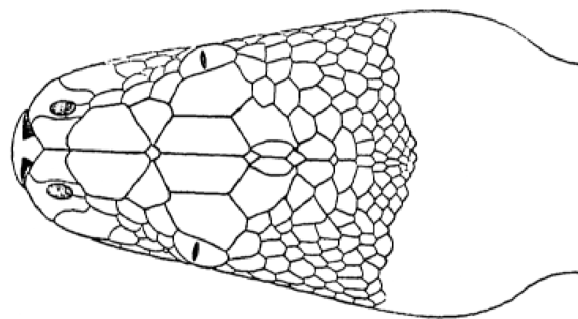
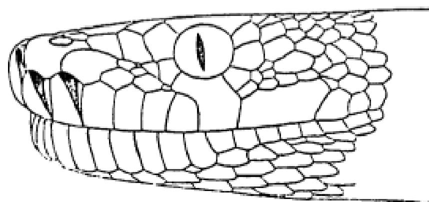
Zanurkuj w Pythonie





*Stworzone na Wikibooks,
bibliotece wolnych podręczników.*

Wydanie I z dnia 17 lutego 2008
Copyright © 2005-2008 użytkownicy Wikibooks.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Udziela się zezwolenia na kopiowanie, rozpowszechnianie i/lub modyfikację treści artykułów polskich Wikibooks zgodnie z zasadami Licencji GNU Wolnej Dokumentacji (GNU Free Documentation License) w wersji 1.2 lub dowolnej późniejszej opublikowanej przez Free Software Foundation; bez Sekcji Niezmiennych, Tekstu na Przedniej Okładce i bez Tekstu na Tylnej Okładce. Kopia tekstu licencji znajduje się w części zatytułowanej "GNU Free Documentation License".

Dodatkowe objaśnienia są podane w dodatku "Dalsze wykorzystanie tej książki".

Wikibooks nie udziela żadnych gwarancji, zapewnień ani obietnic dotyczących poprawności publikowanych treści. Nie udziela też żadnych innych gwarancji, zarówno jednoznacznych, jak i dorozumianych.

Spis treści

1 Wstęp **1**
1.1 O podręczniku 2

2 Instalacja **3**
2.1 Który Python jest dla ciebie najlepszy? 4
2.2 Python w systemie Windows 5
2.3 Python w systemie Mac OS 7
2.4 Python w systemach Linux 10
2.5 Instalacja ze źródeł 13
2.6 Interaktywna powłoka 14
2.7 Podsumowanie 15

3 Pierwszy program **17**
3.1 Nurkujemy 18
3.2 Deklarowanie funkcji 20
3.3 Dokumentowanie funkcji 22
3.4 Wszystko jest obiektem 23
3.5 Wcięcia kodu 26
3.6 Testowanie modułów 28

4 Wbudowane typy danych **29**
4.1 Łańcuchy znaków i unikod 30
4.2 Słowniki 32
4.3 Listy 35
4.4 Krotki 41
4.5 Deklarowanie zmiennych 43
4.6 Formatowanie łańcucha znaków 46
4.7 Odwzorowywanie listy 48
4.8 Łączenie list i dzielenie łańcuchów znaków 50
4.9 Kodowanie znaków 52
4.10 Praca z unikodem 56
4.11 Podsumowanie 59

5 Potęga introspekcji **61**
5.1 Nurkujemy 62
5.2 Argumenty opcjonalne i nazwane 64
5.3 Dwa sposoby importowania modułów 66
5.4 type, str, dir i inne wbudowane funkcje 68

5.5 Funkcja getattr 73
5.6 Filtrowanie listy 76
5.7 Operatory and i or 78
5.8 Wyrażenia lambda 81
5.9 Potęga introspekcji - wszystko razem 84
5.10 Podsumowanie 87

6 Obiekty i klasy 89
6.1 Nurkujemy 90
6.2 Definiowanie klas 93
6.3 Tworzenie instancji klasy 97
6.4 Klasa opakowująca UserDict 99
6.5 Metody specjalne 102
6.6 Zaawansowane metody specjalne 105
6.7 Atrybuty klas 107
6.8 Funkcje prywatne 109
6.9 Podsumowanie 111

7 Wyjątki i operacje na plikach 113
7.1 Obsługa wyjątków 114
7.2 Praca na plikach 118
7.3 Pętla for 123
7.4 Korzystanie z sys.modules 126
7.5 Praca z katalogami 129
7.6 Wyjątki i operacje na plikach - wszystko razem 133
7.7 Wyjątki i operacje na plikach - podsumowanie 135

8 Wyrażenia regularne 137
8.1 Nurkujemy 138
8.2 Analiza przypadku: Adresy ulic 139
8.3 Analiza przypadku: Liczby rzymskie 142
8.4 Składnia ?n, m? 146
8.5 Rozwlekłe wyrażenia regularne 150
8.6 Analiza przypadku: Przetwarzanie numerów telefonów 152
8.7 Podsumowanie 157

9 Przetwarzanie HTML-a 159
9.1 Nurkujemy 160
9.2 Wprowadzenie do sgmllib.py 166
9.3 Wyciąganie danych z dokumentu HTML 169
9.4 Wprowadzenie do BaseHTMLProcessor.py 172
9.5 locals i globals 175
9.6 Formatowanie napisów w oparciu o słowniki 179
9.7 Dodawanie cudzysłowów do wartości atrybutów 181
9.8 Wprowadzenie do dialect.py 183
9.9 Przetwarzanie HTML-a - wszystko razem 187
9.10 Podsumowanie 190

10 Przetwarzanie XML-a **191**
10.1 Nurkowanie 192
10.2 Pakiety 200
10.3 Parsowanie XML-a 203
10.4 Wyszukiwanie elementów 207
10.5 Dostęp do atrybutów elementów 209
10.6 Podsumowanie 211

11 Skrypty i strumienie **213**
11.1 Abstrakcyjne źródła wejścia 214
11.2 Standardowy strumień wejścia, wyjścia i błędów 219
11.3 Buforowanie odszukanego węzła 224
11.4 Wyszukanie bezpośrednich elementów potomnych 226
11.5 Tworzenie oddzielnych funkcji obsługi względem typu węzła 227
11.6 Obsługa argumentów linii poleceń 230
11.7 Skrypty i strumienie - wszystko razem 234
11.8 Podsumowanie 236

12 HTTP **237**
12.1 Nurkujemy 238
12.2 Jak nie pobierać danych poprzez HTTP 241
12.3 Właściwości HTTP 242
12.4 Debugowanie serwisów HTTP 245
12.5 Ustawianie User-Agent 247
12.6 Korzystanie z Last-Modified i ETag 249
12.7 Obsługa przekierowań 253
12.8 Obsługa skompresowanych danych 258
12.9 HTTP - wszystko razem 261
12.10Podsumowanie 264

13 SOAP **265**
13.1 Nurkujemy 266
13.2 Instalowanie odpowiednich bibliotek 268
13.3 Pierwsze kroki z SOAP 270
13.4 Debugowanie serwisu sieciowego SOAP 271
13.5 Wprowadzenie do WSDL 273
13.6 Introspekcja SOAP z użyciem WSDL 274
13.7 Wyszukiwanie w Google 277
13.8 Rozwiązywanie problemów 280
13.9 Podsumowanie 284

14 Testowanie jednostkowe **285**
14.1 Wprowadzenie do liczb rzymskich 286
14.2 Nurkujemy 288
14.3 Wprowadzenie do romantest.py 289
14.4 Testowanie poprawnych przypadków 293
14.5 Testowanie niepoprawnych przypadków 296
14.6 Testowanie zdroworozsądkowe 299

15 Testowanie 2 303
15.1 roman.py, etap 1 . . . 304
15.2 roman.py, etap 2 . . . 309
15.3 roman.py, etap 3 . . . 314
15.4 roman.py, etap 4 . . . 318
15.5 roman.py, etap 5 . . . 321

16 Refaktoryzacja 325
16.1 Obsługa błędów . . . 326
16.2 Obsługa zmieniających się wymagań . . . 329
16.3 Refaktoryzacja . . . 337
16.4 Postscript . . . 342
16.5 Podsumowanie . . . 345
16.6 Nurkujemy . . . 346
16.7 Znajdowanie ścieżki . . . 349
16.8 Filtrowanie listy . . . 352
16.9 Odwzorowywanie listy . . . 355
16.10 Programowanie koncentrujące się na danych . . . 357
16.11 Dynamiczne importowanie modułów . . . 359

17 Programowanie funkcyjne 361
17.1 Programowanie funkcyjne - wszystko razem . . . 362
17.2 Podsumowanie . . . 366

18 Funkcje dynamiczne 367
18.1 Nurkujemy . . . 368
18.2 plural.py, etap 1 . . . 369
18.3 plural.py, etap 2 . . . 372
18.4 plural.py, etap 3 . . . 374
18.5 plural.py, etap 4 . . . 376
18.6 plural.py, etap 5 . . . 379
18.7 plural.py, etap 6 . . . 381
18.8 Podsumowanie . . . 385

19 Optymalizacja szybkości 387
19.1 Nurkujemy . . . 388
19.2 Korzystanie z modułu timeit . . . 391
19.3 Optymalizacja wyrażeń regularnych . . . 393
19.4 Optymalizacja przeszukiwania słownika . . . 397
19.5 Optymalizacja operacji na listach . . . 401
19.6 Optymalizacja operacji na napisach . . . 404
19.7 Podsumowanie . . . 406

A Informacje o pliku 407
A.1 Historia . . . 407
A.2 Informacje o pliku PDF i historia . . . 407
A.3 Autorzy . . . 407
A.4 Grafiki . . . 407

B Dalsze wykorzystanie tej książki **409**
B.1 Wstęp 409
B.2 Status prawny 409
B.3 Wykorzystywanie materiałów z Wikibooks 409

C GNU Free Documentation License **411**

Rozdział 1

Wstęp

1.1 O podręczniku

Podręcznik ten powstaje na podstawie książki *Dive into Python* (w większości jest to tłumaczenie), której autorem jest Mark Pilgrim, a udostępnionej na licencji GNU Free Documentation License.

Autorzy i tłumacze

- Mark Pilgrim (autor książki *Dive into Python*)
- Warszk
- Piotr Kieć
- Roman Frołow
- Andrzej Saski
- Adam Kubiczek

Rozdział 2

Instalacja

2.1 Który Python jest dla ciebie najlepszy?

Witamy w Pythonie. W tym rozdziale zajmiemy się instalacją Pythona.

Który Python jest dla ciebie najlepszy?

Aby móc korzystać z Pythona, najpierw należy go zainstalować. A może już go mamy?

Jeżeli posiadasz konto na jakimkolwiek serwerze, istnieje duże prawdopodobieństwo, że Python jest tam już zainstalowany. Wiele popularnych dystrybucji Linuksa standardowo instaluje ten język programowania. Systemy Mac OS X 10.2 i nowsze posiadają dosyć okrojoną wersję Pythona dostępnego jedynie z poziomu linii poleceń. Zapewne będziesz chciał zainstalować wersję, która da Ci więcej możliwości.

Windows domyślnie nie zawiera żadnej wersji Pythona, ale nie załamuj się! Istnieje wiele sposobów, by w łatwy sposób zainstalować Pythona w tym systemie operacyjnym.

Jak widzisz, wersje Pythona są dostępne na wiele platform i systemów operacyjnych. Możemy zdobyć Pythona zarówno dla Windowsa, Mac OS, Mac OS X, wszystkich wariantów Uniksa, w tym Linuksa czy Solarisa, jak i dla Amigi, OS/2, BeOSa, czy też innych systemów, o których najprawdopodobniej nawet nie słyszałeś.

Co najważniejsze, program napisany w Pythonie na jednej platformie, przy zachowaniu niewielkiej dozy ostrożności, zadziała na jakiejkolwiek innej. Możesz na przykład rozwijać swój program pod Windowsem, a następnie przenieść go do Linuksa.

Wracając do pytania rozpoczynającego sekcję, "Który Python jest dla ciebie najlepszy?". Odpowiedź jest jedna: jakikolwiek, który możesz zainstalować na posiadanym komputerze.

2.2 Python w systemie Windows

Python w Windowsie

W Windowsie mamy parę sposobów zainstalowania Pythona.

Firma ActiveState tworzy instalator Pythona zwany ActivePython. Zawiera on kompletną wersje Pythona, IDE z bardzo dobrym edytorem kodu oraz kilka rozszerzeń dla Windowsa, które zapewniają dostęp do specyficznych dla Windowsa usług, API oraz rejestru.

ActivePython można pobrać nieodpłatnie, ale nie jest produktem *Open Source*. Wydawany jest kilka miesięcy po wersji oryginalnej.

Drugą opcją jest instalacja "oficjalnej" wersji Pythona, rozprowadzanej przez ludzi, którzy rozwijają ten język. Jest to wersja ogólnodostępna, *Open Source* i zawsze najnowsza.

Instalacja ActivePythona

Oto procedura instalacji ActivePythona:

1. Ściągamy ActivePythona ze strony `http://www.activestate.com/Products/ActivePython/`.

2. Jeżeli używamy *Windows 95/98/ME/NT4/2000*, będziemy musieli najpierw zainstalować *Windows Installer 2.0* dla *Windowsa 95/98/Me* lub *Windows Installer 2.0* dla *Windowsa NT4/2000*.

3. Klikamy dwukrotnie na ściągnięty plik `ActivePython-(pobrana wersja)-win32-ix86.msi`

4. Przechodzimy wszystkie kroki instalatora.

5. Po zakończeniu instalacji wybieramy `Start->Programy->ActiveState ActivePython 2.2->PythonWin IDE`. Zobaczymy wtedy ekran z napisem podobnym do poniższego:

```
PythonWin 2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)] on win32.
Portions Copyright 1994-2001 Mark Hammond (mhammond@skippinet.com.au) -
see 'Help/About PythonWin' for further copyright information.
>>>
```

Instalacja Pythona z Python.org

1. Pobieramy z `http://www.python.org/ftp/python/` najnowszą wersję instalatora dla Windowsa, który oczywiście będzie miał rozszerzenie `.exe`.

2. Klikamy dwukrotnie na instalatorze `Python-2.xxx.yyy.msi`. Nazwa zależeć będzie od ściągniętej wersji Pytona.

3. Jeżeli używamy *Windows 95/98/ME/NT4/2000*, będziemy musieli najpierw zainstalować *Windows Installer 2.0* dla *Windowsa 95/98/Me* lub *Windows Installer 2.0* dla *Windowsa NT4/2000*.

4. Przechodzimy przez wszystkie kroki instalatora.

5. Jeżeli nie mamy uprawnień administratora, możemy wybrać `Advanced Options`, a następnie `Non-Admin Install`.

6. Po zakończeniu instalacji, wybieramy `Start->Programy->Python 2.x->IDLE (Python GUI)`. Zobaczymy ekran z napisem podobnym do poniższego:

```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.1
>>>
```

2.3 Python w systemie Mac OS

Python w Mac OS X

W Mac OS X możemy mieć Pythona na dwa sposoby: instalując go lub nie robiąc tego. Zapewne będziesz chciał go zainstalować.

Mac OS X 10.2 i nowsze domyślnie instalują okrojoną wersję Pythona dostępnego jedynie z linii poleceń. Jeżeli nie przeszkadza Ci praca w linii poleceń, to początkowo taka wersja może Tobie wystarczyć. Jednak nie posiada ona parsera XML, więc jeśli dojdziesz do rozdziału mówiącego na ten temat i tak będziesz musiał zainstalować pełną wersję.

Zamiast więc używać domyślnie zainstalowanej wersji, lepiej będzie od razu zainstalować najnowszą, a która też dostarczy nam wygodną, graficzną powłokę.

Uruchamianie wersji domyślnie zainstalowanej z systemem

1. Otwieramy katalog `/Applications`
2. Otwieramy katalog `Utilities`
3. Klikamy dwukrotnie na `Terminal`, by otworzyć okienko terminala, które zapewni nam dostęp do linii poleceń.
4. Wpisujemy polecenie `python`.

Powinniśmy otrzymać mniej więcej takie coś:

```
<span>Welcome to Darwin!
[localhost:~] you\% python
Python 2.2 (\#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you\%</span>
```

Instalacja najnowszej wersji Pythona

Aby to zrobić postępujemy według poniższych kroków:

1. Ściągamy obraz dysku MacPython-OSX z http://homepages.cwi.nl/~jack/macpython/download.html.
2. Jeżeli pobrany program nie zostanie uruchomiony przez przeglądarkę, klikamy dwukrotnie na `MacPython-OSX-(pobrana wersja).dmg` by zamontować obraz dysku w systemie.
3. Klikamy dwukrotnie na instalator `MacPython-OSX.pkg`.
4. Instalator poprosi o login i hasło użytkownika z prawami administratora.
5. Przechodzimy wszystkie kroki instalatora.
6. Po zakończonej instalacji otwieramy katalog `/Applications`.
7. Otwieramy katalog `MacPython-2.x`.

8. Klikamy dwukrotnie na `PythonIDE` by uruchomić Pythona.

MacPython IDE wyświetli ekran powitalny, a następnie interaktywną powłokę. Jeżeli jednak powłoka się nie pojawi, wybieramy `Window->Python Interactive (Cmd-0)`. Otwarte okienko powinno wyglądać podobnie do tego:

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

Po instalacji najnowszej wersji, domyślnie zainstalowana wersja Pythona nadal pozostanie w systemie. Podczas uruchamiania skryptów zwróć uwagę z jakiej wersji korzystasz.

Dwie wersje Pythona w Mac OS X

```
<span>[localhost:~] you\% python
Python 2.2 (\#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you\% /usr/local/bin/python
Python 2.3 (\#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you\%</span>
```

Instalacja Pythona z MacPortów

Ta metoda jest najlepsza. Należy wpierw pobrać i zainstalować MacPorts (http://www.macports.org/). Następnie należy odświeżyć porty

```
sudo port selfupdate
```

Potem możemy wyszukiwać interesujące nasz pakiety. Np. znalezienie wszystkich pakietów do Pythona 2.5.x:

```
port search py25
```

Właściwa instalacja Pythona:

```
sudo port install python25
```

Wszystkie programy instalowane tą metodą są składowane w `/opt/local`. Warto więc dodać do ścieżki `PATH` `/opt/local/bin`.

Dobrze jest też doinstalować `setuptools`, który daje dostęp do pythonowego instalatora pakietów, skryptu `easy_install`.

```
sudo port install py25-setuptools
```

Przydaje się, gdy nie ma w portach pakietu dla naszej wersji Pythona, np. IPythona. Część bibliotek można instalować MacPortami, a resztę za pomocą `easy_setup`. Na przykład IPythona doinstalujemy za pomocą:

```
sudo easy_install ipython
```

Można też aktualizować pakiety:

```
sudo easy_install -U Pylons
```

Duże i małe znaki w nazwach pakietów, w wypadku użycia `easy_install`, nie mają znaczenia.

Python w Mac OS 9

Mac OS 9 nie posiada domyślnie żadnej wersji Pythona, ale samodzielna instalacja jest bardzo prosta.

1. Ściągamy plik `MacPython23full.bin` z http://homepages.cwi.nl/~jack/macpython/download.html.
2. Jeżeli plik nie zostanie automatycznie rozpakowany przez przeglądarkę, klikamy dwukrotnie na `MacPython23full.bin` by to zrobić.
3. Klikamy dwukrotnie instalator `MacPython23full`.
4. Przechodzimy wszystkie kroki instalatora.
5. Po zakończonej instalacji otwieramy katalog `/Applications`.
6. Otwieramy katalog `MacPython-OS9 2.x`.
7. Kliknij dwukrotnie na `Python IDE` by uruchomić Pythona.

MacPython IDE wyświetli ekran powitalny, a następnie interaktywną powłokę. Jeżeli jednak powłoka się nie pojawi, wybieramy `Window->Python Interactive (Cmd-0)`. Otwarte okienko powinno wyglądać podobnie do tego:

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

2.4 Python w systemach Linux

Python w dystrybucjach Linuksa

Instalacja z gotowych pakietów binarnych dla konkretnej dystrybucji Linuksa jest stosunkowo prosta. Większość dystrybucji posiada już zainstalowaną wersję Pythona. Możesz także pokusić się o instalację ze źródeł.

Wiele dystrybucji Linuksa zawiera graficzne narzędzia służące do instalacji oprogramowania. My jednak opiszemy, jak to zrobić w konsoli w wybranych dystrybucjach Linuksa.

Python w dystrybucji Red Hat Linux

Możemy zainstalować Pythona wykorzystując polecenie `rpm`:

```
localhost:~\$ su -
Password: [wpisz hasło roota]
[root@localhost root]# wget http://python.org/ftp/python/2.3/rpms/redhat-9/
python2.3-2.3-5pydotorg.i386.rpm
Resolving python.org... done.
Connecting to python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7,495,111 [application/octet-stream]
...
[root@localhost root]# rpm -Uvh python2.3-2.3-5pydotorg.i386.rpm
Preparing...                ########################################### [100\%]
   1:python2.3              ########################################### [100\%]
[root@localhost root]# python                                   #(1)
Python 2.2.2 (\#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [wciśnij Ctrl+D, żeby wyjść z programu]
[root@localhost root]# python2.3                                #(2)
Python 2.3 (\#1, Sep 12 2003, 10:53:56)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [wciśnij Ctrl+D, żeby wyjść z programu]
[root@localhost root]# which python2.3    #(3)
 /usr/bin/python2.3
```

1. Wpisując polecenie `python` zostaje uruchomiony Python. Jednak jest to starsza jego wersja, domyślnie zainstalowana wraz z systemem. To nie jest to, czego chcemy.

2. Podczas pisania tej książki najnowszą wersją był Python 2.3. Za pomocą polecenia `python2.3` uruchomimy nowszą, właśnie zainstalowaną wersje.

3. Jest to pełna ścieżka do nowszej wersji Pythona, którą dopiero co zainstalowaliśmy.

Python w dystrybucji Debian

Pythona zainstalujemy wykorzystując polecenie `apt-get`.

```
localhost:~$ su -
Password: [wpisz hasło roota]
localhost:~# apt-get install python
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  python2.3
Suggested packages:
  python-tk python2.3-doc
The following NEW packages will be installed:
  python python2.3
0 upgraded, 2 newly installed, 0 to remove and 3 not upgraded.
Need to get 0B/2880kB of archives.
After unpacking 9351kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Selecting previously deselected package python2.3.
(Reading database ... 22848 files and directories currently installed.)
Unpacking python2.3 (from .../python2.3_2.3.1-1_i386.deb) ...
Selecting previously deselected package python.
Unpacking python (from .../python_2.3.1-1_all.deb) ...
Setting up python (2.3.1-1) ...
Setting up python2.3 (2.3.1-1) ...
Compiling python modules in /usr/lib/python2.3 ...
Compiling optimized python modules in /usr/lib/python2.3 ...
localhost:~# exit
logout
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [wciśnij Ctrl+D, żeby wyjść z programu]
```

Python w dystrybucji Mandriva

W konsoli z użytkownika `root` wpisujemy polecenie:

```
$ su -
Password: [wpisz hasło roota]
# urpmi python
```

Python w dystrybucji Fedora/Fedora Core

Aby zainstalować Pythona w dystrybucji *Fedora/Fedora Core* należy w konsoli wpisać:

```
$ su -
Password: [wpisz hasło roota]
# yum install python
```

Można też zainstalować Pythona przy instalacji systemu, wybierając pakiety programistyczne.

Python w dystrybucji Gentoo GNU/Linux

W Gentoo do instalacji Pythona możemy użyć programu `emerge`:

```
$ su -
Password: [wpisz hasło roota]
# emerge python
```

2.5 Instalacja ze źródeł

Instalacja ze źródeł

Jeżeli wolimy zainstalować Pythona ze źródeł, będziemy musieli pobrać kod źródłowy z http://www.python.org/ftp/python/. Wybieramy najnowszą wersję (najwyższy numer) i ściągamy plik `.tgz`, a następnie wykonujemy standardowe komendy instalacyjne (`./configure`, `make`, `make install`).

```
localhost:~\$ su -
Password: [wpisz hasło roota]
localhost:~# wget http://www.python.org/ftp/python/2.3/Python-2.3.tgz
Resolving www.python.org... done.
Connecting to www.python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8,436,880 [application/x-tar]
...
localhost:~# tar xfz Python-2.3.tgz
localhost:~# cd Python-2.3
localhost:~/Python-2.3# ./configure
checking MACHDEP... linux2
checking EXTRAPLATDIR...
checking for --without-gcc... no
...
localhost:~/Python-2.3# make
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -I. \
    -I./Include  -DPy\_BUILD\_CORE -o Modules/python.o Modules/python.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -I. \
    -I./Include  -DPy\_BUILD\_CORE -o Parser/acceler.o Parser/acceler.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -I. \
    -I./Include  -DPy\_BUILD\_CORE -o Parser/grammar1.o Parser/grammar1.c
...
localhost:~/Python-2.3# make install
/usr/bin/install -c python /usr/local/bin/python2.3
...
localhost:~/Python-2.3# exit
logout
localhost:~$ which python
/usr/local/bin/python
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [wciśnij Ctrl+D, żeby wyjść z programu]
localhost:~$
```

2.6 Interaktywna powłoka

Interaktywna powłoka

Teraz kiedy już mamy zainstalowanego Pythona, pewnie się zastanawiamy, co to jest ta **interaktywna powłoka** (*interactive shell*), którą uruchomiliśmy?

Powiedzmy tak: Python umożliwia pracę na dwa sposoby. Jest interpreterem skryptów, które możemy uruchomić z linii poleceń lub, jak inne aplikacje, dwukrotnie klikając na ikonce skryptu, a także jest interaktywną powłoką, dzięki której możemy debugować, sprawdzać działanie potrzebnych funkcji czy możliwości Pythona. Możesz nawet potraktować powłokę jako kalkulator!

Uruchom teraz powłokę Pythona w odpowiedni sposób dla twojego systemu i sprawdźmy co ona potrafi:

Przykład 1.5 Pierwsze kroki w interaktywnej powłoce

```
>>> 1 + 1                  #(1)
2
>>> print 'hello world'    #(2)
hello world
>>> x = 1                  #(3)
>>> y = 2
>>> x + y
3
```

1. Interaktywna powłoka Pythona może wyliczać dowolne wyrażenia matematyczne.
2. Potrafi wykonywać dowolne polecenia Pythona.
3. Możemy przypisać pewne wartości do zmiennych, które są pamiętane tak długo, jak długo jest uruchomiona nasza powłoka (ale nie dłużej).

2.7 Instalacja - podsumowanie

Podsumowanie

W tym momencie powinniśmy już mieć zainstalowanego Pythona.

W zależności od platformy możesz mieć zainstalowaną więcej niż jedną wersję Pythona. Jeżeli tak właśnie jest to musisz uważać na ścieżki dostępu do programu. Pisząc tylko `python` w linii poleceń, może się okazać, że nie uruchamiasz wersji, której akurat potrzebujesz. Być może będziesz musiał podawać pełną ścieżkę dostępu lub odpowiednią nazwę (`python2.2`, `python2.4` itp.)

Gratulujemy i witamy w Pythonie!

Rozdział 3

Pierwszy program

3.1 Pierwszy program

Czy dostrzegliśmy, że większość książek najpierw przedstawia elementarne zasady programowania, a potem opisuje, jak korzystając z nich stworzyć kompletny i działający program? My zrobimy inaczej...

Nurkujemy

Oto kompletny, działający program w Pythonie. Prawdopodobnie jest on dla Ciebie całkowicie niezrozumiały, ale nie przejmuj się tym, ponieważ zaraz przeanalizujemy go dokładnie, linia po linii. Przeczytaj go i sprawdź, czy coś jesteś w stanie z niego zrozumieć.

Przykład 2.1 odbchelper.py

```
#-*- coding: utf-8 -*-

def buildConnectionString(params):
    u"""Tworzy łańcuch znaków na podstawie słownika parametrów.

    Zwraca łańcuch znaków.
    """
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret"
               }
    print buildConnectionString(myParams)
```

Teraz uruchommy ten program i zobaczmy, co się stanie.

W IDE ActivePythona w systemie Windows możemy uruchomić edytowany program wybierając `File->Run... (Ctrl-R)`. Wynik wyświetlany jest w interaktywnym oknie.

W IDE Pythona w systemie Mac OS uruchomimy program wybierając `Python->Run window... (Cmd-R)`, jednak wcześniej musimy ustawić pewną ważną opcję. W tym celu otwieramy plik `.py` w IDE, wywołujemy menu podręczne klikając czarny trójkąt w prawym górnym rogu okna i upewniamy się, że opcja `Run as __main__` jest zaznaczona.

W systemach *Unix* (także w *Mac OS X*) możesz uruchomić program z linii poleceń poleceniem `python odbchelper.py`

W wyniku uruchomienia programu otrzymujemy:

```
pwd=secret;database=master;uid=sa;server=mpilgrim
```

3.2 Deklarowanie funkcji

Deklarowanie funkcji

Python posiada funkcje, podobnie jak wiele innych języków programowania, lecz nie definiujemy ich w oddzielnych plikach nagłówkowych (jak np. w C++), czy też nie dzielimy na sekcje interfejsu i implementacji jak w Pascalu. Jeśli potrzebujemy jakiejś funkcji, po prostu ją deklarujemy, na przykład:

```
def buildConnectionString(params):
```

Słowo kluczowe `def` rozpoczyna deklarację funkcji, następnie podajemy nazwę funkcji, a potem w nawiasach parametry. Większą liczbę parametrów podajemy po przecinkach.

Jak widać funkcja nie definiuje zwracanego typu. Podczas deklarowania Pythonowych funkcji nie określamy, czy mają one zwracać jakąś wartość, a nawet czy mają cokolwiek zwracać. W rzeczywistości każda funkcja zwraca pewną wartość. Jeżeli w funkcji znajduje się instrukcja `return`, funkcja zwróci określoną wartość, wskazaną za pomocą tej instrukcji. W przeciwnym wypadku, gdy dana funkcja nie posiada instrukcji `return`, zostanie zwrócona wartość `None`, czyli tak zwana wartość "pusta", a w innych językach często określana jako `null` lub `nil`.

W Visual Basicu funkcje (te, które zwracają wartość) rozpoczynają się słowem kluczowym `function`, a procedury (z ang. *subroutines*, nie zwracają wartości) deklarujemy za pomocą słowa `sub`. W Pythonie nie ma czegoś takiego jak procedury. Są tylko funkcje, a każda z nich zwraca wartość (nawet jeżeli jest to `None`) i wszystkie rozpoczynają się słowem kluczowym `def`.

Argument `params` nie ma określonego typu. W Pythonie typy zmiennych nie określamy w sposób jawny. Interpreter sam automatycznie rozpoznaje i śledzi typ zmiennej.

W Javie, C++ i innych językach z typami statycznymi musimy określić typ danych zwracany przez funkcję, a także typ każdego argumentu. W Pythonie nigdzie tego nie robimy. Bazując na wartości jaka została przypisana zmiennej, Python sam określa jej typ.

Typy danych w Pythonie a inne języki programowania

Jeśli chodzi o nadawanie typów, języki programowania można podzielić na:

statycznie typowane Są to języki, w których typy są nadawane podczas kompilacji. Wiele tego typu języków programowania wymaga deklarowania wszystkich zmiennych przed ich użyciem, przez podanie ich typu. Przykładami takich języków jest Java, czy też C.

dynamicznie typowane Są to języki, w których typy zmiennych są nadawane podczas działania programu. VBScript i Python są językami dynamicznie typowanymi, ponieważ nadają one typ zmiennej podczas przypisania do niej wartości.

silnie typowane Są to języki, w których między różnymi typami widać wyraźną granicę. Jeśli mamy pewną liczbę całkowitą, to nie możemy jej traktować jak jakiś napis bez wcześniejszego przekonwertowania jej na łańcuch znaków.

słabo typowane Są to języki, w których możemy nie zwracać uwagi na typ zmiennej. Do takich języków zaliczymy *VBScript*. W tym języku możemy, przy tym nie wykonując żadnej wyraźnej konwersji, połączyć łańcuch znaków `'12'` z liczbą całkowitą `3` otrzymując łańcuch `'123'`, a następnie potraktować go jako liczbę całkowitą `123`. Konwersja jest wykonywana automatycznie.

Python jest językiem zarówno dynamicznie typowanym (ponieważ nie wymaga wyraźnej deklaracji typu), jak i silnie typowanym (ponieważ zmienne posiadają wyraźnie ustalone typy, które nie podlegają automatycznej konwersji).

3.3 Dokumentowanie funkcji

Dokumentowanie funkcji

Funkcje można dokumentować wstawiając notkę dokumentacyjną (ang. *doc string*).

Przykład 2.2 Definiowanie notki dokumentacyjnej w funkcji `buildConnectionString`

```
def buildConnectionString(params):
    u"""Tworzy łańcuch znaków na podstawie słownika parametrów.

    Zwraca łańcuch znaków.
    """
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Trzy następujące po sobie cudzysłowy wykorzystuje się do tworzenia ciągu znaków, który może zajmować wiele linii. Wszystko co się znajduje pomiędzy nimi tworzy pojedynczy łańcuch znaków; dotyczy to także znaków *powrotu karetki* i cudzysłowu. Potrójne cudzysłowy możemy używać wszędzie, ale najczęściej wykorzystuje się je do tworzenia notek dokumentacyjnych.

Za pomocą potrójnych cudzysłowów (`"""..."""`) możemy w łatwy sposób zdefiniować łańcuch znaków zawierający pojedyncze jak i podwójne cudzysłowy, podobnie jak w przypadku `qq/../` w Perlu.

Dzięki przedrostkowi `u` Python zapamięta ten łańcuch znaków w taki sposób, że będzie potrafił poprawnie zinterpretować polskie litery. Więcej szczegółów na ten temat dowiemy się w dalszej części tej książki.

W powyższym przykładzie wszystko pomiędzy potrójnymi cudzysłowami jest notką dokumentacyjną funkcji, czyli opisem do czego dana funkcja służy i jak ją używać. Notka dokumentacyjna, o ile istnieje, musi znaleźć się pierwsza w definicji funkcji (czyli zaraz po dwukropku). Python nie wymaga, aby funkcja posiadała notkę dokumentacyjną, lecz powinniśmy ją zawsze tworzyć. Ułatwia ona nam, a także innym, zorientować się w programie. Warto zaznaczyć, że notka dokumentacyjna jest dostępna jako atrybut funkcji nawet w trakcie wykonywania programu.

Wiele Pythonowych IDE wykorzystuje notki dokumentacyjne do "inteligentnej pomocy", czyli sugerowania nazwy funkcji przy jednoczesnym wyświetlaniu informacji o niej (wziętej z notki). Może to stanowić dla nas bardzo dobrą pomoc, oczywiście pod warunkiem, że notki dokumentacyjne zostały przez nas zdefiniowane...

Materiały dodatkowe

Materiały co prawda w języku angielskim, ale na pewno warto je chociaż przejrzeć:

- PEP 257, na temat konwencji notek dokumentacyjnych,
- Python Style Guide, o tym, jak napisać dobrą notkę dokumentacyjną,

3.4 Wszystko jest obiektem

Wszystko jest obiektem

Wspomnieliśmy już wcześniej, że funkcje w Pythonie posiadają **atrybuty**, a one są dostępne podczas pracy programu.

Funkcje, podobnie jak wszystko inne w Pythonie, są **obiektami**.

Otwórzmy swój ulubiony IDE Pythona i wprowadź następujący kod:

Przykład 2.3 Odwoływanie do napisu dokumentacyjnego funkcji `buildConnectionString`

```
>>> import odbchelper                              #(1)
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> print odbchelper.buildConnectionString(params)   #(2)
pwd=secret;database=master;uid=sa;server=mpilgrim
>>> print odbchelper.buildConnectionString.__doc__   #(3)
Tworzy łańcuch znaków na podstawie słownika parametrów.

    Zwraca łańcuch znaków.
```

1. Pierwsza linia importuje program `odbchelper` jako moduł – kawałek kodu, który możemy używać interaktywnie. (W Rozdziale 4 zobaczymy przykłady programów podzielonych na wiele modułów.) Kiedy już zaimportujemy moduł, możemy odwołać się do jego wszystkich publicznych funkcji, klas oraz atrybutów. Moduły także mogą odwoływać się do jeszcze innych modułów.

2. Aby wykorzystać jakąś funkcję zdefiniowaną w zaimportowanym module, musimy przed nazwą funkcji dołączyć nazwę modułu. Nie możemy napisać `buildConnectionString`, lecz zamiast tego możemy dać `odbchelper.buildConnectionString`. Jeśli kiedykolwiek korzystaliśmy z klas w Javie, powinniśmy zauważyć pewne podobieństwa.

3. Tym razem zamiast wywoływać funkcję, zapytaliśmy się o jeden z atrybutów funkcji – atrybut `__doc__`. W nim Python przechowuje notkę dokumentacyjną.

`import` w Pythonie działa podobnie jak `require` w Perlu. Kiedy zaimportujemy jakiś moduł, odwołujemy się do jego funkcji poprzez `modul.funkcja`. W Perlu wygląda to troszkę inaczej, piszemy `modul::funkcja`.

Ścieżka przeszukiwania modułów

Zanim przejdziemy dalej, należy wspomnieć o ścieżce przeszukiwania modułów. W Pythonie przeglądanych jest kilka miejsc w poszukiwaniu importowanego modułu. Generalnie przeszukiwane są wszystkie katalogi zdefiniowane w `sys.path`. Jest to lista, którą możemy w łatwy sposób przeglądać i modyfikować w podobny sposób jak inne listy (jak to robić dowiemy się w kolejnych rozdziałach).

Przykład 2.4 Ścieżka przeszukiwania modułów

```
>>> import sys                      #(1)
>>> sys.path                        #(2)
['', '/usr/local/lib/python2.2', '/usr/local/lib/python2.2/plat-linux2',
'/usr/local/lib/python2.2/lib-dynload', '/usr/local/lib/python2.2/site-packages',
'/usr/local/lib/python2.2/site-packages/PIL', '/usr/local/lib/python2.2/
site-packages/piddle']
>>> sys                             #(3)
<module 'sys' (built-in)>
>>> sys.path.append('/my/new/path') #(4)
```

1. Zaimportowanie modułu `sys` spowoduje, że wszystkie jego funkcje i atrybuty stają się dostępne.

2. `sys.path` to lista nazw katalogów, które są obecnie przeszukiwane podczas importowania modułu. (Zawartość listy zależna jest od systemu operacyjnego, wersji Pythona i położenia jego instalacji, więc na twoim komputerze może wyglądać nieco inaczej.) Python przeszuka te katalogi (w zadanej kolejności) w poszukiwaniu pliku `.py`, który nazywa się tak samo jak importowany moduł.

3. Właściwie to trochę rozminęliśmy się z prawdą. Sytuacja jest bardziej skomplikowana, ponieważ nie wszystkie moduły występują jako pliki z rozszerzeniem `.py`. Niektóre, tak jak `sys` są **wbudowane** w samego Pythona. Wbudowane moduły zachowują się w ten sam sposób co pozostałe, ale nie mamy bezpośredniego dostępu do ich kodu źródłowego, ponieważ nie są napisane w Pythonie (moduł `sys` napisany jest w C).

4. Kiedy dodamy nowy katalog do ścieżki przeszukiwania, Python przy następnych importach przejrzy dodatkowo dodany katalog w poszukiwaniu modułu z rozszerzeniem `.py`. Nowy katalog będzie znajdował się w ścieżkach szukania tak długo, jak długo uruchomiony będzie interpreter. (Dowiesz się więcej o metodzie `append` i innych metodach list w kolejnym rozdziale).

Co to jest obiekt

W Pythonie wszystko jest obiektem i prawie wszystko posiada metody i atrybuty. Każda funkcja posiada wbudowany atrybut `__doc__`, który zwraca napis dokumentacyjny zdefiniowany w kodzie funkcji. Moduł `sys` jest obiektem, który posiada między innymi atrybut `path`.

W dalszym ciągu nie wyjaśniliśmy jeszcze, co to jest obiekt. Każdy język programowania definiuje "obiekt" w inny sposób. W niektórych językach "obiekt" *musi* posiadać atrybuty i metody, a w innych wszystkie "obiekty" mogą dzielić się na różne podklasy. W Pythonie jest inaczej, niektóre obiekty nie posiadają ani atrybutów ani metod (więcej o tym w kolejnym rozdziale) i nie wszystkie obiekty dzielą się na podklasy (więcej o tym w rozdziale 5). Wszystko jest obiektem w tym sensie, że może być przypisane do zmiennej albo stanowić argument funkcji (więcej o tym w rozdziale 4).

Ponieważ jest to bardzo ważne, więc powtórzmy to jeszcze raz: **wszystko w Pythonie jest obiektem**. Łańcuchy znaków to obiekty, listy to obiekty, funkcje to obiekty, a nawet moduły to obiekty...

Materiały dodatkowe

- Python Reference Manual dokładnie opisuje, co to znaczy, że wszystko w Pythonie jest obiektem
- eff-bot podsumowuje obiekty Pythona

3.5 Wcięcia kodu

Wcięcia kodu

Funkcje w Pythonie nie posiadają sprecyzowanych początków i końców oraz żadnych nawiasów służących do zaznaczania, gdzie funkcja się zaczyna, a gdzie kończy. Jedynym separatorem jest dwukropek (`:`) i wcięcia kodu.

Przykład 2.5 Wcięcia w funkcji `buildConnectionString`

```
def buildConnectionString(params):
    u"""Tworzy łańcuch znaków na podstawie słownika parametrów.

    Zwraca łańcuch znaków.
    """
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Bloki kodu definiujemy poprzez wcięcia. Przez "blok kodu" rozumiemy funkcje, instrukcje `if`, pętle `for` i `while` i tak dalej. Wstawiając wcięcie zaczynamy blok, a kończymy go przestając wstawiać wcięcia danej wielkości. Nie ma żadnych nawiasów, klamer czy słów kluczowych. Oznacza to, że białe znaki (spacje itp.) mają znaczenie i ich stosowanie musi być konsekwentne. W powyższym przykładzie kod funkcji (włączając w to notkę dokumentacyjną) został wcięty czterema spacjami. Nie musimy stosować konkretnie czterech spacji, jednak musimy być konsekwentni (tzn. jeśli pierwsze wcięcie w funkcji miało 3 spacje, to kolejne wcięcia także muszą mieć 3 spacje). Linia bez wcięcia znajdować się będzie poza funkcją.

Przykład 2.6 Fragment kodu z wcięciem po instrukcji `if`

```
def fib(n):                        #(1)
    print 'n =', n                 #(2)
    if n > 1:                      #(3)
        return n * fib(n - 1)
    else:                          #(4)
        print 'koniec'
        return 1
```

1. Powyższa funkcja, nazwana `fib` przyjmuje jeden argument: `n`. Cały kod wewnątrz funkcji jest wcięty.

2. Wypisywanie danych (na standardowe wyjście) jest bardzo proste, wystarczy użyć słowa kluczowego `print`. Wyrażenie `print` może przyjąć każdy typ danych, na przykład łańcuchy znaków, liczby całkowite i inne wbudowane typy danych jak słowniki i listy, o których dowiemy się w następnym rozdziale. Możemy nawet drukować na ekran różne wartości w jednej linii. W tym celu podajemy ciąg wartości, które chcemy wyświetlić, oddzielając je przecinkiem. Każda wartość jest wtedy wyświetlana w tej samej linii i oddzielona spacją (znak przecinka nie jest drukowany). Tak więc, kiedy funkcję `fib` wywołamy z argumentem `5`, na ekranie zobaczymy " ".

3. Do bloku kodu zaliczamy także instrukcje `if`. Jeżeli wyrażenie za instrukcją `if` będzie prawdziwe, to zostanie wykonany wcięty kod znajdujący się zaraz pod instrukcją `if`. W przeciwnym wypadku wykonywany jest blok `else`.

4. Oczywiście bloki `if` oraz `else` mogą składać się z większej ilości linii, o ile linie te mają wcięcia z równą ilością spacji. Tutaj blok `else` ma dwie linie. Python nie wymaga żadnej specjalnej składni dla bloków składających się z wielu linii. Po prostu robimy wcięcia o równej liczbie spacji.

Po początkowych problemach i nietrafionych porównaniach do Fortrana, pogodzisz się z tym i zobaczysz pozytywne cechy wcięć. Jedną z głównych zalet jest to, że wszystkie programy w Pythonie wyglądają podobnie, ponieważ wcięcia kodu są wymagane przez sam język i nie zależą od stylu pisania. Dzięki temu jakikolwiek kod jest prostszy do czytania i zrozumienia.

Python używa znaku *powrotu karetki* (ang. *carriage return*), czyli znaku końca linii, by oddzielić instrukcje. Natomiast *dwukropek* i *wcięcia* służą, aby oddzielić bloki kodu. C++ oraz Java używają średników do oddzielania instrukcji, a klamry do separacji bloków kodu.

Materiały dodatkowe

- Python Reference Manual omawia niektóre problemy związane z wcięciami kodu.
- Python Style Guide mówi na temat dobrego stylu tworzenia wcięć.

3.6 Testowanie modułów

Testowanie modułów

Moduły Pythona to obiekty, które posiadają kilka przydatnych atrybutów. Możemy ich użyć do łatwego testowania własnych modułów. Oto przykład zastosowania triku `if __name__`:

```
if __name__ == "__main__":
```

Zwróćmy uwagę na dwie ważne sprawy: pierwsza, że instrukcja warunkowa `if` nie wymaga nawiasów, a druga, że `if` kończy się dwukropkiem, a w następnych liniach wstawiamy wcięty kod.

> Podobnie jak w języku C, Python używa `==` dla porównania i `=` dla przypisania. W przeciwieństwie do C, w Pythonie nie możemy dokonywać przypisania w dowolnych miejscach kodu, w tym w instrukcji warunkowej (np. `if x = 10` nie jest poprawny). Nie ma szansy, aby przypadkowo przypisać wartość do zmiennej, zamiast zastosować porównanie.

Na czym polega zastosowany trik? Moduły to obiekty, a każdy z nich posiada wbudowany atrybut `__name__`. `__name__` zależy od tego, w jaki sposób korzystamy z danego modułu. Jeżeli importujemy moduł, wtedy `__name__` jest nazwą pliku modułu bez ścieżki do katalogu, czy rozszerzenia pliku. Możemy także uruchomić bezpośrednio moduł jako samodzielny program, a wtedy `__name__` przyjmie domyślną wartość `''__main__''`.

```
>>> import odbchelper
>>> odbchelper.__name__
'odbchelper'
```

Wiedząc o tym, możemy zaprojektować test wewnątrz swojego modułu, wykorzystując do tego instrukcję `if`. Jeśli uruchomisz bezpośrednio dany moduł, atrybut `__name__` jest równy `''__main__''`, a więc test zostanie wykonany. Podczas importu tego modułu, `__name__` przyjmie inną wartość, więc test się nie uruchomi. Pomoże nam to rozwijać i wyszukiwać błędy w programie (czyli *debugować*), zanim dany moduł zintegrujemy z większym programem.

> Aby w MacPythonie można było skorzystać z tego triku, należy kliknąć w czarny trójkąt w prawym górnym rogu okna i upewnić się, że zaznaczona jest opcja `Run as __main__`.

Materiały dodatkowe

- Python Reference Manual omawia szczegóły dotyczące importowania modułów, istnieje także polskie tłumaczenie.

Rozdział 4

Wbudowane typy danych

4.1 Łańcuchy znaków i unikod

Za moment powrócimy do naszego pierwszego programu, jednak najpierw musimy zrozumieć, czym są słowniki, krotki i listy. Jeśli choć trochę umiesz programować w Perlu, to pewnie masz już pewną wiedzy na temat słowników, czy list, ale pewnie nie wiesz, czym są krotki. Najpierw zajmiemy się jednym z najczęściej używanych typów danych, czyli łańcuchami znaków.

Łańcuchy znaków i unikod

Łańcuchy znaków służą do przechowywania napisu lub pewnych danych bajtowych. W Pythonie, podobnie jak w większości innych języków programowania tworzymy łańcuchy znaków [1] (ang. *string*) poprzez umieszczenie danego tekstu w cudzysłowach.

Przykład 3.1 Definiowanie łańcucha znaków

```
>>> text = "Nie za długi tekst"      #(1)
>>> text
'Nie za d\xc5\x82ugi tekst'          #(2)
>> print text
Nie za długi tekst                   #(3)
>>> text2 = 'Kolejny napis, ale bez polskich liter'       #(4)
>>> text2
'Kolejny napis, ale bez polskich liter'
>>> text3 = 'Długi tekst,\nktóry po przecinku znajdzie się w następnej linii'  #(5)
>>> print text3
Długi tekst,
który po przecinku znajdzie się w następnej linii
>>> text4 = r'Tutaj znaki specjalne np.\n \t, czy też \x26 nie zostaną zinterpretowane'
>>> print text4
Tutaj znaki specjalne np.\n \t, czy też \x26 nie zostaną zinterpretowane
```

1. W ten sposób stworzyliśmy łańcuch znaków `''Nie za długi tekst''`, który z kolei przypisaliśmy do zmiennej `text`. Zauważmy, że wewnątrz łańcucha mogą się znajdować polskie litery.

2. Otrzymany w tym miejscu wynik na wyjściu może się nieco różnić na różnych systemach. Jest to zależne od kodowania znaków w systemie operacyjnym, z którego korzystamy. Komputer uruchomiony przez autorów korzystał z kodowania UTF-8. Zauważmy, że litera `ł` w systemie UTF-8 to dwa bajty `"\xc5\x82"`.

3. Wszystko zostało ładnie wypisane. Tam gdzie jest `ł` widzimy `ł`, a nie jakieś "krzaki".

4. Łańcuch znaków nie musi być deklarowany w podwójnym cudzysłowie, ale może być też to pojedynczy cudzysłów.

5. Znaki specjalne wstawiamy dodając odwrotny ukośnik (tzw. *backslash*) np. `\n`.

[1] W tym podręczniku łańcuchy znaków będziemy czasami nazywali napisami.

6. Możemy także w stworzyć łańcuch znaków w tzw. sposób surowy. Aby to uczynić, poprzedzamy łańcuch znaków literą `r`. Wewnątrz surowego łańcucha znaków odwrotny ukośnik nie jest interpretowany. Można powiedzieć, że znaki specjalne w takim łańcuchu nie są znakami specjalnymi. To co napiszemy, to będziemy mieli.

Unikod

Ponieważ mówimy w języku polskim, piszemy w tym języku, a ponadto czytamy w tym języku, zapewne chcielibyśmy tworzyć programy, które dobrze sobie dają radę ze znakami tego języka. Doskonałym do tego rozwiązaniem jest unikod (ang. *unicode*). Unikod przechowuje nie tylko polskie znaki, ale jest systemem reprezentowania znaków ze wszystkich języków świata

Przykład 3.2 Definiowanie unikodowego łańcucha znaków

```
>>> text = u"Nie za długi tekst"           #(1)
>>> text
u'Nie za d\u0142ugi tekst'          #(2)
>>> print text
Nie za długi tekst
```

1. Aby utworzyć unikodowy napis, dodajemy przedrostek `u` i tyle.

2. Otrzymasz taki sam wynik. Dane przechowywane w unikodzie nie zależą od systemu kodowania, z którego korzysta Twój komputer.

Pamiętamy, jak w poprzednim rozdziale powiedzieliśmy, że do notek dokumentacyjnych został dodany przedrostek `u`, aby Python potrafił poprawnie zinterpretować polskie znaki. Wtedy właśnie wykorzystaliśmy unikod.

Przykład 3.3 Unikod w `buildConnectionString` def buildConnectionString(params): u"Tworzy łańcuch znaków na podstawie słownika parametrów.

Zwraca łańcuch znaków. "

4.2 Słowniki

Słowniki

Jednym z wbudowanych typów są słowniki (ang. *dictionary*). Określają one wzajemną relację między kluczem, a wartością.

Słowniki w Pythonie są podobne do haszy w Perlu. W Perlu zmienne przechowujące hasz są reprezentowane zawsze przez początkowy znak `%`. W Pythonie typ danych jest automatycznie rozpoznawany. Pythonowy słownik przypomina ponadto instancję klasy `Hashtable` w Javie, a także instancję obiektu `Scripting.Dictionary` w Visual Basicu.

Definiowanie słowników

Przykład 3.4 Definiowanie słowników

```
>>> d = {"server":"mpilgrim", "database":"master"}   #(1)
>>> d
{'database': 'master', 'server': 'mpilgrim'}
>>> d["server"]                                      #(2)
'mpilgrim'
>>> d["database"]                                    #(3)
'master'
>>> d["mpilgrim"]                                    #(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'mpilgrim'
```

1. Najpierw utworzyliśmy nowy słownik z dwoma elementami i przypisaliśmy go do zmiennej `d`. Każdy element w słowniku jest parą *klucz-wartość*, a zbiór elementów jest ograniczony nawiasem klamrowym.

2. `'server'` jest kluczem, a skojarzoną z nim wartością jest `'mpilgrim'`, do której odwołujemy się poprzez `d[‘‘server’’]`.

3. `'database'` jest kluczem, a skojarzoną z nim wartością jest `'master'`, do której odwołujemy się poprzez `d[‘‘database’’]`.

4. Możesz dostać się do wartości za pomocą klucza, ale nie możesz dostać się do klucza za pomocą wartości. Tak więc `d[‘‘server’’]` zwraca `'mpilgrim'`, ale wywołanie `d[‘‘mpilgrim’’]` sprawi, że zostanie rzucony wyjątek, ponieważ `'mpilgrim'` nie jest kluczem słownika `d`.

Modyfikowanie słownika

Przykład 3.5 Modyfikowanie słownika

```
>>> d
{'database': 'master', 'server': 'mpilgrim'}
>>> d["database"] = "pubs"                              #(1)
>>> d
```

```
{'database': 'pubs', 'server': 'mpilgrim'}
>>> d["uid"] = "sa"                                    #(2)
>>> d
{'database': 'pubs', 'uid': 'sa', 'server': 'mpilgrim'}
```

1. Klucze w słowniku nie mogą się powtarzać. Przypisując wartość do istniejącego klucza, będziemy nadpisywać starszą wartość.

2. W każdej chwili możesz dodać nową parę klucz-wartość. Składnia jest identyczna do tej, która modyfikuje istniejącą wartość. Czasami może to spowodować pewien problem. Możemy myśleć, że dodaliśmy nową wartość do słownika, ale w rzeczywistości nadpisaliśmy już istniejącą.

Zauważmy, że słownik nie przechowuje kluczy w sposób posortowany. Wydawałoby się, że klucz `'uid'` powinien się znaleźć za kluczem `'server'`, ponieważ litera *s* jest wcześniej w alfabecie niż *u*. Jednak tak nie jest. Elementy słownika znajdują się w losowej kolejności.

W słownikach nie istnieje określona kolejność układania elementów. Nie możemy dostać się do elementów w słowniku w określonej, uporządkowanej kolejności (np. w porządku alfabetycznym). Oczywiście są sposoby, aby to osiągnąć, ale te "sposoby" nie są wbudowane w sam słownik.

Pracując ze słownikami pamiętajmy, że wielkość liter kluczy ma znaczenie.

Przykład 3.6 Nazwy kluczy są wrażliwe na wielkość liter

```
>>> d = {}
>>> d["klucz"] = "wartość"
>>> d["klucz"] = "inna wartość"                        #(1)
>>> d
{'klucz': 'inna wartość'}
>>> d["Klucz"] = "jeszcze inna wartość"                #(2)
>>> d
{'klucz': 'inna wartość', 'Klucz': 'jeszcze inna wartość'}
```

1. Przypisując wartość do istniejącego klucza zamieniamy starą wartość na nową.

2. Nie jest to przypisanie do istniejącego klucza, a ponieważ łańcuchy znaków w Pythonie są wrażliwe na wielkość liter, dlatego też `'klucz'` nie jest tym samym co `'Klucz'`. Utworzyliśmy nową parę klucz-wartość w słowniku. Obydwa klucze mogą się wydawać podobne, ale dla Pythona są zupełnie inne.

Przykład 3.7 Mieszanie typów danych w słowniku

```
>>> d
{'bazadanych': 'pubs', 'serwer': 'mpilgrim', 'uid': 'sa'}
>>> d["licznik"] = 3                                                        #(1)
>>> d
{'bazadanych': 'pubs', 'serwer': 'mpilgrim', 'licznik': 3, 'uid': 'sa'}
```

```
>>> d[42] = "douglas"                                                    #(2)
>>> d
{42: 'douglas', 'bazadanych': 'pubs', 'serwer': 'mpilgrim', 'licznik': 3, 'uid': 'sa'}
```

1. Słowniki nie są przeznaczone tylko dla łańcuchów znaków. Wartość w słowniku może być dowolnym typem danych: łańcuchem znaków, liczbą całkowitą, obiektem, a nawet innym słownikiem. W pojedynczym słowniku wszystkie wartości nie muszą być tego samego typu; możemy wstawić do niego wszystko, co chcemy.

2. Klucze w słowniku są bardziej restrykcyjne, ale mogą być łańcuchami znaków, liczbami całkowitymi i kilkoma innymi typami. Klucze wewnątrz jednego słownika nie muszą posiadać tego samego typu.

Usuwanie pozycji ze słownika

Przykład 3.8 Usuwanie pozycji ze słownika

```
>>> d
{'licznik': 3, 'bazadanych': 'master', 'serwer': 'mpilgrim', 42: 'douglas',
'uid': 'sa'}
>>> del d[42]
>>> d
{'licznik': 3, 'bazadanych': 'master', 'serwer': 'mpilgrim', 'uid': 'sa'}
>>> d.clear()
>>> d
{}
```

1. Instrukcja `del` każe usunąć określoną pozycję ze słownika, która jest wskazywana przez podany klucz.

2. Instrukcja `clear` usuwa wszystkie pozycje ze słownika. Zbiór pusty ograniczony przez nawiasy klamrowe oznacza, że słownik nie ma żadnego elementu.

Materiały dodatkowe

- How to Think Like a Computer Scientist uczy o słownikach i pokazuje, jak wykorzystać słowniki do tworzenia rzadkich macierzy.
- W Python Knowledge Base możemy znaleść wiele przykładów kodów wykorzystujących słowniki.
- Python Cookbook wyjaśnia, jak sortować wartości słownika względem klucza.
- Python Library Reference opisuje wszystkie metody słownika.

4.3 Listy

Listy

Listy są jednym z najważniejszych typów danych. Można się z nimi spotkać w Visual Basicu. W tym języku są określane jako tablice.

Listy przypominają tablice w Perlu. W Perlu zmienna, która przechowuję listę rozpoczyna się od znaku `@`, natomiast w Pythonie nazwa może być dowolna, ponieważ Python automatycznie rozpoznaje typ.

Listy w Pythonie to coś więcej niż tablice w Javie (chociaż można to traktować jako jedno). Lepszą analogią jest klasa `ArrayList`, w której można przechowywać dowolny obiekt i dynamicznie dodawać nowe pozycje.

Definiowanie list

Przykład 3.9 Definiowanie list

```
>>> li = ["a", "b", "mpilgrim", "z", "przykład"]     #(1)
>>> li
['a', 'b', 'mpilgrim', 'z', 'przykład']
>>> li[0]                                             #(2)
'a'
>>> li[4]                                             #(3)
'przykład'
```

1. Najpierw zdefiniowaliśmy listę pięcioelementową. Zauważmy, że lista zachowuje swój oryginalny porządek i nie jest to przypadkowe. Lista jest uporządkowanym zbiorem elementów ograniczonym nawiasem kwadratowym.
2. Lista może być używana tak jak tablica zaczynająca się od 0. Pierwszym elementem niepustej listy o nazwie `li` jest zawsze `li[0]`.
3. Ostatnim elementem pięcioelementowej listy jest `li[4]`, ponieważ indeksy są liczone zawsze od 0.

Przykład 3.10 Ujemne indeksy w listach

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'przykład']
>>> li[-1]                                       #(1)
'przykład'
>>> li[-3]                                       #(2)
'mpilgrim'
```

1. Za pomocą ujemnych indeksów odnosimy się do elementów idących od końca do początku tzn. `li[-1]` oznacza ostatni element, `li[-2]` przedostatni, `li[-3]` odnosi się do 3 od końca elementu itd. Ostatnim elementem niepustej listy jest zawsze `li[-1]`.

2. Jeśli ciężko ci zrozumieć o co w tym wszystkim chodzi, możesz pomyśleć o tym w ten sposób: `li[-n] == li[len(li) - n]`. `len` to funkcja zwracająca ilość elementów listy. Tak więc w tym przypadku `li[-3] == li[5 - 3] == li[2]`.

Przykład 3.11 Wycinanie list

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'przyklad']
>>> li[1:3]                                   #(1)
['b', 'mpilgrim']
>>> li[1:-1]                                  #(2)
['b', 'mpilgrim', 'z']
>>> li[0:3]                                   #(3)
['a', 'b', 'mpilgrim']
```

1. Możesz pobrać podzbiór listy, który jest nazywany "wycinkiem" (ang. *slice*), poprzez określenie dwóch indeksów. Zwracaną wartością jest nowa lista zawierająca wszystkie elementy z listy rozpoczynające się od pierwszego wskazywanego indeksu (w tym przypadku `li[1]`) i idąc w górę kończy na drugim wskazywanym indeksie, nie dołączając go (w tym przypadku `li[3]`). Kolejność elementów względem wcześniejszej listy jest także zachowana.

2. Możemy także podać ujemną wartość któregoś indeksu. Wycinanie wtedy także dobrze zadziała. Jeśli to pomoże, możemy pomyśleć tak: czytamy listę od lewej do prawej, pierwszy indeks określa pierwszy potrzebny element, a drugi określa element, którego nie chcemy. Zwracana wartość zawiera wszystko między tymi dwoma przedziałami.

3. Listy są indeksowane od zera tzn. w tym przypadku `li[0:3]` zwraca pierwsze trzy elementy listy, rozpoczynając od `li[0]`, a kończąc na `li[2]`, ale nie dołączając `li[3]`.

Przykład 3.12 Skróty w wycinaniu

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'przykład']
>>> li[:3]                                      #(1)
['a', 'b', 'mpilgrim']
>>> li[3:]                                      #(2) (3)
['z', 'przykład']
>>> li[:]                                       #(2) (4)
['a', 'b', 'mpilgrim', 'z', 'przykład']
```

1. Jeśli lewy indeks wynosi `0`, możemy go opuścić, wartość `0` jest domyślna. `li[:3]` jest tym samym, co `li[0:3]` z poprzedniego przykładu.

2. Podobnie, jeśli prawy indeks jest długością listy, możemy go pominąć. Tak więc `li[3:]` jest tym samym, co `li[3:5]`, ponieważ lista ta posiada pięć elementów.

3. Zauważmy pewną symetryczność. W pięcioelementowej liście `li[:3]` zwraca pierwsze 3 elementy, a `li[3:]` zwraca dwa ostatnie (a w sumie $3 + 2 = 5$). W rzeczywistości `li[:n]` będzie zwracał zawsze pierwsze `n` elementów, a `li[n:]`

pozostałą liczbę, bez względu na szerokość listy (n może być większe od długości listy).

4. Jeśli obydwa indeksy zostaną pominięte, wszystkie elementy zostaną dołączone. Nie jest to jednak to samo, co oryginalna lista `li`. Jest to nowa lista, która posiada wszystkie takie same elementy. `li[:]` tworzy po prostu kompletną kopię listy.

Dodawanie elementów do listy

Przykład 3.13 Dodawanie elementów do listy

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'przykład']
>>> li.append("nowy")                                  #(1)
>>> li
['a', 'b', 'mpilgrim', 'z', 'przykład', 'nowy']
>>> li.insert(2, "nowy")                               #(2)
>>> li
['a', 'b', 'nowy', 'mpilgrim', 'z', 'przykład', 'nowy']
>>> li.extend(["dwa", "elementy"])                     #(3)
>>> li
['a', 'b', 'nowy', 'mpilgrim', 'z', 'przykład', 'nowy', 'dwa', 'elementy']
```

1. Dodajemy pojedynczy element do końca listy za pomocą metody `append`.
2. Za pomocą `insert` wstawiamy pojedynczy element do listy. Numeryczny argument jest indeksem, pod którym ma się znaleźć wstawiana wartość; reszta elementów, która znajdowała się pod tym indeksem lub miała większy indeks, zostanie przesunięta o jeden indeks dalej. Zauważmy, że elementy w liście nie muszą być unikalne i mogą się powtarzać; w przykładzie mamy dwa oddzielne elementy z wartością `'nowy'` – `li[2]` i `li[6]`.
3. Za pomocą `extend` łączymy listę z inną listą. Nie możemy wywołać `extend` z wieloma argumentami, trzeba ją wywoływać z pojedynczym argumentem – listą. W tym przypadku ta lista ma dwa elementy.

Przykład 3.14 Różnice między `extend` a `append`

```
>>> li = ['a', 'b', 'c']
>>> li.extend(['d', 'e', 'f'])              #(1)
>>> li
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(li)                                 #(2)
6
>>> li[-1]
'f'
>>> li = ['a', 'b', 'c']
>>> li.append(['d', 'e', 'f'])              #(3)
>>> li
['a', 'b', 'c', ['d', 'e', 'f']]
>>> len(li)                                 #(4)
```

```
4
>>> li[-1]
['d', 'e', 'f']
```

1. Listy posiadają dwie metody – `extend` i `append`, które wyglądają na to samo, ale w rzeczywistości są całkowicie różne. `extend` wymaga jednego argumentu, który musi być listą i dodaje każdy element z tej listy do oryginalnej listy.

2. Rozpoczęliśmy z listą trójelementową (`'a'`, `'b'` i `'c'`) i rozszerzyliśmy ją o inne trzy elementy (`'d'`, `'e'` i `'f'`) za pomocą `extend`, tak więc mamy już sześć elementów.

3. `append` wymaga jednego argumentu, który może być dowolnym typem danych. Metoda ta po prostu dodaje dany element na koniec listy. Wywołaliśmy `append` z jednym argumentem, który jest listą z trzema elementami.

4. Teraz oryginalna lista, pierwotnie zawierająca trzy elementy, zawiera ich cztery. Dlaczego cztery? Ponieważ ostatni element przed chwilą do niej wstawiliśmy. Listy mogą wewnątrz przechowywać dowolny typ danych, nawet inne listy. Nie używajmy `append`, jeśli zamierzamy listę rozszerzyć o kilka elementów.

Przeszukiwanie list

Przykład 3.15 Przeszukiwanie list

```
>>> li
['a', 'b', 'nowy', 'mpilgrim', 'z', 'przykład', 'nowy', 'dwa', 'elementy']
>>> li.index("przykład")                                                  #(1)
5
>>> li.index("nowy")                                                      #(2)
2
>>> li.index("c")                                                         #(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li                                                             #(4)
False
```

1. `index` znajduje pierwsze wystąpienie pewnej wartość w liście i zwraca jego indeks.

2. `index` znajduje pierwsze wystąpienie wartości w liście. W tym przykładzie, `'nowy'` występuje dwa razy – w `li[2]` i `li[6]`, ale metoda `index` będzie zawsze zwracać pierwszy indeks, czyli `2`.

3. Jeśli wartość nie zostanie znaleziona, Python zgłosi wyjątek. Takie zachowanie nie jest często spotykane w innych językach, w wielu językach w takich przypadkach zostaje zwrócony niepoprawny indeks. Takie zachowanie Pythona jest dosyć dobrym posunięciem, ponieważ umożliwia szybkie wychwycenie błędu w kodzie, a dzięki temu program nie będzie błędnie działał operując na niewłaściwym indeksie.

4. Aby sprawdzić czy jakaś wartość jest w liście używamy słowa kluczowego `in`, który zwraca `True`, jeśli wartość zostanie znaleziona lub `False` jeśli nie.

Przed wersją 2.2.1 Python nie posiadał oddzielnego boolowskiego typu danych. Aby to zrekompensować, Python mógł interpretować większość typów danych jako `bool` (czyli wartość logiczną np. w instrukcjach `if`) według poniższego schematu:

- `0` jest fałszem; wszystkie inne liczby są prawdą.
- Pusty łańcuch znaków (`""`) jest fałszem, wszystkie inne są prawdą.
- Pusta lista (`[]`) jest fałszem; wszystkie inne są prawdą.
- Pusta krotka (`()`) jest fałszem; wszystkie inne są prawdą.
- Pusty słownik ({}) jest fałszem; wszystkie inne są prawdą.

Wszystkie powyższe punkty stosowane są w Pythonie 2.2.1 i nowszych, ale obecnie można także używać typu logicznego `bool`, który może przyjmować wartość `True` (prawda) lub `False` (fałsz). Zwróćmy uwagę, że wartości te, tak jak cała składnia języka Python, są wrażliwe na wielkość liter.

Usuwanie elementów z listy

Przykład 3.16 Usuwanie elementów z listy

```
>>> li
['a', 'b', 'nowy', 'mpilgrim', 'z', 'przykład', 'nowy', 'dwa', 'elementy']
>>> li.remove("z")                                                       #(1)
>>> li
['a', 'b', 'nowy', 'mpilgrim', 'przykład', 'nowy', 'dwa', 'elementy']
>>> li.remove("nowy")                                                    #(2)
>>> li
['a', 'b', 'mpilgrim', 'przykład', 'nowy', 'dwa', 'elementy']
>>> li.remove("c")                                                       #(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()                                                             #(4)
'elementy'
>>> li
['a', 'b', 'mpilgrim', 'przykład', 'nowy', 'dwa']
```

1. `remove` usuwa pierwszą występującą wartość w liście.
2. `remove` usuwa tylko pierwszą występującą wartość. W tym przypadku `'nowy'` występuje dwa razy, ale `li.remove("nowy")` usuwa tylko pierwsze wystąpienie.
3. Jeśli wartość nie zostanie znaleziona w liście, Python wygeneruje wyjątek. Naśladuje on w takich sytuacjach postępowanie metody `index`.

4. `pop` jest ciekawą metodą, która wykonuje dwie rzeczy: usuwa ostatni element z listy i zwraca jego wartość. Metoda ta różni się od `li[-1]` tym, że `li[-1]` zwraca jedynie wartość, ale nie zmienia listy, a od `li.remove(value)` tym, że `li.remove(value)` zmienia listę, ale nie zwraca wartości.

Używanie operatorów na listach

Przykład 3.17 Operatory na listach

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['przykład', 'nowy']            #(1)
>>> li
['a', 'b', 'mpilgrim', 'przykład', 'nowy']
>>> li += ['dwa']                             #(2)
>>> li
['a', 'b', 'mpilgrim', 'przykład', 'nowy', 'dwa']
>>> li = [1, 2] * 3                           #(3)
>>> li
[1, 2, 1, 2, 1, 2]
```

1. Aby połączyć dwie listy, można też skorzystać z operatora `+`. Za pomocą `lista = lista + innalista` uzyskujemy ten sam wynik, co za pomocą `list.extend(innalista)`, ale operator `+` zwraca nową listę, podczas gdy `extend` zmienia tylko istniejącą listę. Ogólnie `extend` jest szybszy, szczególnie na dużych listach.

2. Python posiada także operator `+=`. Operacja `li += ['dwa']` jest równoważna `li.extend(['dwa'])`. Operator `+=` działa zarówno na listach, łańcuchach znaków jak i może być nadpisany dla dowolnego innego typu danych.

3. Operator `*` zwielokrotnia podaną listę. `li = [1, 2] * 3` jest równoważne z `li = [1, 2] + [1, 2] + [1, 2]`, które łączy trzy listy w jedną.

Materiały dodatkowe

- How to Think Like a Computer Scientist uczy podstaw związanych z wykorzystywaniem list, a także nawiązuje do przekazywania listy jako argument funkcji.
- Python Tutorial pokazuje, że listy można wykorzystywać jako stos lub kolejkę.
- Python Knowledge Base odpowiada na często zadawane pytania dotyczące list, a także posiada także wiele przykładów kodów źródłowych wykorzystujących listy.
- Python Library Reference opisuje wszystkie metody, które zawiera lista.

4.4 Krotki

Krotki

Krotka (ang. *tuple*) jest niezmienną listą. Zawartość krotki określamy tylko podczas jej tworzenia. Potem nie możemy już jej zmienić.

Przykład 3.18 Definiowanie krotki

```
>>> t = ("a", "b", "mpilgrim", "z", "element")   #(1)
>>> t
('a', 'b', 'mpilgrim', 'z', 'element')
>>> t[0]                                          #(2)
'a'
>>> t[-1]                                         #(3)
'element'
>>> t[1:3]                                        #(4)
('b', 'mpilgrim')
```

1. Krotki definiujemy w identyczny sposób jak listę, lecz z jednym wyjątkiem – zbiór elementów jest ograniczony w nawiasach okrągłych, zamiast w kwadratowych.
2. Podobnie jak w listach, elementy w krotce mają określony porządek. Są one indeksowane od `0`, więc pierwszym elementem w niepustej krotce jest zawsze `t[0]`.
3. Ujemne indeksy idą od końca krotki, tak samo jak w listach.
4. Krotki także można wycinać. Kiedy wycinamy listę, dostajemy nową listę. Podobnie, gdy wycinamy krotkę dostajemy nową krotkę.

Przykład 3.19 Krotki nie posiadają żadnej metody

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'element')
>>> t.append("nowy")                                        #(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")                                           #(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("przyklad")                                     #(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t                                                #(4)
True
```

1. Nie można dodawać elementów do krotki. Krotki nie posiadają metod typu `append`, czy też `extend`.

2. Nie można usuwać elementów z krotki. Nie posiadają one ani metody `remove` ani metody `pop`.

3. Nie można wyszukiwać elementów w krotce. Nie mają one metody `index`.

4. Można jednak wykorzystać operator `in`, aby sprawdzić, czy krotka zawiera dany element.

To w końcu do czego są te krotki przydatne?

- Krotki działają szybciej niż listy. Jeśli definiujemy stały zbiór wartości, który będzie używany tylko do iteracji, skorzystajmy z krotek zamiast z listy.

- Jeśli chcielibyśmy używać danych "zabezpieczonych przed zapisem" (np. po to, żeby program był bezpieczniejszy), wykorzystajmy do tego krotki. Korzystając z krotek, zamiast z list, mamy pewność, że dane w nich zawarte nie zostaną nigdzie zmienione. To trochę tak, jakbyśmy mieli gdzieś ukrytą instrukcję `assert` sprawdzającą, czy dane są stałe. W przypadku, gdy nastąpi próba nadpisania pewnych wartości w krotce, program zwróci wyjątek.

- Pamiętasz, jak powiedzieliśmy, że klucze w słowniku mogą być łańcuchami znaków, liczbami całkowitymi i "kilkoma innymi typami"? Krotki są jednym z tych "innych typów". W przeciwieństwie do list, mogą one zostać użyte jako klucz w słowniku. Dlaczego? Jest to dosyć skomplikowane. Klucze w słowniku muszą być niezmienne. Krotki same w sobie są niezmienne, jednak jeśli wewnątrz krotki znajdzie się lista, to krotka ta nie będzie mogła zostać użyta jako klucz, ponieważ lista jest zmienna. W takim przypadku krotka nie byłaby słownikowo-bezpieczna. Aby krotka mogła zostać wykorzystana jako klucz, musi ona zawierać wyłącznie łańcuchy znaków, liczby i inne słownikowo-bezpieczne krotki.

- Krotki używane są do formatowania tekstu, co zobaczymy wkrótce.

Krotki mogą zostać w łatwy sposób przekonwertowane na listy. Wbudowana funkcja `tuple`, której argumentem jest lista, zwraca krotkę z takimi samymi elementami, natomiast funkcja `list`, której argumentem jest krotka, zwraca listę. W rezultacie `tuple` zamraża listę, a `list` odmraża krotkę.

Materiały dodatkowe

- How to Think Like a Computer Scientist uczy, czym są krotki i pokazuje, jak łączyć je ze sobą.

- Python Knowledge Base pokazuje, jak posortować krotkę.

- Python Tutorial omawia, jak zdefniować krotkę z jednym elementem.

4.5 Deklarowanie zmiennych

Deklarowanie zmiennych

Wiemy już trochę o słownikach, krotkach i o listach, więc wrócimy do przykładowego kodu przedstawionego w rozdziale drugim, do odbchelper.py.

Podobnie jak większość języków programowania Python posiada zarówno zmienne lokalne jak i globalne, choć nie deklarujemy ich w jakiś wyraźny sposób. Zmienne zostają utworzone, gdy przypisujemy im pewne wartości. Natomiast kiedy zmienna wychodzi poza zasięg, zostaje automatycznie usunięta.

Przykład 3.20 Definiowanie zmiennej `myParams`

```
if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
```

Zwróćmy uwagę na wcięcia. Instrukcje warunkowe jako bloki kodu są identyfikowane za pomocą wcięć, podobnie jak funkcje.

Zauważmy też, że dzięki wykorzystaniu backslasha ("\") mogliśmy przypisanie wartości do zmiennej podzielić na kilka linii. Backslashe w Pythonie są specjalnymi znakami, które umożliwiają kontynuację danej instrukcji w następnej linii.

Kiedy polecenie zostanie podzielone na kilka linii za pomocą znaku kontynuacji ("\"), następna linia może zostać wcięta w dowolny sposób. Python nie weźmie tego wcięcia pod uwagę.

Ściśle mówiąc, wyrażenia w nawiasach okrągłych, kwadratowych i klamrowych (jak definiowanie słowników) można podzielić na wiele linii bez używania znaku kontynuacji ("\"). Niektórzy zalecają dodawać backslashe nawet wtedy, gdy nie jest to konieczne. Argumentują to tym, że kod staje się wtedy czytelniejszy. Jest to jednak wyłącznie kwestia gustu.

Wcześniej nigdy nie deklarowaliśmy żadnej zmiennej o nazwie `myParams`, ale właśnie przypisaliśmy do niej wartość. Zachowanie to przypomina trochę VBScript bez instrukcji `option explicit`. Na szczęście, w przeciwieństwie do VBScript, Python nie pozwala odwoływać się do zmiennych, do których nie zostały wcześniej przypisane żadne wartości. Jeśli spróbujemy to zrobić, Python rzuci wyjątek.

Odwoływanie się do zmiennych

Przykład 3.21 Odwoływanie się do niezdefiniowanej zmiennej

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
>>> x = 1
```

```
>>> x
1
```

Kiedyś będziesz za to dziękować...

Wielozmienne przypisania Jednym z lepszych Pythonowych skrótów jest wielozmienne przypisanie (ang. multi-variable assignment), czyli jednoczesne (za pomocą jednego wyrażenia) przypisywanie kilku wartości do kilku zmiennych.

Przykład 3.22 Przypisywanie wielu wartości za pomocą jednego wyrażenia

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v                        #(1)
>>> x
'a'
>>> y
'b'
>>> z
'e'
```

1. `v` jest krotką trzech *elementów*, a `(x, y, z)` jest krotką trzech *zmiennych*. Przypisując jedną krotkę do drugiej, przypisaliśmy każdą z wartości `v` do odpowiednich zmiennych (w odpowiedniej kolejności).

Może to zostać wykorzystane w wielu sytuacjach. Czasami chcemy przypisać pewnym zmiennym pewien zakres wartości np. od `1` do `10`. W języku C możemy utworzyć typy wyliczeniowe (`enum`) poprzez ręczne utworzenie listy stałych i wartości jakie przechowują. Może to być trochę nudną i czasochłonną robotą, w szczególności gdy wartości są kolejnymi liczbami. W Pythonie możemy wykorzystać wbudowaną funkcję `range` i wielozmienne przypisanie. W ten sposób z łatwością przypiszemy kolejne wartości do wielu zmiennych.

Przykład 3.23 Przypisywanie kolejnych wartości

```
>>> range(7)                                                                    #(1)
[0, 1, 2, 3, 4, 5, 6]
>>> (PONIEDZIALEK, WTOREK, SRODA, CZWARTEK, PIATEK, SOBOTA, NIEDZIELA) = range(7) #(2)
>>> PONIEDZIALEK                                                                #(3)
0
>>> WTOREK
1
>>> NIEDZIELA
6
```

1. Wbudowana funkcja `range` zwraca listę liczb całkowitych. W najprostszej formie funkcja ta bierze górną granicę i zwraca listę liczb od `0` do podanej granicy (ale już bez niej). (Możemy także ustawić początkową wartość różną niż `0`, a krok może być inny niż `1`. Aby otrzymać więcej szczegółów wykorzystaj instrukcję `print range.__doc__`.)

2. `PONIEDZIALEK`, `WTOREK`, `SRODA`, `CZWARTEK`, `PIATEK`, `SOBOTA` i `NIEDZIELA` są zmiennymi, które zdefiniowaliśmy. (Ten przykład pochodzi z modułu `calendar`; nazwy zostały spolszczone. Moduł `calendar` umożliwia wyświetlanie kalendarzy, podobnie jak to robi Uniksowy program `cal`. Moduł `calendar` przechowuje dla odpowiednich dni tygodnia odpowiednie stałe.)

3. Teraz każda zmienna ma własną wartość: `PONIEDZIALEK` ma `0`, `WTOREK` ma `1` itd.

Wielozmienne przypisania możemy wykorzystać przy tworzeniu funkcji zwracających wiele wartości w postaci krotki. Zwróconą wartość takiej funkcji możemy potraktować jako normalną krotkę lub też przypisać wszystkie elementy tej krotki do osobnych zmiennych za pomocą wielozmiennego przypisania. Wiele standardowych bibliotek korzysta z tej możliwości np. moduł `os`, który omówimy w rozdziale 6.

Materiały dodatkowe

- Python Reference Manual pokazuje przykłady, kiedy można pominąć znak kontynuacji linii, a kiedy musisz go wykorzystać.
- How to Think Like a Computer Scientist wyjaśnia, jak za pomocą wielozmiennych przypisań zamienić wartości dwóch zmiennych.

4.6 Formatowanie łańcucha znaków

Formatowanie łańcucha znaków

W Pythonie możemy formatować wartości wewnątrz łańcucha znaków. Chociaż możemy tworzyć bardzo skomplikowane wyrażenia, jednak najczęściej w prostych przypadkach wystarczy wykorzystać pole `%s`, aby wstawić pewien łańcuch znaków wewnątrz innego.

> Formatowanie łańcucha znaków w Pythonie używa tej samej składni, co funkcja `sprintf` w języku C.

Przykład 3.24 Pierwsza próba formatowania łańcucha

```
>>> k = "uid"
>>> v = "sa"
>>> "%s=%s" % (k, v)                              #(1)
'uid=sa'
```

1. Rezultatem wykonania tego wyrażenia jest łańcuch znaków. Pierwsze `%s` zostało zastąpione wartością znajdującą się w `k`, a drugie wystąpienie `%s` zostało zastąpione wartością `v`. Wszystkie inne znaki (w tym przypadku znak równości) pozostały w miejscach, w których były.

Zauważmy, że `(k, v)` jest krotką. Niedługo zobaczymy, do czego to może być przydatne.

Mogłoby się wydawać, że formatowanie jest jedną z wielu możliwości połączenia łańcuchów znaków, jednak formatowanie łańcucha znaków nie jest tym samym co łączenie łańcuchów.

Przykład 3.25 Formatowanie tekstu a łączenie

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " nie jest poprawnym hasłem dla " + uid             #(1)
secret nie jest poprawnym hasłem dla sa
>>> print "%s nie jest poprawnym hasłem dla %s" % (pwd, uid)        #(2)
secret nie jest poprawnym hasłem dla sa
>>> userCount = 6
>>> print "Użytkowników: %d" % (userCount, )                   #(3) (4)
Użytkowników: 6
>>> print "Użytkowników: " + userCount                         #(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

1. `+` jest operatorem łączącym łańcuchy znaków.
2. W tym trywialnym przypadku formatowanie daje ten sam wynik co łączenie.

3. `(userCount, )` jest jednoelementową krotką. Składnia ta wygląda trochę dziwnie, jednak takie oznaczenie jest jednoznaczne i wiadomo, że chodzi o krotkę. Zawsze można dołączać przecinek po ostatnim elemencie listy, krotki lub słownika. Jest on jednak wymagany tylko podczas tworzenia jednoelementowej krotki. Jeśli przecinek nie byłby wymagany, Python nie mógłby stwierdzić czy `(userCount)` ma być krotką, czy też tylko wartością zmiennej `userCount`.

4. Formatowanie łańcucha znaków działa z liczbami całkowitymi. W tym celu używamy `%d` zamiast `%s`.

5. Spróbowaliśmy połączyć łańcuch znaków z czymś, co nie jest łańcuchem znaków. Został rzucony wyjątek. W przeciwieństwie do formatowania, łączenie łańcucha znaków możemy wykonywać jedynie na innych łańcuchach.

Tak jak `sprintf` w C, formatowanie tekstu w Pythonie przypomina szwajcarski scyzoryk. Mamy mnóstwo opcji do wyboru, a także wiele pól dla różnych typów wartości.

Przykład 3.26 Formatowanie liczb

```
>>> print "Dzisiejsza cena akcji: %f" % 50.4625          #(1)
Dzisiejsza cena akcji: 50.462500
>>> print "Dzisiejsza cena akcji: %.2f" % 50.4625        #(2)
Dzisiejsza cena akcji: 50.46
>>> print "Zmiana w stosunku do dnia wczorajszego: %+.2f" % 1.5        #(3)
Zmiana w stosunku do dnia wczorajszego: +1.50
```

1. Pole formatowania `%f` traktuje wartość jako liczbę rzeczywistą i pokazuje ją z dokładnością do 6 miejsc po przecinku.

2. Modyfikator ``.2'' pola `%f` umożliwia pokazywanie wartości rzeczywistej z dokładnością do dwóch miejsc po przecinku.

3. Można nawet łączyć modyfikatory. Dodanie modyfikatora `+` pokazuje plus albo minus przed wartością, w zależności od tego jaki znak ma liczba. Modyfikator ``.2'' został na swoim miejscu i nadal nakazuje wyświetlanie liczby z dokładnością dwóch miejsc po przecinku.

Materiały dodatkowe

- Python Library Reference wymienia wszystkie opcje formatowania.
- Effective AWK Programming omawia wszystkie opcje formatowania, a także mówi o innych zaawansowanych technikach.

4.7 Odwzorowywanie listy

Odwzorowywanie list

Jedną z bardzo użytecznych cech Pythona są wyrażenia listowe (ang. *list comprehension*), które pozwalają nam w zwięzły sposób odwzorować pewną listę na inną, wykonując na każdym jej elemencie pewną funkcję.

Przykład 3.27 Wprowadzenie do wyrażeń listowych

```
>>> li = [1, 9, 8, 4]
>>> [element*2 for element in li]      #(1)
[2, 18, 16, 8]
>>> li                                 #(2)
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li]       #(3)
>>> li
[2, 18, 16, 8]
```

1. Aby zrozumieć o co w tym chodzi, spójrzmy na to od strony prawej do lewej. `li` jest listą, którą odwzorowujemy. Python przechodzi po każdym elemencie `li`, tymczasowo przypisując wartość każdego elementu do zmiennej `element`, a następnie wyznacza wartość funkcji `element*2` i wstawia wynik do nowej, zwracanej listy.
2. Wyrażenia listowe nie zmieniają oryginalnej listy.
3. Zwracany wynik możemy przypisać do zmiennej, którą odwzorowujemy. Nie spowoduje to żadnych problemów. Python tworzy nową listę w pamięci, a kiedy operacja odwzorowywania listy dobiegnie końca, do zmiennej zostanie przypisana lista znajdująca się w pamięci.

W funkcji `buildConnectionString` zadeklarowanej w rozdziale 2 skorzystaliśmy z wyrażeń listowych:

```
["%s=%s" % (k, v) for k, v in params.items()]
```

Zauważmy, że najpierw wykonujemy funkcję `items`, znajdującą się w słowniku `params`. Funkcja ta zwraca wszystkie dane znajdujące się w słowniku w postaci listy krotek.

Przykład 3.28 Funkcje `keys`, `values` i `items`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.keys()                        #(1)
['pwd', 'database', 'uid', 'server']
>>> params.values()                      #(2)
['secret', 'master', 'sa', 'mpilgrim']
>>> params.items()                       #(3)
[('pwd', 'secret'), ('database', 'master'), ('uid', 'sa'), ('server', 'mpilgrim')]
```

1. W słowniku metoda `keys` zwraca listę wszystkich kluczy. Lista ta nie jest uporządkowana zgodnie z kolejnością, z jaką definiowaliśmy słownik (pamiętamy, że elementy w słowniku są nieuporządkowane).
2. Metoda `values` zwraca listę wszystkich wartości. Lista ta jest zgodna z porządkiem listy zwracanej przez metodę `keys`, czyli dla wszystkich wartości `x` zachodzi `params.values()[x] == params[params.keys()[x]]`.
3. Metoda `items` zwraca listę krotek w formie `(klucz, wartość)`. Lista zawiera wszystkie dane ze słownika.

Spójrzmy jeszcze raz na funkcję `buildConnectionString`. Przyjmuje ona listę `params.items()`, odwzorowuje ją na nową listę, korzystając dla każdego elementu z formatowania łańcucha znaków. Nowa lista ma tyle samo elementów co `params.items()`, lecz każdy element nowej listy jest łańcuchem znaków, który zawiera zarówno klucz, jak i skojarzoną z nim wartość ze słownika `params`.

Przykład 3.29 Wyrażenia listowe w `buildConnectionString`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.items()
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
>>> [k for k, v in params.items()]                                              #(1)
['server', 'uid', 'database', 'pwd']
>>> [v for k, v in params.items()]                                              #(2)
['mpilgrim', 'sa', 'master', 'secret']
>>> ["%s=%s" % (k, v) for k, v in params.items()]                               #(3)
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

1. Wykonując iteracje po liście `params.items()` używamy dwóch zmiennych. Zauważmy, że w ten sposób korzystamy w pętli z wielozmiennego przypisania. Pierwszym elementem `params.items()` jest `('server', 'mpilgrim')`, dlatego też podczas pierwszej iteracji odwzorowywania listy zmienna `k` będzie przechowywała wartość `'server'`, a `v` wartość `'mpilgrim'`. W tym przypadku ignorujemy wartość `v`, dołączając tylko wartość znajdującą się w `k` do zwracanej listy. Otrzymamy taki sam wynik, gdy wywołamy `params.keys()`.
2. W tym miejscu wykonujemy to samo, ale zamiast zmiennej `v` ignorujemy zmienną `k`. Otrzymamy taki sam wynik, jakbyśmy wywołali `params.values()`.
3. Dzięki temu, że przerobiliśmy obydwa poprzednie przykłady i skorzystaliśmy z formatowania łańcucha znaków, otrzymaliśmy listę łańcuchów znaków. Każdy łańcuch znaków tej listy zawiera zarówno klucz, jak i wartość pewnego elementu ze słownika. Wynik wygląda podobnie do wyjścia pierwszego programu. Ponadto porządek został taki sam, jaki był w słowniku.

Materiały dodatkowe

- Python Tutorial omawia inny sposób odwzorowywania listy – za pomocą wbudowanej funkcji `map`.
- Python Tutorial pokazuje kilka przykładów, jak można korzystać z wyrażeń listowych.

4.8 Łączenie list i dzielenie łańcuchów znaków

Łączenie listy i dzielenie łańcuchów znaków

Mamy listę, której elementy są w formie `klucz=wartość`. Załóżmy, że chcielibyśmy połączyć je wszystkie w pojedynczy łańcuch. Aby to zrobić, wykorzystamy metodę `join` obiektu typu string.

Poniżej został przedstawiony przykład łączenia listy w łańcuch znaków, który wykorzystaliśmy w funkcji `buildConnectionString`:

```
return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Zanim przejdziemy dalej zastanówmy się nad pewną kwestią. Funkcje są obiektami, łańcuchy znaków są obiektami... wszystko jest obiektem. Można by było dojść do wniosku, że także *zmienna* jest obiektem, ale to akurat nie jest prawdą. Spójrzmy na ten przykład i zauważmy, że łańcuch znaków `‘‘;’’` sam w sobie jest obiektem i z niego można wywołać metodę `join`. Zmienne są etykietami dla obiektów.

Metoda `join` łączy elementy listy w jeden łańcuch znaków, a każdy element w zwracanym łańcuchu jest oddzielony od innego separatorem. W naszym przykładzie jest nim `‘‘;’’`, lecz może nim być dowolny łańcuch znaków.

Metoda `join` działa tylko z listami przechowującymi łańcuchy znaków. Nie korzysta ona z żadnych wymuszeń czy konwersji. Łączenie listy, która posiada co najmniej jeden lub więcej elementów niebędących łańcuchem znaków, rzuci wyjątek.

