 135

Balíček `collections` nabízí slovník `collections.defaultdict` a datový typ představující kolekci s názvem `collections.namedtuple`, který jsme probírali již dříve. Kromě toho nabízí typy `collections.UserList` a `collections.UserDict`, ačkoliv častěji se třídy odvozují od vestavěných typů `list` a `dict`. Dalším typem je `collections.deque`, který je podobný seznamu, avšak zatímco seznam je velmi rychlý při přidávání a odstraňování prvků na konci, typ `collections.deque` je velmi rychlý při přidávání a odstraňování prvků na začátku i na konci.

Pojmenovaná n-tice ➤ 113

Uspořádané slovníky ➤ 136

Python 3.1 zavádí třídy `collections.OrderedDict` a `collections.Counter`. Třída `OrderedDict` má stejné rozhraní API jako běžný typ `dict`, avšak při iteraci jsou prvky vždy vraceny ve vkládaném pořadí (tj. od prvního po poslední vložený prvek) a metoda `popitem()` vždy vrátí naposledy přidaný (tj. poslední) prvek. Třída `Counter` je podtřídou typu `dict` a poskytuje rychlý a snadný způsob udržování nejrůznějších počítadel. Při zadání iterovatelného prvku nebo mapování (např. slovníku) dokáže instance třídy `Counter` například vrátit seznam jedinečných prvků nebo seznam nejčastějších prvků jako dvojice (prvek, počet).

V balíčku `collections` se nacházejí také nenumerické abstraktní bázové třídy jazyka Python (tj. třídy, od nichž lze odvozovat další třídy, ale které nelze použít přímo). Budeme se jim věnovat v lekci 8.

Modul `array` nabízí typ představující posloupnost s názvem `array.array`, který dokáže uchovávat čísla nebo znaky velice efektivním způsobem s ohledem na potřebný prostor v paměti. Chová se podobně jako seznam, tedy až na to, že možný typ uchovávaných objektů je zafixován při jeho vytvoření, takže na rozdíl od seznamů nedokáže uchovávat objekty odlišných typů. Výše zmíněný balíček třetí strany `NumPy` nabízí také efektivní pole.

Modul `weakref` nabízí funkční prvky pro vytváření slabých odkazů. Ty se chovají stejně jako běžné odkazy na objekty, ovšem s tím, že pokud jediným odkazem na daný objekt je slabý odkaz, může být tento objekt i tak naplánován na úklid z paměti. Díky tomu není nutné udržovat v paměti objekty jen proto, že na ně máme nějaký odkaz. Přirozeně lze zkontrolovat, zda objekt, na který slabý odkaz ukazuje, stále existuje, a v případě že ano, tak k tomuto objektu přistupovat.

Příklad: Modul heapq

Modul `heapq` nabízí funkce pro převod seznamu na haldu a pro přidávání a odstraňování prvků z haldy při zachování *vlastnosti haldy*. Halda je binární strom, který respektuje vlastnost haldy, která spočívá v tom, že první prvek (na indexové pozici 0) je vždy nejmenší.* Každý podstrom haldy je též haldou, takže také respektuje vlastnost haldy. Zde je příklad vytvoření nové haldy:

* Přesněji řečeno, modul `heapq` nabízí *minimální haldu*. Haldy, v nichž je první prvek vždy největší, se označují jako *maximální haldy*.

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

Máme-li seznam, můžeme jej převést na haldu pomocí funkce `heapq.heapify(`*`seznam`*`)`, která provede veškerá nezbytná přeuspořádání přímo v zadaném seznamu. Nejmenší prvek lze poté odstranit z haldy pomocí funkce `heapq.heappop(`*`halda`*`)`.

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

Funkce `heapq.merge()` přijímá jako argumenty libovolný počet seřazených iterovatelných objektů a vrací iterátor, který prochází všechny prvky ze všech iterovatelných objektů seřazené ve správném pořadí.

Souborové formáty, kódování a perzistence dat

Kódování znaků ➤ 95

Standardní knihovna obsahuje rozsáhlou podporu pro nejrůznější standardní souborové formáty a kódování. Modul base64 má funkce pro čtení a zapisování s použitím kódování Base16, Base32 a Base64 specifikovaných v dokumentu RFC 3548.* Modul `quopri` obsahuje funkce pro čtení a zapisování formátu označovaného jako „quoted-printable". Tento formát je definován v dokumentu RCF 1521 a používá se pro data spadající do rozšíření MIME (Multipurpose Internet Mail Extensions – víceúčelová rozšíření internetové pošty). Modul `uu` má funkce pro čtení a zapisování dat v kódování označovaném jako uuencoding („Unix-to-Unix encoding"). Dokument RFC 1832 definuje standard externí reprezentace dat (External Data Representation Standard) a modul `xdrlib` poskytuje funkce pro čtení a zapisování data v tomto formátu.

K dispozici jsou též moduly pro čtení a zapisování archivních souborů ve většině oblíbených formátů. Modul `bz2` umí pracovat se soubory `.bz2`, modul `gzip` se soubory `.gz`, modul `tarfile` se soubory `.tar`, `.targ.gz` (také `.tgz`) a `.tar.bz2` a modul `zipfile` si poradí se soubory `.zip`. V této podčásti si ukážeme příklad použití modulu `tarfile` a později (strana 223) se podíváme na malý příklad, který používá modul `gzip`. Tento modul uvidíme opět v akci v lekci 7.

