

# Table of Contents

<b>Dive Into Python.....</b>	<b>1</b>
<b>Chapter 1. Installing Python.....</b>	<b>2</b>
1.1. Which Python is right for you?.....	2
1.2. Python on Windows.....	2
1.3. Python on Mac OS X.....	3
1.4. Python on Mac OS 9.....	5
1.5. Python on RedHat Linux.....	5
1.6. Python on Debian GNU/Linux.....	6
1.7. Python Installation from Source.....	6
1.8. The Interactive Shell.....	7
1.9. Summary.....	8
<b>Chapter 2. Your First Python Program.....</b>	<b>9</b>
2.1. Diving in.....	9
2.2. Declaring Functions.....	9
2.3. Documenting Functions.....	10
2.4. Everything Is an Object.....	11
2.5. Indenting Code.....	13
2.6. Testing Modules.....	14
<b>Chapter 3. Native Datatypes.....</b>	<b>15</b>
3.1. Introducing Dictionaries.....	15
3.2. Introducing Lists.....	17
3.3. Introducing Tuples.....	22
3.4. Declaring variables.....	23
3.5. Formatting Strings.....	25
3.6. Mapping Lists.....	26
3.7. Joining Lists and Splitting Strings.....	28
3.8. Summary.....	29
<b>Chapter 4. The Power Of Introspection.....</b>	<b>31</b>
4.1. Diving In.....	31
4.2. Using Optional and Named Arguments.....	32
4.3. Using type, str, dir, and Other Built-In Functions.....	33
4.4. Getting Object References With getattr.....	36
4.5. Filtering Lists.....	38
4.6. The Peculiar Nature of and and or.....	39
4.7. Using lambda Functions.....	41
4.8. Putting It All Together.....	43
4.9. Summary.....	45
<b>Chapter 5. Objects and Object–Orientation.....</b>	<b>47</b>
5.1. Diving In.....	47
5.2. Importing Modules Using from module import.....	49
5.3. Defining Classes.....	50
5.4. Instantiating Classes.....	53
5.5. Exploring UserDict: A Wrapper Class.....	54
5.6. Special Class Methods.....	56
5.7. Advanced Special Class Methods.....	59

# Table of Contents

<b>Chapter 5. Objects and Object–Orientation</b>	
5.8. Introducing Class Attributes.....	60
5.9. Private Functions.....	62
5.10. Summary.....	63
<b>Chapter 6. Exceptions and File Handling.....</b>	<b>64</b>
6.1. Handling Exceptions.....	64
6.2. Working with File Objects.....	66
6.3. Iterating with for Loops.....	70
6.4. Using sys.modules.....	72
6.5. Working with Directories.....	74
6.6. Putting It All Together.....	77
6.7. Summary.....	78
<b>Chapter 7. Regular Expressions.....</b>	<b>81</b>
7.1. Diving In.....	81
7.2. Case Study: Street Addresses.....	81
7.3. Case Study: Roman Numerals.....	83
7.4. Using the {n,m} Syntax.....	85
7.5. Verbose Regular Expressions.....	88
7.6. Case study: Parsing Phone Numbers.....	89
7.7. Summary.....	93
<b>Chapter 8. HTML Processing.....</b>	<b>94</b>
8.1. Diving in.....	94
8.2. Introducing sgmlib.py.....	98
8.3. Extracting data from HTML documents.....	100
8.4. Introducing BaseHTMLProcessor.py.....	102
8.5. locals and globals.....	104
8.6. Dictionary–based string formatting.....	107
8.7. Quoting attribute values.....	108
8.8. Introducing dialect.py.....	109
8.9. Putting it all together.....	111
8.10. Summary.....	113
<b>Chapter 9. XML Processing.....</b>	<b>115</b>
9.1. Diving in.....	115
9.2. Packages.....	121
9.3. Parsing XML.....	123
9.4. Unicode.....	125
9.5. Searching for elements.....	129
9.6. Accessing element attributes.....	131
9.7. Segue.....	132
<b>Chapter 10. Scripts and Streams.....</b>	<b>133</b>
10.1. Abstracting input sources.....	133
10.2. Standard input, output, and error.....	136
10.3. Caching node lookups.....	140
10.4. Finding direct children of a node.....	141
10.5. Creating separate handlers by node type.....	141

# Table of Contents

<b>Chapter 10. Scripts and Streams</b>	
10.6. Handling command–line arguments.....	143
10.7. Putting it all together.....	146
10.8. Summary.....	148
<b>Chapter 11. HTTP Web Services.....</b>	<b>149</b>
11.1. Diving in.....	149
11.2. How not to fetch data over HTTP.....	151
11.3. Features of HTTP.....	152
11.4. Debugging HTTP web services.....	153
11.5. Setting the User–Agent.....	155
11.6. Handling Last–Modified and ETag.....	156
11.7. Handling redirects.....	159
11.8. Handling compressed data.....	163
11.9. Putting it all together.....	165
11.10. Summary.....	167
<b>Chapter 12. SOAP Web Services.....</b>	<b>168</b>
12.1. Diving In.....	168
12.2. Installing the SOAP Libraries.....	169
12.3. First Steps with SOAP.....	171
12.4. Debugging SOAP Web Services.....	172
12.5. Introducing WSDL.....	173
12.6. Introspecting SOAP Web Services with WSDL.....	174
12.7. Searching Google.....	176
12.8. Troubleshooting SOAP Web Services.....	179
12.9. Summary.....	182
<b>Chapter 13. Unit Testing.....</b>	<b>183</b>
13.1. Introduction to Roman numerals.....	183
13.2. Diving in.....	184
13.3. Introducing romantest.py.....	184
13.4. Testing for success.....	187
13.5. Testing for failure.....	189
13.6. Testing for sanity.....	190
<b>Chapter 14. Test–First Programming.....</b>	<b>193</b>
14.1. roman.py, stage 1.....	193
14.2. roman.py, stage 2.....	196
14.3. roman.py, stage 3.....	199
14.4. roman.py, stage 4.....	202
14.5. roman.py, stage 5.....	205
<b>Chapter 15. Refactoring.....</b>	<b>208</b>
15.1. Handling bugs.....	208
15.2. Handling changing requirements.....	210
15.3. Refactoring.....	216
15.4. Postscript.....	219
15.5. Summary.....	221

# Table of Contents

<b>Chapter 16. Functional Programming.....</b>	<b>223</b>
16.1. Diving in.....	223
16.2. Finding the path.....	224
16.3. Filtering lists revisited.....	226
16.4. Mapping lists revisited.....	228
16.5. Data-centric programming.....	229
16.6. Dynamically importing modules.....	230
16.7. Putting it all together.....	231
16.8. Summary.....	234
<b>Chapter 17. Dynamic functions.....</b>	<b>235</b>
17.1. Diving in.....	235
17.2. plural.py, stage 1.....	235
17.3. plural.py, stage 2.....	237
17.4. plural.py, stage 3.....	239
17.5. plural.py, stage 4.....	240
17.6. plural.py, stage 5.....	242
17.7. plural.py, stage 6.....	243
17.8. Summary.....	246
<b>Chapter 18. Performance Tuning.....</b>	<b>247</b>
18.1. Diving in.....	247
18.2. Using the timeit Module.....	249
18.3. Optimizing Regular Expressions.....	250
18.4. Optimizing Dictionary Lookups.....	253
18.5. Optimizing List Operations.....	256
18.6. Optimizing String Manipulation.....	258
18.7. Summary.....	260
<b>Appendix A. Further reading.....</b>	<b>261</b>
<b>Appendix B. A 5-minute review.....</b>	<b>268</b>
<b>Appendix C. Tips and tricks.....</b>	<b>282</b>
<b>Appendix D. List of examples.....</b>	<b>289</b>
<b>Appendix E. Revision history.....</b>	<b>302</b>
<b>Appendix F. About the book.....</b>	<b>314</b>
<b>Appendix G. GNU Free Documentation License.....</b>	<b>315</b>
G.0. Preamble.....	315
G.1. Applicability and definitions.....	315
G.2. Verbatim copying.....	316
G.3. Copying in quantity.....	316
G.4. Modifications.....	317
G.5. Combining documents.....	318
G.6. Collections of documents.....	318
G.7. Aggregation with independent works.....	318

# Table of Contents

## **Appendix G. GNU Free Documentation License**

G.8. Translation.....	318
G.9. Termination.....	319
G.10. Future revisions of this license.....	319
G.11. How to use this License for your documents.....	319

## **Appendix H. Python license.....320**

H.A. History of the software.....	320
H.B. Terms and conditions for accessing or otherwise using Python.....	320

# Dive Into Python

20 May 2004

Copyright © 2000, 2001, 2002, 2003, 2004 Mark Pilgrim (mailto:mark@diveintopython.org)

This book lives at <http://diveintopython.org/>. If you're reading it somewhere else, you may not have the latest version.

Permission is granted to copy, distribute, and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 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 Appendix G, *GNU Free Documentation License*.

The example programs in this book are free software; you can redistribute and/or modify them under the terms of the Python license as published by the Python Software Foundation. A copy of the license is included in Appendix H, *Python license*.

---

# Chapter 1. Installing Python

Welcome to Python. Let's dive in. In this chapter, you'll install the version of Python that's right for you.

## 1.1. Which Python is right for you?

The first thing you need to do with Python is install it. Or do you?

If you're using an account on a hosted server, your ISP may have already installed Python. Most popular Linux distributions come with Python in the default installation. Mac OS X 10.2 and later includes a command-line version of Python, although you'll probably want to install a version that includes a more Mac-like graphical interface.

Windows does not come with any version of Python, but don't despair! There are several ways to point-and-click your way to Python on Windows.

As you can see already, Python runs on a great many operating systems. The full list includes Windows, Mac OS, Mac OS X, and all varieties of free UNIX-compatible systems like Linux. There are also versions that run on Sun Solaris, AS/400, Amiga, OS/2, BeOS, and a plethora of other platforms you've probably never even heard of.

What's more, Python programs written on one platform can, with a little care, run on *any* supported platform. For instance, I regularly develop Python programs on Windows and later deploy them on Linux.

So back to the question that started this section, "Which Python is right for you?" The answer is whichever one runs on the computer you already have.

## 1.2. Python on Windows

On Windows, you have a couple choices for installing Python.

ActiveState makes a Windows installer for Python called ActivePython, which includes a complete version of Python, an IDE with a Python-aware code editor, plus some Windows extensions for Python that allow complete access to Windows-specific services, APIs, and the Windows Registry.

ActivePython is freely downloadable, although it is not open source. It is the IDE I used to learn Python, and I recommend you try it unless you have a specific reason not to. One such reason might be that ActiveState is generally several months behind in updating their ActivePython installer when new version of Python are released. If you absolutely need the latest version of Python and ActivePython is still a version behind as you read this, you'll want to use the second option for installing Python on Windows.

The second option is the "official" Python installer, distributed by the people who develop Python itself. It is freely downloadable and open source, and it is always current with the latest version of Python.

### Procedure 1.1. Option 1: Installing ActivePython

Here is the procedure for installing ActivePython:

1. Download ActivePython from <http://www.activestate.com/Products/ActivePython/>.
2. If you are using Windows 95, Windows 98, or Windows ME, you will also need to download and install Windows Installer 2.0 (<http://download.microsoft.com/download/WindowsInstaller/Install/2.0/W9XMe/EN-US/InstMsiA.exe>) before installing ActivePython.

3. Double-click the installer, `ActivePython-2.2.2-224-win32-ix86.msi`.
4. Step through the installer program.
5. If space is tight, you can do a custom installation and deselect the documentation, but I don't recommend this unless you absolutely can't spare the 14MB.
6. After the installation is complete, close the installer and choose `Start->Programs->ActiveState ActivePython 2.2->PythonWin IDE`. You'll see something like the following:

```
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.
>>>
```

### Procedure 1.2. Option 2: Installing Python from Python.org (<http://www.python.org/>)

1. Download the latest Python Windows installer by going to <http://www.python.org/ftp/python/> and selecting the highest version number listed, then downloading the `.exe` installer.
2. Double-click the installer, `Python-2.xxx.yyy.exe`. The name will depend on the version of Python available when you read this.
3. Step through the installer program.
4. If disk space is tight, you can deselect the HTMLHelp file, the utility scripts (`Tools/`), and/or the test suite (`Lib/test/`).
5. If you do not have administrative rights on your machine, you can select Advanced Options, then choose Non-Admin Install. This just affects where Registry entries and Start menu shortcuts are created.
6. After the installation is complete, close the installer and select `Start->Programs->Python 2.3->IDLE (Python GUI)`. You'll see something like the following:

```
Python 2.3.2 (#49, Oct 2 2003, 20:02:00) [MSC v.1200 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 1.0
>>>
```

## 1.3. Python on Mac OS X

On Mac OS X, you have two choices for installing Python: install it, or don't install it. You probably want to install it.

Mac OS X 10.2 and later comes with a command-line version of Python preinstalled. If you are comfortable with the command line, you can use this version for the first third of the book. However, the preinstalled version does not come with an XML parser, so when you get to the XML chapter, you'll need to install the full version.

Rather than using the preinstalled version, you'll probably want to install the latest version, which also comes with a graphical interactive shell.