Przykład 3.30 Wyjście odbchelper.py

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['pwd=secret', 'database=master', 'uid=sa', 'server=mpilgrim']
>>> ";".join(["%s=%s" % (k, v) for k, v in params.items()])
'pwd=secret;database=master;uid=sa;server=mpilgrim'
```

Powyższy łańcuch znaków otrzymaliśmy podczas uruchamiania odbchelper.py.

W Pythonie znajdziemy także metodę analogiczną do metody `join`, ale która zamiast łączyć, dzieli łańcuch znaków na listę. Jest to funkcja `split`.

Przykład 3.31 Dzielenie łańcuchów znaków

```
>>> li = ['pwd=secret', 'database=master', 'uid=sa', 'server=mpilgrim']
>>> s = ";".join(li)
>>> s
'pwd=secret;database=master;uid=sa;server=mpilgrim'
>>> s.split(";")                                                    #(1)
['pwd=secret', 'database=master', 'uid=sa', 'server=mpilgrim']
>>> s.split(";", 1)                                                 #(2)
['pwd=secret', 'database=master;uid=sa;server=mpilgrim']
```

1. `split`, przeciwieństwo funkcji `join`, dzieli łańcuch znaków na wieloelementową listę. Separator (w przykładzie `‘‘;’’`) nie będzie występował w żadnym elemencie zwracanej listy, zostanie pominięty.

2. Do funkcji `split` możemy dodać opcjonalny drugi argument, który określa, na jaką maksymalną liczbę kawałków ma zostać podzielony łańcuch. (O opcjonalnych argumentach w funkcji dowiemy się w następnym rozdziale.)

`tekst.split(separator, 1)` jest przydatnym sposobem dzielenia łańcucha na dwa fragmenty. Pierwszy fragment dotyczy łańcucha znajdującego się przed pierwszym wystąpieniem separatora (jest on pierwszym elementem zwracanej listy), a drugi fragment zawiera dalszy fragment łańcucha znajdującego się za pierwszym znalezionym separatorem (jest drugim elementem listy). Dalsze separatory nie są brane pod uwagę.

Materiały dodatkowe

- Python Knowledge Base odpowiada na często zadawane pytania dotyczące łańcuchów znaków, a także posiada wiele przykładów wykorzystywania łańcuchów znaków.
- Python Library Reference wymienia wszystkie metody łańcuchów znaków.
- The Whole Python FAQ wyjaśnia, dlaczego `join` jest metodą łańcucha znaków zamiast listy.

4.9 Kodowanie znaków

Kodowanie znaków

W komputerze pewnym znakom odpowiadają pewne liczby, a kodowanie znaków określa, która liczba odpowiada jakiej literze. W łańcuchu znaków każdy symbol zajmuje 8 bitów, co daje nam do dyspozycji tylko 256 różnych symboli. Podstawowym systemem kodowania jest ASCII. Przyporządkowuje on liczbom z zakresu od 0 do 127 znaki alfabetu angielskiego, cyfry i niektóre inne symbole. Pozostałe standardowe systemy kodowania rozszerzają standard ASCII, dlatego znaki z przedziału od 0 do 127 w każdym systemie kodowania są takie same.

Przykład 3.32 Znaki jako liczby i na odwrót

```
>>> ord('a')                              #(1)
97
>>> chr(99)                              #(2)
'c'
>>> ord('%')
37                              #(3)
>>> chr(115)
's'
>>> chr(261)
Traceback (most recent call last):      #(4)
  File "<stdin>", line 1, in ?
ValueError: chr() arg not in range(261)
>>> chr(188)
'\xbc'                          #(5)
```

1. Funkcji `ord` zwraca liczbę, która odpowiada danemu symbolowi. W tym przypadku literze "a" odpowiada liczba 97.

2. Za pomocą funkcji `chr` dowiadujemy się, jaki znak odpowiada danej liczbie. Liczbie 99 odpowiada znak "c".

3. Procent ("%") odpowiada liczbie 37.

4. Każdy symbol odpowiada liczbie z zakresu od 0 do 255. Liczba 261 nie mieści się w jednym bajcie, dlatego wyskoczył nam wyjątek.

5. Co prawda liczba 188 mieści się w 8-bitach, ale nie mieści się w standardzie ASCII i dlatego tego symbolu Python nie może jednoznacznie zinterpretować. W systemie kodowania ISO 8859-2 liczba ta odpowiada znakowi "ź", ale w systemie kodowania Windows-1250 (znany też jako CP-1250) znakowi "Ľ".

Każdy edytor tekstu zapisuje tworzone przez nas programy korzystając z jakiegoś kodowania, choćby z samego ASCII. Dobrze jest korzystać z edytora, który daje nam możliwość ustawienia kodowania znaków. Kiedy wiemy, w jakim systemie kodowania został zapisany nasz skrypt, powinniśmy jeszcze o tym poinformować Pythona.

Informowanie Pythona o kodowaniu znaków

Wróćmy do odbchelper.py. Na samym początku dodaliśmy linię [2]:

```
#-*- coding: utf-8 -*-
```

W ten sposób ustawiamy kodowanie znaków **danego pliku**, a nie całego programu (program może się składać z wielu plików). Zresztą, jeśli nie zdefiniujemy kodowania znaków, Python nas o tym uprzedzi:

```
sys:1: DeprecationWarning: Non-ASCII character '\textbackslash{}xc5' in file test.py on line 5
but no encoding declared; see http://www.python.org/peps/pep-0263.html for detils
```

Jeśli skorzystaliśmy z innego kodowania znaków, zamiast utf-8 oczywiście napiszemy coś innego. Dodając polskie znaki z reguły korzysta się z kodowania UTF-8 lub ISO-8859-2, a czasami w przypadku Windowsa z Windows-1250.

Ale co wtedy, gdy nie mamy możliwości ustawić kodowania znaków i nie wiemy z jakiego korzysta nasz edytor? Można to sprawdzić metodą prób i błędów:

```
#-*- coding: {tu wstawiamy utf-8, iso-8859-2 lub windows-1250} -*-

print "zażółć gęślą jaźń"
```

A może pora zmienić edytor?

Unikod jeszcze raz

Jak wiemy, unikod jest systemem reprezentowania znaków ze wszystkich różnych języków świata.

Zaraz powrócimy do Pythona.

Notatka historyczna. Przed powstaniem unikodu istniały oddzielne systemy kodowania znaków dla każdego języka, a co przed chwilą trochę omówiliśmy. Każdy z nich korzystał z tych samych liczb (0-255) do reprezentowania znaków danego języka. Niektóre języki (jak rosyjski) miały wiele sprzecznych standardów reprezentowania tych samych znaków. Inne języki (np. japoński) posiadają tak wiele znaków, że wymagają wielu bajtów, aby zapisać cały zbiór jego znaków. Wymiana dokumentów pomiędzy tymi systemami była trudna, ponieważ komputer nie mógł stwierdzić, z którego systemu kodowania skorzystał autor. Komputer widział tylko liczby, a liczby mogą oznaczać różne rzeczy. Zaczęto się zastanawiać nad przechowywaniem tych dokumentów w tym samym miejscu (np. w tej samej tabeli bazy danych); trzeba było przechowywać informacje o kodowaniu każdego kawałku tekstu, a także trzeba było za każdym razem informować o kodowaniu przekazywanego tekstu. Wtedy też zaczęto myśleć o wielojęzycznych dokumentach, w których znaki z wielu języków znajdują się w tym samym dokumencie. (Wykorzystywały one zazwyczaj kod ucieczki, aby przełączyć tryb kodowania; ciach, jesteś w rosyjskim trybie, więc 241 znaczy to; ciach, jesteś w greckim trybie, więc 241 znaczy coś innego itd.) Unikod został zaprojektowany po to, aby rozwiązywać tego typu problemy.

Aby rozwiązać te problemy, unikod reprezentuje wszystkie znaki jako 2-bajtowe liczby, od 0 do 65535 [3] Każda 2-bajtowa liczba reprezentuje unikalny znak, który jest

[2] W tym podręczniku będziemy korzystać z kodowania UTF-8

[3] Niestety ciągle jest to nadmiernym uproszczeniem. Unikod został rozszerzony, aby obsługiwać teksty w starożytnych odmianach chińskiego, koreańskiego, czy japońskiego, ale one mają tak dużo znaków, że 2-bajtowy system nie może reprezentować ich wszystkich.

wykorzystywany w co najmniej jednym języku świata. (Znaki które są wykorzystywane w wielu językach świata, mają ten sam kod numeryczny.) Mamy dokładnie jedną liczbę na znak i dokładnie jeden znak na liczbę. Dane unikodu nigdy nie są dwuznaczne.

7-bitowy ASCII przechowuje wszystkie angielskie znaki za pomocą liczb z zakresu od 0 do 127. (65 jest wielką literą "A", 97 jest małą literą "a" itd.) Język angielski posiada bardzo prosty alfabet, więc może się całkowicie zmieścić w 7-bitowym ASCII. Języki zachodniej Europy jak język francuski, hiszpański, czy też niemiecki, korzystają z systemu kodowania nazwanego ISO-8859-1 (inne określenie to "latin-1"), które korzysta z 7-bitowych znaków ASCII dla liczb od 0 do 127, ale rozszerza zakres 128-255 dla znaków typu "ñ" (241), czy "ü" (252). Także unikod wykorzystuje te same znaki jak 7-bitowy ASCII dla zakresu od 0 do 127, a także te same znaki jak ISO-8859-1 w zakresie od 128 do 255, a potem w zakresie od 256 do 65535 rozszerza znaki do innych języków.

Kiedy korzystamy z danych w postaci unikodu, może zajść potrzeba przekonwertowania danych na jakiś inny system kodowania np. gdy potrzebujemy współdziałać z innym komputerowym systemem, a który oczekuje danych w określonym 1-bajtowym systemie kodowania, czy też wysłać dane na terminal, który nie obsługuje unikodu, czy też do drukarki.

I po tej notatce, powróćmy do Pythona.

Przykład 3.33 Unikod w Pythonie

```
>>> ord(u"ą")
261                           #(1)
>>> print unichr(378)            #(2)
ź
```

1. W unikodzie polski znak "ą" jednoznacznie odpowiada cyfrze 261.
2. Za pomocą funkcji `unichr`, dowiadujemy się jakiemu znakowi odpowiada dana liczba. Liczbie 378 odpowiada polska litera "ź". Python automatycznie zakoduje wypisywany napis unikodowy, aby został poprawnie wyświetlony na naszym systemie.

Dlaczego warto korzystać z unikodu? Jest kilka powodów:

- Unikod bardzo dobrze sobie radzi z różnymi międzynarodowymi znakami.
- Reprezentacja unikodowa jest jednoznaczna; jednej liczbie odpowiada dokładnie jeden znak.
- Nie musimy się zamartwiać szczegółami technicznymi np. czy dwa łańcuchy, które ze sobą łączymy są w takim samym systemie kodowania [4].
- Python potrafi właściwie zinterpretować wszystkie znaki (np. co jest literą, co jest białym znakiem, a co jest cyfrą).
- Korzystając z unikodu zapobiegamy wielu problemom.

Dlatego wszędzie, gdzie będziemy korzystali z polski znaków, będziemy korzystali z unikodu.

[4] W szczególności może się to zrobić niewygodne, kiedy korzystamy tylko ze standardowych łańcuchów znaków, a współpracujące ze sobą moduły korzystają z różnych systemów kodowania np. jedne z ISO 8859-2, a inne z UTF-8.

Materiały dodatkowe

- PEP 0263 wyjaśnia, w jaki sposób skonfigurować kodowanie kodu źródłowego.

4.10 Praca z unikodem

Praca z unikodem

Unikodowymi napisami posługujemy się w identyczny sposób jak normalnymi łańcuchami znaków.

Przykład 3.34 Posługiwanie się unikodem

```
>>> errmsg = u'Nie można otworzyć pliku'  #(1)
>>> print errmsg                          #(2)
Nie można otworzyć pliku
>>> print errmsg + u', brak dostępu.'   #(3)
Nie można otworzyć pliku, brak dostępu.
>>> errmsg.split()                      #(4)
[u'Nie', u'mo\u017cna', u'otworzy\u0107', u'pliku']
>>> print u"Błąd: %s"%errmsg
Błąd: Nie można otworzyć pliku
```

1. Tworzymy unikodowy napis i przypisujemy go do zmiennej `errmsg`.
2. Wypisując dowolny unikod, Python go zakoduje w taki sposób, aby był zgodny z kodowaniem znaków wyjścia, a więc dany napis zostanie zawsze wypisany z polskimi znakami.
3. Z unikodem operujemy identycznie, jak z innymi łańcuchami znaków. Możemy na przykład dwa unikody ze sobą łączyć.
4. Możemy także podzielić unikod na listę.
5. Ponadto, podobnie jak w przypadku standardowego łancucha znaków, możemy też unikod formatować.

Unikod a łańcuchy znaków

Funkcjonalność unikodu możemy łączyć ze standardowymi łańcuchami znaków, o ile z operacji tych jasno wynika, co chcemy osiągnąć.

Przykład 3.35 Unikod w połączeniu z łańcuchami znaków

```
>>> file = 'myfile.txt'
>>> errmsg + ' ' + file         #(1)
u'Nie mo\u017cna otworzy\u0107 pliku myfile.txt'
>>> "%s %s"%(errmsg, file)      #(2)
u'Nie mo\u017cna otworzy\u0107 pliku myfile.txt'
>>> errmsg + ', brak dostępu.'   #(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 11:
ordinal not in range(128)
>>> "Błąd: %s"%errmsg           #(4)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 11:
ordinal not in range(128)
```

1. Unikod możemy połączyć z łańcuchem znaków. Powstaje nowy napis unikodowy.

2. Możemy formatować łańcuch znaków korzystając z unikodowych wartości. Tu także powstaje nowy unikod.

3. Python rzucił wyjątek; nie potrafi przekształcić napisu `', brak dostępu'` na napis unikodowy. Z unikodem możemy łączyć jedynie łańcuchy znaków w systemie ASCII (czyli zawierające jedynie angielskie litery i kilka innych symboli np. przecinki, kropki itp.). W tym przypadku Python nie wie, z jakiego kodowania korzystamy.

4. Tutaj mamy analogiczną sytuację. Python nie potrafi przekształcić napisu `'Błąd: %s'` na unikod i rzuca wyjątek.

Python zakłada, że kodowaniem wszystkich łańcuchów znaków jest ASCII [5], dlatego jeśli mieszamy tekst unikodowy z łańcuchami znaków, powinniśmy dopilnować, aby łańcuchy znaków zawierały znaki należące do ASCII (czyli nie mogą posiadać polskich znaków).

`encode` i `decode`

A co wtedy, gdy chcemy przekształcić unikodowy napis na łańcuch znaków? Łańcuchy znaków są jakoś zakodowane, więc aby stworzyć łańcuch znaków, musimy go na coś zakodować np. ISO 8859-2, czy też UTF-8. W tym celu korzystamy z metody `encode` unikodu.

Przykład 3.36 Metoda `encode`

```
>>> errmsg.encode('iso-8859-2')      #(1)
'Nie mo\xbfna otworzy\xe6 pliku'
>>> errmsg.encode('utf-8')
'Nie mo\xc5\xbcna otworzy\xc4\x87 pliku' #(2)
```

1. Za pomocą metody `encode` informujemy Pythona na jakie kodowanie znaków chcemy zakodować pewien unikod. W tym przypadku otrzymujemy łańcuch znaków zakodowany w systemie ISO 8859-2.

2. Tutaj otrzymujemy ten sam napis, ale zakodowany w systemie UTF-8.

Operację odwrotną, czyli odkodowania, wykonujemy za pomocą funkcji `decode`. Na przykład:

[5] Istnieje możliwość zmiany domyślnego kodowania łańcuchów znaków na inny, ale nie powinno się tego robić. Zmiana domyślnego kodowania wprowadza pewne komplikacje i sprawia, że programy stają się nieprzenośne, dlatego nie będziemy tego opisywać w tej książce.

Przykład 3.37 Metoda `decode`

```
>>> msg = errmsg.encode('utf-8')       #(1)
>>> msg.decode('utf-8')                #(2)
u'Nie mo\u017cna otworzy\u0107 pliku'
>>> print msg.decode('utf-8')
Nie można otworzyć pliku
```

1. W tym miejscu zakodowaliśmy napis `errmsg` na UTF-8.
2. Za pomocą metody `decode` odkodowaliśmy zakodowany łańcuch znaków i zwrócony został unikod.

Łańcuch znaków przechowuje znaki w zakodowanej postaci np. w systemie UTF-8, ISO 8859-1, czy ISO 8859-4; może być wieloznaczny. Natomiast unikod jest jednoznaczny, więc nie jest zakodowany w tego typu systemach kodowania. Ponieważ łańcuch znaków jest *zakodowany*, musimy *odkodować* (za pomocą `decode`), aby otrzymać *niezakodowany* unikod. Z kolei unikod, ponieważ jest *niezakodowany*, musimy zakodować (za pomocą `encode`), aby otrzymać *zakodowany* łańcuch znaków.

4.11 Wbudowane typy danych - podsumowanie

Podsumowanie

Teraz już powinniśmy wiedzieć w jaki sposób działa program odbchelper.py i zrozumnieć, dlaczego otrzymaliśmy takie wyjście.

```
def buildConnectionString(params):
    u"""Tworzy łańcuchów znaków na podstawie słownika parametrów.

    Zwraca łańcuch znaków.
    """
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret"
               }
    print buildConnectionString(myParams)
```

Na wyjściu z programu otrzymujemy:

```
pwd=secret;database=master;uid=sa;server=mpilgrim
```

Zanim przejdziemy do następnego rozdziału, upewnijmy się czy potrafimy:

- używać IDE Pythona w trybie interaktywnym
- napisać program i uruchamiać go przy pomocy twojego IDE lub z linii poleceń
- tworzyć łańcuchy znaków i napisy unikodowe
- importować moduły i wywoływać funkcje w nich zawarte
- deklarować funkcje, używać napisów dokumentacyjnych (ang. *docstring*), zmiennych lokalnych, a także używać odpowiednich wcięć
- definiować słowniki, krotki i listy
- dostawać się do atrybutów i metod dowolnego obiektu, włączając w to łańcuchy znaków, listy, słowniki, funkcje i moduły
- łączyć wartości poprzez formatowanie łańcucha znaków
- odwzorowywać listę na inną listę
- dzielić łańcuch znaków na listę i łączyć listę w łańcuch znaków
- poinformować Pythona, z jakiego kodowania znaków korzystamy
- wykorzystywać metody `encode` i `decode`, aby przekształcić unikod w łańcuch znaków.

Rozdział 5

Potęga introspekcji

5.1 Potęga introspekcji

W tym rozdziale dowiemy się o jednej z mocnych stron Pythona – introspekcji. Jak już wiemy, wszystko w Pythonie jest obiektem, natomiast introspekcja jest kodem, który postrzega funkcje i moduły znajdujące się w pamięci jako obiekty, a także pobiera o nich informacje i operuje nimi.

Nurkujemy

Zacznijmy od kompletnego, działającego programu. Przeglądając kod na pewno rozumiesz już niektóre jego fragmenty. Przy niektórych liniach znajdują się liczby w komentarzach; korzystamy tu z koncepcji, które wykorzystywaliśmy już w rozdziale drugim. Nie przejmuj się, jeżeli nie rozumiesz części tego programu. W rozdziale tym wszystkiego się jeszcze nauczysz.

Przykład 4.1 apihelper.py

```
def info(object, spacing=10, collapse=1): #(1) (2) (3)
    u"""Wypisuje metody i ich notki dokumentacyjne.

    Argumentem może być moduł, klasa, lista, słownik, czy też łańcuch znaków."""
    methodList = [e for e in dir(object) if callable(getattr(object, e))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\textbackslash{}n".join(["%s %s" %
                    (method.ljust(spacing),
                     processFunc(unicode(getattr(object, method).__doc__)))
                    for method in methodList])

if __name__ == "__main__":    #(4) (5)
    print info.__doc__
```

1. Ten moduł ma jedną funkcję `info`. Zgodnie ze swoją deklaracją wymaga ona trzech argumentów: `object`, `spacing` oraz `collapse`. Dwa ostatnie parametry są opcjonalne, co za chwilę zobaczymy.

2. Funkcja `info` posiada wieloliniową notkę dokumentacyjną, która opisuje jej zastosowanie. Zauważ, że funkcja nie zwraca żadnej wartości. Ta funkcja będzie wykorzystywana, aby wykonać pewną czynność, a nie żeby otrzymać pewną wartość.

3. Kod wewnątrz funkcji jest wcięty.

4. Sztuczka z `if __name__` pozwala wykonać programowi coś użytecznego, kiedy zostanie uruchomiony samodzielnie. Jeśli powyższy kod zaimportujemy jako moduł do innego programu, kod pod tą instrukcją nie zostanie wykonany. W tym wypadku program wypisuje po prostu notkę dokumentacyjną funkcji `info`.

5. Instrukcja `if` wykorzystuje `==` (dwa znaki równości), aby porównać dwie wartości. W instrukcji `if` nie musimy korzystać z nawiasów okrągłych.

Funkcja `info` została zaprojektowana tak, aby ułatwić sobie pracę w IDE Pythona. IDE bierze jakiś obiekt, który posiada funkcje lub metody (jak na przykład moduł zawierający funkcje lub listę, która posiada metody) i wyświetla funkcje i ich notki dokumentacyjne.

Przykład 4.2 Proste zastosowanie apihelper.py

```
>>> from apihelper import info
>>> li = []
>>> info(li)
[...ciach...]
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(iterable) -- extend list by appending elements from the iterable
index      L.index(value, [start, [stop]]) -> integer -- return first index of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
cmp(x, y) -> -1, 0, 1
```

Domyślnie wynik jest formatowany tak, by był łatwy do odczytania. Notki dokumentacyjne składające się z wielu linii zamieniane są na jednoliniowe, ale tę opcję możemy zmienić ustawiając wartość `0` dla argumentu `collapse`. Jeżeli nazwa funkcji jest dłuższa niż *10 znaków*, możemy określić inną wartość dla argumentu `spacing`, by ułatwić sobie czytanie.

Przykład 4.3 Zaawansowane użycie apihelper.py

```
>>> import odbchelper
>>> info(odbchelper)
buildConnectionString Tworzy łańcuchów znaków na podstawie słownika parametrów.
 Zwraca łańcuch znaków.
>>> info(odbchelper, 30)
buildConnectionString          Tworzy łańcuchów znaków na podstawie słownika parametrów.
Zwraca łańcuch znaków.
>>> info(odbchelper, 30, 0)
buildConnectionString          Tworzy łańcuchów znaków na podstawie słownika parametrów.

    Zwraca łańcuch znaków.
```

5.2 Argumenty opcjonalne i nazwane

Argumenty opcjonalne i nazwane

W Pythonie argumenty funkcji mogą posiadać wartości domyślne. Jeżeli funkcja zostanie wywołana bez podania pewnego argumentu, argumentowi temu zostanie przypisana jego domyślna wartość. Co więcej możemy podawać argumenty w dowolnej kolejności poprzez użycie ich nazw.

Poniżej przykład funkcji `info` z dwoma argumentami opcjonalnymi:

```
def info(object, spacing=10, collapse=1):
```

`spacing` oraz `collapse` są argumentami opcjonalnymi, ponieważ mają przypisane wartości domyślne. Argument `object` jest wymagany, ponieważ nie posiada wartości domyślnej. Jeżeli `info` zostanie wywołana tylko z jednym argumentem, `spacing` przyjmie wartości `10`, a `collapse` wartość `1`. Jeżeli wywołamy tę funkcję z dwoma argumentami, jedynie `collapse` przyjmuje wartość domyślną (czyli `1`).

Załóżmy, że chcielibyśmy określić wartość dla `collapse`, ale dla argumentu `spacing` chcielibyśmy skorzystać z domyślnej wartości. W większości języków programowania jest to niewykonalne, ponieważ wymagają one od nas wywołania funkcji z trzema argumentami. Na szczęście w Pythonie możemy określać argumenty w dowolnej kolejności poprzez odwołanie się do ich nazw.

Przykład 4.4 Różne sposoby wywołania funkcji `info`

```
info(odbchelper)                     #(1)
info(odbchelper, 12)                 #(2)
info(odbchelper, collapse=0)         #(3)
info(spacing=15, object=odbchelper)  #(4)
```

1. Kiedy wywołamy tę funkcję z jednym argumentem, `spacing` przyjmie wartość domyślną równą `10`, a `collapse` wartość `1`.
2. Kiedy podamy dwa argumenty, `collapse` przyjmie wartość domyślną, czyli `1`.
3. Tutaj podajemy argument `collapse` odwołując się do jego nazwy i określamy wartość, którą chcemy mu przypisać. `spacing` przyjmuje wartość domyślną – `10`.
4. Nawet wymagany argument (jak `object`, który nie posiada wartości domyślnej) może zostać określony poprzez swoją nazwę i może wystąpić na jakimkolwiek miejscu w wywołaniu funkcji.

Takie działanie może się wydawać trochę niejasne, dopóki nie zdamy sobie sprawy, że lista argumentów jest po prostu słownikiem. Gdy wywołujemy daną funkcję w sposób "normalny", czyli bez podawania nazw argumentów, Python dopasowuje wartości do określonych argumentów w takiej kolejności w jakiej zostały zadeklarowane. Najczęściej będziemy wykorzystywali tylko "normalne" wywołania funkcji, ale zawsze mamy możliwość bardziej elastycznego podejścia do określania kolejności argumentów.

Jedyną rzeczą, którą musimy zrobić by poprawnie wywołać funkcję jest określenie w jakikolwiek sposób wartości dla każdego *wymaganego* argumentu. Sposób i kolejność określania argumentów zależy tylko od nas.

Materiały dodatkowe

- Python Tutorial omawia w jaki sposób domyślne wartości są określane, czyli co się stanie, gdy domyślny argument będzie listą lub też pewnym wyrażeniem.

5.3 Dwa sposoby importowania modułów

Dwa sposoby importowania modułów

W Pythonie mamy dwa sposoby importowania modułów. Obydwa są przydatne, dlatego też powinniśmy umieć je wykorzystywać. Jednym ze sposobów jest użycie polecenia `import module`, który mogliśmy zobaczyć w podrozdziale "Wszystko jest obiektem". Istnieje inny sposób, który realizuje tę samą czynność, ale posiada pewne różnice. Poniżej został przedstawiony przykład wykorzystujący instrukcję `from module import`:

```
from apihelper import info
```

Jak widzimy, składnia tego wyrażenia jest bardzo podobna do `import module`, ale z jedną ważną różnicą: atrybuty i metody danego modułu są importowane bezpośrednio do lokalnej przestrzeni nazw, a więc będą dostępne bezpośrednio i nie musimy określać, z którego modułu korzystamy. Możemy importować określone pozycje albo skorzystać z `from module import *`, aby zaimportować wszystko.

`from module import *` w Pythonie przypomina `use module` w Perlu, a Pythonowe `import module` przypomina Perlowskie `require module`.

`from module import *` w Pythonie jest analogią do `import module.*` w Javie, a `import module` w Pythonie przypomina `import module` w Javie.

Przykład 4.5 Różnice między `import module` a `from module import`

```
>>> import types
>>> types.FunctionType                      #(1)
<type 'function'>
>>> FunctionType                            #(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'FunctionType' is not defined
>>> from types import FunctionType          #(3)
>>> FunctionType                            #(4)
<type 'function'>
```

1. Moduł `types` nie posiada żadnych metod. Posiada on jedynie atrybuty określające wszystkie typy zdefiniowane przez Pythona. Zauważmy, że atrybut tego modułu (w tym przypadku `FunctionType`) musi być poprzedzony nazwą modułu – `types`.

2. `FunctionType` nie został sam w sobie określony w przestrzeni nazw; istnieje on tylko w kontekście modułu `types`.

3. Za pomocą tego wyrażenia atrybut `FunctionType` z modułu `types` został zaimportowany bezpośrednio do lokalnej przestrzeni nazw.

4. Teraz możemy odwoływać się bezpośrednio do `FunctionType`, bez odwoływania się do `types`.

Kiedy powinniśmy używać `from module import`?

- Kiedy często odwołujemy się do atrybutów i metod, a nie chcemy wielokrotnie wpisywać nazwy modułu, wtedy najlepiej wykorzystać `from module import`.
- Jeśli potrzebujemy selektywnie zaimportować tylko kilka atrybutów lub metod, powinniśmy skorzystać z `from module import`.
- Jeśli moduł zawiera atrybuty lub metody, które posiadają taką samą nazwę jaka jest w naszym module, powinniśmy wykorzystać `import module`, aby uniknąć konfliktu nazw.

W pozostałych przypadkach to kwestia stylu programowania, można spotkać kod napisany obydwoma sposobami.

Używajmy `from module import *` oszczędnie, ponieważ taki sposób importowania utrudnia określenie, skąd pochodzi dana funkcja lub atrybut, a to z kolei utrudnia debugowanie.

Materiały dodatkowe

- eff-bot opowie nam więcej na temat różnic między `import module` a `from module import`.
- Python Tutorial omawia zaawansowane techniki importu, włączając w to `from module import *`.

5.4 type, str, dir i inne wbudowane funkcje

type, str, dir i inne wbudowane funkcje

Python posiada mały zbiór bardzo użytecznych wbudowanych funkcji. Wszystkie inne funkcje znajdują się w różnych modułach. Była to świadoma decyzja projektowa, aby uniknąć przeładowania rdzenia języka, jak to ma miejsce w przypadku innych języków (jak np. Visual Basic czy Object Pascal).

Funkcja type

Funkcja `type` zwraca typ danych podanego obiektu. Wszystkie typy znajdują się w module `types`. Funkcja ta może się okazać przydatna podczas tworzenia funkcji obsługujących kilka typów danych.

Przykład 4.6 Wprowadzenie do type

```
>>> type(1)           #(1)
<type 'int'>
>>> li = []
>>> type(li)          #(2)
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)  #(3)
<type 'module'>
>>> import types      #(4)
>>> type(odbchelper) == types.ModuleType
True
```

1. Argumentem `type` może być cokolwiek: stała, łańcuch znaków, lista, słownik, krotka, funkcja, klasa, moduł, wszystkie typy są akceptowane.
2. Kiedy podamy funkcji `type` dowolną zmienną, zostanie zwrócony jej typ.
3. `type` także działa na modułach.
4. Możemy używać stałych z modułu `types`, aby porównywać typy obiektów. Wykorzystuje to funkcja `info`, co wkrótce zobaczymy.

Funkcja str

Funkcja `str` przekształca dane w łańcuch znaków. Każdy typ danych może zostać przekształcony w łańcuch znaków.

Przykład 4.7 Wprowadzenie do str

```
>>> str(1)                                       #(1)
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen
['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
```

```
>>> str(horsemen)                                #(2)
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbchelper)                              #(3)
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> str(None)                                    #(4)
'None'
```

1. Można było się spodziewać, że `str` działa na tych prostych, podstawowych typach takich jak np. liczby całkowite. Prawie każdy język programowania posiada funkcję konwertującą liczbę całkowitą na łańcuch znaków.

2. Jakkolwiek funkcja `str` działa na każdym obiekcie dowolnego typu, w tym przypadku jest to lista składająca się z kilku elementów.

3. Argumentem funkcji `str` może być także moduł. Zauważmy, że łańcuch reprezentujący moduł zawiera ścieżkę do miejsca, w którym się ten moduł znajduje. Na różnych komputerach może być ona inna.

4. Subtelnym, lecz ważnym zachowaniem funkcji `str` jest to, że argumentem może być nawet wartość `None` (Pythonowej wartości pusta, często określanej w innych językach przez `null`). Dla takiego argumentu funkcja zwraca napis `'None'`. Wkrótce wykorzystamy tę możliwość.

Funkcja `unicode`

Funkcja `unicode` pełni tą samą funkcję, co `str`, ale zamiast łańcucha znaków tworzy unikod.

Przykład 4.8 Wprowadzenie do `unicode`

```
>>> unicode(1)                              #(1)
u'1'
>>> unicode(odbchelper)                     #(2)
u"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> print unicode(horsemen[0])
u'war'
>>> unicode('jeździectwo')                  #(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc5 in position 2:
ordinal not in range(128)
>>> unicode('jeździectwo', 'utf-8')         #(4)
u'je\u017adziectwo'
```

1. Podobnie, jak w przypadku `str`, do funkcji `unicode` możemy przekazać dowolny obiekt np. może to być liczba. Przekonwertowaliśmy liczbę na napis unikodowy.

2. Argumentem funkcji `unicode` może być także moduł.

3. Ponieważ litera "ź" nie należy do ASCII, więc Python nie potrafi jej zinterpretować. Zostaje rzucony wyjątek.

4. Do funkcji `unicode` możemy przekazać drugi, opcjonalny argument `encoding`, który określa, w jakim systemie kodowania jest zakodowany łańcuch znaków. Komputer, na którym został uruchomiony ten przykład, korzystał z kodowania UTF-8, więc przekazany łańcuch znaków także będzie w tym systemie kodowania.

Funkcja `dir`

Kluczową funkcją wykorzystaną w `info` jest funkcja `dir`. Funkcja ta zwraca listę atrybutów i metod pewnego obiektu np. modułu, funkcji, łańcuch znaków, listy, słownika... niemal wszystkiego.

Przykład 4.9 Wprowadzenie do `dir`

```
>>> li = []
>>> dir(li)                                        #(1)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__',
'__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__setslice__', '__str__',
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> d = {}
>>> dir(d)                                         #(2)
[[...,'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems',
'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update',
'values']
>>> import odbchelper
>>> dir(odbchelper)                                #(3)
['__builtins__', '__doc__', '__file__', '__name__', 'buildConnectionString']
```

1. `li` jest listą, dlatego też `dir(li)` zwróci nam listę wszystkich metod, które posiada lista. Zwróćmy uwagę na to, że zwracana lista zawiera *nazwy metod* w formie łańcucha znaków, a nie metody same w sobie. Metody zaczynające się i kończące dwoma dolnymi myślnikami są metodami specjalnymi.

2. `d` jest słownikiem, dlatego `dir(d)` zwróci listę nazw metod słownika. Co najmniej jedna z nich, metoda `keys`, powinna wyglądać znajomo.

3. Dzięki temu funkcja ta staje się interesująca. `odbchelper` jest modułem, więc za pomocą `dir(odbchelper)` otrzymamy listę nazw atrybutów tego modułu włączając w to wbudowane atrybuty np. `__name__`, czy też `__doc__`, a także jakiekolwiek inne np. zdefiniowane przez nas funkcje. W tym przypadku `odbchelper` posiada tylko jedną, zdefiniowaną przez nas metodę – funkcję `buildConnectionString` opisaną w rozdziale "Pierwszy program".

Funkcja `callable`

Funkcja `callable` zwraca `True`, jeśli podany obiekt może być wywoływany, a `False` w przeciwnym przypadku. Do wywoływalnych obiektów zaliczamy funkcje, metody klas, a nawet same klasy. (Więcej o klasach możemy przeczytać w następnym rozdziale.)

Przykład 4.10 Wprowadzenie do `callable`

```
>>> import string
>>> string.punctuation                                  #(1)
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join                                         #(2)
<function join at 00C55A7C>
>>> callable(string.punctuation)                        #(3)
False
>>> callable(string.join)                               #(4)
True
>>> print string.join.__doc__                           #(5)
join(list [,sep]) -> string

    Return a string composed of the words in list, with
    intervening occurrences of sep.  The default separator is a
    single space.

    (joinfields and join are synonymous)
```

1. Nie zaleca się, żeby wykorzystywać funkcje z modułu `string` (chociaż wciąż wiele osób używa funkcji `join`), ale moduł ten zawiera wiele przydatnych stałych jak np. `string.punctuation`, który zawiera wszystkie standardowe znaki przestankowe, więc z niego tutaj skorzystaliśmy.
2. Funkcja `string.join` łączy listę w łańcuch znaków.
3. `string.punctuation` nie jest wywoływalny, jest łańcuchem znaków. (Typ `string` posiada metody, które możemy wywoływać, lecz sam w sobie nie jest wywoływalny.)
4. `string.join` można wywołać. Jest to funkcja przyjmująca dwa argumenty.
5. Każdy wywoływalny obiekt może posiadać notkę dokumentacyjną. Kiedy wykonamy funkcję `callable` na każdym atrybucie danego obiektu, będziemy mogli potencjalnie określić, którymi atrybutami chcemy się bardziej zainteresować (metody, funkcje, klasy), a które chcemy pominąć (stałe itp.).

Wbudowane funkcje

`type`, `str`, `unicode`, `dir` i wszystkie pozostałe wbudowane funkcje są umieszczone w specjalnym module o nazwie `__builtin__` (nazwa z dwoma dolnymi myślnikami przed i po nazwie). Jeśli to pomoże, możemy założyć, że Python automatycznie wykonuje przy starcie polecenie `from __builtin__ import *`, które bezpośrednio importuje wszystkie wbudowane funkcje do używanej przez nas przestrzeni nazw. Zaletą tego, że funkcje te znajdują się w module, jest to, że możemy dostać informacje o wszystkich wbudowanych funkcjach i atrybutach poprzez moduł `__builtin__`. Wykorzystajmy funkcje `info` podając jako argument ten moduł i przejrzyjmy wyświetlony spis. Niektóre z ważniejszych funkcji w module `__builtin__` zgłębimy później. (Niektóre z wbudowanych klas błędów np. AttributeError, powinny wyglądać znajomo.).

Przykład 4.11 Wbudowane atrybuty i funkcje

```
>>> from apihelper import info
>>> import __builtin__
>>> info(__builtin__, 20)
ArithmeticError      Base class for arithmetic errors.
AssertionError       Assertion failed.
AttributeError       Attribute not found.
EOFError             Read beyond end of file.
EnvironmentError     Base class for I/O related errors.
Exception            Common base class for all exceptions.
FloatingPointError   Floating point operation failed.
IOError              I/O operation failed.

[...ciach...]
```

Wraz z Pythonem dostajemy doskonałą dokumentację, zawierającą wszystkie potrzebne informacje o modułach oferowanych przez ten język. Porównując do innych języków, z Pythonem nie dostajemy podręcznika `man`, czy odwołań do innych zewnętrznych podręczników, wszystko co potrzebujemy znajdujemy wewnątrz samego Pythona.

Materiały dodatkowe

- Python Library Reference dokumentuje wszystkie wbudowane funkcje i wszystkie wbudowane wyjątki.

5.5 Funkcja getattr

Funkcja `getattr`

Powinniśmy już wiedzieć, że w Pythonie funkcje są obiektami. Ponadto możemy dostać referencję do funkcji bez znajomości jej nazwy przed uruchomieniem programu. W tym celu podczas działania programu należy wykorzystać funkcję `getattr`.

Przykład 4.12 Funkcja `getattr`

```
>>> li = ["Larry", "Curly"]
>>> li.pop                                 #(1)
<built-in method pop of list object at 010DF884>
>>> getattr(li, "pop")                     #(2)
<built-in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe")           #(3)
>>> li
["Larry", "Curly", "Moe"]
>>> getattr({}, "clear")                   #(4)
<built-in method clear of dictionary object at 00F113D4>
>>> getattr((), "pop")                     #(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

1. Dzięki temu dostaliśmy referencję do metody `pop`. Zauważmy, że w ten sposób nie wywołujemy metody `pop`; aby ją wywołać musielibyśmy wpisać polecenie `li.pop()`. Otrzymujemy referencję do tej metody. (Adres szesnastkowy wygląda inaczej na różnych komputerach, dlatego wyjścia będą się nieco różnić.)

2. Operacja ta także zwróciła referencję do metody `pop`, lecz tym razem nazwa metody jest określona poprzez łańcuch znaków w argumencie funkcji `getattr`. `getattr` jest bardzo przydatną, wbudowaną funkcją, która zwraca pewien atrybut dowolnego obiektu. Tutaj wykorzystujemy obiekt, który jest listą, a atrybutem jest metoda `pop`.

3. Dzięki temu przykładowi możemy zobaczyć, jaki duży potencjał kryje się w funkcji `getattr`. W tym przypadku zwracaną wartością `getattr` jest metoda (referencja do metody). Metodę tę możemy wykonać podobnie, jak byśmy bezpośrednio wywołali `li.append(''Moe'')`. Tym razem nie wywołujemy funkcji bezpośrednio, lecz określamy nazwę funkcji za pomocą łańcucha znaków.

4. `getattr` bez problemu pracuje na słownikach

5. Teoretycznie `getattr` powinien pracować na krotkach, jednak krotki nie posiadają żadnej metody, dlatego `getattr` spowoduje wystąpienie wyjątku związanego z brakiem atrybutu o podanej nazwie.

`getattr` na modułach

`getattr` działa nie tylko na wbudowanych typach danych. Argumentem tej funkcji może być także moduł.

Przykład 4.13 Funkcja `getattr` w apihelper.py

```
>>> import odbchelper
>>> odbchelper.buildConnectionString             #(1)
<function buildConnectionString at 00D18DD4>
>>> getattr(odbchelper, "buildConnectionString") #(2)
<function buildConnectionString at 00D18DD4>
>>> object = odbchelper
>>> method = "buildConnectionString"
>>> getattr(object, method)                      #(3)
<function buildConnectionString at 00D18DD4>
>>> type(getattr(object, method))                #(4)
<type 'function'>
>>> import types
>>> type(getattr(object, method)) == types.FunctionType
True
>>> callable(getattr(object, method))            #(5)
True
```

1. Polecenie to zwraca nam referencję do funkcji `buildConnectionString` z modułu `odbchelper`, który przeanalizowaliśmy w Rozdziale 2.

2. Wykorzystując `getattr`, możemy dostać taką samą referencję, do tej samej funkcji. W ogólności `getattr(obiekt, ''atrybut'')` jest odpowiednikiem `obiekt.atrybut`. Jeśli obiekt jest modułem, atrybutem może być cokolwiek zdefiniowane w tym module np. funkcja, klasa czy zmienna globalna.

3. Tę możliwość wykorzystaliśmy w funkcji `info`. Obiekt o nazwie `object` został przekazany jako argument do funkcji `getattr`, ponadto przekazaliśmy nazwę pewnej metody lub funkcji jako zmienną `method`.

4. W tym przypadku zmienna `method` przechowuję nazwę funkcji, co można sprawdzić pobierając typ zwracanej wartości.

5. Ponieważ zmienna `method` jest funkcją, więc można ją wywoływać. Zatem w wyniku wywołania `callable` otrzymaliśmy wartość `True`.

`getattr` jako funkcja pośrednicząca

Funkcja `getattr` jest powszechnie używana jako funkcja pośrednicząca (ang. *dispatcher*). Na przykład mamy napisany program, który może wypisywać dane w różnych formatach (np. HTML i PS). Wówczas dla każdego formatu wyjścia możemy zdefiniować odpowiednią funkcje, a podczas wypisywania danych na wyjście `getattr` będzie nam pośredniczył między tymi funkcjami. Jeśli wydaje się to trochę zagmatwane, zaraz zobaczymy przykład.

Wyobraźmy sobie program, który potrafi wyświetlać statystyki strony w formacie HTML, XML i w czystym tekście. Wybór właściwego formatu może być określony w linii poleceń lub przechowywany w pliku konfiguracyjnym. Moduł `statsout` definiuje trzy funkcje – `output_html`, `output_xml` i `output_text`, a wówczas program główny może zdefiniować pojedynczą funkcję, która wypisuje dane na wyjście:

Przykład 4.14 Pośredniczenie za pomocą `getattr`

```
import statsout

def output(data, format="text"):                              #(1)
    output_function = getattr(statsout, "output_%s" % format)   #(2)
    return output_function(data)                               #(3)
```

1. Funkcja `output` wymaga jednego argumentu o nazwie `data`, który ma zawierać dane do wypisania na wyjście. Funkcja ta może także przyjąć jeden opcjonalny argument `format`, który określa format wyjścia. Gdy argument `format` nie zostanie określony, przyjmie on wartość `''text''`, a funkcja się zakończy wywołując funkcję `output_text`, która wypisuje dane na wyjście w postaci czystego tekstu.

2. Łańcucha znaków `''output_''` połączyliśmy z argumentem `format`, aby otrzymać nazwę funkcji. Następnie pobraliśmy funkcję o tej nazwie z modułu `statsout`. Dzięki temu w przyszłości będzie łatwiej rozszerzyć program nie zmieniając funkcji pośredniczącej, aby obsługiwał więcej wyjściowych programów. W tym celu wystarczy dodać odpowiednią funkcję do `statsout` np. `output_pdf` i wywołujemy funkcję `output` podając argument `format` jako `''pdf''`.

3. Teraz możemy wywołać funkcję wyjściową w taki sam sposób jak inne funkcje. Zmienna `output_function` jest referencją do odpowiedniej funkcji w `statsout`.

Czy znaleźliśmy błąd w poprzednim przykładzie? Jest to bardzo niestabilne rozwiązanie, ponadto nie ma tu kontroli błędów. Co się stanie gdy użytkownik poda format, który nie zdefiniowaliśmy w `statsout`? Funkcja `getattr` rzuci nam wyjątek związany z błędnym argumentem, czyli podaną nazwą funkcji, która nie istnieje w module `statsout`.

Na szczęście do funkcji `getattr` możemy podać trzeci, opcjonalny argument, czyli domyślnie zwracaną wartość, gdy podany atrybut nie istnieje.

Przykład 4.15 Domyślnie zwracana wartość w `getattr`

```
import statsout

def output(data, format="text"):
    output_function = getattr(statsout, "output_%s" % format, statsout.output_text)
    return output_function(data)        #(1)
```

1. Ta funkcja już na pewno będzie działała poprawnie, ponieważ podaliśmy trzeci argument w wywołaniu funkcji `getattr`. Trzeci argument jest domyślną wartością, która zostanie zwrócona, gdy podany atrybut, czy metoda nie zostanie znaleziona.

Jak mogliśmy zauważyć, funkcja `getattr` jest niezwykle użyteczna. Jest ona sercem introspekcji. W następnych rozdziałach zobaczymy jeszcze więcej przydatnych przykładów.

5.6 Filtrowanie listy

Filtrowanie listy

Jak już wiemy, Python ma potężne możliwości odwzorowania list w inne listy poprzez wyrażenia listowe (rozdział "Odwzorowywanie listy"). Wyrażenia listowe możemy też łączyć z mechanizmem filtrowania, dzięki któremu pewne elementy listy są odwzorowywane a pewne pomijane.

Poniżej przedstawiono składnię filtrowania listy:

```
[wyrażenie odwzorowujące for element in odwzorowywana lista if wyrażenie filtrujące]
```

Jest to wyrażenie listowe z pewnym rozszerzeniem. Początek wyrażenia jest identyczny, ale końcowa część zaczynająca się od `if`, jest *wyrażeniem filtrującym*. *Wyrażenie filtrujące* może być dowolnym wyrażeniem, które może zostać zinterpretowane jako wyrażenie logiczne. Każdy element dla którego wyrażenie to będzie prawdziwe, zostanie dołączony do wyjściowej listy. Wszystkie inne elementy dla których *wyrażenie filtrujące* jest fałszywe, zostaną pominięte i nie trafią do wyjściowej listy.

Przykład 4.16 Filtrowanie listy

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]        #(1)
['mpilgrim', 'foo']
>>> [elem for elem in li if elem != "b"]          #(2)
['a', 'mpilgrim', 'foo', 'c', 'd', 'd']
>>> [elem for elem in li if li.count(elem) == 1]  #(3)
['a', 'mpilgrim', 'foo', 'c']
```

1. W tym przykładzie *wyrażenie odwzorowujące* nie jest skomplikowane (zwraca po prostu wartość każdego elementu), więc skoncentrujmy się na wyrażeniu filtrującym. Kiedy Python przechodzi przez każdy element listy, sprawdza czy *wyrażenie filtrujące* jest prawdziwe dla tego elementu. Jeśli tak będzie, to Python wykona *wyrażenie odwzorowujące* na tym elemencie i wstawi odwzorowany element do zwracanej listy. W tym przypadku odfiltrowujemy wszystkie łańcuchy znaków, które mają co najwyżej jeden znak, tak więc otrzymujemy listę wszystkich dłuższych napisów.

2. Tutaj odfiltrowujemy elementy, które przechowują wartość `''b''`. Zauważmy, że to wyrażenie listowe odfiltrowuje wszystkie wystąpienia `''b''`, ponieważ za każdym razem, gdy dany element będzie równy `''b''`, *wyrażenie filtrujące* będzie fałszywe, a zatem wartość ta nie zostanie wstawiona do zwracanej listy.

3. `count` jest metodą listy, która zwraca ilość wystąpień danej wartości w liście. Można się domyślać, że ten filtr usunie duplikujące się wartości, przez co zostanie zwrócona lista, która zawiera tylko jedną kopię każdej wartości z oryginalnej listy. Jednak tak się nie stanie. Wartości, które pojawiają się dwukrotnie w oryginalnej liście (w tym wypadku `''b''` i `''d''`) zostaną całkowicie odrzucone. Istnieje możliwość usunięcia duplikatów z listy, jednak filtrowanie listy nie daje nam takiej możliwości.

Wróćmy teraz do apihelper.py, do poniższej linii:

```
methodList = [method for method in dir(object) if callable(getattr(object, method))]
```

To wyrażenie listowe wygląda skomplikowanie, a nawet jest skomplikowane, jednak podstawowa struktura jest taka sama. Całe to wyrażenie zwraca listę, która zostaje przypisana do zmiennej `methodList`. Pierwsza część to część odwzorowująca listę. *Wyrażenie odwzorowujące* zwraca wartość danego elementu. `dir(object)` zwraca listę atrybutów i metod obiektu `object`, czyli jest to po prostu lista, którą odwzorowujemy. Tak więc nową częścią jest tylko *wyrażenie filtrujące* znajdujące się za instrukcją `if`.

To wyrażenie nie jest takie straszne, na jakie wygląda. Już poznaliśmy funkcje `callable`, `getattr` oraz `in`. Jak już wiemy z poprzedniego podrozdziału, funkcja `getattr(object, method)` zwraca obiekt funkcji (czyli referencję do tej funkcji), jeśli `object` jest modułem, a `method` jest nazwą funkcji w tym module.

Podsumowując, wyrażenie bierze pewien obiekt (nazwany `object`). Następnie pobiera listę nazw atrybutów tego obiektu, a także metod i funkcji oraz kilka innych rzeczy. Następnie odrzuca te rzeczy, które nas nie interesują, czyli pobieramy nazwy każdego atrybutu/metody/funkcji, a następnie za pomocą `getattr` pobieramy referencje do atrybutów o tych nazwach. Potem za pomocą funkcji `callable` sprawdzamy, czy ten obiekt jest wywoływalny, a dzięki temu dowiadujemy się, czy mamy do czynienia z metodą lub jakąś funkcją. Mogą to być na przykład funkcje wbudowane (np. metoda listy o nazwie `pop`), czy też funkcje zdefiniowane przez użytkownika (np. funkcja `buildConnectionString` z modułu `odbchelper`). Nie musimy natomiast martwić się o inne atrybuty jak np. wbudowany do każdego modułu atrybut `__name__` (nie jest on wywoływalny, czyli `callable` zwróci wartość `False`).

Materiały dodatkowe

Python Tutorial omawia inny sposób filtrowania listy, za pomocą wbudowanej funkcji `filter`.

5.7 Operatory and i or

Operatory `and` i `or`

Operatory `and` i `or` odpowiadają boolowskim operacjom logicznym, jednak nie zwracają one wartości logicznych. Zamiast tego zwracają którąś z podanych wartości.

Przykład 4.17 Poznajemy `and`

```
>>> 'a' and 'b'          #(1)
'b'
>>> '' and 'b'           \#(2)
''
>>> 'a' and 'b' and 'c'  #(3)
'c'
```

1. Podczas używania `and` wartości są oceniane od lewej do prawej. `0`, `''`, `[]`, `()`, `{}` i `None` są fałszem w kontekście logicznym, natomiast wszystko inne jest prawdą. Cóż, prawie wszystko. Domyślnie instancje klasy w kontekście logicznym są prawdą, ale możesz zdefiniować specjalne metody w swojej klasie, które sprawią, że będzie ona fałszem w kontekście logicznym. Wszystkiego o klasach i specjalnych metodach nauczymy się w rozdziale "Obiekty i klasy". Jeśli wszystkie wartości są prawdą w kontekście logicznym, `and` zwraca ostatnią wartość. W tym przypadku `and` najpierw bierze `'a'`, co jest prawdą, a potem `'b'`, co też jest prawdą, więc zwraca ostatnią wartość, czyli `'b'`.
2. Jeśli jakaś wartość jest fałszywa w kontekście logicznym, `and` zwraca pierwszą fałszywą wartość. W tym wypadku `''` jest pierwszą fałszywą wartością.
3. Wszystkie wartości są prawdą, tak więc `and` zwraca ostatnią wartość, `'c'`.

Przykład 4.18 Poznajemy `or`

```
>>> 'a' or 'b'          #(1)
'a'
>>> '' or 'b'           #(2)
'b'
>>> '' or [] or {}      #(3)
{}
>>> def sidefx():
...     print "in sidefx()"
...     return 1
>>> 'a' or sidefx()     #(4)
'a'
```

1. Używając `or` wartości są oceniane od lewej do prawej, podobnie jak w `and`. Jeśli jakaś wartość jest prawdą, `or` zwraca tą wartość natychmiast. W tym wypadku, `'a'` jest pierwszą wartością prawdziwą.
2. `or` wyznacza `''` jako fałsz, ale potem `'b'`, jako prawdę i zwraca `'b'`.
3. Jeśli wszystkie wartości są fałszem, `or` zwraca ostatnią wartość. `or` ocenia `''` jako fałsz, potem `[]` jako fałsz, potem {} jako fałsz i zwraca {}.

4. Zauważmy, że `or` ocenia kolejne wartości od lewej do prawej, dopóki nie znajdzie takiej, która jest prawdą w kontekście logicznym, a pozostałą resztę ignoruje. Tutaj, funkcja `sidefx` nigdy nie jest wywołana, ponieważ już `'a'` jest prawdą i `'a'` zostanie zwrócone natychmiastowo.

Jeśli jesteś osobą programującą w języku C, na pewno znajome jest ci wyrażenie `bool ? a : b`, z którego otrzymamy `a`, jeśli `bool` jest prawdą, lub `b` w przeciwnym wypadku. Dzięki sposobowi działania operatorów `and` i `or` w Pythonie, możemy osiągnąć podobny efekt.

Sztuczka `and-or`

Przykład 4.19 Poznajemy sztuczkę `and-or`

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b      #(1)
'first'
>>> 0 and a or b      #(2)
'second'
```

1. Ta składnia wygląda podobnie do wyrażenia `bool ? a : b` w C. Całe wyrażenie jest oceniane od lewej do prawej, tak więc najpierw określony zostanie `and`. Czyli `1 and 'first'` daje `'first'`, potem `'first' or 'second'` daje `'first'`.

2. `0 and 'first'` daje `0`, a potem `0 or 'second'` daje `'second'`.

Jakkolwiek te wyrażenia Pythona są po prostu logiką boolowską, a nie specjalną konstrukcją języka. Istnieje jedna bardzo ważna różnica pomiędzy Pythonową sztuczką `and-or`, a składnią `bool ? a : b` w C. Jeśli wartość `a` jest fałszem, wyrażenie to nie będzie działało tak, jakbyśmy chcieli. Można się na tym nieźle przejechać, co zobaczymy w poniższym przykładzie.

Przykład 4.20 Kiedy zawodzi sztuczka `and-or`

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b          #(1)
'second'
```

1. Ponieważ `a` jest pustym napisem, który Python uważa za fałsz w kontekście logicznym, więc `1 and ''` daje `''`, a następnie `'' or 'second'` daje `'second'`. Ups! To nie to, czego oczekiwaliśmy.

Sztuczka `and-or`, czyli wyrażenie `bool and a or b`, nie będzie działało w identyczny sposób, jak wyrażenie w C `bool ? a : b`, jeśli `a` będzie fałszem w kontekście logicznym.

Prawdziwą sztuczką kryjącą się za sztuczką `and-or`, jest upewnienie się, czy wartość `a` nigdy nie jest fałszywa. Jednym ze sposobów na wykonanie tego to przekształcenie `a` w `[a]` i `b` w `[b]`, a potem pobranie pierwszego elementu ze zwróconej listy, którym będzie `a` lub `b`.

Przykład 4.21 Bezpieczne użycie sztuczki `and-or`

```
>>> a = ""
>>> b = "second"
>>> (1 and [a] or [b])[0] #(1)
''
```

1. Jako że `[a]` jest nie pustą listą, więc nigdy nie będzie fałszem (tylko pusta lista jest fałszem). Nawet gdy `a` jest równe `0` lub `''` lub inną wartością dającą fałsz, lista `[a]` będzie prawdą, ponieważ ma jeden element.

Jak dotąd, ta sztuczka może wydawać się bardziej uciążliwa niż pomocna. Możesz przecież osiągnąć to samo zachowanie instrukcją `if`, więc po co to całe zamieszanie. Cóż, w wielu przypadkach, wybierasz pomiędzy dwoma stałymi wartościami, więc możesz użyć prostszego zapisu i się nie martwić, ponieważ wiesz, że wartość zawsze będzie prawdą. I nawet kiedy potrzebujesz użyć bardziej skomplikowanej, bezpiecznej formy, istnieją powody, aby i tak czasami korzystać z tej sztuczki. Na przykład, są pewne przypadki w Pythonie gdzie instrukcje `if` nie są dozwolone np. w wyrażeniach `lambda`.

W Pythonie 2.5 wprowadzono odpowiednik wyrażenia `bool ? a : b` z języka C; jest ono postaci: `a if bool else b`. Jeśli wartość `bool` będzie prawdziwa, to z wyrażenia zostanie zwrócona wartość `a`, a jeśli nie, to otrzymamy wartość `b`.

Materiały dodatkowe

- Python Cookbook omawia alternatywy do triku `and-or`.

5.8 Wyrażenia lambda

Wyrażenia `lambda`

Python za pomocą pewnych wyrażeń pozwala nam zdefiniować jednolinijkowe mini-funkcje. Te tzw. funkcje `lambda` są zapożyczone z Lispa i mogą być użyte wszędzie tam, gdzie potrzebna jest funkcja.

Przykład 4.22 Tworzymy funkcje `lambda`

```
>>> def f(x):
...     return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2      #(1)
>>> g(3)
6
>>> (lambda x: x*2)(3)     #(2)
6
```

1. W ten sposób tworzymy funkcję `lambda`, która daje ten sam efekt jak normalna funkcja nad nią. Zwróćmy uwagę na skróconą składnię: nie ma nawiasów wokół listy argumentów, brakuje też słowa kluczowego `return` (cała funkcja może być tylko jednym wyrażeniem). Funkcja nie posiada również nazwy, ale może być wywołana za pomocą zmiennej, do której zostanie przypisana.

2. Możemy użyć funkcji `lambda` bez przypisywania jej do zmiennej. Może taki sposób korzystania z wyrażeń `lambda` nie jest zbyt przydatny, ale w ten sposób możemy zobaczyć, że za pomocą tego wyrażenia tworzymy funkcję jednolinijkową.

Podsumowując, funkcja `lambda` jest funkcją, która pobiera dowolną liczbę argumentów (włączając argumenty opcjonalne) i zwraca wartość, którą otrzymujemy po wykonaniu pojedynczego wyrażenia. Funkcje `lambda` nie mogą zawierać poleceń i nie mogą zawierać więcej niż jednego wyrażenia. Nie próbujmy upchać zbyt dużo w funkcję `lambda`; zamiast tego jeśli potrzebujemy coś bardziej złożonego, zdefiniujmy normalną funkcję.

Korzystanie z wyrażeń `lambda` zależy od stylu programowania. Używanie ich nigdy nie jest wymagane; wszędzie gdzie moglibyśmy z nich skorzystać, równie dobrze moglibyśmy zdefiniować oddzielną, normalną funkcję i użyć jej zamiast funkcji `lambda`. Funkcje `lambda` możemy używać np. w miejscach, w których potrzebujemy specyficznego, nieużywalnego powtórnie kodu. Dzięki temu nie zaśmiecamy kodu wieloma małymi jednoliniowymi funkcjami.

Funkcje `lambda` w prawdziwym świecie

Poniżej przedstawiamy funkcje `lambda` wykorzystaną w apihelper.py:

```
processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

Zauważmy, że użyta jest tu prosta forma sztuczki `and-or`, która jest bezpieczna, ponieważ funkcja `lambda` jest zawsze prawdą w kontekście logicznym. (To nie znaczy, że funkcja lambda nie może zwracać wartości będącej fałszem. Funkcja jest zawsze prawdą w kontekście logicznym, ale jej zwracana wartość może być czymkolwiek.)

Zauważmy również, że używamy funkcji `split` bez argumentów. Widzieliśmy już ją użytą z jednym lub dwoma argumentami. Jeśli nie podamy argumentów, wówczas domyślnym separatorem tej funkcji są białe znaki (czyli spacja, znak nowej linii, znak tabulacji itp.).

Przykład 4.23 `split` **bez argumentów**

```
>>> s = "this   is\na\ttest"  #(1)
>>> print s
this   is
a test
>>> print s.split()           #(2)
['this', 'is', 'a', 'test']
>>> print " ".join(s.split()) #(3)
'this is a test'
```

1. Tutaj mamy wieloliniowy napis, zdefiniowany przy pomocy znaków sterujących zamiast użycia trzykrotnych cudzysłowów. `\n` jest znakiem nowej linii, a `\t` znakiem tabulacji.

2. `split` bez żadnych argumentów dzieli na białych znakach, a trzy spacje, znak nowej linii i znak tabulacji są białymi znakami.

3. Możemy unormować białe znaki poprzez podzielenie napisu za pomocą metody `split`, a potem powtórne złączenie metodą `join`, używając pojedynczej spacji jako separatora. To właśnie robi funkcja `info`, aby zwinąć wieloliniowe notki dokumentacyjne w jedną linię.

Co więc właściwie funkcja `info` robi z tymi funkcjami `lambda`, dzieleniami i sztuczkami `and-or`?

```
processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

`processFunc` jest teraz referencją do funkcji, ale zależną od zmiennej `collapse`. Jeśli `collapse` jest prawdą, `processFunc(string)` będzie zwijać białe znaki, a w przeciwnym wypadku `processFunc(string)` zwróci swój argument niezmieniony.

Aby to zrobić w mniej zaawansowanym języku (np. w Visual Basicu), prawdopodobnie stworzylibyśmy funkcję, która pobiera napis oraz argument `collapse` i używa instrukcji `if`, aby określić czy zawijać białe znaki czy też nie, a potem zwracałaby odpowiednią wartość. Takie podejście nie byłoby zbyt efektywne, ponieważ funkcja musiałaby obsłużyć każdy możliwy przypadek. Za każdym jej wywołaniem, musiałaby zdecydować czy zawijać białe znaki zanim dałaby nam to, co chcemy. W Pythonie logikę wyboru możemy wyprowadzić poza funkcję i zdefiniować funkcję `lambda`, która jest dostosowana do tego, aby dać nam dokładnie to (i tylko to), co chcemy. Takie podejście jest bardziej efektywne, bardziej eleganckie i mniej podatne na błędy typu "Ups! Te argumenty miały być w odwrotnej kolejności...".

Materiały dodatkowe

- Python Knowledge Base omawia, jak wykorzystywać `lambda`, aby wywoływać funkcje w sposób niebezpośredni.
- Python Tutorial pokazuje, jak dostać się do zewnętrznych zmiennych z wnętrza funkcji `lambda`.
- The Whole Python FAQ zawiera przykłady, pokazujące, jak można zagmatwać kod funkcji `lambda`.

5.9 Potęga introspekcji - wszystko razem

Wszystko razem

Ostatnia linia kodu, jedyna której jeszcze nie rozpracowaliśmy, to ta, która odwala całą robotę. Teraz umieszczamy wszystkie puzzle w jednym miejscu i nadchodzi czas, aby je ułożyć.

To jest najważniejsza część apihelper.py:

```
print "\n".join(["%s %s" %
                (method.ljust(spacing),
                 processFunc(unicode(getattr(object, method).__doc__)))
                for method in methodList])
```

Zauważmy, że jest to tylko jedne wyrażenie podzielone na wiele linii, ale które nie używa znaku kontynuacji (znaku odwrotnego ukośnika, \). Pamiętasz jak powiedzieliśmy, że pewne instrukcje mogą być rozdzielone na kilka linii bez używania odwrotnego ukośnika? Wyrażenia listowe są jednym z tego typu wyrażeń, ponieważ całe wyrażenie jest zawarte w nawiasach kwadratowych.

Zacznijmy od końca i posuwajmy się w tył. Wyrażenie

```
for method in methodList
```

okazuje się być wyrażeniem listowym. Jak wiemy, `methodList` jest listą wszystkich metod danego obiektu, które nas interesują, więc za pomocą tej pętli przechodzimy tę listę wykorzystując zmienną `method`.

Przykład 4.24 Dynamiczne pobieranie notki dokumentacyjnej

```
>>> import odbchelper
>>> object = odbchelper                    #(1)
>>> method = 'buildConnectionString'       #(2)
>>> getattr(object, method)                #(3)
<function buildConnectionString at 010D6D74>
>>> print getattr(object, method).__doc__  #(4)
Tworzy łańcuchów znaków na podstawie słownika parametrów.

    Zwraca łańcuch znaków.
```

1. W funkcji `info`, `object` jest obiektem do którego otrzymujemy pomoc, a ten obiekt zostaje przekazany jako argument.

2. Podczas iterowania listy `methodList`, `method` jest nazwą aktualnej metody.

3. Używając funkcji `getattr` otrzymujemy referencję do funkcji `method` z modułu `object`.

4. Teraz wypisanie notki dokumentacyjnej będzie bardzo proste.

Następnym elementem puzzli jest użycie `unicode` na notce dokumentacyjnej. Jak sobie przypominamy, `unicode` jest wbudowaną funkcją, która przekształca dane na unikod, ale notka dokumentacyjna jest zawsze łańcuchem znaków, więc po co ją jeszcze konwertować na unikod? Nie każda funkcja posiada notkę dokumentacyjną, a jeśli ona nie istnieje, atrybut `__doc__` ma wartość `None`.

Przykład 4.25 Po co używać `unicode` na notkach dokumentacyjnych?

```
>>> def foo(): print 2
>>> foo()
2
>>> foo.__doc__     #(1)
>>> foo.__doc__ == None #(2)
True
>>> unicode(foo.__doc__)    #(3)
u'None'
```

1. Możemy łatwo zdefiniować funkcję, która nie posiada notki dokumentacyjnej, tak więc jej atrybut `__doc__` ma wartość `None`. Dezorientujące jest to, że jeśli bezpośrednio odwołamy się do atrybutu `__doc__`, IDE w ogóle nic nie wypisze. Jednak jeśli się trochę zastanowimy nad tym, takie zachowanie IDE ma jednak pewien sens [1]

2. Możemy sprawdzić, że wartość atrybutu `__doc__` aktualnie wynosi `None` przez porównanie jej bezpośrednio z tą wartością.

3. W tym przypadku funkcja `unicode` przyjmuje pustą wartość, `None` i zwraca jej unikodową reprezentację, czyli `'None'`.

4. `li.append.__doc__` jest łańcuchem znaków. Zauważmy, że wszystkie angielskie notki dokumentacyjne Pythona korzystają ze znaków ASCII, dlatego możemy spokojnie je przekonwertować do unikodu za pomocą funkcji `unicode`.

W SQL musimy skorzystać z `IS NULL` zamiast z `= NULL`, aby porównać coś z pustą wartością. W Pythonie możemy użyć albo `== None` albo `is None`, lecz `is None` jest szybsze.

Teraz kiedy już mamy pewność, że otrzymamy unikod, możemy przekazać otrzymany unikodowy napis do `processFunc`, którą już zdefiniowaliśmy jako funkcję zwijającą lub niezwijającą białe znaki (w zależności od przekazanego argumentu). Czy już wiemy, dlaczego wykorzystaliśmy `unicode`? Do przekonwertowania wartości `None` na reprezentację w postaci unikodowego łańcucha znaków. `processFunc` przyjmuje argument będący unikodem i wywołuje jego metodę `split`. Nie zadziałałoby to, gdybyśmy przekazali samo `None`, ponieważ `None` nie posiada metody o nazwie `split` i rzucony zostałby wyjątek. Może się zastanawiasz, dlaczego nie konwertujemy do `str`? Ponieważ tworzone przez nas notki są napisami unikodowymi, w których nie wszystkie znaki należą do ASCII, a zatem `str` rzuciłby wyjątek.

Idąc wstecz, widzimy, że ponownie używamy formatowania łańcucha znaków, aby połączyć wartości zwrócone przez `processFunc` i przez metodę `ljust`. Jest to metoda łańcucha znaków (dodajmy, że napis unikodowy także jest łańcuchem znaków, tylko nieco o większych możliwościach), której jeszcze nie poznaliśmy.

[1] Pamiętamy, że każda funkcja zwraca pewną wartość? Jeśli funkcja nie wykorzystuje instrukcji `return`, zostaje zwrócone `None`, a częste wyświetlanie `None` po wykonaniu pewnych funkcji przez IDE Pythona mogłoby być trochę uciążliwe.

Przykład 4.26 Poznajemy `ljust`

```
>>> s = 'buildConnectionString'
>>> s.ljust(30) #(1)
'buildConnectionString         '
>>> s.ljust(20) #(2)
'buildConnectionString'
```

1. `ljust` wypełnia napis spacjami do zadanej długości. Z tej możliwości korzysta funkcja `info`, aby stworzyć dwie kolumny na wyjściu i aby wszystkie notki dokumentacyjne umieścić w drugiej kolumnie.
2. Jeśli podana długość jest mniejsza niż długość napisu, `ljust` zwróci po prostu napis niezmieniony. Metoda ta nigdy nie obcina łańcucha znaków.

Już prawie skończyliśmy. Mając nazwę metody `method` uzupełnioną spacjami poprzez `ljust` i (prawdopodobnie zwiniętą) notkę dokumentacyjną otrzymaną z wywołania `processFunc`, łączymy je i otrzymujemy pojedynczy napis, łańcuch znaków. Ponieważ odwzorowujemy listę `methodList`, dostajemy listę złożoną z takich łańcuchów znaków. Używając metody `join` z napisu ``\n'', łączymy tę listę w jeden łańcuch znaków, gdzie każdy elementem listy znajduje się w oddzielnej linii i ostatecznie wypisujemy rezultat.

Przykład 4.27 Wypisywanie listy

```
>>> li = ['a', 'b', 'c']
>>> print "\n".join(li)  #(1)
a
b
c
```

1. Ta sztuczka może być pomocna do znajdowania błędów, gdy pracujemy na listach, a w Pythonie zawsze pracujemy na listach.

I to już był ostatni element puzzli. Teraz powinieneś zrozumieć ten kod.

```
print "\n".join(["%s %s" %
               (method.ljust(spacing),
                processFunc(unicode(getattr(object, method).__doc__)))
               for method in methodList])
```

5.10 Potęga introspekcji - podsumowanie

Podsumowanie

Program apihelper.py i jego wyjście powinno teraz nabrać sensu.

```
def info(object, spacing=10, collapse=1):
    u"""Wypisuje metody i ich notki dokumentacyjne.

    Argumentem może być moduł, klasa, lista, słownik, czy też łańcuch znaków."""
    methodList = [e for e in dir(object) if callable(getattr(object, e))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                    (method.ljust(spacing),
                     processFunc(unicode(getattr(object, method).__doc__)))
                    for method in methodList])

if __name__ == "__main__":
    print info.__doc__
```

A z tutaj mamy przykład wyjścia, które otrzymujemy z programu apihelper.py:

```
>>> from apihelper import info
>>> li = []
>>> info(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(iterable) -- extend list by appending elements from the iterable
index      L.index(value, [start, [stop]]) -> integer -- return first index of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
cmp(x, y) -> -1, 0, 1 append
```

Zanim przejdziemy do następnego rozdziału, upewnijmy się, że nie mamy problemów z wykonywaniem poniższych czynności:

- Definiowanie i wywoływanie funkcji z opcjonalnymi i nazwanymi argumentami
- Importowanie modułów za pomocą `import module` i `from module import`
- Używanie `str` do przekształcenia jakiejkolwiek przypadkowej wartości na reprezentację w postaci łańcucha znaków, a także używanie `unicode` do przekształcania wartości na unikod.
- Używanie `getattr` do dynamicznego otrzymywania referencji do funkcji i innych atrybutów
- Tworzenie wyrażeń listowych z filtrowaniem
- Rozpoznawanie sztuczki `and-or` i używanie jej w sposób bezpieczny

- Definiowanie funkcji `lambda`
- Przypisywanie funkcji do zmiennych i wywoływanie funkcji przez zmienną. Trudno jest to mocniej zaakcentować, jednak umiejętność wykonywania tego jest niezbędne do lepszego rozumienia Pythona. Zobaczymy bardziej złożone aplikacje opierające się na tej koncepcji w dalszej części książki.

Rozdział 6

Obiekty i klasy

6.1 Obiekty i klasy

Rozdział ten zaznajomi nas ze zorientowanym obiektowo programowaniem przy użyciu języka Python.

Nurkujemy

Poniżej znajduje się kompletny program, który oczywiście działa. Czytanie notki dokumentacyjnej modułu, klasy, czy też funkcji jest pomocne w zrozumieniu co dany program właściwie robi i w jaki sposób działa. Jak zwykle, nie martwmy się, że nie możemy wszystkiego zrozumieć. W końcu zasada działania tego programu zostanie dokładnie opisana w dalszej części tego rozdziału.

Przykład 5.1 fileinfo.py

```
#-*- coding: utf-8 -*-

u"""Framework do pobierania metadanych specyficznych dla danego typu pliku.

Można utworzyć instancję odpowiedniej klasy podając jej nazwę pliku w konstruktorze.
Zwrócony obiekt zachowuje się jak słownik posiadający parę klucz-wartość
dla każdego fragmentu metadanych.

    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Lub użyć funkcji listDirectory, aby pobrać informacje o wszystkich plikach w katalogu.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework może być roszerzony poprzez dodanie klas dla poszczególnych typów plików, np.:
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Każda klasa jest całkowicie odpowiedzialna
za właściwe sparsowanie swojego pliku; zobacz przykład MP3FileInfo.
"""

import os
import sys

def stripnulls(data):
    u"usuwa białe znaki i nulle"
    return data.replace("\00", " ").strip()

class FileInfo(dict):
    u"przechowuje metadane pliku"
    def __init__(self, filename=None):
        dict.__init__(self)
        self["plik"] = filename

class MP3FileInfo(FileInfo):
```

```
    u"przechowuje znaczniki ID3v1.0 MP3"
    tagDataMap = {u"tytuł"    : (  3,  33, stripnulls),
                  "artysta"   : ( 33,  63, stripnulls),
                  "album"     : ( 63,  93, stripnulls),
                  "rok"       : ( 93,  97, stripnulls),
                  "komentarz" : ( 97, 126, stripnulls),
                  "gatunek"   : (127, 128, ord)}

    def __parse(self, filename):
        u"parsuje znaczniki ID3v1.0 z pliku MP3"
        self.clear()
        try:
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == 'TAG':
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "plik" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    u"zwraca listę obiektów zawierających metadane dla plików o podanych rozszerzeniach"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        u"zwraca klasę metadanych pliku na podstawie podanego rozszerzenia"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]):   #(1)
        print "\n".join("%s=%s" % (k, v) for k, v in info.items())
        print
```

1. Wynik wykonania tego programu zależy od tego jakie pliki mamy na twardym dysku. Aby otrzymać sensowne wyjście, potrzebujemy zmienić ścieżkę, aby wskazywał na katalog, w którym przechowujemy pliki `MP3`.

W wyniku wykonania tego programu możemy otrzymać podobne wyjście do poniższego. Jest niemal niemożliwe, abyś otrzymał identyczne wyjście.

```
album=
rok=1999
komentarz=http://mp3.com/ghostmachine
tytuł=A Time Long Forgotten (Concept
artysta=Ghost in the Machine
gatunek=31
plik=/music/_singles/a_time_long_forgotten_con.mp3

album=Rave Mix
rok=2000
komentarz=http://mp3.com/DJMARYJANE
tytuł=HELLRAISER****Trance from Hell
artysta=***DJ MARY-JANE***
gatunek=31
plik=/music/_singles/hellraiser.mp3

album=Rave Mix
rok=2000
komentarz=http://mp3.com/DJMARYJANE
tytuł=KAIRO****THE BEST GOA
artysta=***DJ MARY-JANE***
gatunek=31
plik=/music/_singles/kairo.mp3

album=Journeys
rok=2000
komentarz=http://mp3.com/MastersofBalan
tytuł=Long Way Home
artysta=Masters of Balance
gatunek=31
plik=/music/_singles/long_way_home1.mp3

album=
rok=2000
komentarz=http://mp3.com/cynicproject
tytuł=Sidewinder
artysta=The Cynic Project
gatunek=18
plik=/music/_singles/sidewinder.mp3

album=Digitosis@128k
rok=2000
komentarz=http://mp3.com/artists/95/vxp
tytuł=Spinning
artysta=VXpanded
gatunek=255
plik=/music/_singles/spinning.mp3
```

6.2 Definiowanie klas

Definiowanie klas

Python jest całkowicie zorientowany obiektowo: możemy definiować własne klasy, dziedziczyć z własnych lub wbudowanych klas, a także tworzyć instancje zdefiniowanych przez siebie klas.

Tworzenie klas w Pythonie jest proste. Podobnie jak z funkcjami, nie używamy oddzielnego interfejsu definicji. Po prostu definiujemy klasę i zaczynamy ją implementować. Klasa w Pythonie rozpoczyna się słowem kluczowym `class`, po którym następuje nazwa klasy, a następnie w nawiasach okrągłych umieszczamy, z jakich klas dziedziczymy.

Przykład 5.3 Prosta klasa

```
class Nicosc(object):   #(1)
    pass                #(2) (3)
```

1. Nazwa tej klasy to `Nicosc`, a klasa ta dziedziczy z wbudowanej klasy `object`. Nazwy klas są zazwyczaj pisane przy użyciu wielkich liter np. `KazdeSlowoOdzieloneWielkaLitera`, ale to kwestia konwencji nazewnictwa; nie jest to wymagane.
2. Klasa ta nie definiuje żadnych metod i atrybutów, ale żeby kod był zgodny ze składnią Pythona, musimy coś umieścić w definicji, tak więc użyliśmy `pass`. Jest to zastrzeżone przez Pythona słowo, które mówi interpreterowi "przejdź dalej, nic tu nie ma". Instrukcja ta nie wykonuje żadnej operacji i powinniśmy stosować ją, gdy chcemy zostawić funkcję lub klasę pustą.
3. Prawdopodobnie zauważyliśmy już, że elementy w klasie są wyszczególnione za pomocą wcięć, podobnie jak kod funkcji, instrukcji warunkowych, pętli itp. Pierwsza nie wcięta instrukcja nie będzie należała już do klasy.

Od Pythona 2.2 wprowadzono *nowy styl klas* (ang. *new-style classes*), którego będziemy się uczyli. Aby pisać klasy w *nowym stylu*, musimy dziedziczyć je z którejś wbudowanej klasy, najczęściej będzie to `object`. W przeciwnym wypadku pisane przez nas klasy będą w *starym stylu* i niektóre możliwości nie będą dostępne. Dzięki takiemu zachowaniu Pythona, jest on kompatybilny wstecz.

Instrukcja `pass` w Pythonie jest analogiczna do pustego zbioru nawiasów klamrowych ({}) w Javie lub w języku C++.

W prawdziwym świecie większość klas definiuje własne metody i atrybuty. Jednak, jak można zobaczyć wyżej, definicja klasy, oprócz swojej nazwy z dodatkiem `object` w nawiasach, nie musi nic zawierać. Programiści C++ mogą zauważyć, że w Pythonie klasy nie mają wyraźnie sprecyzowanych konstruktorów i destruktorów. Pythonowe klasy mają coś, co przypomina konstruktor – metodę `__init__`.

Przykład 5.4 Definiowanie klasy `FileInfo`

```
class FileInfo(dict): #(1)
```

1. Jak już wiemy, klasy, z których chcemy dziedziczyć wyszczególniamy w nawiasach okrągłych, które z kolei umieszczamy bezpośrednio po nazwie naszej klasy. Tak więc klasa `FileInfo` dziedziczy z wbudowanej klasy `dict`, a ta klasa, to po prostu klasa słownika.

W Pythonie klasę, z której dziedziczymy, wyszczególniamy w nawiasach okrągłych, umieszczonych bezpośrednio po nazwie naszej klasy. Python nie posiada specjalnego słowa kluczowego jak np. `extends` w Javie.

Python obsługuje dziedziczenie wielokrotne. Wystarczy w nawiasach okrągłych, umiejscowionych zaraz po nazwie klasy, wstawić nazwy klas, z których chcemy dziedziczyć i oddzielić je przecinkami np. `class klasa(klasa1,klasa2)`.

Inicjalizowanie i implementowanie klasy

Ten przykład przedstawia inicjalizację klasy `FileInfo` za pomocą metody `__init__`.

Przykład 5.5 Inicjalizator klasy FileInfo

```
class FileInfo(dict):
    u"przechowuje metadane pliku"        #(1)
    def __init__(self, filename=None):   #(2) (3) (4)
```

1. Klasy mogą (a nawet powinny) posiadać także notkę dokumentacyjną, podobnie jak moduły i funkcje.

2. Metoda `__init__` jest wywoływana bezpośrednio po utworzeniu instancji klasy. Może kusić, aby nazwać ją konstruktorem klasy, co jednak nie jest prawdą. Metoda `__init__` wygląda podobnie do konstruktora (z reguły `__init__` jest pierwszą metodą definiowaną dla klasy), działa podobnie (jest pierwszym fragmentem kodu wykonywanego w nowo utworzonej instancji klasy), a nawet podobnie brzmi (słowo "init" sugeruje, że jest to konstruktor). Niestety nie jest to prawda, ponieważ obiekt jest już utworzony przed wywołaniem metody `__init__`, a my już otrzymujemy poprawną referencję do świeżo utworzonego obiektu. Jednak `__init__` w Pythonie, jest tym co najbardziej przypomina konstruktor, a ponadto pełni prawie taką samą rolę.

3. Pierwszym argumentem każdej metody znajdującej się w klasie, włączając w to `__init__`, jest zawsze referencja do bieżącej instancji naszej klasy. Według powszechnej konwencji, argument ten jest zawsze nazywany `self`. W metodzie `__init__` `self` odwołuje się do właśnie utworzonego obiektu; w innych metodach klasy, `self` odwołuje się do instancji, z której wywołaliśmy daną metodę. Mimo, że musimy wyraźnie określić argument `self` podczas definiowania metody, ale nie określamy go w czasie wywoływania metody; Python dodaje go automatycznie.

4. Metoda `__init__` może posiadać dowolną liczbę argumentów i podobnie jak w funkcjach, argumenty mogą być zdefiniowane z domyślnymi wartościami (w ten sposób stają się argumentami opcjonalnymi). W tym przypadku argument `filename` ma domyślną wartość określoną jako `None`, który jest Pythonową pustą wartością.

Według konwencji, pierwszy argument metody należącej do pewnej klasy (referencja do bieżącej instancji klasy) jest nazywany `self`. Argument ten pełni tą samą rolę, co zastrzeżone słowo `this` w C++ czy Javie, ale `self` nie jest zastrzeżonym słowem w Pythonie, jest jedynie konwencją nazewnictwa. Niemniej lepiej pozostać przy tej konwencji, ponieważ jest to wręcz bardzo silna umowa.

Przykład 5.6 Kodowanie klasy `FileInfo` class FileInfo(dict):

```
u"przechowuje metadane pliku"
def __init__(self, filename=None):
    dict.__init__(self)             #(1)
    self["plik"] = filename         #(2)
                                    #(3)
```

1. Niektóre języki pseudo-zorientowane obiektowo jak *Powerbuilder* posiadają koncepcję “rozszerzania” konstruktorów i innych zdarzeń, w których metoda należąca do nadklasy jest wykonywana automatycznie przed metodą podklasy. Python takiego czegoś nie wykonuje; zawsze należy wyraźnie wywołać odpowiednią metodę należącą do przodka klasy.

2. Klasa ta działa podobnie jak słownik (w końcu z niego dziedziczymy), co mogliśmy zauważyć po spojrzeniu na tę linię. Przypisaliśmy argument `filename` jako wartość klucza `‘‘plik’’` w naszym obiekcie.

3. Zauważmy, że metoda `__init__` nigdy nie zwraca żadnej wartości.

Kiedy używać `self` i `__init__`

Podczas definiowania metody pewnej klasy, musimy wyraźnie wstawić `self` jako pierwszy argument każdej metody, włączając w to `__init__`. Kiedy wywołujemy metodę z klasy nadrzędnej, musimy dołączyć argument `self`, ale jeśli wywołujemy metodę z zewnątrz, nie określamy argumentu `self`, po prostu go pomijamy. Python automatycznie wstawi odpowiednią referencję za nas. Na początku może się to wydawać trochę namieszane, jednak wynika to z pewnych różnic, o których jeszcze nie wiemy [1].

Metoda `__init__` jest opcjonalna, ale jeśli ją definiujemy, musimy pamiętać o wywołaniu metody `__init__`, która należy do przodka klasy. W szczególności, jeśli potomek chce poszerzyć pewne zachowanie przodka, dana metoda podklasy musi w odpowiednim miejscu bezpośrednio wywoływać metodę należącą do klasy nadrzędnej, oczywiście z odpowiednimi argumentami.

[1] Wynika to z różnic pomiędzy metodami instancji klasy (ang. *bound method*), a metodami samej klasy (ang. *unbound method*)

Materiały dodatkowe

- New-style classes opisuje klasy w nowym stylu
- Python Tutorial opisuje klasy, przestrzenie nazw i dziedziczenie

6.3 Tworzenie instancji klasy

Tworzenie instancji klasy

Tworzenie instancji klas jest dosyć proste. W tym celu wywołujemy klasę tak jakby była funkcją, dodając odpowiednie argumenty, które są określone w metodzie `__init__`. Zwracaną wartością będzie zawsze nowo utworzony obiekt.

Przykład 5.7 Tworzenie instacji klasy `FileInfo`

```
>>> import fileinfo
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3") #(1)
>>> f.__class__                                        #(2)
<class 'fileinfo.FileInfo'>
>>> f.__doc__                                          #(3)
u'przechowuje metadane pliku'
>>> f                                                  #(4)
{'plik': '/music/_singles/kairo.mp3'}
```

1. Utworzyliśmy instancję klasy `FileInfo` (zdefiniowaną w module `fileinfo`) i przypisaliśmy właśnie utworzony obiekt do zmiennej `f`. Skorzystaliśmy z parametru `''/music/_singles/kairo.mp3''`, a który będzie odpowiadał argumentowi `filename` w metodzie `__init__` klasy `FileInfo`.

2. Każda instancja pewnej klasy ma wbudowany atrybut `__class__`, który jest klasą danego obiektu. Programiści Javy mogą być zaznajomieni z klasą `Class`, która posiada metody takie jak `getName`, czy też `getSuperclass`, aby pobrać metadane o pewnym obiekcie. W Pythonie ten rodzaj metadadanych, jest dostępny bezpośrednio z obiektu wykorzystując atrybuty takie jak `__class__`, `__name__`, czy `__bases__`.

3. Możemy pobrać notkę dokumentacyjną w podobny sposób, jak to czyniliśmy w przypadku funkcji czy modułu. Wszystkie instancje klasy współdzielą tę samą notkę dokumentacyjną.

4. Pamiętamy, że metoda `__init__` przypisuje argument `filename` do `self[''plik'']`? To dobrze, w tym miejscu mamy rezultat tej operacji. Argumenty podawane podczas tworzenia instancji pewnej klasy są wysłane do metody `__init__` (wyłączając pierwszy argument, `self`. Python zrobił to za nas).

W Pythonie, aby utworzyć instancję pewnej klasy, wywołujemy klasę tak, jakby to była zwykła funkcja zwracająca pewną wartość. Python nie posiada jakiegoś kluczowego słowa jakim jest np. operator `new` w Javie czy C++.

Odśmiecanie pamięci

Jeśli tworzenie nowej instancji jest proste, to jej usuwanie jest jeszcze prostsze. W ogólności nie musimy wyraźnie zwalniać instancji klasy, ponieważ Python robi to automatycznie, gdy wychodzi one poza swój zasięg. W Pythonie rzadko występują wycieki pamięci.

Przykład 5.8 Próba zaimplementowania wycieku pamięci

```
>>> def leakmem():
...     f = fileinfo.FileInfo('/music/_singles/kairo.mp3')  #(1)
...
>>> for i in range(100):
...     leakmem()                                           #(2)
```

1. Za każdym razem, gdy funkcja `leakmem` jest wywoływana, zostaje utworzona instancja klasy `FileInfo`, a ta zostaje przypisana do zmiennej `f`, która jest lokalną zmienną wewnątrz funkcji. Funkcja ta kończy się bez jakiegokolwiek wyraźnego zwolnienia pamięci zajmowanej przez zmienną `f`, a więc spodziewalibyśmy się wycieku pamięci, lecz tak nie będzie. Kiedy funkcja się kończy, lokalna zmienna `f` wychodzi poza swój zasięg. W tym miejscu nie ma więcej żadnych referencji do nowej instancji `FileInfo`, ponieważ nigdzie nie przypisywaliśmy jej do czegoś innego niż `f`, tak więc Python zniszczy instancję za nas.

2. Niezależnie od tego, jak wiele razy wywołamy funkcję `leakmem`, nigdy nie nastąpi wyciek pamięci, ponieważ za każdym razem kiedy to zrobimy, Python będzie niszczył nowo utworzony obiekt przed wyjściem z funkcji `leakmem`.

Technicznym terminem tego sposobu odśmiecania pamięci jest "zliczanie odwołań" (zobacz w Wikipedii). Python przechowuje listę referencji do każdej utworzonej instancji. W powyższym przykładzie, mamy tylko jedną referencję do instancji `FileInfo` – zmienną `f`. Kiedy funkcja się kończy, zmienna `f` wychodzi poza zasięg, więc licznik odwołań zmniejsza się do `0` i Python zniszczy tę instancję automatycznie.

W poprzednich wersjach Pythona występowały sytuacje, gdy zliczanie odwołań zawodziło i Python nie mógł wyczyścić po nas pamięci. Jeśli tworzyliśmy dwie instancje, które odwoływały się do siebie nawzajem (np. instancja listy dwukierunkowej [2], w których każdy węzeł wskazuje na poprzedni i następny znajdujący się w liście), żadna instancja nie była niszczona automatycznie, ponieważ Python uważał (poprawnie), że ciągle mamy referencję do każdej instancji. Od Pythona 2.0 mamy dodatkowy sposób odśmiecania pamięci nazywany po ang. *mark-and-sweep* (oznacz i zamiataj, zobacz w Wikipedii), dzięki któremu Python w sprytny sposób wykrywa różne wirtualne blokady i poprawnie czyści cykliczne odwołania.

Podsumowując, w języku tym można po prostu zapomnieć o zarządzaniu pamięcią i pozostawić tę sprawę Pythonowi.

Materiały dodatkowe

- Python Library Reference omawia wbudowane atrybuty podobne do `__class__`
- Python Library Reference dokumentuje moduł `gc`, który daje niskopoziomową kontrolę nad odśmiecaniem pamięci

[2] ang. *double linked list*

6.4 Klasa opakowująca UserDict

Klasa opakowująca `UserDict`

Wrócimy na chwilę do przeszłości. Za czasów, kiedy nie można było dziedziczyć wbudowanych typów danych np. słownika, powstawały tzw. klasy opakowujące, które pełniły te same funkcję, co typy wbudowane, ale można je było dziedziczyć. Klasą opakowującą dla słownika była klasa `UserDict`, która nadal jest dostępna wraz z nowymi wersjami Pythona. Przyglądnięcie się implementacji tej klasy może być dla nas cenną lekcją. Zatem zajrzyjmy do kodu źródłowego klasy `UserDict`, który znajdują się w module `UserDict`. Moduł ten z kolei jest przechowywany w katalogu `lib` instalacji Pythona, a pełna nazwa pliku to `UserDict.py` (nazwa modułu z rozszerzeniem `.py`).

W IDE ActivePython na Windowsie możemy szybko otworzyć dowolny moduł, który znajduje się w ścieżce do bibliotek, gdy wybierzemy `File->Locate...` `(Ctrl-L)`.

Przykład 5.9 Definicja klasy `UserDict`

```
class UserDict:              #(1)
    def __init__(self, dict=None):  #(2)
        self.data = {}              #(3)
        if dict is not None: self.update(dict)  #(4) (5)
```

1. Klasa `UserDict` nie dziedziczy nic z innych klas. Jednak nie patrzmy się na to, pamiętajmy, żeby zawsze dziedziczyć z `object` (lub innego wbudowanego typu), bo wtedy mamy dostęp do dodatkowych możliwości, które dają nam klasy w nowym stylu.

2. Jak pamiętamy, metoda `__init__` jest wywoływana bezpośrednio po utworzeniu instancji klasy. Przy tworzeniu instancji klasy `UserDict` możemy zdefiniować początkowe wartości, poprzez przekazanie słownika w argumencie `dict`.

3. W Pythonie możemy tworzyć atrybuty danych (zwane polami w Javie i PowerBuilderze). Atrybuty to kawałki danych przechowywane w konkretnej instancji klasy (moglibyśmy je nazwać *atrybutami instancji*). W tym przypadku każda instancja klasy `UserDict` będzie posiadać atrybut `data`. Aby odwołać się do tego pola z kodu spoza klasy, dodajemy z przodu nazwę instancji np. `instancja.data`; robimy to w identyczny sposób, jak odwołujemy się do funkcji poprzez nazwę modułu, w którym ta funkcja się znajduje. Aby odwołać się do atrybutu danych z wnętrza klasy, używamy `self`. Zazwyczaj wszystkie atrybuty są inicjalizowane sensownymi wartościami już w metodzie `__init__`. Jednak nie jest to wymagane, gdyż atrybuty, podobnie jak zmienne lokalne, są tworzone, gdy po raz pierwszy przypisze się do nich jakąś wartość.

4. Metoda `update` powiela zachowanie metody słownika: kopiuje wszystkie klucze i wartości z jednego słownika do drugiego. Metoda ta *nie* czyści słownika docelowego (tego, z którego wywołaliśmy metodę), ale jeśli były tam już jakieś klucze, to zostaną one nadpisane tymi, które są w słowniku źródłowym; pozostałe klucze nie zmienią się. Myślmy o `update` jak o funkcji łączenia, nie kopiowania.

5. Z tej składni nie korzystaliśmy jeszcze w tej książce. Jest to instrukcja `if`, ale zamiast wciętego bloku, który rozpoczynałby się w następnej linii, korzystamy tu z instrukcji, która znajduje się w jednej linii, zaraz za dwukropkiem. Jest to całkowicie poprawna, skrótowa składnia, której możemy używać, jeśli mamy tylko jedną instrukcję w bloku (tak jak pojedyncza instrukcja bez klamer w C++). Możemy albo skorzystać z tej skrótowej składni, albo tworzyć wcięte bloki, jednak nie możemy ich ze sobą łączyć w odniesieniu do tego samego bloku kodu.

Java i Powerbuilder mogą przeciążać funkcje mające różne listy argumentów, na przykład klasa może mieć kilka metod z taką samą nazwą, ale z różną liczbą argumentów lub z argumentami różnych typów. Inne języki (na przykład PL/SQL) obsługują nawet przeciążanie funkcji, które różnią się jedynie nazwą argumentu np. jedna klasa może mieć kilka metod z tą samą nazwą, tą samą liczbą argumentów o tych samych typów, ale inaczej nazwanych. Python nie ma żadnej z tych możliwości, nie ma tu w ogóle przeciążania funkcji. Metody są jednoznacznie definiowane przez ich nazwy i w danej klasie może być tylko jedna metoda o danej nazwie. Jeśli więc mamy w jakiejś klasie potomnej metodę `__init__`, to zawsze zasłoni ona metodę `__init__` klasy rodzicielskiej, nawet jeśli klasa pochodna definiuje ją z innymi argumentami. Ta uwaga stosuje się do wszystkich metod.

Guido, pierwszy twórca Pythona, tak wyjaśnia zasłanianie funkcji: "Klasy pochodne mogą zasłonić metody klas bazowych. Ponieważ metody nie mają żadnych specjalnych przywilejów, kiedy wywołujemy inne metody tego samego obiektu, może okazać się, że metoda klasy bazowej wywołująca inną metodę zdefiniowaną w tej samej klasie bazowej wywołuje właściwie metodę klasy pochodnej, która ją zasłania. (Dla programistów C++: wszystkie metody w Pythonie zachowują się tak, jakby były wirtualne.)" Jeśli dla Ciebie nie ma to sensu, możesz to zignorować. Po prostu warto było o tym wspomnieć.

Zawsze przypisujmy wartości początkowe wszystkim zmiennym obiektu w jego metodzie `__init__`. Oszczędzi to godzin debugowania w poszukiwaniu wyjątków `AtributeError`, które są spowodowane odwołaniami do niezainicjalizowanych (czyli nieistniejących) atrybutów.

Przykład 5.10 Standardowe metody klasy `UserDict`

```
   def clear(self): self.data.clear()       #(1)
def copy(self):                             #(2)
    if self.__class__ is UserDict:          #(3)
        return UserDict(self.data)
    import copy                             #(4)
    return copy.copy(self)
def keys(self): return self.data.keys()     #(5)
def items(self): return self.data.items()
def values(self): return self.data.values()
```

1. `clear` jest normalną metodą klasy; jest dostępna publicznie i może być wołana przez kogokolwiek w dowolnej chwili. Zauważmy, że w `clear`, jak we wszystkich metodach klas, pierwszym argumentem jest `self`. (Pamiętajmy, że nie dodajemy `self`, gdy wywołujemy metodę; Python robi to za nas.) Zwróćmy uwagę na podstawową cechę tej klasy opakowującej: przechowuje ona prawdziwy słownik w atrybucie `data` i definiuje wszystkie metody wbudowanego słownika, a w każdej z tych metod zwraca wynik identyczny do odpowiedniej metody słownika. (Gdybyśmy zapomnieli, metoda słownika `clear` czyści cały słownik kasując jego wszystkie klucze i wartości.)

2. Metoda słownika o nazwie `copy` zwraca nowy słownik, który jest dokładną kopią oryginału (mający takie same pary klucz-wartość). Natomiast klasa `UserDict` nie może po prostu wywołać `self.data.copy`, ponieważ ta metoda zwraca wbydowany słownik, a my chcemy zwrócić nową instancję klasy tej samej klasy, jaką ma `self`.

3. Używamy atrybutu `__class__`, żeby sprawdzić, czy `self` jest obiektem klasy `UserDict`; jeśli tak, to jesteśmy w domu, bo wiemy, jak zrobić kopię `UserDict`: tworzymy nowy obiekt `UserDict` i przekazujemy mu słownik wyciągnięty z `self.data`, a wtedy możemy od razu zwrócić nowy obiekt `UserDict` nie wykonując nawet instrukcji `import copy` z następnej linii.

4. Jeśli `self.__class__` nie jest `UserDict`-em, to `self` musi być jakąś podklasą `UserDict`-a, a w takim przypadku życie wymaga użycia pewnych trików. `UserDict` nie wie, jak utworzyć dokładną kopię jednego ze swoich potomków. W tym celu możemy np. znając atrybuty zdefiniowane w podklasie, wykonać na nich pętlę, podczas której kopiujemy każdy z tych atrybutów. Na szczęście istnieje moduł, który wykonuje dokładnie to samo, nazywa się on `copy`. Nie będziemy się tutaj wdawać w szczegóły (choć jest to wypaśny moduł, jeśli się w niego trochę wgłębimy). Wystarczy wiedzieć, że `copy` potrafi kopiować dowolne obiekty, a tu widzimy, jak możemy z niego skorzystać.

5. Pozostałe metody są bezpośrednimi przekierowaniami, które wywołują wbudowane metody z `self.data`.

Od Pythona 2.2 nie korzystamy z klasy `UserDict`, ponieważ od tej wersji możemy już dziedziczyć z wbudowanych typów danych.

Materiały dodatkowe

- Python Library Reference dokumentuje moduł `copy`

6.5 Metody specjalne

Pobieranie i ustawianie elementów

Oprócz normalnych metod, jest też kilka (może kilkanaście) metod specjalnych, które można definiować w klasach Pythona. Nie wywołujemy ich bezpośrednio z naszego kodu (jak zwykłe metody). Wywołuje je za nas Python w określonych okolicznościach lub gdy użyjemy określonej składni np. za pomocą metod specjalnych możemy nadpisać operację dodawania, czy też odejmowania.

Z normalnym słownikiem możemy zrobić dużo więcej, niż bezpośrednio wywołać jego metody. Same metody nie wystarczą. Możemy na przykład pobierać i wstawiać elementy dzięki wykorzystaniu odpowiedniej składni, bez jawnego wywoływania metod. Możemy tak robić dzięki metodom specjalnym. Python odpowiednie elementy składni przekształca na odpowiednie wywołania funkcji specjalnych.

Przykład 5.12 Metoda `__getitem__`

```
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3")
>>> f
{'plik':'/music/_singles/kairo.mp3'}
>>> f.__getitem__("plik")                #(1)
'/music/_singles/kairo.mp3'
>>> f["plik"]                            #(2)
'/music/_singles/kairo.mp3'
```

1. Metoda specjalna `__getitem__` wygląda dość prosto. Ta metoda specjalna pozwala słownikowi zwrócić pewną wartość na podstawie podanego klucza. A jak ta metodę możemy wywołać? Możemy to zrobić bezpośrednio, ale w praktyce nie robimy w ten sposób, byłoby to niezbyt wygodne. Najlepiej pozwolić Pythonowi wywołać tę metodę za nas.

2. Z takiej składni korzystamy, by dostać pewną wartość ze słownika. W rzeczywistości Python automatycznie przekształca taką składnię na wywołanie metody `f.__getitem__(''plik'')`. Właśnie dlatego `__getitem__` nazywamy metodą specjalną: nie tylko możemy ją wywołać, ale Python wywołuje tę metodę także za nas, kiedy skorzystamy z odpowiedniej składni.

Istnieje także analogiczna do `__getitem__` metoda `__setitem__`, która zamiast pobierać pewną wartość, zmienia daną wartość korzystając z pewnego klucza.

Przykład 5.13 Metoda `__setitem__`