K dispozici je též podpora pro práci s některými formáty audia. Modul `aifc` podporuje formát AIFF (Audio Interchange File Format – výměnný souborový formát zvuku) a modul `wave` umí pracovat s nekomprimovanými soubory `.wav`. S určitými druhy audiodat lze manipulovat pomocí modulu `audioop` a modul `sndhdr` nabízí několik funkcí pro zjištění, jaký druh zvukových dat je v daném souboru uložen a jaké jsou některé z jeho vlastností, jako je například vzorkovací frekvence.

Formát pro konfigurační soubory (podobný souborům `.ini` ze starých Windows) specifikuje dokument RFC 822, přičemž funkce pro čtení a zapisování těchto souborů nabízí modul `configparser`.

---

* Dokumenty RFC (Request for Comments – žádost o komentáře) se používají pro specifikaci nejrůznějších internetových technologií. Každý má jedinečné identifikační číslo a z mnoha z nich se staly oficiálně přijímané standardy.

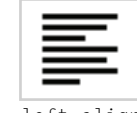
Řada aplikací (např. Excel) dokáže číst a zapisovat data ve formátu CSV (Comma Separated Value – hodnoty oddělené čárkou) nebo jeho varianty, jako jsou kupříkladu data oddělená tabulátorem. Tyto formáty dokáže číst a zapisovat modul `csv`, který umí zohlednit idiosynkrazie, jež znemožňují jejich přímé zpracování.

Standardní knihovna obsahuje kromě podpory rozličných souborových formátů také balíčky a moduly, jež poskytují perzistenci dat. Modul `pickle` se používá pro, ukládání resp., načítání jakýchkoli objektů Pythonu (včetně celých kolekcí) do (resp. ze) souborů na disku. Tento modul budeme podrobně probírat v lekci 7. Knihovna dále podporuje nejrůznější typy souborů DBM. Jedná se o soubory podobné slovníkům, jejichž prvky nejsou uloženy v paměti, ale na disku a jejichž klíče a hodnoty musejí být objekty typu `bytes` nebo řetězce. Modul `shelve`, kterému se budeme věnovat v lekci 12, lze použít pro soubory DBM s řetězcovými klíči a libovolnými objekty Pythonu jako hodnotami. Tento modul v pozadí převádí objekty Pythonu na objekty typu `bytes` a zpět. Moduly pro soubory DBM, databázové rozhraní API Pythonu a použití vestavěné databáze SQLite budeme probírat v lekci 12.

Příklad: Modul base64

Modul base64 se nejčastěji používá pro práci s binárními daty vkládanými do e-mailů jako text v kódování ASCII. Můžeme jej použít také pro ukládání binárních dat do souborů `.py`. Prvním krokem je převod binárních dat do formátu Base64. V tomto příkladu předpokládáme, že modul `base64` již byl importován, a že cesta je společně s názvem souboru `.png` uložena v proměnné `left_align_png`:

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



left_align.png

typ `bytes` ➤ 286

V tomto úryvku kódu čteme soubor v binárním režimu a převádíme jej do řetězce Base64 tvořeného znaky z kódování ASCII. Za každý šedesátý osmý znak přidáváme kombinaci zpětného lomítka a znaku nového řádku. Tím omezíme šířku řádků na 68 znaků ASCII. Při opětovném čtení dat se tyto nové řádky budou ignorovat (protože zpětné lomítko je potlačí). Takto získaný text ASCII můžeme uložit jako literál typu `bytes` do souboru `.py`:

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

Většinu řádků jsme zde vynechali a nahradili výpustkem. Tato data lze převést zpět do původní binární podoby takto:

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

Tato binární data bychom mohli zapsat do souboru pomocí příkazu `open(`*`název_souboru`*`, "wb").write(binary)`. Ukládání binárních dat do souborů `.py` není tak kompaktní jako v jejich původní

podobě, může to však být užitečné v situaci, kdy potřebujeme napsat program jako jediný soubor `.py` vyžadující určitá binární data.

Příklad: Modul tarfile

Většina verzí systému Windows se nedodává s podporou formátu `.tar`, který je na unixových systémech velmi rozšířen. Tento nedostatek lze snadno vyřešit pomocí modulu Pythonu s názvem `tarfile`, který dokáže vytvářet a rozbalovat archivy `.tar` a `.targ.gz` (označované jako *archivy tarball*) a s nainstalovanými správnými knihovnami i archivy `.tar.bz2`. Program `untar.py` umí rozbalovat archivy tarball pomocí modulu `tarfile`. Zde si ukážeme pouze některé jeho klíčové části. Začneme prvním příkazem `import`:

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

Modul `bz2` se používá pro práci s kompresním formátem bzip2. Jeho importování selže, pokud byl Python sestaven bez přístupu ke knihovně bzip2. (Binární distribuce Pythonu pro Windows se vždy sestavuje s vestavěnou kompresí bzip2. Pouze některá unixová sestavení mohou tuto knihovnu postrádat.) Počítáme s možností, že tento modul nemusí být k dispozici, a proto používáme blok `try ... except` a logickou proměnnou, na niž se můžeme později odkázat (ačkoliv kód, v němž se na ni odkazujeme, si zde neukážeme).