### Procedure 1.3. Running the Preinstalled Version of Python on Mac OS X

To use the preinstalled version of Python, follow these steps:

1. Open the `/Applications` folder.



2. Open the Utilities folder.
3. Double-click Terminal to open a terminal window and get to a command line.
4. Type **python** at the command prompt.

Try it out:

```
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%
```

## Procedure 1.4. Installing the Latest Version of Python on Mac OS X

Follow these steps to download and install the latest version of Python:

1. Download the MacPython-OSX disk image from <http://homepages.cwi.nl/~jack/macpython/download.html>.
2. If your browser has not already done so, double-click MacPython-OSX-2.3-1.dmg to mount the disk image on your desktop.
3. Double-click the installer, MacPython-OSX.pkg.
4. The installer will prompt you for your administrative username and password.
5. Step through the installer program.
6. After installation is complete, close the installer and open the /Applications folder.
7. Open the MacPython-2.3 folder
8. Double-click PythonIDE to launch Python.

The MacPython IDE should display a splash screen, then take you to the interactive shell. If the interactive shell does not appear, select Window->Python Interactive (**Cmd-0**). The opening window will look something like this:

```
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
>>>
```

Note that once you install the latest version, the pre-installed version is still present. If you are running scripts from the command line, you need to be aware which version of Python you are using.

## Example 1.1. Two versions of Python

```
[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%
```

## 1.4. Python on Mac OS 9

Mac OS 9 does not come with any version of Python, but installation is very simple, and there is only one choice.

Follow these steps to install Python on Mac OS 9:

1. Download the `MacPython23full.bin` file from <http://homepages.cwi.nl/~jack/macpython/download.html>.
2. If your browser does not decompress the file automatically, double-click `MacPython23full.bin` to decompress the file with Stuffit Expander.
3. Double-click the installer, `MacPython23full`.
4. Step through the installer program.
5. After installation is complete, close the installer and open the `/Applications` folder.
6. Open the `MacPython-OS9 2.3` folder.
7. Double-click `Python IDE` to launch Python.

The MacPython IDE should display a splash screen, and then take you to the interactive shell. If the interactive shell does not appear, select `Window->Python Interactive (Cmd-0)`. You'll see a screen like this:

```
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
>>>
```

## 1.5. Python on RedHat Linux

Installing under UNIX-compatible operating systems such as Linux is easy if you're willing to install a binary package. Pre-built binary packages are available for most popular Linux distributions. Or you can always compile from source.

Download the latest Python RPM by going to <http://www.python.org/ftp/python/> and selecting the highest version number listed, then selecting the `rpms/` directory within that. Then download the RPM with the highest version number. You can install it with the `rpm` command, as shown here:

### Example 1.2. Installing on RedHat Linux 9

```
localhost:~$ su -
Password: [enter your root password]
[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 ❶
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.
>>> [press Ctrl+D to exit]
[root@localhost root]# python2.3 ❷
```

```
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.
>>> [press Ctrl+D to exit]
[root@localhost root]# which python2.3 ❸
/usr/bin/python2.3
```

- ❶ Whoops! Just typing **python** gives you the older version of Python — the one that was installed by default. That's not the one you want.
- ❷ At the time of this writing, the newest version is called **python2.3**. You'll probably want to change the path on the first line of the sample scripts to point to the newer version.
- ❸ This is the complete path of the newer version of Python that you just installed. Use this on the **#!** line (the first line of each script) to ensure that scripts are running under the latest version of Python, and be sure to type **python2.3** to get into the interactive shell.

## 1.6. Python on Debian GNU/Linux

If you are lucky enough to be running Debian GNU/Linux, you install Python through the **apt** command.

### Example 1.3. Installing on Debian GNU/Linux

```
localhost:~$ su -
Password: [enter your root password]
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.
>>> [press Ctrl+D to exit]
```

## 1.7. Python Installation from Source

If you prefer to build from source, you can download the Python source code from <http://www.python.org/ftp/python/>. Select the highest version number listed, download the `.tgz` file), and then do the usual **configure**, **make**, **make**

**install** dance.

### Example 1.4. Installing from source

```
localhost:~$ su -
Password: [enter your root password]
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 xzf 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.
>>> [press Ctrl+D to get back to the command prompt]
localhost:~$
```

## 1.8. The Interactive Shell

Now that you have Python installed, what's this interactive shell thing you're running?

It's like this: Python leads a double life. It's an interpreter for scripts that you can run from the command line or run like applications, by double-clicking the scripts. But it's also an interactive shell that can evaluate arbitrary statements and expressions. This is extremely useful for debugging, quick hacking, and testing. I even know some people who use the Python interactive shell in lieu of a calculator!

Launch the Python interactive shell in whatever way works on your platform, and let's dive in with the steps shown here:

### Example 1.5. First Steps in the Interactive Shell

```
>>> 1 + 1
```



```
2
>>> print 'hello world' ❷
hello world
>>> x = 1                ❸
>>> y = 2
>>> x + y
3
```

- ❶ The Python interactive shell can evaluate arbitrary Python expressions, including any basic arithmetic expression.
- ❷ The interactive shell can execute arbitrary Python statements, including the **print** statement.
- ❸ You can also assign values to variables, and the values will be remembered as long as the shell is open (but not any longer than that).

## 1.9. Summary

You should now have a version of Python installed that works for you.

Depending on your platform, you may have more than one version of Python installed. If so, you need to be aware of your paths. If simply typing **python** on the command line doesn't run the version of Python that you want to use, you may need to enter the full pathname of your preferred version.

Congratulations, and welcome to Python.

# Chapter 2. Your First Python Program

You know how other books go on and on about programming fundamentals and finally work up to building a complete, working program? Let's skip all that.

## 2.1. Diving in

Here is a complete, working Python program.

It probably makes absolutely no sense to you. Don't worry about that, because you're going to dissect it line by line. But read through it first and see what, if anything, you can make of it.

### Example 2.1. `odbchelper.py`


If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.


```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.


    Returns string."""
    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)
```

Now run this program and see what happens.

In the ActivePython IDE on  Windows, you can run the Python program you're editing by choosing File->Run... (**Ctrl-R**). Output is displayed in the interactive window.

In the Python IDE on Mac  you can run a Python program with Python->Run window... (**Cmd-R**), but there is an important option you must set first. Open the .py file in the IDE, pop up the options menu by clicking the black triangle in the upper-right corner of the window, and make sure the Run as \_\_main\_\_ option is checked. This is a per-file setting, but you'll only need to do it once per file.

On UNIX-compatible systems  (including Mac OS X), you can run a Python program from the command line:  
**python odbchelper.py**

The output of `odbchelper.py` will look like this:

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

## 2.2. Declaring Functions

Python has functions like most other languages, but it does not have separate header files like C++ or interface/implementation sections like Pascal. When you need a function, just declare it, like this:

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

Note that the keyword `def` starts the function declaration, followed by the function name, followed by the arguments in parentheses. Multiple arguments (not shown here) are separated with commas.

Also note that the function doesn't define a return datatype. Python functions do not specify the datatype of their return value; they don't even specify whether or not they return a value. In fact, every Python function returns a value; if the function ever executes a `return` statement, it will return that value, otherwise it will return `None`, the Python null value.

In Visual Basic, functions (that return a value) start with `function`, and subroutines (that do not return a value) start with `sub`. There are no subroutines in Python. Everything is a function, all functions return a value (even if it's `None`), and all functions start with `def`.

The argument, `params`, doesn't specify a datatype. In Python, variables are never explicitly typed. Python figures out what type a variable is and keeps track of it internally.

In Java, C++, and other statically-typed languages, you must specify the datatype of the function return value and each function argument. In Python, you never explicitly specify the datatype of anything. Based on what value you assign, Python keeps track of the datatype internally.

### 2.2.1. How Python's Datatypes Compare to Other Programming Languages

An erudite reader sent me this explanation of how Python compares to other programming languages:

*statically typed language*

A language in which types are fixed at compile time. Most statically typed languages enforce this by requiring you to declare all variables with their datatypes before using them. Java and C are statically typed languages.

*dynamically typed language*

A language in which types are discovered at execution time; the opposite of statically typed. VBScript and Python are dynamically typed, because they figure out what type a variable is when you first assign it a value.

*strongly typed language*

A language in which types are always enforced. Java and Python are strongly typed. If you have an integer, you can't treat it like a string without explicitly converting it.

*weakly typed language*

A language in which types may be ignored; the opposite of strongly typed. VBScript is weakly typed. In VBScript, you can concatenate the string `'12'` and the integer `3` to get the string `'123'`, then treat that as the integer `123`, all without any explicit conversion.

So Python is both *dynamically typed* (because it doesn't use explicit datatype declarations) and *strongly typed* (because once a variable has a datatype, it actually matters).

## 2.3. Documenting Functions

You can document a Python function by giving it a `doc string`.

### Example 2.2. Defining the `buildConnectionString` Function's `doc string`

```
def buildConnectionString(params):  
    """Build a connection string from a dictionary of parameters.
```

Returns `string`.`"""`

Triple quotes signify a multi-line string. Everything between the start and end quotes is part of a single string, including carriage returns and other quote characters. You can use them anywhere, but you'll see them most often used when defining a `doc string`.

Triple quotes are also an easy way to define a string with both single and double quotes, like `qq/.../` in Perl. Everything between the triple quotes is the function's `doc string`, which documents what the function does. A `doc string`, if it exists, must be the first thing defined in a function (that is, the first thing after the colon). You don't technically need to give your function a `doc string`, but you always should. I know you've heard this in every programming class you've ever taken, but Python gives you an added incentive: the `doc string` is available at runtime as an attribute of the function.

Many Python IDEs use the `doc string` to provide context-sensitive documentation, so that when you type a function name, its `doc string` appears as a tooltip. This can be incredibly helpful, but it's only as good as the `doc strings` you write.

### Further Reading on Documenting Functions

- PEP 257 (<http://www.python.org/peps/pep-0257.html>) defines `doc string` conventions.
- *Python Style Guide* (<http://www.python.org/doc/essays/styleguide.html>) discusses how to write a good `doc string`.
- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses conventions for spacing in `doc strings` (<http://www.python.org/doc/current/tut/node6.html#SECTION00675000000000000000>).

## 2.4. Everything Is an Object

In case you missed it, I just said that Python functions have attributes, and that those attributes are available at runtime.

A function, like everything else in Python, is an object.

Open your favorite Python IDE and follow along:

### Example 2.3. Accessing the `buildConnectionString` Function's `doc string`

```
>>> import odbchelper ❶
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> print odbchelper.buildConnectionString(params) ❷
server=mpilgrim;uid=sa;database=master;pwd=secret
>>> print odbchelper.buildConnectionString.__doc__ ❸
Build a connection string from a dictionary

Returns string.
```

- ❶ The first line imports the `odbchelper` program as a module — a chunk of code that you can use interactively, or from a larger Python program. (You'll see examples of multi-module Python programs in Chapter 4.) Once you import a module, you can reference any of its public functions, classes, or attributes. Modules can do this to access functionality in other modules, and you can do it in the IDE too. This is an important concept, and you'll talk more about it later.



- ❷ When you want to use functions defined in imported modules, you need to include the module name. So you can't just say `buildConnectionString`; it must be `odbcHelper.buildConnectionString`. If you've used classes in Java, this should feel vaguely familiar.
- ❸ Instead of calling the function as you would expect to, you asked for one of the function's attributes, `__doc__`.

`import` in Python is like `require` in Perl. Once you `import` a Python module, you access its functions with `module.function`; once you `require` a Perl module, you access its functions with `module::function`.

### 2.4.1. The Import Search Path

Before you go any further, I want to briefly mention the library search path. Python looks in several places when you try to import a module. Specifically, it looks in all the directories defined in `sys.path`. This is just a list, and you can easily view it or modify it with standard list methods. (You'll learn more about lists later in this chapter.)

#### Example 2.4. Import Search Path

```
>>> import sys                                ❶
>>> sys.path                                  ❷
['', '/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                                       ❸
<module 'sys' (built-in)>
>>> sys.path.append('/my/new/path')          ❹
```

- ❶ Importing the `sys` module makes all of its functions and attributes available.
- ❷ `sys.path` is a list of directory names that constitute the current search path. (Yours will look different, depending on your operating system, what version of Python you're running, and where it was originally installed.) Python will look through these directories (in this order) for a `.py` file matching the module name you're trying to import.
- ❸ Actually, I lied; the truth is more complicated than that, because not all modules are stored as `.py` files. Some, like the `sys` module, are "built-in modules"; they are actually baked right into Python itself. Built-in modules behave just like regular modules, but their Python source code is not available, because they are not written in Python! (The `sys` module is written in C.)
- ❹ You can add a new directory to Python's search path at runtime by appending the directory name to `sys.path`, and then Python will look in that directory as well, whenever you try to import a module. The effect lasts as long as Python is running. (You'll talk more about `append` and other list methods in Chapter 3.)

### 2.4.2. What's an Object?

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute `__doc__`, which returns the `doc` string defined in the function's source code. The `sys` module is an object which has (among other things) an attribute called `path`. And so forth.

Still, this begs the question. What is an object? Different programming languages define "object" in different ways. In some, it means that *all* objects *must* have attributes and methods; in others, it means that all objects are subclassable. In Python, the definition is looser; some objects have neither attributes nor methods (more on this in Chapter 3), and not all objects are subclassable (more on this in Chapter 5). But everything is an object in the sense that it can be assigned to a variable or passed as an argument to a function (more on this in Chapter 4).