```
>>> f
{'plik':'/music/_singles/kairo.mp3'}
>>> f.__setitem__("gatunek", 31)         #(1)
>>> f
{'plik':'/music/_singles/kairo.mp3', 'gatunek':31}
>>> f["gatunek"] = 32                    #(2)
>>> f
{'plik':'/music/_singles/kairo.mp3', 'gatunek':32}
```

1. Analogicznie do `__getitem__`, możemy za pomocą `__setitem__` zmienić wartość pewnego klucza znajdującego się w słowniku. Podobnie, jak w przypadku `__getitem__` nie musimy jej wywoływać w sposób bezpośredni. Python wywoła `__setitem__`, jeśli tylko użyjemy odpowiedniej składni.

2. W taki praktyczny sposób korzystamy ze słownika. Za pomocą tej linii kodu Python wywołuje w sposób ukryty `f.__setitem__(''gatunek'', 32)`.

`__setitem__` jest metodą specjalną, ponieważ Python wywołuje ją za nas, ale ciągle jest metodą klasy. Kiedy definiujemy klasy, możemy definiować pewne metody, nawet jeśli nadklasa ma już zdefiniowaną tę metodą. W ten sposób nadpisujemy (ang. *override*) metody nadklas. Tyczy się to także metod specjalnych.

Koncepcja ta jest bazą całego szkieletu, który analizujemy w tym rozdziale. Każdy typ plików może posiadać własną klasę obsługi, która wie, w jaki sposób pobrać metadane z konkretnego typu plików. Natychmiast po poznaniu niektórych atrybutów (jak nazwa pliku i położenie), klasa obsługi będzie wiedziała, jak pobrać dalsze meta-atrybuty automatycznie. Możemy to zrobić poprzez nadpisanie metody `__setitem__`, w której sprawdzamy poszczególne klucze i jeśli dany klucz zostanie znaleziony, wykonujemy dodatkowe operacje.

Na przykład `MP3FileInfo` jest podklasą `FileInfo`. Kiedy w `MP3FileInfo` ustawiamy klucz `''plik''`, nie tylko ustawiamy wartość samego klucza `''plik''` (jak to robi słownik), lecz także zaglądamy do samego pliku, odczytujemy tagi MP3 i tworzymy pełny zbiór kluczy. Poniższy przykład pokazuje, w jaki sposób to działa.

Przykład 5.14 Nadpisywanie metody `__setitem__` w klasie `MP3FileInfo`

```
def __setitem__(self, key, item):            #(1)
        if key == "plik" and item:             #(2)
            self.__parse(item)                 #(3)
        FileInfo.__setitem__(self, key, item) #(4)
```

1. Zwróćmy uwagę na kolejność i liczbę argumentów w `__setitem__`. Pierwszym argumentem jest instancja danej klasy (argument `self`), z której ta metoda została wywołana, następnym argumentem jest klucz (argument `key`), który chcemy ustawić, a trzecim jest wartość (argument `item`), którą chcemy skojarzyć z danym kluczem. Kolejność ta jest ważna, ponieważ Python będzie wywoływał tę metodą w takiej kolejności i z taką liczbą argumentów. (Nazwy argumentów nic nie znaczą, ważna jest ich ilość i kolejność.)

2. W tym miejscu zawarte jest sedno całej klasy `MP3FileInfo`: jeśli przypisujemy pewną wartość do klucza `''plik''`, chcemy wykonać dodatkowo pewne operacje.

3. Dodatkowe operacje dla klucza `''plik''` zawarte są w metodzie `__parse`. Jest to inna metoda klasy `MP3FileInfo`. Kiedy wywołujemy metodę `__parse` używamy zmiennej `self`. Gdybyśmy wywołali samo `__parse`, odnieślibyśmy się do normalnej funkcji, która jest zdefiniowana poza klasą, a tego nie chcemy wykonać. Kiedy natomiast wywołamy `self.__parse` będziemy odnosić się do metody znajdującej się wewnątrz klasy. Nie jest to niczym nowym. W identyczny sposób odnosimy się do atrybutów obiektu.

4. Po wykonaniu tej dodatkowej operacji, chcemy wykonać metodę nadklasy. Pamiętajmy, że Python nigdy nie zrobi tego za nas; musimy zrobić to ręcznie.

Zwróćmy uwagę na to, że odwołujemy się do bezpośredniej nadklasy, czyli do FileInfo, chociaż on nie posiada żadnej metody o nazwie __setitem__. Jednak wszystko jest w porządku, ponieważ Python będzie wędrował po drzewie przodków jeszcze wyżej dopóki nie znajdzie klasy, która posiada metodę, którą wywołujemy. Tak więc ta linia kodu znajdzie i wywoła metodę __setitem__, która jest zdefiniowana w samej wbudowanej klasie słownika, w klasie dict.

Kiedy odwołujemy się do danych zawartych w atrybucie instancji, musimy określić nazwę atrybutu np. self.attribute. Podczas wywoływania metody klasy, musimy określić nazwę metody np. self.method.

Przykład 5.15 Ustawianie klucza ‘‘plik’’ w MP3FileInfo

```
>>> import fileinfo
>>> mp3file = fileinfo.MP3FileInfo()                 #(1)
>>> mp3file
{'plik':None}
>>> mp3file["plik"] = "/music/_singles/kairo.mp3"     #(2)
>>> mp3file
{'album': 'Rave Mix', 'rok': '2000', 'komentarz': 'http://mp3.com/DJMARYJANE',
u'tytu\u0142': 'KAIRO****THE BEST GOA', 'artysta': '***DJ MARY-JANE***',
'gatunek': 31, 'plik': '/music/_singles/kairo.mp3'}
>>> mp3file["plik"] = "/music/_singles/sidewinder.mp3" #(3)
>>> mp3file
{'album': '', 'rok': '2000', 'nazwa': '/music/_singles/sidewinder.mp3',
'komentarz': 'http://mp3.com/cynicproject', u'tytu\u0142': 'Sidewinder',
'artysta': 'The Cynic Project', 'gatunek': 18}
```

1. Najpierw tworzymy instancję klasy MP3FileInfo bez podawania nazwy pliku. (Możemy tak zrobić, ponieważ argument filename metody __init__ jest opcjonalny.) Ponieważ MP3FileInfo nie posiada własnej metody __init__, Python idzie wyżej po drzewie nadklas i znajduje metodę __init__ w klasie FileInfo. Z kolei __init__ w tej klasie ręcznie wykonuje metodę __init__ w klasie dict, a potem ustawia klucz ‘‘plik’’ na wartość w zmiennej filename, który wynosi None, ponieważ pominęliśmy nazwę pliku. Ostatecznie mp3file początkowo jest słownikiem (właściwie klasą potomną słownika) z jednym kluczem ‘‘plik’’, którego wartość wynosi None.

2. Teraz rozpoczyna się prawdziwa zabawa. Ustawiając klucz ‘‘plik’’ w mp3file spowoduje wywołanie metody __setitem__ klasy MP3FileInfo (a nie słownika, czyli klasy dict). Z kolei metoda ta zauważa, że ustawiamy klucz ‘‘plik’’ z prawdziwą wartością (item jest prawdą w kontekście logicznym) i wywołuje self.__parse. Chociaż jeszcze nie analizowaliśmy działania metody __parse, możemy na podstawie wyjścia zobaczyć, że ustawia ona kilka innych kluczy jak ‘‘album’’, ‘‘artysta’’, ‘‘gatunek’’, u‘‘tytuł’’ (w unikodzie, bo korzystamy z polskich znaków), ‘‘rok’’, czy też ‘‘comment’’.

3. Kiedy zmienimy klucz ‘‘plik’’, proces ten zostanie wykonany ponownie. Python wywoła __setitem__, który następnie wywoła self.__parse, a ten ustawi wszystkie inne klucze.

6.6 Zaawansowane metody specjalne

Zaawansowane metody specjalne

W Pythonie, oprócz `__getitem__` i `__setitem__`, są jeszcze inne metody specjalne, . Niektóre z nich pozwalają dodać funkcjonalność, której się nawet nie spodziewamy. Ten przykład pokazuje inne metody specjalne znanej już nam klasy `UserDict`.

Przykład 5.16 Inne metody specjalne w klasie `UserDict`

```
def __repr__(self): return repr(self.data)     #(1)
def __cmp__(self, dict):                       #(2)
    if isinstance(dict, UserDict):
        return cmp(self.data, dict.data)
    else:
        return cmp(self.data, dict)
def __len__(self): return len(self.data)       #(3)
def __delitem__(self, key): del self.data[key] #(4)
```

1. `__repr__` jest metodą specjalną, która zostanie wywołana, gdy użyjemy `repr(obiekt)`. `repr` jest wbudowaną funkcją Pythona, która zwraca reprezentację danego obiektu w postaci łańcucha znaków. Działa dla dowolnych obiektów, nie tylko obiektów klas. Już wielokrotnie używaliśmy tej funkcji, nawet o tym nie wiedząc. Gdy w oknie interaktywnym wpisujemy nazwę zmiennej i naciskamy `ENTER`, Python używa `repr` do wyświetlenia wartości zmiennej. Stwórzmy słownik `d` z jakimiś danymi i wywołajmy `repr(d)`, żeby się o tym przekonać.

2. Metoda `__cmp__` zostanie wywoływana, gdy porównujemy za pomocą `==` dwa dowolne obiekty Pythona, nie tylko instancje klas. Aby porównać wbudowane typy danych (i nie tylko), wykorzystywane są pewne reguły np. słowniki są sobie równy, gdy mają dokładnie takie same pary *klucz-wartość*; łańcuchy znaków są sobie równe, gdy mają taką samą długość i zawierają taki sam ciąg znaków. Dla instancji klas możemy zdefiniować metodę `__cmp__` i zaimplementować sposób porównania własnoręcznie, a potem używać `==` do porównywania obiektów klasy. Python wywoła `__cmp__` za nas.

3. Metoda `__len__` zostanie wywołana, gdy użyjemy `len(obiekt)`. `len` jest wbudowaną funkcją Pythona, która zwraca długość obiektu. Ta metoda działa dla dowolnego obiektu, który można uznać za obiekt posiadający jakąś długość. Długość łańcucha znaków jest równa ilości jego znaków, długość słownika, to liczba jego kluczy, a długość listy lub krotki to liczba ich elementów. Dla obiektów klas możesz zdefiniować metodę `__len__` i samemu zaimplementować obliczanie długości, a następnie możemy używać `len(obiekt)`. Python za nas wywoła metodę specjalną `__len__`.

4. Metoda `__delitem__` zostanie wywołana, gdy użyjemy `del obiekt[klucz]`. Dzięki tej funkcji możemy usuwać pojedyncze elementy ze słownika. Kiedy użyjemy `del` dla pewnej instancji, Python wywoła metodę `__delitem__` za nas.

W Javie sprawdzamy, czy dwie referencje do łańcucha znaków zajmują to samo fizyczne miejsce w pamięci, używając `str1 == str2`. Jest to zwane identycznością obiektów i w Pythonie zapisywane jest jako `str1 is str2`. Aby porównać wartości łańcuchów znaków w Javie, użylibyśmy `str1.equals(str2)`. W Pythonie uzyskujemy to samo, gdy napiszemy `str1 == str2`. Programiści Javy, którzy zostali nauczeni, że świat jest lepsze, ponieważ `==` w Javie porównuje identyczność, zamiast wartości, mogą mieć trudności z akceptacją braku tego "dobra" w Pythonie.

Metody specjalne sprawiają, że dowolna klasa może przechowywać pary klucz-wartość w ten sposób, jak to robi słownik. W tym celu definiujemy metodę `__setitem__`. Każda klasa może działać jak sekwencja, dzięki metodzie `__getitem__`. Obiekty dowolnej klasy, które posiadają metodę `__cmp__`, mogą być porównywane przy użyciu `==`. A jeśli dana klasa reprezentuje coś, co ma pewną długość, nie musimy definiować metody `getLength`; po prostu definiujemy metodę `__len__` i korzystamy z `len(obiekt)`.

Inne obiektowo zorientowane języki programowania pozwalają nam zdefiniować tylko fizyczny model obiektu ("Ten obiekt ma metodę `getLength`"). Metody specjalne Pythona pozwalają zdefiniować logiczny model obiektu ("ten obiekt ma długość").

Python posiada wiele innych metod specjalnych. Są takie, które pozwalają klasie zachowywać się jak liczby, umożliwiając dodawanie, odejmowanie i inne operacje arytmetyczne na obiektach. (Klasycznym przykładem jest klasa reprezentująca liczby zespolone z częścią rzeczywistą i urojoną, na których możemy wykonywać wszystkie działania). Metoda `__call__` pozwala klasie zachowywać się jak funkcja, a dzięki czemu możemy bezpośrednio wywoływać instancję pewnej klasy.

Materiały dodatkowe

- Python Reference Manual dokumentuje wszystkie specjalne metody klasy

6.7 Atrybuty klas

Atrybuty klas

Wiemy już, co to są atrybuty, które są częścią konkretnych obiektów. W Python możemy tworzyć też atrybuty klas, czyli zmienne należące do samej klasy (a nie do instancji tej klasy).

Przykład 5.17 Atrybuty klas

```
class MP3FileInfo(FileInfo):
    u"przechowuje znaczniki ID3v1.0 MP3"
    tagDataMap = {u"tytuł"    : (  3,  33, stripnulls),
                  "artysta"   : ( 33,  63, stripnulls),
                  "album"     : ( 63,  93, stripnulls),
                  "rok"       : ( 93,  97, stripnulls),
                  "komentarz" : ( 97, 126, stripnulls),
                  "gatunek"   : (127, 128, ord)}
```

```
>>> import fileinfo
>>> fileinfo.MP3FileInfo                     #(1)
<class 'fileinfo.MP3FileInfo'>
>>> fileinfo.MP3FileInfo.tagDataMap          #(2)
{'album': (63, 93, <function stripnulls at 0xb7c000d4>),
'rok': (93, 97, <function stripnulls at 0xb7c000d4>),
'komentarz': (97, 126, <function stripnulls at 0xb7c000d4>),
u'tytu\u0142': (3, 33, <function stripnulls at 0xb7c000d4>),
'artysta': (33, 63, <function stripnulls at 0xb7c000d4>),
'gatunek': (127, 128, <built-in function ord>)}
>>> m = fileinfo.MP3FileInfo()               #(3)
>>> m.tagDataMap
{'album': (63, 93, <function stripnulls at 0xb7c000d4>),
'rok': (93, 97, <function stripnulls at 0xb7c000d4>),
'komentarz': (97, 126, <function stripnulls at 0xb7c000d4>),
u'tytu\u0142': (3, 33, <function stripnulls at 0xb7c000d4>),
'artysta': (33, 63, <function stripnulls at 0xb7c000d4>),
'gatunek': (127, 128, <built-in function ord>)}
```

1. `MP3FileInfo` jest klasą, nie jest instancją klasy.
2. `tagDataMap` jest atrybutem klasy i jest dostępny już przed stworzeniem jakiegokolwiek obiektu danej klasy.
3. Atrybuty klas są dostępne na dwa sposoby: poprzez bezpośrednie odwołanie do klasy lub poprzez jakąkolwiek instancjętej klasy.

W Javie zarówno zmienne statyczne (w Pythonie zwane atrybutami klas), jak i zmienne obiektów (w Pythonie zwane atrybuty lub atrybutami instancji) są definiowane bezpośrednio za definicją klasy (jedne ze słowem kluczowym `static`, a inne bez niego). W Pythonie tylko atrybuty klas są definiowane bezpośrednio po definicji klasy. Atrybuty instancji definiujemy w metodzie `__init__`.

Atrybuty klas mogą być używane jako stałe tych klas (w takim celu korzystamy z nich w klasie `MP3FileInfo`), ale tak naprawdę nie są stałe. Można je zmieniać.

W Pythonie nie istnieją stałe. Wszystko możemy zmienić, jeśli tylko odpowiednio się postaramy. To efekt jednej z podstawowych zasad Pythona: powinno się zniechęcać do złego działania, ale nie zabraniać go. Jeśli naprawdę chcesz zmienić wartość `None`, możesz to zrobić, ale nie przychodź z płaczem, kiedy twój kod stanie się niemożliwy do debugowania.

Przykład 5.18 Modyfikowanie atrybutów klas

```
>>> class counter(object):
...     count = 0                     #(1)
...     def __init__(self):
...         self.__class__.count += 1   #(2)
...
>>> counter
<class __main__.counter>
>>> counter.count                     #(3)
0
>>> c = counter()
>>> c.count                           #(4)
1
>>> counter.count
1
>>> d = counter()                     #(5)
>>> d.count
2
>>> c.count
2
>>> counter.count
2
```

1. `count` jest atrybutem klasy `counter`.
2. `__class__` jest wbudowanym atrybutem każdego obiektu. Jest to referencja do klasy, której obiektem jest `self` (w tym wypadku do klasy `counter`).
3. Ponieważ `count` jest atrybutem klasy, jest dostępny porzez bezpośrednie odwołanie do klasy, przed stworzeniem jakiegokolwiek obiektu.
4. Kiedy tworzymy instancję tej klasy, automatycznie zostanie wykonana metoda `__init__`, która zwiększa atrybut tej klasy o nazwie `count` o `1`. Operacja ta wpływa na klasę samą w sobie, a nie tylko na dopiero co stworzony obiekt.
5. Stworzenie drugiego obiektu ponownie zwiększy atrybut `count`. Zauważmy, że atrybuty klas są wspólne dla klasy i wszystkich jej instancji.

6.8 Funkcje prywatne

Funkcje prywatne

Jak większość języków programowania, Python posiada koncepcję elementów prywatnych:

- prywatne funkcje, które nie mogą być wywoływane spoza modułów w których są zdefiniowane
- prywatne metody klas, które nie mogą być spoza nich wywołane
- prywatne atrybuty do których nie ma dostępu spoza klasy

Inaczej niż w większości języków, to czy element jest prywatny, czy nie, zależy tylko od jego nazwy.

Jeżeli nazwa funkcji, metody czy atrybutu zaczyna się od dwóch podkreśleń (ale nie kończy się nimi), to wtedy element ten jest prywatny, wszystko inne jest publiczne. Python nie posiada koncepcji chronionych metod (tak jak na przykład w Javie, C++, które są dostępnych tylko w tej klasie oraz w klasach z niej dziedziczących). Metody mogą być tylko prywatne (dostępne tylko z wnętrza klasy), bądź publiczne (dostępne wszędzie).

W klasie `MP3FileInfo` istnieją tylko dwie metody: `__parse` oraz `__setitem__`. Jak już zostało przedstawione, `__setitem__` jest metodą specjalną: zostanie wywołana, gdy skorzystamy ze składni dostępu do słownika bezpośrednio na instancji danej klasy. Jednak ta metoda jest publiczna i można ją wywołać bezpośrednio (nawet z zewnątrz modułu `fileinfo`), jeśli istnieje jakiś dobry do tego powód. Jednak metoda `__parse` jest prywatna, ponieważ posiada dwa podkreślenia na początku nazwy.

W Pythone wszystkie specjalne metody (takie jak `__setitem__`) oraz wbudowane atrybuty (np. `__doc__`) trzymają się standardowej konwencji nazewnictwa. Ich nazwy zaczynają się oraz kończą dwoma podkreśleniami. Nie nazywajmy własnych metod w ten sposób, bo jeśli tak będziemy robić, możemy wprawić w zakłopotanie nie tylko siebie, ale i inne osoby czytające nasz kod później.

Przykład 5.19 Próba wywołania metody prywatnej

```
>>> import fileinfo
>>> m = fileinfo.MP3FileInfo()
>>> m.__parse("/music/_singles/kairo.mp3")    #(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'MP3FileInfo' object has no attribute '__parse'
```

1. Jeśli spróbujemy wywołać prywatną metodę, Python rzuci nieco mylący wyjątek, który informuje, że metoda nie istnieje. Oczywiście istnieje ona, lecz jest prywatna, więc nie jest możliwe uzyskanie dostępu do niej spoza klasy. Ściśle mówiąc, metody prywatne są dostępne spoza klasy w której były zdefiniowane, jednak nie w taki prosty sposób. Tak naprawdę nic w Pythonie nie jest prywatne, nazwy metod prywatnych są kodowane oraz odkodowane "w locie", aby wyglądały na niedostępne, gdy korzystamy z ich prawdziwych nazw. Możemy jednak wywołać

metodę `__parse` klasy `MP3FileInfo` przy pomocy nazwy `_MP3FileInfo.__parse`. Informacja ta jest dosyć interesująca, jednak obiecajmy sobie nigdy nie korzystać z niej w prawdziwym kodzie. Metody prywatne są prywatne nie bez powodu, lecz jak wiele rzeczy w Pythonie, ich prywatność jest kwestią konwencji, nie przymusu.

Materiały dodatkowe

- Python Tutorial omawia działanie prywatnych zmiennych od środka

6.9 Obiekty i klasy - podsumowanie

Podsumowanie

To wszystko, jeśli chodzi o triki z obiektami. W rozdziale dwunastym zobaczymy, w jaki sposób wykorzystywać metody specjalne w normalnej aplikacji w której będziemy zajmowali się tworzeniem pośredników (ang. *proxy*) dla zdalnych usług sieciowych z użyciem `getattr`.

W następnym rozdziale dalej będziemy używać kodu programu fileinfo.py, aby poznać takie pojęcia jak wyjątki, operacje na plikach oraz pętlę `for`.

Zanim zanurkujemy w następnym rozdziale, upewnijmy się, że nie mamy problemów z:

- definiowaniem i tworzeniem instancji klasy
- definiowaniem metody __init__ oraz innych metod specjalnych, a także wiem, kiedy są one wywoływane
- dziedziczeniem innych klas (w ten sposób tworząc podklasę danej klasy)
- definiowaniem atrybutów instancji i atrybutów klas, a także rozumiemy różnice między nimi
- definiowaniem prywatnych metod oraz atrybutów

Rozdział 7

Wyjątki i operacje na plikach

7.1 Obsługa wyjątków

W tym rozdziale zajmiemy się wyjątkami, obiektami pliku, pętlami `for` oraz modułami `os` i `sys`. Jeśli używaliśmy wyjątków w innych językach programowania, możemy tylko szybko przyjrzeć się składni Pythona, która odpowiada za obsługę wyjątków, ale powinniśmy zwrócić uwagę na część, która omawia w jaki sposób zarządzać plikami.

Obsługa wyjątków

Jak wiele innych języków programowania, Python obsługuje wyjątki. Przy pomocy bloków `try...except` przechwytujemy wyjątki, natomiast `raise` zaś rzuca wyjątek.

Python używa słów kluczowych `try` i `except` do obsługi wyjątków, natomiast za pomocą słowa `raise` wyrzuca wyjątki. Java i C++ używają słów `try` i `catch` do przechwytywania wyjątków, a słowa `throw` do ich generacji.

Wyjątki są w Pythonie wszędzie. Praktycznie każdy moduł w bibliotece standardowej Pythona ich używa. Sam interpreter Pythona również rzuca wyjątki w różnych sytuacjach. Już wiele razy widzieliśmy je w tej książce:

- Próba użycia nieistniejącego klucza w słowniku rzuci wyjątek `KeyError`
- Wyszukiwanie w liście nieistniejącej wartości rzuci wyjątek `ValueError`
- Wywołanie nieistniejącej metody obiektu rzuci wyjątek `AttributeError`
- Użycie nieistniejącej zmiennej rzuci wyjątek `NameError`
- Mieszanie niezgodnych typów danych spowoduje wyjątek `TypeError`

W każdym z tych przypadków, gdy używaliśmy IDE Pythona i wystąpił błąd, to został wypisany wyjątek (w zależności od użytego IDE na przykład na czerwono). Jest to tak zwany *nieobsłużony* wyjątek. Kiedy podczas wykonywania programu został rzucony wyjątek, nie było w nim specjalnego kodu, który by go wykrył i zaznajomił się z nim, dlatego obsługa tego wyjątku zostaje zrzucona na domyślne zachowanie Pythona, które z kolei wypisuje trochę informacji na temat błędu i kończy pracę programu. W przypadku IDE nie jest to wielka przeszkoda, ale wyobraźmy sobie, co by się stało, gdyby podczas wykonywania właściwego programu nastąpiłby taki błąd, a co z kolei spowodowałoby, że program wyłączyłby się.

Jednak efektem wyjątku nie musi być katastrofa programu. Kiedy wyjątki zostaną rzucone, mogą zostać *obsłużone*. Czasami przyczyną wystąpienia wyjątku jest błąd w kodzie (na przykład próba użycia zmiennej, która nie istnieje), jednak bardzo często wyjątek możemy przewidzieć. Jeśli otwieramy plik, może on nie istnieć. Jeśli łączysz się z bazą danych, może ona być niedostępna lub możemy nie mieć odpowiednich parametrów dostępu np. hasła. Jeśli wiemy, że jakaś linia kodu może wygenerować wyjątek, powinniśmy próbować ją obsłużyć przy pomocy bloku `try...except`.

Przykład 6.1 Otwieranie nieistniejącego pliku

```
>>> fsock = open("/niemapliku", "r")          #(1)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: '/niemapliku'
>>> try:
...     fsock = open("c:/niemapliku.txt")      #(2)
... except IOError:
...     print "Plik nie istnieje"
... print "Ta linia zawsze zostanie wypisana"  #(3)
Plik nie istnieje
Ta linia zawsze zostanie wypisana
```

1. Używając wbudowanej funkcji `open`, możemy spróbować otworzyć plik do odczytu (więcej o tej funkcji w następnym podrozdziale). Jednak ten plik nie istnieje i dlatego zostanie rzucony wyjątek `IOError`. Ponieważ nie przechwytujemy tego wyjątku Python po prostu wypisuje trochę danych pomocnych przy znajdywaniu błędu, a potem zakańcza działanie programu.

2. Próbujemy otworzyć ten sam nieistniejący plik, jednak tym razem robimy to we wnętrzu bloku `try...except`. Gdy metoda `open` rzuca wyjątek `IOError`, jesteśmy na to przygotowany. Linia `except IOError:` przechwytuje ten wyjątek i wykonuje blok kodu, który w tym wypadku wypisuje bardziej przyjazny opis błędu.

3. Gdy wyjątek zostanie już obsłużony, program wykonuje się dalej w sposób normalny, od pierwszej linii po bloku `try...except`. Zauważmy, że ta linia zawsze wypisze tekst "Ta linia zawsze zostanie wypisana", niezależnie, czy wyjątek wystąpi, czy też nie. Jeśli naprawdę mielibyśmy taki plik na dysku, to i tak ta linia zostałaby wykonana.

Wyjątki mogą wydawać się nieprzyjazne (w końcu, jeśli nie przechwycimy wyjątku, program zostanie przerwany), jednak pomyślmy, z jakiej innej alternatywy moglibyśmy skorzystać. Czy chcielibyśmy dostać bezużyteczny obiekt, który przestawia nieistniejący plik? I tak musielibyśmy sprawdzić jego poprawność w jakiś sposób, a jeśli byśmy tego nie zrobili, to nasz program wykonałby jakieś dziwne, nieprzewidywalne operacje, których byśmy się nie spodziewali. Nie byłaby to wcale dobra zabawa. Z wyjątkami błędy występują natychmiast i możemy je obsługiwać w standardowy sposób u źródła problemu.

Wykorzystanie wyjątków do innych celów

Jest wiele innych sposobów wykorzystania wyjątków, oprócz do obsługi błędów. Dobrym przykładem jest importowanie modułów Pythona, sprawdzając czy nastąpił wyjątek. Jeśli moduł nie istnieje zostanie rzucony wyjątek `ImportError`. Dzięki temu możemy zdefiniować wiele poziomów funkcjonalności, które zależą od modułów dostępnych w czasie wykonania, a dzięki temu możemy wspierać różnorodne platformy (kod zależny od platformy jest podzielony na oddzielne moduły).

Możemy też zdefiniować własne wyjątki, tworząc klasę, która dziedziczy z wbudowanej klasy `Exception`, a następnie możemy rzucać wyjątki przy pomocy polecenia `raise`. Możemy zajrzeć do części "materiały dodatkowe", aby dowiedzieć się więcej na ten temat.

Następny przykład pokazuje, w jaki sposób wykorzystywać wyjątki, aby obsłużyć funkcjonalność zdefiniowaną jedynie dla konkretnej platformy. Kod pochodzi z modułu

`getpass`, który jest modułem opakowującym, którym umożliwia pobranie hasła od użytkownika. Pobieranie hasła jest całkowicie różne na platformach UNIX, Windows, czy Mac OS, ale kod ten obsługuje wszystkie te różnice.

Przykład 6.2 Obsługa funkcjonalności zdefiniowanej dla konkretnej platformy

```
# Bind the name getpass to the appropriate function
try:
    import termios, TERMIOS                    #(1)
except ImportError:
    try:
        import msvcrt                          #(2)
    except ImportError:
        try:
            from EasyDialogs import AskPassword #(3)
        except ImportError:
            getpass = default_getpass          #(4)
        else:                                  #(5)
            getpass = AskPassword
    else:
        getpass = win_getpass
else:
    getpass = unix_getpass
```

1. `termios` jest modułem określonym dla UNIX-a, który dostarcza niskopoziomową kontrolę nad terminalem wejścia. Jeśli moduł ten jest niedostępny (ponieważ, nie ma tego na naszym systemie, ponieważ system tego nie obsługuje), importowanie nawali, a Python rzuci wyjątek `ImportError`, który przechwycimy.

2. OK, nie mamy `termios`, więc spróbujmy z `msvcrt`, który jest modułem charakterystycznym dla systemu Windows, a dostarcza on API do wielu przydatnych funkcji dla tego systemu. Jeśli to także nie zadziała, Python rzuci wyjątek `ImportError`, który także przechwycimy.

3. Jeśli pierwsze dwa nie zadziałają, próbujemy zaimportować funkcję z `EasyDialogs`, która jest w module określonym dla Mac OS-a, który dostarcza funkcje przeznaczone dla wyskakujących okien dialogowych różnego typu. I ponownie, jeśli moduł nie istnieje, Python rzuci wyjątek `ImportError`, który też przechwytujemy.

4. Żaden z tych modułów, które są przeznaczone dla konkretnej platformy, nie są dostępne (jest to możliwe, ponieważ Python został przeportowany na wiele różnych platform), więc musimy powrócić do domyślnej funkcji do pobierania hasła (która jest zdefiniowana gdzieś w module `getpass`). Zauważmy, co robimy: przypisujemy funkcję `default_getpass` do zmiennej `getpass`. Jeśli czytaliśmy oficjalną dokumentację `getpass`, mówi ona, że moduł `getpass` definiuje funkcję `getpass`. Wykonuje to poprzez powiązanie `getpass` z odpowiednią funkcją, która zależy od naszej platformy. Kiedy wywołujemy funkcję `getpass`, tak naprawdę wywołujemy funkcję określoną dla konkretnej platformy, którą określił za nas powyższy kod. Nie musimy się martwić, na jakiej platformie uruchamiamy nasz kod – wywołujemy tylko `getpass`, a funkcja ta wykona zawsze odpowiednią czynność.

5. Blok `try...except`, podobnie jak instrukcja `if`, może posiadać klauzule `else`. Jeśli żaden wyjątek nie zostanie rzucony podczas wykonywania bloku `try`, spowoduje to wywołanie klauzuli `else`. W tym przypadku oznacza to, że import `from EasyDialogs import AskPassword` zadziałał, a więc możemy powiązać `getpass` z funkcją `AskPassword`. Każdy inny blok `try...except` w przedstawionym kodzie posiada podobną klauzulę `else`, która pozwala, gdy zostanie zaimportowany działający moduł, przypisać do `getpass` odpowiednią funkcję.

Materiały dodatkowe

- Python Tutorial mówi na temat definiowania, rzucania własnych wyjątków i jednoczesnej obsługi wielu wyjątków
- Python Library Reference opisuje wszystkie wbudowane wyjątki
- Python Library Reference dokumentuje moduł `getpass`.
- Python Library Reference dokumentuje moduł `traceback`, który zapewnia niskopoziomowy dostęp do atrybutów wyjątków, po tym, jak wyjątek zostanie rzucony.
- Python Reference Manual omawia bardziej technicznie blok `try...except`.

7.2 Praca na plikach

Praca z obiektami plików

Python posiada wbudowaną funkcję `open`, służącą do otwierania plików z dysku. `open` zwraca obiekt pliku posiadający metody i atrybuty, dzięki którym możemy dostać się do pliku i wykonywać na nim pewne operacje.

Przykład 6.3 Otwieranie plików

```
>>> f = open("/muzyka/_single/kairo.mp3", "rb")                    #(1)
>>> f                                                              #(2)
<open file '/muzyka/_single/kairo.mp3', mode 'rb' at 010E3988>
>>> f.mode                                                         #(3)
'rb'
>>> f.name                                                         #(4)
'/muzyka/_single/kairo.mp3'
```

1. Metoda `open` przyjmuje do trzech argumentów: nazwę pliku, tryb i argument buforowania. Tylko pierwszy z nich, nazwa pliku, jest wymagany; pozostałe dwa są opcjonalne. Jeśli nie są podane, plik zostanie otwarty do odczytu w trybie tekstowym. Tutaj otworzyliśmy plik do odczytu w trybie binarnym. (`print open.__doc__` da nam świetne objaśnienie wszystkich możliwych trybów.)
2. Metoda `open` zwraca obiekt (w tym momencie nie powinno to już być zaskoczeniem). Obiekt pliku ma kilka użytecznych atrybutów.
3. Atrybut `mode` obiektu pliku mówi nam, w jakim trybie został on otwarty.
4. Atrybut `name` zwraca ścieżkę do pliku, który jest dostępny z tego obiektu.

Czytanie z pliku

Otworzywszy plik, będziemy chcieli odczytać z niego informacje, tak jak pokazano to w następnym przykładzie.

Przykład 6.4 Czytanie pliku

```
>>> f
<open file '/muzyka/_single/kairo.mp3', mode 'rb' at 010E3988>
>>> f.tell()                                                       #(1)
0
>>> f.seek(-128, 2)                                                #(2)
>>> f.tell()                                                       #(3)
7542909
>>> tagData = f.read(128)                                          #(4)
>>> tagData
'TAGKAIRO****THE BEST GOA         ***DJ MARY-JANE***
Rave Mix                      2000http://mp3.com/DJMARYJANE     \037'
>>> f.tell()                                                       #(5)
7543037
```

1. Obiekt pliku przechowuje stan otwartego pliku. Metoda `tell` zwraca aktualną pozycję w otwartym pliku. Z uwagi na to, że nie robiliśmy jeszcze nic z tym plikiem, aktualna pozycja to `0`, czyli początek pliku.

2. Metoda `seek` obiektu pliku służy do poruszania się po otwartym pliku. Jej drugi argument określa znaczenie pierwszego argument; jeśli argument drugi wynosi `0`, oznacza to, że pierwszy argument odnosi się do pozycji bezwzględnej (czyli licząc od początku pliku), `1` oznacza przeskok do innej pozycji względem pozycji aktualnej (licząc od pozycji aktualnej), `2` oznacza przeskok do danej pozycji względem końca pliku. Jako że tagi MP3, o które nam chodzi, przechowywane są na końcu pliku, korzystamy z opcji `2` i przeskakujemy do pozycji oddalonej o 128 bajtów od końca pliku.

3. Metoda `tell` potwierdza, że rzeczywiście zmieniliśmy pozycję pliku.

4. Metoda `read` czyta określoną liczbę bajtów z otwartego pliku i zwraca dane w postaci łańcucha znaków, które zostały odczytane. Opcjonalny argument określa maksymalną liczbę bajtów do odczytu. Jeśli nie zostanie podany argument, `read` będzie czytał do końca pliku. (W tym przypadku moglibyśmy użyć samego `read()`, ponieważ wiemy dokładnie w jakiej pozycji w pliku jesteśmy i w rzeczywistości odczytujemy ostanie 128 bajtów.) Odczytane dane przypisujemy do zmiennej `tagData`, a bieżąca pozycja zostaje uaktualniana na podstawie ilości odczytanych bajtów.

5. Metoda `tell` potwierdza, że zmieniła się bieżąca pozycja. Jeśli pokusimy się o wykonanie obliczenia, zauważymy, że po odczytaniu `128` bajtów aktualna pozycja wzrosła o `128`.

Zamykanie pliku

Otwarte pliki zajmują zasoby systemu, a inne aplikacje czasami mogą nie mieć do nich dostępu (zależy to od trybu otwarcia pliku), dlatego bardzo ważne jest zamykanie plików tak szybko, jak tylko skończymy na nich pracę.

Przykład 6.5 Zamykanie pliku

```
>>> f
<open file '/muzyka/_single/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed                                                    #(1)
False
>>> f.close()                                                   #(2)
>>> f
<closed file '/muzyka/_single/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed                                                    #(3)
True
>>> f.seek(0)                                                   #(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.tell()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
ValueError: I/O operation on closed file
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.close()                                                    #(5)
```

1. Atrybut `closed` obiektu pliku mówi, czy plik jest otwarty, czy też nie. W tym przypadku plik jeszcze jest otwarty (`closed` jest równe `False`).

2. Aby zamknąć plik należy wywołać metodę `close` obiektu pliku. Zwalnia to blokadę, która nałożona była na plik (jeśli była nałożona), oczyszcza buforowane dane (jeśli jakiekolwiek dane w nim występują), które system nie zdążył jeszcze rzeczywiście zapisać, a następnie zwalnia zasoby.

3. Atrybut `closed` potwierdza, że plik jest zamknięty.

4. To że plik został zamknięty, nie oznacza od razu, że obiekt przestaje istnieć. Zmienna `f` będzie istnieć póki nie wyjdzie poza swój zasięg, lub nie zostanie skasowana ręcznie. Aczkolwiek żadna z metod służących do operowania na otwartym pliku, nie będzie działać od momentu jego zamknięcia; wszystkie rzucają wyjątek.

5. Wywołanie `close` na pliku uprzednio zamkniętym nie zwraca wyjątku; w przypadku błędu cicho sobie z nim radzi.

Błędy wejścia/wyjścia

Zrozumienie kodu fileinfo.py z poprzedniego rozdziału, nie powinno już być problemem. Kolejny przykład pokazuje, jak bezpiecznie otwierać i zamykać pliki oraz jak należycie obchodzić się z błędami.

Przykład 6.6 Obiekty pliku w MP3FileInfo

```
try:                                                             #(1)
    fsock = open(filename, "rb", 0)                              #(2)
    try:
        fsock.seek(-128, 2)                                      #(3)
        tagdata = fsock.read(128)                                #(4)
    finally:                                                     #(5)
        fsock.close()


except IOError:                                                  #(6)
    pass
```

1. Ponieważ otwieranie pliku i czytanie z niego jest ryzykowne, a także operacje te mogą rzucić wyjątek, cały ten kod jest ujęty w blok `try...except`. (Hej, czy zestandaryzowane wcięcia nie są świetne? To moment, w którym zaczynasz je doceniać.)

2. Funkcja `open` może rzucić wyjątek `IOError`. (Plik może nie istnieć.)

3. Funkcja seek może rzucić wyjątek IOError. (Plik może być mniejszy niż 128 bajtów.)

4. Funkcja read może rzucić wyjątek IOError. (Być może dysk posiada uszkodzony sektor, albo plik znajduje się na dysku sieciowym, a sieć właśnie przestała działać.)

5. Nowość: blok try...finally. Nawet po udanym otworzeniu pliku przez open chcemy być całkowicie pewni, że zostanie on zamknięty niezależnie od tego, czy metody seek i read rzucą wyjątki, czy też nie. Właśnie do takich rzeczy służy blok try...finally: kod z bloku finally zostanie zawsze wykonany, nawet jeśli jakaś instrukcja bloku try rzuci wyjątek. Należy o tym myśleć jak o kodzie wykonywanym na zakończenie operacji, niezależnie od tego co działo się wcześniej.

6. Nareszcie poradzimy sobie z wyjątkiem IOError. Może to być wyjątek wywołany przez którąkolwiek z funkcji open, seek, czy read. Tym razem nie jest to dla nas istotne, gdyż jedyną rzeczą, którą zrobimy to zignorowanie tego wyjątku i kontynuowanie dalszej pracy programu. (Pamiętajmy, pass jest wyrażeniem Pythona, które nic nie robi.) Takie coś jest całkowicie dozwolone; to że przechwyciliśmy dany wyjątek, nie oznacza, że musimy z nim cokolwiek robić. Wyjątek zostanie potraktowany tak, jakby został obsłużony, a kod będzie normalnie kontynuowany od następnej linijki kodu po bloku try...except.

Zapisywanie do pliku

Jak można przypuszczać, istnieje również możliwość zapisywania danych do plików w sposób bardzo podobny do odczytywania. Wyróżniamy dwa podstawowe tryby otwierania plików:

- w trybie "append", w którym dane będą dodawane na końcu pliku
- w trybie "write", w którym plik zostanie nadpisany.

Oba tryby, jeśli plik nie będzie istniał, utworzą go automatycznie, dlatego nie ma potrzeby na fikuśne działania typu "jeśli dany plik jeszcze nie istnieje, utwórz nowy pusty plik, aby móc go otworzyć po raz pierwszy". Po prostu otwieramy plik i zaczynamy do niego zapisywać dane.

Przykład 6.7 Pisanie do pliku

```
>>> logfile = open('test.log', 'w')                                      #(1)
>>> logfile.write('udany test')                                          #(2)
>>> logfile.close()
>>> print open('test.log').read()                                        #(3)
udany test
>>> logfile = open('test.log', 'a')                                      #(4)
>>> logfile.write('linia 2')
>>> logfile.close()
>>> print open('test.log').read()                                        #(5)
udany testlinia 2
```

1. Zaczynamy odważnie: tworzymy nowy lub nadpisujemy istniejący plik `test.log`, a następnie otwieramy do zapisu. (Drugi argument `"w"` oznacza otwieranie pliku do zapisu.) Tak, jest to dokładnie tak niebezpieczne, jak brzmi. Miejmy nadzieję, że poprzednia zawartość pliku nie była istota, bo już jej nie ma.

2. Dane do nowo otwartego pliku dodajemy za pomocą metody `write` obiektu zwróconego przez `open`.

3. Ten jednowierszowiec otwiera plik, czyta jego zawartość i drukuje na ekran.

4. Przypadkiem wiemy, że `test.log` istnieje (w końcu właśnie skończyliśmy do niego pisać), więc możemy go otworzyć i dodawać dane. (Argument `ą"` oznacza otwieranie pliku w trybie dodawania danych na koniec pliku.) Właściwie moglibyśmy to zrobić nawet wtedy, gdyby plik nie istniał, ponieważ otworzenie pliku w trybie `ą"` spowoduje jego powstanie, jeśli będzie to potrzebne. Otworzenie w trybie `ą"` nigdy nie uszkodzi aktualnej zawartości pliku.

5. Jak widać, zarówno pierwotnie zapisane, jak i dopisane dane, aktualnie znajdują się w pliku `test.log`. Należy zauważyć, że znaki końca linii nie są uwzględnione. Jako że nie zapisywaliśmy znaków końca linii w żadnym z przypadków, plik ich nie zawiera. Znaki końca linii możemy zapisać za pomocą symbolu `"\n"`. Z uwagi na fakt, iż tego nie zrobiliśmy, całość danych w pliku wylądowała w jednej linijce.

Materiały dodatkowe

- Python Tutorial opisuje, jak czytać i zapisywać pliki, w tym także, jak czytać pliki linia po linii lub jak wczytać wszystkie linie na raz do listy
- Python Knowledge Base odpowiada na najczęściej zadawane pytania dotyczące plików
- Python Library Reference omawia wszystkie metody obiektu pliku.

7.3 Pętla for

Pętla `for`

Podobnie jak wiele innych języków, Python posiada pętlę `for`. Jedynym powodem, dla którego nie widzieliśmy tej pętli wcześniej jest to, że Python posiada tyle innych użytecznych rzeczy, że nie potrzebujemy jej aż tak często.

Wiele języków programowania nie posiada typu danych o takich możliwościach, jakie daje lista w Pythonie, dlatego też w tych językach trzeba wykonywać dużo manualnej pracy, trzeba określać początek, koniec i krok, aby przejść zakres liczb całkowitych, znaków lub innych iteracyjnych jednostek. Jednak pythonowa pętla `for`, przechodzi całą listę w identyczny sposób, jak ma to miejsce w wyrażeniach listowych.

Przykład 6.8 Wprowadzenie do pętli `for`

```
>>> li = ['a', 'b', 'e']
>>> for s in li:         #(1)
...     print s          #(2)
a
b
e
>>> print "\n".join(li)  #(3)
a
b
e
```

1. Składnia pętli `for` jest bardzo podobna do składni wyrażenia listowego. `li` jest listą, a zmienna `s` będzie przyjmować kolejne wartości elementów tej listy podczas wykonywania pętli, zaczynając od elementu pierwszego.
2. Podobnie jak wyrażenie `if` i inne bloki tworzone za pomocą wcięć, pętla `for` może posiadać dowolną liczbę linii kodu.
3. To główna przyczyna, dla której jeszcze nie widzieliśmy pętli `for`. Po prostu jej nie potrzebowaliśmy. Jest to zadziwiające, jak często trzeba wykorzystywać pętle `for` w innych językach, podczas gdy tak naprawdę potrzebujemy metody `join`, czy też wyrażenia listowego.

Wykonanie "normalnej" (zgodnie ze standardami Visual Basica), licznikowej pętli `for` jest także bardzo proste.

Przykład 6.9 Prosty licznik

```
>>> for i in range(5):             #(1)
...     print i
0
1
2
3
4
>>> li = ['a', 'b', 'c', 'd', 'e']
>>> for i in range(len(li)):       #(2)
```

```
...     print li[i]
a
b
c
d
e
```

1. Jak zobaczyliśmy w przykładzie 3.23, "Przypisywanie kolejnych wartości", funkcja `range` tworzy listę liczb całkowitych, a tą listę będziemy chcieli przejść za pomocą pętli. Powyższy kod być może wygląda trochę dziwacznie, lecz bywa przydatny do tworzenia pętli licznikowej.
2. Nigdy tego nieróbmy. Jest to visualbasicowy styl myślenia. Skończmy z nim. Powinno się iterować listę, jak pokazano to w poprzednim przykładzie.

Pętle `for` nie zostały stworzone do tworzenia prostych pętli licznikowych. Służą one raczej do przechodzenia po wszystkich elementach w danym obiekcie. Poniżej pokazano przykład, jak można iterować po słowniku za pomocą pętli `for`:

Przykład 6.10 Iterowanie po elementach słownika

```
>>> import os
>>> for k, v in os.environ.items():      #(1) (2)
...     print "%s=%s" % (k, v)
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...ciach...]
>>> print "\n".join(["%s=%s" % (k, v)
...     for k, v in os.environ.items()]) #(3)
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...ciach...]
```

1. `os.environ` jest słownikiem zmiennych środowiskowych zdefiniowanych w systemie operacyjnym. W Windowsie mamy tutaj zmienne użytkownika i systemu dostępne z MS-DOS-a. W Uniksie mamy tu zmienne wyeksportowane w twojej powłoce przez skrypty startowe. W systemie Mac OS nie ma czegoś takiego jak zmienne środowiskowe, więc słownik ten jest pusty.
2. `os.environ.items()` zwraca listę krotek w postaci `[(klucz1, wartosc1), (klucz2, wartosc2), ...]`. Pętla `for` iteruje po tej liście. W pierwszym przebiegu pętli, przypisuje ona wartość `klucz1` do zmiennej `k`, a wartość `wartosc1` do `v`, więc `k = ''USERPROFILE''`, a `v = ''C:\Documents and Settings\mpilgrim''`. W drugim przebiegu pętli, `k` przyjmuje wartość drugiego klucza, czyli `''OS''`, a `v` bierze odpowiadającą temu kluczowi wartość, czyli `''Windows_NT''`.

3. Za pomocą wielozmiennego przypisania i wyrażeń listowych, możemy zastąpić całą pętle `for` jednym wyrażeniem. Z której metody będziemy korzystać w kodzie, jest kwestią stylu pisania. Niektórym się to może podobać, ponieważ za pomocą wyrażenia listowego jasno stwierdzamy, że to co robimy to odwzorowywanie słownika na listę, a następnie łączymy otrzymaną listę w jeden napis. Inni z kolei preferują pisanie z użyciem pętli `for`. Otrzymane wyjście programu jest identyczne w obydwu przypadkach, jakkolwiek wersja w tym przykładzie jest nieco szybsza, ponieważ tutaj tylko raz wykorzystujemy instrukcję `print`.

Spójrzmy teraz na pętlę `for` w `MP3FileInfo` z przykładu fileinfo.py wprowadzonego w rozdziale 5.

Przykład 6.11 Pętla `for` w `MP3FileInfo`

```
tagDataMap = {u"tytuł"     : (  3,  33, stripnulls),
                "artysta"   : ( 33,  63, stripnulls),
                "album"     : ( 63,  93, stripnulls),
                "rok"       : ( 93,  97, stripnulls),
                "komentarz" : ( 97, 126, stripnulls),
                "gatunek"   : (127, 128, ord)}                          #(1)
    [...]
            if tagdata[:3] == 'TAG':
                for tag, (start, end, parseFunc) in self.tagDataMap.items():  #(2)
                    self[tag] = parseFunc(tagdata[start:end])               #(3)
```

1. `tagDataMap` jest atrybutem klasy, który definiuje tagi, jakie będziemy szukali w pliku MP3. Tagi są przechowywane w polach o ustalonej długości. Ponieważ czytamy ostatnie 128 bajtów pliku, bajty od 3 do 32 z przeczytanych danych są zawsze tytułem utworu, bajty od 33 do 62 są zawsze nazwą artysty, od 63 do 92 mamy nazwę albumu itd. Zauważmy, że `tagDataMap` jest słownikiem krotek, a każda krotka przechowuje dwie liczby całkowite i jedną referencję do funkcji.

2. Wygląda to skomplikowanie, jednak takie nie jest. Struktura zmiennych w pętli `for` odpowiada strukturze elementów listy zwróconej poprzez metodę `items`. Pamiętamy, że `items` zwraca listę krotek w formie `(klucz, wartosc)`. Jeśli pierwszym elementem tej listy jest `(u"tytuł", (3, 33, <function stripnulls at 0xb7c91f7c>))`, tak więc podczas pierwszego przebiegu pętli, `tag` jest równe `u"tytuł"`, `start` przyjmuje wartość 3, `end` wartość 33, a `parseFunc` zawiera funkcję `stripnulls`.

3. Kiedy już wydobędziemy wszystkie parametry tagów pliku MP3, zapisanie danych odnoszących się do określonych tagów będzie proste. W tym celu wycinamy napis `tagdata` od wartości w zmiennej `start` do wartości w zmiennej `end`, aby pobrać aktualne dane dla tego tagu, a następnie wywołujemy funkcję `parseFunc`, aby przetworzyć te dane, a potem przypisujemy zwróconą wartość do klucza `tag` w słowniku `self` (ściślej, `self` jest instancją podklasy słownika). Po przejściu wszystkich elementów w `tagDataMap`, `self` będzie przechowywał wartości dla wszystkich tagów, a my będziemy mogli się dowiedzieć o utworze, co tylko będziemy chcieli.

7.4 Korzystanie z sys.modules

Korzystanie z `sys.modules`

Moduły, podobnie jak wszystko inne w Pythonie, są obiektami. Jeśli wcześniej zaimportowaliśmy pewien moduł, możemy pobrać do niego referencję za pośrednictwem globalnego słownika `sys.modules`.

Przykład 6.12 Wprowadzenie do `sys.modules`

```
>>> import sys                          #(1)
>>> print '\n'.join(sys.modules.keys()) #(2)
win32api
os.path
os
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
```

1. Moduł `sys` zawiera informacje dotyczące systemu jak np. wersję uruchomionego Pythona (`sys.version` lub `sys.version_info`) i opcje dotyczące systemu np. maksymalna dozwolona głębokość rekurencji (`sys.getrecursionlimit()` i `sys.setrecursionlimit()`).
2. `sys.modules` jest słownikiem zawierającym wszystkie moduły, które zostały zaimportowane od czasu startu Pythona. W słowniku tym kluczem jest nazwa danego modułu, a wartością jest obiekt tego modułu. Dodajmy, że jest tu więcej modułów niż nasz program zaimportował. Python wczytuje niektóre moduły podczas startu, a jeśli używasz IDE Pythona , `sys.modules` zawiera wszystkie moduły zaimportowane przez wszystkie programy uruchomione wewnątrz IDE.

Poniższy przykład pokazuje, jak wykorzystywać `sys.modules`.

Przykład 6.13 Korzystanie z `sys.modules`

```
>>> import fileinfo                    #(1)
>>> print '\n'.join(sys.modules.keys())
win32api
os.path
os
fileinfo
exceptions
__main__
ntpath
```

```
nt
sys
__builtin__
site
signal
UserDict
stat
>>> fileinfo
<module 'fileinfo' from 'fileinfo.pyc'>
>>> sys.modules["fileinfo"]            #(2)
<module 'fileinfo' from 'fileinfo.pyc'>
```

1. Podczas importowania nowych modułów, zostają one dodane do `sys.modules`. To tłumaczy dlaczego ponowne zaimportowanie tego samego modułu jest bardzo szybkie. Otóż Python aktualnie posiada wczytany i zapamiętany moduł w `sys.modules`, więc za drugim razem kiedy importujemy moduł, Python spogląda po prostu tylko do tego słownika.

2. Podając nazwę wcześniej zaimportowanego modułu (w postaci łańcucha znaków), możemy pobrać referencję do samego modułu poprzez bezpośrednie wykorzystanie słownika `sys.modules`.

Kolejny przykład pokazuje, jak wykorzystywać atrybut klasy `__module__` razem ze słownikiem `sys.modules`, aby pobrać referencję do modułu, w którym ta klasa jest zdefiniowana.

Przykład 6.14 Atrybut klasy `__module__`

```
>>> from fileinfo import MP3FileInfo
>>> MP3FileInfo.__module__              #(1)
'fileinfo'
>>> sys.modules[MP3FileInfo.__module__]  #(2)
<module 'fileinfo' from 'fileinfo.pyc'>
```

1. Każda klasa Pythona posiada wbudowany atrybut klasy, jakim jest `__module__`, a który przechowuje nazwę modułu, w którym dana klasa jest zdefiniowana.

2. Łącząc to z `sys.modules`, możemy pobrać referencję do modułu, w którym ta klasa jest zdefiniowana.

Teraz już jesteś przygotowany do tego, aby zobaczyć w jaki sposób `sys.modules` jest wykorzystywany w fileinfo.py, czyli przykładowym programie wykorzystanym w rozdziale 5. Poniższy przykład przedstawia fragment kodu.

Przykład 6.15 `sys.modules` w fileinfo.py

```
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):         #(1)
        u"zwraca klasę metadanych pliku na podstawie podanego rozszerzenia"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]          #(2)
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo   #(3)
```

1. Jest to funkcja z dwoma argumentami. Argument `filename` jest wymagany, ale `module` jest argumentem opcjonalnym i domyślnie wskazuje na moduł, który zawiera klasę `FileInfo`. Wygląda to nieefektywnie, ponieważ może się wydawać, że Python wykonuje wyrażenie `sys.modules` za każdym razem, gdy funkcja zostaje wywołana. Tak na prawdę, Python wykonuje domyślne wyrażenia tylko raz, podczas pierwszego zaimportowania modułu. Jak zobaczymy później, nigdy nie wywołamy tej funkcji z argumentem `module`, więc argument `module` służy nam raczej jako stała na poziomie tej funkcji.

2. Funkcji tej przyjrzymy się później, po tym, jak zanurkujemy w module `os`. Na razie zaufaj, że linia ta sprawia, że `subclass` przechowuje nazwę klasy np. `MP3FileInfo`.

3. Już wiemy, że funkcja `getattr` zwraca nam referencje do obiektu poprzez nazwę. `hasattr` jest funkcją uzupełniającą, która sprawdza, czy obiekt posiada określony atrybut. W tym przypadku sprawdzamy, czy moduł posiada określoną klasę (funkcja ta działa na dowolnym obiekcie i dowolnym atrybucie, podobnie jak `getattr`). Ten kod możemy na język polski przetłumaczyć w ten sposób: „Jeśli ten moduł posiada klasę o nazwie zawartej w zmiennej `subclass`, to ją zwróć, w przeciwnym wypadku zwróć klasę `FileInfo`".

Materiały dodatkowe

- Python Tutorial omawia dokładnie, kiedy i jak domyślne argumenty są wyznaczane
- Python Library Reference dokumentuje moduł `sys`

7.5 Praca z katalogami

Praca z katalogami

Moduł `os.path` zawiera kilka funkcji służących do manipulacji plikami i katalogami (w systemie Windows nazywanymi folderami). Przyjrzymy się teraz obsłudze ścieżek i odczytywaniu zawartości katalogów.

Przykład 6.16 Tworzenie ścieżek do plików

```
>>> import os
>>> os.path.join("c:\\music\\ap\\", "mahadeva.mp3")   #(1) (2)
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.join("c:\\music\\ap", "mahadeva.mp3")     #(3)
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.expanduser("~")                           #(4)
'c:\\Documents and Settings\\mpilgrim\\My Documents'
>>> os.path.join(os.path.expanduser("~"), "Python")   #(5)
'c:\\Documents and Settings\\mpilgrim\\My Documents\\Python'
```

1. `os.path` jest referencją do modułu, a ten moduł zależy od platformy z jakiej korzystamy. Tak jak getpass niweluje różnice między platformami ustawiając `getpass` na funkcję odpowiednią dla naszego systemu, tak `os` ustawia `path` na moduł specyficzny dla konkretnej platformy.

2. Funkcja `join` modułu `os.path` tworzy ścieżkę dostępu do pliku z jednej lub kilku ścieżek częściowych. W tym przypadku po prostu łączy dwa łańcuchy znaków. (Zauważmy, że w Windowsie musimy używać podwójnych ukośników.)

3. W tym, trochę bardziej skomplikowanym, przypadku, `join` dopisze dodatkowy ukośnik do ścieżki przed dołączeniem do niej nazwy pliku. Nie musimy pisać małej głupiej funkcji `addSlashIfNecessary`, ponieważ mądrzy ludzie zrobili już to za nas.

4. `expanduser` rozwinie w ścieżce znak na ścieżkę katalogu domowego aktualnie zalogowanego użytkownika. Ta funkcja działa w każdym systemie, w którym użytkownicy mają swoje katalogi domowe, między innymi w systemach Windows, UNIX i Mac OS X, ale w systemie Mac OS nie otrzymujemy żadnych efektów.

5. Używając tych technik, możemy łatwo tworzyć ścieżki do plików i katalogów wewnątrz katalogu domowego.

Przykład 6.17 Rozdzielanie ścieżek

```
>>> os.path.split("c:\\music\\ap\\mahadeva.mp3")                        #(1)
('c:\\music\\ap', 'mahadeva.mp3')
>>> (filepath, filename) = os.path.split("c:\\music\\ap\\mahadeva.mp3") #(2)
>>> filepath                                                            #(3)
'c:\\music\\ap'
>>> filename                                                            #(4)
'mahadeva.mp3'
>>> (shortname, extension) = os.path.splitext(filename)                 #(5)
```

```
>>> shortname
'mahadeva'
>>> extension
'.mp3'
```

1. Funkcja `split` dzieli pełną ścieżkę i zwraca listę, która zawiera ścieżkę do katalogu i nazwę pliku. Pamiętasz, jak mówiliśmy, że można używać wielozmiennego przypisania do zwracania kilku wartości z funkcji? `split` jest taką właśnie funkcją.
2. Przypisujesz wynik działania funkcji `split` do krotki dwóch zmiennych. Każda zmienna będzie teraz zawierać wartość odpowiedniego elementu krotki zwróconej przez funkcję `split`.
3. Pierwsza zmienna, `filepath`, zawiera pierwszy element zwróconej listy – ścieżkę pliku.
4. Druga zmienna, `filename`, zawiera drugi element listy – nazwę pliku.
5. Moduł `os.path` zawiera też funkcję `splitext`, która zwraca krotkę zawierającą właściwą nazwę pliku i jego rozszerzenie. Używamy tej samej techniki, co poprzednio, do przypisania każdej części do osobnej zmiennej.

Przykład 6.18 Wyświetlanie zawartości katalogu

```
>>> os.listdir("c:\\music\\_singles\\")                #(1)
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
>>> dirname = "c:\\"
>>> os.listdir(dirname)                                #(2)
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'cygwin',
'docbook', 'Documents and Settings', 'Incoming', 'Inetpub', 'IO.SYS',
'MSDOS.SYS', 'Music', 'NTDETECT.COM', 'ntldr', 'pagefile.sys',
'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
>>> [f for f in os.listdir(dirname)
...     if os.path.isfile(os.path.join(dirname, f))]   #(3)
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'IO.SYS', 'MSDOS.SYS',
'NTDETECT.COM', 'ntldr', 'pagefile.sys']
>>> [f for f in os.listdir(dirname)
...     if os.path.isdir(os.path.join(dirname, f))]    #(4)
['cygwin', 'docbook', 'Documents and Settings', 'Incoming',
'Inetpub', 'Music', 'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
```

1. Funkcja `listdir` pobiera ścieżkę do katalogu i zwraca listę jego zawartości.
2. `listdir` zwraca zarówno pliki jak i katalogi, bez wskazania które są którymi.
3. Możemy użyć filtrowania listy i funkcji `isfile` modułu `os.path`, aby oddzielić pliki od katalogów. `isfile` przyjmuje ścieżkę do pliku i zwraca `True`, jeśli reprezentuje ona plik albo `False` w innym przypadku. W przykładzie używamy

os.path.join, aby uzyskać pełną ścieżkę, ale isfile pracuje też ze ścieżkami względnymi wobec bieżącego katalogu. Możemy użyć os.getcwd() aby pobrać bieżący katalog.

4. os.path zawiera też funkcję isdir, która zwraca True, jeśli ścieżka reprezentuje katalog i False w innym przypadku. Możemy jej użyć do uzyskania listy podkatalogów.

Przykład 6.19 Listowanie zawartości katalogu w fileinfo.py

```
def listDirectory(directory, fileExtList):
    u"zwraca listę obiektów zawierających metadane dla plików o podanych rozszerzeniach"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]    #(1) (2)
    fileList = [os.path.join(directory, f) for f in fileList
                if os.path.splitext(f)[1] in fileExtList]              #(3) (4) (5)
```

1. os.listdir(directory) zwraca listę wszystkich plików i podkatalogów w katalogu directory.

2. Iterując po liście z użyciem zmiennej f, wykorzystujemy os.path.normcase(f), aby znormalizować wielkość liter zgodnie z domyślną wielkością liter w systemem operacyjnym. Funkcja normcase jest użyteczną, prostą funkcją, która stanowi równoważnik pomiędzy systemami operacyjnymi, w których wielkość liter w nazwie pliku nie ma znaczenia, w którym np. mahadeva.mp3 i mahadeva.MP3 są takimi samymi plikami. Na przykład w Windowsie i Mac OS, normcase będzie konwertował całą nazwę pliku na małe litery, a w systemach kompatybilnych z UNIX-em funkcja ta będzie zwracała niezmienioną nazwę pliku.

3. Iterując ponownie po liście z użyciem f, wykorzystujemy os.path.splitext(f), aby podzielić nazwę pliku na nazwę i jej rozszerzenie.

4. Dla każdego pliku sprawdzamy, czy rozszerzenie jest w liście plików, o które nam chodzi (czyli fileExtList, która została przekazana do listDirectory).

5. Dla każdego pliku, który nas interesuje, wykorzystujemy os.path.join(directory, f), aby skonstruować pełną ścieżkę pliku i zwrócić listę zawierającą pełne ścieżki.

Jeśli to możliwe, powinniśmy korzystać z funkcji w modułach os i os.path do manipulacji plikami, katalogami i ścieżkami. Te moduły opakowują moduły specyficzne dla konkretnego systemu, więc funkcje takie, jak os.path.split pooprawnie działają w systemach UNIX, Windows, Mac OS i we wszystkich innych systemach wspieranych przez Pythona.

Jest jeszcze inna metoda dostania się do zawartości katalogu. Metoda ta jest bardzo potężna i używa zestawu symboli wieloznacznych (ang. *wildcard*), z którymi można się spotkać pracując w linii poleceń.

Przykład 6.20 Listowanie zawartości katalogu przy pomocy glob

```
>>> os.listdir("c:\\music\\_singles\\")              #(1)
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
>>> import glob
>>> glob.glob('c:\\music\\_singles\\*.mp3')          #(2)
['c:\\music\\_singles\\a_time_long_forgotten_con.mp3',
'c:\\music\\_singles\\hellraiser.mp3',
'c:\\music\\_singles\\kairo.mp3',
'c:\\music\\_singles\\long_way_home1.mp3',
'c:\\music\\_singles\\sidewinder.mp3',
'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\_singles\\s*.mp3')         #(3)
['c:\\music\\_singles\\sidewinder.mp3',
'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\*\\*.mp3')                 #(4)
```

1. Jak wcześniej powiedzieliśmy, `os.listdir` pobiera ścieżkę do katalogu i zwraca wszystkie pliki i podkatalogi, które się w nim znajdują.

2. Z drugiej strony, moduł `glob` na podstawie podanego wyrażenia składającego się z symboli wieloznacznych, zwraca pełne ścieżki wszystkich plików, które spełniają te wyrażenie. Tutaj wyrażenie jest ścieżką do katalogu plus `"*.mp3"`, który będzie dopasowywał wszystkie pliki `.mp3`. Dodajmy, że każdy element zwracanej listy jest już pełną ścieżką do pliku.

3. Jeśli chcemy znaleźć wszystkie pliki w określonym katalogu, gdzie nazwa zaczyna się od `s"`, a kończy na `".mp3"`, możemy to zrobić w ten sposób.

4. Teraz rozważ taki scenariusz: mamy katalog z muzyką z kilkoma podkatalogami, wewnątrz których są pliki `.mp3`. Możemy pobrać listę wszystkich tych plików za pomocą jednego wywołania `glob`, wykorzystując połączenie dwóch wyrażeń. Pierwszym jest `"*.mp3"` (wyszukuje pliki .mp3), a drugim są same w sobie ścieżki do katalogów, aby przetworzyć każdy podkatalog w `c:\music`. Ta prosto wyglądająca funkcja daje nam niesamowite możliwości!

Materiały dodatkowe

- Python Knowledge Base odpowiada na najczęściej zadawane pytanie na temat modułu `os`
- Python Library Reference dokumentuje moduł `os` i moduł `os.path`

7.6 Wyjątki i operacje na plikach - wszystko razem

Wszystko razem

Jeszcze raz ułożymy wszystkie puzzle domina w jednym miejscu. Już poznaliśmy, w jaki sposób działa każda linia kodu. Powrócimy do tego jeszcze raz i zobaczymy, jak to wszystko jest ze sobą dopasowane.

Przykład 6.21 `listDirectory`

```
def listDirectory(directory, fileExtList):                #(1)
    u"zwraca listę obiektów zawierających metadane dla plików o podanych rozszerzeniach"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f) for f in fileList
                if os.path.splitext(f)[1] in fileExtList]                      #(2)
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):   #(3)
        u"zwraca klasę metadanych pliku na podstawie podanego rozszerzenia"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]    #(4)
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo  #(5)
    return [getFileInfoClass(f)(f) for f in fileList]                           #(6)
```

1. `listDirectory` jest główną atrakcją tego modułu. Przyjmuje ona na wejściu katalog (np. `c:\music_singles\`) i listę interesujących nas rozszerzeń plików (jak np. `['.mp3']`), a następnie zwraca listę instancji klas, które są podklasami słownika, a przechowują metadane na temat każdego interesującego nas pliku w tym katalogu. I to wszystko jest wykonywane za pomocą kilku prostych linii kodu.

2. Jak dowiedzieliśmy się w poprzednim podrozdziale, ta linia kodu zwraca listę pełnych ścieżek wszystkich plików z danego katalogu, które mają interesujące nas rozszerzenie (podane w argumencie `fileExtList`).

3. Starzy programiści Pascala znają zagnieżdżone funkcje (funkcje wewnątrz funkcji), ale większość ludzi jest zdziwionych, gdy mówi się im, że Python je wspiera. Zagnieżdżona funkcja `getFileInfoClass` może być wywołana tylko z funkcji, w której jest zadeklarowana, czyli z `listDirectory`. Jak w przypadku każdej innej funkcji, nie musimy przejmować się deklaracją interfejsu, ani niczym innym. Po prostu definiujemy funkcję i implementujemy ją.

4. Teraz, gdy już znamy moduł `os`, ta linia powinna nabrać sensu. Pobiera ona rozszerzenie pliku (`os.path.splitext(filename)[1]`), przekształca je do dużych liter (`.upper()`), odcina kropkę (`[1:]`) i tworzy nazwę klasy używając łańcucha formatującego. `c:\music\ap\mahadeva.mp3` zostaje przekształcone na `.mp3`, potem na `.MP3`, a następnie na `MP3` i na końcu otrzymujemy `MP3FileInfo`.

5. Mając nazwę klasy obsługującej ten plik, sprawdzamy czy tak klasa istnieje w tym module. Jeśli tak, zwracamy tę klasę, jeśli nie – klasę bazową `FileInfo`. To bardzo ważne: zwracamy klasę. Nie zwracamy obiektu, ale klasę samą w sobie.

6. Dla każdego pliku z listy `fileList` wywołujemy `getFileInfoClass` z nazwą pli"ku `(f)`. Wywołanie `getFileInfoClass(f)` zwraca klasę. Dokładnie nie wiadomo jaką, ale to nam nie przeszkadza. Potem tworzymy obiekt tej klasy (jaka by

ona nie była) i przekazujemy nazwę pliku (znów `f`) do jej metody `__init__`. Jak pamiętamy z wcześniejszych rozdziałów, metoda `__init__` klasy `FileInfo` ustawia wartość `self[ńame"]`, co powoduje wywołanie `__setitem__` klasy pochodnej, czyli `MP3FileInfo`, żeby odpowiednio przetworzyć plik i wyciągnąć jego metadane. Robimy to wszystko dla każdego interesującego pliku i zwracamy listę obiektów wynikowych.

Zauważmy, że metoda `listDirectory` jest bardzo ogólna. Nie wie w żaden sposób, z jakimi typami plików będzie pracować, ani jakie klasy są zdefiniowane do obsługi tych plików. Zaczyna pracę od przejrzenia katalogu, w poszukiwaniu plików do przetwarzania, a potem analizuje swój moduł, żeby sprawdzić, jakie klasy obsługi (np. `MP3FileInfo`) są zdefiniowane. Możemy rozszerzyć ten program, żeby obsługiwać inne typy plików definiując klasy o odpowiednich nazwach: `HTMLFileInfo` dla plików HTML, `DOCFileInfo` dla plików Worda itp. `listDirectory`, bez potrzeby modyfikacji kodu tej funkcji, obsłuży je wszystkie, zrzucając całe przetwarzanie na odpowiednie klasy i zbierając otrzymane wyniki.

7.7 Wyjątki i operacje na plikach - podsumowanie

Podsumowanie

Program fileinfo.py wprowadzony w rozdziale 5 powinien już być zrozumiały.

```
u"""Framework do pobierania matedanych specyficznych dla danego typu pliku.

Można utworzyć instancję odpowiedniej klasy podając jej nazwę pliku w konstruktorze.
Zwrócony obiekt zachowuje się jak słownik posiadający parę klucz-wartość
dla każdego fragmentu metadanych.

    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Lub użyć funkcji listDirectory, aby pobrać informacje o wszystkich plikach w katalogu.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework może być roszerzony poprzez dodanie klas dla poszczególnych typów plików, np.:
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Każda klasa jest całkowicie odpowiedzialna
za właściwe sparsowanie swojego pliku; zobacz przykład MP3FileInfo.
"""

import os
import sys

def stripnulls(data):
    u"usuwa białe znaki i nulle"
    return data.replace("\00", " ").strip()

class FileInfo(dict):
    u"przechowuje metadane pliku"
    def __init__(self, filename=None):
        dict.__init__(self)
        self["plik"] = filename

class MP3FileInfo(FileInfo):
    u"przechowuje znaczniki ID3v1.0 MP3"
    tagDataMap = \{u"tytuł"    : (  3,  33, stripnulls),
                  "artysta"   : ( 33,  63, stripnulls),
                  "album"     : ( 63,  93, stripnulls),
                  "rok"       : ( 93,  97, stripnulls),
                  "komentarz" : ( 97, 126, stripnulls),
                  "gatunek"   : (127, 128, ord)\}

    def __parse(self, filename):
        u"parsuje znaczniki ID3v1.0 z pliku MP3"
        self.clear()
        try:
```

```
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == 'TAG':
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "plik" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    u"zwraca listę obiektów zawierających metadane dla plików o podanych rozszerzeniach"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        u"zwraca klasę metadanych pliku na podstawie podanego rozszerzenia"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]):
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

Zanim zanurkujemy w następnym rozdziale, upewnijmy się, że nie mamy problemów z:

- przechwytywaniem wyjątków za pomocą `try...except`
- chronieniem zewnętrznych zasobów za pomocą `try...finally`
- czytaniem plików
- korzystaniem z wielozmiennych przypisań w pętli `for`
- Wykorzystywaniem modułu `os` do niezależnego od platformy zarządzania plikami
- Dynamicznym tworzeniem instancji klas nieznanych typów poprzez traktowanie klas jak obiektów

Rozdział 8

Wyrażenia regularne

8.1 Wyrażenia regularne

Wyrażenia regularne są bardzo użytecznymi, a zarazem standardowymi środkami wyszukiwania, zamiany i przetwarzania tekstu wykorzystując skomplikowane wzorce. Jeśli wykorzystywaliśmy już wyrażenia regularne w innych językach (np. w Perlu), to pewnie zauważymy, że składnia jest bardzo podobna, ponadto możemy przeczytać podsumowanie modułu `re`, aby przeglądnąć dostępne funkcje i ich argumenty.

Nurkujemy

Łańcuchy znaków mają metody, które służą wyszukiwaniu (`index`, `find` i `count`), zmienianiu (`replace`) i przetwarzaniu (`split`), ale są one ograniczone do najprostszych przypadków. Metody te wyszukują pojedynczy, zapisany na stałe ciąg znaków i zawsze uwzględniają wielkość liter. Aby wyszukać coś niezależnie od wielkości liter w łańcuchu `s`, musimy użyć `s.lower()` lub `s.upper()` i upewnić się, że nasz tekst do wyszukania ma odpowiednią wielkość liter. Metody służące do zamiany i podziału mają takie same ograniczenia.

Jeśli to, co próbujemy zrobić jest możliwe przy użyciu metod łańcucha znaków, powinniśmy ich użyć. Są szybkie, proste i czytelne. Jeśli jednak okazuje się, że używamy wielu różnych metod i instrukcji `if` do obsługi przypadków szczególnych albo jeśli łączysz je z użyciem `split`, `join` i wyrażeń listowych w dziwny i nieczytelny sposób, możemy być zmuszeni do przejścia na wyrażenia regularne.

Pomimo, że składnia wyrażeń regularnych jest zwarta i niepodobna do normalnego kodu, wynik może być czytelniejszy niż użycie długiego ciągu metod łańcucha znaków. Są nawet sposoby dodawania komentarzy wewnątrz wyrażeń regularnych, aby były one praktycznie samodokumentujące się.

8.2 Analiza przypadku: Adresy ulic

Analiza przypadku: Adresy ulic

Ta seria przykładów została zainspirowana problemami z prawdziwego życia. Kilka lat temu gdzieś nie w Polsce, zaszła potrzeba oczyszczenia i ustandaryzowania adresów ulic zaimportowanych ze starego systemu, zanim zostaną zaimportowane do nowego. (Zauważmy, że nie jest to jakiś wyimaginowany przykład. Może on się okazać przydatny.) Poniższy przykład przybliży nas do tego problemu.

Przykład 7.1 Dopasowywanie na końcu napisu

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')              #(1)
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')              #(2)
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.') #(3)
'100 NORTH BROAD RD.'
>>> import re                             #(4)
>>> re.sub('ROAD$', 'RD.', s)             #(5) (6)
'100 NORTH BROAD RD.'
```

1. Naszym celem jest ustandaryzowanie adresów ulic, więc skrótem od `'ROAD'` jest `'RD.'`. Na pierwszy rzut oka wydaje się, że po prostu można wykorzystać metodę łańcucha znaków, jaką jest `replace`. Zakładamy, że wszystkie dane zapisane są za pomocą wielkich liter, więc nie powinno być problemów wynikających z niedopasowania, ze względu na wielkość liter. Wyszukujemy stały napis, jakim jest `'ROAD'`. Jest to bardzo płytki przykład, więc `s.replace` poprawnie zadziała.

2. Życie niestety jest trochę bardziej skomplikowane, o czym dość szybko można się przekonać. Problem w tym przypadku polega na tym, że `'ROAD'` występuje w adresie dwukrotnie: raz jako część nazwy ulicy (`'BROAD'`) i drugi raz jako oddzielne słowo. Metoda `replace` znajduje te dwa wystąpienia i ślepo je zamienia, niszcząc adres.

3. Aby rozwiązać problem z adresami, gdzie podciąg `'ROAD'` występuje kilka razy, możemy wykorzystać taki pomysł: tylko szukamy i zamieniamy `'ROAD'` w ostatnich czterech znakach adresu (czyli `s[-4:]`), a zostawiamy pozostałą część (czyli s[:-4]). Jednak, jak możemy zresztą zobaczyć, takie rozwiązanie jest dosyć niewygodne. Na przykład polecenie, które chcemy wykorzystać, zależy od długość zamienianego napisu (jeśli chcemy zamienić `'STREET'` na `'ST.'`, wykorzystamy `s[:-6]` i `s[-6:].replace(...)`). Chciałoby się do tego wrócić za sześć miesięcy i to debugować? Pewnie nie.

4. Nadszedł odpowiedni czas, aby przejść do wyrażeń regularnych. W Pythonie cała funkcjonalność wyrażeń regularnych zawarta jest w module `re`.

5. Spójrzmy na pierwszy parametr, `'ROAD$'`. Jest to proste wyrażenie regularne, które dopasuje `'ROAD'` tylko wtedy, gdy wystąpi on na końcu tekstu. Znak `$` znaczy "koniec napisu". (Mamy także analogiczny znak, znak daszka `^`, który znaczy "początek napisu".)

6. Korzystając z funkcji `re.sub`, przeszukujemy napis `s` i podciąg pasujący do wyrażenia regularnego `'ROAD$'` zamieniamy na `'RD.'`. Dzięki temu wyrażeniu dopasowujemy `'ROAD'` na końcu napisu `s`, lecz napis `'ROAD'` nie zostanie dopasowany w słowie `'BROAD'`, ponieważ znajduje się on w środku napisu `s`.

Wracając do historii z porządkowaniem adresów, okazało się, że poprzedni przykład, dopasowujący `'ROAD'` na końcu adresu, nie był poprawny, ponieważ nie wszystkie adresy dołączały `'ROAD'` na końcu adresu. Niektóre adresy kończyły się tylko nazwą ulicy. Wiele razy to wyrażenie zadziałało poprawnie, jednak gdy mieliśmy do czynienia z ulicą `'BROAD'`, wówczas wyrażenie regularne dopasowywało `'ROAD'` na końcu napisu jako część słowa `'BROAD'`, a takiego wyniku nie oczekiwaliśmy.

Przykład 7.2 Dopasowywanie całych wyrazów

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)  #(1)
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)  #(2)
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)  #(3)
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s) #(4)
'100 BROAD RD. APT 3'
```

1. W istocie chcieliśmy odnaleźć `'ROAD'` znajdujące się na końcu napisu i jest samodzielnym słowem, a nie częścią dłuższego wyrazu. By opisać coś takiego za pomocą wyrażeń regularnych korzystamy z `\b`, które znaczy tyle co "tutaj musi znajdować się początek lub koniec wyrazu". W Pythonie jest to nieco skomplikowane przez fakt, iż znaki specjalne (takie jak np. \) muszą być poprzedzone właśnie znakiem \. Zjawisko to określane jest czasem plagą ukośników (ang. *backslash plague*) i wydaje się być jednym z powodów łatwiejszego korzystania z wyrażeń regularnych w Perlu niż w Pythonie. Z drugiej jednak strony, w Perlu składnia wyrażeń regularnych wymieszana jest ze składnią języka, co utrudnia stwierdzenie czy błąd tkwi w naszym wyrażeniu regularnym, czy w błędnym użyciu składni języka.

2. Eleganckim obejściem problemu plagi ukośników jest wykorzystywanie tzw. surowych napisów (ang. *raw string*), które opisywaliśmy w rozdziale 3, poprzez umieszczanie przed napisami litery `r`. Python jest w ten sposób informowany o tym, iż żaden ze znaków specjalnych w tym napisie nie ma być interpretowany; `'\t'` odpowiada znakowi tab, jednak `r'\t'` oznacza tyle, co litera `t` poprzedzona znakiem \. Przy wykorzystaniu wyrażeń regularnych zalecane jest stosowanie surowych napisów; w innym wypadku wyrażenia szybko stają się niezwykle skomplikowane (a przecież już ze swej natury nie są proste).

3. Cóż... Niestety wkrótce okazuje się, iż istnieje więcej przypadków przeczących logice naszego postępowania. W tym przypadku `'ROAD'` było samodzielnym słowem, jednak znajdowało się w środku napisu, ponieważ na jego końcu umieszczony był jeszcze numer mieszkania. Z tego powodu nasze bieżące wyrażenie nie

zostało odnalezione, funkcja `re.sub` niczego nie zamieniła, a co za tym idzie napis został zwrócony w pierwotnej postaci (co nie było naszym celem).

4. Aby rozwiązać ten problem wystarczy zamienić w wyrażeniu regularnym `$` na kolejne `\b`. Teraz będzie ono pasować do każdego samodzielnego słowa `'ROAD'`, niezależnie od jego pozycji w napisie.

8.3 Analiza przypadku: Liczby rzymskie

Analiza przypadku: Liczby rzymskie

Najprawdopodobniej spotkaliśmy się już gdzieś z liczbami rzymskimi. Można je spotkać w starszych filmach oglądanych w telewizji (np. "Copyright MCMXLVI" zamiast "Copyright 1946") lub na ścianach bibliotek, czy uniwersytetów (napisy typu "założone w MDCCCLXXXVIII" zamiast "założone w 1888 roku"). Mogliśmy je także zobaczyć na przykład w referencjach bibliograficznych. Ten system reprezentowania liczb sięga czasów starożytnego Rzymu (stąd nazwa).

W liczbach rzymskich wykorzystuje się siedem znaków, które na różne sposoby się powtarza i łączy, aby zapisać pewną liczbę:

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

Poniżej znajdują się podstawowe zasady konstruowania liczb rzymskich:

- Znaki są addytywne. I to 1, II to 2, a III to 3. VI to 6 (dosłownie, „5 i 1"), VII to 7, a VIII to 8.
- Znaki dziesiątek (I, X, C i M) mogą się powtarzać do trzech razy. Za czwartym należy odjąć od następnego większego znaku piątek. Nie można zapisać liczby 4 jako IIII. Zamiast tego napiszemy IV ("o 1 mniej niż 5"). Liczba 40 zapisujemy jako XL (o 10 mniej niż 50), 41 jako XLI, 42 jako XLII, 43 jako XLIII, a potem 44 jako XLIV (o 10 mniej niż 50, a potem o 1 mniej niż 5).
- Podobnie w przypadku 9. Musimy odejmować od wyższego znaku dziesiątek: 8 to VIII, lecz 9 zapiszemy jako IX (o 1 mniej niż 10), a nie jako VIIII (ponieważ znak nie może się powtarzać cztery razy). Liczba 90 to XC, a 900 zapiszemy jako CM.
- Znaki piątek nie mogą się powtarzać. Liczba 10 jest zawsze reprezentowana przez X, nigdy przez VV. Liczba 100 to zawsze C, nigdy LL.
- Liczby rzymskie są zawsze pisane od najwyższych do najniższych i czytane od lewej do prawej, więc porządek znaków jest bardzo ważny. DC to 600, jednak CD jest kompletnie inną liczbą (400, ponieważ o 100 mniej niż 500). CI to 101, jednak IC nie jest żadną poprawną liczbą rzymską (nie możemy bezpośrednio odejmować 1 od 100, musimy to zapisać jako XCIX, o 10 mniej niż 100, dodać 1 mniej niż 10).

Sprawdzamy tysiące

Jak sprawdzić, czy jakiś łańcuch znaków jest liczbą rzymską? Spróbujmy sprawdzać cyfra po cyfrze. Jako, że liczby rzymskie są zapisywane zawsze od najwyższych do najniższych, zacznijmy od tych najwyższych: tysięcy. Dla liczby 1000 i większych, tysiące zapisywane są przez serię liter M.

Przykład 7.3 Sprawdzamy tysiące

```
>>> import re
>>> pattern = '^M?M?M?$'          #(1)
>>> re.search(pattern, 'M')       #(2)
<SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')      #(3)
<SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')     #(4)
<SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM')    #(5)
>>> re.search(pattern, "")        #(6)
<SRE_Match object at 0106F4A8>
```

1. Ten wzorzec składa się z 3 części:
   - `^`, które umieszczone jest w celu dopasowania jedynie początku łańcucha. Gdybyśmy go nie umieścili, wzorzec pasowałby do każdego wystąpienia znaków `M`, czego przecież nie chcemy. Chcemy, aby dopasowane zostały jedynie znaki `M` znajdujące się na początku łańcucha, o ile w ogóle istnieją.
   - `M?`, które ma wykryć, czy istnieje pojedyncza litera `M`. Jako, że powtarzamy to trzykrotnie, dopasujemy od zera do trzech liter `M` w szeregu.
   - `$`, w celu dopasowania wzorca do końca łańcucha. Gdy połączymy to ze znakiem `^` na początku, otrzymamy wzorzec, który musi pasować do całego łańcucha, bez żadnych znaków przed czy po serii znaków `M`.
2. Sednem modułu `re` jest funkcja `search`, która jako argumenty przyjmuje wyrażenie regularne (wzorzec) i łańcuch znaków (`'M'`), a następnie próbuje go dopasować do wzorca. Gdy zostanie dopasowany, `search` zwraca obiekt który posiada wiele metod, które opisują dopasowanie. Jeśli nie uda się dopasować, `search` zwraca `None`, co jest Pythonową pustą wartością i nic nie oznacza. Na razie jedyne, co nas interesuje, to czy wzorzec został dopasowany, czy nie, a co możemy stwierdzić przez sprawdzenie, co zwróciła funkcja `search`. `'M'` pasuje do wzorca, gdyż pierwsze opcjonalne `M` zostało dopasowane, a drugie i trzecie zostało zignorowane.
3. `'MM'` pasuje, gdyż pierwsze i drugie opcjonalne `M` zostało dopasowane, a trzecie zignorowane.
4. `'MMM'` również pasuje do wzorca, gdyż wszystkie trzy opcjonalne wystąpienia `M` we wzorcu zostały dopasowane.
5. `'MMMM'` nie pasuje, gdyż pomimo dopasowania pierwszych trzech opcjonalnych znaków `M`, za trzecim wzorzec wymaga, aby łańcuch się skończył, a w naszym łańcuchu znaków znajduje się kolejna litera `M`. Tak więc `search` zwraca wartość `None`.

6. Co ciekawe, pusty łańcuch też pasuje do wzorca, gdyż wszystkie wystąpienia `M` są opcjonalne.

Sprawdzamy setki

Setki są nieco trudniejsze, ponieważ schemat zapisu nie jest aż tak prosty jak w wypadku tysięcy. Mamy więc następujące możliwości:

- 100 = C
- 200 = CC
- 300 = CCC
- 400 = CD
- 500 = D
- 600 = DC
- 700 = DCC
- 800 = DCCC
- 900 = CM

Wynika z tego, że mamy 4 wzorce:

- CM
- CD
- Zero do trzech wystąpień C (zero, gdyż może nie być żadnej setki)
- D, po którym następuje zero do trzech C

Ostatnie dwa wzorce możemy połączyć w opcjonalne D, a za nim od zera do trzech C.

Poniższy przykład ilustruje jak sprawdzać setki w liczbach Rzymskich.

Przykład 7.4 Sprawdzamy setki

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)\$'  #(1)
>>> re.search(pattern, 'MCM')             #(2)
<SRE_Match object at 01070390>
>>> re.search(pattern, 'MD')              #(3)
<SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC')          #(4)
<SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC')            #(5)
>>> re.search(pattern, "")                #(6)
<SRE_Match object at 01071D98>
```

1. Ten wzorzec zaczyna się tak samo jak poprzedni, rozpoczynając sprawdzanie od początku łańcucha (`^`), potem sprawdzając tysiące (`M?M?M?`). Tutaj zaczyna się nowa część, która definiuje 3 alternatywne wzorce rozdzielone pionową kreską (`|`): `CM`, `CD`, i `D?C?C?C?` (opcjonalne D, po którym następuje od zera do trzech opcjonalnych znaków C). Analizator wyrażeń regularnych sprawdza każdy ze wzorców w kolejności od lewej do prawej, wybiera pierwszy pasujący i ignoruje resztę.

2. `'MCM'` pasuje, gdyż pierwsza litera `M` pasuje, drugie i trzecie `M` jest ignorowane, i `CM` pasuje (gdyż `CD` oraz `D?C?C?C?` nie są nawet sprawdzane). MCM to rzymska liczba 1900.

3. `'MD'` pasuje, ponieważ pierwsze `M` pasuje, drugie i trzecie `M` z wzorca jest ignorowane, oraz `D` pasuje do wzorca `D?C?C?C?` (wystąpienia znaku `C` jest opcjonalne, więc analizator je ignoruje). MD to rzymska liczba 1500.

4. `'MMMCCC'` pasuje, gdyż pasują wszystkie trzy pierwsze znaki `M`, a fragment `D?C?C?C?` we wzorcu pasuje do `CCC` (`D` jest opcjonalne). MMMCCC to 3300.

5. `'MCMC'` nie pasuje, Pierwsze `M` pasuje, `CM` również, ale `$` już nie, gdyż nasz łańcuch zamiast się skończyć, ma kolejną literę `C`. Nie została ona dopasowana do wzorca `D?C?C?C?`, gdyż został on wykluczony przez wystąpienie wzorca `CM`.

6. Co ciekawe, pusty łańcuch znaków dalej pasuje do naszego wzorca, gdyż wszystkie znaki `M` są opcjonalne, tak jak każdy ze znaków we wzorcu `D?C?C?C?`.

Uff! Widzimy, jak szybko wyrażenia regularne stają się brzydkie? A jak na razie wprowadziliśmy do niego tylko tysiące i setki. Ale jeśli dokładnie śledziliśmy cały ten rozdział, dziesiątki i jednostki nie powinny stanowić dla Ciebie problemu, ponieważ wzór jest identyczny. A teraz zajmiemy się inną metodą wyrażania wzorca.

8.4 Składnia ?n, m?

Składnia {n, m}

W poprzednim podrozdziale poznaliśmy wzorce, w których ten sam znak mógł się powtarzać co najwyżej trzy razy. Tutaj przedstawimy inny sposób zapisania takiego wyrażenia, a który wydaje się być bardziej czytelny. Najpierw spójrzmy na metody wykorzystane w poprzednim przykładzie.

Przykład 7.5 Stary sposób: każdy znak opcjonalny

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')   #(1)
<_sre.SRE_Match object at 0x008EE090>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MM')   #(2)
<_sre.SRE_Match object at 0x008EEB48>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MMM')  #(3)
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM') #(4)
```

1. Instrukcja ta dopasowuje początek napisu, a następnie pierwszą literę `M`, lecz nie dopasowuje drugiego i trzeciego `M` (wszystko jest w porządku, ponieważ są one opcjonalne), a następnie koniec napisu.
2. Tutaj zostaje dopasowany początek napisu, a następnie pierwsze i drugie opcjonalne `M`, jednak nie zostaje dopasowane trzecie `M` (ale wszystko jest w ok, ponieważ jest to opcjonalne), ale zostaje dopasowany koniec napisu.
3. Zostanie dopasowany początek napisu, a następnie wszystkie opcjonalne `M`, a potem koniec napisu.
4. Dopasowany zostanie początek napisu, następnie wszystkie opcjonalne `M`, jednak koniec tekstu nie zostanie dopasowany, ponieważ pozostanie jedno niedopasowane `M`, dlatego też nic nie zostanie dopasowane, a operacja zwróci `None`.

Przykład 7.6 Nowy sposób: od `n` do `m`

```
>>> pattern = '^M{0,3}$'       #(1)
>>> re.search(pattern, 'M')    #(2)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM')   #(3)
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM')  #(4)
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM') #(5)
```

1. Ten wzorzec mówi: "dopasuj początek napisu, potem od zera do trzech znaków `M`, a następnie koniec napisu". `0` i `3` może być dowolną liczbą. Jeśli chcielibyśmy dopasować co najmniej jeden, lecz nie więcej niż trzy znaki `M`, powinienniśmy napisać `M{1,3}`.

2. Dopasowujemy początek napisu, potem jeden znak `M` z trzech możliwych, a następnie koniec napisu.

3. Tutaj zostaje dopasowany początek napisu, następnie dwa `M` z trzech możliwych, a następnie koniec napisu.

4. Zostanie dopasowany początek napisu, potem trzy znaki `M` z trzech możliwych, a następnie koniec napisu.

5. W tym miejscu dopasowujemy początek napisu, potem trzy znaki `M` z pośród trzech możliwych, lecz nie dopasujemy końca napisu. To wyrażenie regularne pozwala wykorzystać tylko trzy litery `M`, zanim dojdzie do końca napisu, a my mamy cztery, więc ten wzorzec niczego nie dopasuje i zwróci `None`.

Nie mamy programowalnej możliwości określenia, czy dwa wyrażenia są równoważne. Najlepszym sposobem, aby to zrobić, jest wykonanie wielu testów w celu przekonania się, czy otrzymujemy takie same wyniki. Więcej na temat pisania testów dowiemy się w dalszej części tej książki.

Sprawdzanie dziesiątek i jedności

Teraz rozszerzmy wyrażenie wykrywające liczby rzymskie, żeby odnajdywało też dziesiątki i jedności. Ten przykład pokazuje sprawdzanie dziesiątek.

Przykład 7.7 Sprawdzanie dziesiątek

```
>>> pattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL')    #(1)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML')     #(2)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX')    #(3)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX')  #(4)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX') #(5)
```

1. To dopasuje początek łańcucha, potem pierwsze opcjonalne `M`, dalej `CM` i `XL`, a potem koniec łańcucha. Zapamiętajmy, że składnia `(A|B|C)` oznacza "dopasuj dokładnie jedno z A, B lub C". W tym wypadku dopasowaliśmy `XL`, więc ignorujemy `XC` i `L?X?X?X?` i przechodzimy do końca łańcucha. MCMXL to 1940.

2. Tutaj dopasowujemy początek łańcucha, pierwsze opcjonalne `M`, potem `CM` i `L?X?X?X?`. Z tego ostatniego elementu dopasowane zostaje tylko `L`, a opcjonalne `X` zostają pominięte. Potem przechodzimy na koniec łańcucha. MCML to 1950.

3. Dopasowuje początek napisu, potem pierwsze opcjonalne `M`, następnie `CM`, potem opcjonalne `L` i pierwsze opcjonalne `X`, pomijając drugie i trzecie opcjonalne `X`, a następnie dopasowuje koniec napisu. MCMLX jest rzymską reprezentacją liczby 1960.

4. Tutaj zostanie dopasowany początek napisu, następnie pierwsze opcjonalne `M`, potem `CM`, następnie opcjonalne `L`, wszystkie trzy opcjonalne znaki `X` i w końcu dopasowany zostanie koniec napisu. MCMLXXX jest rzymską reprezentacją liczby 1980.

5. To dopasuje początek napisu, następnie pierwsze opcjonalne `M`, potem `CM`, opcjonalne `L`, wszystkie trzy opcjonalne znaki `X`, jednak nie może dopasować końca napisu, ponieważ pozostał jeszcze jeden niewliczony znak `X`. Zatem cały wzorzec nie zostanie dopasowany, więc zostanie zwrócone `None`. MCMLXXXX nie jest poprawną liczbą rzymską.

Poniżej przedstawiono podobne wyrażenie, które dodatkowo sprawdza jedności. Oszczędzimy sobie szczegółów.

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

Jak będzie wyglądało to wyrażenie wykorzystując składnię `{n,m}`? Zobaczmy na poniższy przykład.

Przykład 7.8 Sprawdzanie liczb rzymskich korzystając ze składni `{n, m}`

```
>>> pattern = '^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV')             #(1)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMDCLXVI')         #(2)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCCLXXXVIII') #(3)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'I')                #(4)
<_sre.SRE_Match object at 0x008EEB48>
```

1. Dopasowany zostanie początek napisu, potem jeden z możliwych czterech znaków `M` i następnie `D?C{0,3}`. Z kolei `D?C{0,3}` dopasuje opcjonalny znak `D` i zero z trzech możliwych znaków `C`. Idąc dalej, dopasowany zostanie `L?X{0,3}` poprzez dopasowanie opcjonalnego znaku `L` i zero z możliwych trzech znaków `X`. Następnie dopasowujemy `V?I{0,3}` dodając opcjonalne `V` i zero z trzech możliwych znaków `I`, a ostatecznie dopasowujemy koniec napisu. MDLV jest rzymską reprezentacją liczby 1555.