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

Tento příkaz vytváří n-tici `('/', '\', 'A:', 'B:', ..., 'Z:', 'a:', 'b:', ..., 'z:')`. Jakýkoliv soubor rozbalovaný z archivu tarball, jehož název začíná jedním z těchto prefixů, je podezřelý. Archivy tarball by totiž neměly obsahovat absolutní cesty, protože tím bychom riskovali přepsání systémových souborů. Jako opatření proto nerozbalíme žádný z takto identifikovaných souborů.

```
def untar(archive):
    tar = None
    try:
        tar = tarfile.open(archive)
        for member in tar.getmembers():
            if member.name.startswith(UNTRUSTED_PREFIXES):
                print("nedůvěryhodný prefix, ignoruji", member.name)
            elif ".." in member.name:
                print("podezřelá cesta, ignoruji", member.name)
            else:
                tar.extract(member)
                print("unpacked", member.name)
    except (tarfile.TarError, EnvironmentError) as err:
        error(err)
```

```
    finally:
        if tar is not None:
            tar.close()
```

Každý soubor v archivu tarball se nazývá *člen*. Funkce `tar.getmembers()` vrací seznam objektů typu `tarfile.TarInfo`, jeden pro každý člen. Název souboru člena včetně jeho cesty je uložen v atributu `tarfile.TarInfo.name`. Pokud tento název začíná nedůvěryhodným prefixem nebo pokud ve své cestě obsahuje „..“, vypíšeme chybovou zprávu. V opačném případě zavoláme metodu `tar.extract()`, která uloží daný člen na disk. Modul `tarfile` má vlastní skupinu výjimek. My jsme však postupovali co nejjednodušeji, takže při výskytu jakékoli výjimky vypíšeme chybovou zprávu a ukončíme program.

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

Pro úplnost zde uvádíme také funkci `error()`. Funkce `main()` (kterou jsme si zde neukázali) vypíše při zadání volby `-h` nebo `--help` zprávu o použití programu. V opačném případě provede před zavoláním funkce `untar()` s názvem souboru archivu tarball několik základních kontrol.

Práce se soubory, adresáři a procesy

Modul `shutil` nabízí vysokoúrovňové funkce pro práci se soubory a adresáři, mezi něž patří funkce `shutil.copy()` a `shutil.copytree()` pro kopírování souborů a celých adresářových stromů, `shutil.move()` pro přesouvání adresářových stromů a `shutil.rmtree()` pro odstraňování celých (i neprázdných) adresářových stromů.

Dočasné soubory a adresáře by se měly vytvářet pomocí modulu `tempfile`, který poskytuje nezbytné funkce (např. `tempfile.mkstemp()`) a dočasné prostředky vytváří tím nejbezpečnějším možným způsobem.

Funkce `filecmp.cmp()` z modulu `filecmp` lze použít k porovnání souborů a funkci `filecmp.cmpfiles()` můžeme použít k porovnání celých adresářů.

Jednou z oblastí, kde jsou programy Pythonu skutečně silné a efektivní, je organizování běhu jiných programů. K tomuto účelu slouží modul `subprocess`, který umí spouštět jiné procesy, komunikovat s nimi pomocí kanálů (pipes) a získávat jejich výsledky. Tomuto modulu se budeme věnovat v lekci 10. Ještě výkonnější alternativa spočívá v použití modulu `multiprocessing`, který nabízí rozsáhlé prostředky pro rozložení práce na více procesů a pro shromáždění výsledků a který lze často použít jako alternativu k vícevláknovému zpracování.

Modul `os` nabízí na platformě nezávislý přístup k funkčním prvkům operačního systému. Proměnná `os.environ` uchovává mapovací objekt, jehož prvky tvoří názvy proměnných prostředí s jejich hodnotami. Pracovní adresář programu poskytuje funkce `os.getcwd()` a pro jeho změnu slouží funkce `os.chdir()`. Modul dále nabízí funkce pro nízkoúrovňovou práci s popisovači souborů. Funkci `os.access()` lze použít pro zjištění, zda daný soubor existuje nebo zda je možné z něj číst nebo do něj zapisovat, a funkce `os.listdir()` vrací seznam záznamů (např. souborů a adresářů, avšak bez

záznamů „.“ a „..“) v zadaném adresáři. Funkce `os.stat()` vrací nejrůznější údaje o zadaném souboru či adresáři, jako je jeho režim, čas přístupu a velikost.

Adresáře lze vytvářet pomocí funkce `os.mkdir()` nebo v případě tvorby přechodových adresářů také pomocí funkce `os.makedirs()`. Prázdné adresáře lze odstranit funkcí `os.rmdir()` a adresářový strom obsahující pouze prázdné adresáře odstraníme funkcí `os.removedirs()`. Soubory či adresáře můžeme odstranit funkcí `os.remove()` a přejmenovat funkcí `os.rename()`.

Funkce `os.walk()` prochází celý adresářový strom a poskytuje postupně názvy všech souborů a adresářů.

Modul os nabízí také řadu nízkoúrovňových funkcí specifických pro určitou platformu, například pro práci s popisovači a pro unixová systémová volání typu `fork`, `spawn` a `exec`.

Zatímco modul `os` nabízí funkce pro interakci s operačním systémem, zejména pak v kontextu souborového systému, modul `os.path` poskytuje směsici prvků pro manipulaci s řetězci (představující cesty) a několik funkcí pro pohodlnější práci se souborovým systémem. Funkce `os.path.abspath()` vrací absolutní cestu k zadanému argumentu s tím, že se odstraní redundantní oddělovače cest a prvky „..“. Funkce `os.path.split()` vrátí dvojici, v níž první prvek obsahuje cestu a druhý název souboru (který může být prázdný, je-li zadána cesta bez názvu souboru). Tyto dvě části jsou též přístupné přímo prostřednictvím funkcí `os.path.basename()` a `os.path.dirname()`. Název souboru lze také rozdělit na dvě části, název a příponu, pomocí funkce `os.path.splitext()`. Funkce `os.path.join()` přijímá libovolný počet řetězců představujících cestu a vrací jedinou cestu používající oddělovač cesty specifický pro aktuální platformu.