This is so important that I'm going to repeat it in case you missed it the first few times: *everything in Python is an object*. Strings are objects. Lists are objects. Functions are objects. Even modules are objects.

## Further Reading on Objects

- *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) explains exactly what it means to say that everything in Python is an object (<http://www.python.org/doc/current/ref/objects.html>), because some people are pedantic and like to discuss this sort of thing at great length.
- *eff-bot* (<http://www.effbot.org/guides/>) summarizes Python objects (<http://www.effbot.org/guides/python-objects.htm>).

## 2.5. Indenting Code

Python functions have no explicit `begin` or `end`, and no curly braces to mark where the function code starts and stops. The only delimiter is a colon (`:`) and the indentation of the code itself.

### Example 2.5. Indenting the `buildConnectionString` Function

```
def buildConnectionString(params):  
    """Build a connection string from a dictionary of parameters.  
  
    Returns string."""  
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Code blocks are defined by their indentation. By "code block", I mean functions, `if` statements, `for` loops, `while` loops, and so forth. Indenting starts a block and unindenting ends it. There are no explicit braces, brackets, or keywords. This means that whitespace is significant, and must be consistent. In this example, the function code (including the `doc string`) is indented four spaces. It doesn't need to be four spaces, it just needs to be consistent. The first line that is not indented is outside the function.

Example 2.6, `if Statements` shows an example of code indentation with `if` statements.

### Example 2.6. `if` Statements

```
def fib(n):  
    print 'n =', n  
    if n > 1:  
        return n * fib(n - 1)  
    else:  
        print 'end of the line'  
        return 1
```

- ❶ This is a function named `fib` that takes one argument, `n`. All the code within the function is indented.
- ❷ Printing to the screen is very easy in Python, just use `print`. `print` statements can take any data type, including strings, integers, and other native types like dictionaries and lists that you'll learn about in the next chapter. You can even mix and match to print several things on one line by using a comma-separated list of values. Each value is printed on the same line, separated by spaces (the commas don't print). So when `fib` is called with 5, this will print "n = 5".
- ❸ `if` statements are a type of code block. If the `if` expression evaluates to true, the indented block is executed, otherwise it falls to the `else` block.
- ❹ Of course `if` and `else` blocks can contain multiple lines, as long as they are all indented the same amount. This `else` block has two lines of code in it. There is no other special syntax for multi-line code blocks. Just indent and get on with your life.

After some initial protests and several snide analogies to Fortran, you will make peace with this and start seeing its benefits. One major benefit is that all Python programs look similar, since indentation is a language requirement and not a matter of style. This makes it easier to read and understand other people's Python code.

Python uses carriage returns to separate statements and a colon and indentation to separate code blocks. C++ and Java use semicolons to separate statements and curly braces to separate code blocks.

### Further Reading on Code Indentation

- *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) discusses cross-platform indentation issues and shows various indentation errors (<http://www.python.org/doc/current/ref/indentation.html>).
- *Python Style Guide* (<http://www.python.org/doc/essays/styleguide.html>) discusses good indentation style.

## 2.6. Testing Modules

Python modules are objects and have several useful attributes. You can use this to easily test your modules as you write them. Here's an example that uses the `if __name__` trick.

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

Some quick observations before you get to the good stuff. First, parentheses are not required around the `if` expression. Second, the `if` statement ends with a colon, and is followed by indented code.

Like C, Python uses `==` for comparison and `=` for assignment. Unlike C, Python does not support in-line assignment, so there's no chance of accidentally assigning the value you thought you were comparing.

So why is this particular `if` statement a trick? Modules are objects, and all modules have a built-in attribute `__name__`. A module's `__name__` depends on how you're using the module. If you `import` the module, then `__name__` is the module's filename, without a directory path or file extension. But you can also run the module directly as a standalone program, in which case `__name__` will be a special default value, `__main__`.

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

Knowing this, you can design a test suite for your module within the module itself by putting it in this `if` statement. When you run the module directly, `__name__` is `__main__`, so the test suite executes. When you import the module, `__name__` is something else, so the test suite is ignored. This makes it easier to develop and debug new modules before integrating them into a larger program.

On MacPython, there is an additional step to make the `if __name__` trick work. Pop up the module's options menu by clicking the black triangle in the upper-right corner of the window, and make sure `Run as __main__` is checked.

### Further Reading on Importing Modules

- *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) discusses the low-level details of importing modules (<http://www.python.org/doc/current/ref/import.html>).

# Chapter 3. Native Datatypes

You'll get back to your first Python program in just a minute. But first, a short digression is in order, because you need to know about dictionaries, tuples, and lists (oh my!). If you're a Perl hacker, you can probably skim the bits about dictionaries and lists, but you should still pay attention to tuples.

## 3.1. Introducing Dictionaries

One of Python's built-in datatypes is the dictionary, which defines one-to-one relationships between keys and values.

A dictionary in Python is like a hash in Perl. In Perl, variables that store hashes always start with a % character. In Python, variables can be named anything, and Python keeps track of the datatype internally.

A dictionary in Python is like an instance of the `Hashtable` class in Java.

A dictionary in Python is like an instance of the `Scripting.Dictionary` object in Visual Basic.

### 3.1.1. Defining Dictionaries

#### Example 3.1. Defining a Dictionary

```
>>> d = {"server": "mpilgrim", "database": "master"} ❶
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"] ❷
'mpilgrim'
>>> d["database"] ❸
'master'
>>> d["mpilgrim"] ❹
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```

- ❶ First, you create a new dictionary with two elements and assign it to the variable `d`. Each element is a key-value pair, and the whole set of elements is enclosed in curly braces.
- ❷ `'server'` is a key, and its associated value, referenced by `d["server"]`, is `'mpilgrim'`.
- ❸ `'database'` is a key, and its associated value, referenced by `d["database"]`, is `'master'`.
- ❹ You can get values by key, but you can't get keys by value. So `d["server"]` is `'mpilgrim'`, but `d["mpilgrim"]` raises an exception, because `'mpilgrim'` is not a key.

### 3.1.2. Modifying Dictionaries

#### Example 3.2. Modifying a Dictionary

```
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["database"] = "pubs" ❶
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa" ❷
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

- ❶ You can not have duplicate keys in a dictionary. Assigning a value to an existing key will wipe out the old value.
- ❷ You can add new key–value pairs at any time. This syntax is identical to modifying existing values. (Yes, this will annoy you someday when you think you are adding new values but are actually just modifying the same value over and over because your key isn't changing the way you think it is.)

Note that the new element (key 'uid', value 'sa') appears to be in the middle. In fact, it was just a coincidence that the elements appeared to be in order in the first example; it is just as much a coincidence that they appear to be out of order now.

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered. This is an important distinction that will annoy you when you want to access the elements of a dictionary in a specific, repeatable order (like alphabetical order by key). There are ways of doing this, but they're not built into the dictionary.

When working with dictionaries, you need to be aware that dictionary keys are case–sensitive.

### Example 3.3. Dictionary Keys Are Case–Sensitive

```
>>> d = {}
>>> d["key"] = "value"
>>> d["key"] = "other value" ❶
>>> d
{'key': 'other value'}
>>> d["Key"] = "third value" ❷
>>> d
{'Key': 'third value', 'key': 'other value'}
```

- ❶ Assigning a value to an existing dictionary key simply replaces the old value with a new one.
- ❷ This is not assigning a value to an existing dictionary key, because strings in Python are case–sensitive, so 'key' is not the same as 'Key'. This creates a new key/value pair in the dictionary; it may look similar to you, but as far as Python is concerned, it's completely different.

### Example 3.4. Mixing Datatypes in a Dictionary

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
>>> d["retrycount"] = 3 ❶
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d[42] = "douglas" ❷
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
42: 'douglas', 'retrycount': 3}
```

- ❶ Dictionaries aren't just for strings. Dictionary values can be any datatype, including strings, integers, objects, or even other dictionaries. And within a single dictionary, the values don't all need to be the same type; you can mix and match as needed.
- ❷ Dictionary keys are more restricted, but they can be strings, integers, and a few other types. You can also mix and match key datatypes within a dictionary.

### 3.1.3. Deleting Items From Dictionaries

#### Example 3.5. Deleting Items from a Dictionary

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
 42: 'douglas', 'retrycount': 3}
>>> del d[42] ❶
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d.clear() ❷
>>> d
{}
```

- ❶ `del` lets you delete individual items from a dictionary by key.
- ❷ `clear` deletes all items from a dictionary. Note that the set of empty curly braces signifies a dictionary without any items.

#### Further Reading on Dictionaries

- *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) teaches about dictionaries and shows how to use dictionaries to model sparse matrices (<http://www.ibiblio.org/obp/thinkCSpy/chap10.htm>).
- Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) has a lot of example code using dictionaries (<http://www.faqs.com/knowledge-base/index.phtml/fid/541/>).
- Python Cookbook (<http://www.activestate.com/ASPN/Python/Cookbook/>) discusses how to sort the values of a dictionary by key (<http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52306>).
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the dictionary methods (<http://www.python.org/doc/current/lib/typesmapping.html>).

## 3.2. Introducing Lists

Lists are Python's workhorse datatype. If your only experience with lists is arrays in Visual Basic or (God forbid) the datastore in Powerbuilder, brace yourself for Python lists.

A list in Python is like an array in Perl. In Perl, variables that store arrays always start with the `@` character; in Python, variables can be named anything, and Python keeps track of the datatype internally.

A list in Python is much more than an array in Java (although it can be used as one if that's really all you want out of life). A better analogy would be to the `ArrayList` class, which can hold arbitrary objects and can expand dynamically as new items are added.

### 3.2.1. Defining Lists

#### Example 3.6. Defining a List

```
>>> li = ["a", "b", "mpilgrim", "z", "example"] ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[0] ❷
'a'
>>> li[4] ❸
'example'
```

- ❶ First, you define a list of five elements. Note that they retain their original order. This is not an accident. A list is an ordered set of elements enclosed in square brackets.
- ❷ A list can be used like a zero-based array. The first element of any non-empty list is always `li[0]`.
- ❸ The last element of this five-element list is `li[4]`, because lists are always zero-based.

### Example 3.7. Negative List Indices

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[-1] ❶
'example'
>>> li[-3] ❷
'mpilgrim'
```

- ❶ A negative index accesses elements from the end of the list counting backwards. The last element of any non-empty list is always `li[-1]`.
- ❷ If the negative index is confusing to you, think of it this way: `li[-n] == li[len(li) - n]`. So in this list, `li[-3] == li[5 - 3] == li[2]`.

### Example 3.8. Slicing a List

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[1:3] ❶
['b', 'mpilgrim']
>>> li[1:-1] ❷
['b', 'mpilgrim', 'z']
>>> li[0:3] ❸
['a', 'b', 'mpilgrim']
```

- ❶ You can get a subset of a list, called a "slice", by specifying two indices. The return value is a new list containing all the elements of the list, in order, starting with the first slice index (in this case `li[1]`), up to but not including the second slice index (in this case `li[3]`).
- ❷ Slicing works if one or both of the slice indices is negative. If it helps, you can think of it this way: reading the list from left to right, the first slice index specifies the first element you want, and the second slice index specifies the first element you don't want. The return value is everything in between.
- ❸ Lists are zero-based, so `li[0:3]` returns the first three elements of the list, starting at `li[0]`, up to but not including `li[3]`.

### Example 3.9. Slicing Shorthand

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[:3] ❶
['a', 'b', 'mpilgrim']
>>> li[3:] ❷ ❸
['z', 'example']
>>> li[:] ❹
['a', 'b', 'mpilgrim', 'z', 'example']
```

- ❶ If the left slice index is 0, you can leave it out, and 0 is implied. So `li[:3]` is the same as `li[0:3]` from Example 3.8, Slicing a List.
- ❷ Similarly, if the right slice index is the length of the list, you can leave it out. So `li[3:]` is the same as `li[3:5]`, because this list has five elements.

- ❸ Note the symmetry here. In this five-element list, `li[:3]` returns the first 3 elements, and `li[3:]` returns the last two elements. In fact, `li[:n]` will always return the first `n` elements, and `li[n:]` will return the rest, regardless of the length of the list.
- ❹ If both slice indices are left out, all elements of the list are included. But this is not the same as the original `li` list; it is a new list that happens to have all the same elements. `li[:]` is shorthand for making a complete copy of a list.