2. To dopasuje początek napisu, następnie dwa z czterech możliwych znaków `M`, a potem `D?C{0,3}` z `D` i jednym z trzech możliwych znaków `C`. Dalej dopasujemy `L?X{0,3}` z `L` i jednym z trzech możliwych znaków `X`, a następnie `V?I{0,3}` z `V` i jednym z trzech możliwych znaków `I`, a w końcu koniec napisu. MMDCLXVI jest reprezentacją liczby 2666.

3. Tutaj dopasowany zostanie początek napisu, a potem trzy z trzech znaków `M`, a następnie `D?C{0,3}` ze znakiem `D` i trzema z trzech możliwych znaków `C`. Potem dopasujemy `L?X{0,3}` z `L` i trzema z trzech znaków `X`, następnie `V?I{0,3}` z `V` i trzema z trzech możliwych znaków `I`, a ostatecznie koniec napisu. MMMDCCCLXXXVIII jest reprezentacją liczby 3888 i ponadto jest najdłuższą liczbą Rzymską, którą można zapisać bez rozszerzonej składni.

4. Obserwuj dokładnie. Dopasujemy początek napisu, potem zero z czterech `M`, następnie dopasowujemy `D?C{0,3}` pomijając opcjonalne `D` i dopasowując zero z trzech znaków `C`. Następnie dopasowujemy `L?X{0,3}` pomijając opcjonalne `L` i dopasowując zero z trzech znaków `X`, a potem dopasowujemy `V?I{0,3}` pomijając opcjonalne `V`, ale dopasowując jeden z trzech możliwych `I`. Ostatecznie dopasowujemy koniec napisu.

Jeśli prześledziłeś to wszystko i zrozumiałeś to za pierwszym razem, jesteś bardzo bystry. Teraz wyobraźmy sobie sytuację, że próbujemy zrozumieć jakieś inne wyrażenie regularne, które znajduje się w centralnej, krytycznej funkcji wielkiego programu. Albo wyobraź sobie nawet, że wracasz do swojego wyrażenia regularnego po kilku miesiącach. Sytuacje takie mogą nie wyglądać ciekawie...

W następnym podrozdziale dowiemy się o alternatywnej składni, która pomaga zrozumieć i zarządzać wyrażeniami.

8.5 Rozwlekłe wyrażenia regularne

Rozwlekłe wyrażenia regularne

Jak na razie, mieliśmy do czynienia z czymś, co nazywam "zwięzłymi" wyrażeniami regularnymi. Jak pewnie zauważyliśmy, są one trudne do odczytania i nawet, jeśli już je rozszyfrujemy, nie ma gwarancji, że zrobimy to za np. sześć miesięcy. To, czego potrzebujemy, to dokumentacja w ich treści.

Python pozwala na to przez tworzenie *rozwlekłych wyrażeń regularnych* (ang. *verbose regular expressions*). Różnią się one od zwięzłych dwoma rzeczami:

- Białe znaki są ignorowane. Spacje, znaki tabulacji, znaki nowej linii nie są dopasowywane jako spacje, znaki tabulacji lub znaki nowej linii. Znaki te nie są w ogóle dopasowywane. (Jeśli byśmy chcieli jednak dopasować któryś z nich, musisz poprzedzić je odwrotnym ukośnikiem (\).)
- Komentarze są ignorowane. Komentarz w rozwlekłym wyrażeniu regularnym wygląda dokładnie tak samo, jak w kodzie Pythona: zaczyna się od `#` i leci aż do końca linii. W tym przypadku jest to komentarz w wieloliniowym łańcuchu znaków, a nie w kodzie źródłowym, ale zasada działania jest taka sama.

Łatwiej będzie to zrozumieć jeśli skorzystamy z przykładu. Skorzystajmy ze zwięzłego wyrażenia regularnego, które utworzyliśmy wcześniej i zróbmy z niego rozwlekłe. Ten przykład pokazuje jak.

Przykład 7.9 Wyrażenia regularne z komentarzami

```
>>> pattern = """
    ^                   # początek łańcucha znaków
    M{0,3}              # tysiące - 0 do 3 M
    (CM|CD|D?C{0,3})    # setki - 900 (CM), 400 (CD), 0-300 (0 do 3 C),
                        #           lub 500-800 (D, a po nim 0 do 3 C)
    (XC|XL|L?X{0,3})    # dziesiątki - 90 (XC), 40 (XL), 0-30 (0 do 3 X),
                        #        or 50-80 (L, a po nim 0 do 3 X)
    (IX|IV|V?I{0,3})    # jedności - 9 (IX), 4 (IV), 0-3 (0 do 3 I),
                        #        lub 5-8 (V, a po nim 0 do 3 I)
    $                   # koniec łańcucha znaków
    """
>>> re.search(pattern, 'M', re.VERBOSE)                #(1)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)        #(2)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCCLXXXVIII', re.VERBOSE) #(3)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'M')                            #(4)
```

1. Najważniejszą rzeczą o której należy pamiętać, gdy korzystamy z rozwlekłych wyrażeń regularnych jest to, że musimy przekazać dodatkowy argument: `re.VERBOSE`. Jest to stała zdefiniowana w module `re`, która sygnalizuje, że wyrażenie powinno być traktowane jako rozwlekłe. Jak widzimy, ten wzorzec ma mnóstwo białych znaków (które są ignorowane) i kilka komentarzy (które też są ignorowane). Gdy usuniemy białe znaki i komentarze, to pozostanie dokładnie to samo wyrażenie

regularne, jakie otrzymaliśmy w poprzednim przykładzie, ale o wiele bardziej czytelne. (Zauważmy, że co prawda łańcuch znaków posiada polskie znaki, ale nie tworzymy go w unikodzie, ponieważ i tak te znaki nie mają dla nas żadnego znaczenia, ponieważ są w komentarzach.)

2. To dopasowuje początek łańcucha, potem jedno z trzech możliwych `M`, potem `CM`, `L` i trzy z trzech możliwych `X`, a następnie `IX` i koniec łańcucha.

3. To dopasowuje początek łańcucha, potem trzy z trzech możliwych `M`, dalej `D`, trzy z trzech możliwych `C`, `L` z trzema możliwymi `X`, potem `V` z trzema możliwymi `I` i na koniec koniec łańcucha.

4. Tutaj nie udało się dopasować niczego. Czemu? Ponieważ nie przekazaliśmy flagi `re.VERBOSE`, więc funkcja `re.search` traktuje to wyrażenie regularne jako zwięzłe, z dużą ilością białych znaków i kratek. Python nie rozpoznaje samodzielnie, czy każemy mu dopasować zwięzłe, czy może rozwlekłe wyrażenie regularne i przyjmuje, że każde jest zwięzłe, chyba że wyraźnie wskażemy, że tak nie jest.

8.6 Analiza przypadku: Przetwarzanie numerów telefonów

Analiza przypadku: Przetwarzanie numerów telefonów

Do tej pory koncentrowaliśmy się na dopasowywaniu całych wzorców. Albo pasował albo nie. Ale wyrażenia regularne są dużo potężniejsze. Gdy zostanie dopasowane, można wyciągnąć z niego wybrane kawałki i dzięki temu sprawdzić co gdzie zostało dopasowane.

Oto kolejny przykład z życia wzięty, z jakim można się spotkać: przetwarzanie amerykańskich numerów telefonów. Klient chciał móc wprowadzać numer w dowolnej formie w jednym polu, ale potem chciał, żeby przechowywać oddzielnie numer kierunkowy, numer w dwóch częściach i opcjonalny numer wewnętrzny w bazie danych firmy. W Internecie można znaleźć wiele takich wyrażeń regularnych, ale żadne z nich nie jest aż tak bardzo restrykcyjne.

Oto przykłady numerów telefonów jakie miał program przetwarzać:

- 800-555-1212
- 800 555 1212
- 800.555.1212
- (800) 555-1212
- 1-800-555-1212
- 800-555-1212-1234
- 800-555-1212x1234
- 800-555-1212 ext. 1234
- work 1-(800) 555.1212 #1234

Całkiem duże zróżnicowanie! W każdym z tych przypadków musimy wiedzieć, że numerem kierunkowy 800, że pierwszą częścią numeru jest 555, druga 1212, a dla tych z numerem wewnętrznym 1234.

Spróbujmy rozwiązać ten problem. Poniższy przykład pokazuje pierwszy krok.

Przykład 7.10 Odnajdywanie numerów

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$') #(1)
>>> phonePattern.search('800-555-1212').groups()            #(2)
('800', '555', '1212')
>>> phonePattern.search('800-555-1212-1234')                #(3)
>>>
```

1. Zawsze odczytujemy wyrażenie regularne od lewej do prawej. Tutaj dopasowujemy początek łańcucha znaków, potem `(\d{3})`. Co to takiego te `(\d{3})`? `{3}` oznacza “dopasuj dokładnie 3 wystąpienia” (jest to wariacja składni `{n, m}`). `\d` oznacza “jakakolwiek cyfra” (od 0 do 9). Umieszczenie ich w nawiasach oznacza “dopasuj dokładnie 3 cyfry, i zapamiętaj je jako grupę, o którą można zapytać później”. Następnie mamy dopasować myślnik. Dalej dopasuj kolejną grupę

dokładnie trzech cyfr, a następnie kolejny myślnik, i ostatnią grupę tym razem czterech cyfr. Na koniec dopasuje koniec łańcucha znaków.

2. Aby otrzymać grupy, które zapamięta moduł przetwarzania wyrażeń regularnych, należy skorzystać z metody `groups()` obiektu zwracanego przez funkcję `search`. Zwróci ona krotkę z ilością elementów równą ilości grup zdefiniowanych w wyrażeniu regularnym. W tym przypadku mamy trzy grupy: dwie po 3 cyfry i ostatnią czterocyfrową.

3. To jednak nie jest rozwiązaniem naszego problemu, bo nie dopasowuje numeru telefonu z numerem wewnętrznym. Musimy więc je rozszerzyć.

Przykład 7.11 Odnajdywanie numeru wewnętrznego

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$') #(1)
>>> phonePattern.search('800-555-1212-1234').groups()             #(2)
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')                      #(3)
>>>
>>> phonePattern.search('800-555-1212')                           #(4)
>>>
```

1. To wyrażenie regularne jest praktycznie identyczne z wcześniejszym. Tak jak wcześniej, dopasowujemy początek łańcucha, potem zapamiętywaną grupę trzech cyfr, myślnik, zapamiętywaną grupę trzech cyfr, myślnik i zapamiętywaną grupę czterech cyfr. Nową częścią jest kolejny myślnik i zapamiętywana grupa jednej lub więcej cyfr. Na końcu jak w poprzednim przykładzie dopasowujemy koniec łańcucha.

2. Metoda `groups()` zwraca teraz krotkę czterech elementów, ponieważ wyrażenie regularne definiuje teraz cztery grupy do zapamiętania.

3. Niestety nie jest to wersja ostateczna, gdyż zakładamy, że każda część numeru telefonu jest rozdzielona myślnikiem. Co jeśli będą rozdzielone spacją, kropką albo przecinkiem? Potrzebujemy bardziej ogólnego rozwiązania.

4. Ups! Nie tylko to wyrażenie nie robi wszystkiego co powinno, ale cofnęliśmy się wstecz, gdyż teraz nie dopasowuje ono numerów bez numeru wewnętrznego. To nie jest to co chcieliśmy; jeśli w numerze jest podany numer wewnętrzny, to chcemy go znać, ale jeśli go nie ma, to i tak chcemy znać inne części numeru telefonu.

Następny przykład pokazuje wyrażenie regularne, które obsługuje różne separatory między częściami numeru telefonu.

Przykład 7.12 Obsługa różnych separatorów

```
>>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$') #(1)
>>> phonePattern.search('800 555 1212 1234').groups()                   #(2)
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups()                   #(3)
('800', '555', '1212', '1234')
```

```
>>> phonePattern.search('80055512121234')                              #(4)
>>>
>>> phonePattern.search('800-555-1212')                                #(5)
>>>
```

1. Teraz dopasowujemy początek łańcucha, grupę trzech cyfr, potem `\D+`... zaraz, zaraz, co to jest? `\D` dopasowuje dowolny znak, który nie jest cyfrą, a + oznacza "jeden lub więcej". Więc `\D+` dopasowuje jeden lub więcej znaków nie będących cyfrą. Korzystamy z niego, aby dopasować różne separatory, nie tylko myślniki.

2. Korzystanie z `\D+` zamiast z `-` pozwala na dopasowywanie numerów telefonów ze spacjami w roli separatora części.

3. Oczywiście myślniki też działają.

4. Niestety, to dalej nie jest ostateczna wersja, ponieważ nie obsługuje ona braku jakichkolwiek separatorów.

5. No i dalej nie rozwiązany pozostał problem możliwości braku numeru wewnętrznego. Mamy więc dwa problemy do rozwiązania, ale w obu przypadkach rozwiążemy problem tą samą techniką.

Następny przykład pokazuje wyrażenie regularne pasujące także do numeru bez separatorów.

Przykład 7.13 Obsługa numerów bez separatorów

```
>>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') #(1)
>>> phonePattern.search('80055512121234').groups()                     #(2)
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups()                 #(3)
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()                       #(4)
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234')                          #(5)
>>>
```

1. Jedyna zmiana jakiej dokonaliśmy od ostatniego kroku to zamiana wszystkich `+` na `*`. Zamiast `\D+` pomiędzy częściami numeru telefonu dopasowujemy teraz `\D*`. Pamiętasz, że `+` oznacza "1 lub więcej"? `*` oznacza "0 lub więcej". Tak więc teraz jesteśmy w stanie przetworzyć numer nawet bez separatorów.

2. Nareszcie działa! Dlaczego? Dopasowany został początek łańcucha, grupa 3 cyfr (800), potem zero znaków nienumerycznych, potem znowu zapamiętywana grupa 3 cyfr (555), znowu zero znaków nienumerycznych, zapamiętywana grupa 4 cyfr (1212), zero znaków nienumerycznych, numer wewnętrzny (1234) i nareszcie koniec łańcucha.

3. Inne odmiany też działają np. numer rozdzielony kropkami ze spacją i `x`-em przed numerem wewnętrznym.

4. Wreszcie udało się też rozwiązać problem z brakiem numeru wewnętrznego. Tak czy siak `groups()` zwraca nam krotkę z 4 elementami, ale ostatni jest tutaj pusty.

5. Niestety jeszcze nie skończyliśmy. Co tutaj jest nie tak? Przed numerem kierunkowym znajduje się dodatkowy znak "(", a nasze wyrażenie zakłada, że numer kierunkowy znajduje się na samym przodzie. Nie ma problemu, możemy zastosować tę samą metodę co do znaków rozdzielających.

Następny przykład pokazuje jak sobie radzić ze znakami wiodącymi w numerach telefonów.

Przykład 7.14 Obsługa znaków na początku numeru telefonu

```
>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') #(1)
>>> phonePattern.search('(800)5551212 ext. 1234').groups()                 #(2)
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()                           #(3)
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234')                     #(4)
>>>
```

1. Wzorzec w tym przykładzie jest taki sam jak w poprzednim, z wyjątkiem tego, że teraz na początku łańcucha dopasowujemy `\D*` przed pierwszą zapamiętywaną grupą (numerem kierunkowym). Zauważ że tych znaków nie zapamiętujemy (nie są one w nawiasie). Jeśli je napotkamy, to ignorujemy je i przechodzimy do numeru kierunkowego.

2. Teraz udało się przetworzyć numer telefonu z nawiasem otwierającym na początku. (Zamykający był już wcześniej obsługiwany; był traktowany jako nienumeryczny znak pasujący do teraz drugiego `\D*`.)

3. Tak na wszelki wypadek sprawdzamy czy nie popsuliśmy czegoś. Jako, że początkowy znak jest całkowicie opcjonalny, następuje dopasowanie w dokładnie taki sam sposób jak w poprzednim przykładzie.

4. W tym miejscu wyrażenia regularne sprawiają, że chce się człowiekowi rozbić bardzo dużym młotem monitor. Dlaczego to nie pasuje? Wszystko za sprawą 1 przed numerem kierunkowym (numer kierunkowy USA), a przecież przyjęliśmy, że na początku mogą być tylko nienumeryczne znaki. Ech...

Cofnijmy się na chwilę. Jak na razie wszystkie wyrażenia dopasowywaliśmy od początku łańcucha. Ale teraz widać wyraźnie, że na początku naszego łańcucha mamy nieokreśloną liczbę znaków których kompletnie nie potrzebujemy. Po co mamy więc dopasowywać początek łańcucha? Jeśli tego nie zrobimy, to przecież pominie on tyle znaków ile mu się uda, a przecież o to nam chodzi. Takie podejście prezentuje następny przykład.

Przykład 7.15 Numerze telefonu, znajdę cię gdziekolwiek jesteś!

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') #(1)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()       #(2)
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                               #(3)
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234')                             #(4)
('800', '555', '1212', '1234')
```

1. Zauważ, że brakuje ^ w tym wyrażeniu regularnym, Teraz już nie dopasowujemy początku łańcucha, bo przecież nikt nie powiedział, że wyrażenie musi pasować do całego łańcucha, a nie do fragmentu. Mechanizm wyrażeń regularnych sam zadba o namierzenie miejsca do którego ono pasuje (o ile w ogóle).

2. Teraz nareszcie pasuje numer ze znakami na początku (w tym cyframi) i dowolnymi, jakimikolwiek separatorami w środku.

3. Na wszelki wypadek sprawdzamy i to. Działa!

4. To też działa.

Widzimy, jak szybko wyrażenia regularne wymykają się spod kontroli? Rzućmy okiem na jedną z poprzednich przykładów. Widzimy różnice pomiędzy nim i następnym?

Póki jeszcze rozumiemy to co napisaliśmy, rozpiszmy to jako rozwlekłe wyrażenie regularne, żeby nie zapomnieć, co jest co i dlaczego.

Przykład 7.16 Przetwarzanie numerów telefonu (wersja finalna)

```
>>> phonePattern = re.compile(r'''
                # nie dopasowuj początku łańcucha, numer może się zacząć gdziekolwiek
    (\d{3})     # numer kierunkowy - 3 cyfry (np. '800')
    \D*         # opcjonalny nienumeryczny separator
    (\d{3})     # pierwsza część numeru - 3 cyfry (np. '555')
    \D*         # opcjonalny separator
    (\d{4})     # druga część numeru (np. '1212')
    \D*         # opcjonalny separator
    (\d*)       # numer wewnętrzny jest opcjonalny i może mieć dowolną długość
    $           # koniec łańcucha
    ''', re.VERBOSE)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()        #(1)
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                                #(2)
('800', '555', '1212', '')
```

1. Pomijając fakt, że jest ono podzielone na wiele linii, to wyrażenie jest dokładnie takie samo jak po ostatnim kroku, więc nie jest niespodzianką, że dalej działa jak powinno.

2. Jeszcze jedna próba. Tak, działa! Skończone!

8.7 Wyrażenia regularne - podsumowanie

Podsumowanie

To co przedstawiliśmy tutaj, to zaledwie wierzchołek góry lodowej, odnośnie tego co potrafią wyrażenia regularne. Innymi słowy, mimo że jesteśmy teraz nimi przytłoczeni, uwierzmy, że jeszcze nic nie widzieliśmy.

Powinieneś już być zaznajomiony z poniższymi technikami:

- `^` dopasowuje początek napisu.
- `$` dopasowuje koniec napisu.
- `\b` dopasowuje początek lub koniec słowa.
- `\d` dopasowuje dowolną cyfrę.
- `\D` dopasowuje dowolny znak, który nie jest cyfrą.
- `x?` dopasowuje opcjonalny znak `x` (innymi słowy, dopasowuje `x` zero lub jeden raz).
- `x*` dopasowuje `x` zero lub więcej razy.
- `x+` dopasowuje `x` jeden lub więcej razy.
- `x{n,m}` dopasowuje znak `x` co najmniej `n` razy, lecz nie więcej niż `m` razy.
- `(a|b|c)` dopasowuje `a` albo `b` albo `c`.
- `(x)` generalnie jest to zapamiętana grupa. Można otrzymać wartość, która została dopasowana, wykorzystując metodę `groups()` obiektu zwróconego przez `re.search`.

Wyrażenia regularne dają ekstremalnie potężne możliwości, lecz nie zawsze są poprawnym rozwiązaniem do każdego problemu. Powinno się więcej o nich poczytać, aby się dowiedzieć, kiedy będą one odpowiednie podczas rozwiązywania pewnych problemów, czy też kiedy mogą bardziej powodować problemy, niż je rozwiązywać.

> "Niektórzy ludzie, kiedy napotkają problem, myślą: 'Wiem, użyję wyrażeń regularnych'. I teraz mają dwa problemy."
> – Jamie Zawinski

Rozdział 9

Przetwarzanie HTML-a

9.1 Przetwarzanie HTML-a

Nurkujemy

Na comp.lang.python często można zobaczyć pytania w stylu "jak można znaleźć wszystkie nagłówki/obrazki/linki w moim dokumencie HTML?", "jak mogę sparsować/przetłumaczyć/przerobić tekst mojego dokumentu HTML tak, aby zostawić znaczniki w spokoju?" lub też "jak mogę natychmiastowo dodać/usunąć/zacytować atrybuty z wszystkich znaczników mojego dokumentu HTML?". Rozdział ten odpowiada na wszystkie te pytania.

Poniżej przedstawiono w dwóch częściach całkowicie działający program. Pierwsza część, BaseHTMLProcessor.py jest ogólnym narzędziem, które przetwarza pliki HTML przechodząc przez wszystkie znaczniki i bloki tekstowe. Druga część, dialect.py, jest przykładem tego, jak wykorzystać BaseHTMLProcessor.py, aby przetłumaczyć tekst dokumentu HTML, lecz przy tym zostawiając znaczniki w spokoju. Przeczytaj notki dokumentacyjne i komentarze w celu zorientowania się, co się tutaj właściwie dzieje. Duża część tego kodu wygląda jak czarna magia, ponieważ nie jest oczywiste w jaki sposób dowolna z metod klasy jest wywoływana. Jednak nie martw się, wszystko zostanie wyjaśnione w odpowiednim czasie.

Przykład 8.1 BaseHTMLProcessor.py

```
#-*- coding: utf-8 -*-

from sgmllib import SGMLParser
import htmlentitydefs

class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        # dodatek (wywoływane przez SGMLParser.__init__)
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs):
        # wywoływane dla każdego początkowego znacznika
        # attrs jest listą krotek (atrybut, wartość)
        # np. dla <pre class="screen"> będziemy mieli tag="pre",
        # attrs=[("class", "screen")]
        # Chcielibyśmy zrekonstruować oryginalne znaczniki i atrybuty, ale
        # może się zdarzyć, że umieścimy w cudzysłowach wartości, które nie były
        # zacytowane w źródle dokumentu, a także możemy zmienić rodzaj
        # cudzysłowów w wartości danego atrybutu (pojedyncze cudzysłowy lub podwójne).
        # Dodajmy, że niepoprawnie osadzony kod nie-HTML-owy (np. kod JavaScript)
        # może zostać źle sparsowany przez klasę bazową, a to spowoduje błąd
        # wykonania skryptu.
        # Cały nie-HTML musi być umieszczony w komentarzu HTML-a (<!-- kod -->),
        # aby parser zostawił ten niezmieniony (korzystając z handle_comment).
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
```

```
    def unknown_endtag(self, tag):
        # wywoływane dla każdego znacznika końcowego np. dla </pre>, tag będzie równy "pre"
        # Rekonstruuje oryginalny znacznik końcowy w wyjściowym dokumencie
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):
        # wywoływane jest dla każdego odwołania znakowego np. dla " ",
        # ref będzie równe "160"
        # Rekonstruuje oryginalne odwołanie znakowe.
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):
        # wywoływane jest dla każdego odwołania do encji np. dla "&copy;",
        # ref będzie równe "copy"
        # Rekonstruuje oryginalne odwołanie do encji.
        self.pieces.append("&%(ref)s" % locals())
        # standardowe encje HTML-a są zakończone średnikiem; pozostałe encje
        # (encje spoza HTML-a) nie są
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):
        # wywoływane dla każdego bloku czystego teksu np. dla danych spoza dowolnego
        #znacznika, w których nie występują żadne odwołania znakowe, czy odwołania do encji.
        # Przechowuje dosłownie oryginalny tekst.
        self.pieces.append(text)

    def handle_comment(self, text):
        # wywoływane dla każdego komentarza np. <!-- wpis kod JavaScript w tym miejscu -->
        # Rekonstruuje oryginalny komentarz.
        # Jest to szczególnie ważne, gdy dokument zawiera kod przeznaczony
        # dla przeglądarki (np. kod Javascript) wewnątrz komentarza, dzięki temu
        # parser może przejść przez ten kod bez zakłóceń;
        # więcej szczegółów w komentarzu metody unknown_starttag.
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):
        # wywoływane dla każdej instrukcji przetwarzania np. <?instruction>
        # Rekonstruuje oryginalną instrukcję przetwarzania
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        # wywoływane dla deklaracji typu dokumentu, jeśli występuje, np.
        # <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        #     "http://www.w3.org/TR/html4/loose.dtd">
        # Rekonstruuje oryginalną deklarację typu dokumentu
        self.pieces.append("<!%(text)s>" % locals())

    def output(self):
        u"""Zwraca przetworzony HTML jako pojedynczy łańcuch znaków"""
```

```
        return "".join(self.pieces)

if __name__ == "__main__":
    for k, v in globals().items():
        print k, "=", v
```

Przykład 8.2 dialect.py

```
#-*- coding: utf-8 -*-

import re
from BaseHTMLProcessor import BaseHTMLProcessor

class Dialectizer(BaseHTMLProcessor):
    subs = ()

    def reset(self):
        # dodatek (wywoływany przez __init__ klasy bazowej)
        # Resetuje wszystkie atrybuty
        self.verbatim = 0
        BaseHTMLProcessor.reset(self)

    def start_pre(self, attrs):
        # wywoływane dla każdego znacznika <pre> w źródle HTML
        # Zwiększa licznik trybu dosłowności verbatim, a następnie
        # obsługuje ten znacznik normalnie
        self.verbatim += 1
        self.unknown_starttag("pre", attrs)

    def end_pre(self):
        # wywoływane dla każdego znacznika </pre>
        # Zmiejsza licznik trybu dosłowności verbatim
        self.unknown_endtag("pre")
        self.verbatim -= 1

    def handle_data(self, text):
        # metoda nadpisana
        # wywoływane dla każdego bloku tekstu w źródle
        # Jeśli jest w trybie dosłownym, zapisuje tekst niezmieniony;
        # inaczej przetwarza tekst za pomocą szeregu podstawień
        self.pieces.append(self.verbatim and text or self.process(text))

    def process(self, text):
        # wywoływane z handle_data
        # Przetwarza każdy blok wykonując serie podstawień
        # za pomocą wyrażeń regularnych (podstawienia są definiowane przez
        # klasy pochodne)
        for fromPattern, toPattern in self.subs:
```

```
            text = re.sub(fromPattern, toPattern, text)
        return text

class ChefDialectizer(Dialectizer):
    u"""konwertuje HTML na mowę szwedzkiego szefa kuchni

    oparte na klasycznym chef.x, copyright (c) 1992, 1993 John Hagerman
    """
    subs = ((r'a([nu])', r'u\1'),
            (r'A([nu])', r'U\1'),
            (r'a\B', r'e'),
            (r'A\B', r'E'),
            (r'en\b', r'ee'),
            (r'\Bew', r'oo'),
            (r'\Be\b', r'e-a'),
            (r'\be', r'i'),
            (r'\bE', r'I'),
            (r'\Bf', r'ff'),
            (r'\Bir', r'ur'),
            (r'(\w*?)i(\w*?)$', r'\1ee\2'),
            (r'\bow', r'oo'),
            (r'\bo', r'oo'),
            (r'\bO', r'Oo'),
            (r'the', r'zee'),
            (r'The', r'Zee'),
            (r'th\b', r't'),
            (r'\Btion', r'shun'),
            (r'\Bu', r'oo'),
            (r'\BU', r'Oo'),
            (r'v', r'f'),
            (r'V', r'F'),
            (r'w', r'w'),
            (r'W', r'W'),
            (r'([a-z])[.]', r'\1.  Bork Bork Bork!'))

class FuddDialectizer(Dialectizer):
    u"""konwertuje HTML na mowę Elmer Fudda"""
    subs = ((r'[rl]', r'w'),
            (r'qu', r'qw'),
            (r'th\b', r'f'),
            (r'th', r'd'),
            (r'n[.]', r'n, uh-hah-hah-hah.'))

class OldeDialectizer(Dialectizer):
    u"""konwertuje HTML na pozorowany język średnioangielski"""
    subs = ((r'i([bcdfghjklmnpqrstvwxyz])e\b', r'y\1'),
            (r'i([bcdfghjklmnpqrstvwxyz])e', r'y\1\1e'),
            (r'ick\b', r'yk'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'e[ea]([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
```

```
            (r'([bcdfghjklmnpqrstvwxyz])y', r'\1ee'),
            (r'([bcdfghjklmnpqrstvwxyz])er', r'\1re'),
            (r'([aeiou])re\b', r'\1r'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'i\1e'),
            (r'tion\b', r'cioun'),
            (r'ion\b', r'ioun'),
            (r'aid', r'ayde'),
            (r'ai', r'ey'),
            (r'ay\b', r'y'),
            (r'ay', r'ey'),
            (r'ant', r'aunt'),
            (r'ea', r'ee'),
            (r'oa', r'oo'),
            (r'ue', r'e'),
            (r'oe', r'o'),
            (r'ou', r'ow'),
            (r'ow', r'ou'),
            (r'\bhe', r'hi'),
            (r've\b', r'veth'),
            (r'se\b', r'e'),
            (r"'s\b", r'es'),
            (r'ic\b', r'ick'),
            (r'ics\b', r'icc'),
            (r'ical\b', r'ick'),
            (r'tle\b', r'til'),
            (r'll\b', r'l'),
            (r'ould\b', r'olde'),
            (r'own\b', r'oune'),
            (r'un\b', r'onne'),
            (r'rry\b', r'rye'),
            (r'est\b', r'este'),
            (r'pt\b', r'pte'),
            (r'th\b', r'the'),
            (r'ch\b', r'che'),
            (r'ss\b', r'sse'),
            (r'([wybdp])\b', r'\1e'),
            (r'([rnt])\b', r'\1\1e'),
            (r'from', r'fro'),
            (r'when', r'whan'))

def translate(url, dialectName="chef"):
    u"""pobiera plik na podstawie URL-a
    i tłumaczy korzystając z dialektu, gdzie
    dialekt in ("chef", "fudd", "olde")"""
    import urllib
    sock = urllib.urlopen(url)
    htmlSource = sock.read()
    sock.close()
    parserName = "%sDialectizer" % dialectName.capitalize()
    parserClass = globals()[parserName]
```

```
    parser = parserClass()
    parser.feed(htmlSource)
    parser.close()
    return parser.output()

def test(url):
    u"""testuje wszystkie dialekty na pewnym URL-u"""
    for dialect in ("chef", "fudd", "olde"):
        outfile = "%s.html" % dialect
        fsock = open(outfile, "wb")
        fsock.write(translate(url, dialect))
        fsock.close()
        import webbrowser
        webbrowser.open_new(outfile)

if __name__ == "__main__":
    test("http://diveintopython.org/odbchelper_list.html")
```

Uruchamiając ten skrypt, przetłumaczymy podrozdział 3.2, z książki "Dive Into Python", na pozorowany szwedzki kuchmistrza z Muppetów, udawany język Elmer Fudda (z kreskówek Królik Bugs) i pozorowany język średnioangielski (luźno oparty na "Chaucer's The Canterbury Tales"). Jeśli spojrzymy na źródło HTML wyjściowej strony, zobaczymy, że znaczniki i atrybuty zostały nietknięte, lecz tekst między znacznikami został "przetłumaczonyńa udawany język. Jeśli przyglądniemy się jeszcze bardziej, zobaczymy, że tylko tytuły i akapity zostały przetłumaczone. Przedstawione kody i wyniki działania programu zostały niezmienione.

Przykład 8.3 Wyjście z dialect.py

```
<div class="abstract">
<p>Lists awe <span class="application">Pydon</span>'s wowkhowse datatype.
If youw onwy expewience wif wists is awways in
<span class="application">Visuaw Basic</span> ow (God fowbid) de datastowe
in <span class="application">Powewbuiwdew</span>, bwace youwsewf fow
<span class="application">Pydon</span> wists.</p>
</div>
```

9.2 Wprowadzenie do sgmllib.py

Wprowadzenie do sgmllib.py

Przetwarzanie HTML-a jest podzielone na trzy etapy: podzielenie dokumentu na elementy składowe, manipulowanie tymi elementami i ponowna rekonstrukcja tych kawałków do HTML-a. Pierwszy krok jest wykonywany przez `sgmllib.py`, który jest częścią standardowej biblioteki Pythona.

Kluczem do zrozumienia tego rozdziału jest uświadomienie sobie, że HTML to nie tylko tekst, jest to tekst z pewną strukturą. Struktura ta powstaje z mniej lub bardziej hierarchicznych sekwencji znaczników początkowych i znaczników końcowych. Zazwyczaj nie pracujemy z HTML-em w sposób strukturalny, raczej tekstowo w edytorze tekstu lub wizualnie w przeglądarce internetowej, czy innym narzędziu. `sgmllib.py` prezentuje HTML strukturalnie.

`sgmllib.py` zawiera jedną ważną klasę: `SGMLParser`. `SGMLParser` rozbiera HTML na użyteczne kawałki takie jak znaczniki początkowe i znaczniki końcowe. Jak tylko udaje mu się rozebrać jakieś dane na przydatne kawałki, wywołuję odpowiednią metodę, w zależności co zostało znalezione. Żeby wykorzystać parser, tworzymy podklasę `SGMLParser`-a i nadpisujemy te metody. Mówiąc, że `sgmllib.py` prezentuje HTML strukturalnie, mieliśmy na myśli to, że struktura dokumentu HTML jest określana poprzez wywoływane metody, a także argumenty przekazywane do tych metod.

`SGMLParser` parsuje HTML na 8 rodzajów danych i wykonuje odpowiednie metody dla każdego z nich:

Znacznik początkowy Znacznik HTML, który rozpoczyna blok np. `<html>`, `<head>`, `<body>` lub `<pre>` lub samodzielne znaczniki jak `<br>` lub `<img>`. Kiedy odnajdzie znacznik `tagname`, to `SGMLParser` będzie szukał metod o nazwie `start_tagname` lub `do_tagname`. Na przykład, jeśli odnajdzie znacznik `<pre>`, to będzie szukał metod `start_pre` lub `do_pre`. Jeśli je znajdzie, `SGMLParser` wywoła te metody z listą atrybutów tego znacznika. W przeciwnym wypadku wywoła `unknown_starttag` z nazwą znacznika i listą atrybutów.

Znacznik końcowy Znacznik HTML, który kończy blok np. `</html>`, `</head>`, `</body>` lub `</pre>`. Kiedy odnajdzie znacznik końcowy, `SGMLParser` będzie szukał metody o nazwie `end_tagname`. Jeśli ją znajdzie, wywoła tę metodę, jeśli nie, wywoła metodę `unknown_endtag` z nazwą znacznika.

Odwołania znakowe Znak specjalny, do którego dowołujemy się podając jego dziesiętny lub szesnastkowy odpowiednik np. ` `. Kiedy odwołanie znakowe zostanie odnalezione, `SGMLParser` wywoła `handle_charref` z tekstem dziesiętnego lub szesnastkowego odpowiednika znaku.

Odwołanie do encji Encja HTML to np. `©`. Kiedy zostanie znaleziona, `SGMLParser` wywołuje `handle_entityref` z nazwą encji.

Komentarz Element HTML, który jest ograniczony przez `<!-- ... -->`. Kiedy zostanie znaleziony, `SGMLParser` wywołuje `handle_comment` z zawartością komentarza.

Instrukcje przetwarzania Instrukcje przetwarzania HTML są ograniczone przez `<? ... >`. Kiedy zostaną odnalezione, `SGMLParser` wywołuje `handle_pi` z zawartością instrukcji przetwarzania.

Deklaracja Deklaracja HTML np. typu dokumentu (`DOCTYPE`), jest ograniczona przez `<! ... >`. Kiedy zostanie znaleziona, `SGMLParser` wywołuje `handle_decl` z zawartością deklaracji.

Dane tekstowe Bloki tekstu. Wszystko inne, co się nie mieści w innych 7 kategoriach. Kiedy zostaną one znalezione, `SGMLParser` wywoła `handle_data` z tekstem.

`sgmllib.py` posiada zestaw testów, które to ilustrują. Możemy uruchomić `sgmllib.py`, podając w linii poleceń nazwę pliku, a będzie on wyświetlał znaczniki i inne elementy podczas parsowania. Zrobione jest to poprzez utworzenie podklasy `SGMLParser` i zdefiniowanie metod `unknown_starttag`, `unknown_endtag`, `handle_data` i innych metod, które będą po prostu wyświetlać swoje argumenty.

W ActivePython IDE pod Windows możemy określić argumenty linii poleceń za pomocą “Run script”. Różne argumenty oddzielamy spacją.

Przykład 8.4 Test `sgmllib.py`

```
<span>c:\python23\lib> type "c:\downloads\diveintopython\html\toc\index.html"</span>
<!DOCTYPE html
   PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
 <html lang="en">
    <head>
       <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

       <title>Dive Into Python</title>
       <link rel="stylesheet" href="diveintopython.css" type="text/css">

 [...ciach...]
```

Tutaj jest kawałek spisu treści angielskiej wersji tej książki, w HTML-u. Oczywiście ścieżki do plików możesz mieć trochę inne. (Angielską wersję tej książki, w formacie HTML, możesz znaleźć na http://diveintopython.org/.)

Uruchomiając to za pomocą zestawu testów `sgmllib.py`, zobaczymy:

```
<span>c:\python23\lib> python sgmllib.py
"c:\downloads\diveintopython\html\toc\index.html"</span>
data: '\n\n'
 start tag: <html lang="en" >
 data: '\n   '
 start tag: <head>
 data: '\n      '
 start tag: <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" >
 data: '\n   \n      '
 start tag: <title>
 data: 'Dive Into Python'
 end tag: </title>
 data: '\n      '
 start tag: <link rel="stylesheet" href="diveintopython.css" type="text/css" >
 data: '\n      '
```

```
[...ciach...]
```

Taki jest plan reszty tego rozdziału:

- Dziedziczymy po `SGMLParser`, aby stworzyć klasy, które wydobywają interesujące dane z dokumentu HTML.
- Dziedziczymy po `SGMLParser`, aby stworzyć podklasę `BaseHTMLProcessor`, która nadpisuje wszystkie 8 metod obsługi i wykorzystujemy je, aby zrekonstruować oryginalny dokument HTML z otrzymywanych kawałków.
- Dziedziczymy po `BaseHTMLProcessor`, aby utworzyć `Dialectizer`, który dodaje kilka metod w celu specjalnego przetworzenia określonych znaczników HTML. Ponadto nadpisuje metodę `handle_data`, aby zapewnić możliwość przetwarzania bloków tekstowych pomiędzy znacznikami HTML.
- Dziedziczymy po `Dialectizer`, aby stworzyć klasy, które definiują zasady przetwarzania tekstu wykorzystane w `Dialectizer.handle_data`.
- Piszemy zestaw testów, które korzystają z prawdziwej strony internetowej, http://diveintopython.org/, i ją przetwarzają.

Przy okazji dowiemy się, czym jest `locals` i `globals`, a także jak formatować łańcuchy znaków za pomocą słowników.

9.3 Wyciąganie danych z dokumentu HTML

Wyciąganie danych z dokumentu HTML

Aby wyciągnąć dane z dokumentu HTML, tworzymy podklasę klasy `SGMLParser` i definiujemy dla encji lub każdego znacznika, który nas interesuje, odpowiednią metodę.

Pierwszym krokiem do wydobycia danych z dokumentu HTML jest zdobycie jakiegoś dokumentu. Jeśli posiadamy jakiś dokument HTML na swoim twardym dysku, możemy wykorzystać funkcje do obsługi plików, aby go odczytać, jednak prawdziwa zabawa rozpoczyna się, gdy weźmiemy HTML z istniejącej strony internetowej.

Przykład 8.5 Moduł `urllib`

```
>>> import urllib                                       #(1)
>>> sock = urllib.urlopen("http://diveintopython.org/") #(2)
>>> htmlSource = sock.read()                             #(3)
>>> sock.close()                                         #(4)
>>> print htmlSource                                     #(5)
<!DOCTYPE html
  PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

      <title>Dive Into Python</title>
      <link rel="stylesheet" href="diveintopython.css" type="text/css">
      <link rev="made" href="mailto:f8dy@diveintopython.org">
      <meta name="generator" content="DocBook XSL Stylesheets V1.52.2">
      <meta name="description" content=" This book lives at .
 If you're reading it somewhere else, you may not have the latest version.">
      <meta name="keywords" content="Python, Dive Into Python, tutorial,
object-oriented, programming,
 documentation, book, free">
      <link rel="alternate" type="application/rss+xml" title="RSS"
href="http://diveintopython.org/history.xml">
   </head>
   <body>
      <table id="Header" width="100\%" border="0" cellpadding="0" cellspacing="0" summary="">
         <tr>
            <td id="breadcrumb" colspan="6">\ </td>
         </tr>
         <tr>
            <td colspan="3" id="logocontainer">
               <h1 id="logo">Dive Into Python</h1>
               <p id="tagline">Python from novice to pro</p>
            </td>

 [...ciach...]
```

1. Moduł `urllib` jest częścią standardowej biblioteki Pythona. Zawiera on funkcje służące do pobierania informacji o danych, a także pobierania samych danych z

internetu na podstawie adresu URL (głównie strony internetowe).

2. Najprostszym sposobem wykorzystania `urllib`-a jest pobranie całego tekstu strony internetowej przy pomocy funkcji `urlopen`. Otworzenie URL-a jest równie proste, jak otworzenie pliku. Zwracana wartość funkcji `urlopen` przypomina normalny obiekt pliku i posiada niektóre analogiczne metody do obiektu pliku.

3. Najprostszą czynnością, którą możemy wykonać na obiekcie zwróconym przez `urlopen`, jest wywołanie `read`. Metoda ta odczyta cały HTML strony internetowej i zwróci go w postaci łańcucha znaków. Obiekt ten posiada także metodę `readlines`, która czyta tekst linia po linii, dodając kolejne linie do listy.

4. Kiedy skończymy pracę na tym obiekcie, powinniśmy go jeszcze zamknąć za pomocą `close`, podobnie jak normalny plik.

5. Mamy kompletny dokument HTML w postaci łańcucha znaków, pobrany ze strony domowej http://diveintopython.org/ i jesteśmy przygotowani do tego, aby go sparsować.

Przykład 8.6 Wprowadzenie do urllister.py

```
from sgmllib import SGMLParser
class URLLister(SGMLParser):
    def reset(self):                              #(1)
        SGMLParser.reset(self)
        self.urls = []

    def start\_a(self, attrs):                     #(2)
        href = [v for k, v in attrs if k=='href']  #(3) (4)
        if href:
            self.urls.extend(href)
```

1. `reset` jest wywoływany przez metodę `__init__` `SGMLParser`-a, a także można go wywołać ręcznie już po utworzeniu instancji parsera. Zatem, jeśli potrzebujemy powtórnie zainicjalizować instancję parsera, który był wcześniej używany, zrobimy to za pomocą `reset` (nie przez `__init__`). Nie ma potrzeby tworzenia nowego obiektu.

2. Zawsze, kiedy parser odnajdzie znacznik `<a>`, wywoła metodę `start_a`. Znacznik może posiadać atrybut `href`, a także inne jak na przykład `name`, czy `title`. Parametr `attrs` jest listą krotek `[(atrybut1, wartość1), (atrybut2, wartość2), ...]`. Znacznik ten może być także samym `<a>`, poprawnym (lecz bezużytecznym) znacznikiem HTML, a w tym przypadku `attrs` będzie pustą listą.

3. Możemy stwierdzić, czy znacznik `<a>` posiada atrybut `href`, za pomocą prostego wielozmiennego wyrażenie listowego.

4. Porównywanie napisów (np. `k=='href'`) jest zawsze wrażliwe na wielkość liter, lecz w tym przypadku takie użycie jest bezpieczne, ponieważ `SGMLParser` konwertuje podczas tworzenia `attrs` nazwy atrybutów na małe litery.

Przykład 8.7 Korzystanie z urllister.py

```
>>> import urllib, urllister
 >>> usock = urllib.urlopen("http://diveintopython.org/")
 >>> parser = urllister.URLLister()
 >>> parser.feed(usock.read())        #(1)
 >>> usock.close()                    #(2)
 >>> parser.close()                   #(3)
 >>> for url in parser.urls: print url #(4)
toc/index.html
#download
#languages
toc/index.html
appendix/history.html
download/diveintopython-html-5.0.zip
download/diveintopython-pdf-5.0.zip
download/diveintopython-word-5.0.zip
download/diveintopython-text-5.0.zip
download/diveintopython-html-flat-5.0.zip
download/diveintopython-xml-5.0.zip
download/diveintopython-common-5.0.zip

[... ciach ...]
```

1. Wywołujemy metodę `feed` zdefiniowaną w `SGMLParser`, aby "nakarmić" parser przekazując mu kod HTML-a. Metoda ta przyjmuje łańcuch znaków, którym w tym przypadku będzie wartość zwrócona przez `usock.read()`.

2. Podobnie jak pliki, powinniśmy zamknąć swoje obiekty URL, kiedy już nie będą ci potrzebne.

3. Powinieneś także zamknąć obiekt parsera, lecz z innego powodu. Podczas czytania danych przekazujemy je do parsera, lecz metoda `feed` nie gwarantuje, że wszystkie przekazane dane, zostały przetworzone. Parser może te dane zbuforować i czekać na dalszą porcję danych. Kiedy wywołamy `close`, mamy pewność, że bufor zostanie opróżniony i wszystko zostanie całkowicie sparsowane.

4. Ponieważ parser został zamknięty, więc parsowanie zostało zakończone i `parser.urls` zawiera listę wszystkich URL-i, do których linki zawiera dokument HTML. (Twoje wyjście może wyglądać inaczej, ponieważ z biegiem czasu linki mogły ulec zmianie.)

9.4 Wprowadzenie do BaseHTMLProcessor.py

Wprowadzenie do BaseHTMLProcessor.py

`SGMLParser` nie tworzy niczego samodzielnie. On po prostu parsuje, parsuje i parsuje i wywołuje metodę dla każdej interesującej rzeczy jaką znajdzie, ale te metody nie wykonują niczego. `SGMLParser` jest konsumentem HTML-a: bierze HTML-a i rozkłada go na małe, strukturalne części. Jak już widzieliśmy w poprzednim podrozdziale, możemy dziedziczyć po klasie `SGMLParser`, aby zdefiniować klasy, które przechwycą poszczególne znaczniki i jakoś to pożytecznie wykorzystają, np. stworzą listę odnośników na danej stronie internetowej. Teraz pójdziemy krok dalej i zdefiniujemy klasę, która przechwyci wszystko, co zgłosi `SGMLParser` i zrekonstruuje kompletny dokument HTML. Używając terminologii technicznej nasza klasa będzie producentem HTML-a.

`BaseHTMLProcessor` dziedziczy po `SGMLParser` i dostarcza 8 istotnych metod obsługi: `unknown_starttag`, `unknown_endtag`, `handle_charref`, `handle_entityref`, `handle_comment`, `handle_pi`, `handle_decl` i `handle_data`.

Przykład 8.8 Wprowadzenie do BaseHTMLProcessor.py

```
class BaseHTMLProcessor(SGMLParser):
    def reset(self):                       #(1)
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs): #(2)
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())

    def unknown_endtag(self, tag):         #(3)
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):         #(4)
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):       #(5)
        self.pieces.append("&%(ref)s" % locals())
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):           #(6)
        self.pieces.append(text)

    def handle_comment(self, text):        #(7)
        self.pieces.append("\begin{comment} %(text)s \end{comment}
" % locals())

    def handle_pi(self, text):             #(8)
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
```

```
        self.pieces.append("<!%(text)s>" % locals())
```

1. `reset`, wołany przez `SGMLParser.__init__`, inicjalizuje `self.pieces` jako pustą listę przed wywołaniem metody klasy przodka. `self.pieces` jest atrybutem, który będzie przechowywał części konstruowanego dokumentu HTML. Każda metoda będzie rekonstruować HTML parsowany przez `SGMLParser` i każda z tych metod będzie dodawać jakiś tekst do `self.pieces`. Zauważmy, że `self.pieces` jest listą. Moglibyśmy ulec pokusie, aby zdefiniować ten atrybut jako obiekt łańcucha znaków i po prostu dołączać do niego kolejne kawałki tekstu. To także by działało, ale Python jest dużo bardziej wydajny pracując z listami. [1]

2. Ponieważ `BaseHTMLProcessor` nie definiuje żadnej metody dla poszczególnych znaczników (jak np. metoda `start_a` w `URLLister`), `SGMLParser` będzie wywoływał dla każdego początkowego znacznika metodę `unknown_starttag`. Ta metoda przyjmuje na wejściu znacznik (argument `tag`) i listę par postaci nazwa atrybutu/wartość atrybutu (argument `attrs`), a następnie rekonstruuje oryginalnego HTML-a i dodaje do `self.pieces`. Napis formatujący jest tutaj nieco dziwny; rozwikłamy to później w tym rozdziale (a także tą dziwnie wyglądającą funkcję `locals`).

3. Rekonstrukcja znaczników końcowych jest dużo prostsza; po prostu pobieramy nazwę znacznika i opakowujemy nawiasami ostrymi `</...>`.

4. Gdy `SGMLParser` napotka odwołanie znakowe wywołuje metodę `handle_charref` i przekazuje jej samą wartość odwołania. Jeśli dokument HTML zawiera ` `, `ref` przyjmie wartość `160`. Rekonstrukcja oryginalnego kompletnego odwołania znakowego wymaga po prostu dodania znaków `&#...;`.

5. Odwołanie do encji jest podobne do odwołania znakowego, ale nie zawiera znaku kratki (`#`). Rekonstrukcja oryginalnego odwołania do encji wymaga dodania znaków `&;...;`. (Właściwie, jak wskazał na to pewien czytelnik, jest to nieco bardziej skomplikowane. Tylko niektóre standardowe encje HTML-a kończą się znakiem średnika; inne podobnie wyglądające encje już nie. Na szczęście dla nas zbiór standardowych encji HTML-a zdefiniowany jest w Pythonie w słowniku w module o nazwie `htmlentitydefs`. Stąd ta dodatkowa instrukcja `if`.)

6. Bloki tekstu są po prostu dołączane do `self.pieces` bez żadnych zmian, w postaci dosłownej.

7. Komentarze HTML-a opakowywane są znakami `<!--...-->`.

---

[1] Powodem dla którego Python jest lepszy w pracy z listami niż napisami, jest fakt iż listy są modyfikowalne (mutable), a napisy są niemodyfikowalne (immutable). Co oznacza, że zwiększeniem listy jest dodanie do niej po prostu nowego elementu i zaktualizowanie indeksu. Natomiast ponieważ napis nie może być zmieniony po utworzeniu, z reguły kod `s = s + nowy` utworzy całkowicie nowy napis powstały z połączenia oryginalnego napisu `s` i napisu `nowy`, a oryginalny napis zostanie zniszczony. To wymaga wielu kosztownych operacji zarządzania pamięcią, a wielkość zaangażowanego wysiłku rośnie wraz z długością napisu, a więc wykonywanie kodu `s = s + nowy` w pętli jest zabójcze. W terminologii technicznej dodanie n elementów do listy oznacza złożoność `O(n)`, podczas gdy dodanie `n` elementów do napisu złożoność $O(n_2)$. Z drugiej strony Python korzysta z prostej optymalizacji, polegającej na tym, że jeśli dany łańcuch znaków posiada tylko jedno odwołanie, to nie tworzy nowego łańcucha, tylko rozszerza stary i akurat w tym przypadku byłoby nieco szybciej na łańcuchach znaków (wówczas złożoność byłaby `O(n)`).

8. Instrukcje przetwarzania wstawiane są pomiędzy znakami `<?...>`.

Ważne Specyfikacja HTML-a wymaga, aby wszystkie nie-HTML-owe elementy (jak np. JavaScript) były zawarte pomiędzy HTML-owymi znakami komentarza, ale nie wszystkie strony internetowe robią to właściwie (a współczesne przeglądarki internetowe są wyrozumiałe w tym względzie). `BaseHTMLProcessor` natomiast nie jest wyrozumiały; jeśli skrypt jest osadzony niewłaściwie, będzie on sparsowany tak, jakby był HTML-em. Np. jeśli skrypt zawiera znaki mniejszości i równości `SGMLParser` może błędnie pomyśleć, że znalazł znaczniki i atrybuty. `SGMLParser` zawsze konwertuje nazwy znaczników i atrybutów na małe znaki, co może zepsuć działanie skryptu i `BaseHTMLProcessor` zawsze otacza wartości atrybutów znakami cudzysłowu (nawet jeśli oryginalny dokument HTML używał apostrofów lub nie używał niczego), co już na pewno zepsuje skrypt. Zawsze chroń swój skrypt osadzony w HTML-u znakami komentarza.

Przykład 8.9 `BaseHTMLProcessor` **i jego metoda** `output`

```
def output(self):               #(1)
    u"""Zwraca przetworzony HTML jako pojedynczy łańcuch znaków"""
    return "".join(self.pieces)
```

1. To jest jedyna metoda, która nie jest wołana przez klasę przodka, czyli klasę `SGMLParser`. Ponieważ pozostałe metody umieszczają swoje zrekonstruowane kawałki HTML-a w `self.pieces`, ta funkcja jest potrzebna, aby połączyć wszystkie te kawałki w jeden napis. Gdyż jak już wspomniano wcześniej Python jest świetny w obsłudze list i z reguły mierny w obsłudze napisów, kompletny napis wyjściowy tworzony jest tylko wtedy, gdy ktoś o to wyraźnie poprosi.

Materiały dodatkowe

- W3C omawia odwołania znakowe i encje.
- Python Library Reference potwierdza Twoje podejrzenia, iż moduł `htmlentitydefs` jest dokładnie tym na co wygląda.

9.5 locals i globals

`locals` i `globals`

Odejdźmy teraz na minutkę od przetwarzania HTML-a. Porozmawiajmy o tym, jak Python obchodzi się ze zmiennymi. Python posiada dwie wbudowane funkcje, `locals` i `globals`, które pozwalają nam uzyskać w słownikowy sposób dostęp do zmiennych lokalnych i globalnych.

Pamiętasz `locals`? Pierwszy raz mogliśmy ją zobaczyć tutaj:

```
def unknown_starttag(self, tag, attrs):
    strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
    self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
```

Nie, czekaj, nie możesz jeszcze się uczyć o `locals`. Najpierw, musisz nauczyć się, czym są przestrzenie nazw. Przedstawimy teraz trochę suchego materiału, lecz ważnego, dlatego też zachowaj uwagę.

Python korzysta z czegoś, co się nazywa przestrzenią nazw (ang. *namespace*), aby śledzić zmienne. Przestrzeń nazw jest właściwie słownikiem, gdzie kluczami są nazwy zmiennych, a wartościami słownika są wartości tych zmiennych. Możemy dostać się do przestrzeni nazw, jak do Pythonowego słownika, co zresztą zobaczymy za chwilkę.

Z dowolnego miejsca Pythonowego programu mamy dostęp do kliku przestrzeni nazw. Każda funkcja posiada własną przestrzeń nazw, nazywaną *lokalną przestrzenią nazw*, a która śledzi zmienne funkcji, włączając w to jej argumenty i lokalnie zdefiniowane zmienne. Każdy moduł posiada własną przestrzeń nazw, nazwaną *globalną przestrzenią nazw*, a która śledzi zmienne modułu, włączając w to funkcje, klasy i inne zaimportowane moduły, a także zmienne zdefiniowane w tym module i stałe. Jest także wbudowana przestrzeń nazw, dostępna z każdego modułu, a która przechowuje funkcje wbudowane i wyjątki.

Kiedy pewna linia kodu pyta się o wartość zmiennej `x`, Python przeszuka wszystkie przestrzenie nazw, aby ją znaleźć, w poniższym porządku:

1. lokalna przestrzeń nazw – określona dla bieżącej funkcji lub metody pewnej klasy. Jeśli funkcja definiuje jakąś lokalną zmienną `x`, Python wykorzysta ją i zakończy szukanie.

2. przestrzeni nazw, w której dana funkcja została zagnieżdżona i przestrzeniach nazw, które znajdują się wyżej w "zagnieżdżonej" hierarchii.

3. globalna przestrzeń nazw – określona dla bieżącego modułu. Jeśli moduł definiuje zmienną lub klasę o nazwie `x`, Python wykorzysta ją i zakończy szukanie.

4. wbudowana przestrzeń nazw – globalna dla wszystkich modułów. Ponieważ jest to ostatnia deska ratunku, Python przyjmie, że `x` jest nazwą wbudowanej funkcji lub zmiennej.

Jeśli Python nie znajdzie `x` w żadnej z tych przestrzeni nazw, poddaje się i wyrzuca wyjątek `NameError` z wiadomością "name 'x' is not defined", którą zobaczyliśmy w przykładzie 3.21, lecz nie jesteśmy w stanie ocenić, jak Python zadziała, zanim dostaniemy ten błąd.

Python korzysta z zagnieżdżonych przestrzeni nazw (ang. *nested scope*). Jeśli tworzymy wewnątrz pewnej funkcji inną funkcję, Python stworzy nową zagnieżdżoną przestrzeń nazw dla zagnieżdżonej funkcji, która jednocześnie będzie lokalną przestrzenią nazw, ale jednocześnie będziemy mogli korzystać z przestrzeni nazw funkcji, w której daną funkcję zagnieżdżamy i pozostałych przestrzeni nazw (globalnej i wbudowanej).

Zmieszałeś się? Nie panikuj! Jest to naprawdę wypaśne. Podobnie, jak wiele rzeczy w Pythonie, przestrzenie nazw są bezpośrednio dostępne podczas wykonywania programu. Jak? Do lokalnej przestrzeni nazw mamy dostęp poprzez wbudowaną funkcję `locals`, a globalna (na poziomie modułu) przestrzeń nazw jest dostępna poprzez wbudowaną funkcję `globals`.

Przykład 8.10 Wprowadzenie do `locals`

```
>>> def foo(arg): #(1)
...     x = 1
...     print locals()
...
>>> foo(7)        #(2)
{'arg': 7, 'x': 1}
>>> foo('bar')    #(3)
{'arg': 'bar', 'x': 1}
```

1. Funkcja `foo` posiada dwie zmienne w swojej lokalnej przestrzeni nazw: `arg`, której wartość jest przekazana do funkcji, a także `x`, która jest zdefiniowana wewnątrz funkcji.

2. `locals` zwraca słownik par nazwa/wartość. Kluczami słownika są nazwy zmiennych w postaci napisów. Wartościami słownika są bieżące wartości tych zmiennych. Zatem wywołując `foo` z `7`, wypiszemy słownik zawierający dwie lokalne zmienne tej funkcji, czyli `arg` (o wartości `7`) i `x` (o wartości `1`).

3. Pamiętaj, Python jest dynamicznie typowany, dlatego też możemy w prosty sposób jako argument `arg` przekazać napis. Funkcja (a także wywołanie `locals`) będą nadal działać jak należy. `locals` działa z wszystkimi zmiennymi dowolnych typów danych.

To co `locals` robi dla lokalnej (należącej do funkcji) przestrzeni nazw, `globals` robi dla globalnej (modułu) przestrzeni nazw. `globals` jest bardziej ekscytujące, ponieważ przestrzeń nazw modułu jest bardziej pasjonująca [2]. Przestrzeń nazw modułu nie tylko przechowuje zmienne i stałe na poziomie tego modułu, lecz także funkcje i klasy zdefiniowane w tym module. Ponadto dołączone do tego jest cokolwiek, co zostało zaimportowane do tego modułu.

Pamiętasz różnicę między `from module import`, a `import module`? Za pomocą `import module`, zaimportujemy sam moduł, który zachowa własną przestrzeń nazw, a to jest przyczyną, dlaczego musimy odwołać się do nazwy modułu, aby dostać się do jakiejś funkcji lub atrybutu (pisząc `module.function`). Z kolei za pomocą `from module import` rzeczywiście importujemy do własnej przestrzeni nazw określoną funkcje i atrybuty z innego modułu, a dzięki temu odwołujemy się do niego bezpośrednio,

[2] To zdanie za wiele nie wnosi.

bez wskazywania modułu, z którego one pochodzą. Dzięki funkcji `globals` możemy zobaczyć, że rzeczywiście tak jest.

Spójrzmy na poniższy blok kodu, który znajduje się na dole BaseHTMLProcessor.py.

Przykład 8.11 Wprowadzenie do `globals`

```
if __name__ == "__main__":
    for k, v in globals().items():             #(1)
        print k, "=", v
```

1. Na wypadek gdyby wydawało Ci się to straszne, to pamiętaj, że widzieliśmy to już wcześniej. Funkcja `globals` zwraca słownik, po którym następnie iterujemy słownik wykorzystując metodę `items` i wielozmienne przypisanie. Jedyną nową rzeczą jest funkcja `globals`.

Teraz, uruchamiając skrypt z linii poleceń otrzymamy takie wyjście (twoje wyjście może się nieco różnić, jest zależne od systemu i miejsca instalacji Pythona):

```
c:\docbook\dip\py> python BaseHTMLProcessor.py

SGMLParser = sgmllib.SGMLParser                #(1)
htmlentitydefs = <module 'htmlentitydefs' from 'C:\Python23\lib\htmlentitydefs.py'> #(2)
BaseHTMLProcessor = __main__.BaseHTMLProcessor #(3)
__name__ = __main__                            #(4)
[...ciach ...]
```

1. `SGMLParser` został zaimportowany z `sgmllib`, wykorzystując `from module import`. Oznacza to, że został zaimportowany bezpośrednio do przestrzeni nazw modułu i w tym też miejscu jest.

2. W przeciwieństwie do `SGMLParser`a, `htmlentitydefs` został zaimportowany wykorzystując instrukcję `import`. Oznacza to, że moduł `htmlentitydefs` sam w sobie jest przestrzenią nazw, ale zmienna `entitydefs` wewnątrz `htmlentitydefs` już nie.

3. Moduł ten definiuje jedną klasę, `BaseHTMLProcessor` i oto ona. Dodajmy, że ta wartość jest klasą samą w sobie, a nie jakąś specyficzną instancją tej klasy.

4. Pamiętasz trik `if __name__`? Kiedy uruchamiamy moduł (zamiast importować go z innego modułu), to wbudowany atrybut `__name__` ma specjalną wartość, `"__main__"`. Ponieważ uruchomiliśmy ten moduł jako skrypt z linii poleceń, wartość `__name__` wynosi `"__main__"`, dlatego też zostanie wykonany mały kod testowy, który wypisuje `globals`.

Korzystając z funkcji `locals` i `globals` możemy pobrać dynamicznie wartość dowolnej zmiennej, dzięki przekazaniu nazwy zmiennej w postaci napisu. Funkcjonalność ta jest analogiczna do `getattr`, która pozwala dostać się do dowolnej funkcji, dzięki przekazaniu jej nazwy w postaci napisu.

Poniżej pokażemy inną ważną różnicę między funkcjami `locals` i `globals`, a o której powinieniśmy się dowiedzieć, zanim nas to ukąsi. Jakkolwiek to i tak Ciebie ukąsi, ale przynajmniej będziesz pamiętał, że była o tym mowa w tym podręczniku.

Przykład 8.12 `locals` jest tylko do odczytu, a `globals` już nie

```
def foo(arg):
    x = 1
    print locals()    #(1)
    locals()["x"] = 2 #(2)
    print "x=",x      #(3)

z = 7
print "z=",z
foo(3)
globals()["z"] = 8    #(4)
print "z=",z          #(5)
```

1. Ponieważ `foo` zostało wywołane z argumentem `3`, więc zostanie wypisane {`'arg': 3, 'x': 1`}. Nie powinno to być zaskoczeniem.

2. `locals` jest funkcją zwracającą słownik i w tym miejscu zmieniamy wartość w tym słowniku. Możemy myśleć, że wartość zmiennej `x` zostanie zmieniona na `2`, jednak tak nie będzie. `locals` właściwie nie zwraca lokalnej przestrzeni nazw, zwraca jego kopię. Zatem zmieniając ją, nie zmieniamy wartości zmiennych w lokalnej przestrzeni nazw.

3. Zostanie wypisane `x= 1`, a nie `x= 2`.

4. Po tym, jak zostaliśmy poparzeni przez `locals`, możemy myśleć, że ta operacja nie zmieni wartości `z`, ale w rzeczywistości zmieni. W skutek wewnętrznych różnic implementacyjnych [3], `globals` zwraca aktualną, globalną przestrzeń nazw, a nie jej kopię; całkowicie odwrotne zachowanie w stosunku do `locals`. Tak więc dowolna zmiana zwróconego przez `globals` słownika bezpośrednio wpływa na zmienne globalne.

5. Wypisze `z= 8`, a nie `z= 7`.

[3] Nie będziemy się wdawać w szczegóły

9.6 Formatowanie napisów w oparciu o słowniki

Formatowanie napisów w oparciu o słowniki

Dlaczego uczyliśmy się na temat funkcji `locals` i `globals`? Ponieważ teraz możemy się nauczyć formatowania napisów w oparciu o słowniki. Jak już mówiliśmy, regularne formatowanie napisów umożliwia w łatwy sposób wstawianie wartości do napisów. Wartości są wyszczególnione w krotce i w odpowiednim porządku wstawione do napisu, gdzie występuje pole formatujące. O ile jest to skuteczne, nie zawsze tworzy kod łatwy do czytania, zwłaszcza, gdy zostaje wstawianych wiele wartości. Żeby zrozumieć o co chodzi, nie wystarczy po prostu jednorazowo prześledzić napis; trzeba ciągle skakać między czytanym napisem, a czytaną krotką wartości.

Tutaj mamy alternatywną formę formatowania napisu, a która zamiast krotek wykorzystuje słowniki.

Przykład 8.13 Formatowanie napisów w oparciu o słowniki

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> "%(pwd)s" % params                                  #(1)
'secret'
>>> "%(pwd)s nie jest poprawnym hasłem dla %(uid)s" % params #(2)
'secret nie jest poprawnym hasłem dla sa'
>>> "%(database)s of mind, %(database)s of body" % params #(3)
'master of mind, master of body'
```

1. Zamiast korzystać z krotki wartości, formujemy napis formatujący, który korzysta ze słownika `params`. Ponadto zamiast prostego pola `%s` w napisie, pole zawiera nazwę w nawiasach okrągłych. Nazwa ta jest wykorzystana jako klucz w słowniku `params` i zostaje zastąpione odpowiednią wartością, `secret`, w miejscu wystąpienia pola `%(pwd)s`.

2. Takie formatowanie może posiadać dowolną liczbę odwołań do kluczy. Każdy klucz musi istnieć w podanym słowniku, ponieważ inaczej formatowanie zakończy się niepowodzeniem i zostanie rzucony wyjątek `KeyError`.

3. Możemy nawet wykorzystać ten sam klucz kilka razy. Każde wystąpienie zostanie zastąpione odpowiednią wartością.

Zatem dlaczego używać formatowania napisu w oparciu o słowniki? Może to wyglądać na nadmierne wmieszanie słownika z kluczami i wartościami, aby wykonać proste formatowanie napisu. W rzeczywistości jest bardzo przydatne, kiedy już się ma słownik z kluczami o sensownych nazwach i wartościach, jak np. `locals`.

Przykład 8.14 Formatowanie napisu w BaseHTMLProcessor.py

```
def handle_comment(self, text):
        self.pieces.append("<!--%(text)s-->" % locals()) #(1)
```

1. Formatowanie za pomocą słowników jest powszechnie używane z wbudowaną funkcję `locals`. Oznacza to, że możemy wykorzystywać nazwy zmiennych lokalnych wewnątrz napisu formatującego (w tym przypadku `text`, który został przykazany jako argument do metody klasy) i każda nazwa zmiennej zostanie

zastąpiona jej wartością. Jeśli `text` przechowuje wartość `'Początek stopki'`, formatowany napis `«!--%(text)s-->"% locals()` zostanie wygenerowany jako `'<!--Początek stopki-->'`.

Przykład 8.15 Więcej formatowania opartego na słownikach

```
def unknown_starttag(self, tag, attrs):
    strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])   #(1)
    self.pieces.append("<%(tag)s%(strattrs)s>" % locals())                    #(2)
```

1. Kiedy metoda ta zostaje wywołana, `attrs` jest listą krotek postaci klucz/wartość, podobnie jak zwrócona wartość metody słownika `items`, a to oznacza, że możemy wykorzystać wielozmienne przypisanie, aby wykonać na niej iterację. Powinniśmy już być zaznajomieni z tymi operacjami, ale jest tu tego trochę dużo, więc prześledźmy je po kolei:

   (a) Przypuśćmy, że `attrs` wynosi [('href', 'index.html'), ('title', 'Idź do strony domowej')].

   (b) W pierwszym przebiegu odwzorowywania listy, `key` przyjmie wartość `'href'`, a `value` weźmie wartość `'index.html'`.

   (c) Formatowanie napisu `' %s=''%s''' % (key, value)` przekształci się na `' href=''index.html'''`. Napis ten będzie pierwszym elementem zwróconej listy.

   (d) W drugim przebiegu, `key` przyjmie wartość `'title'`, a `value` wartość `'Idź do strony domowej'`.

   (e) Formatowanie napisu przekształci to na `' title=''Idź do strony domowej'''`.

   (f) Po wykonaniu wyrażenia listowego, zwrócona lista będzie przechowywała te dwa wygenerowane napisy, a `strattrs` będzie połączeniem obydwu tych elementów, czyli będzie przechowywał `' href=''index.html'' title=''Go to home page'''`.

2. Teraz formatując napis za pomocą słownika, wstawiamy wartość zmiennej `tag` i `strattrs` do napisu. Zatem jeśli `tag` wynosił `'a'`, w ostateczności otrzymamy wynik `'<a href="index.html"title="Idź do strony domowej'»'` i to następnie dodajemy do `self.pieces`.

Korzystanie z słownikowego formatowania napisu i funkcji `locals` jest wygodnym sposobem, aby tworzyć czytelniejsze skomplikowane wyrażenia listowe, lecz trzeba zapłacić pewną cenę. Jest tutaj drobny narzut wydajności związany z wywołaniem funkcji `locals`, ponieważ `locals` wykonuje kopię lokalnej przestrzeni nazw.

9.7 Dodawanie cudzysłowów do wartości atrybutów

Dodawanie cudzysłowów do wartości atrybutów

Dość powszechnym pytaniem na comp.lang.python jest "Mam kilka dokumentów HTML z wartościami atrybutów bez cudzysłowów i chciałbym odpowiednio te cudzysłowy dodać. Jak mogę to zrobić?" [4] (Przeważnie wynika to z dołączenia do projektu nowego kierownika, będącego wyznawcą HTML-owych standardów i bezwzględnie wymagającego, aby wszystkie strony bezbłędnie przechodziły kontrolę HTML-owych walidatorów. Wartości atrybutów bez cudzysłowów są powszechnym naruszeniem HTML-wego standardu.) Niezależnie od powodu, uzupełnienie cudzysłowów jest łatwe przy pomocy klasy `BaseHTMLProcessor`.

`BaseHTMLProcessor` konsumuje HTML-a (ponieważ jest potomkiem klasy `SGMLParser`) i produkuje równoważny HTML, ale ten wyjściowy HTML nie jest identyczny z wejściowym. Znaczniki i nazwy atrybutów zostaną zapisane małymi literami, nawet jeśli wcześniej były dużymi lub wymieszanymi, a wartości atrybutów zostaną zamknięte w podwójnych cudzysłowach, nawet jeśli wcześniej były otoczone pojedynczymi cudzysłowami lub nie miały żadnych cudzysłowów. To jest taki efekt uboczny, z którego możemy tu skorzystać.

Przykład 8.16 Dodawanie cudzysłowów do wartości atrybutów

```
>>> htmlSource = """        #(1)
...     <html>
...     <head>
...     <title>Test page</title>
...     </head>
...     <body>
...     <ul>
...     <li><a href=index.html>Strona główna</a></li>
...     <li><a href=toc.html>Spis treści</a></li>
...     <li><a href=history.html>Historia zmian</a></li>
...     </body>
...     </html>
...     """
>>> from BaseHTMLProcessor import BaseHTMLProcessor
>>> parser = BaseHTMLProcessor()
>>> parser.feed(htmlSource) #(2)
>>> print parser.output()   #(3)
<html>
<head>
<title>Test page</title>
</head>
<body>
<ul>
<li><a href="index.html">Strona główna</a></li>
```

[4] No dobra, to nie jest aż tak powszechne pytanie. Nie jest częstsze niż "Jakiego edytora powinienem używać do pisania kodu w Pythonie?" (odpowiedź: Emacs) lub "Python jest lepszy czy gorszy od Perla?" (odpowiedź: "Perl jest gorszy od Pythona, ponieważ ludzie chcieli aby był gorszy." -Larry Wall, 10/14/1998). Jednak pytania o przetwarzanie HTML-a pojawiają się w takiej czy innej formie około raz na miesiąc i wśród tych pytań, to jest dość popularne.

```
<li><a href="toc.html">Spis treści</a></li>
<li><a href="history.html">Historia zmian</a></li>
</body>
</html>
```

1. Zauważmy, że wartości atrybutów `href` w znacznikach `<a>` nie są ograniczone cudzysłowami. (Jednocześnie zauważmy, że używamy potrójnych cudzysłowów do czegoś innego niż notki dokumentacyjnej i to bezpośrednio w IDE. Są one bardzo użyteczne.)

2. "Karmimy" parser.

3. Używając funkcji `output` zdefiniowanej w klasie `BaseHTMLProcessor`, otrzymujemy wyjście jako pojedynczy kompletny łańcuch znaków ze wszystkimi wartościami atrybutów w cudzysłowach. Pomyślmy, jak wiele właściwie się tutaj działo: `SGMLParser` sparsował cały dokument HTML, podzielił go na znaczniki, odwołania, dane tekstowe itp.; `BaseHTMLProcessor` użył tych elementów do zrekonstruowania części HTML-a (które nadal są składowane w `parser.pieces`, jeśli chcesz je zobaczyć); na końcu wywołaliśmy `parser.output`, która to metoda połączyła wszystkie części HTML-a w jeden napis.

9.8 Wprowadzenie do dialect.py

Dialectizer jest prostym (i niezbyt mądrym) potomkiem klasy BaseHTMLProcessor. Dokonuje on na bloku tekstu serii podstawień, ale wszystko co znajduje się wewnątrz bloku <pre>...</pre> pozostawia niezmienione.

Aby obsłużyć bloki <pre> definiujemy w klasie Dialectizer metody: start_pre i end_pre.

Przykład 8.17 Obsługa określonych znaczników

```
def start_pre(self, attrs):             #(1)
    self.verbatim += 1                  #(2)
    self.unknown_starttag("pre", attrs) #(3)

def end_pre(self):                      #(4)
    self.unknown_endtag("pre")          #(5)
    self.verbatim -= 1                  #(6)
```

1. start_pre jest wywoływany za każdym razem, gdy SGMLParser znajdzie znacznik <pre> w źródle HTML-a. (Za chwilę zobaczymy dokładnie, jak to się dzieje.) Ta metoda przyjmuje jeden parametr: attrs, który zawiera atrybuty znacznika (jeśli jakieś są). attrs jest listą krotek postaci klucz/wartość, taką samą jaką przyjmuje unknown_starttag.

2. W metodzie reset, inicjalizujemy atrybut, który służy jako licznik znaczników <pre>. Za każdym razem, gdy natrafiamy na znacznik <pre>, zwiększamy licznik, natomiast gdy natrafiamy na znacznik </pre> zmniejszamy licznik. (Moglibyśmy też użyć po prostu flagi i ustawiać ją na wartość True, a następnie False, ale nasz sposób jest równie łatwy, a dodatkowo obsługujemy dziwny (ale możliwy) przypadek zagnieżdżonych znaczników <pre>.) Za chwilę zobaczymy jak można wykorzystać ten licznik.

3. To jest ta jedyna akcja wykonywana dla znaczników <pre>. Przekazujemy tu listę atrybutów do metody unknown_starttag, aby wykonała ona domyślną akcję.

4. Metoda end_pre jest wywoływana za każdym razem, gdy SGMLParser znajdzie znacznik </pre>. Ponieważ znaczniki końcowe nie mogą mieć atrybutów, ta metoda nie przyjmuje żadnych parametrów.

5. Po pierwsze, chcemy wykonać domyślną akcję dla znacznika końcowego.

6. Po drugie, zmniejszamy nasz licznik, co sygnalizuje nam zamknięcie bloku <pre>.

W tym momencie warto się zagłębić nieco bardziej w klasę SGMLParser. Wielokrotnie stwierdzaliśmy, że SGMLParser wyszukuje i wywołuje specyficzne metody dla każdego znacznika, jeśli takowe istnieją. Na przykład właśnie zobaczyliśmy definicje metod start_pre i end_pre do obsługi <pre> i </pre>. Ale jak to się dzieje? No cóż, to nie jest żadna magia. To jest po prostu dobry kawałek kodu w Pythonie.

Przykład 8.18 SGMLParser

```
def finish_starttag(self, tag, attrs):              #(1)
    try:
        method = getattr(self, 'start_' + tag)      #(2)
    except AttributeError:                          #(3)
        try:
            method = getattr(self, 'do_' + tag)     #(4)
        except AttributeError:
            self.unknown_starttag(tag, attrs)       #(5)
            return -1
        else:
            self.handle_starttag(tag, method, attrs) #(6)
            return 0
    else:
        self.stack.append(tag)
        self.handle_starttag(tag, method, attrs)
        return 1                                    #(7)

def handle_starttag(self, tag, method, attrs):
    method(attrs)                                   #(8)
```

1. W tym momencie `SGMLParser` znalazł już początkowy znacznik i sparsował listę atrybutów. Ostatnia rzecz jaka została do zrobienia, to ustalenie czy istnieje specjalna metoda obsługi dla tego znacznika lub czy powinniśmy skorzystać z metody domyślnej (`unknown_starttag`).

2. Za "magią" klasy `SGMLParser` nie kryje się nic więcej niż nasz stary przyjaciel `getattr`. Może jeszcze tego wcześniej nie zauważyliśmy, ale `getattr` poszukuje metod zdefiniowanych zarówno w danym obiekcie jak i w jego potomkach. Tutaj obiektem jest `self`, czyli bieżąca instancja. A więc jeśli `tag` przyjmie wartość `'pre'`, to wywołanie `getattr` będzie poszukiwało metody `start_pre` w bieżącej instancji, którą jest instancja klasy `Dialectizer`.

3. Metoda `getattr` rzuca wyjątek `AttributeError`, jeśli metoda, której szuka nie istnieje w danym obiekcie (oraz w żadnym z jego potomków), ale to jest w porządku, ponieważ wywołanie `getattr` zostało otoczone blokiem `try...except` i wyjątek `AttributeError` zostaje przechwycony.

4. Ponieważ nie znaleźliśmy metody `start_xxx`, sprawdzamy jeszcze metodę `do_xxx` zanim się poddamy. Ta alternatywna grupa metod generalnie służy do obsługi znaczników samodzielnych, jak np. `<br>`, które nie mają znacznika końcowego. Jednak możemy używać metod z obu grup. Jak widać `SGMLParser` sprawdza obie grupy dla każdego znacznika. (Jednak nie powinieneś definiować obu metod obsługi `start_xxx` i `do_xxx` dla tego samego znacznika; wtedy i tak zostanie wywołana tylko `start_xxx`.)

5. Następny wyjątek `AttributeError`, który oznacza, że kolejne wywołanie `getattr` odnoszące się doe `do_xxx` także zawiodło. Ponieważ nie znaleźliśmy ani metody `start_xxx`, ani `do_xxx` dla tego znacznika, przechwytujemy wyjątek i wycofujemy się do metody domyślnej `unknown_starttag`.

6. Pamiętajmy, bloki `try...except` mogą mieć także klauzulę `else`, która jest wywoływana jeśli nie wystąpi żaden wyjątek wewnątrz bloku `try...except`. Logiczne, to oznacza, że znaleźliśmy metodę `do_xxx` dla tego znacznika, a więc wywołujemy ją.

7. A tak przy okazji nie przejmuj się tymi różnymi zwracanymi wartościami; teoretycznie one coś oznaczają, ale w praktyce nie są wykorzystywane. Nie martw się także tym `self.stack.append(tag)`; `SGMLParser` śledzi samodzielnie, czy znaczniki początkowe są zrównoważone z odpowiednimi znacznikami końcowymi, ale jednocześnie do niczego tej informacji nie wykorzystuje. Teoretycznie moglibyśmy wykorzystać ten moduł do sprawdzania, czy znaczniki są całkowicie zrównoważone, ale prawdopodobnie nie warto i wykracza to poza zakres tego rozdziału. W tej chwili masz lepsze powody do zmartwienia.

8. Metody `start_xxx` i `do_xxx` nie są wywoływane bezpośrednio. Znacznik `tag`, metoda `method` i atrybuty `attrs` są przekazywane do tej funkcji, czyli do `handle_starttag`, aby klasy potomne mogły ją nadpisać i tym samym zmienić sposób obsługi znaczników początkowych. Nie potrzebujemy aż tak niskopoziomowej kontroli, a więc pozwalamy tej metodzie zrobić swoje, czyli wywołać metody (`start_xxx` lub `do_xxx`) z listą atrybutów. Pamiętajmy, argument `method` jest funkcją zwróconą przez `getattr`, a funkcje są obiektami. (Wiem, wiem, zaczynasz mieć dość słuchania tego w kółko. Przestaniemy o tym powtarzać, jak tylko zabraknie sposobów na wykorzystanie tego faktu.) Tutaj obiekt funkcji `method` jest przekazywany do metody jako argument, a ta metoda wywołuje tę funkcję. W tym momencie nie istotne jest co to jest za funkcja, jak się nazywa, gdzie jest zdefiniowana; jedyna rzecz jaka jest ważna, to to że jest ona wywoływana z jednym argumentem, `attrs`.

A teraz wróćmy do naszego początkowego programu: `Dialectizer`. Gdy go zostawiliśmy, byliśmy w trakcie definiowania metod obsługi dla znaczników `<pre>` i `</pre>`. Pozostała już tylko jedna rzecz do zrobienia, a mianowicie przetworzenie bloków tekstu przy pomocy zdefiniowanych podstawień. W tym celu musimy nadpisać metodę `handle_data`.

Przykład 8.19 Nadpisanie metody `handle_data`

```
def handle_data(self, text):                                            #(1)
    self.pieces.append(self.verbatim and text or self.process(text)) #(2)
```

1. Metoda `handle_data` jest wywoływana z tylko jednym argumentem, tekstem do przetworzenia.

2. W klasie nadrzędnej `BaseHTMLProcessor` metoda `handle_data` po prostu dodaje tekst do wyjściowego bufora `self.pieces`. Tutaj zasada działania jest tylko trochę bardziej skomplikowana. Jeśli jesteśmy w bloku `<pre>...</pre>`, `self.verbatim` będzie miało jakąś wartość większą od `0` i tekst trafi do bufora wyjściowego nie zmieniony. W przeciwnym razie wywołujemy oddzielną metodę do wykonania podstawień i rezultat umieszczamy w buforze wyjściowym. Wykorzystujemy tutaj jednolinijkowiec, który wykorzystuje sztuczkę `and-or`.

Już jesteś blisko całkowitego zrozumienia `Dialectizer`. Ostatnim brakującym ogniwem jest sam charakter podstawień w tekście. Jeśli znasz Perla, to wiesz, że kiedy

wymagane są kompleksowe zmiany w tekście, to jedynym prawdziwym rozwiązaniem są wyrażenia regularne. Klasy w dalszej części dialect.py definiuje serię wyrażeń regularnych, które operują na tekście pomiędzy znacznikami HTML. My już mamy przeanalizowany cały rozdział o wyrażeniach regularnych. Zapewne nie masz ochoty znowu mozolić się z wyrażeniami regularnymi, prawda? Już wystarczająco dużo się nauczyliśmy, jak na jeden rozdział.

9.9 Przetwarzanie HTML-a - wszystko razem

Wszystko razem

Nadszedł czas, aby połączyć w całość wiedzę, którą zdobyliśmy do tej pory.

Przykład 8.20 Funkcja `translate`, część 1

```
def translate(url, dialectName="chef"):  #(1)
    import urllib                        #(2)
    sock = urllib.urlopen(url)           #(3)
    htmlSource = sock.read()
    sock.close()
```

1. Funkcja `translate` przyjmuje opcjonalny argument `dialectName`, który jest łańcuchem znaków określającym używany dialekt. Zaraz zobaczymy, jak to jest wykorzystywane.
2. Moment, tam jest ważna instrukcja w tej funkcji! Jest to w pełni dozwolone w Pythonie działanie. Instrukcja `import` używaliśmy zwykle na samym początku programu, aby zaimportowany moduł był dostępny w dowolnym miejscu. Ale możemy także importować moduły w samej funkcji, przez co będą one dostępne tylko z jej poziomu. Jeżeli jakiegoś moduły potrzebujemy użyć tylko w jednej funkcji, jest to najlepszy sposób aby zachować modularność twojego programu. (Docenisz to, gdy okaże się, że twój weekendowy hack wyrósł na ważące 800 linii dzieło sztuki, a ty właśnie zdecydujesz się podzielić to na mniejsze części).
3. Tutaj otwieramy połączenie i do zmiennej `htmlSource` pobieramy źródło HTML spod wskazanego adresu URL.

Przykład 8.21 Funkcja `translate`, część 2: coraz ciekawiej

```
    parserName = "%sDialectizer" % dialectName.capitalize() #(1)
    parserClass = globals()[parserName]                     #(2)
    parser = parserClass()                                  #(3)
```

1. `capitalize` jest metodą łańcucha znaków, z którą się jeszcze nie spotkaliśmy; zmienia ona pierwszy znak na wielką literę, a wszystkie pozostałe znaki na małe litery. W połączeniu z prostym formatowaniem napisu, nazwa dialektu zamieniana jest na nazwę odpowiadającej mu klasy. Jeżeli `dialectName` ma wartość `'chef'`, `parserName` przyjmie wartość `'ChefDialectizer'`.
2. W tym miejscu mamy nazwę klasy (w zmiennej `parserName`) oraz dostęp do globalnej przestrzeni nazw, poprzez słownik `globals()`. Łącząc obie informacje dostajemy referencje do klasy o określonej nazwie. (Pamiętajmy, że klasy są obiektami i mogą być przypisane do zmiennej, jak każdy inny obiekt). Jeżeli `parserName` ma wartość `'ChefDialectizer'`, `parserClass` będzie klasą `ChefDialectizer`.
3. Ostatecznie, mając obiekt klasy (`parserClass`) chcemy zainicjować tę klasę. Wiemy już jak zrobić — po prostu wywołujemy klasę w taki sposób, jakby była to funkcja. Fakt, że klasa jest przechowywana w lokalnej zmiennej nie robi żadnej

różnicy, po prostu wywołujemy lokalną zmienną jak funkcję, i na wyjściu wyskakuje instancja klasy. Jeżeli `parserClass` jest klasą `ChefDialectizer`, parser będzie instancją klasy `ChefDialectizer`.

Zastanawiasz się, ponieważ istnieją tylko 3 klasy `Dialectizer`, dlaczego by nie użyć po prostu instrukcji `case`? (W porządku, w Pythonie nie ma instrukcji `case`, ale zawsze można użyć serii instrukcji `if`). Z jednego powodu: elastyczności programu. Funkcja `translate` nie ma pojęcia, jak wiele zdefiniowaliśmy podklas `Dialectizer`-a. Wyobraźmy sobie, że definiujemy jutro nową klasę `FooDialectizer` — funkcja `translate` zadziała bez przeróbek.

Nawet lepiej – wyobraźmy sobie, że umieszczasz klasę `FooDialectizer` w osobnym module i importujesz ją poprzez `from module import`. Jak wcześniej mogliśmy się przekonać, taka operacja dołączy to do `globals()`, więc funkcja `translate` nadal będzie działać prawidłowo bez konieczności dokonywania modyfikacji, nawet wtedy, gdy `FooDialectizer` znajdzie się w oddzielnym pliku.

Teraz wyobraźmy sobie, że nazwa dialektu pochodzi skądś z poza programu, może z bazy danych lub z wartości wprowadzonej przez użytkownika. Możemy użyć jakąkolwiek ilość pythonowych skryptów po stronie serwera, aby dynamicznie generować strony internetowe; taka funkcja mogłaby przekazać URL i nazwę dialektu (oba w postaci łańcucha znaków) w zapytania żądania strony internetowej, i zwrócić "przetłumaczoną" stronę.

Na koniec wyobraźmy sobie framework `Dialectizer` z wbudowaną obsługą pluginów. Możemy umieścić każdą podklasę `Dialectizer`-a w osobnym pliku pozostawiając jedynie w pliku dialect.py funkcję `translate`. Jeżeli zachowasz stały schemat nazewnictwa klas, funkcja `translate` może dynamicznie importować potrzebną klasę z odpowiedniego pliku, jedynie na podstawie podanej nazwy dialektu. (Dynamicznego importowanie omówimy to w dalszej części tego podręcznika). Aby dodać nowy dialekt, wystarczy, że utworzymy odpowiednio nazwany plik (np. `foodialect.py` zawierający klasę `FooDialectizer`) w katalogu z plug-inami. Wywołując funkcję `translate` z nazwą dialektu `'foo'`, odnajdzie ona moduł `foodialect.py` i automatycznie zaimportuje klasę `FooDialectizer`.

Przykład 8.22 Funkcja `translate`, część 3

```
parser.feed(htmlSource) #(1)
parser.close()          #(2)
return parser.output()  #(3)
```

1. Po tym całym wyobrażaniu sobie, co robiło się już nudne, mamy funkcję `feed`, która przeprowadza całą transformację. Ponieważ całe źródło HTML-a mamy w jednym łańcuchu znaków, więc funkcję `feed` wywołujemy tylko raz. Oczywiście możemy wywoływać ją dowolną ilość razy, a parser za każdym razem przeprowadzi transformację. Na przykład, jeżeli obawiamy się o zużycie pamięci (albo wiemy, że będziemy parsowali naprawdę wielkie strony HTML), możemy umieścić tą funkcję w pętli, w której będziemy odczytywał tylko kawałek HTML-a i karmił nim parser. Efekty będą takie same.

2. Ponieważ funkcja `feed` wykorzystuje wewnętrzny bufor, powinniśmy zawsze po zakończeniu operacji wywołać funkcję `close()` parsera (nawet jeżeli przesłaliśmy parserowi całość za jednym razem). W przeciwnym wypadku możemy stwierdzić, że w otrzymanym wyniku brakuje kilku ostatnich bajtów.

3. Pamiętajmy, że funkcja `output`, którą zdefiniowaliśmy samodzielnie w klasie `BaseHTMLProcessor`, łączy wszystkie zbuforowane kawałki i zwraca całość w postaci pojedynczego łańcucha znaków.

I właśnie w taki sposób, "przetłumaczyliśmy" stronę internetową, podając jedynie jej adres URL i nazwę dialektu.

9.10 Przetwarzanie HTML-a - podsumowanie

Podsumowanie

Python dostarcza potężne narzędzie do operowania na HTML-u — bibliotekę `sgmllib.py`, która obudowuje kod HTML w model obiektowy. Możemy używać tego narzędzia na wiele sposobów:

- parsując HTML w poszukiwania specyficznych informacji
- gromadząc wyniki, np. tak jak to robi URL lister.
- modyfikując strukturę w dowolny sposób, np. dodawać cudzysłowy do atrybutów
- transformując HTML w inny format, poprzez manipulowanie tekstem bez ruszania znaczników, np. tak jak nasz `Dialectizer`

Po tych wszystkich przykładach, powinniśmy umieć wykonywać wszystkie z tych operacji:

- Używać odpowiednio `locals()` i `globals()`, aby dostać się do przestrzeni nazw
- Formatować łańcuchy w oparciu o słowniki

Rozdział 10

Przetwarzanie XML-a

10.1 Przetwarzanie XML-a

Nurkujemy

Kolejne dwa rozdziały są na temat przetwarzania XML-a w Pythonie. Będzie to przydatne, jeśli już wiesz, jak wyglądają dokumenty XML, a które są wykonane ze strukturalnych znaczników określających hierarchię elementów itp. Jeśli nic z tego nie rozumiesz, możesz przeczytać coś na ten temat na Wikipedii.

Nawet jeśli nie interesuje Ciebie temat XML-a i tak dobrze by było przeczytać te rozdziały, ponieważ omawiają one wiele ważnych tematów jak pakiety, argumenty linii poleceń, a także jak wykorzystywać `getattr` jako pośrednik metod.

Bycie magistrem filozofii nie jest wymagane, chociaż jeśli kiedyś spotkaliśmy się z tekstami napisanymi przez Immanuel Kanta, lepiej zrozumiemy przykładowy program, niż jeśli byłbyś specjalistą w czymś przydatnym, jak informatyka.

Mamy dwa sposoby pracy z XML-em. Jeden jest nazywany SAX (*Simple API for XML*), który działa w ten sposób, że czyta przez chwilę dokument XML i wywołuje dla każdego odnalezionego elementu odpowiednie metody. (Jeśli przeczytaliśmy rozdział 8, powinno to wyglądać znajomo, ponieważ w taki sposób pracuje moduł `sgmllib`.) Inny jest nazywany DOM (*Document Object Model*), a pracuje w ten sposób, że jednorazowo czyta cały dokument XML i tworzy wewnętrzną reprezentację, wykorzystując klasy Pythona powiązane w strukturę drzewa. Python posiada standardowe moduły do obydwu sposobów parsowania, ale rozdział ten opisze tylko, jak wykorzystywać DOM.

Poniżej znajduje się kompletny program Pythona, który generuje pseudolosowe wyjście oparte na gramatyce bezkontekstowej zdefiniowanej w formacie XML. Nie przejmujmy się, jeśli nie zrozumieliśmy, co to znaczy. Będziemy głębiej badać zarówno wejście programu, jak i jego wyjście w tym i następnym rozdziale.