Pokud potřebujeme více údajů o souboru či adresáři, můžeme použít funkci `os.stat()`. Pokud nám ale stačí jen jediný údaj, můžeme sáhnout po příslušné funkci modulu `os.path`, například `os.path.exists()`, `os.path.getsize()`, `os.path.isfile()` nebo `os.path.isdir()`.

Modul `mimetypes` obsahuje funkci `mimetypes.guess_type()`, která se pokusí odhadnout typ MIME zadaného souboru.

Příklad: Moduly os a os.path

V tomto příkladu si ukážeme, jak pomocí modulů `os` a `os.path` vytvořit slovník se všemi soubory na zadané cestě, v němž je každý klíč názvem souboru (včetně cesty) a každá hodnota časovou známkou (vteřiny od začátku epochy) poslední modifikace příslušného souboru, a to pro:

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

Tento kód je docela jednoduchý, lze jej ale použít pouze pro soubory nebo jediný adresář. Pokud potřebujeme projít celý adresářový strom, můžeme sáhnout po funkci `os.walk()`.

Zde je úryvek kódu z programu `finddup.py`.* Vytváříme v něm slovník, v němž je každý klíč dvojicí (velikost souboru, název souboru) s tím, že název souboru neobsahuje cestu, a každá hodnota seznamem úplných názvů souborů, které se shodují s názvem souboru svého klíče a mají stejnou velikost:

```
data = collections.defaultdict(list)

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

V každém adresáři vrátí funkce `os.walk()` kořen a dva seznamy, jeden s podadresáři v zadaném adresáři a druhý se soubory v tomto adresáři. Pro získání úplné cesty pro daný název souboru stačí zkombinovat kořen a název souboru. Všimněte si, že pro zanoření do podadresářů nemusíme používat rekurzi – tu za nás obstará funkce `os.walk()`. Jakmile vygenerujeme data, můžeme je projít a vytvořit zprávu ohledně možných duplicitních souborů:

```
for size, filename in sorted(data):
    names = data[(size, filename)]
    if len(names) > 1:
        print("{filename} ({size} bajtů) může být duplicitní "
              "({0} souborů):".format(len(names), **locals()))
        for name in names:
            print("\t{0}".format(name))
```

Klíči slovníku jsou n-tice (velikost, název soubor), a proto pro získání seřazení dat podle velikosti nemusíme použít klíčovou funkci. Má-li kterákoli n-tice (velikost, název souboru) ve svém seznamu více než jeden název souboru, může jít o duplicity.