### 3.2.2. Adding Elements to Lists

#### Example 3.10. Adding Elements to a List

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li.append("new") ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li.insert(2, "new") ❷
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> li.extend(["two", "elements"]) ❸
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
```

- ❶ `append` adds a single element to the end of the list.
- ❷ `insert` inserts a single element into a list. The numeric argument is the index of the first element that gets bumped out of position. Note that list elements do not need to be unique; there are now two separate elements with the value `'new'`, `li[2]` and `li[6]`.
- ❸ `extend` concatenates lists. Note that you do not call `extend` with multiple arguments; you call it with one argument, a list. In this case, that list has two elements.

#### Example 3.11. The Difference between `extend` and `append`

```
>>> li = ['a', 'b', 'c']
>>> li.extend(['d', 'e', 'f']) ❶
>>> li
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(li) ❷
6
>>> li[-1]
'f'
>>> li = ['a', 'b', 'c']
>>> li.append(['d', 'e', 'f']) ❸
>>> li
['a', 'b', 'c', ['d', 'e', 'f']]
>>> len(li) ❹
4
>>> li[-1]
['d', 'e', 'f']
```

- ❶ Lists have two methods, `extend` and `append`, that look like they do the same thing, but are in fact completely different. `extend` takes a single argument, which is always a list, and adds each of the elements of that list to the original list.
- ❷ Here you started with a list of three elements (`'a'`, `'b'`, and `'c'`), and you extended the list with a list of another three elements (`'d'`, `'e'`, and `'f'`), so you now have a list of six elements.
- ❸



On the other hand, `append` takes one argument, which can be any data type, and simply adds it to the end of the list. Here, you're calling the `append` method with a single argument, which is a list of three elements.

- ④ Now the original list, which started as a list of three elements, contains four elements. Why four? Because the last element that you just appended *is itself a list*. Lists can contain any type of data, including other lists. That may be what you want, or maybe not. Don't use `append` if you mean `extend`.

### 3.2.3. Searching Lists

#### Example 3.12. Searching a List

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.index("example") ❶
5
>>> li.index("new")      ❷
2
>>> li.index("c")        ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li            ❹
False
```

- ❶ `index` finds the first occurrence of a value in the list and returns the index.
- ❷ `index` finds the *first* occurrence of a value in the list. In this case, 'new' occurs twice in the list, in `li[2]` and `li[6]`, but `index` will return only the first index, 2.
- ❸ If the value is not found in the list, Python raises an exception. This is notably different from most languages, which will return some invalid index. While this may seem annoying, it is a good thing, because it means your program will crash at the source of the problem, rather than later on when you try to use the invalid index.
- ❹ To test whether a value is in the list, use `in`, which returns `True` if the value is found or `False` if it is not.

Before version 2.2.1, Python had no separate boolean datatype. To compensate for this, Python accepted almost anything in a boolean context (like an `if` statement), according to the following rules:

- 0 is false; all other numbers are true.
- An empty string (" ") is false, all other strings are true.
- An empty list ([ ]) is false; all other lists are true.
- An empty tuple (( )) is false; all other tuples are true.
- An empty dictionary ({ }) is false; all other dictionaries are true.

These rules still apply in Python 2.2.1 and beyond, but now you can also use an actual boolean, which has a value of `True` or `False`. Note the capitalization; these values, like everything else in Python, are case-sensitive.

### 3.2.4. Deleting List Elements

#### Example 3.13. Removing Elements from a List

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("z") ❶
>>> li
['a', 'b', 'new', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("new") ❷
```

```

>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("c") ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop() ❹
'elements'
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']

```

- ❶ `remove` removes the first occurrence of a value from a list.
- ❷ `remove` removes *only* the first occurrence of a value. In this case, 'new' appeared twice in the list, but `li.remove("new")` removed only the first occurrence.
- ❸ If the value is not found in the list, Python raises an exception. This mirrors the behavior of the `index` method.
- ❹ `pop` is an interesting beast. It does two things: it removes the last element of the list, and it returns the value that it removed. Note that this is different from `li[-1]`, which returns a value but does not change the list, and different from `li.remove(value)`, which changes the list but does not return a value.

### 3.2.5. Using List Operators

#### Example 3.14. List Operators

```

>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new'] ❶
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
>>> li += ['two'] ❷
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
>>> li = [1, 2] * 3 ❸
>>> li
[1, 2, 1, 2, 1, 2]

```

- ❶ Lists can also be concatenated with the `+` operator. `list = list + otherlist` has the same result as `list.extend(otherlist)`. But the `+` operator returns a new (concatenated) list as a value, whereas `extend` only alters an existing list. This means that `extend` is faster, especially for large lists.
- ❷ Python supports the `+=` operator. `li += ['two']` is equivalent to `li.extend(['two'])`. The `+=` operator works for lists, strings, and integers, and it can be overloaded to work for user-defined classes as well. (More on classes in Chapter 5.)
- ❸ The `*` operator works on lists as a repeater. `li = [1, 2] * 3` is equivalent to `li = [1, 2] + [1, 2] + [1, 2]`, which concatenates the three lists into one.

#### Further Reading on Lists

- *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) teaches about lists and makes an important point about passing lists as function arguments (<http://www.ibiblio.org/obp/thinkCSpy/chap08.htm>).
- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) shows how to use lists as stacks and queues (<http://www.python.org/doc/current/tut/node7.html#SECTION0071100000000000000000>).
- Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) answers common questions about lists (<http://www.faqs.com/knowledge-base/index.phtml/fid/534>) and has a lot of example code using lists (<http://www.faqs.com/knowledge-base/index.phtml/fid/540>).

- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the list methods (<http://www.python.org/doc/current/lib/typesseq-mutable.html>).

### 3.3. Introducing Tuples

A tuple is an immutable list. A tuple can not be changed in any way once it is created.

#### Example 3.15. Defining a tuple

```
>>> t = ("a", "b", "mpilgrim", "z", "example") ❶
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]                                         ❷
'a'
>>> t[-1]                                       ❸
'example'
>>> t[1:3]                                       ❹
('b', 'mpilgrim')
```

- ❶ A tuple is defined in the same way as a list, except that the whole set of elements is enclosed in parentheses instead of square brackets.
- ❷ The elements of a tuple have a defined order, just like a list. Tuples indices are zero-based, just like a list, so the first element of a non-empty tuple is always `t[0]`.
- ❸ Negative indices count from the end of the tuple, just as with a list.
- ❹ Slicing works too, just like a list. Note that when you slice a list, you get a new list; when you slice a tuple, you get a new tuple.

#### Example 3.16. Tuples Have No Methods

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new") ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z") ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("example") ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t ❹
True
```

- ❶ You can't add elements to a tuple. Tuples have no `append` or `extend` method.
- ❷ You can't remove elements from a tuple. Tuples have no `remove` or `pop` method.
- ❸ You can't find elements in a tuple. Tuples have no `index` method.
- ❹ You can, however, use `in` to see if an element exists in the tuple.

So what are tuples good for?

- Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is

iterate through it, use a tuple instead of a list.

- It makes your code safer if you "write-protect" data that does not need to be changed. Using a tuple instead of a list is like having an implied `assert` statement that shows this data is constant, and that special thought (and a specific function) is required to override that.
- Remember that I said that dictionary keys can be integers, strings, and "a few other types"? Tuples are one of those types. Tuples can be used as keys in a dictionary, but lists can't be used this way. Actually, it's more complicated than that. Dictionary keys must be immutable. Tuples themselves are immutable, but if you have a tuple of lists, that counts as mutable and isn't safe to use as a dictionary key. Only tuples of strings, numbers, or other dictionary-safe tuples can be used as dictionary keys.
- Tuples are used in string formatting, as you'll see shortly.

Tuples can be converted into lists, and vice-versa. The built-in `tuple` function takes a list and returns a tuple with the same elements, and the `list` function takes a tuple and returns a list. In effect, `tuple` freezes a list, and `list` thaws a tuple.

### Further Reading on Tuples

- *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) teaches about tuples and shows how to concatenate tuples (<http://www.ibiblio.org/obp/thinkCSpy/chap10.htm>).
- Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) shows how to sort a tuple (<http://www.faqs.com/knowledge-base/view.phtml/aid/4553/fid/587>).
- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) shows how to define a tuple with one element (<http://www.python.org/doc/current/tut/node7.html#SECTION00730000000000000000>).

## 3.4. Declaring variables

Now that you know something about dictionaries, tuples, and lists (oh my!), let's get back to the sample program from Chapter 2, `odbchelper.py`.

Python has local and global variables like most other languages, but it has no explicit variable declarations. Variables spring into existence by being assigned a value, and they are automatically destroyed when they go out of scope.

### Example 3.17. Defining the `myParams` Variable

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

Notice the indentation. An `if` statement is a code block and needs to be indented just like a function.

Also notice that the variable assignment is one command split over several lines, with a backslash ("`\`") serving as a line-continuation marker.

When a command is split among several lines with the line-continuation marker ("`\`"), the continued lines can be indented in any manner; Python's normally stringent indentation rules do not apply. If your Python IDE auto-indents the continued line, you should probably accept its default unless you have a burning reason not to.

Strictly speaking, expressions in parentheses, straight brackets, or curly braces (like defining a dictionary) can be split into multiple lines with or without the line continuation character ("`\`"). I like to include the backslash even when it's not required because I think it makes the code easier to read, but that's a matter of style.

Third, you never declared the variable `myParams`, you just assigned a value to it. This is like VBScript without the option `explicit` option. Luckily, unlike VBScript, Python will not allow you to reference a variable that has never been assigned a value; trying to do so will raise an exception.

### 3.4.1. Referencing Variables

#### Example 3.18. Referencing an Unbound Variable

```
>>> x
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'x'
>>> x = 1
>>> x
1
```

You will thank Python for this one day.

### 3.4.2. Assigning Multiple Values at Once

One of the cooler programming shortcuts in Python is using sequences to assign multiple values at once.

#### Example 3.19. Assigning multiple values at once

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

- ❶ `v` is a tuple of three elements, and `(x, y, z)` is a tuple of three variables. Assigning one to the other assigns each of the values of `v` to each of the variables, in order.

This has all sorts of uses. I often want to assign names to a range of values. In C, you would use `enum` and manually list each constant and its associated value, which seems especially tedious when the values are consecutive. In Python, you can use the built-in `range` function with multi-variable assignment to quickly assign consecutive values.

#### Example 3.20. Assigning Consecutive Values

```
>>> range(7) ❶
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7) ❷
>>> MONDAY ❸
0
>>> TUESDAY
1
>>> SUNDAY
```

- ❶ The built-in `range` function returns a list of integers. In its simplest form, it takes an upper limit and returns a zero-based list counting up to but not including the upper limit. (If you like, you can pass other parameters to specify a base other than 0 and a step other than 1. You can print `range.__doc__` for details.)
- ❷ `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY` are the variables you're defining. (This example came from the `calendar` module, a fun little module that prints calendars, like the UNIX program `cal`. The `calendar` module defines integer constants for days of the week.)
- ❸ Now each variable has its value: `MONDAY` is 0, `TUESDAY` is 1, and so forth.

You can also use multi-variable assignment to build functions that return multiple values, simply by returning a tuple of all the values. The caller can treat it as a tuple, or assign the values to individual variables. Many standard Python libraries do this, including the `os` module, which you'll discuss in Chapter 6.

### Further Reading on Variables

- *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) shows examples of when you can skip the line continuation character (<http://www.python.org/doc/current/ref/implicit-joining.html>) and when you need to use it (<http://www.python.org/doc/current/ref/explicit-joining.html>).
- *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) shows how to use multi-variable assignment to swap the values of two variables (<http://www.ibiblio.org/obp/thinkCSpy/chap09.htm>).

## 3.5. Formatting Strings

Python supports formatting values into strings. Although this can include very complicated expressions, the most basic usage is to insert values into a string with the `%s` placeholder.

String formatting in Python uses the same syntax as the `sprintf` function in C.

### Example 3.21. Introducing String Formatting

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

- ❶ The whole expression evaluates to a string. The first `%s` is replaced by the value of `k`; the second `%s` is replaced by the value of `v`. All other characters in the string (in this case, the equal sign) stay as they are.

Note that `(k, v)` is a tuple. I told you they were good for something.

You might be thinking that this is a lot of work just to do simple string concatenation, and you would be right, except that string formatting isn't just concatenation. It's not even just formatting. It's also type coercion.

### Example 3.22. String Formatting vs. Concatenating

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " is not a good password for " + uid ❶
secret is not a good password for sa
>>> print "%s is not a good password for %s" % (pwd, uid) ❷
```

```
secret is not a good password for sa
>>> userCount = 6
>>> print "Users connected: %d" % (userCount, )
Users connected: 6
>>> print "Users connected: " + userCount
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

- ❶ + is the string concatenation operator.
- ❷ In this trivial case, string formatting accomplishes the same result as concatenation.
- ❸ (userCount, ) is a tuple with one element. Yes, the syntax is a little strange, but there's a good reason for it: it's unambiguously a tuple. In fact, you can always include a comma after the last element when defining a list, tuple, or dictionary, but the comma is required when defining a tuple with one element. If the comma weren't required, Python wouldn't know whether (userCount) was a tuple with one element or just the value of userCount.
- ❹ String formatting works with integers by specifying %d instead of %s.
- ❺ Trying to concatenate a string with a non-string raises an exception. Unlike string formatting, string concatenation works only when everything is already a string.

As with printf in C, string formatting in Python is like a Swiss Army knife. There are options galore, and modifier strings to specially format many different types of values.

### Example 3.23. Formatting Numbers

```
>>> print "Today's stock price: %f" % 50.4625
50.462500
>>> print "Today's stock price: %.2f" % 50.4625
50.46
>>> print "Change since yesterday: %+.2f" % 1.5
+1.50
```

- ❶ The %f string formatting option treats the value as a decimal, and prints it to six decimal places.
- ❷ The ".2" modifier of the %f option truncates the value to two decimal places.
- ❸ You can even combine modifiers. Adding the + modifier displays a plus or minus sign before the value. Note that the ".2" modifier is still in place, and is padding the value to exactly two decimal places.