Przykład 9.1 kgp/kgp.py

```
u"""Generator Kanta dla Pythona

Generuje pseudofilozofię opartą na gramatyce bezkontekstowej

Użycie: python kgp.py [options] [source]

Opcje:
  -g ..., --grammar=...   używa określonego pliku gramatyki lub adres URL
  -h, --help              wyświetla ten komunikat pomocy
  -d                      wyświetla informacje debugowania podczas parsowania

Przykłady:
  kgp.py                  generuje kilka akapitów z filozofią Kanta
  kgp.py -g husserl.xml   generuje kilka akapitów z filozofią Husserla
  kpg.py "<xref id='paragraph'/>"  generuje akapit Kanta
  kgp.py template.xml     czyta template.xml, aby określić, co ma generować
 """

from xml.dom import minidom
import random
import toolbox
```

```
import sys
import getopt

_debug = 0

class NoSourceError(Exception): pass

class KantGenerator(object):
    u"""generuje pseudofilozofię opartą na gramatyce bezkontekstowej"""

    def __init__(self, grammar, source=None):
        self.loadGrammar(grammar)
        self.loadSource(source and source or self.getDefaultSource())
        self.refresh()

    def _load(self, source):
        u"""wczytuje XML-owe źródłow wejścia, zwraca sparsowany dokument XML

        - adres URL z plikiem XML ("http://diveintopython.org/kant.xml")
        - nazwę lokalnego pliku XML ("~/diveintopython/common/py/kant.xml")
        - standardowe wejście ("-")
        - bieżący dokument XML w postaci łańcucha znaków
        """
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc

    def loadGrammar(self, grammar):
        u"""wczytuje gramatykę bezkontekstową"""
        self.grammar = self._load(grammar)
        self.refs = {}
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref

    def loadSource(self, source):
        u"""wczytuje źródło source"""
        self.source = self._load(source)

    def getDefaultSource(self):
        u"""zgaduje domyślne źródło bieżącej gramatyki

        Domyślnym źródłem będzie jeden z <ref>-ów, do którego nic się
        nie odwołuje. Może brzmi to skomplikowanie, ale tak naprawdę nie jest.
        Przykład: Domyślnym źródłem dla kant.xml jest
        "<ref id='section'/>", ponieważ 'section' jest jednym <ref>-em, który
        nie jest nigdzie <xref>-em w gramatyce.
        W wielu gramatykach, domyślne źródło będzie tworzyło
        najdłuższe (i najbardziej interesujące) wyjście.
        """
```

```
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        if not standaloneXrefs:
            raise NoSourceError, "can't guess source, and no source specified"
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)

    def reset(self):
        u"""resetuje parser"""
        self.pieces = []
        self.capitalizeNextWord = 0

    def refresh(self):
        u"""resetuje bufor wyjściowy, ponownie parsuje cały plik źródłowy
        i zwraca wyjście

        Ponieważ parsowanie dosyć dużo korzysta z przypadkowości, jest to
        łatwy sposób, aby otrzymać nowe wyjście bez potrzeby ponownego wczytywania
        pliku gramatyki.
        """
        self.reset()
        self.parse(self.source)
        return self.output()

    def output(self):
        u"""wyjściowy, wygenerowany tekst"""
        return "".join(self.pieces)

    def randomChildElement(self, node):
        u"""wybiera przypadkowy potomek węzła

        Jest to użyteczna funkcja wykorzystywana w do_xref i do_choice.
        """
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE]
        chosen = random.choice(choices)
        if _debug:
            sys.stderr.write('%s available choices: %s\n' % \
                (len(choices), [e.toxml() for e in choices]))
            sys.stderr.write('Chosen: %s\n' % chosen.toxml())
        return chosen

    def parse(self, node):
        u"""parsuje pojedynczy węzeł XML

        Parsowany dokument XML (from minidom.parse) jest drzewem węzłów
        złożonym z różnych typów. Każdy węzeł reprezentuje instancję
        odpowiadającej jej klasy Pythona (Element dla znacznika, Text
```

```
        dla danych tekstowych, Document dla dokumentu).  Poniższe wyrażenie
        konstruuje nazwę klasy opartej na typie węzła, który parsujemy
        ("parse_Element" dla węzła o typie Element,
        "parse_Text" dla węzła o typie Text itp.), a następnie wywołuje te metody.
        """
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__)
        parseMethod(node)

    def parse_Document(self, node):
        u"""parsuje węzeł dokumentu

        Węzeł dokument sam w sobie nie jest interesujący (przynajmnie dla nas), ale
        jego jedyne dziecko, node.documentElement jest: jest głównym węzłem
        gramatyki.
        """
        self.parse(node.documentElement)

    def parse_Text(self, node):
        u"""parsuje węzeł tekstowy

        Tekst węzła tekstowego jest zazwyczaj dodawany bez zmiany do wyjściowego bufora.
        Jedynym wyjątkiem jest to, że <p class='sentence'> ustawia flagę, aby
        pierwszą literę następnego słowa była wielka. Jeśli ta flaga jest ustawiona,
        pierwszą literę tekstu robimy wielką i resetujemy tę flagę.
        """
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)

    def parse_Element(self, node):
        u"""parsuje element

        XML-owy element odpowiada bieżącemu znacznikowi źródła:
        <xref id='...'>, <p chance='...'>, <choice> itp.
        Każdy typ elementu jest obsługiwany za pomocą odpowiedniej, własnej metody.
        Podobnie jak to robiliśmy w parse(), konstruujemy nazwę metody
        opartej na nazwie elementu ("do_xref" dla znacznika <xref> itp.), a potem
        wywołujemy tę metodę.
        """
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)

    def parse_Comment(self, node):
        u"""parsuje komentarz

        Gramatyka może zawierać komentarze XML, ale my je pominiemy
```

```
        """
        pass

    def do_xref(self, node):
        u"""obsługuje znacznik <xref id='...'>

        Znacznik <xref id='...'> jest odwołaniem do znacznika <ref id='...'>.
        Znacznik <xref id='sentence'/> powoduje to, że zostaje wybrany w przypadkowy spo
        potomek znacznika <ref id='sentence'>.
        """
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))

    def do_p(self, node):
        u"""obsługuje znacznik <p>

        Znacznik <p> jest jądrem gramatyki. Może zawierać niemal
        wszystko: tekst w dowolnej formie, znaczniki <choice>, znaczniki <xref>,
        a nawet inne znaczniki <p>. Jeśli atrybut "class='sentence'" zostanie
        znaleziony, flaga zostaje ustawiona i następne słowo będzie zapisane
        dużą literą.  Jeśli zostanie znaleziony atrybut "chance='X'", to mamy
        X% szansy, że znacznik zostanie wykorzystany
        (i mamy (100-X)% szansy, że zostanie całkowicie pominięty)
        """
        keys = node.attributes.keys()
        if "class" in keys:
            if node.attributes["class"].value == "sentence":
                self.capitalizeNextWord = 1
        if "chance" in keys:
            chance = int(node.attributes["chance"].value)
            doit = (chance > random.randrange(100))
        else:
            doit = 1
        if doit:
            for child in node.childNodes: self.parse(child)

    def do_choice(self, node):
        u"""obsługuje znacznik <choice>

        Znacznik <choice> zawiera jeden lub więcej znaczników <p>. Jeden znacznik <p>
        zostaje wybrany przypadkowo i jest następnie wykorzystywany do generowania
        tekstu wyjściowego.
        """
        self.parse(self.randomChildElement(node))

def usage():
    print __doc__

def main(argv):
    grammar = "kant.xml"
```

```
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage()
            sys.exit()
        elif opt == '-d':
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):
            grammar = arg

    source = "".join(args)
    k = KantGenerator(grammar, source)
    print k.output()

if __name__ == "__main__":
    main(sys.argv[1:])
```

Przykład 9.2 kgp/toolbox.py

```
u"""Różnorodne użyteczne funkcje"""

def openAnything(source):
    u"""URI, nazwa pliku lub łańcuch znaków --> strumień

    Funkcja ta pozwala zdefiniować parser, który przyjmuje dowolne źródło wejścia
    (URL, ścieżkę do lokalnego pliku lub znajdującego się gdzieś w sieci,
    czy też bieżące dane w postaci łańcucha znaków)
    i traktuje je w odpowiedni sposób. Zwracany obiekt będzie zawierał
    wszystkie podstawowe metody odczytu (read, readline, readlines).
    Kiedy już obiekt nie będzie potrzebny, należy go
    zamknąć za pomocą metody .close().

    Przykłady:
    >>> from xml.dom import minidom
    >>> sock = openAnything("http://localhost/kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("c:\\inetpub\\wwwroot\\kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("<ref id='conjunction'><text>and</text><text>or</text></ref>")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    """
```

```
    if hasattr(source, "read"):
        return source

    if source == "-":
        import sys
        return sys.stdin

    # próbuje otworzyć za pomocą modułu urllib (gdy source jest plikiem
    # dostępnym z http, ftp lub URL-a)
    import urllib
    try:
        return urllib.urlopen(source)
    except (IOError, OSError):
        pass

    # próbuje otworzyć za pomocą wbudowanej funkcji open (jeśli source jest ścieżką
    # do lokalnego pliku)
    try:
        return open(source)
    except (IOError, OSError):
        pass

    # traktuje source jako łańcuch znaków
    import StringIO
    return StringIO.StringIO(str(source))
```

Uruchom sam program `kgp.py`, który będzie parsował domyślną, opartą na XML gramatykę w kgp/kant.xml, a następnie wypisze kilka filozoficznych akapitów w stylu Immanuela Kanta.

Przykład 9.3 Przykładowe wyjście kgp/kgp.py

```
[you@localhost kgp]$ python kgp.py
    As is shown in the writings of Hume, our a priori concepts, in
reference to ends, abstract from all content of knowledge; in the study
of space, the discipline of human reason, in accordance with the
principles of philosophy, is the clue to the discovery of the
Transcendental Deduction.  The transcendental aesthetic, in all
theoretical sciences, occupies part of the sphere of human reason
concerning the existence of our ideas in general; still, the
never-ending regress in the series of empirical conditions constitutes
the whole content for the transcendental unity of apperception.  What
we have alone been able to show is that, even as this relates to the
architectonic of human reason, the Ideal may not contradict itself, but
it is still possible that it may be in contradictions with the
employment of the pure employment of our hypothetical judgements, but
natural causes (and I assert that this is the case) prove the validity
of the discipline of pure reason.  As we have already seen, time (and
it is obvious that this is true) proves the validity of time, and the
architectonic of human reason, in the full sense of these terms,
```

```
abstracts from all content of knowledge.  I assert, in the case of the
discipline of practical reason, that the Antinomies are just as
necessary as natural causes, since knowledge of the phenomena is a
posteriori.
    The discipline of human reason, as I have elsewhere shown, is by
its very nature contradictory, but our ideas exclude the possibility of
the Antinomies.  We can deduce that, on the contrary, the pure
employment of philosophy, on the contrary, is by its very nature
contradictory, but our sense perceptions are a representation of, in
the case of space, metaphysics.  The thing in itself is a
representation of philosophy.  Applied logic is the clue to the
discovery of natural causes.  However, what we have alone been able to
show is that our ideas, in other words, should only be used as a canon
for the Ideal, because of our necessary ignorance of the conditions.

[...ciach...]
```

Jest to oczywiście kompletny bełkot. No dobra, nie całkowity bełkot. Jest składniowo i gramatycznie poprawny (chociaż bardzo wielomówny). Niektóre fragmenty mogą być rzeczywiście prawdą (lub przy najmniej z niektórymi Kant by się zgodził), a niektóre są ewidentnie nieprawdziwe, a wiele fragmentów jest po prostu niespójnych. Lecz wszystko jest w stylu Immanuela Kanta.

Interesującą rzeczą w tym programie jest to, że nie ma tu nic, co określa Kanta. Cała zawartość poprzedniego przykładu pochodzi z pliku gramatyki, kgp/kant.xml. Jeśli każemy programowi wykorzystać inny plik gramatyki (który możemy określić z linii poleceń), wyjście będzie kompletnie różne.

Przykład 9.4 Proste wyjście kgp/kgp.py

```
[you@localhost kgp]$ python kgp.py -g binary.xml
00101001
[you@localhost kgp]$ python kgp.py -g binary.xml
10110100
```

10.2 Pakiety

Pakiety

W rzeczywistości przetwarzanie dokumentu XML jest bardzo proste, wystarczy jedna linia kodu. Jednakże, zanim dojdziemy do tej linii kodu, będziemy musieli krótko omówić, czym są pakiety.

Przykład 9.5 Ładowanie dokumentu XML

```
>>> from xml.dom import minidom    #(1)
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')
```

1. Tej składni jeszcze nie widzieliśmy. Wygląda to niemal, jak `from module import`, który znamy i kochamy, ale z "." wygląda na coś wyższego i innego niż proste `import`. Tak na prawdę `xml` jest czymś, co jest znane pod nazwą *pakiet* (ang. *package*), `dom` jest zagnieżdżonym pakietem wewnątrz `xml`-a, a `minidom` jest modułem znajdującym się wewnątrz `xml.dom`.

Brzmi to skomplikowanie, ale tak naprawdę nie jest. Jeśli spojrzymy na konkretną implementację, może nam to pomóc. Pakiet to niewiele więcej niż katalog z modułami, a zagnieżdżone pakiety są podkatalogami. Moduły wewnątrz pakietu (lub zagnieżdżonego pakietu) są nadal zwykłymi plikami `.py` z wyjątkiem tego, że są w podkatalogu, zamiast w głównym katalogu `lib/` instalacji Pythona.

Przykład 9.6 Plikowa struktura pakietu

```
Python21/ katalog główny instalacji Pythona (katalog domowy plików wykonywalnych)
 |
 +--lib/ katalog bibliotek (katalog domowy standardowych modułow)
    |
    +-- xml/ pakiet xml (w rzeczywistości katalog z innymi rzeczami wewnątrz niego)
        |
        +--sax/ pakiet xml.sax (ponownie, po prostu katalog)
        |
        +--dom/ pakiet xml.dom (zawiera minidom.py)
        |
        +--parsers/ pakiet xml.parsers (używany wewnętrznie)
```

Dlatego kiedy powiesz `from xml.dom import minidom`, Python zrozumie to jako "znajdź w katalogu `xml` katalog `dom`, a następnie szukaj tutaj modułu `minidom` i zaimportuj go jako `minidom`". Lecz Python jest nawet mądrzejszy; nie tylko możemy zaimportować cały moduł zawarty wewnątrz pakietu, ale także możemy wybiórczo zaimportować wybrane klasy czy funkcje z modułu znajdującego się wewnątrz pakietu. Możemy także zaimportować sam pakiet jako moduł. Składnia będzie taka sama; Python wywnioskuje, co masz na myśli na podstawie struktury plików pakietu i automatycznie wykona poprawną czynność.

Przykład 9.7 Pakiety także są modułami

```
>>> from xml.dom import minidom          #(1)
>>> minidom
<module 'xml.dom.minidom' from 'C:\Python21\lib\xml\dom\minidom.pyc'>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml.dom.minidom import Element  #(2)
>>> Element
<class xml.dom.minidom.Element at 01095744>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml import dom                  #(3)
>>> dom
<module 'xml.dom' from 'C:\Python21\lib\xml\dom\__init__.pyc'>
>>> import xml                           #(4)
>>> xml
<module 'xml' from 'C:\Python21\lib\xml\__init__.pyc'>
```

1. W tym miejscu importujemy moduł (`minidom`) z zagnieżdżonego pakietu (`xml.dom`). W wyniku tego `minidom` został zaimportowany do naszej przestrzeni nazw. Aby się odwołać do klasy wewnątrz tego modułu (np. `Element`), będziemy musieli nazwę klasy poprzedzić nazwą modułu.
2. Tutaj importujemy klasę (`Element`) z modułu (`minidom`), a ten moduł z zagnieżdżonego pakietu (`xml.dom`). W wyniku tego `Element` został zaimportowany bezpośrednio do naszje przestrzeni nazw. Dodajmy, że nie koliduje to z poprzednim importem; teraz do klasy `Element` możemy się odwoływać na dwa sposoby (lecz nadal jest to ta sama klasa).
3. W tym miejscu importujemy pakiet `dom` (zagnieżdżony pakiet `xml`-a) jako sam w sobie moduł. Dowolny poziom pakietu może być traktowany jako moduł, co zresztą zobaczymy za moment. Może nawet mieć swoje własne atrybuty i metody, tak jak moduły, które widzieliśmy wcześniej.
4. Tutaj importujemy jako moduł główny poziom pakietu `xml`.

Więc jak może pakiet (który na dysku jest katalogiem) zostać zaimportowany i traktowany jako moduł (który jest zawsze plikiem na dysku)? Odpowiedzią jest magiczny plik `__init__.py`. Wiemy, że pakiety nie są po prostu katalogami, ale są one katalogami ze specyficznym plikiem wewnątrz, `__init__.py`. Plik ten definiuje atrybuty i metody tego pakietu. Na przykład `xml.dom` posiada klasę `Node`, która jest zdefiniowana w `xml/dom/__init__.py`. Kiedy importujemy pakiet jako moduł (np. `dom` z `xml`-a), to tak naprawdę importujemy jego plik `__init__.py`.

Pakiet jest katalogiem ze specjalnym plikiem `__init__.py` wewnątrz niego. Plik `__init__.py` definiuje atrybuty i metody pakietu. Nie musi on definiować niczego, może być nawet pustym plikiem, lecz musi istnieć. Jeśli nie istnieje plik `__init__.py`, katalog jest tylko katalogiem, a nie pakietem, więc nie może być zaimportowany lub zawierać modułów, czy też zagnieżdżonych pakietów.

Więc dlaczego męczyć się z pakietami? Umożliwiają one logiczne pogrupowanie powiązanych ze sobą modułów. Zamiast stworzenia pakietu `xml` z wewnętrznymi pakietami `sax` i `dom`, autorzy mogliby umieścić całą funkcjonalność `sax` w `xmlsax.py`, a

całą funkcjonalność `dom` w `xmldom.py`, czy też nawet zamieścić wszystko w pojedynczym module. Jednak byłoby to niewygodne (podczas pisania tego podręcznika pakiet `xml` posiadał prawie 6000 linii kodu) i trudne w zarządzaniu (dzięki oddzielnym plikom źródłowym, wiele osób może równocześnie pracować nad różnymi częściami).

Jeśli kiedykolwiek będziemy planowali napisać wielki podsystem w Pythonie (lub co bardziej prawdopodobne, kiedy zauważymy, że nasz mały podsystem rozrósł się do dużego), zainwestujmy trochę czasu w zaprojektowanie dobrej architektury systemu pakietów. Jest to jedna z wielu rzeczy w Pythonie, w których jest dobry, więc skorzystajmy z tej zalety.

10.3 Parsowanie XML-a

Parsowanie XML

Jak już mówiliśmy, parsowanie XML-a właściwie jest bardzo proste: jedna linijka kodu. Co z tym zrobimy dalej, to już zależy wyłącznie od nas samych.

Przykład 9.8 Ładowanie dokumentu XML (tym razem naprawdę)

```
>>> from xml.dom import minidom                                          #(1)
>>> xmldoc = minidom.parse('~/zanurkuj_w_pythonie/py/kgp/binary.xml')    #(2)
>>> xmldoc                                                               #(3)
<xml.dom.minidom.Document instance at 010BE87C>
>>> print xmldoc.toxml()                                                 #(4)
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

1. Jak już widzieliśmy w poprzednim podrozdziale, ta instrukcja importuje moduł `minidom` z pakietu `xml.dom`.

2. Tutaj jest ta jedna linia kodu, która wykonuje całą robotę: `minidom.parse` pobiera jeden argument i zwraca sparsowaną reprezentację dokumentu XML. Argumentem może być wiele rzeczy; w tym wypadku jest to po prostu nazwa pliku dokumentu XML na lokalnym dysku. (Aby kontynuować musimy zmienić ścieżkę tak, aby wskazywała na katalog, w którym przechowujemy pobrane z sieci przykład.) Możemy także jako parametr przekazać obiekt pliku lub nawet *obiekt plikopodobny* (ang. *file-like object*). Skorzystamy z tej elastyczności później w tym rozdziale.

3. Obiektem zwróconym przez `minidom.parse` jest obiekt `Document`, który jest klasą pochodną klasy `Node`. Ten obiekt `Document` jest korzeniem złożonej struktury drzewiastej połączonych ze sobą obiektów Pythona, która w pełni reprezentuje dokument XML przekazany funkcji `minidom.parse`.

4. `toxml` jest metodą klasy `Node` (a zatem jest też dostępna w obiekcie `Document` otrzymanym z `minidom.parse`). `toxml` wypisuje XML reprezentowany przez dany obiekt `Node`. Dla węzła, którym jest obiekt `Document`, wypisuje ona cały dokument XML.

Skoro już mamy dokument XML w pamięci, możemy zacząć po nim wędrować.

Przykład 9.9 Pobieranie węzłów potomnych

```
>>> xmldoc.childNodes        #(1)
[<DOM Element: grammar at 17538908>]
>>> xmldoc.childNodes[0]     #(2)
<DOM Element: grammar at 17538908>
>>> xmldoc.firstChild        #(3)
<DOM Element: grammar at 17538908>
```

1. Każdy węzeł posiada atrybut `childNodes`, który jest listą obiektów `Node`. Obiekt `Document` zawsze ma tylko jeden węzeł potomny, element główny (korzeń) dokumentu XML (w tym przypadku element `grammar`).
2. Aby dostać się do pierwszego (i w tym wypadku jedynego) węzła potomnego, używamy po prostu zwykłej składni do obsługi list. Pamiętajmy, tu nie dzieje się nic nadzwyczajnego; to jest po prostu zwykła lista Pythona zwykłych pythonowych obiektów.
3. Ponieważ pobieranie pierwszego węzła potomnego danego węzła jest bardzo użyteczną i częstą czynnością, klasa `Node` posiada atrybut `firstChild`, który jest synonimem dla `childNodes[0]`. (Jest też atrybut `lastChild`, który jest synonimem dla `childNodes[-1]`.)

Przykład 9.10 Metoda `toxml` działa w każdym węźle

```
>>> grammarNode = xmldoc.firstChild
>>> print grammarNode.toxml()       #(1)
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

1. Ponieważ metoda `toxml` jest zdefiniowana w klasie `Node`, jest ona dostępna w każdym węźle XML-a, nie tylko w elemencie `Document`.

Przykład 9.11 Węzłami potomnymi może być także tekst

```
>>> grammarNode.childNodes                  #(1)
[<DOM Text node "\n">, <DOM Element: ref at 17533332>, \
<DOM Text node "\n">, <DOM Element: ref at 17549660>, <DOM Text node "\n">]
>>> print grammarNode.firstChild.toxml()    #(2)



>>> print grammarNode.childNodes[1].toxml() #(3)
<ref id="bit">
```

```
  <p>0</p>
  <p>1</p>
</ref>
>>> print grammarNode.childNodes[3].toxml() #(4)
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
>>> print grammarNode.lastChild.toxml()     #(5)
```

1. Patrząc na XML w kgp/binary.xml, moglibyśmy pomyśleć, że węzeł `grammar` ma tylko dwa węzły potomne, czyli dwa elementy `ref`. Ale chyba o czymś zapominamy: o znakach końca linii! Za elementem `'<grammar>'` i przed pierwszym `'<ref>'` jest znak końca linii i zalicza się on do węzłów potomnych elementu `grammar`. Podobnie jest też znak końca linii po każdym `'</ref>'`; to także zalicza się do węzłów potomnych. Tak więc `grammar.childNodes` jest właściwie listą 5 obiektów: 3 obiekty `Text` i 2 obiekty `Element`.

2. Pierwszym potomkiem jest obiekt `Text` reprezentujący znak końca linii za znacznikiem `'<grammar>'` i przed pierwszym `'<ref>'`.

3. Drugim potomkiem jest obiekt `Element` reprezentujący pierwszy element `ref`.

4. Czwartym potomkiem jest obiekt `Element` reprezentujący drugi element `ref`.

5. Ostatnim potomkiem jest obiekt `Text` reprezentujący znak końca linii za znacznikiem końcowym `'</ref>'` i przed znacznikiem końcowym `'</grammar>'`.

Przykład 9.12 Drążenie aż do tekstu

```
>>> grammarNode
<DOM Element: grammar at 19167148>
>>> refNode = grammarNode.childNodes[1] #(1)
>>> refNode
<DOM Element: ref at 17987740>
>>> refNode.childNodes                  #(2)
[<DOM Text node "\n">, <DOM Text node "  ">, <DOM Element: p at 19315844>, \
<DOM Text node "\n">, <DOM Text node "  ">, \
<DOM Element: p at 19462036>, <DOM Text node "\n">]
>>> pNode = refNode.childNodes[2]
>>> pNode
<DOM Element: p at 19315844>
>>> print pNode.toxml()                 #(3)
<p>0</p>
>>> pNode.firstChild                    #(4)
<DOM Text node "0">
>>> pNode.firstChild.data               #(5)
u'0'
```

1. Jak już widzieliśmy w poprzednim przykładzie, pierwszym elementem `ref` jest `grammarNode.childNodes[1]`, ponieważ `childNodes[0]` jest węzłem typu `Text` dla znaku końca linii.

2. Element `ref` posiada swój zbiór węzłów potomnych, jeden dla znaku końca linii, oddzielny dla znaków spacji, jeden dla elementu `p` i tak dalej.

3. Możesz użyć metody `toxml` nawet tutaj, głęboko wewnątrz dokumentu.

4. Element `p` ma tylko jeden węzeł potomny (nie możemy tego zobaczyć na tym przykładzie, ale spójrzmy na `pNode.childNodes` jeśli nie wierzymy) i jest nim obiekt `Text` dla pojedynczego znaku `'0'`.

5. Atrybut `.data` węzła `Text` zawiera rzeczywisty napis, jaki ten tekstowy węzeł reprezentuje. Zauważmy, że wszystkie dane tekstowe przechowywane są w unikodzie.

10.4 Wyszukiwanie elementów

Wyszukiwanie elementów

Przemierzanie dokumentu XML poprzez przechodzenie przez każdy węzeł z osobna mogłoby być nużące. Jeśli poszukujesz czegoś szczególnego, co jest zagrzebane głęboko w dokumencie XML, istnieje skrót, którego możesz użyć, aby znaleźć to szybko: `getElementsByTagName`.

W tym podrozdziale używać będziemy pliku gramatyki kgp/binary.xml, który wygląda tak:

Przykład 9.20 binary.xml

```
<span><?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar></span>
```

Zawiera on dwa elementy `ref`: `'bit'` i `'byte'`. `'bit'` może przyjmować wartości `'0'` lub `'1'`, a `'byte'` może się składać z ośmiu bitów.

Przykład 9.21 Wprowadzenie do `getElementsByTagName`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref') #(1)
>>> reflist
[<DOM Element: ref at 136138108>, <DOM Element: ref at 136144292>]
>>> print reflist[0].toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print reflist[1].toxml()
<ref id="byte">
   <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
 <xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
 </ref>
```

1. `getElementsByTagName` przyjmuje jeden argument: nazwę elementu, który chcemy znaleźć. Zwraca listę obiektów `Element`, odpowiednią do znalezionych elementów XML posiadających podaną nazwę. W tym przypadku znaleźliśmy dwa elementy `ref`.

Przykład 9.22 Każdy element możemy przeszukiwać

```
>>> firstref = reflist[0]                     #(1)
>>> print firstref.toxml()
<ref id="bit">
   <p>0</p>
   <p>1</p>
 </ref>
>>> plist = firstref.getElementsByTagName("p") #(2)
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>]
>>> print plist[0].toxml()                    #(3)
<p>0</p>
>>> print plist[1].toxml()
<p>1</p>
```

1. Kontynuując poprzedni przykład, pierwszy obiekt naszej listy `reflist` jest elementem `ref 'bit'`.
2. Możemy użyć tej samej metody `getElementsByTagName` na tym obiekcie klasy `Element`, aby znaleźć wszystkie elementy ¡p¿ wewnątrz tego elementu `ref 'bit'`.
3. Tak jak poprzednio metoda `getElementsByTagName` zwraca listę wszystkich elementów jakie znajdzie. W tym przypadku mamy dwa, po jednym na każdy bit.

Przykład 9.23 Przeszukiwanie jest właściwie rekurencyjne

```
>>> plist = xmldoc.getElementsByTagName("p") #(1)
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>,
<DOM Element: p at 136146124>]
>>> plist[0].toxml()                        #(2)
'<p>0</p>'
>>> plist[1].toxml()
'<p>1</p>'
>>> plist[2].toxml()                        #(3)
'<p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
 <xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>'
```

1. Zauważmy różnicę pomiędzy tym i poprzednim przykładem. Poprzednio szukaliśmy elementów `p` wewnątrz `firstref`, lecz teraz szukamy elementów `p` wewnątrz `xmldoc`, czyli obiektu najwyższego poziomu reprezentującego cały dokument XML. To wyszukiwanie znajduje elementy `p` zagnieżdżone wewnątrz elementów `ref` wewnątrz głównego elementu gramatyki.
2. Pierwsze dwa elementy `p` znajdują się wewnątrz pierwszego elementu `ref` (element `ref 'bit'`).
3. Ostatni element `p`, to ten wewnątrz drugiego elementu `ref` (element `ref 'byte'`).

10.5 Dostęp do atrybutów elementów

Dostęp do atrybutów elementów

Elementy XML-a mogą mieć jeden lub wiele atrybutów i jest niewiarygodnie łatwo do nich dotrzeć, gdy dokument XML został już sparsowany.

W tym podrozdziale będziemy korzystać z pliku kgp/binary.xml, który już widzieliśmy w poprzednim podrozdziale.

Podrozdział ta może być lekko zagmatwany z powodu nakładającej się terminologii. Elementy dokumentu XML mają atrybuty, a obiekty Pythona także mają atrybuty. Gdy parsujemy dokument XML, otrzymujemy jakąś grupę obiektów Pythona, które reprezentują wszystkie części dokumentu XML, a niektóre z tych obiektów Pythona reprezentują atrybuty elementów XML-a. Jednak te obiekty (Python), które reprezentują atrybuty (XML), także mają atrybuty (Python), które są używane do pobierania różnych części atrybutu (XML), który ten obiekt reprezentuje. Uprzedzaliśmy, że to jest trochę zagmatwane.

Przykład 9.24 Dostęp do atrybutów elementów

```
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref')
>>> bitref = reflist[0]
>>> print bitref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> bitref.attributes          #(1)
<xml.dom.minidom.NamedNodeMap instance at 0x81e0c9c>
>>> bitref.attributes.keys()   #(2) (3)
[u'id']
>>> bitref.attributes.values() #(4)
[<xml.dom.minidom.Attr instance at 0x81d5044>]
>>> bitref.attributes["id"]    #(5)
<xml.dom.minidom.Attr instance at 0x81d5044>
```

1. Każdy obiekt `Element` ma atrybut o nazwie `attributes`, który jest obiektem klasy `NamedNodeMap`. Brzmi groźnie, ale takie nie jest, ponieważ obiekt `NamedNodeMap` jest obiektem działającym jak słownik, a więc już wiemy, jak go używać.

2. Traktując obiekt `NamedNodeMap` jak słownik, możemy pobrać listę nazw atrybutów tego elementu używając `attributes.keys()`. Ten element ma tylko jeden atrybut: `'id'`.

3. Nazwy atrybutów, jak każdy inny tekst w dokumencie XML, są zapisane w postaci unikodu.

4. Znowu traktując `NamedNodeMap` jak słownik, możemy pobrać listę wartości atrybutów używając `attributes.values()`. Wartości same w sobie także są obiektami typu `Attr`. Jak wydobyć użyteczne informacje z tego obiektu zobaczymy w następnym przykładzie.

5. Nadal traktując `NamedNodeMap` jak słownik, możemy dotrzeć do poszczególnych atrybutów poprzez ich nazwy, używając normalnej składni dla słowników. (Szczególnie uważni czytelnicy już wiedzą jak klasa `NamedNodeMap` realizuje ten fajny trik: poprzez definicję metody specjalnej o nazwie `__getitem__`. Inni czytelnicy mogą pocieszyć się faktem, iż nie muszą rozumieć jak to działa, aby używać tego efektywnie.)

Przykład 9.25 Dostęp do poszczególnych atrybutów

```
>>> a = bitref.attributes["id"]
>>> a
<xml.dom.minidom.Attr instance at 0x81d5044>
>>> a.name  #(1)
u'id'
>>> a.value #(2)
u'bit'
```

1. Obiekt `Attr` w całości reprezentuje pojedynczy atrybut XML-a pojedynczego elementu XML-a. Nazwa atrybutu (ta sama, której użyliśmy do znalezienia tego obiektu w `bitref.attributes` pseudo-słownikowym obiekcie `NamedNodeMap`) znajduje się w `a.name`.

2. Właściwa wartość tekstowa tego atrybutu XML-a znajduje się w `a.value`.

Podobnie jak w słowniku atrybuty elementu XML-a nie są uporządkowane. Może się zdarzyć, że w oryginalnym dokumencie XML atrybuty będą ułożone w pewnym określonym porządku i może się zdarzyć, że obiekty `Attr` będą również ułożone w takim samym porządku po sparsowaniu dokumentu do obiektów Pythona, ale te uporządkowania są tak naprawdę przypadkowe i nie powinny mieć żadnego specjalnego znaczenia. Zawsze powinienniśmy odwoływać się do poszczególnych atrybutów poprzez nazwę, tak jak do elementów słownika poprzez klucz.

10.6 Przetwarzanie XML-a - podsumowanie

Podsumowanie

OK, to by było na tyle ciężkich tematów o XML-u. Następny rozdział będzie nadal wykorzystywał te same przykładowe programy, ale będzie zwracał uwagę na inne aspekty, które sprawiają, że program jest bardziej elastyczny: wykorzystywanie strumieni do przetwarzania wejścia, używanie funkcji `getattr` jako pośrednika, a także korzystanie z flag w linii poleceń, aby pozwolić użytkownikom skonfigurować program bez zmieniania kodu źródłowego.

Przed przejściem do następnego rozdziału, powinniśmy nie mieć problemów z:

- parsowaniem dokumentów XML za pomocą `minidom`-a, przeszukiwaniem sparsowanego dokumentu, a także z dostępem do dowolnego atrybutu
- uporządkowywaniem złożonych bibliotek w pakiety

Rozdział 11

Skrypty i strumienie

11.1 Abstrakcyjne źródła wejścia

Abstrakcyjne źródła wejścia

Jedną z najważniejszych możliwości Pythona jest jego dynamiczne wiązanie, a jednym z najbardziej przydatnych przykładów wykorzystania tego jest *obiekt plikopodobny* (ang. *file-like object*).

Wiele funkcji, które wymagają jakiegoś źródła wejścia, mogłyby po prostu przyjmować jako argument nazwę pliku, następnie go otwierać, czytać, a na końcu go zamykać. Jednak tego nie robią. Zamiast działać w ten sposób, jako argument przyjmują obiekt pliku lub *obiekt plikopodobny*.

W najprostszym przypadku *obiekt plikopodobny* jest dowolnym obiektem z metodą `read`, która przyjmuje opcjonalny parametr wielkości, `size`, a następnie zwraca łańcuch znaków. Kiedy wywołujemy go bez parametru `size`, odczytuje wszystko, co jest do przeczytania ze źródła wejścia, a potem zwraca te wszystkie dane jako pojedynczy łańcuch znaków. Natomiast kiedy wywołamy metodę `read` z parametrem `size`, to odczyta ona tyle bajtów ze źródła wejścia, ile wynosi wartość `size`, a następnie zwróci te dane. Kiedy ponownie wywołamy tę metodę, zostanie odczytana i zwrócona dalsza porcja danych (czyli dane będą czytane od miejsca, w którym wcześniej skończono czytać).

Powyżej opisaliśmy, w jaki sposób działają prawdziwe pliki. Jednak nie musimy się ograniczać do prawdziwych plików. Źródłem wejścia może być wszystko: plik na dysku, strona internetowa, czy nawet jakiś łańcuch znaków. Dopóki przekazujemy do funkcji *obiekt plikopodobny*, a funkcja ta po prostu wywołuje metodę `read`, to funkcja może obsłużyć dowolny rodzaj wejścia, bez posiadania jakiegoś specjalnego kodu dla każdego rodzaju wejścia.

Może się zastanawiamy, co ma to wspólnego z przetwarzaniem XML-a? Otóż `minidom.parse` jest taką funkcją, do której możemy przekazać *obiekt plikopodobny*.

Przykład 10.1 Parsowanie XML-u z pliku

```
>>> from xml.dom import minidom
>>> fsock = open('binary.xml')    #(1)
>>> xmldoc = minidom.parse(fsock) #(2)
>>> fsock.close()                 #(3)
>>> print xmldoc.toxml()          #(4)
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

1. Najpierw otwieramy plik z dysku. Otrzymujemy przez to obiekt pliku.
2. Przekazujemy obiekt pliku do funkcji `minidom.parse`, która wywołuje metodę `read` z `fsock` i czyta dokument XML z tego pliku.

3. Koniecznie wywołujemy metodę `close` obiektu pliku, jak już skończyliśmy na nim pracę. `minidom.parse` nie zrobi tego za nas.

4. Wywołując ze zwróconego dokumentu XML metodę `toxml()`, wypisujemy cały dokument.

Dobrze, to wszystko wygląda jak kolosalne marnotrawstwo czasu. W końcu już wcześniej widzieliśmy, że `minidom.parse` może przyjąć jako argument nazwę pliku i wykonać całą robotę z otwieraniem i zamykaniem automatycznie. Prawdą jest, że jeśli chcemy sparsować lokalny plik, możemy przekazać nazwę pliku do `minidom.parse`, a funkcja ta będzie umiała mądrze to wykorzystać. Lecz zauważmy jak podobne i łatwe jest także parsowanie dokumentu XML pochodzącego bezpośrednio z Internetu.

Przykład 10.2 Parsowanie XML-a z URL-a

```
>>> import urllib
>>> usock = urllib.urlopen('http://slashdot.org/slashdot.rdf') #(1)
>>> xmldoc = minidom.parse(usock)                              #(2)
>>> usock.close()                                              #(3)
>>> print xmldoc.toxml()                                       #(4)
<?xml version="1.0" ?>
<rdf:RDF xmlns="http://my.netscape.com/rdf/simple/0.9/"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns\#">

<channel>
<title>Slashdot</title>
<link>http://slashdot.org/</link>
<description>News for nerds, stuff that matters</description>
</channel>

<image>
<title>Slashdot</title>
<url>http://images.slashdot.org/topics/topicslashdot.gif</url>
<link>http://slashdot.org/</link>
</image>

<item>
<title>To HDTV or Not to HDTV?</title>
<link>http://slashdot.org/article.pl?sid=01/12/28/0421241</link>
</item>

[...ciach...]
```

1. Jak już zaobserwowaliśmy w poprzednim rozdziale, `urlopen` przyjmuje adres URL strony internetowej i zwraca *obiekt plikopodobny*. Ponadto, co jest bardzo ważne, obiekt ten posiada metodę `read`, która zwraca źródło danej strony internetowej.

2. Teraz przekazujemy ten *obiekt plikopodobny* do `minidom.parse`, która posłusznie wywołuje metodę `read` i parsuje dane XML, które zostają zwrócone przez `read`. Fakt, że te dane przychodzą teraz bezpośrednio z Internetu, jest kompletnie

nieistotny. `minidom.parse` nie ma o stronach internetowych żadnego pojęcia; on tylko wie coś o *obiektach plikopodobnych*.

3. Jak tylko *obiekt plikopodobny*, który podarował nam `urlopen`, nie będzie potrzebny, koniecznie zamykamy go.

4. Przy okazji, ten URL jest prawdziwy i on naprawdę jest dokumentem XML. Reprezentuje on aktualne nagłówki, techniczne newsy i plotki w Slashdocie.

Przykład 10.3 Parsowanie XML-a z łańcucha znaków (prosty sposób, ale mało elastyczny)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> xmldoc = minidom.parseString(contents) #(1)
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
 <grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

1. `minidom` posiada metodę `parseString`, która przyjmuje cały dokument XML w postaci łańcucha znaków i parsuje go. Możemy ją wykorzystać zamiast `minidom.parse`, jeśli wiemy, że posiadamy cały dokument w formie łańcucha znaków.

OK, to możemy korzystać z funkcji `minidom.parse` zarówno do parsowania lokalnych plików jak i odległych URL-ów, ale do parsowania łańcuchów znaków wykorzystujemy... inną funkcję. Oznacza to, że jeśli chcielibyśmy, aby nasz program mógł dać wyjście z pliku, adresu URL lub łańcucha znaków, potrzebujemy specjalnej logiki, aby sprawdzić czy mamy do czynienia z łańcuchem znaków, a jeśli tak, to wywołać funkcję `parseString` zamiast `parse`. Jakie to niesatysfakcjonujące...

Gdyby tylko był sposób, aby zmienić łańcuch znaków na obiekt *plikopodobny*, to moglibyśmy po prostu przekazać ten obiekt do `minidom.parse`. I rzeczywiście, istnieje moduł specjalnie zaprojektowany do tego: `StringIO`.

Przykład 10.4 Wprowadzenie do `StringIO`

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> import StringIO
>>> ssock = StringIO.StringIO(contents)   #(1)
>>> ssock.read()                          #(2)
"<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock.read()                          #(3)
''
>>> ssock.seek(0)                         #(4)
>>> ssock.read(15)                        #(5)
'<grammar><ref i'
>>> ssock.read(15)
"d='bit'><p>0</p"
>>> ssock.read()
'><p>1</p></ref></grammar>'
>>> ssock.close()                         #(6)
```

1. Moduł `StringIO` zawiera tylko jedną klasę, także nazwaną `StringIO`, która pozwala zamienić napis w *obiekt plikopodobny*. Klasa `StringIO` podczas tworzenia instancji przyjmuje jako parametr łańcuch znaków.

2. Teraz już mamy *obiekt plikopodobny* i możemy robić wszystkie możliwe *plikopodobne* operacje. Na przykład `read`, która zwraca oryginalny łańcuch.

3. Wywołując ponownie `read` otrzymamy pusty napis. W ten sposób działa prawdziwy obiekt pliku; kiedy już zostanie przeczytany cały plik, nie można czytać więcej bez wyraźnego przesunięcia do początku pliku. Obiekt `StringIO` pracuje w ten sam sposób.

4. Możemy jawnie przesunąć się do początku napisu, podobnie jak możemy się przesunąć w pliku, wykorzystując metodę `seek` obiektu klasy `StringIO`.

5. Możemy także czytać fragmentami łańcuch znaków, dzięki przekazaniu parametr wielkości `size` do metody `read`.

6. Za każdym razem, kiedy wywołamy `read`, zostanie nam zwrócona pozostała część napisu, która nie została jeszcze przeczytana. W dokładnie ten sam sposób działa obiekt pliku.

Przykład 10.5 Parsowanie XML-a z łańcucha znaków (sposób z obiektem plikopodobnym)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock = StringIO.StringIO(contents)
>>> xmldoc = minidom.parse(ssock) #(1)
>>> ssock.close()
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

1. Teraz możemy przekazać obiekt plikopodobny (w rzeczywistości instancję `StringIO`) do funkcji `minidom.parse`, która z kolei wywoła metodę `read` z tego *obiektu plikopodobnego* i szczęśliwie wszystko przeparsuje, nie zdając sobie nawet sprawy, że wejście to pochodzi z łańcucha znaków.

To już wiemy, jak za pomocą pojedynczej funkcji, `minidom.parse`, sparsować dokument XML przechowywany na stronie internetowej, lokalnym pliku, czy w łańcuchu znaków. Dla strony internetowej wykorzystamy `urlopen`, aby dostać *obiekt plikopodobny*; dla lokalnego pliku, wykorzystamy `open`; a w przypadku łańcucha znaków skorzystamy z `StringIO`. Lecz teraz pójdźmy trochę do przodu i uogólnijmy też te różnice.

Przykład 10.6 `openAnything`

```
def openAnything(source):                   #(1)
    # próbuje otworzyć za pomocą urllib (jeśli source jest URL-em do http, ftp itp.)
    import urllib
    try:
        return urllib.urlopen(source)      #(2)
    except (IOError, OSError):
        pass

    # próbuje otworzyć za pomocą wbudowanej funkcji open (gdy source jest ścieżką do pliku)
```

```
    try:
        return open(source)              #(3)
    except (IOError, OSError):
        pass

    # traktuje source jako łańcuch znaków z danymi
    import StringIO
    return StringIO.StringIO(str(source))  #(4)
```

1. Funkcja `openAnything` przyjmuje pojedynczy argument, `source`, i zwraca *obiekt plikopodobny.* `source` jest łańcuchem znaków o różnym charakterze. Może się odnosić do adresu URL (np. `'http://slashdot.org/slashdot.rdf'`), może być globalną lub lokalną ścieżką do pliku (np. `'binary.xml'`), czy też łańcuchem znaków przechowującym dokument XML, który ma zostać sparsowany.

2. Najpierw sprawdzamy, czy `source` jest URL-em. Robimy to brutalnie: próbujemy otworzyć to jako URL i cicho pomijamy błędy spowodowane próbą otworzenia czegoś, co nie jest URL-em. Jest to właściwie eleganckie w tym sensie, że jeśli `urllib` będzie kiedyś obsługiwał nowe typy URL-i, nasz program także je obsłuży i to bez konieczności zmiany kodu. Jeśli `urllib` jest w stanie otworzyć `source`, to `return` wykopie nas bezpośrednio z funkcji i poniższe instrukcje `try` nie będą nigdy wykonywane.

3. W innym przypadku, gdy `urllib` wrzasnął na nas i powiedział, że `source` nie jest poprawnym URL-em, zakładamy, że jest to ścieżka do pliku znajdującego się na dysku i próbujemy go otworzyć. Ponownie, nic nie robimy, by sprawdzić, czy `source` jest poprawną nazwą pliku (zasady określające poprawność nazwy pliku są znacząco różne na różnych platformach, dlatego prawdopodobnie i tak byśmy to źle zrobili). Zamiast tego, na ślepo otwieramy plik i cicho pomijamy wszystkie błędy.

4. W tym miejscu zakładamy, że `source` jest łańcuchem znaków, który przechowuje dokument XML (ponieważ nic innego nie zadziałało), dlatego wykorzystujemy `StringIO`, aby utworzyć *obiekt plikopodobny* i zwracamy go. (Tak naprawdę, ponieważ wykorzystujemy funkcję `str`, `source` nie musi być nawet łańcuchem znaków; może być nawet dowolnym obiektem, a z którego zostanie wykorzystana jego tekstowa reprezentacja, a która jest zdefiniowana przez specjalną metodę `__str__`.)

Teraz możemy wykorzystać funkcję `openAnything` w połączeniu z `minidom.parse`, aby utworzyć funkcję, która przyjmuje źródło `source`, które w jakiś sposób odwołuje się do dokumentu XML (może to robić za pomocą adresu URL, lokalnego pliku, czy też dokumentu przechowywanego jako łańcuch znaków), i parsuje je.

Przykład 10.7 Wykorzystanie `openAnything` w kgp/kgp.py

```
class KantGenerator:
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc
```

11.2 Standardowy strumień wejścia, wyjścia i błędów

Standardowy strumień wejścia, wyjścia i błędów

Użytkownicy Uniksa są już prawdopodobnie zapoznani z koncepcją standardowego wejścia, standardowego wyjścia i standardowego strumienia błędów. Ten podrozdział jest dla pozostałych osób.

Standardowe wyjście i strumień błędów (powszechnie używana skrócona forma to *stdout* i *stderr*) są strumieniami danych wbudowanymi do każdego systemu Unix. Kiedy coś wypisujemy, idzie to do strumienia *stdout*; kiedy wystąpi błąd w programie, a program wypisze informacje pomocne przy debugowaniu (jak traceback w Pythonie), to wszystko pójdzie do strumienia *stderr*. Te dwa strumienie są zwykle połączone z oknem terminala, na którym pracujemy, więc jeżeli program coś wypisuje, zobaczymy to na wyjściu, a kiedy program spowoduje błąd, zobaczymy informacje debugujące. (Jeśli pracujemy w systemie z okienkowym IDE Pythona, *stdout* i *stderr* domyślnie będą połączone z "interaktywnym oknem".)

Przykład 10.8 Wprowadzenie do *stdout* i *stderr*

```
>>> for i in range(3):
...     print 'Nurkujemy'            #(1)
Nurkujemy
Nurkujemy
Nurkujemy
>>> import sys
>>> for i in range(3):
...     sys.stdout.write('Nurkujemy') #(2)
NurkujemyNurkujemyNurkujemy
>>> for i in range(3):
...     sys.stderr.write('Nurkujemy') #(3)
NurkujemyNurkujemyNurkujemy
```

1. Jak zobaczyliśmy w przykładzie 6.9, "Prosty licznik", możemy wykorzystać wbudowaną funkcje `range`, aby zbudować prostą pętlę licznikową, która powtarza pewną operację określoną liczbę razy.

2. `stdout` jest *obiektem plikopodobnym*; wywołując jego funkcję `write` będziemy wypisywać na wyjście napis, który przekazaliśmy. W rzeczywistość, to właśnie funkcja `print` naprawdę robi; dodaje ona znak nowej linii do wypisywanego napisu, a następnie wywołuje `sys.stdout.write`.

3. W tym prostym przypadku `stdout` i `stderr` wysyłają wyjście do tego samego miejsca: do IDE Pythona (jeśli jesteśmy w nim) lub do terminala (jeśli mamy uruchomionego Pythona z linii poleceń). Podobnie jak `stdout`, `stderr` nie dodaje znaku nowej linii za nas; jeśli chcemy, aby ten znak został dodany, musimy to zrobić sami.

Zarówno `stdout` i `stderr` są *obiektami plikopodobnymi*, a które omawialiśmy w podrozdziale 10.1, "Abstrakcyjne źródła wejścia", lecz te są tylko do zapisu. Nie posiadają one metody `read`, tylko `write`. Jednak nadal są one *obiektami plikopodobnymi* i

możemy do nich przypisać inny obiekt pliku lub *obiekt plikopodobny*, aby przekierować ich wyjście.

Przykład 10.9 Przekierowywanie wyjścia

```
[you@localhost kgp]$ python stdout.py
Nurkujemy
[you@localhost kgp]$ cat out.log
Ta wiadomość będzie logowana i nie zostanie wypisana na wyjście
```

(W Windowsie możemy wykorzystać polecenie `type`, zamiast `cat`, aby wyświetlić zawartość pliku.)

```
#-*- coding: utf-8 -*-
#stdout.py
import sys

print 'Nurkujemy'                                                       #(1)
saveout = sys.stdout                                                    #(2)
fsock = open('out.log', 'w')                                            #(3)
sys.stdout = fsock                                                      #(4)
print 'Ta wiadomość będzie logowana i nie zostanie wypisana na wyjście' #(5)
sys.stdout = saveout                                                    #(6)
fsock.close()                                                           #(7)
```

1. To zostanie wypisane w interaktywnym oknie IDE (lub w terminalu, jeśli skrypt został uruchomiony z linii poleceń).
2. Zawsze, zanim przekierujemy standardowe wyjście, przypisujemy gdzieś `stdout`, dzięki temu, będziemy potem mogli do niego normalnie wrócić.
3. Otwieramy plik do zapisu. Jeśli plik nie istnieje, zostanie utworzony. Jeśli istnieje, zostanie nadpisany.
4. Całe późniejsze wyjście zostanie przekierowane do pliku, który właśnie otworzyliśmy.
5. Zostanie to wypisane tylko do pliku `out.log`; nie będzie widoczne w oknie IDE lub w terminalu.
6. Przywracamy `stdout` do początkowej, oryginalnej postaci.
7. Zamykamy plik `out.log`.

Dodajmy, że w wypisywanym łańcuchu znaków użyliśmy polskich znaków, a ponieważ nie skorzystaliśmy z unikodu, więc napis ten zostanie wypisany w takiej samej postaci, w jakiej został zapisany w pliku Pythona (czyli wiadomość zostanie zapisana w kodowaniu `utf-8`). Gdybyśmy skorzystali z unikodu, musielibyśmy wyraźnie zakodować ten napis do jakiegoś kodowania za pomocą metody `encode`, ponieważ Python nie wie, z jakiego kodowania chce korzystać utworzony przez nas plik (plik `out.log` przypisany do zmiennej `stdout`).

Przekierowywanie standardowego strumienia błędów (*stderr*) działa w ten sam sposób, wykorzystując `sys.stderr`, zamiast `sys.stdout`.

Przykład 10.10 Przekierowywanie informacji o błędach

```
[you@localhost kgp]$ python stderr.py
[you@localhost kgp]$ cat error.log
Traceback (most recent line last):
File "stderr.py", line 6, in ?
   raise Exception('ten błąd będzie logowany)
Exception: ten błąd będzie logowany

#stderr.py
#-*- coding: utf-8 -*-
import sys

fsock = open('error.log', 'w')              #(1)
sys.stderr = fsock                          #(2)
raise Exception('ten błąd będzie logowany') #(3) (4)
```

1. Otwieramy plik `error.log`, gdzie chcemy przechowywać informacje debugujące.
2. Przekierowujemy standardowy strumień błędów, dzięki przypisaniu obiektu nowo otwartego pliku do `sys.stderr`.
3. Rzucamy wyjątek. Zauważmy, że na ekranie wyjściowym nic nie zostanie wypisane. Wszystkie informacje *traceback* zostały zapisane w `error.log`.
4. Zauważmy także, że nie zamknęliśmy jawnie pliku `error.log`, a nawet nie przypisaliśmy do `sys.stderr` jego pierwotnej wartości. To jest wspaniałe, że kiedy program się rozwali (z powodu wyjątku), Python wyczyści i zamknie wszystkie pliki za nas. Nie ma żadnej różnicy, czy `stderr` zostanie przywrócony, czy też nie, ponieważ program się rozwala, a Python kończy działanie. Przywrócenie wartości do oryginalnej, jest bardziej ważne dla `stdout`, jeśli zamierzasz później wykonywać jakieś inne operacje w tym samym skrypcie.

Ponieważ powszechnie wypisuje się informacje o błędach na standardowy strumień błędów, Python posiada skrótową składnie, która można wykorzystać do bezpośredniego przekierowywania wyjścia.

Przykład 10.11 Wypisywanie do `stderr`

```
>>> print 'wchodzimy do funkcji'
wchodzimy do funkcji
>>> import sys
>>> print >> sys.stderr, 'wchodzimy do funkcji' #(1)
wchodzimy do funkcji
```

1. Ta skrótowa składnia wyrażenia `print` może być wykorzystywana do pisania do dowolnego, otwartego pliku, lub do *obiektu plikopodobnego*. W tym przypadku, możemy przekierować pojedynczą instrukcję `print` do `stderr` bez wpływu na następne instrukcje `print`.

Z innej strony, standardowe wejścia jest obiektem pliku tylko do odczytu i reprezentuje dane przechodzące z niektórych wcześniejszych programów. Prawdopodobnie

nie jest to zrozumiałe dla klasycznych użytkowników Mac OS-a lub nawet dla użytkowników Windows, którzy nie mieli za wiele do czynienia z linią poleceń MS-DOS-a. Działa to w ten sposób, że konstruujemy ciąg poleceń w jednej linii, w taki sposób, że to co jeden program wypisuje na wyjście, następny w tym ciągu traktuje jako wejście. Pierwszy program prosto wypisuje wszystko na standardowe wyjście (bez korzystania ze specjalnych przekierowań, wykorzystuje normalną instrukcję `print` itp.), a następny program czyta ze standardowego wejścia, a system operacyjny udostępnia połączenie pomiędzy wyjściem pierwszego programu, a wyjściem kolejnego.

Przykład 10.12 Ciąg poleceń

```
[you@localhost kgp]$ python kgp.py -g binary.xml    #(1)
01100111
[you@localhost kgp]$ cat binary.xml                     #(2)
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator Pro v1.0//EN"
"kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
[you@localhost kgp]$ cat binary.xml | python kgp.py -g - #(3) (4)
10110001
```

1. Jak zobaczyliśmy w podrozdziale 9.1, "Nurkujemy", polecenie to wyświetli ciąg ośmiu przypadkowych bitów, `0` i `1`.

2. Dzięki temu po prostu wypiszemy całą zawartość pliku `binary.xml`. (Użytkownicy Windowsa powinni wykorzystać polecenie `type` zamiast `cat`.)

3. Polecenie to wypisuje zawartość pliku kgp/binary.xml, ale znak "|" (ang. *pipe*), oznacza, że standardowe wyjście nie zostanie wypisana na ekran. Zamiast tego, zawartość standardowego wyjścia zostanie wykorzystane jako standardowe wejście następnego programu, który w tym przypadku jest skryptem Pythona.

4. Zamiast określać modułu (np. `binary.xml`), dajemy "-", który każe naszemu skryptowi wczytać gramatykę ze standardowego wejścia, zamiast z określonego pliku na dysku. (Więcej o tym, w jaki sposób to się dzieje w następnym przykładzie.) Zatem efekt będzie taki sam, jak w pierwszym poleceniu, gdzie bezpośrednio określamy plik gramatyki, ale tutaj zwróćmy uwagę na rozszerzone możliwości. Zamiast wywoływać `cat binary.xml`, moglibyśmy uruchomić skrypt, który by dynamicznie generował gramatykę, a następnie mógłby ją doprowadzić do naszego skryptu. Dane mogłyby przyjść skądkolwiek: z bazy danych, innego skryptu generującego gramatykę lub jeszcze inaczej. Zaletą tego jest to, że nie musimy zmieniać w żaden sposób kgp/kgp.py, aby dołączyć jakąś funkcjonalność.

Jedynie, co potrzebujemy, to możliwość wczytania gramatyki ze standardowego wejścia, a całą logikę dodatkowej funkcjonalności możemy rozdzielić wewnątrz innego programu.

Więc w jaki sposób skrypt "wie", żeby czytać ze standardowego wejścia, gdy plik gramatyki to "-"? To nie jest żadna magia; to tylko właśnie prosty kod.

Przykład 10.13 Czytanie ze standardowego wejścia w kgp/kgp.py

```
def openAnything(source):
    if source == "-":    #(1)
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:

[... ciach ...]
```

1. Jest to funkcja `openAnything` z kgp/toolbox.py, którą wcześniej badaliśmy w podrozdziale 10.1, Ąbstrakcyjne źródła wejścia". Wszystko, co musimy zrobić, to dodanie trzech linii kodu na początku, aby sprawdzić, czy źródłem nie jest `-`"; jeśli tak, to zwracamy `sys.stdin`. Naprawdę, to tylko tyle! Pamiętasz, `stdin` jest *obiektem plikopodobnym* z metodą `read`, więc pozostałą część kodu (w kgp/kgp.py, gdzie wywołujemy funkcję `openAnything`) w żaden sposób nie zmieniamy.

11.3 Buforowanie odszukanego węzła

Buforowanie odszukanego węzła

kgp/kgp.py stosuje kilka sztuczek, które mogą, lecz nie muszą, być użyteczne przy przetwarzaniu XML-a. Pierwsza z nich wykorzystuje spójną strukturę dokumentów wejściowych do utworzenia bufora węzłów.

Plik gramatyki definiuje szereg elementów `ref`. Każdy z nich zawiera jeden lub więcej elementów `p`, które mogą zawierać wiele różnych rzeczy, włącznie z elementami `xref`. Gdy napotykamy element `xref`, wyszukujemy odpowiedni element `ref` z tym samym atrybutem `id` i wybieramy jeden z elementów potomnych elementu `ref` i parsujemy go. (W następnym podrozdziale zobaczymy jak dokonywany jest ten losowy wybór.)

W taki sposób rozwijamy gramatykę: definiujemy elementy `ref` dla najmniejszych części, następnie definiujemy elementy `ref`, które zawierają te pierwsze elementy `ref` poprzez użycie `xref` itd. Potem parsujemy "największą" referencję, przechodzimy po kolei do każdego elementu `xref` i ostatecznie generujemy prawdziwy tekst. Ten wygenerowany tekst zależy od tej (losowej) decyzji podjętej przy wypełnianiu elementu `xref`, a więc efekt może być inny za każdym razem.

To wszystko jest bardzo elastyczne, ale ma jedną wadę: wydajność. Gdy napotkamy element `xref` i potrzebujemy odszukać odpowiedniego dla niego elementu `ref`, to pojawia się problem. Element `xref` ma atrybut `id` i chcemy odszukać element `ref`, który ma taki sam atrybut `id`, ale nie ma prostego sposobu aby to zrobić. Powolnym sposobem na zrobienie tego byłoby pobranie pełnej listy elementów `ref` za każdym razem, a następnie przeszukanie jej w pętli pod kątem atrybutu `id`. Szybkim sposobem jest utworzenie takiej listy raz, a następnie utworzenie bufora w postaci słownika.

Przykład 10.14 `loadGrammar`

```
def loadGrammar(self, grammar):
        self.grammar = self._load(grammar)
        self.refs = {}                                        #(1)
        for ref in self.grammar.getElementsByTagName("ref"): #(2)
            self.refs[ref.attributes["id"].value] = ref      #(3) (4)
```

1. Rozpoczynamy od utworzenia pustego słownika `self.refs`.

2. Jak już widzieliśmy w podrozdziale 9.5, "Wyszukiwanie elementów", `getElementsByTagName` zwraca listę wszystkich elementów o podanej nazwie. Także łatwo możemy uzyskać listę wszystkich elementów `ref`, a następnie po prostu przeszukać ją w pętli.

3. Jak już widzieliśmy w podrozdziale 9.6, "Dostęp do atrybutów elementów", możemy pobrać atrybut elementu poprzez nazwę używając standardowej składni słownikowej. Także kluczami słownika `self.refs` będą wartości atrybutu `id` każdego elementu `ref`.

4. Wartościami słownika `self.refs` będą elementy `ref` jako takie. Jak już widzieliśmy w podrozdziale 9.3, "Parsowanie XML-a", każdy element, każdy węzeł, każdy komentarz, każdy kawałek tekstu w parsowanym dokumencie XML jest obiektem.

Gdy tylko bufor (cache) zostanie utworzony, po napotkaniu elementu `xref`, aby odnaleźć element `ref` z takim samym atrybutem `id`, możemy po prostu sięgnąć do słownika `self.refs`.

Przykład 10.15 Użycie bufora elementów `ref`

```
def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

Funkcję `randomChildElement` zgłębimy w następnym podrozdziale.

11.4 Wyszukanie bezpośrednich elementów potomnych

Wyszukanie bezpośrednich elementów potomnych

Inną przydatną techniką przy parsowaniu dokumentów XML jest odnajdywanie wszystkich bezpośrednich elementów potomnych (dzieci) danego elementu. Na przykład w pliku gramatyki element `ref` może mieć szereg elementów `p`, a każdy z nich może zawierać wiele rzeczy, włącznie z innymi elementami `p`. Chcemy wyszukać tylko te elementy `p`, które są potomkami elementu `ref`, a nie elementy `p`, które są potomkami innych elementów `p`.

Pewnie myślisz, że możesz do tego celu po prostu użyć funkcji `getElementsByTagName`, ale niestety nie możesz. Funkcja `getElementsByTagName` przeszukuje rekurencyjnie i zwraca pojedyncza listę wszystkich elementów jakie znajdzie. Ponieważ elementy `p` mogą zawierać inne elementy `p`, nie możemy użyć funkcji `getElementsByTagName`. Zwróciłaby ona zagnieżdżone elementy `p`, a tego nie chcemy. Aby znaleźć tylko bezpośrednie elementy potomne, musimy to wykonać samodzielnie.

Przykład 10.16 Wyszukanie bezpośrednich elementów potomnych

```
def randomChildElement(self, node):
    choices = [e for e in node.childNodes
               if e.nodeType == e.ELEMENT_NODE] #(1) (2) (3)
    chosen = random.choice(choices)             #(4)
    return chosen
```

1. Jak już widzieliśmy w przykładzie 9.9, "Pobieranie węzłów potomnych", atrybut `childNodes` zwraca listę wszystkich elementów potomnych danego elementu.

2. Jednakże, jak już widziałeś w przykładzie 9.11, "Węzłami potomnymi może być także tekst", lista zwrócona przez `childNodes` zawiera całą różnorodność typów węzłów, włączając to węzły tekstowe. W tym wypadku szukamy jednak tylko potomków, które są elementami.

3. Każdy węzeł posiada atrybut `nodeType`, który może przyjmować wartości `ELEMENT_NODE`, `TEXT_NODE`, `COMMENT_NODE` i wiele innych. Pełna lista możliwych wartości znajduje się w pliku `__init__.py` w pakiecie `xml.dom`. (Zajrzyj do podrozdziału 9.2, "Pakiety", aby się więcej dowiedzieć o pakietach.) Ale my jesteśmy zainteresowani węzłami, które są elementami, a więc możemy odfiltrować z listy tylko te elementy, których atrybut `nodeType` jest równy `ELEMENT_NODE`.

4. Gdy tylko mamy już listę właściwych elementów, wybór losowego elementu jest łatwy. Python udostępnia moduł o nazwie `random`, który zawiera kilka funkcji. Funkcja `random.choice` pobiera listę z dowolną ilością elementów i zwraca losowy element. Np. jeśli element `ref` zawiera kilka elementów `p`, to `choices` będzie listą elementów `p`, a do `chosen` zostanie przypisany dokładnie jeden z nich, wybrany losowo.

11.5 Tworzenie oddzielnych funkcji obsługi względem typu węzła

Tworzenie oddzielnych funkcji obsługi względem typu węzła

Trzecim użytecznym chwytem podczas przetwarzania XML-a jest podzielenie kodu w logiczny sposób na funkcje oparte na typie węzła i nazwie elementu. Parsując dokument przetwarzamy rozmaite typy węzłów, które są reprezentowane przez obiekty Pythona. Poziom główny dokumentu jest bezpośrednio reprezentowany przez obiekt klasy `Document`. Z kolei `Document` zawiera jeden lub więcej obiektów klasy `Element` (reprezentujące znaczniki XML-a), a każdy z nich może zawierać inne obiekty klasy `Element`, obiekty klasy `Text` (fragmenty tekstu), czy obiektów `Comment` (osadzone komentarze w dokumencie). Python pozwala w łatwy sposób napisać funkcję pośredniczącą, która rozdziela logikę dla każdego rodzaju węzła.

Przykład 10.17 Nazwy klas parsowanych obiektów XML

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('kant.xml') #(1)
>>> xmldoc
<xml.dom.minidom.Document instance at 0x01359DE8>
>>> xmldoc.__class__                  #(2)
<class xml.dom.minidom.Document at 0x01105D40>
>>> xmldoc.__class__.__name__         #(3)
'Document'
```

1. Załóżmy na moment, że kgp/kant.xml jest w bieżącym katalogu.
2. Jak powiedzieliśmy w podrozdziale "Pakiety", obiekt zwrócony przez parsowany dokument jest instancją klasy `Document`, która została zdefiniowana w `minidom.py` w pakiecie `xml.dom`. Jak zobaczyliśmy w podrozdziale "Tworzenie instancji klasy", `__class__` jest wbudowanym atrybutem każdego obiektu Pythona.
3. Ponadto `__name__` jest wbudowanym atrybutem każdej klasy Pythona. Atrybut ten przechowuje napis, a napis ten nie jest niczym tajemniczym, jest po prostu nazwą danej klasy. (Zobacz podrozdział "Definiowanie klas".)

To fajnie, możemy pobrać nazwę klasy dowolnego węzła XML-a (ponieważ węzły są reprezentowane przez Pythonowe obiekty). Jak można wykorzystać tę zaletę, aby rozdzielić logikę parsowania dla każdego typu węzła? Odpowiedzią jest `getattr`, który pierwszy raz zobaczyliśmy w podrozdziale "Funkcja `getattr`".

Przykład 10.18 `parse`, ogólna funkcja pośrednicząca dla węzła XML

```
def parse(self, node):
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__) #(1) (2)
        parseMethod(node) #(3)
```

1. Od razu, zauważmy, że konstruujemy dłuższy napis oparty na nazwie klasy przekazanego węzła (jako argument `node`). Zatem, jeśli przekażemy węzeł `Document`-u, konstruujemy napis `'parse_Document'` itd.

2. Teraz, jeśli potraktujemy tę nazwę jako nazwę funkcji, otrzymamy dzięki `getattr` referencję do funkcji.

3. Ostatecznie, możemy wywołać tę funkcję, przekazując sam `node` jako argument. Następny przykład przedstawia definicję tych funkcji.

Przykład 10.19 Funkcje wywoływane przez funkcję pośredniczącą `parse`

```
def parse_Document(self, node): #(1)
        self.parse(node.documentElement)

    def parse_Text(self, node):    #(2)
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)

    def parse_Comment(self, node): #(3)
        pass

    def parse_Element(self, node): #(4)
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

1. `parse_Document` jest wywołany tylko raz, ponieważ jest tylko jeden węzeł klasy `Document` w dokumencie XML i tylko jeden obiekt klasy `Document` w przeparsowanej reprezentacji XML-a. Tu po prostu idziemy dalej i parsujemy część główną pliku gramatyki.

2. `parse_Text` jest wywoływany tylko na węzłach reprezentujących fragmenty tekstu. Funkcja wykonuje kilka specjalnych operacji związanych z automatycznym wstawianiem dużej litery na początku słowa pierwszego zdania, ale w innym wypadku po prostu dodaje reprezentowany tekst do listy.

3. `parse_Comment` jest tylko "przejażdżką"; metoda ta nic nie robi, ponieważ nie musimy się troszczyć o komentarze wstawione w plikach definiującym gramatykę. Pomimo tego, zauważmy, że nadal musimy zdefiniować funkcję i wyraźnie stwierdzić, żeby nic nie robiła. Jeśli funkcja nie będzie istniała, funkcja `parse` nawali tak szybko, jak napotka się na komentarz, ponieważ będzie próbowała znaleźć nieistniejącą funkcję `parse_Comment`. Definiując oddzielną funkcję dla każdego typu węzła, nawet jeśli nam ta funkcja nie jest potrzebna, pozwalamy ogólnej funkcji parsującej być prostą i krótką.

4. Metoda `parse_Element` jest w rzeczywistości funkcją pośredniczącą, opartą na nazwie znacznika elementu. Idea jest taka sama: weź odróżniające się od siebie elementy (elementy, które różnią się nazwą znacznika) i wyślij je do odpowiedniej, odrębnej funkcji. Konstruujemy napis typu `'do_xref'` (dla znacznika `<xref>`), znajdujemy funkcję o takiej nazwie i wywołujemy ją. I robimy podobnie dla

każdej innej nazwy znacznika, która zostanie znaleziona, oczywiście w pliku gramatyki (czyli znaczniki `<p>`, czy też `<choice>`).

W tym przykładzie funkcja pośrednicząca `parse` i `parse_Element` po prostu znajdują inne metody w tej samej klasie. Jeśli przetwarzanie jest bardzo złożone (lub mamy bardzo dużo nazw znaczników), powinniśmy rozdzielić swój kod na kilka oddzielnych modułów i wykorzystać dynamiczne importowanie, aby zaimportować każdy moduł, a następnie wywołać wszystkie potrzebne nam funkcje. Dynamiczne importowanie zostanie omówione w rozdziale “Programowanie funkcyjne”.

11.6 Obsługa argumentów linii poleceń

Obsługa argumentów linii poleceń

Python całkowicie wspomaga tworzenie programów, które mogą zostać uruchomione z linii poleceń, łącznie z argumentami linii poleceń, czy zarówno z krótkim lub długim stylem flag, które określają opcje. Nie ma to nic wspólnego z XML-em, ale omawiany skrypt wykorzystuje w dobry sposób linię poleceń, dlatego też nadeszła odpowiednia pora, aby to omówić.

Ciężko mówić o linii poleceń bez wiedzy, w jaki sposób argumenty linii poleceń są ujawniane do programu, dlatego też napiszmy prosty program, aby to zobaczyć.

Przykład 10.20 Wprowadzenie do `sys.argv`

```
#argecho.py
import sys

for arg in sys.argv: #(1)
    print arg
```

1. Każdy argument linii poleceń przekazany do programu, zostanie umieszczony w `sys.argv`, który jest właściwie listą. W tym miejscu wypisujemy każdy argument w oddzielnej linii.

Przykład 10.21 Zawartość `sys.argv`

```
[you@localhost py]$ python argecho.py          #(1)
argecho.py
[you@localhost py]$ python argecho.py abc def  #(2)
argecho.py
abc
def
[you@localhost py]$ python argecho.py --help   #(3)
argecho.py
--help
[you@localhost py]$ python argecho.py -m kant.xml #(4)
argecho.py
-m
kant.xml
```

1. Najpierw musimy sobie uświadomić, że `sys.argv` przechowuje nazwę uruchomionego skryptu. Wiedzę tę wykorzystamy później, w rozdziale "Programowanie funkcyjne". Na razie nie zamartwiaj się tym.

2. Argumenty linii poleceń są oddzielane przez spacje i każdy z nich ukazuje się w liście `sys.argv` jako oddzielny argument.

3. Flagi linii poleceń np. `--help`, także pokażą się jako osobne elementy w `sys.argv`.

4. Żeby było ciekawiej, niektóre z flag linii poleceń same przyjmują, wymagają argumentów. Na przykład, tutaj mamy jedną flagę (`-m`), która na dodatek także przyjmuje argument (w przykładzie `kant.xml`). Zarówno flaga sama w sobie, a

także argument flagi są kolejnymi elementami w liście `sys.argv`. Python w żaden sposób nie będzie próbował ich powiązać; otrzymamy samą listę.

Jak możemy zobaczyć, z pewnością mamy wszystkie informacje przekazane do linii poleceń, ale nie wyglądają na tak proste, aby z nich faktycznie skorzystać. Dla nieskomplikowanych programów, które przyjmują tylko jeden argument bez żadnych flag, możemy po prostu wykorzystać `sys.argv[1]`, aby się do niego dostać. Nie ma się czym wstydzić. Dla bardziej złożonych programów będzie potrzebny moduł `getopt`.

Przykład 10.22 Wprowadzenie do `getopt`

```
def main(argv):
    grammar = "kant.xml"                #(1)
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="]) #(2)
    except getopt.GetoptError:          #(3)
        usage()                         #(4)
        sys.exit(2)

[...]

 if __name__ == "__main__":
     main(sys.argv[1:])
```

1. Od razu zobacz na sam dół przykładu. Wywołujemy funkcję `main` z argumentem `sys.argv[1:]`. Zapamiętaj, `sys.argv[0]` jest nazwą skryptu, który uruchomiliśmy; nie martwimy się o to, w jaki sposób jest przetwarzana linia poleceń, więc odcinamy i przekazujemy resztę listy.

2. To najciekawsze miejsce w tym przykładzie. Funkcja `getopt` modułu `getopt` przyjmuje trzy parametry: listę argumentów (którą otrzymaliśmy z `sys.argv[1:]`), napis zawierający wszystkie możliwe jedno-znakowe flagi, które program akceptuje, a także listę dłuższych flag, które są odpowiednikami krótszych, jednoznakowych wersji. Na pierwszy rzut oka wydaje się to trochę zamotane, ale w dalszej części szerzej to omówimy.

3. Jeśli coś nie poszło pomyślnie podczas parsowania flag linii poleceń, `getopt` rzuca wyjątek, który następnie przechwytujemy. Informujemy funkcję `getopt` o wszystkich flagach, które rozumie nasz program, zatem ostatecznie oznacza, że użytkownik przekazał niektóre niezrozumiałe przez nas flagi.

4. Jest to praktycznie standard wykorzystywany w świecie Uniksa, kiedy do skryptu zostaną przekazane niezrozumiałe flagi, wypisujemy streszczoną pomoc dotyczącą użycia programu i wdzięcznie zakańczamy go. Dodajmy, że nie przedstawiliśmy tutaj funkcji `usage`. Jeszcze trzeba będzie ją gdzieś zaimplementować, aby wypisywała streszczenie pomocy; nie dzieje się to automatycznie.

Więc czym są te wszystkie parametry przekazane do funkcji `getopt`? A więc, pierwszy jest po prostu surową listą argumentów i flag przekazanych do linii poleceń (bez pierwszego elementu, czyli nazwy skryptu, który wycięliśmy przed wywołaniem funkcji `main`). Drugi parametr jest listą krótkich flag linii poleceń, które akceptuje skrypt.

```
"hg:d"

-h
    wyświetla streszczoną pomoc
-g ...
    korzysta z określonego pliku gramatyki lub URL-a
-d
    pokazuje informacje debugujące podczas parsowania
```

Pierwsza i trzecia flaga są zwykłymi, samodzielnymi flagami; możemy je określić lub nie. Flagi te wykonują pewne czynności (wypisują pomoc) lub zmieniają stan (włączają debugowanie). Jakkolwiek, za drugą flagą (`-g`) musi się znaleźć pewien argument, który będzie nazwą pliku gramatyki, który ma zostać przeczytany. W rzeczywistości może być nazwą pliku lub adresem strony strony web, ale my jeszcze nie wiemy, czym jest (będziemy z tym kombinować później), ale wiemy, że ma być czymś. Poinformowaliśmy `getopt` o tym, że ma być coś za tą flagą, poprzez wstawienie w drugim parametrze dwukropka po literze `g`.

Żeby to bardziej skomplikować, skrypt akceptuje zarówno krótkie flagi (np. `-h`, jak i długie flagi (jak `--help`), a my chcemy, żeby służyły one do tego samego. I po to jest trzeci parametr w `getopt`. Określa on listę długich flag, które odpowiadają krótkim flagom zdefiniowanym w drugim parametrze.

```
["help", "grammar="]

--help
    wyświetla streszczoną pomoc
--grammar ...
    korzysta z określonego pliku gramatyki lub URL-a
```

Zwróćmy uwagę na trzy sprawy:

1. Wszystkie długie flagi w linii poleceń są poprzedzone dwoma myślnikami, ale podczas wywoływania `getopt` nie dołączamy tych myślników.

2. Po fladze `--grammar` musi zawsze wystąpić dodatkowy argument, identycznie jak z flagą `-g`. Informujemy o tym poprzez znak równości w `"grammar="`.

3. Lista długich flag jest krótsza niż lista krótkich flag, ponieważ flaga `-d` nie ma swojego dłuższego odpowiednika. Jedynie `-d` będzie włączał debugowanie. Jednak porządek krótkich i długich flag musi być ten sam, dlatego też najpierw musimy określić wszystkie krótkie flagi odpowiadające dłuższym flagom, a następnie pozostałą część krótszych flag, które nie mają swojego dłuższego odpowiednika.

Jeszcze się nie pogubiłeś? To spójrz na właściwy kod i zobacz, czy nie staje się dla ciebie zrozumiały.

Przykład 10.23 Obsługa argumentów linii poleceń w kgp/kgp.py

```
def main(argv):                          #(1)
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
```

```
except getopt.GetoptError:
    usage()
    sys.exit(2)
for opt, arg in opts:                #(2)
    if opt in ("-h", "--help"):      #(3)
        usage()
        sys.exit()
    elif opt == '-d':                #(4)
        global _debug
        _debug = 1
    elif opt in ("-g", "--grammar"): #(5)
        grammar = arg

source = "".join(args)               #(6)

k = KantGenerator(grammar, source)
print k.output()
```

1. Zmienna `grammar` będzie przechowywać ścieżkę do pliku gramatyki, z którego będziemy korzystać. W tym miejscu inicjalizujemy ją tak, aby w przypadku, gdy nie zostanie określona w linii poleceń (za pomocą flagi `-g` lub `--grammar`) miała jakąś domyślną wartość.

2. Zmienną `opts`, którą otrzymujemy z wartość zwróconej przez `getopt`, przechowuje listę krotek: flagę i argument. Jeśli flaga nie przyjmuje argumentu, to argument będzie miał wartość `None`. Ułatwia to wykonywanie pętli na flagach.

3. `getopt` kontroluje, czy flagi linii poleceń są akceptowalne, ale nie wykonuje żadnej konwersji między długimi, a krótkimi flagami. Jeśli określimy flagę `-h`, `opt` będzie zawierać `-h"`, natomiast jeśli określimy flagę `--help`, `opt` będzie zawierać `--help"`. Zatem musimy kontrolować obydwa warianty.

4. Pamiętamy, że fladze `-d` nie odpowiada żadna dłuższa wersja, dlatego też kontrolujemy tylko tę krótką flagę. Jeśli zostanie ona odnaleziona, ustawiamy globalną zmienną, do której później będziemy się odwoływać, aby wypisywać informacje debugujące. (Flaga ta była wykorzystywana podczas projektowania skryptu. A co, myślisz, że wszystkie przedstawione przykłady działały od razu??)

5. Jeśli znajdziemy plik gramatyki spotykając flagę `-g` lub −−`grammar`, zapisujemy argument, który następuje po tej fladze (przechowywany w zmiennej `arg`), do zmiennej `grammar`, nadpisując przy tym domyślną wartość, zainicjalizowaną na początku funkcji `main`.

6. Ok. Wykonaliśmy pętlę przez wszystkie flagi i przetworzyliśmy je. Oznacza to, że pozostała część musi być argumentami linii poleceń, a zostały one zwrócone przez funkcje `getopt` do zmiennej `args`. W tym przypadku traktujemy je jako materiał źródłowy dla parsera. Jeśli nie zostały określone żadne argumenty linii poleceń, `args` będzie pustą listą, więc `source` w wyniku tego będzie pustym napisem.

11.7 Skrypty i strumienie - wszystko razem

Wszystko razem

Przemierzyliśmy kawał drogi. Zatrzymajmy się na chwilę i zobaczmy jak te wszystkie elementy do siebie pasują. Zaczniemy od skryptu, który pobiera argumenty z linii poleceń używając modułu `getopt`.

```
def main(argv):
...
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
...
    for opt, arg in opts:
...
```

Tworzymy nową instancję klasy `KantGenerator` i przekazujemy jej plik z gramatyką oraz źródło, które może być, ale nie musi, podane w linii poleceń.

```
k = KantGenerator(grammar, source)
```

Instancja klasy `KantGenerator` automatycznie wczytuje gramatykę, która jest plikiem XML. Wykorzystujemy naszą funkcję `openAnything` do otwarcia pliku (który może być ulokowany lokalnie lub na zdalnym serwerze), następnie używamy wbudowanego zestawu funkcji parsujących `minidom` do sparsowania XML-a do postaci drzewa obiektów Pythona.

```
def _load(self, source):
    sock = toolbox.openAnything(source)
    xmldoc = minidom.parse(sock).documentElement
    sock.close()
```

Ach i po drodze wykorzystujemy naszą wiedzę o strukturze dokumentu XML do utworzenia małego bufora referencji, którymi są po prostu elementy dokumentu XML.

```
def loadGrammar(self, grammar):
    for ref in self.grammar.getElementsByTagName("ref"):
        self.refs[ref.attributes["id"].value] = ref
```

Jeśli został podany jakiś materiał źródłowy w linii poleceń, używamy go. W przeciwnym razie na podstawie gramatyki wyszukujemy referencję na najwyższym poziomie (tą do której nie mają odnośników żadne inne elementy) i używamy jej jako punktu startowego.

```
def getDefaultSource(self):
    xrefs = {}
    for xref in self.grammar.getElementsByTagName("xref"):
    xrefs[xref.attributes["id"].value] = 1
    xrefs = xrefs.keys()
    standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
    return '<xref id="%s"/>' % random.choice(standaloneXrefs)
```

Teraz przedzieramy się przez materiał źródłowy. Ten materiał to także XML i parsujemy go węzeł po węźle. Aby podzielić nieco kod i uczynić go łatwiejszym w utrzymaniu, używamy oddzielnych funkcji obsługi (ang. *handlers*) dla każdego typu węzła.

```
def parse_Element(self, node):
    handlerMethod = getattr(self, "do_%s" % node.tagName)
    handlerMethod(node)
```

Przelatujemy przez gramatykę, parsując wszystkie elementy potomne każdego elementu `p`,

```
def do_p(self, node):
...
    if doit:
        for child in node.childNodes: self.parse(child)
```

zastępując elementy `choice` losowym elementem potomnym,

```
def do_choice(self, node):
    self.parse(self.randomChildElement(node))
```

i zastępując elementy `xref` losowym elementem potomnym odpowiedniego elementu `ref`, który wcześniej został zachowany w buforze.

```
def do_xref(self, node):
    id = node.attributes["id"].value
    self.parse(self.randomChildElement(self.refs[id]))
```

W końcu parsujemy wszystko do zwykłego tekstu,

```
def parse_Text(self, node):
    text = node.data
...
    self.pieces.append(text)
```

który wypisujemy.

```
def main(argv):
...
    k = KantGenerator(grammar, source)
    print k.output()
```

11.8 Skrypty i strumienie - podsumowanie

Wszystko razem

Python posiada zestaw potężnych bibliotek do parsowania i manipulacji dokumentami XML. Moduł `minidom` parsuje plik XML zamieniając go w obiekt Pythona i pozwalając na swobodny dostęp do dowolnych jego elementów. Idąc dalej, w rozdziale tym pokazaliśmy, w jaki sposób użyć Pythona do tworzenia "prawdziwych" skryptów linii poleceń przyjmujących argumenty, obsługujących różne błędy, a nawet potrafiących pobrać dane z wyjścia innego programu.

Zanim przejdziemy do następnego rozdziału, powinniśmy z łatwością:

- posługiwać się standardowym strumień wejścia, wyjścia i błędów.
- definiować dynamiczne funkcje pośredniczące za pomocą funkcji `getattr`.
- korzystać z argumentów linii poleceń i sprawdzać ich poprawność za pomocą modułu `getopt`.

Rozdział 12

HTTP

12.1 HTTP

Nurkujemy

Do tej pory nauczyliśmy się już przetwarzać HTML i XML, zobaczyliśmy jak pobrać stronę internetową, a także jak parsować dane XML pobrane poprzez URL. Zagłębmy się teraz nieco bardziej w tematykę usług sieciowych HTTP.

W uproszczeniu, usługi sieciowe HTTP są programistycznym sposobem wysyłania i odbierania danych ze zdalnych serwerów, wykorzystując do tego bezpośrednio transmisje po HTTP. Jeżeli chcesz pobrać dane z serwera, użyj po prostu metodę `GET` protokołu HTTP; jeżeli chcesz wysłać dane do serwera, użyj POST. (Niektóre, bardziej zaawansowane API serwisów HTTP definiują także sposób modyfikacji i usuwania istniejących danych – za pomocą metod HTTP `PUT` i `DELETE`). Innymi słowy, "czasowniki" wbudowane w protokół HTTP (`GET`, `POST`, `PUT` i `DELETE`) pośrednio przekształcają operacje HTTP na operacje na poziomie aplikacji: odbierania, wysyłania, modyfikacji i usuwania danych.

Główną zaletą tego podejścia jest jego prostota. Popularność tego rozwiązania została udowodniona poprzez ogromną liczbę różnych witryn. Dane – najczęściej w formacie XML – mogą być wygenerowane i przechowane statycznie, albo też generowane dynamicznie poprzez skrypty po stronie serwera, i inne popularne języki, włączając w to bibliotekę HTTP. Łatwiejsze jest także debugowanie, ponieważ możemy wywołać dowolną usługę sieciową w dowolnej przeglądarce internetowej i obserwować zwracane surowe dane. Współczesne przeglądarki także czytelnie sformatują otrzymane dane XML, i pozwolą na szybką nawigację wśród nich.

Przykłady użycia czystych usług sieciowych typu XML poprzez HTTP:

- Amazon API (http://www.amazon.com/webservices) pozwala na pobieranie informacji o produktach oferowanych w sklepie Amazon.com
- National Weather Service (http://www.nws.noaa.gov/alerts/) (United States) i Hong Kong Observatory (http://demo.xml.weather.gov.hk/) (Hong Kong) oferuje informowanie o pogodzie w formie usługi sieciowej
- Atom API (http://atomenabled.org/) – zarządzanie zawartością stron www

W kolejnych rozdziałach zapoznamy się z różnymi API, które wykorzystują protokół HTTP jako nośnik do wysyłania i odbierania danych, ale które nie przekształcają operacji na poziomie aplikacji na operacje w HTTP (zamiast tego tunelują wszystko poprzez HTTP `POST`). Ale ten rozdział koncentruje się na wykorzystywaniu metody `GET` protokołu HTTP do pobierania danych z serwera – poznamy kilka cech HTTP, które pozwolą nam jak najlepiej wykorzystać możliwości czystych usług sieciowych HTTP.

Poniżej jest bardziej zaawansowana wersja modułu openanything.py, który przedstawiliśmy w poprzednim rozdziale:

```
import urllib2, urlparse, gzip
from StringIO import StringIO

USER_AGENT = 'OpenAnything/%s +http://diveintopython.org/http_web_services/'
% __version__

class SmartRedirectHandler(urllib2.HTTPRedirectHandler):
```

```
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301(
            self, req, fp, code, msg, headers)
        result.status = code
        return result

    def http_error_302(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
        return result

class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):
    def http_error_default(self, req, fp, code, msg, headers):
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code
        return result

def openAnything(source, etag=None, lastmodified=None, agent=USER_AGENT):
    u"""URL, nazwa pliku lub łańcuch znaków --> strumień

    Funkcja ta pozwala tworzyć parsery, które przyjmują jakieś źródło wejścia
    (URL, ścieżkę do pliku lokalnego lub gdzieś w sieci lub dane w postaci łańcucha znaków),
    a następnie zaznajamia się z nim w odpowiedni sposób. Zwracany obiekt będzie
    posiadał wszystkie podstawowe metody czytania wejścia (read, readline, readlines).
    Ponadto korzystamy z .close(), gdy obiekt już nam nie będzie potrzebny.

    Kiedy zostanie podany argument etag, zostanie on wykorzystany jako wartość
    nagłówka żądania URL-a If-None-Match.

    Jeśli argument lastmodified zostanie podany, musi być on formie
    łańcucha znaków określającego czas i datę w GMT.
    Data i czas sformatowana w tym łańcuchu zostanie wykorzystana
    jako wartość nagłówka żądania If-Modified-Since.

    Jeśli argument agent zostanie określony, będzie on wykorzystany
    w nagłówku żądania User-Agent.
    """

    if hasattr(source, 'read'):
        return source

    if source == '-':
        return sys.stdin

    if urlparse.urlparse(source)[0] == 'http':
        # otwiera URL za pomocą urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)
```

```
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)
        if etag:
            request.add_header('If-None-Match', etag)
        request.add_header('Accept-encoding', 'gzip')
        opener = urllib2.build_opener(SmartRedirectHandler(), DefaultErrorHandler())
        return opener.open(request)

    # próbuje otworzyć za pomocą wbudowanej funkcji open (jeśli source to nazwa pliku)
    try:
        return open(source)
    except (IOError, OSError):
        pass

    # traktuje source jak łańcuch znaków
    return StringIO(str(source))

def fetch(source, etag=None, lastmodified=None, agent=USER_AGENT):
    u"""Pobiera dane z URL, pliku, strumienia lub łańcucha znaków"""
    result = {}
    f = openAnything(source, etag, lastmodified, agent)
    result['data'] = f.read()
    if hasattr(f, 'headers'):
        # zapisuje ETag, jeśli go wysłał do nas serwer
        result['etag'] = f.headers.get('ETag')
        # zapisuje nagłówek Last-Modified, jeśli został do nas wysłany
        result['lastmodified'] = f.headers.get('Last-Modified')
        if f.headers.get('content-encoding') == 'gzip':
            # odkompresowuje otrzymane dane, ponieważ są one zakompresowane jako gzip
            result['data'] = gzip.GzipFile(fileobj=StringIO(result['data'])).read()
    if hasattr(f, 'url'):
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):
        result['status'] = f.status
    f.close()
    return result
```

12.2 Jak nie pobierać danych poprzez HTTP

Jak nie pobierać danych poprzez HTTP

Załóżmy, że chcemy pobrać jakiś zasób poprzez HTTP, jak np. RSS (Really Simple Syndication). Jednak nie chcemy pobrać go tylko jednorazowo, lecz cyklicznie, np. co godzinę, aby mieć najświeższe informacje ze strony, która udostępnia RSS. Zróbmy to najpierw w bardzo prosty i szybki sposób, a potem zobaczymy jak można to zrobić lepiej.

Przykład 11.2 Pobranie RSS w szybki i prosty sposób

```
>>> import urllib
>>> data = urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()    #(1)
>>> print data
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>

  [... ciach ...]
```

1. Pobranie czegokolwiek poprzez HTTP jest w Pythonie niesamowicie łatwe; właściwie jest to jedna linijka kodu. Moduł `urllib` ma bardzo poręczną funkcję `urlopen`, która przyjmuje na wejściu adres strony, a zwraca *obiekt pliko-podobny*, z którego można odczytać całą zawartość strony przy pomocy metody `read`. Prościej już nie może być.

A więc co jest z tym nie tak? Cóż, do szybkiej próby podczas testowania, czy programowania, to w zupełności wystarczy. Chcieliśmy zawartość RSS-a i mamy ją. Ta sama technika działa dla każdej strony internetowej. Jednak gdy zaczniemy myśleć w kategoriach serwisów internetowych, z których chcemy korzystać regularnie – pamiętajmy, że chcieliśmy pobierać tego RSS-a co godzinę – wtedy takie działanie staje się niewydajne i prymitywne.

Porozmawiajmy o podstawowych cechach HTTP.

12.3 Właściwości HTTP

Możemy wyróżnić pięć ważnych właściwości HTTP, z których powinniśmy korzystać.

`User-Agent`

User-Agent jest to po prostu sposób w jaki klient może poinformować serwer kim jest w trakcie żądania strony internetowej, RSS (Really Simple Syndication) lub jakiejkolwiek usługi internetowej poprzez HTTP. Gdy klient zgłasza żądanie do danego zasobu, to zawsze powinien informować kim jest tak szczegółowo, jak to tylko możliwe. Pozwala to administratorowi serwera na skontaktowanie się z programistą, twórcą aplikacji klienckiej, jeśli coś idzie nie tak.

Domyślnie Python wysyła standardowy nagłówek postaci `User-Agent: Python-urllib/1.15`. W następnej sekcji zobaczymy jak to zmienić na coś bardziej szczegółowego.

Przekierowania

Czasami zasoby zmieniają położenie. Witryny internetowe są reorganizowane a strony przesuwane pod nowe adresy. Nawet usługi internetowe ulegają reorganizacji. RSS spod adresu http://example.com/index.xml może się przesunąć do http://example.com/xml/atom.xml. Może się także przesunąć cała domena, gdy reorganizacja jest przeprowadzana na większą skalę, np. http://www.example.com/index.xml może zostać przekierowana do http://server-farm-1.example.com/index.xml.

Zawsze gdy żądamy jakiegoś zasobu od serwera HTTP, serwer ten dołącza do swojej odpowiedzi kod statusu. Kod statusu 200 oznacza "wszystko w porządku, oto strona o którą prosiliśmy". Kod statusu 404 oznacza "strona nieznaleziona". (Prawdopodobnie spotkaliśmy się z błędami 404 podczas surfowania w sieci.)

Protokół HTTP ma dwa różne sposoby, aby dać do zrozumienia, że dany zasób został przesunięty. Kod statusu 302 jest to tymczasowe przekierowanie i oznacza on "ups, to zostało tymczasowo przesunięte tutaj" (a ten tymczasowy adres umieszczany jest w nagłówku `Location:`). Kod statusu 301 jest to przekierowanie trwałe i oznacza "ups, to zostało przesunięte na stałe" (a nowy adres jest podawany w nagłówku `Location:`). Gdy otrzymamy kod statusu 302 i nowy adres, to specyfikacja HTTP mówi, że powinniśmy użyć nowego adresu, aby pobrać to czego dotyczyło żądanie, ale następnym razem przy próbie dostępu do tego samego zasobu powinniśmy spróbować ponownie starego adresu. Natomiast gdy dostaniemy kod statusu 301 i nowy adres, to powinniśmy już od tego momentu używać tylko tego nowego adresu.

`urllib.urlopen` automatycznie "śledzi" przekierowania, jeśli otrzyma stosowny kod statusu od serwera HTTP, ale niestety nie informuje o tym fakcie. Ostatecznie otrzymujemy dane, o które prosiliśmy, ale nigdy nie wiadomo, czy biblioteka nie podążyła samodzielnie za przekierowaniem. Tak więc nieświadom niczego dalej możemy próbować korzystać ze starego adresu i za każdym razem nastąpi przekierowanie pod nowy adres. To powoduje wydłużenie drogi, co nie jest zbyt wydajne! W dalszej części tego rozdziału zobaczymy, jak sobie radzić z trwałymi przekierowaniami właściwie i wydajnie.

`Last-Modified/If-Modified-Since`

Niektóre dane zmieniają się bez przerwy. Strona domowa `cnn.com` jest aktualizowana co pięć minut. Z drugiej strony strona domowa `google.com` zmienia się raz na kilka tygodni (gdy wrzucają jakieś świąteczne logo lub reklamują jakąś nową usługę). Usługi internetowe nie różnią się pod tym względem. Serwer zwykle wie kiedy dane, które pobieramy się ostatnio zmieniły, a protokół HTTP pozwala serwerowi na dołączenie tej daty ostatniej modyfikacji do żądanych danych.

Gdy poprosimy o te same dane po raz drugi (lub trzeci, lub czwarty), możemy podać serwerowi datę ostatniej modyfikacji (ang. last-modified date), którą dostaliśmy ostatnim razem. Wysyłamy serwerowi nagłówek `If-Modified-Since` wraz ze swoim żądaniem, wraz z datą otrzymaną od serwera ostatnim razem. Jeśli dane nie uległy zmianie od tamtego czasu, to serwer odsyła specjalny kod statusu 304, który oznacza "dane nie zmieniły się od czasu, gdy o nie ostatnio pytałeś". Dlaczego jest to lepsze rozwiązanie? Bo gdy serwer odsyła kod 304, to nie wysyła ponownie danych. Wszystko co otrzymujemy to kod statusu. Tak więc nie musimy ciągle pobierać tych samych danych w kółko, jeśli nie uległy zmianie. Serwer zakłada, że już mamy te dane zachowane gdzieś lokalnie.

Wszystkie nowoczesne przeglądarki internetowe wspierają sprawdzanie daty ostatniej modyfikacji. Być może kiedyś odwiedziliśmy jakąś stronę jednego dnia, a potem odwiedziliśmy ją ponownie następnego dnia i zauważyliśmy, że nie uległa ona zmianie, a jednocześnie zadziwiająco szybko się załadowała. Przeglądarka zachowała zawartość tej strony w lokalnym buforze podczas pierwszej wizyty, a podczas drugiej automatycznie wysłała datę ostatniej modyfikacji otrzymaną za pierwszym razem. Serwer po prostu odpowiedział kodem `304: Not Modified`, a więc przeglądarka wiedziała, że musi załadować stronę z lokalnego bufora. Usługi internetowe mogą być także takie sprytne.

Biblioteka URL Pythona nie ma wbudowanego wsparcia dla kontroli daty ostatniej modyfikacji, ale ponieważ możemy dodawać dowolne nagłówki do każdego żądania i czytać dowolne nagłówki z każdej odpowiedzi, to możemy dodać taką kontrolę samodzielnie.

`ETag/If-None-Match`

Znaczniki `ETag` są alternatywnym sposobem na osiągnięcie tego samego celu, co poprzez kontrolę daty ostatniej modyfikacji: nie pobieramy ponownie danych, które się nie zmieniły. A działa to tak: serwer wysyła jakąś sumę kontrolną danych (w nagłówku `ETag`) razem z żądanymi danymi. Jak ta suma kontrolna jest ustalana, to zależy wyłącznie od serwera. Gdy po raz drugi chcemy pobrać te same dane, dołączamy sumę kontrolną z nagłówka `ETag` w nagłówku `If-None-Match:` i jeśli dane się nie zmieniły serwer odeśle kod statusu 304. Tak jak w przypadku kontroli daty ostatniej modyfikacji, serwer odsyła tylko kod 304; nie wysyła po raz drugi tych samych danych. Poprzez dołączenie sumy kontrolnej `ETag` do drugiego żądania mówimy serwerowi, iż nie ma potrzeby, aby wysyłał po raz drugi tych samych danych, jeśli nadal odpowiadają one tej sumie kontrolnej, ponieważ cały czas mamy dane pobrane ostatnio.

Biblioteka URL Pythona nie ma wbudowanego wsparcia dla znaczników `ETag`, ale zobaczymy, jak można je dodać w dalszej części tego rozdziału.