```
...
shell32.dll (8460288 bajtů) může být duplicitní (2 soubory):
\windows\system32\shell32.dll
\windows\system32\dllcache\shell32.dll
```

Jedná se o poslední prvek z 3282 řádků výstupu vytvořeného spuštěním programu `finddup.py \ windows` v systému Windows.

Sítě a Internet

Balíčky a moduly pro práci se sítí a Internetem jsou hlavní součástí standardní knihovny Pythonu. Na nejnižší úrovni je modul `socket`, který poskytuje nejzákladnější síťové funkční prvky s funkcemi pro vytváření soketů, vyhledávání v systémech DNS (Domain Name System – systém doménových jmen) a zpracování IP adres (Internet Protocol – internetový protokol). Šifrované a autentizované

* Mnohem sofistikovanější program pro hledání duplicit s názvem `findduplicates-t.py`, který používá více vláken a kontrolu MD5, si ukážeme v lekci 10.

sokety lze zřizovat pomocí modulu `ssl`. Modul `socketserver` poskytuje servery TCP (Transmission Control Protocol – protokol pro řízení přenosu) a UDP (User Datagram Protocol – uživatelský datagramový protokol). Tyto servery dokážou zpracovat požadavky přímo nebo mohou pro zpracování každého požadavku vytvořit samostatný proces (rozvětvením procesu) nebo samostatné vlákno. Asynchronní zpracování soketů na straně klienta i serveru lze realizovat pomocí modulu `asyncore` a také vysokoúrovňového modulu `asynchat`, který je postaven na modulu `asyncore`.

Python definuje rozhraní WSGI (Web Server Gateway Interface – rozhraní brány webového serveru) poskytující standardní rozhraní mezi webovými servery a webovými aplikacemi napsanými v Pythonu. K podpoře tohoto standardu nabízí balíček `wsgiref` referenční implementaci rozhraní WSGI, která obsahuje moduly pro poskytování serverů HTTP splňující standard WSGI a pro zpracování hlaviček odpovědi a skriptů CGI (Common Gateway Interface – obecné rozhraní brány). Kromě toho modul `http.server` nabízí server HTTP, kterému lze předat obsluhu požadavků (k dispozici je též jedna standardní) pro spouštění skriptů CGI. Moduly `http.cookies` a `http.cookiejar` poskytují funkce pro správu souborů cookie a moduly `cgi` a `cgitb` nabízejí podporu skriptů CGI.

Klientský přístup k požadavkům protokolu HTTP nabízí modul `http.client`, i když balíček na vyšší úrovni `urllib` obsahuje moduly `urllib.parse`, `urllib.request`, `urllib. response`, `urllib.error` a `urllib.robotparser`, které poskytují snadný a pohodlnější přístup k adresám URL. Stažení souboru z Internetu tedy může být takto jednoduché:

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

Funkce `urllib.request.urlopen()` vrací objekt, který se chová podobně jako objekt souboru otevřený v režimu binárního čtení. Zde získáváme z webu Pythonu soubor `index.html` (jako objekt typu `bytes`) a ukládáme jej jako řetězec do proměnné `html`. Pomocí funkce `urllib.request.urlretrieve()` lze stáhnout soubory a uložit je do místních souborů.

Dokumenty HTML a XHTML je možné analyzovat pomocí modulu `html.parser`, adresy URL analyzujeme a vytváříme pomocí modulu `urllib.parse` a soubory `robots.txt` lze analyzovat modulem `urllib.robotparser`. Data ve formátu JSON (JavaScript Object Notation – objektová notace JavaScriptu) můžeme číst a zapisovat prostřednictvím modulu `json`.

Knihovna nabízí kromě podpory serveru a klienta HTTP také podporu pro volání XML-RPC (Remote Procedure Call – vzdálené volání procedury), kterou zajišťují moduly `xmlrpc.client` a `xmlrpc.server`. Funkčnost na straně klienta pro protokol FTP (File Transfer Protocol – protokol pro přenos souborů) poskytuje modul `ftplib`, pro protokol NNTP (Network News Transfer Protocol – přenosový protokol pro síťové diskuzní skupiny) modul `nntplib` a pro protokol TELNET modul `telnetlib`.

Modul `smtp` nabízí server SMTP (Simple Mail Transfer Protocol – jednoduchý protokol pro přenos pošty) a pro e-mailové klienty slouží moduly `smtplib` pro protokol SMTP, `imaplib` pro protokol IMAP4 (Internet Message Access Protocol – protokol pro přístup k internetové poště) a `poplib` pro protokol POP3 (Post Office Protocol – protokol pro příjem pošty). K poštovním schránkám v nejrůznějších formátech lze přistupovat pomocí modulu `mailbox`. Jednotlivé zprávy (včetně několikadílných zpráv) lze vytvářet a upravovat prostřednictvím modulu `email`.

Jsou-li pro tuto oblast balíčky a moduly standardní knihovny nedostatečně, je zde ještě projekt Twisted (*www.twistedmatrix.com*), který nabízí ucelenou síťovou knihovnu třetí strany. K dispozici je také řada knihoven pro webové programování třetích stran, mezi něž patří například Django (*www.djangoproject.com*) a Turbogears (*www.turbogears.org*) pro tvorbu webových aplikací a dále knihovny Plone (*www.plone.org*) a Zope (*www.zope.org*), které poskytují kompletní webové rámce a systémy pro správu obsahu. Všechny tyto knihovny jsou napsány v Pythonu.

XML

Dokumenty XML se obvykle analyzují dvěma způsoby. Jeden z nich je založen na modelu DOM (Document Object Model – objektový model dokumentu) a druhý na rozhraní SAX (Simple API for XML – jednoduché rozhraní API pro XML). K dispozici jsou dva analyzátory modelu DOM, jeden v modulu `xml.dom` a druhý v modulu `xml.dom.minidom`. Analyzátor pro rozhraní SAX nabízí modul `xml.sax`. Modul `xml.sax.saxutils` jsme již použili kvůli funkci `xml.sax.saxutils.escape()` (pro zakódování znaků „&“, „<“ a „>“ pro dokument XML). K dispozici je též funkce `xml.sax.saxutils.quoteattr()`, která provádí to stejné, ale navíc zakóduje uvozovky (aby byl zadaný text vhodný pro atributy značek), a funkce `xml.sax.saxutils.unescape()`, která provádí obrácený převod (dekódování).

Dále jsou zde dva další analyzátory. Modul `xml.parsers.expat` lze použít pro analýzu dokumentů XML pomocí knihovny expat, ovšem za předpokladu, že je tato knihovna k dispozici, a prostřednictvím modulu `xml.etree.ElementTree` můžeme analyzovat dokumenty XML s použitím jistého druhu rozhraní ve stylu slovníku a seznamu. (Samotné analyzátory modelu DOM a elementových stromů standardně používají analyzátor `expat`.)

Ručnímu zapisování kódu XML a zapisování kódu XML pomocí modelu DOM a elementových stromů a analyzování pomocí analyzátorů pro model DOM, rozhraní SAX a elementové stromy se budeme věnovat v lekci 7.

K dispozici je také knihovna třetí stránky lxml (*www.codespeak.net/lxml*), která o sobě tvrdí, že je „knihovnou s nejbohatší výbavou a nejsnadnějším používáním určenou pro práci s XML a HTML v jazyku Python“. Tato knihovna nabízí rozhraní, které je v podstatě nadmnožinou toho, co poskytuje modul pro práci s elementovými stromy, společně s dalšími prvky, mezi něž patří podpora pro jazyk XPath, XSLT a řadu dalších technologií kolem jazyka XML.