### Further Reading on String Formatting

- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the string formatting format characters (<http://www.python.org/doc/current/lib/typesseq-strings.html>).
- *Effective AWK Programming* ([http://www-gnats.gnu.org:8080/cgi-bin/info2www?\(gawk\)Top](http://www-gnats.gnu.org:8080/cgi-bin/info2www?(gawk)Top)) discusses all the format characters ([http://www-gnats.gnu.org:8080/cgi-bin/info2www?\(gawk\)Control+Letters](http://www-gnats.gnu.org:8080/cgi-bin/info2www?(gawk)Control+Letters)) and advanced string formatting techniques like specifying width, precision, and zero-padding ([http://www-gnats.gnu.org:8080/cgi-bin/info2www?\(gawk\)Format+Modifiers](http://www-gnats.gnu.org:8080/cgi-bin/info2www?(gawk)Format+Modifiers)).

## 3.6. Mapping Lists

One of the most powerful features of Python is the list comprehension, which provides a compact way of mapping a list into another list by applying a function to each of the elements of the list.

### Example 3.24. Introducing List Comprehensions

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li] ❶
[2, 18, 16, 8]
>>> li ❷
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li] ❸
>>> li
[2, 18, 16, 8]
```

- ❶ To make sense of this, look at it from right to left. `li` is the list you're mapping. Python loops through `li` one element at a time, temporarily assigning the value of each element to the variable `elem`. Python then applies the function `elem*2` and appends that result to the returned list.
- ❷ Note that list comprehensions do not change the original list.
- ❸ It is safe to assign the result of a list comprehension to the variable that you're mapping. Python constructs the new list in memory, and when the list comprehension is complete, it assigns the result to the variable.

Here are the list comprehensions in the `buildConnectionString` function that you declared in Chapter 2:

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

First, notice that you're calling the `items` function of the `params` dictionary. This function returns a list of tuples of all the data in the dictionary.

### Example 3.25. The `keys`, `values`, and `items` Functions

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.keys() ❶
['server', 'uid', 'database', 'pwd']
>>> params.values() ❷
['mpilgrim', 'sa', 'master', 'secret']
>>> params.items() ❸
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
```

- ❶ The `keys` method of a dictionary returns a list of all the keys. The list is not in the order in which the dictionary was defined (remember that elements in a dictionary are unordered), but it is a list.
- ❷ The `values` method returns a list of all the values. The list is in the same order as the list returned by `keys`, so `params.values()[n] == params[params.keys()[n]]` for all values of `n`.
- ❸ The `items` method returns a list of tuples of the form `(key, value)`. The list contains all the data in the dictionary.

Now let's see what `buildConnectionString` does. It takes a list, `params.items()`, and maps it to a new list by applying string formatting to each element. The new list will have the same number of elements as `params.items()`, but each element in the new list will be a string that contains both a key and its associated value from the `params` dictionary.

### Example 3.26. List Comprehensions in `buildConnectionString`, Step by Step

```
>>> 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()] ❶
['server', 'uid', 'database', 'pwd']
>>> [v for k, v in params.items()] ❷
```



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

- ❶ Note that you're using two variables to iterate through the `params.items()` list. This is another use of multi-variable assignment. The first element of `params.items()` is `('server', 'mpilgrim')`, so in the first iteration of the list comprehension, `k` will get `'server'` and `v` will get `'mpilgrim'`. In this case, you're ignoring the value of `v` and only including the value of `k` in the returned list, so this list comprehension ends up being equivalent to `params.keys()`.
- ❷ Here you're doing the same thing, but ignoring the value of `k`, so this list comprehension ends up being equivalent to `params.values()`.
- ❸ Combining the previous two examples with some simple string formatting, you get a list of strings that include both the key and value of each element of the dictionary. This looks suspiciously like the output of the program. All that remains is to join the elements in this list into a single string.

### Further Reading on List Comprehensions

- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses another way to map lists using the built-in `map` function (<http://www.python.org/doc/current/tut/node7.html#SECTION0071300000000000000000>).
- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) shows how to do nested list comprehensions (<http://www.python.org/doc/current/tut/node7.html#SECTION0071400000000000000000>).

## 3.7. Joining Lists and Splitting Strings

You have a list of key–value pairs in the form `key=value`, and you want to join them into a single string. To join any list of strings into a single string, use the `join` method of a string object.

Here is an example of joining a list from the `buildConnectionString` function:

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

One interesting note before you continue. I keep repeating that functions are objects, strings are objects... everything is an object. You might have thought I meant that string *variables* are objects. But no, look closely at this example and you'll see that the string `" ; "` itself is an object, and you are calling its `join` method.

The `join` method joins the elements of the list into a single string, with each element separated by a semi-colon. The delimiter doesn't need to be a semi-colon; it doesn't even need to be a single character. It can be any string.

`join` works only on lists of strings; it does not do any type coercion. Joining a list that has one or more non-string elements will raise an exception.

### Example 3.27. Output of `odbchelper.py`

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

This string is then returned from the `odbchelper` function and printed by the calling block, which gives you the output that you marveled at when you started reading this chapter.

You're probably wondering if there's an analogous method to split a string into a list. And of course there is, and it's called `split`.

### Example 3.28. Splitting a String

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

- ❶ `split` reverses `join` by splitting a string into a multi-element list. Note that the delimiter ("`;`") is stripped out completely; it does not appear in any of the elements of the returned list.
- ❷ `split` takes an optional second argument, which is the number of times to split. ("Ooooooh, optional arguments..." You'll learn how to do this in your own functions in the next chapter.)

`anystring.split(delimiter, 1)` is a useful technique when you want to search a string for a substring and then work with everything before the substring (which ends up in the first element of the returned list) and everything after it (which ends up in the second element).

### Further Reading on String Methods

- Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) answers common questions about strings (<http://www.faqs.com/knowledge-base/index.phtml/fid/480/>) and has a lot of example code using strings (<http://www.faqs.com/knowledge-base/index.phtml/fid/539/>).
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the string methods (<http://www.python.org/doc/current/lib/string-methods.html>).
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `string` module (<http://www.python.org/doc/current/lib/module-string.html>).
- *The Whole Python FAQ* (<http://www.python.org/doc/FAQ.html>) explains why `join` is a string method (<http://www.python.org/cgi-bin/faqw.py?query=4.96&querytype=simple&casefold=yes&req=search>) instead of a list method.

### 3.7.1. Historical Note on String Methods

When I first learned Python, I expected `join` to be a method of a list, which would take the delimiter as an argument. Many people feel the same way, and there's a story behind the `join` method. Prior to Python 1.6, strings didn't have all these useful methods. There was a separate `string` module that contained all the string functions; each function took a string as its first argument. The functions were deemed important enough to put onto the strings themselves, which made sense for functions like `lower`, `upper`, and `split`. But many hard-core Python programmers objected to the new `join` method, arguing that it should be a method of the list instead, or that it shouldn't move at all but simply stay a part of the old `string` module (which still has a lot of useful stuff in it). I use the new `join` method exclusively, but you will see code written either way, and if it really bothers you, you can use the old `string.join` function instead.

## 3.8. Summary

The `odbcHelper.py` program and its output should now make perfect sense.

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.
```

```

Returns string."""
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)

```

Here is the output of `odbc helper.py`:

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

Before diving into the next chapter, make sure you're comfortable doing all of these things:

- Using the Python IDE to test expressions interactively
- Writing Python programs and running them from within your IDE, or from the command line
- Importing modules and calling their functions
- Declaring functions and using `doc strings`, local variables, and proper indentation
- Defining dictionaries, tuples, and lists
- Accessing attributes and methods of any object, including strings, lists, dictionaries, functions, and modules
- Concatenating values through string formatting
- Mapping lists into other lists using list comprehensions
- Splitting strings into lists and joining lists into strings

# Chapter 4. The Power Of Introspection

This chapter covers one of Python's strengths: introspection. As you know, everything in Python is an object, and introspection is code looking at other modules and functions in memory as objects, getting information about them, and manipulating them. Along the way, you'll define functions with no name, call functions with arguments out of order, and reference functions whose names you don't even know ahead of time.

## 4.1. Diving In

Here is a complete, working Python program. You should understand a good deal about it just by looking at it. The numbered lines illustrate concepts covered in Chapter 2, *Your First Python Program*. Don't worry if the rest of the code looks intimidating; you'll learn all about it throughout this chapter.

### Example 4.1. `apihelper.py`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
def info(object, spacing=10, collapse=1): ❶ ❷ ❸
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])

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

- ❶ This module has one function, `info`. According to its function declaration, it takes three parameters: `object`, `spacing`, and `collapse`. The last two are actually optional parameters, as you'll see shortly.
- ❷ The `info` function has a multi-line `doc string` that succinctly describes the function's purpose. Note that no return value is mentioned; this function will be used solely for its effects, rather than its value.
- ❸ Code within the function is indented.
- ❹ The `if __name__` trick allows this program do something useful when run by itself, without interfering with its use as a module for other programs. In this case, the program simply prints out the `doc string` of the `info` function.
- ❺ `if` statements use `==` for comparison, and parentheses are not required.

The `info` function is designed to be used by you, the programmer, while working in the Python IDE. It takes any object that has functions or methods (like a module, which has functions, or a list, which has methods) and prints out the functions and their `doc strings`.

### Example 4.2. Sample Usage of `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(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence 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([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1

```

By default the output is formatted to be easy to read. Multi-line doc strings are collapsed into a single long line, but this option can be changed by specifying 0 for the *collapse* argument. If the function names are longer than 10 characters, you can specify a larger value for the *spacing* argument to make the output easier to read.

### Example 4.3. Advanced Usage of `apihelper.py`

```

>>> import odbchelper
>>> info(odbchelper)
buildConnectionString Build a connection string from a dictionary Returns string.
>>> info(odbchelper, 30)
buildConnectionString          Build a connection string from a dictionary Returns string.
>>> info(odbchelper, 30, 0)
buildConnectionString          Build a connection string from a dictionary
                                   Returns string.

```

## 4.2. Using Optional and Named Arguments

Python allows function arguments to have default values; if the function is called without the argument, the argument gets its default value. Furthermore, arguments can be specified in any order by using named arguments. Stored procedures in SQL Server Transact/SQL can do this, so if you're a SQL Server scripting guru, you can skim this part.

Here is an example of `info`, a function with two optional arguments:

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

`spacing` and `collapse` are optional, because they have default values defined. `object` is required, because it has no default value. If `info` is called with only one argument, `spacing` defaults to 10 and `collapse` defaults to 1. If `info` is called with two arguments, `collapse` still defaults to 1.

Say you want to specify a value for `collapse` but want to accept the default value for `spacing`. In most languages, you would be out of luck, because you would need to call the function with three arguments. But in Python, arguments can be specified by name, in any order.

### Example 4.4. Valid Calls of `info`

```

info(odbchelper)           ❶
info(odbchelper, 12)       ❷
info(odbchelper, collapse=0)  ❸
info(spacing=15, object=odbchelper)  ❹

```

- ❶ With only one argument, `spacing` gets its default value of 10 and `collapse` gets its default value of 1.
- ❷ With two arguments, `collapse` gets its default value of 1.

- ③ Here you are naming the `collapse` argument explicitly and specifying its value. `spacing` still gets its default value of 10.
- ④ Even required arguments (like `object`, which has no default value) can be named, and named arguments can appear in any order.

This looks totally whacked until you realize that arguments are simply a dictionary. The "normal" method of calling functions without argument names is actually just a shorthand where Python matches up the values with the argument names in the order they're specified in the function declaration. And most of the time, you'll call functions the "normal" way, but you always have the additional flexibility if you need it.

The only thing you need to do to call a function is specify a value (somehow) for each required argument; the manner and order in which you do that is up to you.

### Further Reading on Optional Arguments

- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses exactly when and how default arguments are evaluated (<http://www.python.org/doc/current/tut/node6.html#SECTION006710000000000000000>), which matters when the default value is a list or an expression with side effects.

## 4.3. Using `type`, `str`, `dir`, and Other Built-In Functions

Python has a small set of extremely useful built-in functions. All other functions are partitioned off into modules. This was actually a conscious design decision, to keep the core language from getting bloated like other scripting languages (cough cough, Visual Basic).

### 4.3.1. The `type` Function

The `type` function returns the datatype of any arbitrary object. The possible types are listed in the `types` module. This is useful for helper functions that can handle several types of data.

#### Example 4.5. Introducing `type`

```
>>> type(1)           ❶
<type 'int'>
>>> li = []
>>> type(li)          ❷
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)  ❸
<type 'module'>
>>> import types      ❹
>>> type(odbchelper) == types.ModuleType
True
```

- ❶ `type` takes anything — and I mean anything — and returns its datatype. Integers, strings, lists, dictionaries, tuples, functions, classes, modules, even types are acceptable.
- ❷ `type` can take a variable and return its datatype.
- ❸ `type` also works on modules.
- ❹ You can use the constants in the `types` module to compare types of objects. This is what the `info` function does, as you'll see shortly.

### 4.3.2. The `str` Function

The `str` coerces data into a string. Every datatype can be coerced into a string.