Kompresja

Ostatnią istotną właściwością HTTP, którą możemy ustawić, jest kompresja `gzip`. Gdy mówimy o usługach sieciowych za pośrednictwem HTTP, to zwykle mówimy o przesyłaniu XML-i tam i z powrotem. XML jest tekstem i to zwykle całkiem rozwlekłym tekstem, a tekst dobrze się kompresuje. Gdy żądamy jakiegoś zasobu poprzez HTTP, to możemy poprosić serwer, jeśli ma jakieś nowe dane do wysłania, aby wysłał je w formie skompresowanej. Dołączamy wtedy nagłówek `Accept-encoding: gzip` do żądania i jeśli serwer wspiera kompresję, odeśle on dane skompresowane w formacie `gzip` i oznaczy je nagłówkiem `Content-encoding: gzip`.

Biblioteka URL Pythona nie ma wbudowanego wsparcia dla kompresji `gzip` jako takiej, ale możemy dodawać dowolne nagłówki do żądania a Python posiada oddzielny moduł `gzip`, który zawiera funkcje, których można użyć do dekompresji danych samodzielnie.

Zauważmy, że nasz jednolinijkowy skrypt pobierający RSS nie uwzględnia żadnej z tych właściwości HTTP. Zobaczmy, jak możemy go udoskonalić.

12.4 Debugowanie serwisów HTTP

Debugowanie serwisów HTTP

Na początek włączmy debugowanie w pythonowej bibliotece HTTP i zobaczmy co zostanie przesłane. Wiadomości zdobyte poprzez przeanalizowanie wypisanych informacji debugujących, będą przydatne w tym rozdziale, gdy będziemy chcieli dodać nowe możliwości do naszego programu.

Przykład 11.3 Debugowanie HTTP

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1             #(1)
>>> import urllib
>>> feeddata = urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()
connect: (diveintomark.org, 80)                       #(2)
send: '
GET /xml/atom.xml HTTP/1.0                            #(3)
Host: diveintomark.org                                #(4)
User-agent: Python-urllib/1.15                        #(5)
'
reply: 'HTTP/1.1 200 OK\r\n'                          #(6)
header: Date: Wed, 14 Apr 2004 22:27:30 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT  #(7)
header: ETag: "e8284-68e0-4de30f80"                   #(8)
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

1. `urllib` jest zależny od innej standardowej biblioteki Pythona: `httplib`. Zwykle nie musimy importować modułu `httplib` bezpośrednio (`urllib` robi to automatycznie), ale w tym wypadku to zrobiliśmy, a więc możemy ustawić flagę trybu debugowania w klasie `HTTPConnection`, którą moduł `urllib` wykorzystuje wewnętrznie do nawiązania połączenia z serwerem HTTP. To jest niezwykle przydatna technika. Kilka innych bibliotek Pythona ma podobne flagi trybu debugowania, ale nie ma jakiegoś szczególnego standardu nazywania ich i ustawiania; trzeba przeczytać dokumentację każdej biblioteki, aby zobaczyć, czy taka flaga jest dostępna.

2. Teraz gdy już flagę debugowania mamy ustawioną, informacje na temat żądań HTTP i odpowiedzi są wyświetlane w czasie rzeczywistym. Pierwszą rzeczą jaką możemy zauważyć jest to, iż łączymy się z serwerem `diveintomark.org` na porcie 80, który jest standardowym portem dla HTTP.

3. Gdy zgłaszamy żądanie po zasobów RSS, `urllib` wysyła trzy linie do serwera. Pierwsza linia zawiera polecenie HTTP i ścieżkę do zasobu (bez nazwy domeny). Wszystkie żądania w tym rozdziale będą używały polecenia `GET`, ale w następnym rozdziale o SOAP zobaczymy, że tam do wszystkiego używane jest polecenie `POST`. Podstawowa składnia jest jednak taka sama niezależnie od polecenia.

4. Druga linia odnosi się do nagłówka `Host`, który zawiera nazwę domeny serwisu, do którego kierujemy żądanie. Jest to istotne, ponieważ pojedynczy serwer HTTP może obsługiwać wiele oddzielnych domen. Na tym serwerze jest aktualnie obsługiwanych 12 domen; inne serwery mogą obsługiwać setki lub nawet tysiące.

5. Trzecia linia to nagłówek `User-Agent`. To co tu widać, to jest standardowy nagłówek `User-Agent` dodany domyślnie przez bibliotekę `urllib`. W następnej sekcji pokażemy jak zmienić to na coś bardziej konkretnego.

6. Serwer odpowiada kodem statusu i kilkoma nagłówkami (być może z jakimiś danymi, które zostały zachowane w zmiennej `feeddata`). Kodem statusu jest tutaj liczba 200, która oznacza "wszystko w porządku, proszę to dane o które prosiłeś". Serwer także podaje datę odpowiedzi na żądanie, trochę informacji na temat samego serwera i typ zawartości (ang. *content type*) zwracanych danych. W zależności od aplikacji, może być to przydatne lub też nie. Zauważmy, że zażądaliśmy RSS-a i faktycznie otrzymaliśmy RSS-a (`application/atom+xml` jest to zarejestrowany typ zawartości dla zasobów RSS).

7. Serwer podaje, kiedy ten RSS był ostatnio modyfikowany (w tym wypadku około 13 minut temu). Możemy odesłać tę datę serwerowi z powrotem następnym razem, gdy zażądamy tego samego zasobu, a serwer będzie mógł wykonać sprawdzenie daty ostatniej modyfikacji.

8. Serwer podaje także, że ten RSS ma sumę kontrolną `ETag` o wartości "e8284-68e0-4de30f80". Ta suma kontrolna sama w sobie nie ma żadnego znaczenia; nie można z nią zrobić nic poza wysłaniem jej z powrotem do serwera przy następnej próbie dostępu do tego zasobu. Wtedy serwer może jej użyć do sprawdzenia, czy dane się zmieniły od ostatniego razu czy nie.

12.5 Ustawianie User-Agent

Ustawianie `User-Agent`

Pierwszym krokiem, aby udoskonalić swój klient serwisu HTTP jest właściwe zidentyfikowanie siebie za pomocą nagłówka `User-Agent`. Aby to zrobić, potrzebujemy wyjść poza prosty `urllib` i zanurkować w `urllib2`.

Przykład 11.4 Wprowadzenie do `urllib2`

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1                              #(1)
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml') #(2)
>>> opener = urllib2.build_opener()                                    #(3)
>>> feeddata = opener.open(request).read()                             #(4)
connect: (diveintomark.org, 80)
 send: '
 GET /xml/atom.xml HTTP/1.0
 Host: diveintomark.org
 User-agent: Python-urllib/2.1
 '
 reply: 'HTTP/1.1 200 OK\r\n'
 header: Date: Wed, 14 Apr 2004 23:23:12 GMT
 header: Server: Apache/2.0.49 (Debian GNU/Linux)
 header: Content-Type: application/atom+xml
 header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
 header: ETag: "e8284-68e0-4de30f80"
 header: Accept-Ranges: bytes
 header: Content-Length: 26848
 header: Connection: close
```

1. Jeśli w dalszym ciągu masz otwarty IDE na przykładzie z poprzedniego podrozdziału, możesz ten punkt pominąć. Polecenie to włącza debugowanie HTTP, dzięki której możemy zobaczyć, co rzeczywiście wysyłamy i co zostaje przysłane do nas.

2. Pobieranie zasobów HTTP za pomocą `urllib2` składa się z trzech krótkich etapów. Pierwszy krok to utworzenie obiektu żądania (instancji klasy `Request`). Klasa `Request` przyjmuje jako parametr URL zasobów, z których będziemy ewentualnie pobierać dane. Dodajmy, że za pomocą tego kroku jeszcze nic nie pobieramy.

3. Drugim krokiem jest zbudowanie *otwieracza* (ang. *opener*) URL-a. Może on przyjąć dowolną liczbę funkcji obsługi, które kontrolują, w jaki sposób obsługiwać odpowiedzi. Ale możemy także zbudować *otwieracz* bez podawania żadnych funkcji obsługi, a co robimy w tym fragmencie. Później, kiedy będziemy omawiać przekierowania, zobaczymy w jaki sposób zdefiniować własne funkcje obsługi.

4. Ostatnim krokiem jest kazanie otwieraczowi, aby korzystając z utworzonego obiektu żądania otworzył URL. Widzimy, że wszystkie otrzymane informacje

debugujące zostały wypisane. W tym kroku tak pobieramy zasoby, a zwrócone dane przechowujemy w `feeddata`.

Przykład 11.5 Dodawanie nagłówków do żądania

```
>>> request                                                 #(1)
<urllib2.Request instance at 0x00250AA8>
>>> request.get_full_url()
http://diveintomark.org/xml/atom.xml
>>> request.add_header('User-Agent',
...     'OpenAnything/1.0 +http://diveintopython.org/')    #(2)
>>> feeddata = opener.open(request).read()                  #(3)
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: OpenAnything/1.0 +http://diveintopython.org/   #(4)
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:45:17 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284-68e0-4de30f80"
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

1. Kontynuujemy z poprzedniego przykładu. Mamy już utworzony obiekt `Request` z URL-em, do którego chcemy się dostać.

2. Za pomocą metody `add_header` obiektu klasy `Request`, możemy dodać do żądania dowolny nagłówek HTTP. Pierwszy argument jest nagłówkiem, a drugi jest wartością dostarczoną do tego nagłówka. Konwencja dyktuje, aby `User-Agent` powinien być w takim formacie: nazwa aplikacji, następnie slash, a potem numer wersji. Pozostała część może mieć dowolną formę, jednak znaleźć mnóstwo wariacji wykorzystania tego nagłówka, ale gdzieś powinno się umieścić URL aplikacji. `User-Agent` jest zazwyczaj logowany przez serwer wraz z innymi szczegółami na temat twojego żądania. Włączając URL twojej aplikacji, pozwalasz administratorom danego serwera skontaktować się z tobą, jeśli jest coś źle.

3. Obiekt `opener` utworzony wcześniej może być także ponownie wykorzystany. Ponownie wysyłamy te same dane, ale tym razem ze zmienionym nagłówkiem `User-Agent`.

4. Tutaj wysyłamy ustawiony przez nas nagłówek `User-Agent`, w miejsce domyślnego, wysyłanego przez Pythona. Jeśli będziesz uważnie patrzył, zobaczysz, że zdefiniowaliśmy nagłówek `User-Agent`, ale tak naprawdę wysłaliśmy nagłówek `User-agent`. Widzisz różnicę? `urllib2` zmienia litery w ten sposób, że tylko pierwsza litera jest wielka. W rzeczywistości nie ma to żadnego znaczenia. Specyfikacja HTTP mówi, że wielkość liter nazwy pola nagłówka nie jest ważna.

12.6 Korzystanie z Last-Modified i ETag

Korzystanie z Last-Modified i ETag

Teraz gdy już wiesz jak dodawać własne nagłówki do swoich żądań HTTP, zobaczmy jak wykorzystać nagłówki Last-Modified i ETag.

Poniższe przykłady pokazują wyjście z wyłączonym debugowaniem. Jeśli nadal masz je włączone (tak jak w poprzedniej sekcji), możesz je wyłączyć poprzez takie ustawienie: `httplib.HTTPConnection.debuglevel = 0`. Albo możesz pozostawić debugowanie włączone, jeśli to Ci pomoże.

Przykład 11.6 Testowanie `Last-Modified`

```
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener()
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.dict                       #(1)
{'date': 'Thu, 15 Apr 2004 20:42:41 GMT',
 'server': 'Apache/2.0.49 (Debian GNU/Linux)',
 'content-type': 'application/atom+xml',
 'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
 'etag': '"e842a-3e53-55d97640"',
 'content-length': '15955',
 'accept-ranges': 'bytes',
 'connection': 'close'}
>>> request.add_header('If-Modified-Since',
...     firstdatastream.headers.get('Last-Modified'))  #(2)
>>> seconddatastream = opener.open(request)            #(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\urllib2.py", line 326, in open
    '_open', req)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 901, in http_open
    return self.do_open(httplib.HTTP, req)
  File "c:\python23\lib\urllib2.py", line 895, in do_open
    return self.parent.error('http', req, fp, code, msg, hdrs)
  File "c:\python23\lib\urllib2.py", line 352, in error
    return self._call_chain(*args)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 412, in http_error_default
    raise HTTPError(req.get_full_url(), code, msg, hdrs, fp)
urllib2.HTTPError: HTTP Error 304: Not Modified
```

1. Pamiętasz wszystkie te nagłówki HTTP, które były wyświetlane przy włączonym debugowaniu? To jest sposób na dotarcie do nich programowo: `firstdatastream.headers` jest obiektem działającym jak słownik i pozwala on na pobranie każdego nagłówka zwracanego przez serwer HTTP.

2. Przy drugim żądaniu dodajemy nagłówek `If-Modified-Since` z datą ostatniej modyfikacji z pierwszego żądania. Jeśli data nie uległa zmianie, serwer powinien zwrócić kod statusu 304.

3. To wystarczy do stwierdzenia, że dane nie uległy zmianie. Możemy zobaczyć w zrzucie błędów, że `urllib2` rzucił wyjątek specjalny: `HTTPError` w odpowiedzi na kod statusu 304. To jest trochę niezwykłe i nie całkiem pomocne. W końcu to nie jest błąd; specjalnie poprosiliśmy serwer o nie przesyłanie żadnych danych, jeśli nie uległy one zmianie i dane nie uległy zmianie, a więc serwer powiedział, iż nie wysłał żadnych danych. To nie jest błąd; to jest dokładnie to czego oczekiwaliśmy.

`urllib2` rzuca wyjątek `HTTPError` także w sytuacjach, które można traktować jak błędy, np. 404 (strona nieznaleziona). Właściwie to rzuca on wyjątek `HTTPError` dla każdego kodu statusu innego niż 200 (OK), 301 (stałe przekierowanie), lub 302 (tymczasowe przekierowanie). Do naszych celów byłoby przydatne przechwycenie tych kodów statusów i po prostu zwrócenie ich bez rzucania żadnych wyjątków. Aby to zrobić, musimy zdefiniować własną klasę obsługi URL-i (ang. URL handler).

Poniższa klasa obsługi URL-i jest częścią modułu openanything.py.

Przykład 11.7 Definiowanie klas obsługi URL-i

```
class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):    #(1)
    def http_error_default(self, req, fp, code, msg, headers): #(2)
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code                                   #(3)
        return result
```

1. `urllib2` jest zaprojektowana jako zbiór klas obsługi URL-i. Każda z tych klas może definiować dowolną liczbę metod. Gdy coś się wydarzy – jak np. błąd HTTP lub nawet kod 304 – `urllib2` używa introspekcji do odnalezienia w liście zdefiniowanych klas obsługi URL-i metody, która może obsłużyć to zdarzenie. Używaliśmy podobnej introspekcji w rozdziale 9-tym, Przetwarzanie XML-a do zdefiniowania metod obsługi dla różnych typów węzłów, ale `urllib2` jest bardziej elastyczny i przeszukuje tyle klas obsługi ile jest zdefiniowanych dla bieżącego żądania.

2. `urllib2` przeszukuje zdefiniowane klasy obsługi i wywołuje metodę `http_error_default`, gdy otrzyma kod statusu 304 od serwera. Definiując własną klasę obsługi błędów, możemy zapobiec rzucaniu wyjątków przez `urllib2`. Zamiast tego tworzymy obiekt `HTTPError`, ale zwracamy go, zamiast rzucania go jako wyjątek.

3. To jest kluczowa część: przed zwróceniem zachowujemy kod statusu zwrócony przez serwer HTTP. To pozwala na dostęp do niego programowi wywołującemu.

Przykład 11.8 Używanie własnych klas obsługi URL-i

```
>>> request.headers                               #(1)
{'If-modified-since': 'Thu, 15 Apr 2004 19:45:21 GMT'}
>>> import openanything
>>> opener = urllib2.build_opener(
...     openanything.DefaultErrorHandler())   #(2)
```

```
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status                    #(3)
304
>>> seconddatastream.read()                    #(4)
''
```

1. Kontynuujemy poprzedni przykład, a więc obiekt Request jest już utworzony i nagłówek If-Modified-Since został już dodany.

2. To jest klucz: ponieważ mamy zdefiniowaną własną klasę obsługi URL-i, musimy powiedzieć `urllib2`, aby teraz jej używał. Pamiętasz jak mówiłem, iż `urllib2` podzielił proces dostępu do zasobów HTTP na trzy etapy i to nie bez powodu? Oto dlaczego wywołanie funkcji `build_opener` jest odrębnym etapem. Ponieważ na jej wejściu możesz podać własne klasy obsługi URL-i, które powodują zmianę domyślnego działania `urllib2`.

3. Teraz możemy zasób otworzyć po cichu a z powrotem otrzymujemy obiekt, który wraz z nagłówkami (użyj `seconddatastream.headers.dict`, aby je pobrać), zawiera także kod statusu HTTP. W tym wypadku, tak jak oczekiwaliśmy, tym statusem jest 304, który oznacza że dane nie zmieniły się od ostatniego razu, gdy o nie prosiliśmy.

4. Zauważ, że gdy serwer odsyła kod statusu 304, to nie przesyła ponownie danych. I o to w tym wszystkim chodzi: aby oszczędzić przepustowość poprzez niepobieranie ponowne danych, które nie uległy zmianie. A więc jeśli potrzebujesz tych danych, to musisz zachować je w lokalnym buforze po pierwszym pobraniu.

Z nagłówka ETag korzystamy bardzo podobnie. Jednak zamiast sprawdzania nagłówka Last-Modified i przesyłania If-Modified-Since, sprawdzamy nagłówek ETag a przesyłamy If-None-Match. Zacznijmy całkiem nową sesję w naszym IDE.

Przykład 11.9 Użycie `ETag/If-None-Match`

```
>>> import urllib2, openanything
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener(
...     openanything.DefaultErrorHandler())
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.get('ETag')        #(1)
'"e842a-3e53-55d97640"'
>>> firstdata = firstdatastream.read()
>>> print firstdata                            #(2)
<?xml version="1.0" encoding="iso-8859-1"?>
 <feed version="0.3"
   xmlns="http://purl.org/atom/ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xml:lang="en">
   <title mode="escaped">dive into mark</title>
   <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
   [.. ciach ..]
>>> request.add_header('If-None-Match',
```

```
...     firstdatastream.headers.get('ETag'))   #(3)
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status                    #(4)
304
>>> seconddatastream.read()                    #(5)
''
```

1. Używając pseudo-słownika `firstdatastream.headers` możemy pobrać nagłówek ETag zwrócony przez serwer. (Co się stanie, jeśli serwer nie zwróci nagłówka ETag? Wtedy ta linia powinna zwrócić `None`.)
2. OK, mamy dane.
3. Teraz przy drugim wywołaniu ustawiamy w nagłówku If-None-Match wartość sumy kontrolnej z ETag otrzymanego przy pierwszym wywołaniu.
4. Drugie wywołanie działa prawidłowo bez żadnych zakłóceń (bez rzucania żadnych wyjątków) i ponownie widzimy, że serwer odesłał status kodu 304. Bazując na sumie kontrolnej nagłówka ETag, którą wysłaliśmy za drugim razem, wie on że dane nie zmieniły się.
5. Niezależnie od tego, czy kod 304 jest rezultatem sprawdzania daty Last-Modified czy sumy kontrolnej ETag, nigdy nie otrzymamy z powrotem ponownie tych samych danych, a jedynie kod statusu 304. I o to chodziło.

W tych przykładach serwer HTTP obsługiwał zarówno nagłówek Last-Modified jak i ETag, ale nie wszystkie serwery to potrafią. Jako użytkownik serwisów internetowych powinieneś być przygotowany do używania ich obu, ale musisz programować defensywnie na wypadek gdyby serwer obsługiwał tylko jeden z tych nagłówków lub żaden.

12.7 Obsługa przekierowań

Obsługa przekierowań

Możemy obsługiwać trwałe i tymczasowe przekierowania używając różnego rodzaju własnych klas obsługi URL-i.

Po pierwsze zobaczmy dlaczego obsługa przekierowań jest konieczna.

Przykład 11.10 Dostęp do usługi internetowej bez obsługi przekierowań

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1          #(1)
>>> request = urllib2.Request(
...     'http://diveintomark.org/redir/example301.xml') #(2)
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redir/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'             #(3)
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml  #(4)
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0                              #(5)
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.url                                               #(6)
'http://diveintomark.org/xml/atom.xml'
>>> f.headers.dict
{'content-length': '15955',
'accept-ranges': 'bytes',
'server': 'Apache/2.0.49 (Debian GNU/Linux)',
```

```
'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
connection': 'close',
'etag': '"e842a-3e53-55d97640"',
'date': 'Thu, 15 Apr 2004 22:06:25 GMT',
'content-type': 'application/atom+xml'}
>>> f.status
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: addinfourl instance has no attribute 'status'
```

1. Lepiej będziesz mógł zobaczyć co się dzieje, gdy włączysz tryb debugowania.
2. To jest URL, który ma ustawione trwałe przekierowanie do RSS-a pod adresem http://diveintomark.org/xml/atom.xml.
3. Gdy próbujemy pobrać dane z tego adresu, to serwer odsyła kod statusu 301 informujący o tym, że ten zasób został przeniesiony na stałe.
4. Serwer przesyła także nagłówek `Location:`, który zawiera nowy adres tych danych.
5. `urllib2` zauważa ten kod statusu dotyczący przekierowania i automatycznie próbuje pobrać dane spod nowej lokalizacji podanej w nagłówku `Location:`.
6. Obiekt zwrócony przez `opener` zawiera już nowy adres (po przekierowaniu) i wszystkie nagłówki zwrócone po drugim żądaniu (zwrócone z nowego adresu). Jednak brakuje kodu statusu, a więc nie mamy możliwości programowego stwierdzenia, czy to przekierowanie było trwałe, czy tylko tymczasowe. A to ma wielkie znaczenie: jeśli to było przekierowanie tymczasowe, wtedy musimy ponowne żądania kierować pod stary adres, ale jeśli to było trwałe przekierowanie (jak w tym przypadku), to nowe żądania od tego momentu powinny być kierowane do nowej lokalizacji.

To nie jest optymalne, ale na szczęście łatwe do naprawienia. `urllib2` nie zachowuje się dokładnie tak, jak tego chcemy, gdy napotyka na kody 301 i 302, a więc zmieńmy to zachowanie. Jak? Przy pomocy własnej klasy obsługi URL-i, tak jak to zrobiliśmy w przypadku kodu 304.

Poniższa klasa jest zdefiniowana w openanything.py.

Przykład 11.11 Definiowanie klasy obsługi przekierowań

```
class SmartRedirectHandler(urllib2.HTTPRedirectHandler):     #(1)
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301( #(2)
            self, req, fp, code, msg, headers)
        result.status = code                                 #(3)
        return result

    def http_error_302(self, req, fp, code, msg, headers):   #(4)
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
```

```
        return result
```

1. Obsługa przekierowań w `urllib2` jest zdefiniowana w klasie o nazwie `HTTPRedirectHandler`. Nie chcemy całkowicie zmieniać działania tej klasy, chcemy je tylko lekko rozszerzyć, a więc dziedziczymy po niej, a wtedy będziemy mogli wywoływać metody klasy nadrzędnej do wykonania całej ciężkiej roboty.

2. Gdy napotkany zostanie status kodu 301 przesłany przez serwer, `urllib2` przeszuka listę klas obsługi i wywoła metodę `http_error_301`. Pierwszą rzeczą, jaką wykona nasza wersja, jest po prostu wywołanie metody `http_error_301` przodka, która wykona całą robotę związaną ze znalezieniem nagłówka `Location:` i przekierowaniem żądania pod nowy adres.

3. Tu jest kluczowa sprawa: zanim wykonamy `return`, zachowujemy kod statusu (301), aby program wywołujący mógł go później odczytać.

4. Przekierowania tymczasowe (kod statusu 302) działają w ten sam sposób: nadpisujemy metodę `http_error_302`, wywołujemy metodę przodka i zachowujemy kod statusu przed powrotem z metody.

A więc jaką mamy z tego korzyść? Możemy teraz utworzyć klasę pozwalającą na dostęp do zasobów internetowych wraz z naszą własną klasą obsługi przekierowań i będzie ona nadal dokonywała przekierowań automatycznie, ale tym razem będzie ona także udostępniała kod statusu przekierowania.

Przykład 11.12 Użycie klasy obsługi przekierowań do wykrycia przekierowań trwałych

```
>>> request = urllib2.Request('http://diveintomark.org/redir/example301.xml')
>>> import openanything, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> opener = urllib2.build_opener(
...     openanything.SmartRedirectHandler())           #(1)
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: 'GET /redir/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'           #(2)
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
```

```
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.status                                       #(3)
301
>>> f.url
'http://diveintomark.org/xml/atom.xml'
```

1. Po pierwsze tworzymy `opener` z przed chwilą zdefiniowaną klasą obsługi przekierowań.
2. Wysłaliśmy żądanie i otrzymaliśmy w odpowiedzi kod statusu 301. W tym momencie wołana jest metoda `http_error_301`. Wywołujemy metodę przodka, która odnajduje przekierowanie i wysyła żądanie pod nową lokalizację (http://diveintomark.org/xml/atom.xml).
3. Tu jest nasza korzyść: teraz nie tylko mamy dostęp do nowego URL-a, ale także do kodu statusu przekierowania, a więc możemy stwierdzić, że było to przekierowanie trwałe. Przy następnym żądaniu tych danych, powinniśmy użyć nowego adresu (http://diveintomark.org/xml/atom.xml, jak widać w `f.url`). Jeśli mamy zachowaną daną lokalizację w pliku konfiguracyjnym lub w bazie danych, to powinniśmy zaktualizować ją, aby nie odwoływać się ponownie do starego adresu. To jest pora do aktualizacji książki adresowej.

Ta sama klasa obsługi przekierowań może także pokazać, że nie powinniśmy aktualizować naszej książki adresowej.

Przykład 11.13 Użycie klasy obsługi przekierowań do wykrycia przekierowań tymczasowych

```
>>> request = urllib2.Request(
...     'http://diveintomark.org/redir/example302.xml')   #(1)
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redir/example302.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 302 Found\r\n'                          #(2)
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
```

```
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 314
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0                                #(3)
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.status                                              #(4)
302
>>> f.url
http://diveintomark.org/xml/atom.xml
```

1. To jest przykładowy URL skonfigurowany tak, aby powiadamiać klientów o tymczasowym przekierowaniu do http://diveintomark.org/xml/atom.xml.
2. Serwer odsyła z powrotem kod statusu 302 wskazujący na tymczasowe przekierowanie. Tymczasowa lokalizacja danych jest podana w nagłówku `Location:`.
3. `urllib2` wywołuje naszą metodę `http_error_302`, która wywołuje metodę przodka o tej samej nazwie w `urllib2.HTTPRedirectHandler`, która wykonuje przekierowanie do nowej lokalizacji. Wtedy nasza metoda `http_error_302` zachowuje kod statusu (302), a więc wywołująca aplikacja może go później odczytać.
4. I oto mamy prawidłowo wykonane przekierowanie do http://diveintomark.org/xml/atom.xml. `f.status` informuje, iż było to przekierowanie tymczasowe, co oznacza, że ponowne żądania powinniśmy kierować pod stary adres (http://diveintomark.org/redir/example302.xml). Może następnym razem znowu nastąpi przekierowanie, a może nie. Może nastąpi przekierowanie pod całkiem inny adres. Nie do nas należy ta decyzja. Serwer powiedział, że to przekierowanie było tylko tymczasowe, a więc powinniśmy to uszanować. Teraz dostarczamy wystarczającą ilość informacji, aby aplikacja wywołująca była w stanie to uszanować.

12.8 Obsługa skompresowanych danych

Obsługa skompresowanych danych

Ostatnią ważną właściwością HTTP, którą będziemy chcieli obsłużyć, będzie kompresja. Wiele serwisów sieciowych posiada zdolność wysyłania skompresowanych danych, dzięki czemu wielkość wysyłanych danych może zmaleć nawet o 60% lub więcej. Sprawdza się to w szczególności w XML-owych serwisach sieciowych, ponieważ dane XML kompresują się bardzo dobrze.

Serwery nie dadzą nam skompresowanych danych, jeśli im nie powiemy, że potrafimy je obsłużyć.

Przykład 11.14. Informowanie serwera, że chcielibyśmy otrzymać skompresowane dane

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> request.add_header('Accept-encoding', 'gzip')        #(1)
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
Accept-encoding: gzip                                    #(2)
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:24:39 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Vary: Accept-Encoding
header: Content-Encoding: gzip                           #(3)
header: Content-Length: 6289                             #(4)
header: Connection: close
header: Content-Type: application/atom+xml
```

1. To jest najważniejsza część: kiedy utworzymy obiekt `Request`, dodajemy nagłówek `Accept-encoding`, aby powiedzieć serwerowi, że akceptujemy dane zakodowane jako `gzip`. `gzip` jest nazwą wykorzystanego algorytmu kompresji. Teoretycznie powinny być dostępne inne algorytmy kompresji, ale `gzip` jest algorytmem kompresji wykorzystywanym przez 99% serwisów sieciowych.

2. W tym miejscu nagłówek idzie przez linie.

3. I w tym miejscu otrzymujemy informacje o tym, co serwer przesyła nam z powrotem: nagłówek `Content-Encoding: gzip` oznacza, że dane które otrzymaliśmy zostały skompresowane jako `gzip`.

4. Nagłówek `Content-Length` oznacza długość danych skompresowanych, a nie zdekompresowanych. Jak zobaczymy za minutkę, rzeczywista wielkość zdekompresowanych danych wynosi 15955, zatem dane zostały skompresowane o ponad 60%.

Przykład 11.15. Dekompresowanie danych

```
>>> compresseddata = f.read()                              #(1)
>>> len(compresseddata)
6289
>>> import StringIO
>>> compressedstream = StringIO.StringIO(compresseddata)   #(2)
>>> import gzip
>>> gzipper = gzip.GzipFile(fileobj=compressedstream)      #(3)
>>> data = gzipper.read()                                  #(4)
>>> print data                                             #(5)
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
>>> len(data)
15955
```

1. Kontynuując z poprzedniego przykładu, `f` jest obiektem *plikopodobnym* zwróconym przez otwieracz URL-a. Korzystając z jego metody `read()` zazwyczaj dostaniemy nieskompresowane dane, ale ponieważ te dane będą skompresowane gzipem, to jest dopiero pierwszy krok, aby otrzymać dane, które naprawdę chcemy.

2. OK, ten krok jest troszeczkę okrężny. Python posiada moduł `gzip`, który czyta (i właściwie także zapisuje) pliki skompresowane jako gzip. Jednak my nie mamy pliku na dysku, mamy skompresowany bufor w pamięci, a nie chcemy tworzyć tymczasowego pliku, aby te dane dekompresować. Zatem tworzymy obiekt *plikopodobny* przechowujący w pamięci skompresowane dane (`compresseddata`) korzystając z modułu `StringIO`. Pierwszy raz wspomnieliśmy o `StringIO` w poprzednim rozdziale, ale teraz znaleźliśmy kolejny sposób, aby go wykorzystać.

3. Teraz tworzymy instancję klasy `GzipFile`. Ten "plik" jest obiektem *plikopodobnym* `compressedstream`.

4. To jest linia, która wykonuje całą właściwą pracę: "czyta" z `GzipFile` zdekompresowane dane. Ten "plik" nie jest prawdziwym plikiem na dysku. `gzipper` w rzeczywistości "czyta" z obiektu *plikopodobnego*, który stworzyliśmy za pomocą `StringIO`, aby opakować skompresowane dane znajdujące się tylko w pamięci w zmiennej `compresseddata` w obiekt *plikopodobny*. Ale skąd przyszły te skompresowane dane? Oryginalnie pobraliśmy je z odległego serwera HTTP dzięki "odczytaniu" obiektu *plikopodobnego*, który utworzyliśmy za pomocą `urllib2.build_opener`. I fantastycznie, to wszystko po prostu działa. Żaden element w tym łańcuchu nie ma pojęcia, że jego poprzednik tylko udaje, że jest tym za co się podaje.

5. Zobaczmy, są to prawdziwe dane. (tak naprawdę 15955 bajtów)

"Lecz czekaj!" — usłyszałem Twój krzyk, "Można to nawet zrobić prościej". Myślisz, że skoro `opener.open` zwraca obiekt *plikopodobny*, więc dlaczego nie wyciąć pośrednika `StringIO` i po prostu przekazać `f` bezpośrednio do `GzipFile`? OK, może tak nie myślałeś, ale tak czy inaczej nie przejmuj się tym, ponieważ to nie działa.

Przykład 11.16. Dekompresowanie danych bezpośrednio z serwera

```
>>> f = opener.open(request)                  #(1)
>>> f.headers.get('Content-Encoding')         #(2)
'gzip'
>>> data = gzip.GzipFile(fileobj=f).read()    #(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\gzip.py", line 217, in read
    self._read(readsize)
  File "c:\python23\lib\gzip.py", line 252, in _read
    pos = self.fileobj.tell()   # Save current position
AttributeError: addinfourl instance has no attribute 'tell'
```

1. Kontynuując z poprzedniego przykładu, mamy już obiekt żądania `Request` z ustawionym nagłówkiem `Accept-encoding: gzip header`.

2. Zaraz po otworzeniu żądania otrzymujemy nagłówki z serwera (ale nie pobieramy jeszcze żadnych danych). Jak możemy zobaczyć ze zwróconego nagłówka `Content-Encoding`, dane te zostały skompresowane na `gzip`.

3. Ponieważ `opener.open` zwraca obiekt *plikopodobny*, a z dopiero co odczytanego nagłówka wynika, że otrzymane dane będą skompresowane na `gzip`-a, to dlaczego nie przekazać prosto otrzymany obiekt plikopodobny bezpośrednio do `GzipFile`? Kiedy "czytamy" z instancji `GzipFile`-a, będziemy "czytali" skompresowane dane z serwera HTTP i dekompresowali je w locie. Jest to dobry pomysł, ale niestety nie działa. `GzipFile` potrzebuje zapisywać swoją pozycję i przesuwać się bliżej lub dalej po skompresowanym pliku, ponieważ w taki sposób działa kompresja `gzip`. Nie działa to, kiedy "plikiem" jest strumień bajtów przychodzących z zewnętrznego serwera; jedynie co możemy z tym zrobić, to jednorazowo pobierać dane bajty nie przesuwając się wstecz lub do przodu w strumieniu danych. Zatem nieelegancki sposób z wykorzystaniem `StringIO` jest najlepszym rozwiązaniem: pobieramy skompresowane dane, tworzymy z nich obiekt *plikopodobny* za pomocą `StringIO`, a następnie dekompresujemy dane wewnątrz niego.

12.9 HTTP - wszystko razem

Python/HTTP — wszystko razem

Widzieliśmy już wszystkie elementy potrzebne do utworzenia inteligentnego klienta usługi internetowej. Teraz zobaczmy jak to wszystko do siebie pasuje.

Przykład 11.17. Funkcja `openanything`

Ta funkcja jest zdefiniowana w pliku `openanything.py`.

```
def openAnything(source, etag=None, lastmodified=None, agent=USER_AGENT):
    # non-HTTP code omitted for brevity
    if urlparse.urlparse(source)[0] == 'http':                                   #(1)
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)                                  #(2)
        if etag:
            request.add_header('If-None-Match', etag)                            #(3)
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)                #(4)
        request.add_header('Accept-encoding', 'gzip')                            #(5)
        opener = urllib2.build_opener(SmartRedirectHandler(), DefaultErrorHandler()) #(6)
        return opener.open(request)                                              #(7)
```

1. `urlparse` to bardzo poręczny moduł do, na pewno zgadłeś, parsowania URL-i. Jego podstawowa funkcja, także nazywająca się `urlparse`, przyjmuje na wejściu URL-a i dzieli go na taką krotkę (schemat, domena, ścieżka, parametry, parametry w żądaniu i identyfikator fragmentu). Jednyną wsród tych rzeczy jaką musimy się przejmować jest schemat, który decyduje o tym czy mamy do czynienie z URL-em HTTP (który to moduł `urllib2` może obsłużyć).

2. Przedstawiamy się serwerowi HTTP przy pomocy nagłówka `User-Agent` przesłanego przez funkcję wywołującą. Jeśli nie zostałaby podana wartość `User-Agent`, użylibysmy wcześniej zdefiniowanej wartośći w `openanything.py`. Nigdy nie należy używać domyślnej wartości zdefiniowanej w `urllib2`.

3. Jeśli została podana suma kontrolna dla `ETag`, wysyłamy ją w nagłówku `If-None-Match`.

4. Jeśli została podana data ostatniej modyfikacji, wysyłamy ją w nagłówku `If-Modified-Since`.

5. Powiadamiamy serwer, że chcemy dane skompresowane, jeśli to jest tylko możliwe.

6. Wywołujemy funkcję `build_opener`, która wykorzystuje nasze własne klasy obsługi URL-i: `SmartRedirectHandler` do obsługi przekierowań 301 i 302 i `DefaultErrorHandler` do taktownej obsługi 304, 404, i innych błędnych sytuacji.

7. I to wszystko! Otwieramy URL-a i zwracamy pliko-podobny (ang. file-like) obiekt do funkcji wywołującej.

Przykład 11.18. Funkcja `fetch`

Ta funkcja jest zdefiniowana w pliku `openanything.py`.

```
def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)            #(1)
    result['data'] = f.read()                                        #(2)
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')                       #(3)
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')      #(4)
        if f.headers.get('content-encoding', ) == 'gzip':          #(5)
            # data came back gzip-compressed, decompress it
            result['data'] = gzip.GzipFile(fileobj=StringIO(result['data']])).read()
    if hasattr(f, 'url'):                                            #(6)
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):                                         #(7)
        result['status'] = f.status
    f.close()
    return result
```

1. Po pierwsze wywołujemy funkcję `openAnything` z URL-em, sumą kontrolną ETag, datą ostatniej modyfikacji (ang. Last-Modified date) i wartością `User-Agent`.
2. Czytamy aktualne dane zwrócone przez serwer. Mogą one być spakowane; jeśli tak, to później je rozpakowujemy.
3. Zachowujemy sumę kontrolną `ETag` wróconą przez serwer, także aplikacja wywołująca może ją przesłać ponownie następnym razem i możemy ją przekazać dalej do `openAnything`, która może ją wetknąć do nagłówka `If-None-Match` i przesłać do zdalnego serwera.
4. Zachowujemy także datę ostatniej modyfikacji.
5. Jeśli serwer powiedział, że wysłał spakowane dane, rozpakowujemy je.
6. Jeśli dostaliśmy URL-a z powrotem od serwera, zachowujemy go i zakładamy, że kod statusu wynosi 200, dopóki nie przekonamy się, że jest inaczej.
7. Jeśli któraś z naszych klas obsługi URL-i przechwyci jakiś kod statusu, zachowujemy go także.

Przykład 11.19. Użycie `openanything.py`

```
>>> import openanything
>>> useragent = 'MyHTTPWebServicesApp/1.0'
>>> url = 'http://diveintopython.org/redir/example301.xml'
>>> params = openanything.fetch(url, agent=useragent)              #(1)
>>> params                                                          #(2)
{'url': 'http://diveintomark.org/xml/atom.xml',
'lastmodified': 'Thu, 15 Apr 2004 19:45:21 GMT',
'etag': '"e842a-3e53-55d97640"',
```

```
'status': 301,
'data': '<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
<-- rest of data omitted for brevity -->'}
>>> if params['status'] == 301:                                    #(3)
...     url = params['url']
>>> newparams = openanything.fetch(
...     url, params['etag'], params['lastmodified'], useragent)    #(4)
>>> newparams
{'url': 'http://diveintomark.org/xml/atom.xml',
'lastmodified': None,
'etag': '"e842a-3e53-55d97640"',
'status': 304,
'data': }                                                         #(5)
```

1. Za pierwszym razem, gdy pobieramy jakiś zasób, nie mamy żadnej sumy kontrolnej ETag ani daty ostatniej modyfikacji, a więc opuszczamy te parametry. (To są parametry opcjonalne.)

2. Z powrotem otrzymujemy słownik kilku użytecznych nagłówków, kod statusu HTTP i aktualne dane zwrócone przez serwer. Funkcja `openanything` zajmuje się samodzielnie rozpakowaniem archiwum `gzip`; nie zajmujemy się tym na tym poziomie.

3. Jeśli kiedykolwiek otrzymamy kod statusu 301, czyli trwałe przekierowanie, to musimy zaktualizować naszego URL-a na nowy adresu.

4. Gdy po raz drugi pobieramy ten sam zasób, to mamy wiele informacji, które możemy przekazać: (być może zaktualizowany) URL, `ETag` z ostatniego razu, data ostatniej modyfikacji i oczywiście nasz `User-Agent`.

5. Z powrotem ponownie otrzymujemy słownik, ale dane nie uległy zmianie, a więc wszystko co dostaliśmy to był kod statusu 304 i żadnych danych.

12.10 HTTP - podsumowanie

Podsumowanie

Teraz moduł openanything.py i jego funkcje powinny mieć dla Ciebie sens.

Możemy wyróżnić 5 ważnych cech usług internetowych na bazie HTTP, które każdy klient powinien uwzględniać:

- Identyfikacja aplikacji poprzez właściwe ustawienie nagłówka User-Agent.
- Właściwa obsługa trwałych przekierowań.
- Uwzględnienie sprawdzania daty ostatniej modyfikacji (ang. Last-Modified), aby uniknąć ponownego pobierania danych, które nie uległy zmianie.
- Uwzględnienie sprawdzania sum kontrolnych z nagłówka ETag, aby uniknąć ponownego pobierania danych, które nie uległy zmianie.
- Uwzględnienie kompresji gzip, aby zredukować wielkość przesyłanych danych, nawet gdy dane uległy zmianie.

Rozdział 13

SOAP

13.1 SOAP

SOAP

Rozdział 11 przybliżył temat serwisów sieciowych HTTP zorientowanych na dokumenty. "Wejściowym parametrem" był URL, a "zwracaną wartością" był konkretny dokument XML, który można było sparsować.

Ten rozdział przybliży serwis sieciowy SOAP, który jest bardziej strukturalnym podejściem do problemu. Zamiast zajmować się bezpośrednio żądaniami HTTP i dokumentami XML, SOAP pozwala nam symulować wywoływanie funkcji, które zwracają natywne typy danych. Jak zobaczymy, złudzenie to jest niemal perfekcyjne: "wywołujemy" funkcję za pomocą biblioteki SOAP korzystając ze standardowej, wywołującej składni Pythona a funkcja zdaje się zwracać obiekty i wartości Pythona. Jednak pod tą przykrywką, biblioteka SOAP w rzeczywistości wykonuje złożoną transakcję wymagającą wielu dokumentów XML i zdalnego serwera.

SOAP jest złożoną specyfikacją i powiedzenie, że SOAP służy wyłącznie do zdalnego wywoływania funkcji będzie trochę wprowadzało w błąd. Niektórzy mogliby stwierdzić, że SOAP pozwala na jednostronne, asynchroniczne przekazywanie komunikatów i zorientowane na dokumenty serwisy sieciowe. I Ci ludzie także mieliby rację; SOAP może być wykorzystywany w ten sposób, a także na wiele innych. Jednak ten rozdział przybliży tak zwany "styl RPC" (Remote Procedure Call), czyli wywoływanie zewnętrznych funkcji i otrzymywanie z nich wyników.

Nurkujemy

Korzystasz z Google, prawda? Jest to popularna wyszukiwarka. Chciałeś kiedyś mieć programowy dostęp do wyników wyszukiwania za pomocą Google? Teraz możesz. Poniżej mamy program, który poszukuje w Google za pomocą Pythona.

Przykład 12.1. `search.py`

```
from SOAPpy import WSDL

# you'll need to configure these two values;
# see http://www.google.com/apis/
WSDLFILE = '/path/to/copy/of/GoogleSearch.wsdl'
APIKEY = 'YOUR_GOOGLE_API_KEY'

_server = WSDL.Proxy(WSDLFILE)
def search(q):
    """Search Google and return list of {title, link, description}"""
    results = _server.doGoogleSearch(
        APIKEY, q, 0, 10, False, "", False, "", "utf-8", "utf-8")
    return [{"title": r.title.encode("utf-8"),
             "link": r.URL.encode("utf-8"),
             "description": r.snippet.encode("utf-8")}
            for r in results.resultElements]

if __name__ == '__main__':
    import sys
    for r in search(sys.argv[1])[:5]:
        print r['title']
```

```
        print r['link']
        print r['description']
        print
```

Możesz importować to jako moduł i wykorzystywać to w większych programach, a także możesz uruchomić ten skrypt z linii poleceń. W linii poleceń przekazujemy zapytanie szukania jako argument linii poleceń, a program wypisuje nam URL, tytuł i opis z pięciu pierwszych wyników wyszukiwania.

Tutaj mamy przykładowe wyjście, gdy wyszkujemy słowo "python".

Przykład 12.2. Przykładowe użycie `search.py`

```
C:\diveintopython\common\py> python search.py "python"
 <b>Python</b> Programming Language
 http://www.python.org/
 Home page for <b>Python</b>, an interpreted, interactive, object-oriented,
 extensible<br> programming language. <b>...</b> <b>Python</b>
 is OSI Certified Open Source: OSI Certified.

 <b>Python</b> Documentation Index
 http://www.python.org/doc/
  <b>...</b> New-style classes (aka descrintro). Regular expressions. Database
 API. Email Us.<br> docs@<b>python</b>.org. (c) 2004. <b>Python</b>
 Software Foundation. <b>Python</b> Documentation. <b>...</b>

 Download <b>Python</b> Software
 http://www.python.org/download/
 Download Standard <b>Python</b> Software. <b>Python</b> 2.3.3 is the
 current production<br> version of <b>Python</b>. <b>...</b>
 <b>Python</b> is OSI Certified Open Source:

 Pythonline
 http://www.pythonline.com/


 Dive Into <b>Python</b>
 http://diveintopython.org/
 Dive Into <b>Python</b>. <b>Python</b> from novice to pro. Find:
 <b>...</b> It is also available in multiple<br> languages. Read
 Dive Into <b>Python</b>. This book is still being written. <b>...</b>
```

13.2 Instalowanie odpowiednich bibliotek

Instalowanie odpowiednich bibliotek

W odróżnieniu od pozostałego kodu w tej książce, ten rozdział wymaga bibliotek, które nie są instalowane wraz z Pythonem. Zanim zanurkujemy w usługi SOAP, musisz doinstalować trzy biblioteki: `PyXML`, `fpconst` i `SOAPpy`.

12.2.1. Instalacja `PyXML`

Pierwszą biblioteką jakiej potrzebujemy jest `PyXML`, zbiór bibliotek do obsługi XML, które dostarczają większą funkcjonalność niż wbudowane biblioteki XML, które omawialiśmy w rozdziale 9.

Procedura 12.1.

Oto sposób instalacji `PyXML`:

1. Wejdź na http://pyxml.sourceforge.net/, kliknij Downloads i pobierz ostatnią wersję dla Twojego systemu operacyjnego.
2. Jeśli uzywasz Windowsa, to masz kilka możliwości. Upewnij się, że pobierasz wersję `PyXML`, która odpowiada wersji Pythona, którego używasz.
3. Kliknij dwukrotnie na pliku instalatora. Jeśli pobrałeś PyXML 0.8.3 dla Windowsa i Pythona 2.3, to programem instalatora będzie plik PyXML-0.8.3.win32-py2.3.exe.
4. Wykonaj wszystkie kroki instalatora.
5. Po zakończeniu instalacji zamknij instalator. Nie będzie żadnych widocznych skutków tego, iż instalacja zakończyła się powodzeniem (żadnych programów zaistalowanych w menu Start lub nowych skrótów na pulpicie). `PyXML` jest po prostu zbiorem bibliotek XML używanych przez inne programy.

Aby zweryfikować czy `PyXML` zainstalował się poprawnie, uruchom IDE Pythona i sprawdź wersję zainstalowanych bibliotek XML, tak jak w tym przykładzie.

Przykład 12.3. Weryfikacja instalacji `PyXML`

```
>>> import xml
>>> xml.__version__
'0.8.3'
```

Ta wersja powinna odpowiadać numerowi wersji instalatora `PyXML`, który pobrałeś i uruchomiłeś.

12.2.2. Instalacja `fpconst`

Drugą biblioteką jaką potrzebujemy jest `fpconst`, zbiór stałych i funkcji do obsługi wartości zmienno-przecinkowych IEEE754. Dostarcza ona wartości specjalne To-Nie-Liczba (ang. Not-a-Number) (NaN), Dodatnia Nieskończoność (ang. Positive Infinity) (Inf) i Ujemna Nieskończoność (ang. Negative Infinity) (-Inf), które są częścią specyfikacji typów danych SOAP.

Procedura 12.2.

A oto procedura instalacji `fpconst`:

1. Pobierz ostatnią wersję `fpconst` z `http://www.analytics.washington.edu/statcomp/projects/rzope/fpconst/` lub `http://www.python.org/pypi/fpconst/`.

2. Są tam dwa pliki do pobrania, jeden w formacie `.tar.gz`, a drugi w formacie `.zip`. Jeśli używasz Windowsa, pobierz ten w formacie `.zip`; w przeciwnym razie ten w formacie `.tar.gz`.
3. Rozpakuj pobrany plik. W Windows XP możesz kliknąć prawym przyciskiem na pliku i wybrać pozycję `Extract All`; na wcześniejszych wersjach Windowsa będzie potrzebny dodatkowy program, np. WinZip. Na Mac OS X możesz kliknąć dwukrotnie na spakowanym pliku, aby go rozpakować przy pomocy `Stuffit Expander`.
4. Otwórz okno linii poleceń i przejdź do katalogu, w którym rozpakowałeś pliki `fpconst`.
5. Wpisz python setup.py install, aby uruchomić program instalujący.

Aby zweryfikować, czy `fpconst` zainstalował się prawidłowo, uruchom IDE Pythona i sprawdź numer wersji.

Przykład 12.4. Weryfikacja instalacji `fpconst`

```
>>> import fpconst
>>> fpconst.__version__
'0.6.0'
```

Ten numer wersji powinien odpowiadać wersji archiwum `fpconst`, które pobrałeś i zainstalowałeś.

12.2.3. Instalacja SOAPpy

Trzecim i ostatnim wymogiem jest sama biblioteka: `SOAPpy`.

Procedura 12.3.

A oto procedura instalacji `SOAPpy`:

1. Wejdź na http://pywebsvcs.sourceforge.net/ i wybierz Ostatnie Oficjalne Wydanie (ang. Latest Official Release) w sekcji `SOAPpy`.
2. Są tam dwa pliki do wyboru. Jeśli używasz Windowsa, pobierz plik `.zip`; w przeciwnym wypadku pobierz plik `.tar.gz`.
3. Rozpakuj pobrany plik, tak jak to zrobiłeś z `fpconst`.
4. Otwórz okno linii poleceń i przejdź do katalogu, w którym rozpakowałeś pliki `SOAPpy`.
5. Wpisz python setup.py install, aby uruchomić program instalujący.

Aby zweryfikować, czy SOAPpy zostało zainstalowane poprawnie, uruchom IDE Pythona i sprawdź numer wersji.

Przykład 12.5. Weryfikacja instalacji `SOAPpy`

```
>>> import SOAPpy
>>> SOAPpy.__version__
'0.11.4'
```

Ten numer wersji powinien odpowiadać wersji archiwum SOAPpy, które pobrałeś i zainstalowałeś.

13.3 Pierwsze kroki z SOAP

Pierwsze kroki z SOAP

Sercem SOAP jest zdolność wywoływania zdalnych funkcji. Jest wiele publicznie dostępnych serwerów SOAP, które udostępniają proste funkcje do celów demonstracyjnych.

Najbardziej popularnym publicznie dostępnym serwerem SOAP jest http://www.xmethods.net/. Poniższy przykład wykorzystuje funkcję demostracyjną, która pobiera kod pocztowy w USA i zwraca aktualną temperaturę w tym regionie.

Przykład 12.6. Pobranie aktualnej temperatury

```
>>> from SOAPpy import SOAPProxy            #(1)
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> namespace = 'urn:xmethods-Temperature'  #(2)
>>> server = SOAPProxy(url, namespace)      #(3)
>>> server.getTemp('27502')                 #(4)
80.0
```

1. Dostęp do zdalnego serwera SOAP możemy uzyskać poprzez klasę proxy `SOAPProxy`. Proxy wykonuje całą wewnętrzną robotę związaną z SOAP, włącznie z utworzeniem dokumentu XML żądania z nazwy funkcji i listy jej argumentów, wysłaniem żądania za pośrednictwem HTTP do zdalnego serwera SOAP, sparsowaniem dokumentu XML odpowiedzi i utworzeniem wbudowanych wartości Pythona, które zwraca. Zobaczymy jak wyglądają te dokumenty XML w następnej sekcji.

2. Każda usługa SOAP posiada URL, który obsługuje wszystkie żądania. Ten sam URL jest używany do wywoływania wszystkich funkcji. Ta konkretna usługa ma tylko jedną funkcję, ale później w tym rozdziale zobaczymy przykłady API Google'a, które ma kilka funkcji. URL usługi jest współdzielony przez wszystkie funkcje. Każda usługa SOAP ma także przestrzeń nazw, która jest definiowana przez serwer i jest zupełnie dowolna. Jest ona po prostu częścią konfiguracji wymaganą do wywoływania metod SOAP. Pozwala ona serwerowi na wykorzystywanie jednego URL-a dla usługi i odpowiednie przekierowywanie żądań pomiędzy kilkoma niepowiązanymi ze sobą usługami. To jest podobne do podziału modułów Pythona na pakiety.

3. Tworzymy instancję `SOAPProxy` podając URL usługi i przestrzeń nazw usługi. Ta operacja nie wykonuje jeszcze żadnego połączenia z serwerem SOAP; ona po prostu tworzy lokalny obiekt Pythona.

4. Teraz, gdy wszystko jest odpowiednio skonfigurowane, możemy właściwie wywołać zdalne metody SOAP tak jakby to były lokalne funkcje. Przekazujemy argumenty tak jak do normalnych funkcji i pobieramy wartości zwrotne też tak jak od normalnych funcji. Ale pod spodem tak naprawdę dzieje się niezwykle dużo.

A więc zajrzyjmy pod spód.

13.4 Debugowanie serwisu sieciowego SOAP

Debugowanie serwisu sieciowego SOAP

Biblioteki SOAP dostarczają łatwego sposobu na zobaczenie co się tak naprawdę dzieje za kulisami.

Włączenie debugowania to jest po prostu kwestia ustawienia dwóch flag w konfiguracji `SOAPProxy`.

Przykład 12.7. Debugowanie serwisów SOAP

```
>>> from SOAPpy import SOAPProxy
 >>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
 >>> n = 'urn:xmethods-Temperature'
 >>> server = SOAPProxy(url, namespace=n)     #(1)
 >>> server.config.dumpSOAPOut = 1            #(2)
 >>> server.config.dumpSOAPIn = 1
 >>> temperature = server.getTemp('27502')    #(3)
 *** Outgoing SOAP ******************************************************
 <?xml version="1.0" encoding="UTF-8"?>
 <SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
 <SOAP-ENV:Body>
 <ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
 <v1 xsi:type="xsd:string">27502</v1>
 </ns1:getTemp>
 </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
 ************************************************************************
 *** Incoming SOAP ******************************************************
 <?xml version='1.0' encoding='UTF-8'?>
 <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <SOAP-ENV:Body>
 <ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <return xsi:type="xsd:float">80.0</return>
 </ns1:getTempResponse>

 </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
 ************************************************************************

 >>> temperature
 80.0
```

1. Po pierwsze tworzymy normalnie `SOAPProxy` podając URL serwisu i przestrzeń nazw.

2. Po drugie włączamy debugowanie poprzez ustawienie `server.config.dumpSOAPIn` i `server.config.dumpSOAPOut`.

3. Po trzecie wywołujemy jak zwykle zdalną metodę SOAP. Biblioteka SOAP wyświetli zarówno wychodzący dokument XML żądania, jak i przychodzący dokument XML odpowiedzi. To jest cała ciężka praca jaką `SOAPProxy` wykonuje dla Ciebie. Przerażające, nie prawdaż? Rozbieżmy to na czynniki.

Większość dokumentu XML żądania, który jest wysyłany do serwera, to są elementy stałe. Zignoruj wszystkie te deklaracje przestrzeni nazw; one nie ulegają zmianie (lub są bardzo podobne) w trakcie wszystkich wywołań SOAP. Sercem "wywołania funkcji" jest ten fragment w elemencie <Body>:

```
<ns1:getTemp                                   #(1)
  xmlns:ns1="urn:xmethods-Temperature"         #(2)
  SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">27502</v1>           #(3)
</ns1:getTemp>
```

1. Nazwą elementu jest nazwa funkcji: `getTemp`. `SOAPProxy` używa `getattr` jako dyspozytora. Zamiast wywoływania poszczególnych metod lokalnych bazując na nazwie metody, używa on nazwy metody do skonstruowania dokumentu XML żądania.

2. Element XML-a dotyczący funkcji zawarty jest w konkretnej przestrzeni nazw, która to jest ta podana podczas tworzenia instancji klasy `SOAPProxy`. Nie przejmuj się tym `SOAP-ENC:root`; to też jest stały element.

3. Argumenty funkcji także zostały przekształcone na XML-a. `SOAPProxy` używając introspekcji analizuje każdy argument, aby okreslić jego typ (w tym wypadku jest to string). Typ argumentu trafia do atrybutu `xsi:type`, a zaraz za nim podana jest jego wartość.

Zwracany dokument XML jest równie prosty do zrozumienia, jeśli tylko wiesz co należy zignorować. Skup się na tym fragmencie wewnątrz elementu <Body>:

```
<ns1:getTempResponse                               #(1)
   xmlns:ns1="urn:xmethods-Temperature"             #(2)
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <return xsi:type="xsd:float">80.0</return>        #(3)
 </ns1:getTempResponse>
```

1. Serwer zawarł wartość zwracaną przez funkcję w elemencie <`getTempResponse`>. Zgodnie z konwencją ten element jest nazwą funkcji plus słowo Response. Ale tak naprawdę to może być prawie cokolwiek; ważną rzeczą jaką SOAPProxy rozpatruje nie jest nazwa elementu, ale przestrzeń nazw.

2. Serwer zwraca odpowiedź w tej samej przestrzeni nazw, której użyliśmy w żądaniu, tej samej przestrzeni nazw, którą podaliśmy, gdy po raz pierwszy tworzyliśmy obiekt klasy `SOAPProxy`. Dalej w tym rozdziale zobaczymy co się stanie, jeśli zapomnimy podać przestrzeń nazw podczas tworzenia obiektu `SOAPProxy`.

3. Zwracana wartość jest podana wraz z jej typem (czyli float). `SOAPProxy` korzysta z tego typu danych do utworzenia właściwego wbudowanego typu danych Pythona i zwraca go.

13.5 Wprowadzenie do WSDL

Wprowadzenie do WSDL

Klasa SOAPProxy przeźroczyście przekształca wywołania lokalnych metod na wywołania zdalnych metod SOAP. Jak mogliśmy zobaczyć, jest z tym dużo roboty, ale SOAPProxy wykonuje to szybko i niewidocznie. Jednak nie dostarcza on żadnych środków służących do introspekcji metod.

Rozważmy to: dwa poprzednie podrozdziały pokazały przykłady wywoływania prostych zdalnych metod SOAP z jednym argumentem i jedną zwracaną wartością, a obydwa były prostym typem danych. Wymagało to znajomości URL-a serwisu, przestrzeni nazw serwisu, nazwy funkcji, liczby argumentów i typu danych każdego argumentu. Jeśli coś z tego pominiemy lub popełnimy w czymś błąd, wszystko nawali.

Jednak nie powinno to być wielką niespodzianką. Jeśli chcemy wywołać lokalną funkcję, musimy znać pakiet lub moduł, w którym ona się znajduje (odpowiednik URL-a serwisu i przestrzeni nazw). Potrzebujemy także znać poprawną nazwę funkcji i poprawną liczbę argumentów. Python doskonale radzi sobie z typami danych bez wyraźnego ich określenia, jednak my nadal musimy wiedzieć, ile argumentów mamy przekazać i na ile zwróconych wartości będziemy oczekiwać.

Ogromna różnica tkwi w introspekcji. Jak zobaczyliśmy w Rozdziale 4, Python pozwala Tobie odkrywać moduły i funkcje podczas wykonywania programu. Możemy wypisać wszystkie funkcje dostępne w danym module, a także gdy trochę popracujemy dokopać się do deklaracji i argumentów pojedynczych funkcji.

WSDL pozwala robić to samo z serwisami internetowymi SOAP. W języku angielskim WSDL jest skrótem od "Web Services Description Language". Mimo że został elastycznie napisany, aby opisywać wiele rodzajów różnych serwisów sieciowych, jest często wykorzystywany do opisywania serwisów SOAP.

Plik WSDL jest właśnie... plikiem. A dokładniej, jest plikiem XML. Zazwyczaj znajduje się na tym samym serwerze, który wykorzystujemy do użycia serwisu SOAP. Później w tym rozdziale, pobierzemy plik opisujący API Google i wykorzystamy go lokalnie. Nie oznacza to, że będziemy wywoływać Google lokalnie, ponieważ plik WSDL nadal będzie opisywał zewnętrzne funkcje rezydujące gdzieś na serwerze Google.

Plik WSDL przechowuje opis wszystkiego, co jest związane z wywoływaniem serwisu SOAP, czyli:

- URL serwisu i przestrzeń nazw
- typ serwisu sieciowego (prawdopodobnie wywołania funkcji są wykonywane za pomocą SOAP, jednak, jak było powiedziane wcześniej, WSDL jest wystarczająco elastyczny, aby opisać całą gammę różnych serwisów)
- listę dostępnych funkcji
- argumenty każdej funkcji
- typy danych każdego argumentu
- zwracane wartości każdej funkcji i ich typy danych

Innymi słowy, plik WSDL mówi o wszystkim, co potrzebujemy wiedzieć, aby móc wywoływać serwisy SOAP.

13.6 Introspekcja SOAP z użyciem WSDL

Introspekcja SOAP z użyciem WSDL

Podobnie jak wiele rzeczy w obszarze serwisów sieciowych, WSDL posiada burzliwą historię, pełną politycznych sporów i intryg. Jednak przeskoczymy ten wątek historyczny, ponieważ może wydawać się nudny. Istnieje także trochę innych standardów, które pełnią podobną funkcję, jednak WDSL jest najbardziej popularny, zatem nauczmy się go używać.

Najbardziej fundamentalną rzeczą, na którą nam pozwala WSDL jest odkrywanie dostępnych metod oferowanych przez serwer SOAP.

Przykład 12.8. Odkrywanie dostępnych metod

```
>>> from SOAPpy import WSDL          #(1)
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl')
>>> server = WSDL.Proxy(wsdlFile)    #(2)
>>> server.methods.keys()            #(3)
[u'getTemp']
```

1. `SOAPpy` zawiera parser `WSDL`. W czasie pisania tego podrozdziału, parser ten określany był jako moduł *na wczesnym etapie rozwoju*, jednak nie było problemów podczas testowania z parsowanymi plikami WSDL.

2. Aby skorzystać z pliku *WSDL*, ponownie korzystamy z klasy *pośredniczącej* (ang. *proxy*), `WSDL.Proxy`, która przyjmuje pojedynczy argument: plik `WSDL`. Zauważmy, że w tym przypadku przekazaliśmy adres URL pliku WSDL, który jest przechowywany gdzieś na zdalnym serwerze, ale klasa *pośrednicząca* równie dobrze sobie z nim radzi jak z lokalną kopią pliku WSDL. Podczas tworzenia 'pośrednika" WSDL, plik WSDL zostanie pobrany i sparsowany, więc jeśli wystąpią jakieś błędy w pliku WSDL (lub gdy będziemy mieli problemy z siecią), będziemy o tym wiedzieć natychmiast.

3. Klasa *pośrednicząca* WSDL przechowuje dostępne funkcje w postaci pythonowego słownika, `server.methods`. Zatem, aby pobrać listę dostępnych metod, wystarczy wywołać metodę `keys` należącą do słownika.

OK, więc już wiemy, że ten serwer SOAP oferuje jedną metodę: `getTemp`. Jednak w jaki sposób ją wywołać? Obiekt *pośredniczący* WSDL może nam także o tym powiedzieć.

Przykład 12.9. Odkrywanie argumentów metody

```
>>> callInfo = server.methods['getTemp']  #(1)
>>> callInfo.inparams                     #(2)
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AD0>]
>>> callInfo.inparams[0].name             #(3)
u'zipcode'
>>> callInfo.inparams[0].type             #(4)
(u'http://www.w3.org/2001/XMLSchema', u'string')
```

1. Słownik `server.methods` jest wypełniony określoną przez `SOAPpy` strukturą nazwaną `CallInfo`. Obiekt `CallInfo` zawiera informacje na temat jednej określonej funkcji, włączając w to argumenty funkcji.

2. Argumenty funkcji są przechowywane w `callInfo.inparams`, która jest pythonową listą obiektów `ParameterInfo`, które z kolei zawierają informacje na temat każdego parametru.

3. Każdy obiekt `ParameterInfo` przechowuje atrybut `name`, który jest nazwą argumentu. Nie trzeba znać nazwy argumentu, aby wywołać funkcje poprzez SOAP, jednak SOAP obsługuje argumenty nazwane w wywołaniach funkcji (podobnie jak Python), a za pomocą WSDL.Proxy będziemy mogli poprawnie obsługiwać nazywane argumenty, które zostają przekazywane do zewnętrznej funkcji (oczywiście, jeśli to włączymy).

4. Ponadto każdy parametr ma wyraźnie określony typ, a korzysta tu z typów zdefiniowanych w XML Schema. Widzieliśmy to już wcześniej; przestrzeń nazw "XML Schema" była częścią "formularza umowy", jednak to zignorowaliśmy i nadal możemy to ignorować, ponieważ tutaj do niczego nie jest nam to potrzebne. Parametr `zipcode` jest łańcuchem znaków i jeśli przekażemy pythonowy łańcuch znaków do obiektu WSDL.Proxy, zostanie on poprawnie zmapowany i wysłany na serwer.

WSDL także nas informuje o zwracanych przez funkcję wartościach.

Przykład 12.10. Odkrywanie zwracanych wartości metody

```
>>> callInfo.outparams            #(1)
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AF8>]
>>> callInfo.outparams[0].name    #(2)
u'return'
>>> callInfo.outparams[0].type
(u'http://www.w3.org/2001/XMLSchema', u'float')
```

1. Uzupełnieniem do argumentów funkcji `callInfo.inparams` jest `callInfo.outparams`, który odnosi się do zwracanej wartości. Jest to także lista, ponieważ funkcje wywoływane poprzez SOAP mogą zwracać wiele wartości, podobnie zresztą jak funkcje Pythona.

2. Każdy obiekt `ParameterInfo` zawiera atrybuty `name` i `type`. Funkcja ta zwraca pojedynczą wartości nazwaną `return`, która jest liczbą zmiennoprzecinkową (czyli `float`)..

Teraz połączmy zdobytą wiedzę i wywołajmy serwis sieciowy SOAP poprzez pośrednika WSDL.

Przykład 12.11. Wywoływanie usługi sieciowej poprzez `WSDL.Proxy`

```
>>> from SOAPpy import WSDL
 >>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl')
 >>> server = WSDL.Proxy(wsdlFile)              #(1)
 >>> server.getTemp('90210')                    #(2)
 66.0
 >>> server.soapproxy.config.dumpSOAPOut = 1     #(3)
 >>> server.soapproxy.config.dumpSOAPIn = 1
 >>> temperature = server.getTemp('90210')
 *** Outgoing SOAP ******************************************************
 <?xml version="1.0" encoding="UTF-8"?>
```

```
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">90210</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
*** Incoming SOAP ******************************************************
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">66.0</return>
</ns1:getTempResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************

>>> temperature
66.0
```

1. Widzimy, że konfiguracja jest prostsza niż wywoływanie serwisu SOAP bezpośrednio, ponieważ plik WSDL zawiera informację zarówno o URL serwisu, jak i o przestrzeni nazw, którą potrzebujemy, aby wywołać serwis. Tworzony obiekt `WSDL.Proxy` pobiera plik WSDL, parsuje go i konfiguruje obiekt `SOAPProxy`, który będzie wykorzystywał do wywoływania konkretnego serwisu SOAP.

2. Po utworzeniu obiektu `WSDL.Proxy`, możemy wywoływać funkcje równie prosto jak za pomocą obiektu `SOAPProxy`. Nie jest to zaskakujące; `WSDL.Proxy` jest właśnie otoczką (ang. *wrapper*) dla `SOAPProxy` z kilkoma dodanymi metodami, a więc składnia wywoływania funkcji jest taka sama.

3. Możemy dostać się do obiektu `SOAPProxy` w `WSDL.Proxy` za pomocą `server.soapproxy`. Ta opcja jest przydatna, aby włączyć debugowanie, dlatego też kiedy wywołujemy funkcję poprzez pośrednika `WSDL`, jego `SOAPProxy` będzie pokazywał przychodzące i wychodzące przez łącze dokumenty XML.

13.7 Wyszukiwanie w Google

Wyszukiwanie w Google

Powróćmy wreszcie do przykładu zamieszczonego na początku rozdziału, który robi coś bardziej użytecznego i interesującego niż mierzenie obecnej temperatury.

Google dostarcza API SOAP dla korzystania z wyników wyszukiwania wewnątrz programów. By móc z niego korzystać musisz zarejestrować konto w Google Web Services.

Procedura 12.4. Zakładanie konta w Google Web Services

1. Wejdź na stronę http://www.google.com/apis/ i stwórz konto Google. Potrzebny jest Ci do tego tylko adres email. Po rejestracji pocztą elektroniczną dostaniesz swój klucz licencyjny Google API. Będziesz z niego korzystać przy każdym wywołaniu funkcji wyszukiwarki Google.

2. Również ze strony http://www.google.com/apis/ pobierz zestaw dewelopera Google Web API. Zawiera on przykładowy kod w kilku językach programowania (ale nie w Pythonie) i, co istotniejsze, plik WSDL.

3. Rozpakuj tenże zestaw i odnajdź w nim plik GoogleSearch.wsdl. Skopiuj go w bezpieczne miejsce na swoim dysku. Przyda się w dalszej części rozdziału.

Gdy będziesz już mieć klucz dewelopera i plik WSDL Google w jakimś pewnym miejscu możesz zacząć zabawę z Google Web Services.

Przykład 12.12. Wgląd w głąb Google Web Services

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl')              #(1)
>>> server.methods.keys()                                               #(2)
[u'doGoogleSearch', u'doGetCachedPage', u'doSpellingSuggestion']
>>> callInfo = server.methods['doGoogleSearch']
>>> for arg in callInfo.inparams:                                       #(3)
...     print arg.name.ljust(15), arg.type
key             (u'http://www.w3.org/2001/XMLSchema', u'string')
q               (u'http://www.w3.org/2001/XMLSchema', u'string')
start           (u'http://www.w3.org/2001/XMLSchema', u'int')
maxResults      (u'http://www.w3.org/2001/XMLSchema', u'int')
filter          (u'http://www.w3.org/2001/XMLSchema', u'boolean')
restrict        (u'http://www.w3.org/2001/XMLSchema', u'string')
safeSearch      (u'http://www.w3.org/2001/XMLSchema', u'boolean')
lr              (u'http://www.w3.org/2001/XMLSchema', u'string')
ie              (u'http://www.w3.org/2001/XMLSchema', u'string')
oe              (u'http://www.w3.org/2001/XMLSchema', u'string')
```

1. Rozpoczęcie korzystania z Google Web Services jest proste: utwórz obiekt `WSDL.Proxy` i wskaż mu miejsce, gdzie znajduje się Twoja lokalna kopia pliku WSDL Google.

2. Wedle zawartości pliku WSDL, Google udostępnia trzy funkcje: `doGoogleSearch`, `doGetCachedPage` i `doSpellingSuggestion`. Robią dokładnie to, co sugerują ich nazwy. Pierwsza z nich wykonuje wyszukiwanie i zwraca jego wyniki, druga daje dostęp do kopii strony na serwerach Google (z okresu, kiedy była ostatnio odwiedzona przez googlebota), a trzecia sugeruje poprawę błędów literowych we wpisywanych hasłach.

3. Funkcja `doGoogleSearch` ma kilka parametrów różnego typu. Zauważ, że o ile z zawartości pliku WSDL można wywnioskować rodzaj i typ argumentów, o tyle niemożliwe jest stwierdzenie jak je wykorzystać. Teoretycznie mogłyby być także określone przedziały, do których muszą należeć argumenty, jednak plik WSDL Google nie jest tak szczegółowy. `WSDL.Proxy` nie czyni cudów — może dostarczyć Ci tylko informacji zawartych w pliku WSDL.

Poniżej znajduje się zestawienie parametrów funkcji `doGoogleSearch`:

- `key` — Twój klucz licencyjny otrzymany po rejestracji konta Google Web Services.
- `q` — Słowo lub wyrażenie, którego szukasz. Składnia jest dokładnie taka sama jak formularza wyszukiwania na stronie www Google, więc zadziałają tutaj wszelkie znane Ci sztuczki lub zaawansowana składnia wyszukiwarki.
- `start` — Indeks wyniku wyszukiwania, od którego będą liczone zwrócone wyniki. Podobnie do wersji interaktywnej wyszukiwarki, funkcja ta zwraca 10 wyników na raz. Chcąc uzyskać drugą "stronę" wyników wyszukiwania podajemy tutaj 10.
- `maxResults` — Liczba wyników do zwrócenia. Ograniczona z góry do 10, aczkolwiek, gdy interesuje cię tylko kilka wyników, w celu oszczędzenia transferu można podać wartość mniejszą.
- `filter` — Podana wartość True spowoduje, iż Google odfiltruje duplikaty stron z wyników wyszukiwania.
- `restrict` — Ustawienie `countryXX`, gdzie XX to kod państwa spowoduje wyświetlenie wyników tylko dla danego państwa, np. `countryUK` spowoduje wyszukiwanie tylko dla Zjednoczonego Królestwa. Dopuszczalnymi wartościami są też `linux`, `mac` i `bsd`, które spowodują wyszukiwanie w zdefiniowanych przez Google zbiorach stron o tematyce technicznej, lub `unclesam`, które spowoduje wyszukiwanie w materiałach dotyczących rządu i administracji Stanów Zjednoczonych.
- `safeSearch` — Dla wartości True Google odfiltruje z wyników strony pornograficzne.
- `lr` (ang. "language restrict" — ograniczenie językowe) — Ustawienie konkretnego kodu języka spowoduje wyświetlenie tylko stron w podanym języku.
- `ie` and `oe` (ang. "input encoding" — kodowanie wejściowe, ang. "output encoding" — kodowanie wyjściowe) — Parametry przestarzałe. Oba muszą przyjąć wartość `utf-8`.

Przykład 12.13. Wyszukiwanie w Google

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl')
>>> key = 'YOUR_GOOGLE_API_KEY'
>>> results = server.doGoogleSearch(key, 'mark', 0, 10, False, "",
...     False, "", "utf-8", "utf-8")                                #(1)
>>> len(results.resultElements)                                      #(2)
10
```

```
>>> results.resultElements[0].URL                                     #(3)
'http://diveintomark.org/'
>>> results.resultElements[0].title
'dive into <b>mark</b>'
```

1. Po przygotowaniu obiektu `WSDL.Proxy` możemy wywołać `server.doGoogleSearch` z wszystkimi dziesięcioma parametrami. Pamiętaj o korzystaniu z własnego klucza licencyjnego Google API otrzymanego podczas rejestracji w Google Web Services.

2. Funkcja zwraca mnóstwo informacji, ale wpierw spójrzmy właśnie na wyniki wyszukiwania. Są przechowywane w `results.resultElements`, a dostać się do nich możemy tak jak do elementów zwykłej pythonowej listy.

3. Każdy ze składników `resultElements` jest obiektem zawierającym adres URL (`URL`), tytuł (`title`), urywek tekstu strony (`snippet`) oraz inne użyteczne atrybuty. W tym momencie możesz już korzystać z normalnych technik introspekcji Pythona do podejrzenia zawartości tego obiektu (np. `dir(results.resultElements[0])`). Możesz także tę zawartość podejrzeć przy pomocy obiektu WSDL proxy i atrybutu `outparams` samej funkcji. Obie techniki dają ten sam rezultat.

Obiekt wynikowy zawiera więcej niż tylko wyniki wyszukiwania. Na przykład: informacje na temat procesu szukania (ile trwał, ile wyników znaleziono — pomimo tego, że zwrócono tylko 10). Interfejs www wyszukiwarki pokazuje te informacje, więc są też dostępne metodami programistycznymi.

Przykład 12.14. Pobieranie z Google informacji pomocniczych

```
>>> results.searchTime                                      #(1)
0.224919
>>> results.estimatedTotalResultsCount                      #(2)
29800000
>>> results.directoryCategories                             #(3)
[<SOAPpy.Types.structType item at 14367400>:
 {'fullViewableName':
  'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mark',
  'specialEncoding': }]
>>> results.directoryCategories[0].fullViewableName
'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mark'
```

1. To wyszukiwanie zajęło 0.224919 sekund. Wynik ten nie uwzględnia czasu poświęconego na przesył oraz odbiór dokumentów XML protokołu SOAP. Jest to wyłącznie czas poświęcony przez silnik Google na przetworzenie zapytania, już po otrzymaniu go.

2. Znaleziono około 30 milionów pasujących stron. Dostęp do kolejnych dziesiątek z tego zbioru uzyskamy za pomocą zmiany argumentu `start` metody `server.doGoogleSearch` i kolejnych jej wywołań.

3. Dla niektórych zapytań Google zwraca także listę powiązanych kategorii z katalogu Google. Dołączając zwrócone w ten sposób URL-e do przedrostka http://directory.google.com/ uzyskamy adresy odpowiednich stron katalogu.

13.8 Rozwiązywanie problemów

Rozwiązywanie problemów

Oczywiście świat serwisów SOAP to nie jest tylko kraina mlekiem i miodem płynąca. Czasami coś idzie nie tak.

Jak już widziałeś w tym rozdziale na SOAP składa się kilka warstw. Jest tam warstwa HTTP, ponieważ SOAP wysyła dokumenty XML do i odbiera te dokumenty od serwera HTTP. A więc wszystkie techniki dotyczące debugowania, których nauczyłeś się w rozdziale 11 HTTP, mają zastosowanie także tutaj. Możesz zaimportować `httplib` i ustawić `httplib.HTTPConnection.debuglevel = 1`, aby zobaczyć cały ruch odbywający się poprzez HTTP.

Poza tą warstwą HTTP jest wiele rzeczy, którą mogą sie nie powieść. `SOAPpy` wykonuje godną podziwu robotę ukrywając przed Tobą składnię SOAP, ale to oznacza także, że może być trudne zdiagnozowanie problemu, gdy takowy się pojawi.

Oto kilka przykładów pomyłek, które robiłem używając serwisów SOAP i komunikaty błędów jakie one spowodowały.

Przykład 12.15. Wywoływanie metod z niewłaściwie skonfigurowanym Proxy

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> server = SOAPProxy(url)                                     #(1)
>>> server.getTemp('27502')                                     #(2)
<Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element namespaced?>
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element namespaced?>
```

1. Zauważyłeś pomyłkę? Tworzymy ręcznie `SOAPProxy` i prawidłowo podajemy URL serwisu, ale nie podaliśmy przestrzeni nazw. Ponieważ wiele serwisów może działaś na tym samym URL-u, przestrzeń nazw jest bardzo istotna, aby ustalić do którego serwisu próbujemy się odwołać, a następnie jaką metodę właściwie wywołujemy.

2. Serwer odpowiada poprzez wysłanie SOAP Fault, który SOAPpy zamienia na pythonowy wyjątek typu `SOAPpy.Types.faultType`. Wszystkie błędy zwracane przez serwer SOAP zawsze będą obiektami SOAP Fault, a więc łatwo możemy te wyjątki przechwycić. W tym przypadku, ta czytelna dla człowieka część SOAP Fault daje wskazówkę do tego jaki jest problem: element metoda nie jest zawarty w przestrzeni nazw, ponieważ oryginalny obiekt `SOAPProxy` nie został skonfigurowany z przestrzenią nazw serwisu.

Błędna konfiguracja podstawowych elementów serwisu SOAP jest jednym z problemów, które ma za zadanie rozwiązać WSDL. Plik WSDL zawiera URL serwisu i przestrzeń nazw, a więc nie można ich podać błędnie. Oczywiście nadal są inne rzeczy, które mogą zostać podane błędnie.

Przykład 12.16. Wywołanie metody z nieprawidłowymi argumentami

```
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> temperature = server.getTemp(27502)                              #(1)
<Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>   #(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>
```

1. Zauważyłeś pomyłkę? To jest subtelna pomyłka: wywołujemy `server.getTemp` z liczbą całkowitą (ang. integer) zamiast z łańcuchem znaków (ang. string). Jak już widziałeś w pliku WSDL, funkcja `getTemp`() SOAP przyjmuje pojedyńczy argument, kod pocztowy, który musi być łańcuchem znaków. `WSDL.Proxy` nie będzie konwertował typów danych; musimy podać dokładnie te typy danych jakich serwer oczekuje.

2. I znowu, serwer zwraca SOAP Fault i czytelna dla człowieka część komunikatu błędu daje wskazówkę do tego, gdzie leży problem: wywołujemy funkcję `getTemp` z liczbą całkowitą, ale nie ma zdefiniowanej funkcji o tej nazwie, która przyjmowałaby liczbę całkowitą. W teorii SOAP pozwala na przeciążanie funkcji, a więc jeden serwis SOAP mógłby posiadać dwie funkcje o tej samej nazwie i z taką samą liczbą argumentów, ale z argumentami o różnych typach. O to dlaczego tak ważne jest podawanie właściwych typów. i dlaczego `WSDL.Proxy` nie konwertuje typów danych. Gdyby to robił, to mogłoby się zdarzyć, że wywołalibyśmy zupełnie inną funkcję! Powodzenia w debugowaniu takiego błędu. Dużo łatwiej jest być krytycznym wobec typów danych i zgłaszać błędy tak szybko jak to tylko możliwe.

Jest także możliwe napisanie kodu Pythona, który oczekuje innej liczby zwracanych wartości, niż zdalna funkcja właściwie zwraca.

Przykłąd 12.17. Wywołanie metody i oczekiwanie niewłaściwej liczby zwracanych wartości

```
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> (city, temperature) = server.getTemp(27502)  #(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unpack non-sequence
```

1. Zauważyłeś pomyłkę? `server.getTemp` zwraca tylko jedną wartość, liczbę zmiennoprzecinkową (ang. float), ale my napisaliśmy kod, który zakłada, że otrzymamy dwie wartości i próbuje je przypisać do dwóch oddzielnych zmiennych. Zauważ, że tutaj nie pojawił się wyjątek SOAP fault. Co do zdalnego serwera, to wszystko odbyło się jak należy. Błąd pojawił się dopiero po zakończeniu transakcji SOAP, `WSDL.Proxy` zwrócił liczbę zmiennoprzecinkową a nasz lokalny interpreter Pythona próbował zgodnie z naszym zaleceniem podzielić ją pomiędzy dwie zmienne. Ponieważ funkcja zwróciła tylko jedną wartość, został zgłoszony wyjątek Pythona, a nie SOAP Fault.

A co z serwisem Google? Najczęstszym problemem jaki z nim miałem było to, że zapominałem właściwie ustawić klucz aplikacji.

Przykład 12.18. wywołanie metody z błędem specyficznym dla aplikacji

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy(r'/path/to/local/GoogleSearch.wsdl')
>>> results = server.doGoogleSearch('foo', 'mark', 0, 10, False, "", #(1)
...     False, "", "utf-8", "utf-8")
<Fault SOAP-ENV:Server:                                                #(2)
 Exception from service object: Invalid authorization key: foo:
 <SOAPpy.Types.structType detail at 14164616>:
 {'stackTrace':
  'com.google.soap.search.GoogleSearchFault: Invalid authorization key: foo
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:220)
   at com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
   at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
     GoogleSearchService.java:825)
   at com.google.soap.search.GoogleSearchService.doGoogleSearch(
     GoogleSearchService.java:121)
   at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
   at org.apache.soap.providers.RPCJavaProvider.invoke(
     RPCJavaProvider.java:129)
   at org.apache.soap.server.http.RPCRouterServlet.doPost(
     RPCRouterServlet.java:288)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
   at com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
   at com.google.gse.HttpConnection.run(HttpConnection.java:195)
   at com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of wrong size.
   at com.google.soap.search.UserKey.<init>(UserKey.java:59)
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:217)
   ... 14 more
'}>
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception from service object:
Invalid authorization key: foo:
<SOAPpy.Types.structType detail at 14164616>:
{'stackTrace':
  'com.google.soap.search.GoogleSearchFault: Invalid authorization key: foo
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:220)
   at com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
   at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
     GoogleSearchService.java:825)
   at com.google.soap.search.GoogleSearchService.doGoogleSearch(
     GoogleSearchService.java:121)
   at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
   at org.apache.soap.providers.RPCJavaProvider.invoke(
     RPCJavaProvider.java:129)
   at org.apache.soap.server.http.RPCRouterServlet.doPost(
     RPCRouterServlet.java:288)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
   at com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
   at com.google.gse.HttpConnection.run(HttpConnection.java:195)
   at com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of wrong size.
   at com.google.soap.search.UserKey.<init>(UserKey.java:59)
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:217)
   ... 14 more
'}>
```

1. Zauważyłeś pomyłkę? Nie ma błędów w samej składni wywołania lub w liczbie argumentów lub w typach danych. Problem jest specyficzny dla tej konkretnej aplikacji: pierwszym argumentem powinien być nasz klucz aplikacji, ale `foo` nie jest prawidłowym kluczem dla Google.

2. Serwer Google odpowiada poprzez SOAP Fault i niesamowicie długi komunikat błędu, który zawiera kompletny zrzut stosu Javy. Zapamiętaj, że wszystkie błędy SOAP są oznaczane poprzez SOAP Faults: błędy w konfiguracjach, błędy w argumentach funkcji i błędy specyficzne dla aplikacji jak ten. Zakopana gdzieś tam jest kluczowa informacja: `Invalid authorization key: foo` (niewłaściwy klucz autoryzacji).

13.9 SOAP - podsumowanie

Podsumowanie

Serwisy internetowe SOAP są bardzo skomplikowane. Specyfikacja jest bardzo ambitna i próbuje sprostać wielu różnym przypadkom użycia dla serwisów internetowych. Ten rozdział dotknął jednego z prostszych przypadków użycia.

Zanim zanurkujemy do następnego rozdziału, upewnij się, że opanowałeś następujące kwestie:

- Połączenie się z serwerem SOAP i wywołanie zdalnych metod
- Załadowanie pliku WSDL i użycie go do introspekcji zdalnych metod
- Debugowanie wywołań SOAP ze śledzeniem komunikacji sieciowej
- Rozwiązywanie problemów z najczęstszymi błędami dotyczącymi SOAP

Rozdział 14

Testowanie jednostkowe

14.1 Wprowadzenie do liczb rzymskich

Wprowadzenie do liczb rzymskich

W poprzednich rozdziałach "nurkowaliśmy" poprzez bezpośrednie przyglądanie się kodowi, aby zrozumieć go tak szybko, jak to możliwe. Teraz, gdy już trochę obeznaliśmy Pythona, trochę się cofniemy i spojrzymy na kroki, które trzeba wykonać przed napisaniem kodu.

Kilka rozdziałów wcześniej pisaliśmy, debugowaliśmy i optymalizowaliśmy zbiór użytecznych funkcji, które służyły do konwersji z i na liczby rzymskie. W Podrozdziale 7.3, "Analiza przypadku: Liczby rzymskie", opisaliśmy mechanizm konstruowania i sprawdzania poprawności liczb w zapisie rzymskim, lecz teraz cofnijmy się trochę i zastanówmy się, co moglibyśmy uwzględnić, aby rozszerzyć to narzędzie, by w dwóch kierunkach.

Zasady tworzenia liczb rzymskich prowadzą do kilku interesujących obserwacji:

1. Istnieje tylko jeden poprawny sposób reprezentowania pewnej liczby w postaci rzymskiej.

2. Odwrotność też jest prawdą: jeśli ciąg znaków jest poprawną liczbą rzymską, to reprezentuje ona tylko jedną liczbę (tzn. możemy ją przeczytać tylko w jeden sposób).

3. Tylko ograniczony zakres liczb może być zapisany jako liczby rzymskie, a dokładniej liczby od 1 do 3999 (Liczby rzymskiej posiadają kilka sposobów wyrażania większych liczb np. poprzez dodanie nadkreślenia nad cyframi rzymskimi, co oznacza, że normalną wartość tej liczby trzeba pomnożyć przez 1000, jednak nie będziemy się wdawać w szczegóły. Dla potrzeb tego rozdziału, założymy, że liczby rzymskie idą od 1 do 3999).

4. Nie mamy możliwość zapisania 0 jako liczby rzymskiej. (Co ciekawe, starożytni Rzymianie nie wyobrażali sobie 0 jako liczby. Za pomocą liczb liczymy, ile czegoś mamy, jednak jak możemy policzyć coś, czego nie mamy?)

5. Nie możemy w postaci liczby rzymskiej zapisać liczby ujemnej.

6. W postaci liczby rzymskiej nie możemy zapisywać ułamków, czy liczb, które nie są całkowite.

Biorąc to wszystko pod uwagę, co możemy oczekiwać od zbioru funkcji, które konwertują z i na liczby rzymskie? Wymagania `roman.py`:

1. `toRoman` powinien zwracać rzymską reprezentację wszystkich liczb całkowitych z zakresu od 1 do 3999.

2. `toRoman` powinien nie zadziałać (ang. *fail*), gdy otrzyma liczbę całkowitą z poza przedziału od 1 do 3999.

3. `toRoman` powinien nie zadziałać, gdy otrzyma niecałkowitą liczbę.

4. `fromRoman` powinien przyjmować poprawną liczbę rzymską i zwrócić liczbę, która ją reprezentuje.

5. `fromRoman` powinien nie zadziałać, kiedy otrzyma niepoprawną liczbę rzymską.

6. Kiedy daną liczbę konwertujemy na liczbę rzymską, a następnie z powrotem na liczbę, powinniśmy otrzymać tę samą liczbę, z którą zaczynaliśmy. Więc dla każdego `n` od 1 do 3999 `fromRoman(toRoman(n)) == n`.

7. `toRoman` powinien zawsze zwrócić liczbę rzymską korzystając z wielkich liter.

8. `fromRoman` powinien akceptować jedynie liczby rzymskie składające się z wielkich liter (tzn. powinien nie zadziałać, gdy otrzyma wejście złożone z małych liter).

14.2 Testowanie - nurkujemy

Nurkujemy

Teraz, kiedy w pełni zdefiniowaliśmy zachowanie funkcji konwertujących, zrobimy coś odrobinę niespodziewanego: napiszemy zestaw testów, który pokaże, co te funkcje potrafią, a także upewni nas, że robią dokładnie to, co chcemy. Dobrze usłyszeliście: zaczniemy od napisania kodu testującego kod, który nie został jeszcze napisany.

Takie podejście nazywa się testowaniem jednostkowym, ponieważ zestaw dwóch funkcji konwertujących może być napisany i przetestowany jako jednostka, niezależnie od kodu większego programu, jakiego częścią może się ów zestaw stać w przyszłości. Python posiada gotowe narzędzie służące do testowania jednostkowego — moduł o nazwie `unittest`.

Moduł `unittest` jest częścią dystrybucji języka Python od wersji 2.1 wzwyż. Użytkownicy starszych wersji (np. Python 2.0) mogą pobrać ten moduł ze strony pyunit.sourceforge.net.

Testowanie jednostkowe jest ważnym elementem strategii rozwoju oprogramowania, w której na pierwszym miejscu stawia się testowanie. Jeśli ma być napisany jakiś test, to ważne jest, aby był on napisany jak najwcześniej (możliwie przed napisaniem testowanego kodu) oraz aby był aktualizowany wraz ze zmieniającymi się wymaganiami. Testowanie jednostkowe nie zastępuje testowania wyższego poziomu, takiego jak testowanie funkcjonalne czy systemowe, ale jest bardzo istotne we wszystkich fazach rozwoju oprogramowania:

1. Jeszcze przed napisaniem kodu zmusza nas do sprecyzowania i wyrażenia wymagań w użyteczny sposób.

2. Podczas pisania kodu chroni nas od niepotrzebnego kodowania. Kiedy wszystkie testy przechodzą, testowana funkcja jest już gotowa.

3. Podczas refaktoryzacji upewnia nas, że nowa wersja kodu zachowuje się tak samo, jak stara.

4. W procesie utrzymania kodu ochrania nas, kiedy ktoś przychodzi do nas z krzykiem, że nasza ostatnia zmiana popsuła jego stary kod ("Ależ proszę pana, w momencie mojego czekinowania wszystkie testy przechodziły...").

5. Podczas programowania w zespole zwiększa pewność, że nowy kod, który chcemy dodać, nie popsuje kodu innych osób, ponieważ najpierw uruchomimy ich testy (Widziałem to już podczas tzw. sprintów. Zespół dzieli zadanie między siebie, każdy otrzymuje specyfikację tego, nad czym będzie pracować, pisze do tego testy jednostkowe, a następnie dzieli się tymi testami z pozostałymi członkami zespołu. W ten sposób nikt nie posunie się zbyt daleko w rozwijaniu kodu, który nie współdziała z kodem innych osób).

14.3 Wprowadzenie do romantest.py

Wprowadzenie do romantest.py

Poniżej przedstawiono pełny zestaw testów do funkcji konwertujących, które nie zostały jeszcze napisane, ale wkrótce znajdą się w roman.py. Nie jest wcale oczywiste, jak to wszystko ze sobą działa; żadna z poniższych klas i metod nie odnosi się do żadnej innej. Jak wkrótce zobaczymy, ma to swoje uzasadnienie.