Příklad: Modul xml.etree.ElementTree

Analyzátory pro model DOM a rozhraní SAX poskytují rozhraní API, na která jsou zkušení programátoři pracující s jazykem XML zvyklí. Modul `xml.etree.ElementTree` naproti tomu nabízí přístup k analýze a zápisu kódu jazyka XML, který více odpovídá povaze programování v jazyku Python. Modul pro práci s elementovými stromy je poměrně nedávným přírůstkem do standardní knihovny*, a proto může být některým čtenářům neznámý. Z tohoto důvodu si zde ukážeme velice krátký příklad, abychom měli ponětí, o co vlastně jde. Lekce 7 nabízí mnohem hutnější příklad a pro srovnání také kód používající model DOM a rozhraní SAX.

* Modul `xml.etree.ElementTree` se poprvé objevil v Pythonu 2.5.

Webová stránka americké vládní organizace NOAA (National Oceanic and Atmospheric Administration – Národní úřad pro výzkum oceánů a atmosféry) poskytuje pestrou škálu dat, včetně souboru XML, který uvádí meteorologické stanice ve Spojených státech. Tento soubor obsahuje více než 20 000 řádků a najdeme v něm podrobné informace o přibližně dvou tisícovkách stanic. Zde je typický záznam:

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

Několik řádků jsme vynechali a zredukovali jsme také odsazení používané v tomto souboru. Celý soubor má přibližně 840 KB, takže jsme jej zkomprimovali pomocí programu **gzip** na přijatelnějších 72 KB. Jenže analyzátor elementových stromů vyžaduje buď název souboru, nebo objekt souboru, který má načíst. Komprimovaný soubor mu ale předat nemůžeme, protože ten by se mu jevil jen jako náhodná binární data. Tento problém můžeme vyřešit pomocí dvou počátečních kroků:

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

Funkce `gzip.open()` modulu `gzip` je podobná vestavěné funkci `open()` s výjimkou toho, že čte soubory zkomprimované algoritmem gzip (tj. s příponou `.gz`) jako binární data. Data potřebujeme mít k dispozici ve formě souboru, s nímž dokáže pracovat analyzátor elementových stromů, a proto binární data převádíme pomocí metody `bytes.decode()` na řetězec s kódováním UTF-8 (které používají soubory XML), z něhož pak vytváříme objekt typu `io.StringIO` chovající se jako soubor.

`io.StringIO` ➤ 210

typ `bytes` ➤ 286

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

Zde vytváříme nový objekt typu `xml.etree.ElementTree.ElementTree` a předáváme mu objekt souboru, z něhož má načíst data XML, která chceme analyzovat. Z pohledu analyzátoru elementových stromů se jedná o objekt souboru otevřený pro čtení, ačkoliv ve skutečnosti jde o řetězec uvnitř objektu typu `io.StringIO`. Potřebujeme extrahovat názvy všech meteorologických stanic, což provedeme snadno pomocí metody `xml.etree.ElementTree.ElementTree.getiterator()`, která vrací iterátor, který vrací všechny objekty typu `xml.etree.ElementTree.Element` se zadaným názvem značky. Pro získání textu pak už jen stačí použít atribut `text` daného elementu. Podobně jako při použití funkce `os.walk()` nemusíme ani zde provádět žádnou rekurzi, o kterou se za nás postará iterátor. Dokonce ani nemusíme uvádět značku – v takovém případě pak iterátor vrátí každý element v celém dokumentu XML.

Další moduly

Na probrání všech 200 balíčků a modulů dostupných ve standardní knihovně zde není místo. Nicméně tento všeobecný přehled by měl být dostačující, abyste získali jistou představu o tom, co vše knihovna nabízí a jaké jsou její klíčové balíčky v hlavních oblastech jejího nasazení. Několik dalších oblastí si ještě probereme v této poslední podčásti.

V předchozí části jsme si ukázali, jak snadné je vytvářet a spouštět testy v dokumentačních řetězcích pomocí modulu `doctest`. Knihovna kromě toho nabízí rámec pro testování jednotek poskytovaný modulem `unittest`. Jedná se o verzi testovacího rámce JUnit z Javy přenesenou do prostředí Pythonu. Modul `doctest` nabízí také jistou základní integraci s modulem `unittest`. (Testování se budeme detailně věnovat v lekci 9.) K dispozici je též několik testovacích rámců třetích stran, například py.test z *codespeak.net/py/dist/test/test.html* a nose z *code.google.com/p/python-nose*.

Neinteraktivní aplikace, jako jsou kupříkladu servery, používají pro hlášení problémů obvykle zápis do souborů protokolu. Modul `logging` nabízí jednotné rozhraní pro protokolování a kromě možnosti protokolovat do souborů dokáže zaznamenávat zprávy také pomocí požadavků HTTP GET či POST nebo pomocí e-mailu či soketů.

Knihovna nabízí spoustu modulů pro introspekci a manipulaci s kódem. Většina z nich jde sice nad rámec této knihy, je zde ale jeden, který stojí za zmínku. Modul `pprint` obsahuje funkce pro „hezké vypisování“ objektů Pythonu včetně datových typů představujících kolekce, což se někdy hodí pro ladění. Jednoduché použití modulu `inspect`, který nahlíží do živých objektů, si ukážeme v lekci 8.

Modul `threading` poskytuje podporu pro vytváření vícevláknových aplikací a modul `queue` poskytuje tři odlišné druhy front bezpečných vzhledem k vícevláknovému zpracování. Práci s vlákny budeme probírat v lekci 10.

Python neobsahuje žádnou nativní podporu pro programování grafických uživatelských rozhraní (GUI), programy napsané v Pythonu však mohou využít několik knihoven GUI. Knihovna Tk je dostupná prostřednictvím modulu `tkinter` a obvykle se instaluje jako standard. S programováním grafických uživatelských rozhraní se seznámíme v lekci 15.

Modul `abc` (Abstract Base Class – abstraktní bázová třída) nabízí funkce nezbytné pro vytváření abstraktní bázových tříd. Tento modul si rozebereme v lekci 8.

Mělké a hloubkové kopírování
➢ 146

Modul `copy` poskytuje funkce `copy.copy()` a `copy.deepcopy()`, které jsme probírali v lekci 3.

Přístup k *cizím funkcím*, tedy k funkcím ve sdílených knihovnách (soubory `.dll` ve Windows, soubory `.dylib` v Mac OS X a soubory `.so` v Linuxu), lze realizovat prostřednictvím modulu `ctypes`. Python dále nabízí rozhraní API pro jazyk C, je tedy možné vytvořit vlastní datové typy a funkce v jazyku C a zpřístupnit je Pythonu. Modul `ctypes` a rozhraní API Pythonu pro jazyk C je nad rámec této knihy.