#### Example 4.6. Introducing `str`

```
>>> str(1) ❶
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen
['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
>>> str(horsemen) ❷
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbc helper) ❸
"<module 'odbc helper' from 'c:\\docbook\\dip\\py\\odbc helper.py'>"
>>> str(None) ❹
'None'
```

- ❶ For simple datatypes like integers, you would expect `str` to work, because almost every language has a function to convert an integer to a string.
- ❷ However, `str` works on any object of any type. Here it works on a list which you've constructed in bits and pieces.
- ❸ `str` also works on modules. Note that the string representation of the module includes the pathname of the module on disk, so yours will be different.
- ❹ A subtle but important behavior of `str` is that it works on `None`, the Python null value. It returns the string `'None'`. You'll use this to your advantage in the `info` function, as you'll see shortly.

At the heart of the `info` function is the powerful `dir` function. `dir` returns a list of the attributes and methods of any object: modules, functions, strings, lists, dictionaries... pretty much anything.

#### Example 4.7. Introducing `dir`

```
>>> li = []
>>> dir(li) ❶
['append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
>>> d = {}
>>> dir(d) ❷
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'setdefault', 'update', 'values']
>>> import odbc helper
>>> dir(odbc helper) ❸
['__builtins__', '__doc__', '__file__', '__name__', 'buildConnectionString']
```

- ❶ `li` is a list, so `dir(li)` returns a list of all the methods of a list. Note that the returned list contains the names of the methods as strings, not the methods themselves.
- ❷ `d` is a dictionary, so `dir(d)` returns a list of the names of dictionary methods. At least one of these, `keys`, should look familiar.
- ❸ This is where it really gets interesting. `odbc helper` is a module, so `dir(odbc helper)` returns a list of all kinds of stuff defined in the module, including built-in attributes, like `__name__`, `__doc__`, and whatever other attributes and methods you define. In this case, `odbc helper` has only one user-defined method, the `buildConnectionString` function described in Chapter 2.

Finally, the callable function takes any object and returns `True` if the object can be called, or `False` otherwise.

Callable objects include functions, class methods, even classes themselves. (More on classes in the next chapter.)

### Example 4.8. Introducing callable

```
>>> import string
>>> string.punctuation ❶
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join ❷
<function join at 00C55A7C>
>>> callable(string.punctuation) ❸
False
>>> callable(string.join) ❹
True
>>> print string.join.__doc__ ❺
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)
```

- ❶ The functions in the `string` module are deprecated (although many people still use the `join` function), but the module contains a lot of useful constants like this `string.punctuation`, which contains all the standard punctuation characters.
- ❷ `string.join` is a function that joins a list of strings.
- ❸ `string.punctuation` is not callable; it is a string. (A string does have callable methods, but the string itself is not callable.)
- ❹ `string.join` is callable; it's a function that takes two arguments.
- ❺ Any callable object may have a `doc` string. By using the `callable` function on each of an object's attributes, you can determine which attributes you care about (methods, functions, classes) and which you want to ignore (constants and so on) without knowing anything about the object ahead of time.

### 4.3.3. Built-In Functions

`type`, `str`, `dir`, and all the rest of Python's built-in functions are grouped into a special module called `__builtin__`. (That's two underscores before and after.) If it helps, you can think of Python automatically executing `from __builtin__ import *` on startup, which imports all the "built-in" functions into the namespace so you can use them directly.

The advantage of thinking like this is that you can access all the built-in functions and attributes as a group by getting information about the `__builtin__` module. And guess what, Python has a function called `info`. Try it yourself and skim through the list now. We'll dive into some of the more important functions later. (Some of the built-in error classes, like `AttributeError`, should already look familiar.)

### Example 4.9. Built-in Attributes and Functions

```
>>> 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.

[...snip...]

Python comes with excellent reference manuals, which you should peruse thoroughly to learn all the modules Python has to offer. But unlike most languages, where you would find yourself referring back to the manuals or man pages to remind yourself how to use these modules, Python is largely self-documenting.

### Further Reading on Built-In Functions

- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents all the built-in functions (<http://www.python.org/doc/current/lib/built-in-funcs.html>) and all the built-in exceptions (<http://www.python.org/doc/current/lib/module-exceptions.html>).

## 4.4. Getting Object References With `getattr`

You already know that Python functions are objects. What you don't know is that you can get a reference to a function without knowing its name until run-time, by using the `getattr` function.

### Example 4.10. Introducing `getattr`

```
>>> li = ["Larry", "Curly"]
>>> li.pop
<built-in method pop of list object at 010DF884>
>>> getattr(li, "pop")
<built-in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe")
>>> li
["Larry", "Curly", "Moe"]
>>> getattr({}, "clear")
<built-in method clear of dictionary object at 00F113D4>
>>> getattr((), "pop")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

- 1 This gets a reference to the `pop` method of the list. Note that this is not calling the `pop` method; that would be `li.pop()`. This is the method itself.
- 2 This also returns a reference to the `pop` method, but this time, the method name is specified as a string argument to the `getattr` function. `getattr` is an incredibly useful built-in function that returns any attribute of any object. In this case, the object is a list, and the attribute is the `pop` method.
- 3 In case it hasn't sunk in just how incredibly useful this is, try this: the return value of `getattr` is the method, which you can then call just as if you had said `li.append("Moe")` directly. But you didn't call the function directly; you specified the function name as a string instead.
- 4 `getattr` also works on dictionaries.
- 5 In theory, `getattr` would work on tuples, except that tuples have no methods, so `getattr` will raise an exception no matter what attribute name you give.

### 4.4.1. getattr with Modules

getattr isn't just for built-in datatypes. It also works on modules.

#### Example 4.11. The getattr Function in apihelper.py

```
>>> import odbchelper
>>> odbchelper.buildConnectionString ❶
<function buildConnectionString at 00D18DD4>
>>> getattr(odbchelper, "buildConnectionString") ❷
<function buildConnectionString at 00D18DD4>
>>> object = odbchelper
>>> method = "buildConnectionString"
>>> getattr(object, method) ❸
<function buildConnectionString at 00D18DD4>
>>> type(getattr(object, method)) ❹
<type 'function'>
>>> import types
>>> type(getattr(object, method)) == types.FunctionType
True
>>> callable(getattr(object, method)) ❺
True
```

- ❶ This returns a reference to the `buildConnectionString` function in the `odbchelper` module, which you studied in Chapter 2, *Your First Python Program*. (The hex address you see is specific to my machine; your output will be different.)
- ❷ Using `getattr`, you can get the same reference to the same function. In general, `getattr(object, "attribute")` is equivalent to `object.attribute`. If `object` is a module, then `attribute` can be anything defined in the module: a function, class, or global variable.
- ❸ And this is what you actually use in the `info` function. `object` is passed into the function as an argument; `method` is a string which is the name of a method or function.
- ❹ In this case, `method` is the name of a function, which you can prove by getting its type.
- ❺ Since `method` is a function, it is callable.

### 4.4.2. getattr As a Dispatcher

A common usage pattern of `getattr` is as a dispatcher. For example, if you had a program that could output data in a variety of different formats, you could define separate functions for each output format and use a single dispatch function to call the right one.

For example, let's imagine a program that prints site statistics in HTML, XML, and plain text formats. The choice of output format could be specified on the command line, or stored in a configuration file. A `statsout` module defines three functions, `output_html`, `output_xml`, and `output_text`. Then the main program defines a single output function, like this:

#### Example 4.12. Creating a Dispatcher with getattr

```
import statsout

def output(data, format="text"):
    output_function = getattr(statsout, "output_%s" % format) ❶
    return output_function(data) ❷ ❸
```

- ❶ The output function takes one required argument, `data`, and one optional argument, `format`. If `format` is not specified, it defaults to `text`, and you will end up calling the plain text output function.
- ❷ You concatenate the `format` argument with `"output_"` to produce a function name, and then go get that function from the `statsout` module. This allows you to easily extend the program later to support other output formats, without changing this dispatch function. Just add another function to `statsout` named, for instance, `output_pdf`, and pass `"pdf"` as the `format` into the output function.
- ❸ Now you can simply call the output function in the same way as any other function. The `output_function` variable is a reference to the appropriate function from the `statsout` module.

Did you see the bug in the previous example? This is a very loose coupling of strings and functions, and there is no error checking. What happens if the user passes in a format that doesn't have a corresponding function defined in `statsout`? Well, `getattr` will return `None`, which will be assigned to `output_function` instead of a valid function, and the next line that attempts to call that function will crash and raise an exception. That's bad.

Luckily, `getattr` takes an optional third argument, a default value.

### Example 4.13. `getattr` Default Values

```
import statsout

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

- ❶ This function call is guaranteed to work, because you added a third argument to the call to `getattr`. The third argument is a default value that is returned if the attribute or method specified by the second argument wasn't found.

As you can see, `getattr` is quite powerful. It is the heart of introspection, and you'll see even more powerful examples of it in later chapters.

## 4.5. Filtering Lists

As you know, Python has powerful capabilities for mapping lists into other lists, via list comprehensions (Section 3.6, Mapping Lists ). This can be combined with a filtering mechanism, where some elements in the list are mapped while others are skipped entirely.

Here is the list filtering syntax:

```
[mapping-expression for element in source-list if filter-expression]
```

This is an extension of the list comprehensions that you know and love. The first two thirds are the same; the last part, starting with the `if`, is the filter expression. A filter expression can be any expression that evaluates true or false (which in Python can be almost anything). Any element for which the filter expression evaluates true will be included in the mapping. All other elements are ignored, so they are never put through the mapping expression and are not included in the output list.

### Example 4.14. Introducing List Filtering

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1] ❶
['mpilgrim', 'foo']
>>> [elem for elem in li if elem != "b"] ❷
```

```
['a', 'mpilgrim', 'foo', 'c', 'd', 'd']
>>> [elem for elem in li if li.count(elem) == 1] ❸
['a', 'mpilgrim', 'foo', 'c']
```

- ❶ The mapping expression here is simple (it just returns the value of each element), so concentrate on the filter expression. As Python loops through the list, it runs each element through the filter expression. If the filter expression is true, the element is mapped and the result of the mapping expression is included in the returned list. Here, you are filtering out all the one-character strings, so you're left with a list of all the longer strings.
- ❷ Here, you are filtering out a specific value, b. Note that this filters all occurrences of b, since each time it comes up, the filter expression will be false.
- ❸ `count` is a list method that returns the number of times a value occurs in a list. You might think that this filter would eliminate duplicates from a list, returning a list containing only one copy of each value in the original list. But it doesn't, because values that appear twice in the original list (in this case, b and d) are excluded completely. There are ways of eliminating duplicates from a list, but filtering is not the solution.

Let's get back to this line from `apihelper.py`:

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

This looks complicated, and it is complicated, but the basic structure is the same. The whole filter expression returns a list, which is assigned to the `methodList` variable. The first half of the expression is the list mapping part. The mapping expression is an identity expression, which it returns the value of each element. `dir(object)` returns a list of `object`'s attributes and methods — that's the list you're mapping. So the only new part is the filter expression after the `if`.

The filter expression looks scary, but it's not. You already know about `callable`, `getattr`, and `in`. As you saw in the previous section, the expression `getattr(object, method)` returns a function object if `object` is a module and `method` is the name of a function in that module.

So this expression takes an object (named `object`). Then it gets a list of the names of the object's attributes, methods, functions, and a few other things. Then it filters that list to weed out all the stuff that you don't care about. You do the weeding out by taking the name of each attribute/method/function and getting a reference to the real thing, via the `getattr` function. Then you check to see if that object is callable, which will be any methods and functions, both built-in (like the `pop` method of a list) and user-defined (like the `buildConnectionString` function of the `odbc helper` module). You don't care about other attributes, like the `__name__` attribute that's built in to every module.

## Further Reading on Filtering Lists

- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses another way to filter lists using the built-in `filter` function (<http://www.python.org/doc/current/tut/node7.html#SECTION0071300000000000000000>).

## 4.6. The Peculiar Nature of `and` and `or`

In Python, `and` and `or` perform boolean logic as you would expect, but they do not return boolean values; instead, they return one of the actual values they are comparing.

### Example 4.15. Introducing `and`

```
>>> 'a' and 'b' ❶
```

```
'b'
>>> '' and 'b'      ❷
''
>>> 'a' and 'b' and 'c' ❸
'c'
```

- ❶ When using `and`, values are evaluated in a boolean context from left to right. `0`, `''`, `[]`, `()`, `{}`, and `None` are false in a boolean context; everything else is true. Well, almost everything. By default, instances of classes are true in a boolean context, but you can define special methods in your class to make an instance evaluate to false. You'll learn all about classes and special methods in Chapter 5. If all values are true in a boolean context, `and` returns the last value. In this case, `and` evaluates `'a'`, which is true, then `'b'`, which is true, and returns `'b'`.
- ❷ If any value is false in a boolean context, `and` returns the first false value. In this case, `''` is the first false value.
- ❸ All values are true, so `and` returns the last value, `'c'`.

#### Example 4.16. Introducing `or`

```
>>> 'a' or 'b'      ❶
'a'
>>> '' or 'b'      ❷
'b'
>>> '' or [] or {} ❸
{}
>>> def sidefx():
...     print "in sidefx()"
...     return 1
>>> 'a' or sidefx() ❹
'a'
```

- ❶ When using `or`, values are evaluated in a boolean context from left to right, just like `and`. If any value is true, `or` returns that value immediately. In this case, `'a'` is the first true value.
- ❷ `or` evaluates `''`, which is false, then `'b'`, which is true, and returns `'b'`.
- ❸ If all values are false, `or` returns the last value. `or` evaluates `''`, which is false, then `[]`, which is false, then `{}`, which is false, and returns `{}`.
- ❹ Note that `or` evaluates values only until it finds one that is true in a boolean context, and then it ignores the rest. This distinction is important if some values can have side effects. Here, the function `sidefx` is never called, because `or` evaluates `'a'`, which is true, and returns `'a'` immediately.