Przykład 13.1. romantest.py

Jeśli jeszcze tego nie zrobiłeś, możesz pobrać ten oraz inne przykłady (http://diveintopython.org/download/diveintopython-examples-5.4.zip) używane w tej książce.

```
"""Unit test for roman.py"""

import roman
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
```

```
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)
```

```
    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

Materiały dodatkowe

- Na stronie domowej PyUnit znajduje się szeroka dyskusja na temat używania zrębu unittest, łącznie z zagadnieniami zaawansowanymi, których w tym rozdziale nie poruszono.
- Często Zadawane Pytania dotyczące PyUnit wyjaśniają, dlaczego przypadki testowe umieszczane są oddzielnie w stosunku do kodu, który testują.
- Python Library Reference podsumowuje moduł unittest
- ExtremeProgramming.org omawia przyczyny, dla których należy pisać testy jednostkowe
- Na stronach The Portland Pattern Repository można znaleźć trwającą wciąż dyskusję na temat testów jednostkowych, standardową definicję testu jednostkowego, przyczyny pisania testów przed pisaniem kodu oraz wiele dogłębnych analiz na ten temat.

14.4 Testowanie poprawnych przypadków

Tworzenie poszczególnych przypadków testowych należy do najbardziej podstawowych elementów testowania jednostkowego. Przypadek testowy stanowi odpowiedź na pewne pytanie dotyczące kodu, który jest testowany.

Przypadek testowy powinien:

- ...działać bez konieczności wprowadzania danych przez człowieka. Testowanie jednostkowe powinno być zautomatyzowane.
- ...samodzielnie stwierdzać, czy testowana funkcja działa poprawnie, czy nie, bez konieczności interpretacji wyników przez człowieka.
- ...działać w izolacji, oddzielnie i niezależnie od innych przypadków testowych (nawet wówczas, gdy testują one te same funkcje). Każdy przypadek testowy powinien być "wyspą".

Zbudujmy więc pierwszy przypadek testowy, biorąc powyższe pod uwagę. Mamy następujące wymaganie:

1. Funkcja `toRoman` powinna zwracać tekstową reprezentację w zapisie rzymskim wszystkich liczb całkowitych z przedziału od 1 do 3999.

Przykład 13.2. testToRomanKnownValues

```
class KnownValues(unittest.TestCase):                                    #(1)
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
```

```
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'))                          #(2)

  def testToRomanKnownValues(self):                               #(3)
      """toRoman should give known result with known input"""
      for integer, numeral in self.knownValues:
          result = roman.toRoman(integer)                         #(4) #(5)
          self.assertEqual(numeral, result)                       #(6)
```

1. W celu utworzenia przypadku testowego tworzymy nową podklasę klasy `TestCase` z modułu `unittest`. Klasa `TestCase` udostępnia wiele użytecznych metod, które można użyć we własnym przypadku testowym celem przetestowania określonych warunków.
2. Jest to lista par liczba całkowita/wartość w zapisie rzymskim, których poprawność sprawdziłem ręcznie. Zawiera ona dziesięć najniższych liczb, liczbę największą, każdą liczbę, która jest reprezentowana przy pomocy jednego znaku w zapisie rzymskim oraz pewne inne, losowo wybrane wartości. Celem przypadku testowego nie jest przetestowanie wszystkich mogących się pojawić danych wejściowych, lecz pewnej reprezentatywnej próbki.

3. Każdy pojedynczy test posiada swoją metodę, która nie bierze żadnych parametrów oraz nie zwraca żadnej wartości. Jeśli metoda zakończy się normalnie bez rzucenia wyjątku, uznaje się wówczas, że taki test przeszedł; jeśli z metody zostanie rzucony wyjątek, wówczas uznaje się, że test nie przeszedł.

4. W tym miejscu wołamy funkcję `toRoman`. (Rzeczywiście, funkcja ta nie została jeszcze napisana, ale kiedy już ją napiszemy, ta właśnie linijka spowoduje jej wywołanie). Zauważmy, że właśnie zdefiniowaliśmy API funkcji `toRoman`: pobiera ona argument typu `int` (liczbę, która ma zostać przekształcona na zapis rzymski) i zwraca wartość typu `string` (rzymską reprezentację wartości przekazanej w parametrze). Jeśli rzeczywiste API będzie inne, ten test zakończy się niepowodzeniem.

5. Zauważmy również, że podczas wywoływania `toRoman` nie łapiemy żadnych wyjątków. Jest to celowe. Funkcja `toRoman` nie powinna zgłaszać wyjątków w sytuacji, gdy wywołujemy ją z prawidłowymi wartościami, a wszystkie wartości, z którymi ją wywołujemy, są poprawne. Jeśli `toRoman` rzuci wyjątek, test zakończy się niepowodzeniem.

6. Jeśli założymy, że funkcja `toRoman` została poprawnie zdefiniowana i wywołana oraz poprawnie się zakończyła, zwracając pewną wartość, to ostatnią rzeczą, jaką musimy sprawdzić, jest to, czy zwrócona wartość jest poprawna. Tego rodzaju sprawdzenie jest bardzo powszechne, a w klasie `TestCase` istnieje metoda `assertEqual`, która może w tym pomóc: sprawdza ona, czy dwie wartości są sobie równe. Jeśli wartość zwrócona przez funkcję `toRoman` (`result`) nie jest równa znanej nam, spodziewanej wartości (`numeral`), `assertEqual` spowoduje rzucenie wyjątku, a test zakończy się niepowodzeniem. Jeśli te dwie wartości są równe, metoda ta nic nie robi. Jeśli każda wartość zwrócona przez `toRoman` pasuje do wartości, której się spodziewamy, to `assertEqual` nigdy nie rzuci wyjątku, a więc `testToRomanKnownValues` zakończy się normalnie, co oznacza, że funkcja `toRoman` przeszła ten test.

14.5 Testowanie niepoprawnych przypadków

Testowanie funkcji w sytuacji, w której na wejściu pojawiają się wyłącznie poprawne wartości, nie jest wystarczające; należy dodatkowo sprawdzić, że funkcja kończy się niepowodzeniem, gdy otrzymuje ona niepoprawne dane wejściowe. Nie może to być jednak dowolne niepowodzenie; musi być ono dokładnie takie, jakiego się spodziewamy.

Przypomnijmy sobie pozostałe wymagania dotyczące funkcji `toRoman`:

2. Funkcja `toRoman` powinna kończyć się niepowodzeniem, gdy przekazana jest jej wartość spoza przedziału od 1 do 3999.

3. Funkcja `toRoman` powinna kończyć się niepowodzeniem, gdy przekazana jest jej wartość nie będąca liczbą całkowitą.

W języku Python funkcje kończą się niepowodzeniem wówczas, gdy rzucają wyjątki. W module `unittest` znajdują się natomiast metody, dzięki którym można wykryć, czy funkcja, otrzymawszy niepoprawne dane wejściowe, rzuca odpowiedni wyjątek:

Przykład 13.3. Testowanie niepoprawnych danych wejściowych do funkcji `toRoman`

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)          #(1)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)             #(2)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)           #(3)
```

1. Klasa `TestCase` z modułu `unittest` udostępnia metodę `assertRaises`, która przyjmuje następujące argumenty: wyjątek, którego się spodziewamy, funkcję, którą testujemy oraz argumenty, które mają być przekazane do funkcji (jeśli testowana funkcja przyjmuje więcej niż jeden argument, należy je wszystkie przekazać po kolei do funkcji `assertRaises`, która przekaże je w tej właśnie kolejności do testowanej funkcji). Zwróćcie baczną uwagę na to, co tutaj robimy: zamiast ręcznego wywoływania funkcji i sprawdzania, czy został rzucony wyjątek odpowiedniego typu (poprzez otoczenie wywołania blokiem try...except), używamy funkcji `assertRaises`, która robi to wszystko za nas. Wszystko, co należy zrobić, to przekazać typ wyjątku (`roman.OutOfRangeError`), funkcję (`toRoman`) oraz jej argument (4000), a `assertRaises` zajmie się wywołaniem `toRoman` z przekazanym parametrem oraz sprawdzeniem, czy rzucony wyjątek to rzeczywiście `roman.OutOfRangeError`. (Zauważmy również, że do funkcji `assertRaises` przekazujemy funkcję `toRoman` jako parametr; nie wywołujemy jej ani nie przekazujemy jej nazwy w postaci napisu. Czy wspominałem ostatnio, jak bardzo

przydatne jest to, że w języku Python wszystko jest obiektem, włączając w to funkcje i wyjątki?)

2. Oprócz przetestowania wartości zbyt dużych należy też przetstować wartości zbyt małe. Pamiętajmy, że w zapisie rzymskim nie można wyrazić ani wartości 0, ani liczb ujemnych, więc dla każdej z tych sytuacji mamy przypadek testowy (`testZero` i `testNegative`). W funkcji `testZero` sprawdzamy, czy `toRoman` rzuca wyjątek `roman.OutOfRangeError`, gdy jest wywołana z wartością 0; jeśli nie rzuci tego wyjątku (zarówno z powodu zwrócenia pewnej wartości jak i rzucenia jakiegoś innego wyjątku), test powinien zakończyć się niepowodzeniem.

3. Wymaganie #3 określa, że `toRoman` nie może przyjąć jako danych wejściowych liczb niecałkowitych, więc tutaj upewniamy się, że dla wartości 0.5 `toRoman` rzuci wyjątek `roman.NotIntegerError`. Jeśli `toRoman` nie rzuci takiego wyjątku, test ten powinien zakończyć się niepowodzeniem.

Kolejne dwa wymagania są podobne do pierwszych trzech, przy czym odnoszą się one do funkcji `fromRoman` zamiast `toRoman`:

4. Funkcja `fromRoman` powinna przyjmować napis będący poprawną liczbą w zapisie rzymskim i zwracać liczbę całkowitą, którą ten napis reprezentuje.

5. Funkcja `fromRoman` powinna zakończyć się niepowodzeniem, gdy otrzyma na wejściu napis nie będący poprawną liczbą w zapisie rzymskim.

Wymaganie #4 jest obsługiwane w podobny sposób, jak wymaganie #1, poprzez iterowanie po zestawie znanych wartości i testowanie każdej z nich. Wymaganie #5 jest z kolei obsługiwane podobnie, jak wymagania #2 i #3, poprzez testowanie serii niepoprawnych ciągów wejściowych i sprawdzanie, czy `fromRoman` rzuca odpowiedni wyjątek.

Przykład 13.4. Testowanie niepoprawnych danych wejściowych do funkcji `fromRoman`

```
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)   #(1)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
```

1. Nie ma tu nic nowego do powiedzenia: wzorzec postępowania jest dokładnie taki sam jak w przypadku testowania niepoprawnego wejścia do funkcji `toRoman`. Zaznaczę tylko, że mamy teraz nieco inny wyjątek: `roman.InvalidRomanNumeralError`. Okazało się więc, że potrzebujemy trzech określonych przez nas wyjątków, które

powinny zostać zdefiniowane w `roman.py` (wraz z `roman.OutOfRangeError` i `roman.NotIntegerError`). Kiedy już zajmiemy się implementacją `roman.py` w dalszej części tego rozdziału, dowiesz się, jak definiować własne wyjątki.

14.6 Testowanie zdroworozsądkowe

Dość często zdarza się, że pewien fragment kodu zawiera zbiór funkcji powiązanych ze sobą; zwykle są to funkcje konwertujące, z których pierwsza przekształca A do B, a druga przekształca B do A. W takim przypadku rozsądnie jest utworzyć "zdroworozsądkowe sprawdzenie", dzięki któremu upewnimy się, że możemy przekształcić A do B i z powrotem do A bez utraty dokładności, bez wprowadzania błędów zaokrągleń i bez powodowania jakichkolwiek błędów innego typu.

Rozważmy następujące wymaganie:

6. Jeśli mamy pewną wartość liczbową, którą przekształcamy na reprezentację w zapisie rzymskim, a tę przekształcamy z powrotem do wartości liczbowej, powinniśmy otrzymać wartość, od której rozpoczynaliśmy przekształcenie. A więc `fromRoman`(`toRoman`(`n`)) == `n` dla każdego `n` w przedziale 1..3999.

Przykład 13.5. Testowanie `toRoman` względem `fromRoman`

```
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):                             #(1) #(2)
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)                      #(3)
```

1. Funkcję `range` widzieliśmy już wcześniej, z tym, że tutaj wywołana jest ona z dwoma parametrami, dzięki czemu zwraca listę kolejnych liczb całkowitych z przedziału od wartości będącej pierwszym argumentem funkcji (1) do wartości będącej drugim argumentem funkcji (4000), bez tej wartości. Zwróci więc kolejne liczby z przedziału 1..3999, które stanowią zakres poprawnych wartości wejściowych do funkcji konwertującej na notację rzymską.

2. Jeśli już tu jesteśmy, to wspomnę tylko, że integer nie jest słowem kluczowym języka Python; zostało ono użyte po prostu jako nazwa zmiennej.

3. Właściwa logika testująca jest oczywista: bierzemy liczbę całkowitą (`integer`), przekształcamy ją do reprezentacji rzymskiej (`numeral`), następnie reprezentację tą przekształcamy z powrotem do wartości całkowitej (`result`) i upewniamy się, że otrzymaliśmy tę samą wartość, od której rozpoczęliśmy przekształcenia. Jeśli nie jest to prawdą, wówczas `assertEqual` rzuci wyjątek, a test natychmiast zakończy się niepowodzeniem. Jeśli zaś każda liczba po przekształceniach jest równa wartości początkowej to `assertEqual` zakończy się prawidłowo, również `testSanity` zakończy się prawidłowo, a test zakończy się powodzeniem.

Ostatnie dwa wymagania różnią się od poprzednich, ponieważ wydają się arbitralne i trywialne zarazem:

7. Funkcja `toRoman` powinna zwracać napis reprezentujący liczbę w notacji rzymskiej przy użyciu wyłącznie wielkich liter.

8. Funkcja `fromRoman` powinna akceptować na wejściu napisy reprezentujące liczby w notacji rzymskiej pisane wyłącznie wielkimi literami (tj. powinna zakończyć się niepowodzeniem, gdy w napisie wejściowym znajdują się małe litery).

Nie da się ukryć, że wymagania te są trochę arbitralne. Moglibyśmy przecież ustalić, że `fromRoman` przyjmuje zarówno napisy składające się z małych liter, jak również napisy zawierające zarówno małe, jak i duże litery. Z drugiej strony, wymagania te nie są całkowicie arbitralne: jeśli `toRoman` zawsze zwraca napisy składające się z wielkich liter, wówczas `fromRoman` musi akceptować na wejściu przynajmniej te napisy, które składają się wyłącznie z wielkich liter, inaczej "zdroworozsądkowe sprawdzenie" (wymaganie #6) zakończy się niepowodzeniem. Ustalenie, że na wejściu przyjmujemy napisy złożone wyłącznie z wielkich liter, jest arbitralne, jednak — jak potwierdzi to każdy integrator systemów — wielkość znaków ma zawsze znaczenie, a więc warto od razu tę kwestię wyspecyfikować. A skoro warto ją wyspecyfikować, to warto ją również przetestować.

Przykład 13.6. Testowanie wielkości znaków

```
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())                      #(1)

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())                                #(2) #(3)
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())             #(4)
```

1. Najciekawsze w powyższym teście jest to, jak wielu rzeczy on nie testuje. Nie testuje tego, czy wartość zwrócona przez `toRoman` jest prawidłowa czy choćby spójna; na te pytania odpowiadają inne przypadki testowe. Ten przypadek testowy sprawdza wyłącznie wielkość liter. Ponieważ zarówno on jak i "sprawdzenie zdroworozsądkowe" przebiegają przez wszystkie wartości z zakresu i wywołują `toRoman`, to możecie spotkać się z pokusą, aby obydwa te przypadki połączyć w jeden[1]. Jednak działanie takie pogwałciłoby jedną z podstawowych zasad testowania: każdy przypadek testowy powinien odpowiadać na dokładnie jedno pytanie. Wyobraźmy sobie, że połączyliśmy sprawdzenie wielkości liter ze sprawdzeniem zdroworozsądkowym, a nowopowstały przypadek testowy zakończył się niepowodzeniem. W takiej sytuacji stanęlibyśmy przed koniecznością głębszego przeanalizowania tego przypadku, aby dowiedzieć się, w której części testu pojawił się problem, a więc co tak naprawdę owo niepowodzenie oznacza. Jeśli musicie analizować wyniki testów po to, aby dowiedzieć się, co one oznaczają, to jest to oczywisty znak, że wasze przypadki testowe zostały źle zaprojektowane.

2. Podobną lekcję otrzymujemy w tym miejscu: nawet, jeśli "wiemy", że funkcja `toRoman` zawsze zwraca wielkie litery, to aby przetestować, że `fromRoman` przyjmuje napis złożony z wielkich liter, tutaj jawnie przekształcamy wartość wynikową `toRoman` do wielkich liter. Dlaczego to robimy? Otóż dlatego, że zwracanie

[1] "Oprę się wszystkiemu za wyjątkiem pokusy." –Oscar Wilde

przez `toRoman` wielkich liter wynika z niezależnego wymagania. Jeśli to wymaganie zostanie zmienione tak, że na przykład, funkcja ta będzie zawsze zwracała małe litery, to choć `testToRomanCase` będzie musiał się zmienić, ten test będzie wciąż działał. To kolejna z podstawowych zasad testowania: każdy przypadek testowy musi działać niezależnie od innych przypadków. Każdy test jest wyspą.

3. Zauważcie, że wartości zwracanej przez `fromRoman` nigdzie nie przypisujemy. W języku Python taka składnia jest poprawna; jeśli funkcja zwraca pewną wartość, ale nikt nie jest nią zainteresowany, Python po prostu tę wartość wyrzuca. W tym przypadku właśnie tego chcemy. Ten przypadek testowy w żaden sposób nie testuje wartości zwracanej; testuje jedynie to, czy `fromRoman` akceptuje napis złożony z wielkich liter i nie rzuca przy tym wyjątku.

4. Ta linijka, choć skomplikowana, bardzo przypomina to, co zrobiliśmy w testach `ToRomanBadInput` i `FromRomanBadInput`. W tym teście upewniamy się, że wywołanie pewnej funkcji (`roman.fromRoman`) z pewnym szczególnym parametrem (`numeral.lower()`, bieżąca wartość rzymska pochodząca z pętli, pisana małymi literami) rzuci określony wyjątek (`roman.InvalidRomanNumeralError`). Jeśli tak się stanie (dla każdej wartości z pętli), test zakończy się powodzeniem; jeśli zaś przynajmniej raz zdarzy się coś innego (zostanie rzucony inny wyjątek lub zostanie zwrócona wartość bez rzucania wyjątku), test zakończy się niepowodzeniem.

W następnym rozdziale zobaczymy, jak napisać kod, który wszystkie te testy przechodzi.

Rozdział 15

Testowanie 2

15.1 roman.py, etap 1

Teraz, gdy już są gotowe testy jednostkowe, nadszedł czas na napisanie testowanego przez nie kodu. Zrobimy to w kilku etapach, dzięki czemu będziecie mieli okazję najpierw zobaczyć, że wszystkie testy kończą się niepowodzeniem, a następnie prześledzić, w jaki sposób zaczynają przechodzić, jeden po drugim, tak, że w końcu zapełnione zostaną wszelkie luki w module `roman1.py`.

Przykład 14.1. roman1.py

Plik jest dostępny w katalogu in py/roman/stage1/ wewnątrz katalogu examples.

Jeśli jeszcze tego nie zrobiliście, możecie pobrać ten oraz inne przykłady używane w tej książce stąd.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass                #(1)
class OutOfRangeError(RomanError): pass          #(2)
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass #(3)

def toRoman(n):
    """convert integer to Roman numeral"""
    pass                                         #(4)

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

1. W ten sposób w języku Python definiujemy nasze własne wyjątki. Wyjątki są klasami, a tworzy się je przez utworzenie klasy pochodnej po jednej z już istniejących klas reprezentujących wyjątki. Zaleca się (choć nie jest to wymagane), aby klasy pochodne tworzyć po klasie `Exception` będącej klasą bazową dla wszystkich wyjątków wbudowanych. W tym miejscu definiuję `RomanError`, która stanowić będzie kasą bazową dla wszystkich nowych klas wyjątków, o których powiem później. Utworzenie bazowej klasy wyjątku jest kwestią stylu; równie łatwo mógłbym każdą nową klasę wyjątku wyprowadzić bezpośrednio z klasy `Exception`.

2. Wyjątki `OutOfRangeError` oraz `NotIntegerError` będą wykorzystywane przez funkcję `fromRoman` do poinformowania otoczenia o różnych nieprawidłowościach w danych wejściowych, tak jak zostało to zdefiniowane w `ToRomanBadInput`.

3. Wyjątek `InvalidRomanNumeralError` będzie wykorzystany przez funkcję `fromRoman` do oznaczenia nieprawidłowości w danych wejściowych, tak jak zostało to zdefiniowane w `FromRomanBadInput`.

4. Na tym etapie dążymy do tego, aby zdefiniować API każdej z naszych funkcji, jednak nie chcemy jeszcze pisać ich kodu. Sygnalizujemy to używając słowa kluczowego `pass`.

Nadeszła teraz wielka chwila (wchodzą werble!): możemy w końcu uruchomić testy na naszym małym, kadłubkowym module. W tej chwili każdy przypadek testowy

powinien zakończyć się niepowodzeniem. W istocie, jeśli na etapie 1 którykolwiek test przejdzie, powinniśmy wrócić do `romantests.py` i zastanowić się, dlaczego napisaliśmy tak bezużyteczny test, że przechodzi on dla funkcji, które w rzeczywistości nic nie robią.

Uruchomcie `romantest1.py` podając w linii poleceń opcję -v, dzięki której otrzymamy dokładniejsze informacje i będziemy mogli prześledzić, ktory test jest uruchamiany. Przy odrobinie szczęścia wyjście powinno wyglądać tak:

Przykład 14.2. Wyjście programu romantest1.py testującego roman1.py

```
fromRoman should only accept uppercase input ... ERROR
toRoman should always return uppercase ... ERROR
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... FAIL
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL

======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 154, in testFromRomanCase
    roman1.fromRoman(numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 148, in testToRomanCase
    self.assertEqual(numeral, numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 127, in testRepeatedPairs
```

```
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 122,
in testTooManyRepeatedNumerals
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 99,
in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 93,
in testToRomanKnownValues
    self.assertEqual(numeral, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: I != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 116, in testNonInteger
    self.assertRaises(roman1.NotIntegerError, roman1.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
```

```
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 112, in testNegative
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 104, in testTooLarge
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input                                 #(1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 108, in testZero
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError                                        #(2)
----------------------------------------------------------------------
Ran 12 tests in 0.040s                                                 #(3)

FAILED (failures=10, errors=2)                                         #(4)
```

1. Po uruchomieniu skryptu zostaje wywołana funkcja `unittest.main()`, która z kolei wywołuje każdą z metod zdefiniowanych w każdej klasie wewnątrz `romantest.py`. Dla każdego przypadku testowego wypisywany jest napis dokumentujący odpowiadającej mu metody oraz to, czy przypadek testowy przeszedł, czy nie. Tak, jak się spodziewaliśmy, żaden test nie przeszedł.

2. Dla każdego przypadku testowego, który zakończył się niepowodzeniem, `unittest` wypisuje zawartość stosu, dzięki czemu widać dokładnie, co się stało. W tym przypadku wywołanie funkcji `assertRaises` (znanej również pod nazwą `failUnlessRaises`) spowodowało rzucenie wyjątku `AssertionError` z tego powodu, że w teście spodziewaliśmy się, że `toRoman` rzuci `OutOfRangeError`, a taki wyjątek nie został rzucony.

3. Po wypisaniu szczegółów, `unittest` wypisuje podsumowanie zawierające informacje o tym, ile testów zostało uruchomionych oraz jak długo one trwały.

4. Ogólnie rzecz biorąc, test jednostkowy nie przechodzi, jeśli przynajmniej jeden przypadek testowy nie przechodzi. Kiedy przypadek testowy nie przej-

dzie, `unittest` rozróżnia niepowodzenia (failures) i błędy (errors). Niepowodzenie występuje w przypadku wywołań metod assertXYZ, np. `assertEqual` czy `assertRaises`, które kończą się niepowodzeniem, ponieważ nie został spełniony pewien zakładany warunek albo nie został rzucony spodziewany wyjątek. Błąd natomiast występuje wówczas, gdy zostanie rzucony jakikolwiek inny wyjątek i to zarówno w kodzie testowanym, jak i w kodzie samego testu. Na przykład błąd wystąpił w metodzie `testFromRomanCase` ("Funkcja `fromRoman` powinna akceptować na wejściu napisy zawierające wyłącznie wielkie litery"), ponieważ wywołanie `numeral.upper()` rzuciło wyjątek `AttributeError`: `toRoman` miało zwrócić napis, a tego nie zrobiło. Natomiast `testZero` ("Funkcja `toRoman` otrzymująca na wejściu wartość 0 powinna zakończyć się niepowodzeniem") zakończyła się niepowodzeniem, ponieważ wywołanie `fromRoman` nie rzuciło wyjątku `InvalidRomanNumeral`, którego spodziewał się `assertRaises`.

15.2 roman.py, etap 2

Strukturę modułu `roman` mamy już z grubsza określoną, nadszedł więc czas na napisanie kodu i sprawienie, że nasze testy zaczną w końcu przechodzić.

Przykład 14.3. roman2.py

Plik jest dostępny w katalogu in py/roman/stage2/ wewnątrz katalogu examples.

Jeśli jeszcze tego nie zrobiliście, możecie pobrać ten oraz inne przykłady używane w tej książce stąd.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),                  #(1)
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:                       #(2)
            result += numeral
            n -= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

1. `romanNumeralMap` jest krotką krotek, która definiuje trzy elementy:

    (a) reprezentację znakową najbardziej podstawowych liczb rzymskich; zauważcie, że nie są to wyłącznie liczby, których reprezentacja składa się z jednego znaku; zdefiniowane są również pary dwuznakowe, takie jak CM (“o

sto mniej niż tysiąc"), dzięki którym kod funkcji `toRoman` będzie znacznie prostszy

(b) porządek liczb rzymskich; są one uporządkowane malejąco względem ich liczbowej wartości od M do I

(c) wartość liczbową odpowiadającą reprezentacji rzymskiej; każda wewnętrzna krotka jest parą (reprezentacja rzymska, wartość liczbowa)

2. To jest właśnie miejsce, w którym widać, że opłacało się wprowadzić opisaną wyżej bogatą strukturę danych — nie potrzebujemy żadnej specjalnej logiki do obsłużenia reguły odejmowania. Aby przekształcić wartość liczbową do reprezentacji rzymskiej wystarczy przeiterować po `romanNumeralMap` szukając najwyższej wartości całkowitej mniejszej bądź równej wartości wejściowej. Po jej znalezieniu dopisujemy odpowiadającą jej reprezentację rzymską na koniec napisu wyjściowego, odejmujemy jej wartość od wartości wejściowej, pierzemy, płuczemy, powtarzamy.

Przykład 14.4. Jak działa `toRoman`

Jeśli sposób działania funkcji `toRoman` nie jest całkiem jasny, dodajcie na koniec pętli `while` instrukcję `print`:

```
        while n >= integer:
            result += numeral
            n -= integer
            print 'subtracting', integer, 'from input, adding', numeral, 'to output'
```

```
>>> import roman2
>>> roman2.toRoman(1424)
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

Funkcja `toRoman` wydaje się działać, przynajmniej w przypadku tego szybkiego, ręcznego sprawdzenia. Czy jednak przechodzi ona testy? Cóż, niezupełnie.

Przykład 14.5. Wyjście programu romantest2.py testującego roman2.py

Pamiętajcie o tym, aby uruchomić `romantest2.py` z opcją -v w linii poleceń, dzięki czemu włączy się tryb "rozwlekły".

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok                          #(1)
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok               #(2)
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL                    #(3)
toRoman should fail with negative input ... FAIL
```

```
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL
```

1. Ponieważ w `romanNumeralMap` reprezentacja liczb rzymskich jest wyrażona przy pomocy wielkich liter, funkcja `toRoman` rzeczywiście zawsze zwraca napisy złożone z wielkich liter. A więc ten test przechodzi.

2. Tu pojawia się istotna wiadomość: obecna wersja `toRoman` przechodzi test znanych wartości. Choć test ten nie jest zbyt wyczerpujący, sprawdza on wiele spośród poprawnych danych wejściowych, wliczając w to wartości, które powinny dać w wyniku każdą reprezentację jednoliterową, największą możliwą wartość (3999) czy też wartość, która daje w wyniku najdłuższą reprezentację rzymską (3888). Na tej podstawie możemy być raczej pewni, że funkcja zwróci poprawną reprezentację dla wszystkich poprawnych danych wejściowych.

3. Niestety, funkcja "nie działa" dla nieprawidłowych danych wejściowych; nie przechodzi żaden test badający działanie funkcji dla niepoprawnych danych. Ma to sens, ponieważ nie umieściliśmy jeszcze w kodzie funkcji żadnego sprawdzenia dotyczącego błędnych danych. Testy, o których tu mówimy, sprawdzają (używając `assertRaises`), czy w takich sytuacjach zostaje rzucony odpowiedni wyjątek, a my nigdzie go nie rzucamy. Zrobimy to jednak już w następnym etapie.

Poniżej znajduje się dalszy ciąg wyjścia po uruchomieniu testów jednostkowych, prezentujący szczegóły niepowodzeń. Jest ich aż 10.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 156, in testFromRomanCase
    roman2.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 127, in testRepeatedPairs
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
```

```
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 122,
in testTooManyRepeatedNumerals
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 99,
in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 116, in testNonInteger
    self.assertRaises(roman2.NotIntegerError, roman2.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 112, in testNegative
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
```

```
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 104, in testTooLarge
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 108, in testZero
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
----------------------------------------------------------------------
Ran 12 tests in 0.320s

FAILED (failures=10)
```

15.3 roman.py, etap 3

roman.py, etap 3

Teraz już toRoman odpowiednio sobie radzi z dobrym wejściem (liczbami całkowitymi od 1 do 3999), więc teraz jest czas zająć się niepoprawnym wejściem (wszystkim innym).

Przykład 14.6. roman3.py

Plik ten jest dostępny z py/roman/stage3/ w katalogu przykładów.

Jeśli jeszcze tego nie zrobiłeś, możesz pobrać ten i inne przykłady wykorzystane w tej książce.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):                                              #(1)
        raise OutOfRangeError, "number out of range (must be 1..3999)" #(2)
    if int(n) <> n:                                                     #(3)
        raise NotIntegerError, "non-integers can not be converted"

    result = ""                                                         #(4)
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

def fromRoman(s):
```

```
    """convert Roman numeral to integer"""
    pass
```

1. Jest to przyjemny pythonowy skrót: wielokrotne porównanie. Jest to odpowiednik do `if not ((0 < n) and (n < 4000))`, jednak łatwiejszy do odczytu. Za pomocą tego kontrolujemy zakres wartości i sprawdzamy, czy wprowadzona liczba nie jest za duża, ujemna, czy też równa zero.
2. Wyrzucamy wyjątek za pomocą wyrażenia `raise`. Możemy wyrzucić każdy wbudowane wyjątek, a także inny zdefiniowany przez nas wyjątek. Drugi parametr, *wiadomość błędu*, jest opcjonalny; jeśli dostaniemy wyjątek i nigdzie jego nie obsłużymy, zostanie on wyświetlone w traceback (w postaci śladów stosu).
3. Za pomocą tego sprawdzamy, czy liczba nie jest całkowita. Liczby nie będące liczbami całkowitymi nie mogą zostać przekonwertowane na system rzymski.
4. Pozostała część funkcji jest niezmieniona.

Przykład 14.7. Obserwujemy, jak `toRoman` radzi sobie z błędnym wejściem

```
>>> import roman3
>>> roman3.toRoman(4000)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 27, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
>>> roman3.toRoman(1.5)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 29, in toRoman
    raise NotIntegerError, "non-integers can not be converted"
NotIntegerError: non-integers can not be converted
```

Przykład 14.8. Wyjście `romantest3.py` w zależności od `roman3.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok #(1)
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... ok        #(2)
toRoman should fail with negative input ... ok           #(3)
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

1. `toRoman` dalej przechodzi testy o znanych wartościach, co jest pocieszające. Ponadto przechodzi wszystkie testy, które przechodził w etapie 2, zatem ostatni kod niczego nie popsuł.

2. Bardziej ekscytujący jest fakty, że teraz nasz program przechodzi wszystkie testy z niepoprawnym wejściem. Przechodzi ten test (czyli `testNonInteger`), ponieważ kontrolujemy, czy `int(n)` <> `n`. Kiedy do funkcji `toRoman` zostanie przekazana wartość nie będąca liczbą całkowitą, porównanie `int(n)` <> `n` wyłapie to i wyrzuci wyjątek `NotIntegerError`, a tego oczekuje test `testNonInteger`.

3. Program przechodzi ten test (test `testNegative`), ponieważ w przypadku prawdziwości wyrażenia `not (0 < n < 4000)` zostanie wyrzucony wyjątek `OutOfRangeError`, a którego oczekuje test `testNegative`.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 156, in testFromRomanCase
    roman3.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 133,
in testMalformedAntecedent
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 127, in testRepeatedPairs
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 122,
in testTooManyRepeatedNumerals
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
```

```
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
----------------------------------------------------------------------
Ran 12 tests in 0.401s

FAILED (failures=6) #(1)
```

1. Teraz liczba niezaliczonych testów zmniejszyła się do 6 i wszystkie je powoduje `fromRoman`, czyli: test znanych wartości, trzy testy dotyczące niepoprawnych argumentów, kontrola wielkości znaków i kontrola zdroworozsądkowa (czyli `fromRoman(toRoman(n))==n`). Oznacza to, że `toRoman` przeszedł wszystkie testy, które mógł przejść samemu. (Nawala w teście zdroworozsądkowym, ale test ten wymaga także napisania funkcji `fromRoman`, a to jeszcze nie zostało zrobione.) Oznacza to, że musimy przestać już kodować `toRoman`. Już nie ulepszamy, nie kombinujemy, bez ekstra "a może ten". Stop. Teraz odejdziemy od klawiatury.

Jedną z najistotniejszych spraw jest to, że rozumowy unit testing mówi tobie, kiedy przestać kodować. Kiedy funkcja przechodzi wszystkie unit testy przeznaczone dla niej, kończymy kodować tę funkcję. Kiedy wszystkie unit test dla całego modułu zostaną zaliczone, przestajemy kodować moduł.

15.4 roman.py, etap 4

Implementacja funkcji `toRoman` została zakończona, czas zająć się funkcją `fromRoman`. Dzięki bogatej strukturze danych przechowującej pewne wartości w reprezentacji rzymskiej wraz z ich wartościami liczbowymi, zadanie to nie będzie wcale trudniejsze, niż napisanie funkcji `toRoman`.

Przykład 14.9. roman4.py

Plik jest dostępny w katalogu in py/roman/stage4/ wewnątrz katalogu examples.

Jeśli jeszcze tego nie zrobiliście, możecie pobrać ten oraz inne przykłady używane w tej książce stąd.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

# toRoman function omitted for clarity (it hasn't changed)

def fromRoman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral: #(1)
            result += integer
            index += len(numeral)
    return result
```

1. Sposób działania jest taki sam jak w `toRoman`. Iterujemy po reprezentacjach rzymskich w strukturze danych (będącej krotką krotek), jednak zamiast dopasowywania największej wartości całkowitej tak często, jak to możliwe, dopasowujemy “najwyższą” reprezentację rzymską tak często, jak to możliwe.

Przykład 14.10. Jak działa `fromRoman`
Jeśli wciąż nie jesteście pewni, jak działa `fromRoman`, na końcu pętli `while` dodajcie instrukcję `print`:

```
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print 'found', numeral, 'of length', len(numeral), ', adding', integer
```

```
>>> import roman4
>>> roman4.fromRoman('MCMLXXII')
found M , of length 1, adding 1000
found CM , of length 2, adding 900
found L , of length 1, adding 50
found X , of length 1, adding 10
found X , of length 1, adding 10
found I , of length 1, adding 1
found I , of length 1, adding 1
1972
```

Przykład 14.11. Wyjście programu romantest4.py testującego roman4.py

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... ok #(1)
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok                #(2)
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

1. Mamy tu dwie interesujące wiadomosci: po pierwsze, `fromRoman` działa dla poprawnych danych wejściowych, przynajmniej dla tych, które są zdefiniowane w teście poprawnych wartości.

2. Po drugie, test zdroworozsądkowy również przeszedł. Wiedząc o tym, że przeszedł również test znanych wartości, możemy być raczej pewni, że zarówno `fromRoman` jak i `toRoman` działają poprawnie dla poprawnych danych wejściowych. (Nic nam tego jednak nie gwarantuje; teoretycznie jest możliwe, że w funkcji `toRoman` ukryty jest jakiś błąd, przez który dla pewnego zestawu danych wejściowych generowane są niepoprawne reprezentacje rzymskie, natomaist `fromRoman` zawierać może symetryczny błąd, który z kolei powoduje, że dla tych właśnie rzymskich reprezentacji generowane są niepoprawne wartości liczbowe. W zależności od zastosowań waszego kodu, a także wymagań, jakim ten kod podlega, może to stanowić dla was pewien problem; jeśli tak jest, dopiszcie więcej bardziej wszechstronnych testów tak, aby zmniejszyła się wasza niepewność.)

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 156, in testFromRomanCase
    roman4.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 133,
in testMalformedAntecedent
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 127, in testRepeatedPairs
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 122,
in testTooManyRepeatedNumerals
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 12 tests in 1.222s

FAILED (failures=4)
```

15.5 roman.py, etap 5

Funkcja `fromRoman` działa poprawnie dla poprawnych danych wejściowych, nadszedł więc czas na dołożenie ostatniego klocka w naszej układance: napisanie kodu, dzięki któremu funkcja ta będzie działała poprawnie również dla niepoprawnych danych wejściowych. Oznacza to, że musimy znaleźć sposób na ustalenie, czy dany napis stanowi poprawną rzymską reprezentację pewnej wartości. To zadanie jest znacznie trudniejsze, niż sprawdzenie poprawności wartości liczbowej w funkcji `toRoman`, jednak możemy do tego celu użyć silnego narzędzia: wyrażeń regularnych.

Jeśli nie znacie wyrażeń regularnych i nie przeczytaliście jeszcze podrozdziału 7 Wyrażenia regularne, nadszedł właśnie doskonały moment, aby to zrobić.

Jak widzieliśmy w podrozdziale 7.3 Analiza przypadku: Liczby rzymskie, istnieje kilka prostych reguł, dzięki którym można skonstruować napis reprezentujący wartość liczbową w zapisie rzymskim, używając liter M, D, C, L, X, V oraz I. Prześledźmy je po kolei:

1. Znaki można dodawać. I to 1, II to 2, III to 3. VI to 6 (dosłownie: "5 i 1"), VII to 7, a VIII to 8.

2. Liczby składające się z jedynki i (być może) zer (I, X, C oraz M) — liczby "dziesiątkowe" — mogą być powtarzane do trzech razy. Przy czwartym należy odjąć tę wartość od znaku reprezentującego liczbę składającą się z piątki i (być może) zer — liczbę "piątkową". Nie można przedstawić liczby 4 jako IIII, należy przedstawić ją jako IV ("1 odjęte od 5"). Liczbę 40 zapisujemy jako XL ("10 odjęte od 50"), 41 jako XLI, 42 jako XLII, 43 jako XLIII, a 44 jako XLIV ("10 odjęte od 50 oraz 1 odjęte od 5").

3. Podobnie tworzymy liczby "dziewiątkowe": należy odejmować od najbliższej liczby dziesiątkowej: 8 to VIII, jednak 9 to IX ("1 odjęte od 10"), a nie VIIII (ponieważ I nie może być powtórzone więcej niż trzy razy), zaś 90 to XC, a 900 to CM.

4. Liczby "piątkowe" nie mogą być powtarzane. 10 zawsze reprezentowane jest jako X, a nie VV, 100 jako C, nigdy zaś jako LL.

5. Liczby w reprezentacji rzymskiej są zawsze zapisywane od największych do najmniejszych i odczytywane od lewej do prawej, a więc porządek znaków ma ogromne znaczenie. DC to 600; CD to zupełnie inna liczba (400, "100 odjęte od 500"). CI to 101, IC zaś nie jest poprawną wartością w zapisie rzymskim, ponieważ nie można bezpośrednio odjąć 1 od 100: należałoby zapisać XCIX ("10 odjęte od 100 oraz 1 odjęte od 10").

Przykład 14.12. roman5.py

Plik jest dostępny w katalogu in py/roman/stage5/ wewnątrz katalogu examples.

Jeśli jeszcze tego nie zrobiliście, możecie pobrać ten oraz inne przykłady używane w tej książce stąd.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
```

```
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError, "number out of range (must be 1..3999)"
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' #(1)

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not re.search(romanNumeralPattern, s):                                    #(2)
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

1. To kontynuacja wyrażenia, o którym dyskutowaliśmy w podrozdziale 7.3 Ana-

liza przypadku: Liczby rzymskie. Miejsce “dziesiątki” jest w napisie XC (90), XL (40) oraz w napisie złożonym z opcjonalnego L oraz następującym po niej opcjonalnym znaku X powtórzonym od 0 do 3 razy. Miejsce “jedynki” jest w napisie IX (9), IV (4) oraz przy opcjonalnym V z następującym po niej, opcjonalnym znakiem I powtórzonym od 0 do 3 razy.

2. Po wpisaniu tej logiki w wyrażenie regularne otrzymamy trywialny kod sprawdzający poprawność napisów potencjalnie reprezentujących liczby rzymskie. Jeśli `re.search` zwróci obiekt, wyrażenie regularne zostało dopasowane, a więc dane wejściowe są poprawne; w przeciwnym wypadku, dane wejściowe są niepoprawne.

W tym momencie macie prawo być nieufni wobec tego wielkiego, brzydkiego wyrażenia regularnego, które ma się rzekomo dopasować do wszystkich poprawnych napisów reprezentujących liczby rzymskie. Oczywiście, nie musicie mi wierzyć, spójrzcie zatem na wyniki testów:

Example 14.13. Output of romantest5.py against roman5.py

```
fromRoman should only accept uppercase input ... ok          #(1)
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... ok      #(2)
fromRoman should fail with repeated pairs of numerals ... ok #(3)
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 2.864s

OK                                                           #(4)
```

1. Jedna rzecz o jakiej nie wspomniałem w kontekście wyrażeń regularnych to fakt, że są one zależne od wielkości znaków. Ze względu na to, że wyrażenie regularne `romanNumeralPattern` zostało zapisane przy użyciu wielkich liter, sprawdzenie `re.search` odrzuci wszystkie napisy, które zawierają przynajmniej jedną małą literę. Dlatego też test wielkich liter przechodzi.

2. Co więcej, przechodzą również testy nieprawidłowych danych wejściowych. Przykładowo test niepoprawnych poprzedników sprawdza przypadki takie, jak MCMC. Jak widzimy, wyrażenie regularne nie pasuje do tego napisu, a więc `fromRoman` rzuca wyjątek `InvalidRomanNumeralError`, i jest to dokładnie taki wyjątek, jakiego spodziewa się test niepoprawnych poprzedników, a więc test ten przechodzi.

3. Rzeczywiście przechodzą wszystkie testy sprawdzające niepoprawne dane wejściowe. Wyrażenie regularne wyłapuje wszystkie przypadki, o jakich myśleliśmy podczas przygotowywania naszych przypadków testowych.

4. Nagrodę największego rozczarowania roku otrzymuje słówko “OK”, które zostało wypisane przez moduł `unittest` w chwili gdy okazało się, że wszystkie testy zakończyły się powodzeniem.

Kiedy wszystkie testy przechodzą, przestań kodować.

Rozdział 16

Refaktoryzacja

16.1 Obsługa błędów

Mimo wielkiego wysiłku wkładanego w pisanie testów jednostkowych błędy wciąż się zdarzają. Co mam na myśli pisząc "błąd"? Błąd to przypadek testowy, który nie został jeszcze napisany.

Przykład 15.1. Błąd

```
>>> import roman5
>>> roman5.fromRoman("")                                                  #(1)
0
```

1. Czy pamiętasz poprzedni rozdział, w którym okazało się, że pusty napis pasuje do wyrażenia regularnego używanego do sprawdzania poprawności liczb rzymskich? Otóż okazuje się, że jest to prawdą nawet w ostatecznej wersji wyrażenia regularnego. I to jest właśnie błąd; pożądanym rezultatem przekazania pustego napisu, podobnie jak każdego innego napisu, który nie reprezentuje poprawnej liczby rzymskiej, jest rzucenie wyjątku InvalidRomanNumeralError.

Po udanym odtworzeniu błędu, ale przed jego naprawieniem, powinno się napisać przypadek testowy, który nie działa, uwidaczniając w ten sposób znaleziony błąd.

Przykład 15.2. Testowanie błędu (romantest61.py)

```
class FromRomanBadInput(unittest.TestCase):
    # previous test cases omitted for clarity (they haven't changed)
    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, "")  #(1)
```

1. Sprawa jest prosta. Wywołujemy fromRoman z pustym napisem i upewniamy się, że został rzucony wyjątek InvalidRomanNumeralError. Najtrudniejszą częścią było znalezienie błędu; teraz, kiedy już o nim wiemy, testowanie okazuje się łatwe.

Mamy już odpowiedni przypadek testowy, jednak nie będzie on działał, ponieważ w kodzie wciąż jest błąd:

Przykład 15.3. Wyjście programu romantest61.py testującego roman61.py

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... FAIL
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

```
======================================================================
FAIL: fromRoman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage6\romantest61.py", line 137, in testBlank
    self.assertRaises(roman61.InvalidRomanNumeralError, roman61.fromRoman, "")
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 13 tests in 2.864s

FAILED (failures=1)
```

Teraz możemy przystąpić do naprawy błędu.

Przykład 15.4. Poprawiane błędu (roman62.py)

Plik jest dostępny w katalogu py/roman/stage6/ znajdującym się w katalogu z przykładami.

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:                                                           #(1)
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

1. Potrzebne są tylko dwie dodatkowe linie kodu: jawne sprawdzenie pustego napisu oraz wyrażenie raise.

Przykład 15.5. Wyjście programu romantest62.py testującego roman62.py

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok                          #(1)
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
```

```
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 13 tests in 2.834s

OK                                                                    #(2)
```

1. Test pustego napisu przechodzi, a więc błąd udało się naprawić.
2. Wszystkie pozostałe testy przechodzą, co oznacza, że poprawka błędu nie zepsuła kodu w innych miejscach. Koniec kodowania.

Ten sposób kodowania nie sprawi, że znajdowanie błędów stanie się łatwiejsze. Proste błędy (takie, jak ten w przykładzie) wymagają prostych testów jednostkowych; błędy bardziej złożone będą wymagały testów odpowiednio bardziej złożonych. W środowisku, w którym na testowanie kładzie się duży nacisk, może się początkowo wydawać, że poprawienie błędu zabiera znacznie więcej czasu: najpierw należy dokładnie wyrazić w kodzie, na czym polega błąd (czyli napisać przypadek testowy), a później dopiero go poprawić. Następnie, jeśli przypadek testowy nie przechodzi, należy sprawdzić, czy to poprawka była niewystarczająca, czy może kod przypadku testowego został niepoprawnie zaimplementowany. Jednak w długiej perspektywie takie przełączanie się między kodem i testami niewątpliwie się opłaca, ponieważ poprawienie błędu za pierwszym razem jest o wiele bardziej prawdopodobne. Dodatkowo, możliwość uruchomienia wszystkich testów łącznie z dopisanym nowym przypadkiem testowym pozwala łatwo sprawdzić, czy poprawka błędu nie spowodowała problemów w starym kodzie. Dzisiejszy test jednostkowy staje się więc jutrzejszym testem regresyjnym.

16.2 Obsługa zmieniających się wymagań

Choćbyśmy próbowali przyszpilić swoich klientów do ziemi w celu uzyskania od nich dokładnych wymagań, używając tak przerażających narzędzi tortur, jak nożyce czy gorący wosk, to i tak te wymagania się zmienią. Większość klientów nie wie, czego chce, dopóki tego nie zobaczy, a nawet jak już zobaczy, to nie jest w stanie wyartykułować tego wystarczająco precyzyjnie, aby było to użyteczne. Nawet, gdyby im się to udało, to zapewne i tak w kolejnym wydaniu będą chcieli czegoś więcej. Tak więc lepiej bądźmy przygotowani na aktualizowanie swoich przypadków testowych w miarę jak zmieniają się wymagania.

Przypuśćmy, na przykład, że chcieliśmy rozszerzyć zakres funkcji konwertujących liczby rzymskie. Czy pamiętacie regułę, która mówi, że żadna litera nie może być powtórzona więcej niż trzy razy? Otóż Rzymianie chcieli uczynić wyjątek od tej reguły tak, aby móc reprezentować wartość 4000 stawiając obok siebie cztery litery M. Jeśli wprowadzimy tę zmianę, będziemy mogli rozszerzyć zakres liczb możliwych do przekształcenia na liczbę rzymską z 1..3999 do 1..4999. Najpierw jednak musimy wprowadzić kilka zmian do przypadków testowych.

Przykład 15.6. Zmiana przypadków testowych przy nowych wymaganiach (romantest71.py)

Plik jest dostępny w katalogu py/roman/stage7/ znajdującym się w katalogu examples.

Jeśli jeszcze tego nie zrobiliście, ściągnijcie ten oraz inne przykłady (`http://diveintopython.org/download/diveintopython-examples-5.4.zip`) używane w tej książce.

```
import roman71
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
```

```
                        (782, 'DCCLXXXII'),
                        (870, 'DCCCLXX'),
                        (941, 'CMXLI'),
                        (1043, 'MXLIII'),
                        (1110, 'MCX'),
                        (1226, 'MCCXXVI'),
                        (1301, 'MCCCI'),
                        (1485, 'MCDLXXXV'),
                        (1509, 'MDIX'),
                        (1607, 'MDCVII'),
                        (1754, 'MDCCLIV'),
                        (1832, 'MDCCCXXXII'),
                        (1993, 'MCMXCIII'),
                        (2074, 'MMLXXIV'),
                        (2152, 'MMCLII'),
                        (2212, 'MMCCXII'),
                        (2343, 'MMCCCXLIII'),
                        (2499, 'MMCDXCIX'),
                        (2574, 'MMDLXXIV'),
                        (2646, 'MMDCXLVI'),
                        (2723, 'MMDCCXXIII'),
                        (2892, 'MMDCCCXCII'),
                        (2975, 'MMCMLXXV'),
                        (3051, 'MMMLI'),
                        (3185, 'MMMCLXXXV'),
                        (3250, 'MMMCCL'),
                        (3313, 'MMMCCCXIII'),
                        (3408, 'MMMCDVIII'),
                        (3501, 'MMMDI'),
                        (3610, 'MMMDCX'),
                        (3743, 'MMMDCCXLIII'),
                        (3844, 'MMMDCCCXLIV'),
                        (3888, 'MMMDCCCLXXXVIII'),
                        (3940, 'MMMCMXL'),
                        (3999, 'MMMCMXCIX'),
                        (4000, 'MMMM'),
                        (4500, 'MMMMD'),
                        (4888, 'MMMMDCCCLXXXVIII'),
                        (4999, 'MMMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.fromRoman(numeral)
```

```
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 5000)                #(2)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman71.NotIntegerError, roman71.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):                    #(3)
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, "")

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 5000):                                                   #(4)
            numeral = roman71.toRoman(integer)
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
```

```
        """toRoman should always return uppercase"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            roman71.fromRoman(numeral.upper())
            self.assertRaises(roman71.InvalidRomanNumeralError,
                              roman71.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

1. Istniejące wartości nie zmieniają się (to wciąż rozsądne wartości do przetestowania), jednak musimy dodać kilka do poszerzonego zakresu. Powyżej dodałem 4000 (najkrótszy napis), 4500 (drugi najkrótszy), 4888 (najdłuższy) oraz 4999 (największy co do wartości).

2. Zmieniła się definicja "dużych danych wejściowych". Ten test miał nie przechodzić dla wartości 4000 i zgłaszać w takiej sytuacji błąd; teraz wartości z przedziału 4000-4999 są poprawne, a jako pierwszą niepoprawną wartość należy przyjąć 5000.

3. Zmieniła się definicja "zbyt wielu powtórzonych cyfr rzymskich". Ten test wywoływał fromRoman z wartością 'MMMM' i spodziewał się błędu. Obecnie MMMM jest poprawną liczbą rzymską, a więc należy zmienić niepoprawną wartość na 'MMMMM'.

4. Testy prostego sprawdzenia i sprawdzenia wielkości liter iterują po wartościach z przedziału od 1 do 3999. Ze względu na poszerzenie tego przedziału rozszerzamy też pętle w testach tak, aby uwzględniały wartości do 4999.

Teraz przypadki testowe odzwierciedlają już nowe wymagania, jednak nie uwzględnia ich jeszcze kod, a więc można się spodziewać, że pewne testy nie przejdą:

Przykład 15.7. Wyjście programu romantest71.py testującego roman71.py

```
fromRoman should only accept uppercase input ... ERROR            #(1)
toRoman should always return uppercase ... ERROR
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ERROR     #(2)
toRoman should give known result with known input ... ERROR       #(3)
fromRoman(toRoman(n))==n for all n ... ERROR                      #(4)
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

1. Test sprawdzający wielkość liter nie przechodzi, ponieważ pętla uwzględnia wartości od 1 do 4999, natomiast toRoman akceptuje wartości z przedziału od 1 do 3999. Jak tylko licznik pętli osiągnie wartość 4000, test nie przechodzi.
2. Test poprawnych wartości używający toRoman nie przechodzi dla napisu 'MMMM', ponieważ toRoman wciąż sądzi, że jest to wartość niepoprawna.
3. Test poprawnych wartości używający toRoman nie przechodzi dla wartości 4000, ponieważ toRoman wciąż sądzi, że jest to wartość spoza zakresu.
4. Test poprawności również nie przechodzi dla wartości 4000, ponieważ toRoman wciąż sądzi, że jest to wartość spoza zakresu.

```
======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 161, in testFromRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 155, in testToRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 102, in testFromRomanKnownValues
    result = roman71.fromRoman(numeral)
  File "roman71.py", line 47, in fromRoman
    raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
InvalidRomanNumeralError: Invalid Roman numeral: MMMM
======================================================================
ERROR: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 96, in testToRomanKnownValues
    result = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman(toRoman(n))==n for all n
```

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 147, in testSanity
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
----------------------------------------------------------------------
Ran 13 tests in 2.213s

FAILED (errors=5)
```

Kiedy już mamy przypadki testowe, które ze względu na nowe wymagania przestały przechodzić, możemy myśleć o poprawieniu kodu tak, aby był zgodny z testami (kiedy zaczyna się pisać testy jednostkowe, należy się przyzwyczaić do jednej rzeczy: testowany kod nigdy nie "wyprzedza" przypadków testowych. Zdarza się, że kod "nie nadąża", co oznacza, że wciąż jest coś do zrobienia, przy czym jak tylko kod "dogoni" testy, zadanie jest już wykonane).

Przykład 15.8. Implementacja nowych wymagań (roman72.py)

Plik jest umieszczony w katalogu py/roman/stage7/ znajdującym się w katalogu examples.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 5000):
        raise OutOfRangeError, "number out of range (must be 1..4999)"
```

```
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'        #(2)

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

1. toRoman wymaga jednej małej zmiany w sprawdzeniu zakresu. Tam, gdzie było sprawdzenie $0 < n < 4000$, powinno być $0 < n < 5000$. Należy też zmienić treść komunikatu błędu, tak, aby odzwierciedlał on nowy, akceptowalny zakres wartości (1..4999 zamiast 1..3999). Nie trzeba dokonywać żadnych innych zmian w kodzie funkcji, który już obsługuje nowe wymaganie. (Kod dodaje 'M' dla każdego pełnego tysiąca; gdy na wejściu jest wartość 4000, otrzymamy liczbę rzymską 'MMMM'. Jedynym powodem tego, że funkcja nie działała tak wcześniej było jej jawne zakończenie przy wartości przekraczającej dopuszczalny zakres.

2. Nie trzeba w ogóle zmieniać fromRoman. Jedyna wymagana zmiana dotyczy wzorca romanNumeralPattern; po bliższym przyjrzeniu się widać, że wystarczyło dodać jeszcze jedno opcjonalne 'M' w pierwszej części wyrażenia regularnego. Pozwoli to na dopasowanie maksymalnie czterech znaków M zamiast trzech, a więc uwzględni wartości rzymskie z przedziału do 4999 zamiast do 3999. Obecna implementacja fromRoman jest bardzo ogólna: wyszukuje ona powtarzające się znaki w zapisie rzymskim, a następnie sumuje ich odpowiednie wartości, nie przejmując się liczbą ich wystąpień. Funkcja ta nie obsługiwała wcześniej napisu 'MMMM' wyłącznie dlatego, że napis taki nie zostałby wcześniej dopasowany do wyrażenia regularnego.

Możecie być odrobinę sceptyczni wobec stwierdzenia, że te dwie małe zmiany to wszystko, czego potrzebujemy. Nie wierzcie mi na słowo, po prostu to sprawdźcie:

Przykład 15.9. Wyjście programu romantest72.py testującego roman72.py

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 13 tests in 3.685s

OK                                                                    #(
```

1. Wszytskie testy przechodzą. Kończymy kodowanie.

Pełne testowanie jednostkowe oznacza, że nigdy nie trzeba polegać na słowach programisty mówiącego: “Zaufaj mi”.

16.3 Refaktoryzacja

Najcenniejszą rzeczą, jaką daje testowanie jednostkowe, nie jest uczucie, jakiego doświadczamy, kiedy wszystkie testy przechodzą, ani nawet uczucie w chwili, gdy ktoś obwinia nas o popsucie swojego kodu, a my jesteśmy w stanie udowodnić, że to nie nasza wina. Najcenniejszą rzeczą w testowaniu jednostkowym jest to, że daje nam ono nieskrępowaną wolność podczas refaktoryzacji.

Refaktoryzacja to proces, który polega na tym, że bierze się działający kod i zmienia go tak, aby działał jeszcze lepiej. Zwykle "lepiej" znaczy "szybciej", choć może to również znaczyć "przy mniejszym zużyciu pamięci", "przy mniejszym zużyciu przestrzeni dyskowej" czy nawet "bardziej elegancko". Czymkolwiek refaktoryzacja jest dla was, dla waszego projektu czy waszego środowiska pracy, służy ona utrzymaniu programu w dobrym zdrowiu przez długi czas.

W naszym przypadku "lepiej" znaczy "szybciej". Dokładnie rzecz ujmując, funkcja `fromRoman` jest wolniejsza niż musiałaby być ze względu na duże, brzydkie wyrażenie regularne, którego używamy do zweryfikowania, czy napis stanowi poprawną reprezentację liczby w notacji rzymskiej. Prawdopodobnie nie opłaca się całkowicie eliminować tego wyrażenia (byłoby to trudne i mogłoby doprowadzić do powstania jeszcze wolniejszego kodu), jednak można uzyskać pewne przyspieszenie dzięki temu, że wyrażenie regularne zostanie wstępnie skompilowane.

Przykład 15.10. Kompilacja wyrażenia regularnego

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')                                     #(1)
<SRE_Match object at 01090490>
>>> compiledPattern = re.compile(pattern)                       #(2)
>>> compiledPattern
<SRE_Pattern object at 00F06E28>
>>> dir(compiledPattern)                                        #(3)
['findall', 'match', 'scanner', 'search', 'split', 'sub', 'subn']
>>> compiledPattern.search('M')                                 #(4)
<SRE_Match object at 01104928>
```

1. To składnia, którą już wcześniej widzieliście: `re.search` pobiera wyrażenie regularne jako napis (`pattern`) oraz napis, do którego wyrażenie będzie dopasowywane ('M'). Jeśli wyrażenie zostanie dopasowane, funkcja zwróci obiekt `match`, który można następnie odpytać, aby dowiedzieć się, co zostało dopasowane i w jaki sposób.

2. To jest już nowa składnia: `re.compile` pobiera wyrażenie regularne jako napis i zwraca obiekt `pattern`. Zauważmy, że nie przekazujemy napisu, do którego będzie dopasowywane wyrażenie. Kompilacja wyrażenia regularnego nie ma nic wspólnego z dopasowywaniem wyrażenia do konkretnego napisu (jak np. 'M'); dotyczy ona wyłącznie samego wyrażenia.

3. Obiekt `pattern` zwrócony przez funkcję `re.compile` posiada wiele pożytecznie wyglądających funkcji, między innymi kilka takich, które są dostępne bezpośrednio w module `re` (np. `search` czy `sub`).

4. Wywołując funkcji `search` na obiekcie `pattern` z napisem 'M' jako parametrem osiągamy ten sam efekt, co wywołując `re.search` z wyrażeniem regularnym i

napisem 'M' jako parametrami. Z tą różnicą, że osiągamy go o wiele, wiele szybciej. (W rzeczywistości funkcja `re.search` kompiluje wyrażenie regularne i na obiekcie będącym wynikiem tej kompilacji wywołuje metodę `search`.)

Kompilacja wyrażeń regularnych. Jeśli kiedykolwiek będziecie potrzebowali użyć wyrażenia regularnego więcej niż raz, powinniście najpierw skompilować je do obiektu `pattern`, a następnie wywoływać bezpośrednio jego metody

Przykład 15.11.Skompilowane wyrażenie regularne w roman81.py

Plik jest dostępny w katalogu in py/roman/stage8/ wewnątrz katalogu examples.

Jeśli jeszcze tego nie zrobiliście, możecie pobrać ten oraz inne przykłady używane w tej książce stąd.

```
# toRoman and rest of module omitted for clarity

romanNumeralPattern = \
    re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$') (1)

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not romanNumeralPattern.search(s):                                    (2)
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

1. Wygląda podobnie, choć w rzeczywistości bardzo dużo się zmieniło. `romanNumeralPattern` nie jest już napisem; to obiekt `pattern`, który został zwrócony przez `re.compile`.
2. Ta linia oznacza, że na obiekcie `romanNumeralPattern` można bezpośrednio wywoływać metody. Będą one wykonane o wiele szybciej, niż np. podczas każdorazowego wywołania `re.search`. Tutaj wyrażenie regularne zostało skompilowane dokładnie jeden raz i zapamiętane pod nazwą `romanNumeralPattern` w momencie pierwszego importu modułu; od tego momentu. ilekroć będzie wywołana metoda `fromRoman`, gotowe wyrażenie będzie dopasowywane do napisu wejściowego, bez żadnych kroków pośrednich odbywających się niejawnie.

Wobec tego o ile szybciej działa kod po skompilowaniu wyrażenia regularnego? Sprawdźcie sami:

Przykład 15.12. Wyjście programu romantest81.py testującego roman81.py

```
.............                                                               #(1)
----------------------------------------------------------------------
```

```
Ran 13 tests in 3.385s                                                                        #(2)

OK                                                                                            #(3)
```

1. Tutaj tylko mała uwaga: tym razem uruchomiłem testy bez podawania opcji -v, dlatego też zamiast pełnego napisu komentującego dla każdego testu, który zakończył się powodzeniem, została wypisana kropka. (Gdyby test zakończył się niepowodzeniem, zostałaby wypisana litera F, a w przypadku błędu — litera E. Potencjalne problemy można wciąż łatwo zidentyfikować, ponieważ w razie niepowodzeń lub błędów wypisywana jest zawartość stosu.)

2. Uruchomienie 13 testów zajęło 3.385 sekund w porównaniu z 3.685 sekund, jakie zajęły testy bez wcześniejszej kompilacji wyrażenia regularnego. To poprawa w wysokości 8%, a warto pamiętać, że przez większość czasu w testach jednostkowych wykonywane są także inne rzeczy. (Gdy przetestowałem same wyrażenia regularne, niezależnie od innych testów, okazało się, że kompilacja wyrażenia regularnego polepszyła czas operacji wyszukiwania średnio o 54%.) Nieźle, jak na taką niewielką poprawkę.

3. Och, gdybyście się jeszcze zastanawiali, prekompilacja wyrażenia regularnego niczego nie zepsuła, co właśnie udowodniliśmy.

Jest jeszcze jedna optymalizacja wydajności, którą chciałem wypróbować. Nie powinno być niespodzianką, że przy wysokiej złożoności składni wyrażeń regularnych istnieje więcej niż jeden sposób napisania tego samego wyrażenia. W trakcie dyskusji nad tym rozdziałem, jaka odbyła się na comp.lang.python ktoś zasugerował, żebym dla powtarzających się opcjonalnych znaków spróbował użyć składni {m, n}.

Przykład 15.13. roman82.py

Plik jest dostępny w katalogu in py/roman/stage8/ wewnątrz katalogu examples.

Jeśli jeszcze tego nie zrobiliście, możecie pobrać ten oraz inne przykłady używane w tej książce stąd.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#   re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$')

#new version
romanNumeralPattern = \
    re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')          #(1)
```

1. Zastąpiliśmy M?M?M?M? wyrażeniem M{0,4}. Obydwa zapisy oznaczają to samo: “dopasuj od 0 do 4 znaków M”. Podobnie C?C?C? zostało zastąpione C{0,3} (“dopasuj od 0 do 3 znaków C”) i tak dalej dla X oraz I.

Powyższy zapis wyrażenia regularnego jest odrobinę krótszy (choć nie bardziej czytelny). Pytanie brzmi: czy jest on szybszy?

Przykład 15.14. Wyjście programu romantest82.py testującego roman82.py

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s                                                #(1)

OK                                                                    #(2)
```

1. Przy tej formie wyrażenia regularnego testy jednostkowe działały w sumie 2% szybciej. Nie brzmi to może zbyt ekscytująco, dlatego przypomnę, że wywołania funkcji wyszukującej stanowią niewielką część wszystkich testów; przez większość czasu testy robią co innego. (Gdy niezależnie od innych testów przetestowałem wyłącznie wydajność wyrażenia regularnego, okazało się, że jest ona o 11% większa przy nowej składni). Dzięki prekompilacji wyrażenia regularnego i zmianie jego składni udało się poprawić wydajność samego wyrażenia o ponad 60%, a wszystkich testów łącznie o ponad 10%.

2. Znacznie ważniejsze od samego wzrostu wydajności jest to, że moduł wciąż doskonale działa. To jest właśnie wolność, o której wspominałem już wcześniej: wolność poprawiania, zmieniania i przepisywania dowolnego fragmentu kodu i możliwość sprawdzenia, że zmiany te w międzyczasie wszystkiego nie popsuły. Nie chodzi tu o poprawki dla samych poprawek; mieliśmy bardzo konkretny cel ("przyspieszyć `toRoman`") i byliśmy w stanie go zrealizować bez zbytnich wahań i troski o to, czy nie wprowadziliśmy do kodu nowych błędów.

Chcę zrobić jeszcze jedną, ostatnią zmianę i obiecuję, że na niej skończę refaktoryzację i dam już temu modułowi spokój. Jak wielokrotnie widzieliście, wyrażenia regularne szybko stają się bardzo nieporządne i mocno tracą na swej czytelności. Naprawdę nie chciałbym dostać tego modułu do utrzymania za sześć miesięcy. Oczywiście, testy przechodzą, więc mam pewność, że kod działa, jednak jeśli nie jestem całkowicie pewien, w jaki sposób on działa, to będzie mi trudno dodawać do niego nowe wymagania, poprawiać błędy czy w inny sposób go utrzymywać. W podrozdziale idzieliście, że Python umożliwia dokładne udokumentowanie logiki kodu.

Przykład 15.15. roman83.py

Plik jest dostępny w katalogu in py/roman/stage8/ wewnątrz katalogu examples.

Jeśli jeszcze tego nie zrobiliście, możecie pobrać ten oraz inne przykłady używane w tej książce stąd.

```
# rest of program omitted for clarity

  #old version
  #romanNumeralPattern = \
  #   re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')

  #new version
  romanNumeralPattern = re.compile(
'''
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                        #            or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
```

```
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    ''', re.VERBOSE)                                                    #(1)
```

1. Funkcja `re.compile` może przyjmować drugi (opcjonalny) argument, będący zbiorem znaczników kontrolujących wiele aspektów skompilowanego wyrażenia regularnego. Powyżej wyspecyfikowaliśmy znacznik `re.VERBOSE`, który podpowiada kompilatorowi języka Python, że wewnątrz wyrażenia regularnego znajdują się komentarze. Komentarze te wraz z białymi znakami nie stanowią części wyrażenia regularnego; funkcja `re.compile` nie bierze ich pod uwagę podczas kompilacji. Dzięki temu zapisowi, choć samo wyrażenie jest identyczne jak poprzednio, jest ono niewątpliwie znacznie bardziej czytelne.

Przykład 15.16. Wyjście z programu romantest83.py testującego roman83.py

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s                                                 #(1)

OK                                                                     #(2)
```

1. Nowa, "rozwlekła" wersja wyrażenia regularnego działa dokładnie tak samo, jak wersja poprzednia. Rzeczywiście, skompilowane obiekty wyrażeń regularnych będą identyczne, ponieważ `re.compile` wyrzuca z wyrażenia dodatkowe znaki, które umieściliśmy tam w charakterze komentarza.

2. Nowa, "rozwlekła" wersja wyrażenia regularnego przechodzi wszystkie testy, tak jak wersja poprzednia. Nie zmieniło się nic oprócz tego, że programista, który wróci do kodu po sześciu miesiącach będzie miał możliwość zrozumienia, w jaki sposób działa wyrażenie regularne.

16.4 Postscript

Sprytny czytelnik po przeczytaniu poprzedniego podrozdziału byłby w stanie jeszcze bardziej polepszyć kod programu. Największym bowiem problemem (i dziurą (?) wydajnościową) programu w obecnym kształcie jest wyrażenie regularne, wymagane ze względu na to, że nie ma innego sensownego sposobu weryfikacji poprawności liczby w zapisie rzymskim. Tych liczb jest jednak tylko 5000; dlaczego by więc nie zbudować tablicy przeszukiwania tylko raz, a później po prostu z niej korzystać? Ten pomysł wyda się jeszcze lepszy, gdy zdamy sobie sprawę, że w ogóle nie musimy korzystać z wyrażeń regularnych. Skoro można zbudować tablicę przeszukiwań służącą do konwersji wartości liczbowych w ich rzymską reprezentację, to można również zbudować tablicę odwrotną do przekształcania liczb rzymskich w ich wartość liczbową.

Najlepsze zaś jest to, że ów sprytny czytelnik miałby już do swojej dyspozycji pełen zestaw testów jednostkowych. Choć zmodyfikowałby połowę kodu w module, to testy pozostałyby takie same, a więc mógłby on udowodnić, że kod po zmianach działa tak samo, jak wcześniej.

Przykład 15.17. roman9.py

Plik jest dostępny w katalogu in py/roman/stage9/ wewnątrz katalogu examples.

Jeśli jeszcze tego nie zrobiliście, możecie pobrać ten oraz inne przykłady używane w tej książce stąd.

```
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Roman numerals must be less than 5000
MAX_ROMAN_NUMERAL = 4999

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

#Create tables for fast conversion of roman numerals.
#See fillLookupTables() below.
toRomanTable = [ None ]  # Skip an index since Roman numerals have no zero
fromRomanTable = {}
```

```
def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n <= MAX_ROMAN_NUMERAL):
        raise OutOfRangeError, "number out of range (must be 1..%s)" % MAX_ROMAN_NUMERAL
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"
    return toRomanTable[n]

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, "Input can not be blank"
    if not fromRomanTable.has_key(s):
        raise InvalidRomanNumeralError, "Invalid Roman numeral: %s" % s
    return fromRomanTable[s]

def toRomanDynamic(n):
    """convert integer to Roman numeral using dynamic programming"""
    result = ""
    for numeral, integer in romanNumeralMap:
        if n >= integer:
            result = numeral
            n -= integer
            break
    if n > 0:
        result += toRomanTable[n]
    return result

def fillLookupTables():
    """compute all the possible roman numerals"""
    #Save the values in two global tables to convert to and from integers.
    for integer in range(1, MAX_ROMAN_NUMERAL + 1):
        romanNumber = toRomanDynamic(integer)
        toRomanTable.append(romanNumber)
        fromRomanTable[romanNumber] = integer

fillLookupTables()
```

A jak szybki jest taki kod?

Przykład 15.18. Wyjście programu romantest9.py testującego roman9.py

```
.............
----------------------------------------------------------------------
Ran 13 tests in 0.791s

OK
```

Pamiętajmy, że najlepszym wynikiem, jaki udało nam się do tej pory uzyskać, był czas 3.315 sekund dla 13 testów. Oczywiście, to porównanie nie jest zbyt uczciwe, ponieważ w tej wersji moduł będzie się dłużej importował (będą generowane tablice

przeszukiwań). Jednak ze względu na to, ze importowanie modułu odbywa się jednokrotnie, czas, jaki ono zajmuje, można uznać za pomijalny.

Morał z tej historii?

- Prostota jest cnotą.
- Szczególnie wtedy, gdy chodzi o wyrażenia regularne.
- A testy jednostkowe dają nam pewność i odwagę do refaktoryzacji w dużej skali... nawet, jeśli to nie my pisaliśmy oryginalny kod.

16.5 Podsumowanie

Testowanie jednostkowe to bardzo silna koncepcja, która, jeśli zostanie poprawnie wdrożona, może zarówno zredukować koszty utrzymywania, jak i zwiększyć elastyczność każdego trwającego długo projektu informatycznego. Ważne jest, aby zrozumieć, że testowanie jednostkowe nie jest panaceum, Magicznym Rozwiązywaczem Problemów czy srebrną kulą. Napisanie dobrych przypadków testowych jest trudne, a utrzymywanie ich wymaga ogromnej dyscypliny (szczególnie w sytuacjach, gdy klienci żądają natychmiastowych poprawek krytycznych błędów). Testowanie jednostkowe nie zastępuje również innych form testowania, takich jak testy funkcjonalne, testy integracyjne czy testy akceptacyjne. Jest jednak wykonalne, działa, a kiedy już zobaczycie je w działaniu, będziecie się zastanawiać, jak w ogóle mogliście bez nich żyć.

W tym rozdziale poruszyliśmy wiele spraw; część z nich nie dotyczyła wyłącznie języka Python. Biblioteki do testowania jednostkowego istnieją dla wielu różnych języków, a ich używanie wymaga jedynie zrozumienia kilku podstawowych koncepcji:

- projektowania przypadków testowych, które są specyficzne, zautomatyzowane i niezależne
- pisania przypadków testowych przed napisaniem testowanego kodu
- pisania przypadków testowych, które uwzględniają poprawne dane wejściowe i spodziewają się poprawnych wyników
- pisania przypadków testowych, które uwzględniają niepoprawne dane wejściowe i spodziewają się odpowiednich niepowodzeń
- pisania i aktualizowania przypadków testowych przedstawiających błędy lub odzwierciedlających nowe wymagania
- bezwzględnej refaktoryzacji w celu poprawy wydajności, skalowalności, czytelności, utrzymywalności oraz każdej innej -ności, która nie jest wystarczająca

Po przeczytaniu tego rozdziału nie powinniście mieć również żadnych problemów z wykonywaniem zadań specyficznych dla języka Python:

- tworzeniem klas pochodnych po `unittest.TestCase` i pisaniem metod będących szczególnymi przypadkami testowymi
- używaniem `assertEqual` do sprawdzania, czy funkcja zwróciła spodziewaną wartość
- używaniem `assertRaises` do sprawdzania, czy funkcja rzuca spodziewany wyjątek
- wywoływaniem `unittest.main()` w klauzuli `if __name__` w celu uruchomienia wszystkich przypadków testowych na raz
- uruchamianiem zestawu testów jednostkowych zarówno w trybie normalnym, jak i rozwlekłym

16.6 Programowanie funkcyjne

Nurkujemy

W rozdziale 13 ("Testowanie") poznaliście filozofię testowania jednostkowego. Rozdział 14 ("Testowanie 2") pozwolił wam zaimplementować podstawowe testy jednostkowe w języku Python. Rozdział 15 ("Refaktoryzacja") uświadomił wam, że dzięki testom jednostkowym refaktoryzacja na wielką skalę staje się znacznie prostsza. W tym zaś rozdziale, choć wciąż będziemy bazować na wcześniejszych, przykładowych programach, skupimy się bardziej na zaawansowanych technikach stosowanych w języku Python, niż na samym testowaniu.

Poniżej przedstawiony jest pełny kod programu będącego tanim i prostym sposobem uruchamiania testów regresyjnych. Pobiera on testy jednostkowe, jakie zostały napisane dla poszczególnych modułów, zbiera je do jednego, wielkiego zestawu testowego i uruchamia je wszystkie jako całość. Obecnie, podczas pisania tej książki, skrypt ten służy mi jako część procesu budowania; napisałem testy jednostkowe dla wielu przykładowych programów (nie tylko dla `roman.py` przedstawionego w rozdziale 13, "Testowanie"), a pierwszą rzeczą jaką robi skrypt do automatycznego budowania jest uruchomienie tego właśnie programu, dzięki czemu mogę się upewnić, że wszystkie moje przykłady wciąż działają. Jeśli zakończy się on niepowodzeniem, wówczas automatyczne budowanie zostaje natychmiast przerwane. Nie chcę publikować niedziałających przykładów, podobnie jak wy nie chcecie pobierać ich z sieci, a później długo siedzieć, drapać się po głowie i zastanawiać, dlaczego nie działają.

Przykład 16.1. regression.py

Jeśli jeszcze tego nie zrobiliście, możecie pobrać ten oraz inne przykłady używane w tej książce.

```
"""Regression testing framework

This module will search for scripts in the same directory named
XYZtest.py.  Each such script should be a test suite that tests a
module through PyUnit.  (As of Python 2.1, PyUnit is included in
the standard library as "unittest".)  This script will aggregate all
found test suites into one big test suite and run them all at once.
"""

import sys, os, re, unittest

def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
    load = unittest.defaultTestLoader.loadTestsFromModule
    return unittest.TestSuite(map(load, modules))

if __name__ == "__main__":
```

```
    unittest.main(defaultTest="regressionTest")
```

Uruchomienie programu w tym samym katalogu, w którym znajdują się pozostałe przykładowe skrypty używane w tej książce, spowoduje wyszukanie wszystkich testów jednostkowych o nazwie `moduletest.py`, uruchomienie ich wszystkich jako jeden test, a następnie stwierdzenie, czy jako całość przeszły, czy nie.

Przykład 16.2. Przykładowe wyjście z programu regression.py

```
[you@localhost py]$ python regression.py -v
help should fail with no object ... ok                            #(1)
help should return known result for apihelper ... ok
help should honor collapse argument ... ok
help should honor spacing argument ... ok
buildConnectionString should fail with list input ... ok          #(2)
buildConnectionString should fail with string input ... ok
buildConnectionString should fail with tuple input ... ok
buildConnectionString handles empty dictionary ... ok
buildConnectionString returns known result with known input ... ok
fromRoman should only accept uppercase input ... ok               #(3)
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
kgp a ref test ... ok
kgp b ref test ... ok
kgp c ref test ... ok
kgp d ref test ... ok
kgp e ref test ... ok
kgp f ref test ... ok
kgp g ref test ... ok

----------------------------------------------------------------------
Ran 29 tests in 2.799s

OK
```

1. Pierwszych 5 testów pochodzi z `apihelpertest.py`, który testuje przykładowy skrypt z rozdziału 4 (Potęga introspekcji).

2. Kolejne 5 testów pochodzi z `odbchelpertest.py`, który testuje przykładowy skrypt z rozdziału 2 (Pierwszy program)

3. Pozostałe testy pochodzą z `romantest.py`, zestawu testów, który zgłębialiśmy w rozdziale 13, ("Testowanie").

16.7 Znajdowanie ścieżki

Czasami, kiedy uruchomimy skrypt języka Python z linii poleceń, chcielibyśmy wiedzieć, w jakim miejscu na dysku ten skrypt się znajduje.

To jeden z tych brzydkich, małych trików, które ciężko wymyślić samemu (o ile to w ogóle możliwe), ale za to łatwo zapamiętać, jeśli już się to zobaczy. Kluczem do tego problemu jest sys.argv. Jak widzieliście w rozdziale 9 ("Przetwarzanie XML"), jest to lista przechowująca argumenty linii poleceń. Dodatkowo, lista ta przechowuje również nazwę uruchamianego programu, dokładnie taką, jaka została przekazana w linii poleceń, a na jej podstawie można już ustalić położenie programu na dysku.

Przykład 16.3. fullpath.py

Jeśli jeszcze tego nie zrobiliście, możecie pobrać ten oraz inne przykłady używane w tej książce.

```
import sys, os

print 'sys.argv[0] =', sys.argv[0]             #(1)
pathname = os.path.dirname(sys.argv[0])        #(2)
print 'path =', pathname
print 'full path =', os.path.abspath(pathname) #(3)
```

1. Niezależnie od tego, w jaki sposób uruchomicie skrypt, `sys.argv[0]` będzie zawsze zawierać nazwę skryptu, w dokładnie takiej postaci, w jakiej pojawiła się ona w linii poleceń. Jak wkrótce zobaczymy, nazwa może, choć nie musi, zawierać informację o pełnej ścieżce.

2. `os.path.dirname` pobiera napis zawierający nazwę pliku i zwraca fragment tego napisu zawierający ścieżkę do katalogu, w którym plik się znajduje. Jeśli podana nazwa pliku nie zawiera informacji o ścieżce, wywołanie `os.path.dirname` zwróci napis pusty.

3. Kluczową funkcją jest `os.path.abspath`. Pobiera ona nazwę ścieżkową, która może być częściowa (względna) lub pusta, i zwraca pełną kwalifikowaną nazwę ścieżkową.

Funkcja `os.path.abspath` wymaga pewnych wyjaśnień. Jest ona bardzo elastyczna i może przyjmować nazwy ścieżkowe w dowolnej postaci.

Przykład 16.4. Dalsze wyjaśnienia dotyczące os.path.abspath

```
>>> import os
>>> os.getcwd()                        #(1)
/home/you
>>> os.path.abspath()                #(2)
/home/you
>>> os.path.abspath('.ssh')            #(3)
/home/you/.ssh
>>> os.path.abspath('/home/you/.ssh')  #(4)
/home/you/.ssh
>>> os.path.abspath('.ssh/../foo/')    #(5)
/home/you/foo
```

1. `os.getcwd()` zwraca bieżący katalog roboczy.
2. Wywołanie `os.path.abspath` z napisem pustym zwraca bieżący katalog roboczy, tak samo jak `os.getcwd()`.
3. Wywołanie `os.path.abspath` z częściową nazwą ścieżkową powoduje skonstruowanie pełnej kwalifikowanej nazwy ścieżkowej w oparciu o bieżący katalog roboczy.
4. Wywołanie `os.path.abspath` z pełną nazwą ścieżkową zwraca tę nazwę.
5. `os.path.abspath` normalizuje nazwę ścieżkową, którą zwraca. Zwróćcie uwagę, że powyższy przykład będzie działał nawet wówczas, jeśli katalog "foo" nie istnieje. Funkcja `os.path.abspath` nigdy nie sprawdza istnienia elementów składowych ścieżki na dysku; dokonuje ona jedynie manipulacji na napisach.

Katalogi i pliki, których nazwy są przekazywane do `os.path.abspath` nie muszą istnieć w systemie plików.

`os.path.abspath` nie tylko konstruuje pełne nazwy ścieżkowe (ścieżki bezwzględne), lecz również je normalizuje. Oznacza to, że wywołując `os.path.abspath('bin/../local/bin')` z katalogu /usr/ otrzyma się napis `/usr/local/bin`. Normalizacja oznacza tutaj największe możliwe uproszczenie ścieżki. W celu znormalizowania nazwy ścieżkowej bez przekształcania jej w pełną ścieżkę należy użyć funkcji `os.path.normpath`.

Przykład 16.5. Przykładowe wyjście z programu fullpath.py

```
[you@localhost py]$ python /home/you/diveintopython/common/py/fullpath.py #(1)
sys.argv[0] = /home/you/diveintopython/common/py/fullpath.py
path = /home/you/diveintopython/common/py
full path = /home/you/diveintopython/common/py
[you@localhost diveintopython]$ python common/py/fullpath.py              #(2)
sys.argv[0] = common/py/fullpath.py
path = common/py
full path = /home/you/diveintopython/common/py
[you@localhost diveintopython]$ cd common/py
[you@localhost py]$ python fullpath.py                                    #(3)
sys.argv[0] = fullpath.py
path =
full path = /home/you/diveintopython/common/py
```

1. W pierwszym przypadku `sys.argv[0]` zawiera pełną ścieżkę do skryptu. Można użyć funkcji `os.path.dirname` w celu usunięcia nazwy skryptu, otrzymując pełną ścieżkę do katalogu, w którym znajduje się skrypt. Funkcja `os.path.abspath` zwraca dokładnie to samo, co otrzymała na wejściu.
2. Jeśli skrypt jest uruchomiony przy użyciu ścieżki względnej, `sys.argv[0]` w dalszym ciągu zwraca dokładnie to, co pojawiło się w linii poleceń. Wywołanie

os.path.dirname zwróci częściową nazwę ścieżkową (ścieżkę względną względem bieżącego katalogu), natomiast os.path.abspath z częściowej nazwy ścieżkowej skonstruuje pełną ścieżkę (ścieżkę bezwzględną).

3. Jeśli skrypt jest uruchomiony z bieżącego katalogu bez podawania jakiejkolwiek ścieżki, os.dir.pathname zwróci po prostu pusty napis. Podając pusty napis do os.path.abspath otrzymamy ścieżkę do bieżącego katalogu, a tego dokładnie oczekujemy, ponieważ z tego właśnie katalogu uruchamialiśmy skrypt.

Podobnie jak inne funkcje w modułach os oraz os.path, os.path.abspath jest funkcją działającą na wszystkich platformach. Jeśli używacie systemu operacyjnego Windows (który w charakterze separatorów elementów ścieżki używa odwróconych ukośników) lub MacOS (który używa dwukropków) , wyjście waszych programów będzie się odrobinę różniło od przedstawionego w tej książce, ale przykłady będą nadal działały. I o to właśnie chodzi w module os.

Dodatek. Jeden z czytelników był rozczarowany zaprezentowanym wyżej rozwiązaniem, ponieważ chciał uruchomić wszystkie testy jednostkowe znajdujące się w bieżącym katalogu, niekoniecznie zaś w katalogu, w którym umieszczony jest program regression.py. Zasugerował on następujące podejście:

Przykład 16.6. Uruchomienie skryptu z bieżącego katalogu

```
import sys, os, re, unittest

def regressionTest():
    path = os.getcwd()       #(1)
    sys.path.append(path)    #(2)
    files = os.listdir(path) #(3)
```

1. Zamiast ustalania ścieżki z testami na katalog, w którym znajduje się obecnie wykonywany skrypt, ustalamy ją na bieżący katalog roboczy. Będzie to ten katalog, w którym byliśmy w momencie uruchomienia skryptu, a więc niekoniecznie oznacza katalog, w którym znajduje się skrypt. (Jeśli nie chwytasz tego od razu, przeczytaj to zdanie powoli kilka razy).

2. Dodajemy tę ścieżkę do ścieżki wyszukiwania bibliotek języka Python, dzięki czemu w momencie dynamicznego importowania modułów z testami jednostkowymi Python będzie mógł je odnaleźć. Nie trzeba było tego robić w sytuacji, w której ścieżką z testami była ścieżka do uruchomionego skryptu, ponieważ Python zawsze przeszukuje katalog, w którym znajduje się uruchomiony skrypt.

3. Pozostała część funkcji pozostaje bez zmian.

Dzięki tej technice możliwe jest powtórne użycie skryptu regression.py w wielu projektach. Wystarczy umieścić skrypt w pewnym katalogu wspólnym dla wielu projektów, a następnie, przed jego uruchomieniem, zmienić katalog na katalog projektu, którego testy chcemy uruchomić. Po uruchomieniu skryptu zostaną odnalezione i uruchomione wszystkie testy projektu, znajdujące się w katalogu projektu, nie zaś testy znajdujące się w katalogu wspólnym dla projektów, w którym umieszczony został skrypt.

16.8 Filtrowanie listy

Jeszcze o filtrowaniu list

Zapoznaliście się już z filtrowaniem list przy użyciu wyrażeń listowych (ang. *list comprehension*). Istnieje jeszcze jeden sposób na osiągnięcie tego celu, przez wiele osób uznawany za bardziej wyrazisty.

W języku Python istnieje wbudowana funkcja filtrująca (`filter`) przyjmująca dwa parametry, funkcję oraz listę, i zwracająca listę [1]. Funkcja przekazana jako pierwszy argument do funkcji `filter` musi przyjmować jeden argument, natomiast lista zwrócona przez funkcję filtrującą będzie zawierała te elementy z listy przekazanej do funkcji filtującej, dla których funkcja przekazana w pierwszym argumencie zwróciła wartość `true`.

Czy wszystko jasne? To nie takie trudne, jak się wydaje.

Przykład 16.7. Wprowadzenie do funkcji `filter`

```
>>> def odd(n):                 #(1)
...     return n % 2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> filter(odd, li)             #(2)
[1, 3, 5, 9, -3]
>>> [e for e in li if odd(e)]   #(3)
[1, 3, 5, 9, -3]
>>> filteredList = []
>>> for n in li:                #(4)
...     if odd(n):
...         filteredList.append(n)
...
>>> filteredList
[1, 3, 5, 9, -3]
```

1. funkcja `odd` zwraca `True` jeśli n jest nieparzyste a `False` w przeciwnym przypadku; używa do tego wbudowanej funkcji modulo ("%").

2. funkcja `filter` przyjmuje dwa argumenty: funkcję `odd` oraz listę `li`. `filter` iteruje po liście i dla każdego jej elementu wywołuje `odd`. Jeśli `odd` zwróci wartość `true` (pamiętajcie, że każda niezerowa wartość ma w języku Python logiczną wartość `true`), wówczas element jest dodawany do listy wynikowej, w przeciwnym przypadku jest on pomijany. W rezultacie otrzymujemy listę nieparzystych elementów z listy oryginalnej, w takiej samej kolejności, w jakiej elementy pojawiały się na oryginalnej liście.

3. Jak widzieliśmy w podrozdziale 4.5 Filtrowanie listy, ten sam cel można osiągnąć używając wyrażeń listowych.

[1] Patrząc z technicznego punktu widzenia, drugim argumentem funkcji `filter` może być dowolna sekwencja, włączając w to listy, krotki oraz klasy, które funkcjonują jak listy, ponieważ mają zdefiniowaną metodę specjalną `__getitem__`. Jeśli to możliwe, funkcja `filter` zwraca ten sam typ danych, który otrzymała, a więc wynikiem filtrowania listy będzie lista, a wynikiem filtrowania krotki będzie krotka. Uwagi te dotyczą również funkcji `map`, o której będzie mowa w następnym podrozdziale.

4. Można również użyć pętli. W zależności od tego, jakim doświadczeniem programistycznym dysponujecie, ten sposób może się wam wydawać bardziej "bezpośredni", jednak użycie funkcji takich jak `filter` jest znacznie bardziej wyraziste. Nie tylko jest prostsze w zapisie, jest również łatwiejsze w czytaniu. Czytanie kodu pętli przypomina patrzenie na obraz ze zbyt małej odległości; widzi się detale, jednak potrzeba kilku chwil, by móc odsunąć się na tyle, aby dojrzeć całe dzieło: "Och, to tylko filtrowanie listy!".

Przykład 16.8. funkcja `filter` w regression.py

```
files = os.listdir(path)                                  #(1)
test = re.compile("test\.py$", re.IGNORECASE)             #(2)
files = filter(test.search, files)                        #(3)
```

1. Jak widzieliśmy w podrozdziale 16.2 Znajdowanie ścieżki, ścieżka (`path`) może zawierać pełną lub częściową nazwę ścieżki do katalogu, w którym znajduje się właśnie wykonywany skrypt lub może zawierać pusty napis, jeśli skrypt został uruchomiony z bieżącego katalogu. Jakkolwiek by nie było, na liście `files` znajdą się nazwy plików z tego samego katalogu, w którym znajduje się uruchomiony skrypt.

2. Jest to skompilowane wyrażenie regularne. Jak widzieliśmy w podrozdziale 15.3 sec:Refaktoryzacja]Refaktoryzacja, jeśli to samo wyrażenie ma być używane więcej niż raz, warto je skompilować w celu poprawienia wydajności programu. Obiekt powstały w wyniku kompilacji posiada metodę `search`, która pobiera jeden argument: napis, do którego powinno się dopasować wyrażenie regularne. Jeśli dopasowanie nastąpi, metoda `search` zwróci obiekt klasy `Match` zawierający informację o sposobie dopasowania wyrażenia regularnego; w przeciwnym przypadku zwróci `None`, czyli zdefiniowaną w języku Python wartość null.

3. Metoda `search` powinna zostać wywołana na obiekcie skompilowanego wyrażenia regularnego dla każdego elementu na liście `files`. Jeśli wyrażenie zostanie dopasowane do elementu, metoda ta zwróci obiekt klasy `Match`, który ma wartość logiczną `true`, a więc element zostanie dołączony do listy wynikowej zwróconej przez filtr. Jeśli wyrażenie nie zostanie dopasowane, metoda `search` zwróci wartość `None`, która ma wartość logiczną `false`, a więc dopasowywany element nie zostanie dołączony do listy wynikowej.

Notatka historyczna. Wersje języka Python wcześniejsze niż 2.0 nie obsługiwały jeszcze wyrażeń listowych, a więc nie można było ich użyć w celu przefiltrowania listy; istniała jednak funkcja `filter`. Nawet po wprowadzeniu wyrażeń listowych w 2.0 niektóre osoby wciąż wolą używać starej metody filtrującej `filter` (oraz towarzyszącej jej funkcji `map`, o której powiem jeszcze w tym rozdziale). W chwili obecnej działają obydwie te techniki, a wybór jednej z nich jest po prostu kwestią stylu. Toczy się dyskusja, czy `map` i `filter` nie powinny zostać "przedawnione" w przyszłych wersjach języka, jednak dotychczas ostateczna decyzja nie zapadła.

Przykład 16.9. Filtrowanie przy użyciu wyrażeń listowych

```
files = os.listdir(path)
test = re.compile("test\.py$", re.IGNORECASE)
files = [f for f in files if test.search(f)] #(1)
```

1. Wykonanie tej linii kodu będzie miało dokładnie ten sam efekt, co użycie funkcji filtrującej. Który sposób jest bardziej ekspresyjny? Wszystko zależy od was.

16.9 Odwzorowywanie listy

Jeszcze o odwzorowywaniu list

Wiecie już, w jaki sposób użyć wyrażeń listowych w celu odwzorowania jednej listy w inną. Można to osiągnąć również w inny sposób, używając wbudowanej funkcji `map`. Działa ona podobnie do funkcji `filter`.

Przykład 16.10. Wprowadzenie do funkcji `map`

```
>>> def double(n):
...     return n*2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> map(double, li)                          #(1)
[2, 4, 6, 10, 18, 20, 512, -6]
>>> [double(n) for n in li]                  #(2)
[2, 4, 6, 10, 18, 20, 512, -6]
>>> newlist = []
>>> for n in li:                             #(3)
...     newlist.append(double(n))
...
>>> newlist
[2, 4, 6, 10, 18, 20, 512, -6]
```

1. Funkcja `map` pobiera jako parametry funkcję oraz listę [1], a zwraca nową listę, która powstaje w wyniku wywołania funkcji przekazanej w pierwszym parametrze dla każdego elementu listy przekazanej w drugim parametrze. W tym przypadku każdy element listy został pomnożony przez 2.

2. Ten sam efekt można osiągnąć wykorzystując wyrażenia listowe. Wyrażenia listowe pojawiły się w języku Python w wersji 2.0; funkcja `map` istniała w języku od zawsze.

3. Jeśli bardzo chcecie myśleć jak programista Visual Basica, to moglibyście również do tego celu użyć pętli.

Przykład 16.11. funkcja `map` z listami zawierającymi elementy różnych typów

```
>>> li = [5, 'a', (2, 'b')]
>>> map(double, li)                          #(1)
[10, 'aa', (2, 'b', 2, 'b')]
```

1. Chciałbym zwrócić uwagę, że funkcja, której używamy jako argumentu `map` może być bez problemu zastosowana do list zawierających elementy różnych typów, o ile oczywiście poprawnie obsługuje każdy z typów, jakie posiadają elementy listy. W tym przypadku funkcja `double` mnoży swój argument przez 2, a Python wykona w tym momencie operację właściwą dla typu tego argumentu. W przypadku wartości całkowitych oznacza to pomnożenie wartości przez 2; w przypadku napisów oznacza to wydłużenie napisu o samego siebie; dla krotek oznacza to utworzenie nowej krotki zawierającej wszystkie elementy krotki oryginalnej, a po nich ponownie wszystkie elementy krotki oryginalnej.

[1] Patrz przypis w poprzednim podrozdziale, Filtrowanie listy

Dobra, koniec zabawy. Popatrzmy na kawałek prawdziwego kodu.
Przykład 16.12. Funkcja `map` w regression.py

```
filenameToModuleName = lambda f: os.path.splitext(f)[0] #(1)
moduleNames = map(filenameToModuleName, files)         #(2)
```

1. Jak widzieliśmy w podrozdziale 4.7 Wyrażenia lambda, `lambda` pozwala na zdefiniowanie funkcji w locie. W przykładzie 6.17 "Rozdzielanie ścieżek" (w podrozdziale Praca z katalogami), `os.path.splitext` pobiera nazwę pliku i zwraca parę (nazwa, rozszerzenie). A więc funkcja `filenameToModuleName` bierze nazwę pliku jako parametr i odcina od niej rozszerzenie, zwracając nazwę bez rozszerzenia.

2. Wywołanie `map` spowoduje wywołanie funkcji `filenameToModuleName` dla każdego elementu z listy `files` , a w rezultacie zwrócenie listy wartości, jakie powstały po każdym z tych wywołań. Innymi słowy, odcinamy rozszerzenie dla każdej nazwy pliku i zbieramy powstałe w ten sposób nazwy bez rozszerzeń do listy `moduleNames`.

Jak zobaczycie w dalszej części rozdziału, tego typu myślenie, które jest mocno skoncentrowane na przetwarzanych danych, można rozciągnąć przez wszystkie etapy pisania kodu, aż do ostatecznego celu, jakim jest zdefiniowanie i uruchomienie pojedynczego zestawu testów jednostkowych zawierającego testy ze wszystkich poszczególnych przypadków testowych.

16.10 Programowanie koncentrujące się na danych

W tym momencie zastanawiacie się zapewne, dlaczego takie podejście może być uznane za lepsze od podejścia, w którym używa się pętli i bezpośrednich wywołań funkcji. I to jest bardzo dobre pytanie. Przede wszystkim jest to kwestia przyjęcia pewnej optyki. Użycie funkcji `map` oraz `filter` zmusza do skoncentrowania się na przetwarzanych danych.

W tym przypadku zaczęliśmy od sytuacji, w której w ogóle nie było żadnych danych; pierwszą rzeczą, jaką zrobiliśmy, było uzyskanie ścieżki do katalogu, w którym znajdował się uruchomiony skrypt, a kolejną — uzyskanie na tej podstawie listy plików znajdujących się w tym katalogu. W ten sposób zaczęliśmy i dzięki tym krokom zdobyliśmy dane, na których mogliśmy dalej pracować: listę nazw plików.

Wiedzieliśmy jednak, że nie interesują nas wszystkie pliki, a jedynie te, które są zestawami testów. Mieliśmy zbyt dużo danych, a więc potrzebowaliśmy je jakoś przefiltrować. Skąd wiedzieliśmy, które dane zachować? Potrzebowaliśmy funkcji, która by to sprawdzała, a którą mogliśmy przekazać do funkcji filtrującej. W tym akurat przypadku użyliśmy wyrażenia regularnego, ale koncepcja jest wciąż taka sama, niezależnie od tego, w jaki sposób skonstruowana została funkcja sprawdzająca.

Po tym kroku posiadaliśmy już listę nazw plików będących zestawami testowymi (i tylko nimi, ponieważ wszystkie inne pliki zostały odfiltrowane), jednak w rzeczywistości potrzebowaliśmy jedynie listę nazw modułów. Mieliśmy wszystkie dane, jednak były one w złym formacie. Zdefiniowaliśmy więc funkcję, która przekształcała nazwę pliku w nazwę modułu i każdy element z listy nazw plików odwzorowaliśmy przy pomocy tej funkcji w nazwę modułu, uzyskując listę nazw modułów. Z każdej nazwy pliku powstała jedna nazwa modułu, a z listy nazw plików powstała lista nazw modułów.

Zamiast funkcji `filter` mogliśmy użyć pętli `for` z instrukcją `if`. Zamiast funkcji `map` mogliśmy użyć pętli `for` z wywołaniem funkcji. Jednak używanie pętli w ten sposób jest zajęciem czasochłonnym. W najlepszym przypadku stracimy niepotrzebnie czas, a w najgorszym wprowadzimy brzydkie błędy. Na przykład, odpowiadając na pytanie: "czy ten plik jest zestawem testów?" zastanawiamy się nad logiką specyficzną dla danego zastosowania i żaden język programowania nie wyrazi tego za nas. Jednak kiedy już wiemy, jak na takie pytanie odpowiedzieć, czy naprawdę potrzebujemy tego kłopotliwego tworzenia nowej, pustej listy, napisania pętli `for` i instrukcji `if`, a następnie ręcznego wywoływania funkcji `append`, aby dodać element, który przeszedł przez test w warunku `if` do tej listy, a dodatkowo jeszcze śledzenia, jaka zmienna przechowuje dane już przefiltrowane, a jaka te, które dopiero będą filtrowane? Dlaczego nie mielibyśmy po prostu zdefiniować odpowiedniego warunku, a całą resztę zrobi za nas Python?

Oczywiście, moglibyśmy być sprytni i nie tworzyć nowej listy, lecz usuwać niepotrzebne elementy z listy wejściowej. Ale już się na tym sparzyliśmy: próba modyfikowania listy, po której właśnie iterujemy, może powodować błędy. Usuwamy element, przechodzimy do następnego elementu, i tym samym przeskakujemy przez jakiś element. Czy Python to jeden z tych języków, w których usuwanie elementów działa w ten właśnie sposób? Ile czasu zajmie nam ustalenie tego faktu? Czy będziemy pamiętać, czy taka iteracja jest bezpieczna, czy nie, kiedy będziemy robić to ponownie? Programiści tracą zbyt wiele czasu i popełniają wiele błędów podczas zajmowania się takimi — czysto przecież technicznymi — kwestiami, co jest przecież bezcelowe. Nie posuwa to pracy nad programem ani o jotę, tylko niepotrzebnie zajmuje czas.

Kiedy uczyłem się języka Python po raz pierwszy, stroniłem od wyrażeń listowych, a od funkcji `map` i `filter` stroniłem jeszcze dłużej. Upierałem się, aby moje życie było

bardziej skomplikowane, ponieważ przylgnąłem do znanego mi sposobu programowania zorientowanego na kod: używałem pętli `for` oraz instrukcji warunkowych. Moje programy przypominały programy pisane w języku Visual Basic, przedstawiały bowiem dokładnie każdy krok każdej operacji w każdej funkcji. W nich wszystkich pojawiały się też wciąż te same, małe problemy i brzydkie błędy. I nie miało to większego sensu.

Zapomnijmy o tym. Szczegółowe rozpisywanie kodu nie jest ważne. Ważne są dane. Dane nie są trudne, to tylko dane. Jeśli mamy ich za dużo, przefiltrujmy je. Jeśli nie są dokładnie takie, jakich sobie życzymy, użyjmy odwzorowania `map`. Skoncentrujmy się na danych, a niepotrzebną pracę zostawmy za sobą.

16.11 Dynamiczne importowanie modułów

Dynamiczne importowanie modułów

OK, dość filozofowania. Pogadajmy o dynamicznym importowaniu modułów.

Najpierw zerknijmy jak normalnie importuje się moduły. Składnia polecenia import module sprawdza ścieżkę w poszukiwaniu nazwanego modułu i importuje go po nazwie. W ten sposób można importować kilka modułów na raz, podając nazwy modułów oddzielone przecinkiem. Z resztą, robiliśmy już to w pierwszej linii skryptu z tego rozdziału.

Przykład 16.13. Importowanie wielu modułów na raz