Pokud žádný z balíčků a modulů zmíněných v této části nenabízí funkčnost, kterou potřebujete, pak ještě předtím, než si začnete potřebné funkce vytvářet sami, nahlédněte do rozcestníku dokumentace k Pythonu (Global Module Index) a podívejte se, zda není k dispozici nějaký vhodný modul, protože zde nejsme schopni zmínit se o každém z nich. Pokud ani zde nenajdete, co hledáte, zkuste se podívat na rozcestník balíčků Pythonu (*pypi.python.org/pypi*), který obsahuje několik tisíc doplňků

Pythonu od malých modulů tvořených jedním souborem až po rozsáhlé knihovny a rámcové balíčky obsahující desítky až stovky modulů.

Shrnutí

Na začátku této lekce jsme se seznámili s několika syntaxemi pro importování balíčků, modulů a objektů uvnitř modulů. Řekli jsme si, že řada programátorů používá pro zabránění kolizím názvů pouze syntaxi `import importovatelný_prvek` a že musíme být opatrní, abychom nedali svému programu či modulu stejný název, jaký má zastřešující modul či adresář Pythonu.

Probírali jsme také balíčky Pythonu. Jedná se o obyčejné adresáře se souborem `__init__.py` a s jedním nebo více moduly `.py`. Soubor `__init__.py` může být prázdný, ale kvůli podpoře syntaxe `from importovatelný_prvek import *` může obsahovat speciální proměnnou `__all__` nastavenou na seznam názvů modulů. Do souboru `__init__.py` můžeme také umístit libovolný inicializační kód. Řekli jsme si, že balíčky lze vnořovat prostým vytvářením podadresářů, z nichž každý obsahuje svůj vlastní soubor `__init__.py`.

Popsali jsme si dva vlastní moduly. První poskytoval jen pár funkcí a měl velice jednoduché dokumentační testy. Druhý byl mnohem propracovanější a obsahoval jednu vlastní výjimku, pomocí dynamické tvorby funkcí vytvářel funkci s implementací specifickou pro používanou platformu, obsahoval soukromá globální data, volání inicializační funkce a více propracovanějších dokumentačních testů.

Přibližně polovina lekce byla věnována přehledu standardní knihovny Pythonu. Zmínili jsme se o několika modulech pro práci s řetězci a ukázali jsme si pár příkladů se třídou `io.StringIO`. V jednom příkladu jsme viděli, jak zapisovat text do souboru buď pomocí vestavěné funkce `print()`, nebo metody `write()` objektu souboru a jak místo skutečného souboru použít objekt typu `io.StringIO`. V předchozích lekcích jsme zpracovávali volby na příkazovém řádku tak, že jsme sami analyzovali data v seznamu `sys.argv`, avšak při probírání podpory knihovny pro programování na příkazovém řádku jsme se seznámili s modulem `optparse`, který výrazným způsobem zjednodušuje práci s argumenty příkazového řádku. Od této chvíle budeme tento modul využívat mnohem častěji.

Zmínili jsme se o skvělé podpoře Pythonu pro čísla a o číselných typech knihovny a jejich třech modulech s matematickými funkcemi a také o podpoře pro vědecké a inženýrské výpočty poskytované projektem SciPy. Stručně jsme si popsali třídy knihovny a třetích stran pro práci s datem a časem a ukázali jsme si, jak získat aktuální datum a čas a jak provádět převody mezi typem `datetime.datetime` a počtem vteřin od počátku epochy. Dále jsme probírali další datové typy představující kolekce a algoritmy pro práci s uspořádanými posloupnostmi nabízenými standardní knihovnou společně s několika příklady použití funkcí modulu `heapq`.

Probírali jsme moduly, které podporují nejrůznější kódování souborů (kromě kódování znaků), a také moduly pro zabalování a rozbalování nejznámějších archivačních formátů a moduly podporující zvuková data. Ukázali jsme si, jak pomocí kódování Base64 uložit binární data do souboru `.py` a také program pro rozbalování archivů tarball. K dispozici máme též značnou podporu pro práci s adresáři a soubory, přičemž vše je abstrahováno do funkcí nezávislých na používané platformě. Ukázali jsme si příklady pro vytváření slovníku s klíči tvořenými názvy souborů a hodnotami ve formě časové

známky poslední modifikace a pro provádění rekurzivního procházení adresáře za účelem nalezení případných duplicitních souborů na základě jejich názvu a velikosti.

Velká část knihovny je věnována síťovému a internetovému programování. Stručně jsme prozkoumali, co vše je k dispozici, od holých soketů (včetně šifrovaných soketů) přes servery TCP a UDP až po server HTTP a podporu pro rozhraní WSGI. Zmínili jsme se také o modulech pro práci se soubory cookie, skripty CGI a daty HTTP a pro analyzování HTML, XHTML a adres URL. Mezi další zmíněné moduly patří ty, které podporují vzdálená volání procedur XML-RPC, protokoly vyšší úrovně, jako jsou FTP a NNTP, e-mailové klienty a servery používající protokol SMTP a podporu protokolů IMAP4 a POP3 na straně klientů.

Zmínili jsme též komplexní podporu knihovny pro zapisování a analyzování kódu jazyka XML včetně analyzátorů modelu DOM, rozhraní SAX a elementových stromů a modulu `expat`. Ukázali jsme si také jeden příklad využívající modul pro analýzu elementových stromů. Dále jsme se zmínili o několika z mnoha dalších balíčků a modulů nabízených standardní knihovnou.

Standardní knihovna Pythonu představuje neobyčejně užitečný zdroj, který může ušetřit značné množství času a úsilí a v řadě případů nám díky nabízeným funkčním prvkům umožňuje psát mnohem menší programy. Kromě toho existují doslova tisíce balíčků třetích stran vyplňujících mezery, které bychom mohli ve standardní knihovně objevit. Díky této předdefinované funkčnosti se můžeme v mnohem větší míře zaměřit na to, co chceme, aby náš program prováděl, přičemž necháme na modulech knihovny, aby se postaraly o většinu detailů.

Tato Lekce nás přivedla na konec základů procedurálního programování. V pozdějších lekcích, zvláště v lekci 8, se podíváme na pokročilejší a specializovanější procedurální techniky a v následující lekci se seznámíme s objektově orientovaným programováním. Python sice můžeme používat jen jako čistě procedurální jazyk, což může být zvláště u malých prográmků praktické, ale pro středně velké až rozsáhlé programy, pro vlastní balíčky a moduly a pro dlouhodobou udržovatelnost je objektově orientovaný přístup daleko lepší. Naštěstí je vše, co jsme se dosud naučili, užitečné a relevantní také v oblasti objektově orientovaného programování, takže v následujících lekcích budeme pokračovat v budování našich znalostí a dovedností v oblasti jazyka Python na všech dosud položených základech.

Cvičení

Napište program vypisující obsah adresáře jako příkaz `dir` ve Windows nebo `ls` v Unixu. Výhoda vytvoření našeho vlastního programu spočívá v tom, že do něj můžeme zabudovat preferované výchozí chování a hodnoty a používat na všech platformách stejný program bez toho, abychom si museli pamatovat odlišnosti mezi příkazy `dir` a `ls`. Vytvořte program, který podporuje následující rozhraní:

```
Usage: ls.py [volby] [cesta1 [cesta2 [... cestaN]]]
Cesty jsou volitelné. Nejsou-li zadány, použije se aktuální adresář.
Options:
  -h, --help            show this help message and exit
  -H, --hidden          zobrazí skryté soubory [výchozí: off]
  -m, --modified        zobrazí datum a čas poslední modifikace [výchozí: off]
```

```
  -o ORDER, --order=ORDER
                        seřadí výstup podle ('name', 'n', 'modified', 'm',
                        'size', 's') [výchozí: name]
  -r, --recursive       sestupuje rekurzivně  do podadresářů [výchozí: off]
  -s, --sizes           zobrazí velikosti [výchozí: off]
 (Výstup programu byl pro účely knihy trošku upraven.)
Zde je ukázkový výstup na malém adresáři s použitím příkazového řádku ls.py -ms -os
Graphics/:
2009-11-20 10:04:52              49 Graphics/__init__.py
2009-11-20 10:04:52             351 Graphics/Bmp.py
2009-11-20 10:04:52             351 Graphics/Xpm.py
2009-11-20 10:04:52             357 Graphics/Jpeg.py
2009-11-20 10:04:52             357 Graphics/Tiff.py
2009-11-20 10:04:52             368 Graphics/Png.py
                                    Graphics/Vector/
6 souborů, 1 adresář
```

Na příkazovém řádku jsme použili seskupování voleb (automaticky se o to postará modul `optparse`), stejného výsledku bychom dosáhli také při použití samostatných voleb, například `ls.py -m -s -os Graphics/`, nebo dokonce s použitím ještě většího seskupení `ls.py -msos Graphics/` nebo s použitím dlouhých voleb `ls.py --modified --sizes --order=size Graphics/` nebo libovolnou kombinací těchto možností. Všimněte si, že „skryté" soubory či adresáře definujeme jako soubory či adresáře, jejichž název začíná tečkou (.).

Toto cvičení je docela náročné. Budete si muset pročíst dokumentaci k modulu `optparse`, z níž se dozvíte, jak definovat volby, které nastavují hodnotu `True`, a jak nabídnout fixní seznam možností. Pokud uživatel nastaví rekurzivní volbu, budete muset zpracovat soubory pomocí funkce `os.walk()`. V opačném případě musíte použít funkci `os.listdir()` a zpracovat soubory a adresáře ručně.

Jeden ze složitějších aspektů tkví v tom, jak zabránit, aby se při rekurzi použily skryté adresáře. Lze je odříznout od seznamu `dirs` funkce `os.walk()` (která je tudíž přeskočí) modifikací tohoto seznamu. Buďte však opatrní, abyste nepřiřazovali do samotné proměnné `dirs`, protože tím by se seznam, na který ukazuje, nijak nezměnil, ale jen by se (zbytečně) nahradil jiným. V modelovém řešení přiřazujeme do řezu celého seznamu, což znamená `dirs[:] = [dir for dir in dirs if not dir.startswith(".")]`.

Znaky pro seskupování čísel představujících velikosti souboru získáte nejlépe importem modulu `locale`, zavoláním funkce `locale.setlocale()` pro získání výchozího národního prostředí uživatele a použitím formátovacího znaku `n`. Výsledný program `ls.py` má přibližně 130 řádků rozdělených do čtyř funkcí.

`locale.setlocale()` ➤ 90