If you're a C hacker, you are certainly familiar with the `bool ? a : b` expression, which evaluates to `a` if `bool` is true, and `b` otherwise. Because of the way `and` and `or` work in Python, you can accomplish the same thing.

### 4.6.1. Using the `and-or` Trick

#### Example 4.17. Introducing the `and-or` Trick

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b ❶
'first'
>>> 0 and a or b ❷
'second'
```

❶

This syntax looks similar to the `bool ? a : b` expression in C. The entire expression is evaluated from left to right, so the `and` is evaluated first. `1 and 'first'` evaluates to `'first'`, then `'first' or 'second'` evaluates to `'first'`.

❷ `0 and 'first'` evaluates to `False`, and then `0 or 'second'` evaluates to `'second'`.

However, since this Python expression is simply boolean logic, and not a special construct of the language, there is one extremely important difference between this `and-or` trick in Python and the `bool ? a : b` syntax in C. If the value of `a` is false, the expression will not work as you would expect it to. (Can you tell I was bitten by this? More than once?)

#### Example 4.18. When the `and-or` Trick Fails

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b
'second'
```

❶

❶ Since `a` is an empty string, which Python considers false in a boolean context, `1 and ''` evaluates to `''`, and then `'' or 'second'` evaluates to `'second'`. Oops! That's not what you wanted.

The `and-or` trick, `bool and a or b`, will not work like the C expression `bool ? a : b` when `a` is false in a boolean context.

The real trick behind the `and-or` trick, then, is to make sure that the value of `a` is never false. One common way of doing this is to turn `a` into `[a]` and `b` into `[b]`, then taking the first element of the returned list, which will be either `a` or `b`.

#### Example 4.19. Using the `and-or` Trick Safely

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

❶

❶ Since `[a]` is a non-empty list, it is never false. Even if `a` is `0` or `''` or some other false value, the list `[a]` is true because it has one element.

By now, this trick may seem like more trouble than it's worth. You could, after all, accomplish the same thing with an `if` statement, so why go through all this fuss? Well, in many cases, you are choosing between two constant values, so you can use the simpler syntax and not worry, because you know that the `a` value will always be true. And even if you need to use the more complicated safe form, there are good reasons to do so. For example, there are some cases in Python where `if` statements are not allowed, such as in `lambda` functions.

#### Further Reading on the `and-or` Trick

- Python Cookbook (<http://www.activestate.com/ASPN/Python/Cookbook/>) discusses alternatives to the `and-or` trick (<http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52310>).

## 4.7. Using `lambda` Functions

Python supports an interesting syntax that lets you define one-line mini-functions on the fly. Borrowed from Lisp, these so-called `lambda` functions can be used anywhere a function is required.

### Example 4.20. Introducing lambda Functions

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

- ❶ This is a lambda function that accomplishes the same thing as the normal function above it. Note the abbreviated syntax here: there are no parentheses around the argument list, and the `return` keyword is missing (it is implied, since the entire function can only be one expression). Also, the function has no name, but it can be called through the variable it is assigned to.
- ❷ You can use a lambda function without even assigning it to a variable. This may not be the most useful thing in the world, but it just goes to show that a lambda is just an in-line function.

To generalize, a lambda function is a function that takes any number of arguments (including optional arguments) and returns the value of a single expression. lambda functions can not contain commands, and they can not contain more than one expression. Don't try to squeeze too much into a lambda function; if you need something more complex, define a normal function instead and make it as long as you want.

lambda functions are a matter of style. Using them is never required; anywhere you could use them, you could define a separate normal function and use that instead. I use them in places where I want to encapsulate specific, non-reusable code without littering my code with a lot of little one-line functions.

#### 4.7.1. Real-World lambda Functions

Here are the lambda functions in `apihelper.py`:

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

Notice that this uses the simple form of the `and-or` trick, which is okay, because a lambda function is always true in a boolean context. (That doesn't mean that a lambda function can't return a false value. The function is always true; its return value could be anything.)

Also notice that you're using the `split` function with no arguments. You've already seen it used with one or two arguments, but without any arguments it splits on whitespace.

### Example 4.21. `split` With No Arguments

```
>>> s = "this is\na\ttest" ❶  
>>> print s  
this is  
a test  
>>> print s.split() ❷  
['this', 'is', 'a', 'test']  
>>> print " ".join(s.split()) ❸  
'this is a test'
```

❶

This is a multiline string, defined by escape characters instead of triple quotes. `\n` is a carriage return, and `\t` is a tab character.

- ❷ `split` without any arguments splits on whitespace. So three spaces, a carriage return, and a tab character are all the same.
- ❸ You can normalize whitespace by splitting a string with `split` and then rejoining it with `join`, using a single space as a delimiter. This is what the `info` function does to collapse multi-line doc strings into a single line.

So what is the `info` function actually doing with these lambda functions, splits, and and-or tricks?

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

`processFunc` is now a function, but which function it is depends on the value of the `collapse` variable. If `collapse` is true, `processFunc(string)` will collapse whitespace; otherwise, `processFunc(string)` will return its argument unchanged.

To do this in a less robust language, like Visual Basic, you would probably create a function that took a string and a `collapse` argument and used an `if` statement to decide whether to collapse the whitespace or not, then returned the appropriate value. This would be inefficient, because the function would need to handle every possible case. Every time you called it, it would need to decide whether to collapse whitespace before it could give you what you wanted. In Python, you can take that decision logic out of the function and define a lambda function that is custom-tailored to give you exactly (and only) what you want. This is more efficient, more elegant, and less prone to those nasty oh-I-thought-those-arguments-were-reversed kinds of errors.

### Further Reading on lambda Functions

- Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) discusses using lambda to call functions indirectly (<http://www.faqs.com/knowledge-base/view.phtml/aid/6081/fid/241>).
- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) shows how to access outside variables from inside a lambda function (<http://www.python.org/doc/current/tut/node6.html#SECTION0067400000000000000000>). (PEP 227 (<http://python.sourceforge.net/peps/pep-0227.html>) explains how this will change in future versions of Python.)
- *The Whole Python FAQ* (<http://www.python.org/doc/FAQ.html>) has examples of obfuscated one-liners using lambda (<http://www.python.org/cgi-bin/faqw.py?query=4.15&querytype=simple&casefold=yes&req=search>).

## 4.8. Putting It All Together

The last line of code, the only one you haven't deconstructed yet, is the one that does all the work. But by now the work is easy, because everything you need is already set up just the way you need it. All the dominoes are in place; it's time to knock them down.

This is the meat of `apihelper.py`:

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

Note that this is one command, split over multiple lines, but it doesn't use the line continuation character (`\`). Remember when I said that some expressions can be split into multiple lines without using a backslash? A list



comprehension is one of those expressions, since the entire expression is contained in square brackets.

Now, let's take it from the end and work backwards. The

```
for method in methodList
```

shows that this is a list comprehension. As you know, `methodList` is a list of all the methods you care about in object. So you're looping through that list with `method`.

#### Example 4.22. Getting a `doc string` Dynamically

```
>>> import odbchelper
>>> object = odbchelper
>>> method = 'buildConnectionString'
>>> getattr(object, method)
<function buildConnectionString at 010D6D74>
>>> print getattr(object, method).__doc__
Build a connection string from a dictionary of parameters.

Returns string.
```

- ❶ In the `info` function, `object` is the object you're getting help on, passed in as an argument.
- ❷ As you're looping through `methodList`, `method` is the name of the current method.
- ❸ Using the `getattr` function, you're getting a reference to the `method` function in the `object` module.
- ❹ Now, printing the actual `doc string` of the method is easy.

The next piece of the puzzle is the use of `str` around the `doc string`. As you may recall, `str` is a built-in function that coerces data into a string. But a `doc string` is always a string, so why bother with the `str` function? The answer is that not every function has a `doc string`, and if it doesn't, its `__doc__` attribute is `None`.

#### Example 4.23. Why Use `str` on a `doc string`?

```
>>> >>> def foo(): print 2
>>> >>> foo()
2
>>> >>> foo.__doc__
>>> foo.__doc__ == None
True
>>> str(foo.__doc__)
'None'
```

- ❶ You can easily define a function that has no `doc string`, so its `__doc__` attribute is `None`. Confusingly, if you evaluate the `__doc__` attribute directly, the Python IDE prints nothing at all, which makes sense if you think about it, but is still unhelpful.
- ❷ You can verify that the value of the `__doc__` attribute is actually `None` by comparing it directly.
- ❸ The `str` function takes the null value and returns a string representation of it, `'None'`.

In SQL, you must use `IS NULL` instead of `= NULL` to compare a null value. In Python, you can use either `== None` or `is None`, but `is None` is faster.

Now that you are guaranteed to have a string, you can pass the string to `processFunc`, which you have already defined as a function that either does or doesn't collapse whitespace. Now you see why it was important to use `str` to convert a `None` value into a string representation. `processFunc` is assuming a string argument and calling its `split` method, which would crash if you passed it `None` because `None` doesn't have a `split` method.

Stepping back even further, you see that you're using string formatting again to concatenate the return value of `processFunc` with the return value of method's `ljust` method. This is a new string method that you haven't seen before.

#### Example 4.24. Introducing `ljust`

```
>>> s = 'buildConnectionString'
>>> s.ljust(30) ❶
'buildConnectionString      '
>>> s.ljust(20) ❷
'buildConnectionString'
```

- ❶ `ljust` pads the string with spaces to the given length. This is what the `info` function uses to make two columns of output and line up all the doc strings in the second column.
- ❷ If the given length is smaller than the length of the string, `ljust` will simply return the string unchanged. It never truncates the string.

You're almost finished. Given the padded method name from the `ljust` method and the (possibly collapsed) doc string from the call to `processFunc`, you concatenate the two and get a single string. Since you're mapping `methodList`, you end up with a list of strings. Using the `join` method of the string `"\n"`, you join this list into a single string, with each element of the list on a separate line, and print the result.

#### Example 4.25. Printing a List

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

- ❶ This is also a useful debugging trick when you're working with lists. And in Python, you're always working with lists.

That's the last piece of the puzzle. You should now understand this code.

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

## 4.9. Summary

The `apihelper.py` program and its output should now make perfect sense.

```
def info(object, spacing=10, collapse=1):
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])

if __name__ == "__main__":
```

```
print info.__doc__
```

Here is the output of `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(list) -- extend list by appending list elements
index       L.index(value) -> integer -- return index of first occurrence 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([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

Before diving into the next chapter, make sure you're comfortable doing all of these things:

- Defining and calling functions with optional and named arguments
- Using `str` to coerce any arbitrary value into a string representation
- Using `getattr` to get references to functions and other attributes dynamically
- Extending the list comprehension syntax to do list filtering
- Recognizing the `and-or` trick and using it safely
- Defining `lambda` functions
- Assigning functions to variables and calling the function by referencing the variable. I can't emphasize this enough, because this mode of thought is vital to advancing your understanding of Python. You'll see more complex applications of this concept throughout this book.

# Chapter 5. Objects and Object–Orientation

This chapter, and pretty much every chapter after this, deals with object–oriented Python programming.

## 5.1. Diving In

Here is a complete, working Python program. Read the doc strings of the module, the classes, and the functions to get an overview of what this program does and how it works. As usual, don't worry about the stuff you don't understand; that's what the rest of the chapter is for.

### Example 5.1. fileinfo.py

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
"""Framework for getting filetype-specific metadata.

Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key-value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title" : ( 3, 33, stripnulls),
                  "artist" : ( 33, 63, stripnulls),
                  "album" : ( 63, 93, stripnulls),
                  "year" : ( 93, 97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre" : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        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 == "name" and item:
            self.__parse(item)
            FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    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__]):
        "get file info class from filename extension"
        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"]): ❶
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print

```

- ❶ This program's output depends on the files on your hard drive. To get meaningful output, you'll need to change the directory path to point to a directory of MP3 files on your own machine.

This is the output I got on my machine. Your output will be different, unless, by some startling coincidence, you share my exact taste in music.