```
import sys, os, re, unittest #(1)
```

1. Importowane są cztery moduły na raz: sys (funkcje systemowe oraz dostępu do parametrów przekazywanych z linii poleceń), os (wykonywanie funkcji systemowych takich jak np. listowanie katalogów), re (wyrażenia regularne), oraz unittest (testy jednostkowe).

A teraz zróbmy to samo, jednak przy użyciu dynamicznego importowania.

Przykład 16.14. Dynamiczne importowanie modułów

```
>>> sys = __import__('sys')           #(1)
>>> os = __import__('os')
>>> re = __import__('re')
>>> unittest = __import__('unittest')
>>> sys                               #(2)
>>> <module 'sys' (built-in)>
>>> os
>>> <module 'os' from '/usr/local/lib/python2.2/os.pyc'>
```

1. Wbudowana funkcja __import__ robi to samo co użycie polecenia import, jednak jest to funkcja rzeczywista, która przyjmuje ciąg znaków jako argument.
2. Zmienna sys staje się modułem sys, to tak jakby napisać import sys. Zmienna os staje się modułem os i tak dalej.

Reasumując, __import__ importuje moduł, jednak aby tego dokonać, pobiera jako argument ciąg znaków. W tym przypadku moduł, który zaimportowaliśmy był po prostu na sztywno zakodowanym ciągiem znaków, jednak nic nie stało na przeszkodzie, aby była to zmienna lub wynik działania funkcji. Zmienna, pod którą podstawiamy moduł, nie musi się nazywać tak samo jak nazwa modułu, który importujemy. Równie dobrze moglibyśmy zaimportować szereg modułów i przypisać je do listy.

Przykład 16.15. Dynamiczne importowanie listy modułów

```
>>> moduleNames = ['sys', 'os', 're', 'unittest'] #(1)
>>> moduleNames
['sys', 'os', 're', 'unittest']
>>> modules = map(__import__, moduleNames)        #(2)
>>> modules                                       #(3)
[<module 'sys' (built-in)>,
<module 'os' from 'c:\Python22\lib\os.pyc'>,
```

```
<module 're' from 'c:\Python22\lib\re.pyc'>,
<module 'unittest' from 'c:\Python22\lib\unittest.pyc'>]
>>> modules[0].version                              #(4)
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
>>> import sys
>>> sys.version
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
```

1. `moduleNames` jest po prostu listą ciągów znaków. Nic nadzwyczajnego, za wyjątkiem tego, że akurat te ciągi znaków są nazwami modułów, które moglibyśmy zaimportować, jeśli byśmy chcieli.

2. Wyobraźmy sobie, że chcieliśmy je zaimportować, a dokonaliśmy tego poprzez mapowanie funkcji `__import__` na listę. Pamiętajmy jednak, że każdy element listy (`moduleNames`) będzie przekazany jako argument do wywołania raz za razem funkcji (`__import__`), dzięki czemu zostanie zbudowana i zwrócona lista wartości wynikowych

3. Tak więc z listy ciągów znaków stworzyliśmy tak na prawdę listę rzeczywistych modułów. (Nasze ścieżki mogą się różnić w zależności od systemu operacyjnego, na którym zainstalowaliśmy Pythona, faz księżyca i innych takich tam.)

4. Aby upewnić się, że są to tak na prawdę moduły, zerknijmy na niektóre ich atrybuty. Pamiętajmy, że `modules[0]` jest modułem sys, więc `modules[0].version` odpowiada sys.version. Wszystkie pozostałe atrybuty i metody tych modułów są także dostępne. Nie ma nic niezwykłego w poleceniu import, tak samo jak nie ma nic magicznego w modułach. Moduły są obiektami. Wszystko jest obiektem.

Teraz już powinniśmy móc wszystko to poskładać do kupy i rozszyfrować o co tak na prawdę chodzi w kodzie zamieszczonych tutaj przykładów.

Rozdział 17

Programowanie funkcyjne

17.1 Programowanie funkcyjne - wszystko razem

Dowiedzieliście się już wystarczająco dużo, by móc odczytać pierwszych siedem linii kodu z przykładu podanego na początku rozdziału, w którym odczytywane są pliki w katalogu a następnie importowane wybrane spośród nich moduły.

Przykład 16.16. Funkcja `regressionTest`

```
def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
load = unittest.defaultTestLoader.loadTestsFromModule
return unittest.TestSuite(map(load, modules))
```

Spójrzmy na ten kod w sposób interaktywny, linia po linii. Załóżmy, że katalogiem bieżącym jest c:\diveintopython\py, w którym znajdują się przykłady dołączone do tej książki, z omawianym w tym rozdziale skryptem włącznie. Jak widzieliśmy w podrozdziale 16.2 Znajdowanie ścieżki, nazwa katalogu, w którym znajduje się skrypt, trafi do zmiennej `path`, spróbujmy więc tę część zakodować na sztywno, po czym dopiero przejść dalej.

Przykład 16.17. Krok 1: Pobieranie wszystkich plików

```
>>> import sys, os, re, unittest
>>> path = r'c:\diveintopython\py'
>>> files = os.listdir(path)
>>> files #(1)
['BaseHTMLProcessor.py', 'LICENSE.txt', 'apihelper.py', 'apihelpertest.py',
'argecho.py', 'autosize.py', 'builddialectexamples.py', 'dialect.py',
'fileinfo.py', 'fullpath.py', 'kgptest.py', 'makerealworddoc.py',
'odbchelper.py', 'odbchelpertest.py', 'parsephone.py', 'piglatin.py',
'plural.py', 'pluraltest.py', 'pyfontify.py', 'regression.py', 'roman.py',
'romantest.py', 'uncurly.py', 'unicode2koi8r.py', 'urllister.py', 'kgp',
'plural', 'roman', 'colorize.py']
```

1. files jest listą nazw wszystkich plików i katalogów znajdujących się w katalogu, z którego pochodzi skrypt. (Jeśli wcześniej uruchamialiście już jakieś przykłady, na liście możecie również zauważyć pliki .pyc)

Przykład 16.18. Krok 2: Filtrowanie w celu znalezienia interesujących plików

```
>>> test = re.compile("test\.py$", re.IGNORECASE)           #(1)
>>> files = filter(test.search, files)                      #(2)
>>> files                                                   #(3)
['apihelpertest.py', 'kgptest.py', 'odbchelpertest.py', 'pluraltest.py',
'romantest.py']
```

1. Wyrażenie regularne zostanie dopasowane do każdego napisu zakończonego na test.py. Zauważcie, że kropka musi zostać poprzedzona sekwencją unikową; w wyrażeniach regularnych kropka oznacza "dopasuj dowolny znak", jednak nam zależy na dosłownym dopasowaniu znaku kropki.
2. Skompilowane wyrażenie regularne działa jak funkcja, a więc możemy jej użyć do przefiltrowania długiej listy nazw plików i katalogów, dzięki czemu uzyskamy listę nazw, do których zostało dopasowane wyrażenie.
3. Otrzymaliśmy więc listę skryptów będących testami jednostkowymi, ponieważ tylko one mają nazwę JAKASNAZWAtest.py.

Przykład 16.19. Krok 3: Odwzorowanie nazw plików na nazwy modułów

```
>>> filenameToModuleName = lambda f: os.path.splitext(f)[0] #(1)
>>> filenameToModuleName('romantest.py')                    #(2)
'romantest'
>>> filenameToModuleName('odchelpertest.py')
'odbchelpertest'
>>> moduleNames = map(filenameToModuleName, files)          #(3)
>>> moduleNames                                             #(4)
['apihelpertest', 'kgptest', 'odbchelpertest', 'pluraltest', 'romantest']
```

1. Jak widzieliśmy w podrozdziale 4.7 Wyrażenia lambda, `lambda` pozwala na szybkie zdefiniowanie jednolinijkowych funkcji w locie. Tutaj funkcja lambda pobiera nazwę pliku wraz z rozszerzeniem i zwraca część nazwy bez rozszerzenia, używając do tego funkcji `os.path.splitext` z biblioteki standardowej, którą poznaliśmy w przykładzie 6.17 "Rozdzielanie ścieżek" w podrozdziale Praca z katalogami.
2. `filenameToModuleName` jest funkcją. W porównaniu ze zwykłymi funkcjami, które tworzy się przy pomocy instrukcji `def`, w funkcjach lambda nie ma niczego magicznego. Możemy wywołać `filenameToModuleName` jak każdą inną funkcję, a robi ona dokładnie to, czego potrzebujemy: odcina rozszerzenie z napisu przekazanego jej w parametrze wejściowym.
3. Tutaj możemy wywołać tę funkcję na każdej nazwie pliku znajdującej się na liście plików będących testami jednostkowymi. Używamy do tego funkcji `map`.
4. W wyniku otrzymujemy to, czego oczekiwaliśmy: listę nazw modułów.

Przykład 16.20. Krok 4: Odwzorowanie nazw modułów na moduły

```
>>> modules = map(__import__, moduleNames)                  #(1)
>>> modules                                                 #(2)
[<module 'apihelpertest' from 'apihelpertest.py'>,
<module 'kgptest' from 'kgptest.py'>,
<module 'odbchelpertest' from 'odbchelpertest.py'>,
<module 'pluraltest' from 'pluraltest.py'>,
<module 'romantest' from 'romantest.py'>]
>>> modules[-1]                                             #(3)
<module 'romantest' from 'romantest.py'>
```

1. Jak widzieliśmy w podrozdziale 16.6 Dynamiczne importowanie modułów, w celu odwzorowania listy nazw modułów (napisów) na właściwe moduły (obiekty wywoływalne, do których można mieć dostęp jak do jakichkolwiek innych modułów), można użyć funkcji `map` oraz `__import__`.

2. `modules` to teraz lista modułów, do których można mieć taki sam dostęp, jak do jakichkolwiek innych modułów.

3. Ostatnim modułem na liście jest moduł `romantest`, tak jak gdybyśmy napisali: `import romantest`

Przykład 16.21. Krok 5: Ładowanie modułów do zestawu testów

```
>>> load = unittest.defaultTestLoader.loadTestsFromModule
>>> map(load, modules)                     #(1)
[<unittest.TestSuite tests=[
  <unittest.TestSuite tests=[<apihelpertest.BadInput testMethod=testNoObject>]>,
  <unittest.TestSuite tests=[<apihelpertest.KnownValues testMethod=testApiHelper>]>,
  <unittest.TestSuite tests=[
    <apihelpertest.ParamChecks testMethod=testCollapse>,
    <apihelpertest.ParamChecks testMethod=testSpacing>]>,
    ...
  ]
]
>>> unittest.TestSuite(map(load, modules)) #(2)
```

1. To są prawdziwe obiekty modułów. Nie tylko mamy do nich dostęp taki, jak do innych modułów i możemy tworzyć instancje klas oraz wywoływać funkcje, mamy również możliwość introspekcji (wglądu) w moduł, której możemy użyć przede wszystkim do tego, aby dowiedzieć się, jakie klasy i funkcje dany moduł posiada. To właśnie robi metoda `loadTestsFromModule`: dokonuje introspekcji, a następnie dla każdego modułu zwraca obiekt `unittest.TestSuite`. Każdy taki obiekt zawiera listę obiektów `unittest.TestSuite`, po jednym dla każdej klasy dziedziczącej po `TestCase` zdefiniowanej w module. Każdy z obiektów na tej liście zawiera z kolei listę metod testowych zdefiniowanych w klasie testowej.

2. Na końcu umieszczamy listę obiektów `TestSuite` wewnątrz jednego, dużego zestawu testów. Moduł `unittest` nie ma problemów z przechodzeniem po drzewie zestawów testowych zagnieżdżonych w zestawach testowych; dotrze on do każdej metody testowej i ją wywoła, sprawdzając, czy przeszła, czy nie, a następnie przejdzie do kolejnej metody.

Moduł `unittest` zwykle przeprowadza za nas proces introspekcji. Czy pamiętacie magiczną funkcję `unittest.main()`, którą wywoływały poszczególne moduły, aby odpalić wszystkie znajdujące się w nich testy? Metoda `unittest.main()` w rzeczywistości tworzy instancję klasy `unittest.TestProgram`, która z kolei tworzy instancję `unittest.defaultTestLoader`, służącą do załadowania modułu, z którego została wywołana. (Skąd jednak ma referencję do modułu, z którego została wywołana, jeśli nie dostała jej od nas? Otóż dzięki równie magicznemu poleceniu `__import__('__main__')`, które dynamicznie importuje wykonywany właśnie moduł. Mógłbym napisać całą książkę

na temat trików i technik używanych w module `unittest`, ale chyba bym jej nie skończył.)

Przykład 16.22. Krok 6: Powiedzieć modułowi `unittest`, aby użył naszego zestawu testowego

```
if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest") #(1)
```

1. Zamiast pozwalać modułowi `unittest` wykonać całą magię za nas, większość zrobiliśmy sami. Utworzyliśmy funkcję (`regressionTest`), która sama importuje moduły i woła `unittest.defaultTestLoader`, a następnie utworzyliśmy duży zestaw testów. Teraz potrzebujemy jedynie, aby `unittest`, zamiast szukać testów i budować zestaw testowy w standardowy sposób, uruchomił funkcję `regressionTest`, która zwróci gotwy do użycia obiekt `testSuite`.

17.2 Programowanie funkcyjne - podsumowanie

Program `regression.py` i wynik jego działania powinien być teraz całkiem zrozumiały. Powinniście też bez kłopotu wykonywać następujące zadania:

- Przekształcanie informacji o ścieżce otrzymanej z linii poleceń
- Filtrowanie list przy użyciu metody `filter` zamiast używania wyrażeń listowych
- Odwzorowywanie list przy użyciu metody `map` zamiast używania wyrażeń listowych
- Dynamiczne importowanie modułów

Rozdział 18

Funkcje dynamiczne

18.1 Funkcje dynamiczne

Nurkujemy

Chcę teraz opowiedzieć o rzeczownikach w liczbie mnogiej. Także o funkcjach zwracających inne funkcje, o zaawansowanych wyrażeniach regularnych oraz o generatorach, które pojawiły się w języku Python w wersji 2.3. Zacznę jednak od tego, w jaki sposób tworzy się rzeczowniki w liczbie mnogiej.

Jeśli jeszcze nie przeczytaliście rozdziału 7 (Wyrażenia regularne), nadszedł doskonały moment, aby to zrobić. W tym rozdziale chcę szybko przejść do bardziej zaawansowanego użycia wyrażeń regularnych, zakładam więc, że dobrze rozumiecie podstawy.

Język angielski jest językiem schizofrenicznym, który sporo zapożyczył z innych języków; zasady tworzenia rzeczowników w liczbie mnogiej na podstawie liczby pojedynczej są zróżnicowane i złożone. Istnieją pewne zasady, jednak istnieją również wyjątki od tych zasad, a nawet wyjątki od tych wyjątków.

Jeśli dorastaliście w kraju, w którym mówi się po angielsku lub uczyliście się angielskiego w czasie, gdy chodziliście do szkoły, poniższe reguły powinny być wam dobrze znane:

1. Jeśli słowo kończy się na S, X lub Z, należy dodać ES. “Bass” staje się “basses”, “fax” staje się “faxes” a “waltz” staje się “waltzes”.

2. Jeśli słowo kończy się na dźwięczne H, należy dodać ES; jeśli kończy się na nieme H, należy dodać samo S. Co to jest “dźwięczne H”? Takie, które po połączeniu z innymi głoskami można usłyszeć. A więc “coach” staje się “coaches” a “rash” staje się “rashes”, ponieważ głoski CH i SH są dźwięczne. Jednak “cheetah” staje się “cheetahs”, ponieważ występuje tutaj H bezdźwięczne.

3. Jeśli słowo kończy się na Y, które brzmi jak I, należy zmienić Y na IES; jeśli Y jest połączony z głoską, która brzmi inaczej, należy dodać S. A więc “vacancy” staje się “vacancies”, ale “day” staje się “days”.

4. Jeśli wszystko zawiedzie, należy dodać S i mieć nadzieję, że się uda.

(Wiem, jest mnóstwo wyjątków. “Man” staje się “men” a “woman” staje się “women”, jednak “human” staje się “humans”. “Mouse” staje się “mice”,a “louse” staje się “lice”, jednak “house” staje się “houses”. “Knife” staje się “knives” a “wife” staje się “wives”, jednak “lowlife” staje się “lowlifes”. Nie mówcie mi nawet o słowach, które same w sobie oznaczają liczbę mnogą, jak “sheep”, “deer” czy “haiku”.)

W innych językach wygląda to oczywiście zupełnie inaczej.

Zaprojektujemy więc moduł, który dla każdego rzeczownika utworzy odpowiedni rzeczownik w liczbie mnogiej. Zaczniemy od rzeczowników w języku angielskim i od powyższych czterech zasad, jednak musimy mieć na uwadze, że obsługiwanie nowych reguł (a nawet nowych języków) jest nieuniknione.

18.2 plural.py, etap 1

Patrzymy na słowa, które — przynajmniej w języku angielskim — składają się z liter. Dysponujemy też regułami, które mówią, że musimy znaleźć w słowie pewne kombinacje liter, a następnie odpowiednio to słowo zmodyfikować. Brzmi to dokładnie jak zadanie dla wyrażeń regularnych.

Przykład 17.1. plural1.py

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):             #(1)
        return re.sub('$', 'es', noun)        #(2)
    elif re.search('[^aeioudgkprt]h$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

1. Rzeczywiście, jest to wyrażenie regularne, jednak używa ono składni, jakiej w rozdziale 7 (Wyrażenia regularne) nie widzieliście. Nawias kwadratowy oznacza: "dopasuj dokładnie jeden z wymienionych tu znaków". A więc [sxz] oznacza "s albo x, albo z", ale tylko jeden znak na raz. Znak $ powinien być wam znany; dopasowuje się on do końca napisu. A więc sprawdzamy tutaj, czy rzeczownik kończy się na jedną z liter s, x lub z.

2. Funkcja `re.sub` dokonuje podstawienia w oparciu o wyrażenie regularne. Przyjrzyjmy się jej bliżej.

Przykład 17.2. Wprowadzenie funkcji `re.sub`

```
>>> import re
>>> re.search('[abc]', 'Mark')   #(1)
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.sub('[abc]', 'o', 'Mark') #(2)
'Mork'
>>> re.sub('[abc]', 'o', 'rock') #(3)
'rook'
>>> re.sub('[abc]', 'o', 'caps') #(4)
'oops'
```

1. Czy napis Mark zawiera jedną z liter a, b lub c? Tak, zawiera a.

2. W porządku; znajdź a, b lub c i zastąp je literą o. Mark zmienia się w Mork.

3. Ta sama funkcja zmienia rock w rook.

4. Może się wydawać, że ta linia zmieni caps w oaps, jednak dzieje się inaczej. Funkcja `re.sub` zastępuje wszystkie wystąpienia, nie tylko pierwsze. Caps zmienia się w oops ponieważ zarówno c jak i a zostają zastąpione literą o.

Przykład 17.3. Z powrotem do plural1.py

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)        #(1)
    elif re.search('[^aeioudgkprt]h$', noun): #(2)
        return re.sub('$', 'es', noun)        #(3)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

1. Wróćmy do funkcji `plural`. Co robimy? Zamieniamy końcówkę napisu na "es". Innymi słowy, dodajemy "es" do napisu. Moglibyśmy osiągnąć ten cel używając dodawania napisów, na przykład stosując wyrażenie: rzeczownik + "es", jednak tutaj wyrażeń regularnych będę używał do wszystkiego, ze względu na spójność oraz z innych powodów, które zostaną wyjaśnione w dalszej części rozdziału.

2. Patrzcie uważne, to kolejna nowość. Znak ^ znajdujący się wewnątrz nawiasów kwadratowych oznacza coś szczególnego: negację. [^abc] oznacza "dowolny znak oprócz a, b oraz c". Wyrażenie [^aeioudgkprt] oznacza "każdy znak za wyjątkiem a, e, i, o, u, d, g, k, p, r oraz t. Po tym znaku powinien znaleźć się znak h kończący napis. Tutaj szukamy słów kończących się na H, które można usłyszeć.

3. Tutaj podobnie: dopasowujemy słowa kończące się na Y, przy czym znak stojący przed Y musi być dowolnym znakiem za wyjątkiem a, e, i, o oraz u. Szukamy słów, które kończą się na Y i brzmią jak I.

Przykład 17.4. Więcej na temat negacji w wyrażeniach regularnych

```
>>> import re
>>> re.search('[^aeiou]y$', 'vacancy') #(1)
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.search('[^aeiou]y$', 'boy')     #(2)
>>>
>>> re.search('[^aeiou]y$', 'day')
>>>
>>> re.search('[^aeiou]y$', 'pita')    #(3)
>>>
```

1. Wyrażenie zostanie dopasowane do "vacancy" ponieważ słowo to kończy się na cy, a c nie jest a, e, i, o ani u.

2. Nie zostanie dopasowane do "boy", który kończy się na oy, a powiedzieliśmy wyraźnie, że znakiem stojącym przed y nie może być o. "day" nie zostanie dopasowane, ponieważ kończy się na ay.

3. "pita" również nie zostanie dopasowana, ponieważ nie kończy się na y.

Przykład 17.5. Więcej na temat `re.sub`

```
>>> re.sub('y$', 'ies', 'vacancy')              #(1)
'vacancies'
>>> re.sub('y$', 'ies', 'agency')
'agencies'
>>> re.sub('([^aeiou])y$', r'\1ies', 'vacancy') #(2)
'vacancies'
```

1. To wyrażenie regularne przekształca "vacancy" w "vacancies" oraz "agency" w "agencies", dokładnie tak, jak chcemy. Zauważmy, że wyrażenie to przekształciłoby "boy" w "boies", gdyby nie fakt, że w funkcji użyliśmy wcześniej `re.search`, aby dowiedzieć się, czy powinniśmy również dokonać podstawienia przy użyciu `re.sub`.

2. Chciałbym nadmienić mimochodem, że możliwe jest połączenie tych dwóch wyrażeń regularnych (jednego, które sprawdza, czy pewna zasada ma zastosowanie, i drugiego, które faktycznie tę zasadę stosuje) w jedno. Wyglądałoby ono dokładnie tak. Większość powinna być już wam znana: aby zapamiętać znak, który stoi przed y, używamy zapamiętanej grupy, o której była mowa w podrozdziale 7.6 (Analiza przypadku: Przetwarzanie numerów telefonów). W podstawianym napisie pojawia się nowa składnia, \1, które oznacza: "czy pamiętasz grupę numer 1? wstaw ją tutaj". W tym przypadku jako znak stojący przed y zostanie zapamiętane c, po czym dokonane zostanie podstawienie c w miejsce c oraz ies w miejsce y. (Jeśli potrzebujemy więcej niż jednej zapamiętanej grupy, używamy \2, \3 itd.)

Podstawienia wyrażeń regularnych stanowią niezwykle silny mechanizm, a składnia \1 czyni go jeszcze silniejszym. Z drugiej strony przedstawienie całej operacji w postaci jednego wyrażenia regularnego sprawiłoby, że stałaby się ona mało czytelna i niewiele by miała wspólnego ze sposobem, w jaki na początku opisywaliśmy sposób konstruowania liczby mnogiej. Utworzyliśmy reguły takie jak "jeśli słowo kończy się na S, X lub Z, dodaj ES" i kiedy teraz patrzymy na funkcję `plural`, widzimy dwie linijki kodu, które mówią "jeśli słowo kończy się na S, X lub Z, dodaj ES". Nie można tego zrobić bardziej bezpośrednio.

18.3 plural.py, etap 2

Dodamy teraz warstwę abstrakcji. Zaczęliśmy od zdefiniowania listy reguł: jeśli jest tak, wtedy zrób tak, w przeciwnym przypadku idź do następnej reguły. Teraz skomplikujemy pewną część programu po to, by móc uprościć inną.

Przykład 17.6. plural2.py

```
import re

def match_sxz(noun):
    return re.search('[sxz]$', noun)

def apply_sxz(noun):
    return re.sub('$', 'es', noun)

def match_h(noun):
    return re.search('[^aeioudgkprt]h$', noun)

def apply_h(noun):
    return re.sub('$', 'es', noun)

def match_y(noun):
    return re.search('[^aeiou]y$', noun)

def apply_y(noun):
    return re.sub('y$', 'ies', noun)

def match_default(noun):
    return 1

def apply_default(noun):
    return noun + 's'

rules = ((match_sxz, apply_sxz),
         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
         )                                      #(1)

def plural(noun):
    for matchesRule, applyRule in rules:       #(2)
        if matchesRule(noun):                  #(3)
            return applyRule(noun)             #(4)
```

1. Choć ta wersja jest bardziej skomplikowana (z pewnością jest dłuższa), robi ona dokładnie to samo: próbuje dopasować kolejno cztery reguły, a następnie, jeśli dopasowanie się powiedzie, stosuje ona odpowiednie wyrażenie regularne. Różnica polega na tym, że każda reguła dopasowująca oraz modyfikująca jest zdefiniowana w swojej własnej funkcji, przy czym funkcje te zostały zebrane w zmiennej `rules`, która jest krotką krotek.

2. Używając pętli `for`, możemy z krotki `rules` wyciągać po dwie reguły na raz (jedną dopasowującą i jedną modyfikującą). Podczas pierwszej iteracji pętli `for` `matchesRule` przyjmie wartość `match_sxz`, a `applyRule` wartość `apply_sxz`. Podczas drugiej iteracji (jeśli taka nastąpi), `matchesRule` przyjmie wartość `match_h`, a `applyRule` przyjmie wartość `apply_h`.

3. Pamiętajcie, że w jezyku Python wszystko jest obiektem, nawet funkcje. Krotka `rules` składa się z dwuelementowych krotek zawierających funkcje. Nie są to nazwy funkcji, lecz rzeczywiście funkcje. W pętli są one przypisywane do `applyRule` oraz `matchesRule`, które stają się funkcjami, a więc obiektami, które można wywołać. W tym miejscu podczas w pierwszej iteracji pętli zostanie wykonany kod równoważny wywołaniu: `matches_sxz(noun)`.

4. W tym zaś miejscu podczas pierwszej iteracji pętli `for` zostanie wykonany kod równoważmy wywołaniu `apply_sxz(noun)`.

Jeśli ten dodatkowy poziom abstrakcji wydaje się zagmatwany, spróbujmy "odwikłać" powyższą funkcję w celu lepszego uwidocznienia równoważności. Pętla w funkcji `plural` jest równoważna następującej pętli:

Przykład 17.7. Rozwikływanie funkcji `plural`

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

Zysk jest taki, że funkcja `plural` znacznie się uprościła. Bierze ona listę reguł zdefiniowanych w innym miejscu i w sposób bardzo ogólny iteruje po nich: bierze regułę dopasowującą; czy reguła pasuje? Jeśli tak, wywołuje regułę modyfikującą. Reguły mogą być zdefiniowane w innym miejscu, w dowolny sposób. Funkcji `plural` pochodzenie reguł nie interesuje.

Zastanówmy się, czy warto było wprowadzać tę warstwę abstrakcji. Raczej nie. Zastanówmy się, co musielibyśmy zrobić, aby dodać do funkcji nową regułę. Cóż, w poprzednim podrozdziale e do funkcji `plural` należałoby dodać instrukcję `if`. W tym podrozdziale należałoby dodać dwie funkcje, `macth_foo` i `apply_foo`, a następnie zaktualizować listę reguł wstawiając je w takim miejscu, żeby w stosunku do innych reguł zostały one wywołane w odpowiedniej kolejności.

Tak naprawdę to było tylko wprowadzenie do kolejnego podrozdziału. Idźmy więc dalej.

18.4 plural.py, etap 3

Zauważmy, że definiowanie osobnych, nazwanych funkcji dla każdej reguły dopasowującej i modyfikującej nie jest tak naprawdę konieczne. Nigdy nie wywołujemy tych funkcji bezpośrednio; definiujemy je w krotce `rules` i wywołujemy je również przy użyciu tej krotki. Spróbujmy zatem przekształcić je do funkcji anonimowych.

Przykład 17.8. plural3.py

```
import re

rules = \
  (
    (
     lambda word: re.search('[sxz]$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeioudgkprt]h$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeiou]y$', word),
     lambda word: re.sub('y$', 'ies', word)
    ),
    (
     lambda word: re.search('$', word),
     lambda word: re.sub('$', 's', word)
    )
   )                                            #(1)

def plural(noun):
    for matchesRule, applyRule in rules:        #(2)
        if matchesRule(noun):
            return applyRule(noun)
```

1. To ten sam zestaw reguł, który widzieliśmy na etapie 2. Jedyna różnica polega na tym, że zamiast definiować funkcje nazwane, takie jak `match_sxz` czy `apply_sxz`, włączyliśmy treść tych funkcji bezpośrednio do zmiennej `rules` używając funkcji `lambda`.

2. Zauważmy, że funkcja `plural` w ogóle sie nie zmieniła. Iteruje ona po zestawie funkcji reprezentujących reguły, sprawdza pierwszą regułę, a jeśli zwróci ona wartość `true`, wywołuje ona drugą regułę i zwraca jej wynik. Dokładnie tak samo, jak wcześniej. Jedyna różnica polega teraz na tym, że funkcje z regułami zostały zdefiniowane "inline", jako funkcje anonimowe, przy użyciu funkcji `lambda`. Jednak dla funkcji `plural` sposób zdefiniowania funkcji nie ma żadnego znaczenia; otrzymuje ona listę reguł i wykonuje na niej swoją pracę.

Aby dodać nową regułę tworzenia liczby mnogiej, wystarczy zdefiniować nowe funkcje (regułę dopasowującą oraz regułę modyfikującą) bezpośrednio w samej krotce

`rules`. Teraz jednak, kiedy mamy zdefiniować reguły wewnątrz krotki, widzimy wyraźnie, że pojawiły się liczne niepotrzebne powtórzenia kodu. Mamy bowiem cztery pary funkcji, a każda z nich jest napisana według tego samego wzorca. Funkcje dopasowujące składają się z pojedynczego wywołania `re.search`, a funkcje modyfikujące — z wywołania `re.sub`. Spróbujmy zrefaktoryzować te podobieństwa.

18.5 plural.py, etap 4

Aby definiowanie nowych reguł było prostsze, spróbujemy usunąć z kodu występujące tam powtórzenia.

Przykład 17.9. plural4.py

```
import re

def buildMatchAndApplyFunctions((pattern, search, replace)):
    matchFunction = lambda word: re.search(pattern, word)      #(1)
    applyFunction = lambda word: re.sub(search, replace, word) #(2)
    return (matchFunction, applyFunction)                      #(3)
```

1. buildMatchAndApplyFunctions to funkcja, której zadaniem jest dynamiczne konstruowanie innych funkcji. Pobiera ona trzy parametry: `pattern`, `search` oraz `replace` (właściwie pobiera jeden parametr będący krotką, ale o tym za chwilę), dzięki którym można zbudować funkcję dopasowującą przy użyciu składni `lambda` tak, aby pobierała ona jeden parametr (`word`), a następnie wywoływała `re.search` z wzorcem (`pattern`) przekazanym do funkcji `buildMatchAndApplyFunctions` oraz z parametrem `word` przekazanym do właśnie budowanej funkcji dopasowującej. Ojej.

2. W taki sam sposób odbywa się budowanie funkcji modyfikującej. Funkcja modyfikująca pobiera jeden parametr i wywołuje `re.sub` z parametrami `search` i `replace` przekazanymi do funkcji `buildMatchAndApplyFunctions` oraz parametrem `word` przekazanym do właśnie budowanej funkcji modyfikującej. Pokazana tutaj technika używania wartości zewnętrznych parametrów w funkcjach budowanych dynamicznie nosi nazwę dopełnień (ang. closures). W gruncie rzeczy, podczas budowania funkcji modyfikującej, zdefiniowane zostają dwie stałe: funkcja pobiera jeden parametr (`word`), jednak dodatkowo używa ona dwóch innych wartości (`search` oraz `replace`), które zostają ustalone w momencie definiowania funkcji modyfikującej.

3. Na końcu funkcja `buildMatchAndApplyFunctions` zwraca krotkę zawierającą dwie wartości: dwie właśnie utworzone funkcje. Stałe, które zostały zdefiniowane podczas ich budowania (`pattern` w `matchFunction` oraz `search` i `replace` w `applyFunction`) pozostają zapamiętane w tych funkcjach, nawet po powrocie z funkcji `buildMatchAndApplyFunctions`. To szalenie fajna sprawa.

Jeśli jest to wciąż niesamowicie zagmatwane (powinno być, bo rzecz jest złożona), spróbujmy zobaczyć z bliska, jak tego użyć — może się odrobinę wyjaśni.

Przykład 17.10. Ciąg dalszy plural4.py

```
patterns = \
  (
    ('[sxz]$', '$', 'es'),
    ('[^aeioudgkprt]h$', '$', 'es'),
    ('(qu|[^aeiou])y$', 'y$', 'ies'),
    ('$', '$', 's')
  )                                                 #(1)
rules = map(buildMatchAndApplyFunctions, patterns)  #(2)
```

1. Nasze reguły tworzenia liczby mnogiej są teraz zdefiniowane jako seria napisów (nie funkcji). Pierwszy napis to wyrażenie regularne, które zostanie użyte w funkcji `re.search` w celu zbadania, czy reguła pasuje do zadanego rzeczownika; drugi i trzeci napis to parametry `search` oraz `replace` funkcji `re.sub`, która zostanie użyta w ramach funkcji modyfikującej do zmiany zadanego rzeczownika w odpowiednią postać liczby mnogiej.

2. To jest magiczna linijka. Pobiera ona listę napisów jako parametr `patterns`, a następnie przekształca je w listę funkcji. W jaki sposób? Otóż przez odwzorowanie listy napisów na listę funkcji przy użyciu funkcji `buildMatchAndApplyFunctions`, która, tak się akurat składa, pobiera trzy napisy jako parametr i zwraca krotkę zawierającą dwie funkcje. Oznacza to, że zmienna `rules` będzie miała ostatecznie taką samą wartość, jak w poprzednim przykładzie: listę dwuelementowych krotek, spośród których każda zawiera dwie funkcje: pierwszą jest funkcja dopasowująca, która wywołuje `re.search` a drugą jest funkcja modyfikująca, która wywołuje `re.sub`.

Przysięgam, że nie zmyślam: zmienna `rules` będzie zawierała dokładnie taką samą listę funkcji, jaką zawierała w poprzednim przykładzie. Odwikłajmy definicję zmiennej `rules` i sprawdźmy:

Przykład 17.11. Odwikłanie definicji zmiennej `rules`

```
rules = \
  (
    (
     lambda word: re.search('[sxz]$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeioudgkprt]h$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeiou]y$', word),
     lambda word: re.sub('y$', 'ies', word)
    ),
    (
     lambda word: re.search('$', word),
     lambda word: re.sub('$', 's', word)
    )
   )
```

Przykład 17.12. Dokończenie plural4.py

```
def plural(noun):
    for matchesRule, applyRule in rules:            #(1)
        if matchesRule(noun):
            return applyRule(noun)
```

1. Ponieważ lista reguł zaszyta w zmiennej `rules` jest dokładnie taka sama, jak w poprzednim przykładzie, nikogo nie powinno dziwić, że funkcja `plural` nie

zmieniła się. Pamiętajmy, że jest ona zupełnie ogólna; pobiera listę funkcji definiujących reguły i wywołuje je w podanej kolejności. Nie ma dla niej znaczenia, w jaki sposób reguły zostały zdefiniowane. Na etapie 2 były to osobno zdefiniowane funkcje nazwane. Na etapie 3 były zdefiniowane jako anonimowe funkcje `lambda`. Teraz, na etapie 4, są one budowane dynamicznie poprzez odwzorowanie listy napisów przy użyciu funkcji `buildMatchAndApplyFunctions`. Nie ma to znaczenia; funkcja `plural` działa cały czas tak samo.

Na wypadek, gdyby jeszcze nie bolała was od tego głowa, przyznam się, że w definicji funkcji `buildMatchAndApplyFunctions` znajduje się pewna subtelność, o której dotychczas nie mówiłem. Wróćmy na chwilę i przyjrzyjmy się bliżej:

Przykład 17.13. Bliższe spotkanie z funkcją buildMatchAndApplyFunctions

```
def buildMatchAndApplyFunctions((pattern, search, replace)):   #(1)
```

1. Zauważyliście podwójne nawiasy? Funkcja ta tak naprawdę nie przyjmuje trzech parametrów; przyjmuje ona dokładnie jeden parametr będący trzyelementową krotką. W momencie wywoływania funkcji, krotka ta jest rozwijana, a trzy elementy krotki są przypisywane do trzech różnych zmiennych o nazwach `pattern`, `search` oraz `replace`. Jesteście już zagubieni? Zobaczmy to w działaniu.

Przykład 17.14. Rozwijanie krotek podczas wywoływania funkcji

```
>>> def foo((a, b, c)):
...     print c
...     print b
...     print a
>>> parameters = ('apple', 'bear', 'catnap')
>>> foo(parameters) #(1)
catnap
bear
apple
```

1. Poprawnym sposobem wywołania funkcji `foo` jest przekazanie jej trzyelementowej krotki. W momencie wywoływania tej funkcji, elementy krotki są przypisywane różnym zmiennym lokalnym wewnątrz funkcji `foo`.

Wróćmy na chwilę do naszego programu i sprawdźmy, dlaczego trik w postaci automatycznego rozwijania krotek był w ogóle potrzebny. Zauważmy, że lista `patterns` to lista krotek, a każda krotka posiada trzy elementy. Wywołanie `map(buildMatchAndApplyFunctions, patterns)` oznacza, że funkcja `buildMatchAndApplyFunctions` nie zostanie wywołana z trzema parametrami. Użycie `map` do odwzorowania listy przy pomocy funkcji zawsze odbywa się przez wywołanie tej funkcji z dokładnie jednym parametrem: każdym elementem listy. W przypadku listy `patterns`, każdy element listy jest krotką, a więc `buildMatchAndApplyFunctions` zawsze jest wywoływana z krotką, przy czym automatyczne rozwinięcie tej krotki zostało zastosowane w definicji funkcji `buildMatchAndApplyFunctions` po to, aby elementy tej krotki zostały automatycznie przypisane do nazwanych zmiennych, z którymi można dalej pracować.

18.6 plural.py, etap 5

Usunęliśmy już z kodu powtórzenia i dodaliśmy wystarczająco dużo abstrakcji, aby reguły zamiany rzeczownika na liczbę mnogą znajdowały się na liście napisów. Następnym logicznym krokiem będzie umieszczenie tych napisów (definiujących reguły) w osobnym pliku, dzięki czemu będzie można utrzymywać listę reguł niezależnie od używającego ją kodu.

Na początku utworzymy plik tekstowy zawierający reguły. Nie ma tu złożonych struktur, to po prostu napisy rozdzielone spacjami (lub znakami tabulacji) w trzech kolumnach. Plik ten nazwiemy rules.en; "en" oznacza "English", reguły te dotyczą bowiem rzeczowników języka angielskiego. Później będziemy mogli dodać pliki z regułami dla innych języków.

Przykład 17.15. rules.en

```
[sxz]$                  $               es
[^aeioudgkprt]h$        $               es
[^aeiou]y$              y$              ies
$                       $               s
```

Zobaczmy, jak możemy użyć tego pliku.

Przykład 17.16. plural5.py

```
import re
import string

def buildRule((pattern, search, replace)):
    return lambda word: re.search(pattern, word) and re.sub(search, replace, word) #(1)

def plural(noun, language='en'):                               #(2)
    lines = file('rules.%s' % language).readlines()            #(3)
    patterns = map(string.split, lines)                        #(4)
    rules = map(buildRule, patterns)                           #(5)
    for rule in rules:
        result = rule(noun)                                    #(6)
        if result: return result
```

1. W dalszym ciągu używamy tutaj techniki dopełnień (dynamicznego budowania funkcji, które używają zmiennych zdefiniowanych na zewnątrz funkcji), jednak teraz połączyliśmy osobne funkcje dopasowującą oraz modyfikującą w jedną. (Powód, dla którego to zrobiliśmy, stanie się jasny w następnym podrozdziale). Pozwoli to nam osiągnąć ten sam cel, jaki osiągaliśmy przy użyciu dwóch funkcji, będziemy jedynie musieli, jak zobaczymy za chwilę, trochę inaczej tę nową funkcję wywołać.

2. Funkcja `plural` pobiera teraz opcjonalny parametr, `language`, który ma domyślną wartość `en`.

3. Parametru `language` używamy do skonstruowania nazwy pliku, następnie otwieramy ten plik i wczytujemy jego zawartość do listy. Jeśli `language` ma wartość `en`, wówczas zostanie otworzony plik `rules.en`, wczytana jego zawartość, podzielona na podstawie znaków nowego wiersza, i zwrócona w postaci listy. Każda linia z pliku zostanie wczytana jako jeden element listy.

4. Jak widzieliśmy wcześniej, każda linia w pliku zawiera trzy wartości, które są oddzielone białymi znakami (spacją lub znakiem tabulacji, nie ma to znaczenia). Odwzorowanie powstałej z wczytania pliku listy przy pomocy funkcji `string.split` pozwoli na utworzenie nowej listy, której elementami są trzyelementowe krotki. Linia taka, jak: "[sxz]$ $ es" zostanie zamieniona w krotkę ('[sxz]$', '$', 'es'). Oznacza to, że w zmiennej `patterns` znajdą się trzyelementowe krotki zawierające napisy, dokładnie tak, jak to wcześniej, na etapie 4, zakodowaliśmy na sztywno.

5. Jeśli `patterns` jest listą krotek, to `rules` będzie listą funkcji zdefiniowanych dynamicznie przy każdym wywołaniu `buildRule`. Wywołanie `buildRule(('[sxz]$', '$', 'es'))` zwróci funkcję, która pobiera jeden parametr, `word`. Kiedy zwrócona funkcja jest wywoływana, zostanie wykonany kod: `re.search('[sxz]$', word) and re.sub('$', 'es', word)`.

6. Ponieważ teraz budujemy funkcję, która łączy w sobie dopasowanie i modyfikację, musimy trochę inaczej ją wywołać. Jeśli wywołamy tę funkcję i ona coś nam zwróci, to będzie to forma rzeczownika w liczbie mnogiej; jeśli zaś nie zwróci nic (zwróci `None`), oznacza to, że reguła dopasowująca nie zdołała dopasować wyrażenia do podanego rzeczownika i należy w tej sytuacji spróbować dopasować kolejną regułę.

Poprawa kodu polegała na tym, że udało nam się całkowicie wyłączyć reguły tworzenia liczby mnogiej do osobnego pliku. Dzięki temu nie tylko będzie można utrzymywać ten plik niezależnie od kodu, lecz także, dzięki wprowadzeniu odpowiedniej notacji nazewniczej, używać funkcji `plural` z różnymi plikami zawierającymi reguły dla różnych języków.

Wadą tego rozwiązania jest fakt, że ilekroć chcemy użyć funkcji `plural`, musimy na nowo wczytywać plik z regułami. Myślałem, że uda mi się napisać tę książkę bez używania frazy: "zostawiam to jako zagadnienie jako ćwiczenie dla czytelników", ale nie mogę się powstrzymać: zbudowanie mechanizmu buforującego dla zależnego od języka pliku z regułami, który automatycznie odświeża się, jeśli plik z regułami zmienił się między wywołaniami, zostawiam jako ćwiczenie dla czytelników. Bawcie się dobrze!

18.7 plural.py, etap 6

Teraz jesteście już gotowi, aby porozmawiać o generatorach.

Przykład 17.17. plural6.py

```
import re

def rules(language):
    for line in file('rules.%s' % language):
        pattern, search, replace = line.split()
        yield lambda word: re.search(pattern, word) and re.sub(search, replace, word)

def plural(noun, language='en'):
    for applyRule in rules(language):
        result = applyRule(noun)
        if result: return result
```

Powyższy kod używa generatora. Nie zacznę nawet tłumaczyć, na czym polega ta technika, dopóki nie przyjrzycie się najpierw prostszemu przykładowi.

Przykład 17.18. Wprowadzenie do generatorów

```
>>> def make_counter(x):
...     print 'entering make_counter'
...     while 1:
...         yield x              #(1)
...         print 'incrementing x'
...         x = x + 1
...
>>> counter = make_counter(2) #(2)
>>> counter                  #(3)
<generator object at 0x001C9C10>
>>> counter.next()           #(4)
entering make_counter
2
>>> counter.next()           #(5)
incrementing x
3
>>> counter.next()           #(6)
incrementing x
4
```

1. Obecność słowa kluczowego `yield` w definicji `make_counter` oznacza, że nie jest to zwykła funkcja. To specjalny rodzaj funkcji, która generuje wartości przy każdym wywołaniu. Możecie myśleć o niej jak o funkcji kontynuującej swoje działanie: wywołanie jej zwraca obiekt generatora, który może zostać użyty do generowania kolejnych wartości x.

2. Aby uzyskać instancję generatora `make_counter`, wystarczy wywołać funkcję `make_counter`. Zauważcie, że nie spowoduje to jeszcze wykonania kodu tej funkcji. Można to stwierdzić na podstawie faktu, że pierwszą instrukcją w tej funkcji jest instrukcja `print`, a nic jeszcze nie zostało wypisane.

3. Funkcja `make_counter` zwraca obiekt będący generatorem.

4. Kiedy po raz pierwszy wywołamy `next()` na obiekcie generatora, zostanie wykonany kod funkcji `make_counter` aż do pierwszej instrukcji `yield`, która spowoduje zwrócenie pewnej wartości. W tym przypadku będzie to wartość 2, ponieważ utworzyliśmy generator wywołując `make_counter(2)`.

5. Kolejne wywołania `next()` na obiekcie generatora spowodują kontynuowanie wykonywania kodu funkcji od miejsca, w którym wykonywanie funkcji zostało przerwane aż do kolejnego napotkania instrukcji `yield`. W tym przypadku następną linią kodu oczekującą na wykonanie jest instrukcja `print`, która wypisuje tekst "incrementing x", a kolejną — instrukcja przypisania $x = x + 1$, która zwiększa wartość x. Następnie wchodzimy w kolejny cykl pętli `while`, w którym wykonywana jest instrukcja `yield`, zwracająca bieżącą wartość x (obecnie jest to 3).

6. Kolejne wywołanie `counter.next` spowoduje wykonanie tych samych instrukcji, przy czym tym razem x osiągnie wartość 4. I tak dalej. Ponieważ `make_counter` zawiera nieskończoną pętlę, więc teoretycznie moglibyśmy robić to w nieskończoność: generator zwiększałby wartość x i wypluwał jego bieżącą wartość. Spójrzmy jednak na bardziej produktywne przypadki użycia generatorów.

Przykład 17.19. Użycie generatorów w miejsce rekurencji

```
def fibonacci(max):
    a, b = 0, 1       #(1)
    while a < max:
        yield a       #(2)
        a, b = b, a+b #(3)
```

1. Ciąg Fibonacciego składa się z liczb, z których każda jest sumą dwóch poprzednich (za wyjątkiem dwóch pierwszych liczb tego ciągu). Ciąg rozpoczynają liczby 0 i 1, kolejne wartości rosną powoli, następnie różnice między nimi coraz szybciej się zwiększają. Aby rozpocząć generowanie tego ciągu, potrzebujemy dwóch zmiennych: a ma wartość 0, b ma wartość 1.

2. a to bieżąca wartość w ciągu, więc zwracamy ją

3. b jest kolejną wartością, więc przypisujemy ją do a, jednocześnie obliczając kolejną wartość (a+b) i przypisując ją do b do późniejszego użycia. Zauważcie, że dzieje się to równolegle; jeśli a ma wartość 3 a b ma wartość 5, wówczas w wyniku wartościowania wyrażenia a, b = b, a+b do zmiennej a zostanie przypisana wartość 5 (wcześniejsza wartość b), a do b wartość 8 (suma poprzednich wartości a i b).

Mamy więc funkcję, która wyrzuca z siebie kolejne wartości ciągu Fibonacciego. Oczywiście moglibyśmy to samo osiągnąć przy pomocy rekurencji, jednak ten sposób jest znacznie prostszy w zapisie. No i świetnie się sprawdza w przypadku pętli:

Przykład 17.20. Generatory w pętlach

```
>>> for n in fibonacci(1000): #(1)
...     print n,              #(2)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

1. Generatory (takie jak `fibonacci`) mogą być używane bezpośrednio w pętlach. Pętla `for` utworzy obiekt generatora i będzie wywoływać na nim metodę `next()`, przypisując ją do zmiennej sterującej pętli (n).

2. W każdym przebiegu pętli `for` n będzie miało nową wartość, zwróconą przez generator w instrukcji `yield` funkcji `fibonacci`, i ta wartość zostanie wypisana. Jeśli generatorowi `fibonacci` skończą się generowane wartości (zmienna a przekroczy max, które w tym przypadku jest równe 1000), pętla `for` zakończy działanie.

OK, wróćmy teraz do funkcji `plural` i sprawdźmy, jak tam został użyty generator.

Przykład 17.21. Generator tworzący dynamicznie funkcje

```
def rules(language):
    for line in file('rules.%s' % language):                                       #(1)
        pattern, search, replace = line.split()                                    #(2)
        yield lambda word: re.search(pattern, word) and re.sub(search, replace, word) #(3)

def plural(noun, language='en'):
    for applyRule in rules(language):  #(4)
        result = applyRule(noun)
        if result: return result
```

1. `for line in file(...)` to często spotykany w języku Python idiom służący wczytaniu po jednej wszystkich linii z pliku. Działa on w ten sposób, że `file` zwraca generator, którego metoda `next()` zwraca kolejną linię z pliku. To szalenie fajna sprawa, robię się cały mokry, kiedy tylko o tym pomyślę.

2. Tu nie ma żadnej magii. Pamiętajcie, że każda linia w pliku z regułami zawiera trzy wartości oddzielone białym znakiem, a więc `line.split` zwróci trzyelementową krotkę, a jej trzy wartości są tu przypisywane do trzech zmiennych lokalnych.

3. A teraz następuje instrukcja `yield`. Co zwraca `yield`? Zwraca ona funkcję zbudowaną dynamicznie przy użyciu notacji `lambda`, będącą w rzeczywistości domknięciem (używa bowiem zmiennych lokalnych `pattern`, `search` oraz `replace` jako stałych). Innymi słowy, `rules` jest generatorem który generuje funkcje reprezentujące reguły.

4. Jeśli `rules` jest generatorem, to znaczy, że możemy użyć go bezpośrednio w pętli `for`. W pierwszym przebiegu pętli wywołanie funkcji `rules` spowoduje otworzenie pliku z regułami, przeczytanie pierwszej linijki oraz dynamiczne zbudowanie funkcji, która dopasowuje i modyfikuje na podstawie pierwszej odczytanej reguły z pliku, po czym nastąpi zwrócenie zbudowanej w ten sposób funkcji w instrukcji `yield`. W drugim przebiegu pętli `for` wykonywanie funkcji `rules` rozpocznie się tam, gdzie się ostatnio zakończyło (a więc w środku pętli `for line in file(...)`), zostanie więc odczytana druga linia z pliku z regułami, zostanie dynamicznie zbudowana inna funkcja dopasowująca i modyfikująca na podstawie zapisanej w pliku reguły, po czym ta funkcja zostanie zwrócona w instrukcji `yield`. I tak dalej.

Co takiego osiągnęliśmy w porównaniu z etapem 5? W etapie 5 wczytywaliśmy cały plik z regułami i budowaliśmy listę wszystkich możliwych reguł zanim nawet wypróbowaliśmy pierwszą z nich. Teraz, przy użyciu generatora, podchodzimy do sprawy w sposób leniwy: otwieramy plik i wczytujemy pierwszą linię w celu zbudowania funkcji, jednak jeśli ta funkcja zadziała (reguła zostanie dopasowana, a rzeczownik zmodyfikowany), nie będziemy niepotrzebnie czytać dalej linii z pliku ani tworzyć jakichkolwiek innych funkcji.

18.8 Funkcje dynamiczne - podsumowanie

W tym rozdziale rozmawialiśmy o wielu zaawansowanych technikach programowania. Należy pamiętać, że nie wszystkie nadają się do stosowania w każdej sytuacji.

Powinniście teraz dobrze orientować się w następujących technikach:

- Zastępowanie napisów przy pomocy wyrażeń regularnych.
- Traktowanie funkcji jak obiektów, przechowywanie ich na listach, przypisywanie ich do zmiennych i wywoływanie ich przy pomocy tych zmiennych.
- Budowanie funkcji dynamicznych przy użyciu notacji `lambda`.
- Budowanie dopełnień — funkcji dynamicznych, których definicja używa otaczających je zmiennych w charakterze wartości stałych.
- Budowanie generatorów — funkcji, które można kontynuować, dzięki którym realizowana jest pewna przyrostowa logika, a za każdym ich wywołaniem może zostać zwrócona inna wartość.

Dodawanie abstrakcji, dynamiczne budowanie funkcji, tworzenie domknięć i używanie generatorów może znacznie uprościć kod, może sprawić, że stanie się on czytelniejszy i bardziej elastyczny. Może jednak sprawić, że ten sam kod stanie się później znacznie trudniejszy do debugowania. Do was należy znalezienie właściwej równowagi między prostotą i mocą.

Rozdział 19

Optymalizacja szybkości

19.1 Optymalizacja szybkości

Optymalizacja szybkości jest niesamowitą sprawą. Tylko dlatego, że Python jest językiem interpretowanym, nie oznacza, że nie powinniśmy martwić się o optymalizację pod kątem szybkości działania. Jednak nie należy się tym aż tak bardzo przejmować.

Nurkujemy

Istnieje sporo pułapek związanych z optymalizacją kodu, ciężko stwierdzić, od czego należałoby zacząć.

Zacznijmy więc tutaj: czy jesteśmy pewni, że w ogóle powinniśmy się do tego zabierać? Czy nasz kod jest na prawdę tak kiepski? Czy warto poświęcić na to czas? Ile tak na prawdę czasu podczas działania aplikacji będzie zajmowało wykonywanie kodu, w porównaniu z czasem poświęconym na oczekiwanie na odpowiedź zdalnej bazy danych czy też akcję użytkownika?

Po drugie, czy jesteśmy pewni, że skończyliśmy kodowanie? Przedwczesna optymalizacja jest jak nakładanie lodowej polewy na w pół upieczone ciasto. Poświęcamy godziny czy nawet dni na optymalizację kodu pod kątem wydajności, aby w końcu przekonać się, że nie robi on tego, co byśmy chcieli. Stracony czas, robota wyrzucona w błoto.

Nie chodzi o to, że optymalizacja kodu jest bezwartościowa, raczej powinniśmy spojrzeć z dystansu na cały system aby nabrać przeświadczenia, czy czas przeznaczony na nią jest najlepszą inwestycją. Każda minuta poświęcona na optymalizację kodu jest minutą, której nie poświęcamy na dodawanie nowych funkcjonalności, pisanie dokumentacji, zabawę z naszymi dziećmi czy też pisanie testów jednostkowych.

A no właśnie, testy jednostkowe. Nie powinniśmy nawet zaczynać optymalizacji nie mając pełnego zestawu takich testów. Ostatnią rzeczą jakiej byśmy chcieli to pojawienie się nowego błędu podczas dłubania w algorytmie.

Mając na uwadze powyższe porady, zerknijmy na niektóre techniki optymalizacji kodu Pythona. Nasz kod to implementacja algorytmu Soundex. Soundex był metodą kategoryzowania nazwisk w spisie ludności w Stanach Zjednoczonych na początku 20 wieku. Grupował on podobnie brzmiące słowa, przez co naukowcy mieli szansę na odnalezienie nawet błędnie zapisanych nazwisk. Soundex jest do dziś używany głównie z tego samego powodu, lecz oczywiści w tym celu używane są skomputeryzowane bazy danych. Większość silników baz danych posiada funkcję Soundex.

Istnieje kilka nieco różniących się od siebie wersji algorytmu Soundex. Poniżej znajduje się ta, której używamy w tym rozdziale:

1. Weź pierwszą literę nazwy.
2. Przekształć pozostałe litery na cyfry według poniższej listy:

- B, F, P, oraz V stają się 1.
- C, G, J, K, Q, S, X, oraz Z stają się 2.
- D oraz T stają się 3.
- L staje się 4.
- M oraz N stają się 5.
- R staje się 6.

- Wszystkie pozostałe litery stają się 9.

1. Usuń wszystkie duplikaty.
2. Usuń wszystkie dziewiątki.
3. Jeśli wynik jest krótszy niż cztery znaki (pierwsza litera plus trzy cyfry), dopełnij wynik zerami z prawej strony ciągu do czterech znaków.
4. Jeżeli wynik jest dłuższy niż cztery znaki, pozbądź się wszystkiego po czwartym znaku.

Na przykład, moje imię, Pilgrim, staje się P942695. Nie posiada następujących po sobie duplikatów, więc nic tutaj nie robimy. Następnie usuwamy 9tki pozostawiając P4265. Jednak znaków jest za dużo, więc pomijamy ich nadmiar w wyniku otrzymując P426.

Inny przykład: Woo staje się W99, które staje się W9, które staje się W, które natomiast dopełniamy zerami z prawej strony do czterech znaków aby otrzymać W000.

A oto pierwsze podejście do funkcji Soundex:

Przykład 18.1. soundex/stage1/soundex1a.py

```
import string, re

charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}

def soundex(source):
```

```
    "convert string to Soundex equivalent"

    # Soundex requirements:
    # source string must be at least 1 character
    # and must consist entirely of letters
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"

    # Soundex algorithm:
    # 1. make first character uppercase
    source = source[0].upper() + source[1:]

    # 2. translate all other characters to Soundex digits
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]

    # 3. remove consecutive duplicates
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d

    # 4. remove all "9"s
    digits3 = re.sub('9', , digits2)

    # 5. pad end with "0"s to 4 characters
    while len(digits3) < 4:
        digits3 += "0"

    # 6. return first 4 characters
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

19.2 Korzystanie z modułu timeit

Najważniejszą rzeczą, jaką powinniście wiedzieć na temat optymalizacji kodu w języku Python jest to, że nie powinniście pisać własnej funkcji mierzącej czas wykonania kodu.

Pomiar czasu wykonania niewielkich fragmentów kodu to zagadnienie niezwykle złożone. Jak dużo czasu procesora zużywa wasz komputer podczas działania tego kodu? Czy istnieją jakieś inne procesy działające w tym samym czasie w tle? Czy jesteście tego pewni? Na każdym nowoczesnym komputerze działają jakieś procesy w tle, niektóre przez cały czas a niektóre okresowo. Zadania w cronie zostają uruchomione w dokładnie określonych porach; usługi działające w tle co jakiś czas "budzą się", aby wykonać różne pożyteczne prace, takie jak sprawdzenie nowej poczty, połączenie się z serwerami typu "instant messaging", sprawdzanie, czy istnieją już aktualizacje zainstalowanych programów, skanowanie antywirusowe, sprawdzanie, czy w ciągu ostatnich 100 nanosekund do napędu CD została włożona płyta i tak dalej. Zanim rozpoczniecie testy z pomiarami czasu, wyłączcie wszystko i odłączcie komputer od sieci. Następnie wyłączcie to wszystko, o czym zapomnieliście za pierwszym razem, później wyłączcie usługę, która sprawdza, czy nie przyłączyliście się ponownie do sieci, a następnie...

Do tego dochodzą jeszcze różne aspekty wprowadzane przez mechanizm pomiaru czasu. Czy interpreter języka Python zapamiętuje raz wyszukane nazwy metod? Czy zapamiętuje bloki skompilowanego kodu? Wyrażenia regularne? Czy w waszym kodzie objawią się jakieś skutki uboczne, gdy uruchomicie go więcej niż jeden raz? Nie zapominajcie, że mamy tutaj do czynienia z małymi ułamkami sekundy, więc małe błędy w mechanizmie pomiaru czasu przełożą się na nienaprawialne zafałszowanie rezultatów tych pomiarów.

W społeczności Pythona funkcjonuje powiedzenie: "Python z bateriami w zestawie". Nie piszcie własnego mechanizmu mierzącego czas. Python 2.3 posiada służący do tego celu doskonały moduł o nazwie `timeit`.

Przykład 18.2. Wprowadzenie do modułu `timeit`

```
>>> import timeit
>>> t = timeit.Timer("soundex.soundex('Pilgrim')",
...     "import soundex")   #(1)
>>> t.timeit()              #(2)
8.21683733547
>>> t.repeat(3, 2000000)    #(3)
[16.48319309109, 16.46128984923, 16.44203948912]
```

1. W module `timeit` zdefiniowana jest jedna klasa, `Timer`, której konstruktor przyjmuje dwa argumenty. Obydwa argumenty są napisami. Pierwszy z nich to instrukcja, której czas wykonania chcemy zmierzyć; w tym przypadku mierzymy czas wywołania funkcji `soundex` z modułu `soundex` z parametrem 'Pilgrim'. Drugi argument przekazywany do konstruktora obiektu Timer to instrukcja `import`, która ma przygotować środowisko do wykonywania instrukcji, której czas trwania mierzymy. Wewnętrzne działanie `timeit` polega na przygotowaniu wirtualnego, wyizolowanego środowiska, wykonaniu instrukcji przygotowujących (zaimportowaniu modułu `soundex`) oraz kompilacji i wykonaniu instrukcji poddawanej pomiarowi (wywołanie funkcji `soundex`).

2. Najłatwiejsza rzecz, jaką można zrobić z obiektem klasy `Timer`, to wywołanie na nim metody `timeit()`, która wywołuje podaną funkcję milion razy i zwraca

liczbę sekund, jaką zajęło to wywoływanie.

3. Inną ważną metodą obiektu `Timer` jest `repeat()`, która pobiera dwa opcjonalne argumenty. Pierwszy argument to liczba określająca ile razy ma być powtórzony cały test; drugi argument to liczba określająca ile razy ma zostać wykonana mierzona instrukcja w ramach jednego testu. Obydwa argumenty są opcjonalne, a ich wartości domyślne to, odpowiednio, 3 oraz 1000000. Metoda `repeat()` zwraca listę zawierającą czas wykonania każdego testu wyrażony w sekundach.

Moduł `timeit` można również wykorzystać, aby z linii poleceń zmierzyć czas trwania dowolnego programu napisanego w języku Python, bez konieczności modyfikowania jego źródeł. Opcje linii poleceń dostępne są w dokumentacji modułu timeit.

Zauważcie, że `repeat()` zwraca listę czasów. Czasy te prawie nigdy nie będą identyczne; jest to spowodowane faktem, że interpreter języka Python nie zawsze otrzymuje taką samą ilość czasu procesora (oraz istnieniem różnych procesów w tle, których nie tak łatwo się pozbyć). Wasza pierwsza myśl może być taka: "Obliczmy średnią i uzyskajmy Prawdziwą Wartość".

W rzeczywistości takie podejście jest prawie na pewno złe. Testy, które trwały dłużej, nie trwały dłużej z powodu odchyleń w waszym kodzie albo kodzie interpretera języka; trwały dłużej, ponieważ w tym samym czasie w systemie działały w tle inne procesy albo z powodów niezależnych od interpretera języka, których nie można było całkowicie wyeliminować. Jeśli różnice w pomiarach czasu dla różnych testów różnią się o więcej niż kilka procent, wówczas są one zbyt znaczne, aby ufać takim pomiarom. W przeciwnym przypadku jako czas trwania testu należy przyjąć wartość najmniejszą, a inne wartości odrzucić.

Python posiada użyteczną funkcję `min`, która zwraca najmniejszą wartość z podanej w parametrze listy, a którą można w tym przypadku wykorzystać:

```
>>> min(t.repeat(3, 1000000))
8.22203948912
```

19.3 Optymalizacja wyrażeń regularnych

Pierwszą rzeczą, jaką musi sprawdzić funkcja `Soundex`, jest to, czy na wejściu znajduje się niepusty ciąg znaków. Jak można to najlepiej zrobić?

Jeśli odpowiedzieliście: "używając wyrażeń regularnych", to idziecie do kąta i kontemplujecie tam swoje złe instynkty. "Wyrażenia regularne" prawie nigdy nie stanowią dobrej odpowiedzi; jeśli to możliwe, należy ich raczej unikać. Nie tylko ze względu na wydajność, lecz także z tego prostego powodu, że są one niezwykle trudne w debugowaniu i w dalszym utrzymaniu. Ale również ze względu na wydajność.

Poniższy fragment kodu pochodzi z programu soundex/stage1/soundex1a.py i sprawdza, czy zmienna `source` będąca argumentem funkcji jest napisem zbudowanym wyłącznie z liter, przy czym zawiera co najmniej jedną literę (nie jest pustym napisem):

```
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"
```

Czy soundex1a.py to wydajny program? Dla ułatwienia, w sekcji `__main__` skryptu znajduje się poniższy kod, który wywołuje moduł `timeit`, konstruuje test do pomiaru złożony z trzech różnych napisów, testuje każdy napis trzy razy i dla każdego z napisów wyświetla minimalny czas:

```
if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

A więc czy soundex1a.py używający wyrażeń regularnych jest wydajny?

```
C:\samples\soundex\stage1>python soundex1a.py
Woo             W000 19.3356647283
Pilgrim         P426 24.0772053431
Flingjingwaller F452 35.0463220884
```

Możemy się spodziewać, algorytm ten będzie działał znacznie dłużej, jeśli zostanie on wywołany ze znacznie dłuższymi napisami. Możemy zrobić wiele, aby zmniejszyć tę szczelinę (sprawić, że funkcja będzie potrzebowała względnie mniej czasu dla dłuższych danych wejściowych), jednak natura tego algorytmu wskazuje na to, że nie będzie on nigdy działał w stałym czasie.

Warto mieć na uwadze, że testujemy reprezentatywną próbkę imion. Woo to przypadek trywialny; to imię zostanie skrócone do jednej litery i uzupełnione zerami. Pilgrim to średnio złożony przypadek, posiadający średnią długość i zawierający zarówno litery znaczące, jak i te, które zostaną zignorowane. Flingjingwaller zaś to wyjątkowo długie imię zawierające następujące po sobie powtórzenia. Inne testy mogłyby również być przydatne, jednak te trzy pokrywają duży zakres przypadków.

Co zatem z wyrażeniem regularnym? Cóż, okazało się ono nieefektywne. Ponieważ wyrażenie sprawdza zakresy znaków (duże litery A-Z oraz małe a-z), możemy użyć skróconej składni wyrażeń regularnych. Poniżej soundex/stage1/soundex1b.py:

```
    if not re.search('^[A-Za-z]+$', source):
        return "0000"
```

Moduł `timeit` mówi, że soundex1b.py jest odrobinę szybszy, niż soundex1a.py, nie ma w tym jednak nic, czy można by się ekscytować:

```
C:\samples\soundex\stage1>python soundex1b.py
Woo              W000 17.1361133887
Pilgrim          P426 21.8201693232
Flingjingwaller F452 32.7262294509
```

Jak widzieliśmy w podrozdziale 15.3 [Python/Refaktoryzacja], wyrażenia regularne mogą zostać skompilowane i użyte powtórnie, dzięki czemu osiąga się lepsze rezultaty. Ze względu na to, że nasze wyrażenie regularne nie zmienia się między wywołaniami, kompilujemy je i używamy wersji skompilowanej. Poniżej znajduje się fragment soundex/stage1/soundex1c.py:

```
isOnlyChars = re.compile('^[A-Za-z]+$').search
def soundex(source):
    if not isOnlyChars(source):
        return "0000"
```

Użycie skompilowanej wersji wyrażenia regularnego jest już znacznie szybsze:

```
C:\samples\soundex\stage1>python soundex1c.py
Woo              W000 14.5348347346
Pilgrim          P426 19.2784703084
Flingjingwaller F452 30.0893873383
```

Ale czy to nie jest przypadkiem błędna ścieżka? Logika jest przecież prosta: napis wejściowy nie może być pusty i musi być cały złożony z liter. Czy nie byłoby szybciej, gdybyśmy pozbyli się wyrażenia regularnego i zapisali pętlę, która sprawdza każdy znak?

Poniżej soundex/stage1/soundex1d.py:

```
    if not source:
        return "0000"
    for c in source:
        if not ('A' <= c <= 'Z') and not ('a' <= c <= 'z'):
            return "0000"
```

Okazuje się, że ta technika w soundex1d.py nie jest szybsza, niż użycie skompilowanego wyrażenia regularnego (ale jest szybsza, niż użycie nieskompilowanego wyrażenia regularnego):

```
C:\samples\soundex\stage1>python soundex1d.py
Woo              W000 15.4065058548
Pilgrim          P426 22.2753567842
Flingjingwaller F452 37.5845122774
```

Dlaczego soundex1d.py nie jest szybszy? Odpowiedź leży w naturze języka Python, który jest językiem interpretowanym. Silnik wyrażeń regularnych napisany jest w C i skompilowany odpowiednio do architektury waszego komputera. Z drugiej strony, pętla napisana jest w języku Python i jest uruchamiana przez interpreter języka Python. Choć pętla jest stosunkowo prosta, jej prostota nie wystarcza, aby pozbyć się narzutu związanego z tym, że jest ona interpretowana. Wyrażenia regularne nigdy nie są dobrym rozwiązaniem... chyba, że akurat są.

Okazuje się, że Python posiada pewną starą metodę w klasie string. Jesteście całkowicie usprawiedliwieni, jeśli o niej nie wiedzieliście, bowiem nie wspominałem jeszcze o niej w tej książce. Metoda nazywa się `isalpha()` i sprawdza, czy napis składa się wyłącznie z liter.

Oto soundex/stage1/soundex1e.py:

```
if (not source) and (not source.isalpha()):
    return "0000"
```

Czy zyskaliśmy coś używając tej specyficznej metody w soundex1e.py? Trochę zyskaliśmy.

```
C:\samples\soundex\stage1>python soundex1e.py
Woo             W000 13.5069504644
Pilgrim         P426 18.2199394057
Flingjingwaller F452 28.9975225902
```

Przykład 18.3. Dotychczas najlepszy rezultat: soundex/stage1/soundex1e.py

```
import string, re

charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
```

```
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}

def soundex(source):
    if (not source) and (not source.isalpha()):
        return "0000"
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', , digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

19.4 Optymalizacja przeszukiwania słownika

Kolejnym krokiem w algorytmie Soundex jest przekształcenie znaków w cyfry według pewnego szczególnego wzorca. Jaki jest najlepszy sposób, aby to zrobić?

Najbardziej oczywiste rozwiązanie polega na zdefiniowaniu słownika, w którym poszczególne znaki są kluczami, a wartościami — odpowiadające im cyfry, a następnie przeszukiwaniu słownika dla każdego pojawiającego się znaku. Tak właśnie działa program soundex/stage1/soundex1c.py (który osiąga najlepsze jak do tej pory rezultaty wydajnościowe):

```
charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}

def soundex(source):
    # ... input check omitted for brevity ...
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
```

Mierzyliśmy już czas wykonania soundex1c.py; oto, jak się przestawiają pomiary:

```
C:\samples\soundex\stage1>python soundex1c.py
Woo             W000 14.5341678901
Pilgrim         P426 19.2650071448
Flingjingwaller F452 30.1003563302
```

Powyższy kod jest prosty, ale czy stanowi najlepsze możliwe rozwiązanie? Nieefektywne wydaje się w szczególności wywoływanie na każdym znaku metody `upper()`; zrobilibyśmy prawdopodobnie lepiej wywołując `upper()` na całym napisie wejściowym.

Dochodzi do tego kwestia przyrostowego budowania napisu złożonego z cyfr. Takie przyrostowe budowanie napisu jest niezwykle niewydajne; wewnętrznie interpreter języka musi w każdym przebiegu pętli utworzyć nowy napis, a następnie zwolnić stary.

Wiadomo jednak, że Python świetnie sobie radzi z listami. Może automatycznie potraktować napis jako listę znaków. Listę zaś łatwo przekształcić z powrotem do napisu przy użyciu metody `join()`.

Poniżej soundex/stage2/soundex2a.py, który konwertuje litery na cyfry przy użyciu metody `join` i notacji `lambda`:

```
def soundex(source):
    # ...
    source = source.upper()
    digits = source[0] + "".join(map(lambda c: charToSoundex[c], source[1:]))
```

Co ciekawe, program soundex2a.py nie jest wcale szybszy:

```
C:\samples\soundex\stage2>python soundex2a.py
Woo             W000 15.0097526362
Pilgrim         P426 19.254806407
Flingjingwaller F452 29.3790847719
```

Narzut wprowadzony przez funkcję anonimową utworzoną przy pomocy notacji `lambda` przekroczył cały zysk wydajności, jaki osiągnęliśmy używając napisu w charakterze listy znaków.

Program soundex/stage2/soundex2b.py w miejsce notacji `lambda` używa wyrażeń listowych:

```
    source = source.upper()
    digits = source[0] + "".join([charToSoundex[c] for c in source[1:]])
```

Użycie wyrażeń listowych w soundex2b.py jest szybsze niż używanie notacji `lambda`, ale wciąż nie jest szybsze, niż oryginalny kod (soundex1c.py, w którym napis wynikowy jest budowany przyrostowo):

```
C:\samples\soundex\stage2>python soundex2b.py
Woo             W000 13.4221324219
Pilgrim         P426 16.4901234654
Flingjingwaller F452 25.8186157738
```

Nadszedł czas na wprowadzenie radykalnej zmiany w naszym podejściu. Przeszukiwanie słownika to narzędzie ogólnego zastosowania. Kluczami w słownikach mogą być napisy dowolnej długości (oraz wiele innych typów danych), jednak w tym przypadku posługiwaliśmy sie jedynie napisami o długości jednego znaku, zarówno w charakterze klucza, jak i wartości. Okazuje się, że język Python posiada specjalizowaną funkcję służącą do obsługi dokładnie tej sytuacji, funkcję o nazwie `string.maketrans`.

Oto program soundex/stage2/soundex2c.py:

```
allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
def soundex(source):
    # ...
    digits = source[0].upper() + source[1:].translate(charToSoundex)
```

Co właściwie się tu dzieje? Funkcja `string.maketrans` tworzy wektor przekształceń między dwoma napisami: pierwszym i drugim argumentem. W tym przypadku pierwszym argumentem jest napis ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz, a drugim napis 9123912992245591262391929291239129922455912623919292. Rozpoznajecie ten wzorzec? To ten sam, którego używaliśmy w przypadku słownika: A jest przekształcane do 9, B do 1, C do 2 i tak dalej. Nie jest to jednak słownik; to specjalizowana struktura danych, do której mamy dostęp przy użyciu metody `translate()`, przekształcającej każdy znak argumentu w odpowiadającą mu cyfrę, zgodnie z wektorem przekształceń zdefiniowanym w `string.maketrans`.

Moduł `timeit` pokazuje, że program soundex2c.py jest znacznie szybszy niż ten, w którym definiowaliśmy słownik, iterowaliśmy w pętli po napisie wejściowym i budowaliśmy przyrostowo napis wyjściowy:

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 11.437645008
Pilgrim         P426 13.2825062962
Flingjingwaller F452 18.5570110168
```

Nie uda się nam osiągnąć o wiele lepszych rezultatów niż pokazany wyżej. Specjalizowana funkcja języka Python robi dokładnie to, czego potrzebujemy; używamy więc jej i idziemy dalej.

Przykład 18.4. Dotychczas najlepszy rezultat: soundex/stage2/soundex2c.py

```
import string, re

allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
isOnlyChars = re.compile('^[A-Za-z]+$').search

def soundex(source):
    if not isOnlyChars(source):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', , digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
```

```
names = ('Woo', 'Pilgrim', 'Flingjingwaller')
for name in names:
    statement = "soundex('%s')" % name
    t = Timer(statement, "from __main__ import soundex")
    print name.ljust(15), soundex(name), min(t.repeat())
```

19.5 Optymalizacja operacji na listach

Optymalizacja operacji na listach

Trzecim krokiem a optymalizacji algorytmu `soundex` jest eliminacja kolejnych powtarzających się cyfr. Jak najlepiej to zrobić?

Taki kod otrzymaliśmy dotąd, znajduje się on w `soundex/stage2/soundex2c.py`:

```
digits2 = digits[0]
for d in digits[1:]:
    if digits2[-1] != d:
        digits2 += d
```

Takie wyniki wydajnościowe otrzymujemy dla `soundex2c.py`:

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 12.6070768771
Pilgrim         P426 14.4033353401
Flingjingwaller F452 19.7774882003
```

Pierwszą rzeczą do rozważenia jest efektywność kontrola `digits[-1]` w każdej iteracji pętli. Czy indeksowanie listy jest kosztowne? A może lepiej przechowywać ostatnią cyfrę w oddzielnej zmiennej i sprawdzać nią zamiast listy?

Odpowiedź na to pytanie pomoże nam znaleźć `soundex/stage3/soundex3a.py`:

```
digits2 =
last_digit =
for d in digits:
    if d != last_digit:
        digits2 += d
        last_digit = d
```

`soundex3a.py` nie działa ani trochę szybciej niż `soundex2c.py`, a nawet może być troszeczkę wolniejsze (aczkolwiek jest to za mała różnica, aby coś powiedzieć pewnie):

```
C:\samples\soundex\stage3>python soundex3a.py
Woo             W000 11.5346048171
Pilgrim         P426 13.3950636184
Flingjingwaller F452 18.6108927252
```

Dlaczego `soundex3a.py` nie jest szybsze? Okazuje się, że indeksowanie list w Pythonie jest ekstremalnie efektywne. Powtarzenie dostępu do digits2[-1] nie stanowi w ogóle problemu. Z innej strony, kiedy manualnie zarządzamy ostatnią cyfrą w oddzielnej zmiennej, korzystamy z dwóch przypisań do zmiennych dla każdej przechowywanej cyfry, a te operacje zastępują mały koszt związany z korzystania z listy.

Spróbujemy teraz czegoś radykalnie innego. Jest możliwe, aby traktować dany napis jako listę znaków, zatem można by było wykorzystać wyrażenie listowe, aby przeiterować listę znaków. Jednak występuje problem związany z tym, że potrzebujemy dostępu do poprzedniego znaku w liście, a to nie jest proste w przypadku prostych wyrażeń listowych.

Jakkolwiek jest możliwe tworzenie listy liczbowych indeksów za pomocą wbudowanej funkcji `range()`, a następnie wykorzystać te indeksy do stopniowego przeszukiwania listy i wkładania każdego znaku różnego od znaku poprzedzającego. Dzięki temu

otrzymamy listę znaków, a następnie możemy wykorzystać metodę łańcucha znaków `join()`, a by zrekonstruować z tego listę.

Poniżej mamy `soundex/stage3/soundex3b.py`:

```
    digits2 = "".join([digits[i] for i in range(len(digits))
                       if i == 0 or digits[i-1] != digits[i]])
```

Czy jest to szybsze? Jednym słowem, nie.

```
C:\samples\soundex\stage3>python soundex3b.py
Woo             W000 14.2245271396
Pilgrim         P426 17.8337165757
Flingjingwaller F452 25.9954005327
```

Być może szybkie techniki powinny się skupiać wokół łańcuchów znaków. Python może konwertować łańcuch znaków na listę znaków za pomocą jednego polecenia: `list('abc')`, które zwróci `['a', 'b', 'c']`. Ponadto listy mogą być bardzo szybko modyfikowane w miejscu. Zamiast zwiększać liczbę tworzonych nowych list (lub łańcuchów znaków) z naszego początkowego łańcucha, dlaczego by nie przenieść wszystkich elementów do pojedynczej listy?

Poniżej przedstawiono `soundex/stage3/soundex3c.py`, który modyfikuje listę w miejscu i usuwa kolejno duplujące się elementy:

```
    digits = list(source[0].upper() + source[1:].translate(charToSoundex))
    i=0
    for item in digits:
        if item==digits[i]: continue
        i+=1
        digits[i]=item
    del digits[i+1:]
    digits2 = "".join(digits)
```

Czy jest to szybsze od `soundex3a.py` lub `soundex3b.py`? Nie, w rzeczywistości jest to jeszcze wolniejsze:

```
C:\samples\soundex\stage3>python soundex3c.py
Woo             W000 14.1662554878
Pilgrim         P426 16.0397885765
Flingjingwaller F452 22.1789341942
```

Ciągle nie wykonaliśmy tutaj żadnego podstępu, z wyjątkiem tego, że wykorzystaliśmy i wypróbowaliśmy kilka "mądrych" technik. Najszybszym kodem, który jak dotąd widzieliśmy nadal pozostał oryginał, najbardziej prosta metoda (`soundex2c.py`). Czasami nie popłaca być mądrym.

Przykład 18.5 Najlepszy wynik do tej pory: `soundex/stage2/soundex2c.py`

```
import string, re

allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
isOnlyChars = re.compile('^[A-Za-z]+$').search
```

```
def soundex(source):
    if not isOnlyChars(source):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', , digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

19.6 Optymalizacja operacji na napisach

Ostatnim krokiem algorytmu Soundex jest dopełnienie krótkich napisów wynikowych zerami oraz przycięcie napisów zbyt długich. W jaki sposób można to zrobić najlepiej?

Dotychczasowy kod w programie soundex/stage2/soundex2c.py wygląda następująco:

```
digits3 = re.sub('9', '', digits2)
     while len(digits3) < 4:
         digits3 += "0"
     return digits3[:4]
```

Oto rezultaty pomiarów wydajności w soundex2c.py:

```
C:\samples\soundex\stage2>python soundex2c.py
Woo              W000 12.6070768771
Pilgrim          P426 14.4033353401
Flingjingwaller F452 19.7774882003
```

Pierwszą rzeczą, jaką należy rozważyć, jest zastąpienie wyrażenia regularnego pętlą. Oto kod programu soundex/stage4/soundex4a.py:

```
digits3 = ''
     for d in digits2:
         if d != '9':
             digits3 += d
```

Czy soundex4a.py jest szybszy? Oczywiście:

```
C:\samples\soundex\stage4>python soundex4a.py
Woo              W000 6.62865531792
Pilgrim          P426 9.02247576158
Flingjingwaller F452 13.6328416042
```

Czekajcie chwilę. Pętla, która usuwa znaki z napisu? Możemy użyć do tego prostej metody z klasy `string`. Oto soundex/stage4/soundex4b.py:

```
    digits3 = digits2.replace('9', )
```

Czy soundex4b.py jest szybszy? To interesujące pytanie. Zależy to od dancyh wejściowych:

```
C:\samples\soundex\stage4>python soundex4b.py
Woo              W000 6.75477414029
Pilgrim          P426 7.56652144337
Flingjingwaller F452 10.8727729362
```

Dla większości nazw z programu soundex4b.py metoda z klasy `string` jest szybsza niż pętla, jest jednak odrobinę wolniejsza niż soundex4a.py dla przypadku trywialnego (dla bardzo krótkiej nazwy). Optymalizacje wydajnościowe nie zawsze są jednorodne; poprawki, które sprawią, że w pewnych przypadkach program będzie działał szybciej, mogą sprawić, że w innych przypadkach ten sam program będzie działał wolniej. W

naszym programie uzyskujemy poprawę dla większości przypadków, więc zostawimy tę poprawkę, warto jednak na przyszłość mieć tę ważną zasadę na uwadze.

Na koniec, choć to wcale nie jest sprawa najmniejszej wagi, prześledźmy dwa ostatnie kroki algorytmu: uzupełnianie zerami krótkich napisów wynikowych oraz przycinanie napisów zbyt długich do czterech znaków. Kod, który widzicie w soundex4b.py robi dokładnie to, co trzeba, jednak robi to w bardzo niewydajny sposób. Spójrzmy na soundex/stage4/soundex4c.py, aby przekonać się, dlaczego tak się dzieje.

```
digits3 += '000'
     return digits3[:4]
```

Dlaczego potrzebujemy pętli `while` do wyrównania napisu wynikowego? Wiemy przecież z góry, że zamierzamy obciąć napis do czterech znaków; wiemy również, że mamy juz przynajmniej jeden znak (pierwszą literę zmiennej `source`, która w wyniku tego algorytmu nie zmienia się). Oznacza to, że możemy po prostu dodać trzy zera do napisu wyjściowego, a następnie go przyciąć. Nie dajcie się nigdy zbić z tropu przez dosłowne sformułowanie problemu; czasami wystarczy spojrzeć na ten problem z trochę innej perspektywy, a już pojawia się prostsze rozwiązanie.

Czy usuwając pętlę `while` zyskaliśmy na prędkości programu soundex4c.py? Nawet znacznie:

```
C:\samples\soundex\stage4>python soundex4c.py
Woo             W000 4.89129791636
Pilgrim         P426 7.30642134685
Flingjingwaller F452 10.689832367
```

Na koniec warto zauważyć, że jest jeszcze jedna rzecz, jaką można zrobić, aby przyspieszyć te trzy linijki kodu: można przekształcić je do jednej linii. Spójrzmy na soundex/stage4/soundex4d.py:

```
return (digits2.replace('9', '') + '000')[:4]
```

Umieszczenie tego kodu w jednej linii sprawiło, że stał się on zaledwie odrobinę szybszy, niż soundex4c.py:

```
C:\samples\soundex\stage4>python soundex4d.py
Woo             W000 4.93624105857
Pilgrim         P426 7.19747593619
Flingjingwaller F452 10.5490700634
```

Za cenę tego niewielkiego wzrostu wydajności stał się przy okazji znacznie mniej czytelny. Czy warto było to zrobić? Mam nadzieję, że potraficie to dobrze skomentować. Wydajność to nie wszystko. Wysiłki związane z optymalizacją kodu muszą być zawsze równoważone dążeniem do tego, aby kod był czytelny i łatwy w utrzymaniu.

19.7 Optymalizacja szybkości - podsumowanie

Podsumowanie

Rozdział ten zilustrował kilka ważnych aspektów dotyczących zoptymalizowania czasu działania programu w Pythonie, jak i w ogólności optymalizacji czasu działania.

- Jeśli masz wybrać między wyrażeniami regularnymi, a pisanie własnej pętli, wybierz wyrażenia regularne. Wyrażenia regularne są przekompilowane w C, więc będą się wykonywały bezpośrednio przez twój komputer; twoja pętla jest pisana w Pythonie i działa za pośrednictwem interpretera Pythona.
- Jeśli masz wybrać między wyrażeniami regularnymi, a metodami łańcucha znaków, wybierz metody łańcucha znaków. Obydwa są przekompilowane w C, więc wybieramy prostsze wersje.
- Ogólne zastosowanie związane z przeszukiwaniem słowników jest szybkie, jednak specjalistyczne funkcja takie jak `string.maketrans`, czy też metody takie jak `isalpha()` są szybsze. Jeśli Python posiada funkcję specjalnie przystosowaną dla ciebie, wykorzystaj ją.
- Nie bądź za mądry. Czasami najbardziej oczywiste algorytmy są najszybsze, jakie byśmy wymyślili.
- Nie męcz się za bardzo. Szybkość wykonywania nie jest wszystkim.

Trzeba podkreślić, że ostatni punkt jest dosyć znaczący. Przez cały kurs tego rozdziału, przerobiliśmy tę funkcję tak, aby działała trzy razy szybciej i zaoszczędziliśmy 20 sekund na 1 milion wywołań tej funkcji. Wspaniale! Ale pomyślmy teraz: podczas wykonywania miliona wywołań tej funkcji, jak długo będziemy musieli czekać na połączenie bazy danych? Albo jak długo będziemy czekać wykonując operacje wejścia/wyjścia na dysku? Albo jak długo będziemy czekać na wejście użytkownika? Nie marnuj za dużo czasu na czasową optymalizację jednego algorytmu. Pobaw się bardziej nad istotnymi fragmentami kodu, które mają działać szybko, następnie popraw wyłapane błędy, a resztę zostaw w spokoju.

Dodatek A

Informacje o pliku i historia

A.1 Historia

Ta książka została stworzona na polskojęzycznej wersji projektu Wikibooks przez autorów wymienionych poniżej w sekcji Autorzy. Najnowsza wersja podręcznika jest dostępna pod adresem `http://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie`.

A.2 Informacje o pliku PDF i historia

PDF został utworzony przez Derbetha dnia 17 lutego 2008 na podstawie wersji z 9 lutego 2008 podręcznika na Wikibooks. Wykorzystany został poprawiony program Wiki2LaTeX autorstwa użytkownika angielskich Wikibooks, Hagindaza. Wynikowy kod po ręcznych poprawkach został przekształcony w książkę za pomocą systemu składu LaTeX.

Najnowsza wersja tego PDF-u jest postępna pod adresem `http://pl.wikibooks.org/wiki/Image:Zanurkuj_w_Pythonie.pdf`.

A.3 Autorzy

Akira, Ciastek, Derbeth, Diodac, Farin mag, Fred 4, Fservant, JaroslawZabiello, Jau, Kabturek, KamilaChyla, Kamils, Kangel, Kj, Kocio, Koko, Krzysiu Jarzyna, Kubik, Lewico, Lwh, MTM, Migol, Myki, Nata, Neverous, Pepson, Pietras1988, Piotr, Prasuk historyk, Rafał Rawicki, Robwolfe, Rofrol, Roman 92, Salmon, Sasek, Skalee vel crensh, Sqrll, Sznurek, Tertulian, Turbofart, Tyfus, Warszk, Yurez, Zeo, Zyx i anonimowi autorzy.

A.4 Grafiki

Autorzy i licencje grafik:

- grafika na okładce: Python molurus z "Fauna of British India" G. A. Boulengera, źródło Wikimedia Commons, public domain
- logo Wikibooks: zastrzeżony znak towarowy, © & ™All rights reserved, Wikimedia Foundation, Inc.

Dodatek B

Dalsze wykorzystanie tej książki

B.1 Wstęp

Ideą Wikibooks jest swobodne dzielenie się wiedzą, dlatego też uregulowania Wikibooks mają na celu jak najmniejsze ograniczanie możliwości osób korzystających z serwisu i zapewnienie, że treści dostępne tutaj będą mogły być łatwo kopiowane i wykorzystywane dalej. Prosimy wszystkich odwiedzających, aby poświęcili chwilę na zaznajomienie się z poniższymi zasadami, by uniknąć w przyszłości nieporozumień.

B.2 Status prawny

Cała zawartość Wikibooks (o ile w danym miejscu nie jest zaznaczone inaczej; szczegóły niżej) jest udostępniona na następujących warunkach:

Udziela się zezwolenia na kopiowanie, rozpowszechnianie i/lub modyfikację treści artykułów polskich Wikibooks zgodnie z zasadami Licencji GNU Wolnej Dokumentacji (GNU Free Documentation License) w wersji 1.2 lub dowolnej późniejszej opublikowanej przez Free Software Foundation; bez Sekcji Niezmiennych, Tekstu na Przedniej Okładce i bez Tekstu na Tylnej Okładce. Kopia tekstu licencji znajduje się na stronie GNU Free Documentation License.

Zasadniczo oznacza to, że artykuły pozostaną na zawsze dostępne na zasadach open source i mogą być używane przez każdego z obwarowaniami wyszczególnionymi poniżej, które zapewniają, że artykuły te pozostaną wolne.

Grafiki i pliku multimedialne wykorzystane na Wikibooks mogą być udostępnione na warunkach innych niż GNU FDL. Aby sprawdzić warunki korzystania z grafiki, należy przejść do jej strony opisu, klikając na grafice.

B.3 Wykorzystywanie materiałów z Wikibooks

Jeśli użytkownik polskich Wikibooks chce wykorzystać materiały w niej zawarte, musi to zrobić zgodnie z GNU FDL. Warunki te to w skrócie i uproszczeniu:

Publikacja w Internecie

1. dobrze jest wymienić Wikibooks jako źródło (jest to nieobowiązkowe, lecz jest dobrym zwyczajem)
2. należy podać listę autorów lub wyraźnie opisany i funkcjonujący link do oryginalnej treści na Wikibooks lub historii edycji (wypełnia to obowiązek podania autorów oryginalnego dzieła)
3. trzeba jasno napisać, że treść publikowanego dzieła jest objęta licencją GNU FDL
4. należy podać link to tekstu licencji (najlepiej zachowanej na własnym serwerze)
5. postać tekstu nie może ograniczać możliwości jego kopiowania

Druk

1. dobrze jest wymienić Wikibooks jako źródło (jest to nieobowiązkowe, lecz jest dobrym zwyczajem)
2. należy wymienić 5 autorów z listy w rozdziale *Autorzy* danego podręcznika (w przypadku braku podobnego rozdziału — lista autorów jest dostępna pod odnośnikiem "historia" na górze strony). Gdy podręcznik ma mniej niż 5 autorów, należy wymienić wszystkich.
3. trzeba jasno napisać na przynajmniej jednej stronie, że treść publikowanego dzieła jest objęta licencją GNU FDL
4. *pełny* tekst licencji, w oryginale i bez zmian, musi być zawarty w książce
5. jeśli zostanie wykonanych więcej niz 100 kopii książki konieczne jest:
   (a) dostarczenie płyt CD, DVD, dysków lub innych nośników danych z treścia książki w formie możliwą do komputerowego przetwarzania; lub:
   (b) dostarczenie linku do strony z czytelną dla komputera formą książki (link musi być aktywny przynajmniej przez rok od publikacji; można użyć linku do spisu treści danego podręcznika na Wikibooks)

Nie ma wymogu pytania o zgodę na wykorzystanie tekstu jakichkolwiek osób z Wikibooks. Autorzy nie mogą zabronić nikomu wykorzystywania ich tekstów zgodnie z licencją GNU FDL. Można korzystać z książek jako całości albo z ich fragmentów. Materiały bazujące na treści z Wikibooks mogą być bez przeszkód sprzedawane; zyskami nie trzeba dzielić się z autorami oryginalnego dzieła.

Jeżeli w wykorzystanych materiałach z polskich Wikibooks są treści objęte innymi niż GNU FDL licencjami, użytkownik musi spełnić wymagania zawarte w tych licencjach (dotyczy to szczególnie grafik, które mogą mieć różne licencje).

Dodatek C

GNU Free Documentation License

Version 1.2, November 2002
Copyright © 2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and

is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or

"**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under

this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with . . . Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.