```

album=
artist=Ghost in the Machine
title=A Time Long Forgotten (Concept
genre=3l
name=/music/_singles/a_time_long_forgotten_con.mp3
year=1999
comment=http://mp3.com/ghostmachine

album=Rave Mix
artist=***DJ MARY-JANE***
title=HELLRAISER***Trance from Hell
genre=3l
name=/music/_singles/hellraiser.mp3
year=2000
comment=http://mp3.com/DJMARYJANE

album=Rave Mix
artist=***DJ MARY-JANE***
title=KAIRO***THE BEST GOA
genre=3l
name=/music/_singles/kairo.mp3
year=2000

```

```
comment=http://mp3.com/DJMARYJANE

album=Journeys
artist=Masters of Balance
title=Long Way Home
genre=31
name=/music/_singles/long_way_home1.mp3
year=2000
comment=http://mp3.com/MastersofBalan

album=
artist=The Cynic Project
title=Sidewinder
genre=18
name=/music/_singles/sidewinder.mp3
year=2000
comment=http://mp3.com/cynicproject

album=Digitosis@128k
artist=VXpanded
title=Spinning
genre=255
name=/music/_singles/spinning.mp3
year=2000
comment=http://mp3.com/artists/95/vxp
```

## 5.2. Importing Modules Using `from module import`

Python has two ways of importing modules. Both are useful, and you should know when to use each. One way, `import module`, you've already seen in Section 2.4, *Everything Is an Object*. The other way accomplishes the same thing, but it has subtle and important differences.

Here is the basic `from module import` syntax:

```
from UserDict import UserDict
```

This is similar to the `import module` syntax that you know and love, but with an important difference: the attributes and methods of the imported module `types` are imported directly into the local namespace, so they are available directly, without qualification by module name. You can import individual items or use `from module import *` to import everything.

`from module import *` in Python is like use `module` in Perl; `import module` in Python is like `require module` in Perl.

`from module import *` in Python is like `import module.*` in Java; `import module` in Python is like `import module` in Java.

### Example 5.2. `import module` vs. `from module import`

```
>>> import types
>>> types.FunctionType
<type 'function'>
>>> FunctionType
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'FunctionType'
>>> from types import FunctionType
```

```
>>> FunctionType
<type 'function'>
```

④

- ❶ The `types` module contains no methods; it just has attributes for each Python object type. Note that the attribute, `FunctionType`, must be qualified by the module name, `types`.
- ❷ `FunctionType` by itself has not been defined in this namespace; it exists only in the context of `types`.
- ❸ This syntax imports the attribute `FunctionType` from the `types` module directly into the local namespace.
- ❹ Now `FunctionType` can be accessed directly, without reference to `types`.

When should you use `from module import`?

- If you will be accessing attributes and methods often and don't want to type the module name over and over, use `from module import`.
- If you want to selectively import some attributes and methods but not others, use `from module import`.
- If the module contains attributes or functions with the same name as ones in your module, you must use `import module` to avoid name conflicts.

Other than that, it's just a matter of style, and you will see Python code written both ways.

Use `from module import *` sparingly, because it makes it difficult to determine where a particular function or attribute came from, and that makes debugging and refactoring more difficult.

### Further Reading on Module Importing Techniques

- `eff-bot` (<http://www.effbot.org/guides/>) has more to say on `import module` vs. `from module import` (<http://www.effbot.org/guides/import-confusion.htm>).
- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses advanced import techniques, including `from module import *` (<http://www.python.org/doc/current/tut/node8.html#SECTION008410000000000000000>).

## 5.3. Defining Classes

Python is fully object-oriented: you can define your own classes, inherit from your own or built-in classes, and instantiate the classes you've defined.

Defining a class in Python is simple. As with functions, there is no separate interface definition. Just define the class and start coding. A Python class starts with the reserved word `class`, followed by the class name. Technically, that's all that's required, since a class doesn't need to inherit from any other class.

### Example 5.3. The Simplest Python Class

```
class Loaf: ❶
    pass    ❷ ❸
```

- ❶ The name of this class is `Loaf`, and it doesn't inherit from any other class. Class names are usually capitalized, `EachWordLikeThis`, but this is only a convention, not a requirement.
- ❷ This class doesn't define any methods or attributes, but syntactically, there needs to be something in the definition, so you use `pass`. This is a Python reserved word that just means "move along, nothing to see here". It's a statement that does nothing, and it's a good placeholder when you're stubbing out functions or classes.

- ③ You probably guessed this, but everything in a class is indented, just like the code within a function, `if` statement, `for` loop, and so forth. The first thing not indented is not in the class.

The `pass` statement in Python is like an empty set of braces (`{ }`) in Java or C.

Of course, realistically, most classes will be inherited from other classes, and they will define their own class methods and attributes. But as you've just seen, there is nothing that a class absolutely must have, other than a name. In particular, C++ programmers may find it odd that Python classes don't have explicit constructors and destructors. Python classes do have something similar to a constructor: the `__init__` method.

### Example 5.4. Defining the `FileInfo` Class

```
from UserDict import UserDict

class FileInfo(UserDict): ①
```

- ① In Python, the ancestor of a class is simply listed in parentheses immediately after the class name. So the `FileInfo` class is inherited from the `UserDict` class (which was imported from the `UserDict` module). `UserDict` is a class that acts like a dictionary, allowing you to essentially subclass the dictionary datatype and add your own behavior. (There are similar classes `UserList` and `UserString` which allow you to subclass lists and strings.) There is a bit of black magic behind this, which you will demystify later in this chapter when you explore the `UserDict` class in more depth.

In Python, the ancestor of a class is simply listed in parentheses immediately after the class name. There is no special keyword like `extends` in Java.

Python supports multiple inheritance. In the parentheses following the class name, you can list as many ancestor classes as you like, separated by commas.

### 5.3.1. Initializing and Coding Classes

This example shows the initialization of the `FileInfo` class using the `__init__` method.

### Example 5.5. Initializing the `FileInfo` Class

```
class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None): ② ③ ④
```

- ① Classes can (and should) have doc strings too, just like modules and functions.
- ② `__init__` is called immediately after an instance of the class is created. It would be tempting but incorrect to call this the constructor of the class. It's tempting, because it looks like a constructor (by convention, `__init__` is the first method defined for the class), acts like one (it's the first piece of code executed in a newly created instance of the class), and even sounds like one ("init" certainly suggests a constructor-ish nature). Incorrect, because the object has already been constructed by the time `__init__` is called, and you already have a valid reference to the new instance of the class. But `__init__` is the closest thing you're going to get to a constructor in Python, and it fills much the same role.
- ③ The first argument of every class method, including `__init__`, is always a reference to the current instance of the class. By convention, this argument is always named `self`. In the `__init__` method, `self` refers to the newly created object; in other class methods, it refers to the instance whose method was called. Although you need to specify `self` explicitly when defining the method, you do *not* specify it when calling the method; Python will add it for you automatically.



- ④ `__init__` methods can take any number of arguments, and just like functions, the arguments can be defined with default values, making them optional to the caller. In this case, `filename` has a default value of `None`, which is the Python null value.

By convention, the first argument of any Python class method (the reference to the current instance) is called `self`. This argument fills the role of the reserved word `this` in C++ or Java, but `self` is not a reserved word in Python, merely a naming convention. Nonetheless, please don't call it anything but `self`; this is a very strong convention.

### Example 5.6. Coding the `FileInfo` Class

```
class FileInfo(UserDict):  
    "store file metadata"  
    def __init__(self, filename=None):  
        UserDict.__init__(self)  ❶  
        self["name"] = filename  ❷  
                                ❸
```

- ❶ Some pseudo-object-oriented languages like Powerbuilder have a concept of "extending" constructors and other events, where the ancestor's method is called automatically before the descendant's method is executed. Python does not do this; you must always explicitly call the appropriate method in the ancestor class.
- ❷ I told you that this class acts like a dictionary, and here is the first sign of it. You're assigning the argument `filename` as the value of this object's `name` key.
- ❸ Note that the `__init__` method never returns a value.

### 5.3.2. Knowing When to Use `self` and `__init__`

When defining your class methods, you *must* explicitly list `self` as the first argument for each method, including `__init__`. When you call a method of an ancestor class from within your class, you *must* include the `self` argument. But when you call your class method from outside, you do not specify anything for the `self` argument; you skip it entirely, and Python automatically adds the instance reference for you. I am aware that this is confusing at first; it's not really inconsistent, but it may appear inconsistent because it relies on a distinction (between bound and unbound methods) that you don't know about yet.

Whew. I realize that's a lot to absorb, but you'll get the hang of it. All Python classes work the same way, so once you learn one, you've learned them all. If you forget everything else, remember this one thing, because I promise it will trip you up:

`__init__` methods are optional, but when you define one, you must remember to explicitly call the ancestor's `__init__` method (if it defines one). This is more generally true: whenever a descendant wants to extend the behavior of the ancestor, the descendant method must explicitly call the ancestor method at the proper time, with the proper arguments.

### Further Reading on Python Classes

- *Learning to Program* (<http://www.freenetpages.co.uk/hp/alan.gauld/>) has a gentler introduction to classes (<http://www.freenetpages.co.uk/hp/alan.gauld/tutclass.htm>).
- *How to Think Like a Computer Scientist* (<http://www.ibiblio.org/obp/thinkCSpy/>) shows how to use classes to model compound datatypes (<http://www.ibiblio.org/obp/thinkCSpy/chap12.htm>).
- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) has an in-depth look at classes, namespaces, and inheritance (<http://www.python.org/doc/current/tut/node11.html>).
- Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) answers common questions about classes (<http://www.faqs.com/knowledge-base/index.phtml/fid/242>).

## 5.4. Instantiating Classes

Instantiating classes in Python is straightforward. To instantiate a class, simply call the class as if it were a function, passing the arguments that the `__init__` method defines. The return value will be the newly created object.

### Example 5.7. Creating a `FileInfo` Instance

```
>>> import fileinfo
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3") ❶
>>> f.__class__ ❷
<class fileinfo.FileInfo at 010EC204>
>>> f.__doc__ ❸
'store file metadata'
>>> f ❹
{'name': '/music/_singles/kairo.mp3'}
```

- ❶ You are creating an instance of the `FileInfo` class (defined in the `fileinfo` module) and assigning the newly created instance to the variable `f`. You are passing one parameter, `/music/_singles/kairo.mp3`, which will end up as the `filename` argument in `FileInfo`'s `__init__` method.
- ❷ Every class instance has a built-in attribute, `__class__`, which is the object's class. (Note that the representation of this includes the physical address of the instance on my machine; your representation will be different.) Java programmers may be familiar with the `Class` class, which contains methods like `getName` and `getSuperclass` to get metadata information about an object. In Python, this kind of metadata is available directly on the object itself through attributes like `__class__`, `__name__`, and `__bases__`.
- ❸ You can access the instance's `doc` string just as with a function or a module. All instances of a class share the same `doc` string.
- ❹ Remember when the `__init__` method assigned its `filename` argument to `self["name"]`? Well, here's the result. The arguments you pass when you create the class instance get sent right along to the `__init__` method (along with the object reference, `self`, which Python adds for free).

In Python, simply call a class as if it were a function to create a new instance of the class. There is no explicit new operator like C++ or Java.

### 5.4.1. Garbage Collection

If creating new instances is easy, destroying them is even easier. In general, there is no need to explicitly free instances, because they are freed automatically when the variables assigned to them go out of scope. Memory leaks are rare in Python.

### Example 5.8. Trying to Implement a Memory Leak

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

- ❶ Every time the `leakmem` function is called, you are creating an instance of `FileInfo` and assigning it to the variable `f`, which is a local variable within the function. Then the function ends without ever freeing `f`, so you would expect a memory leak, but you would be wrong. When the function ends, the local variable `f` goes out of scope. At this point, there are no longer any references to the newly created instance of `FileInfo` (since you never assigned it to anything other than `f`), so Python destroys the

instance for us.

- ❷ No matter how many times you call the `leakmem` function, it will never leak memory, because every time, Python will destroy the newly created `FileInfo` class before returning from `leakmem`.

The technical term for this form of garbage collection is "reference counting". Python keeps a list of references to every instance created. In the above example, there was only one reference to the `FileInfo` instance: the local variable `f`. When the function ends, the variable `f` goes out of scope, so the reference count drops to 0, and Python destroys the instance automatically.

In previous versions of Python, there were situations where reference counting failed, and Python couldn't clean up after you. If you created two instances that referenced each other (for instance, a doubly-linked list, where each node has a pointer to the previous and next node in the list), neither instance would ever be destroyed automatically because Python (correctly) believed that there is always a reference to each instance. Python 2.0 has an additional form of garbage collection called "mark-and-sweep" which is smart enough to notice this virtual gridlock and clean up circular references correctly.

As a former philosophy major, it disturbs me to think that things disappear when no one is looking at them, but that's exactly what happens in Python. In general, you can simply forget about memory management and let Python clean up after you.

### Further Reading on Garbage Collection

- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes built-in attributes like `__class__` (<http://www.python.org/doc/current/lib/specialattrs.html>).
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `gc` module (<http://www.python.org/doc/current/lib/module-gc.html>), which gives you low-level control over Python's garbage collection.

## 5.5. Exploring UserDict: A Wrapper Class

As you've seen, `FileInfo` is a class that acts like a dictionary. To explore this further, let's look at the `UserDict` class in the `UserDict` module, which is the ancestor of the `FileInfo` class. This is nothing special; the class is written in Python and stored in a `.py` file, just like any other Python code. In particular, it's stored in the `lib` directory in your Python installation.

In the ActivePython IDE on Windows, you can quickly open any module in your library path by selecting **File->Locate...** (**Ctrl-L**).

### Example 5.9. Defining the UserDict Class

```
class UserDict:
    def __init__(self, dict=None):
        self.data = {}
        if dict is not None: self.update(dict)
```

❶  
❷  
❸  
❹ ❺

- ❶ Note that `UserDict` is a base class, not inherited from any other class.
- ❷ This is the `__init__` method that you overrode in the `FileInfo` class. Note that the argument list in this ancestor class is different than the descendant. That's okay; each subclass can have its own set of arguments, as long as it calls the ancestor with the correct arguments. Here the ancestor class has a way to define initial values (by passing a dictionary in the `dict` argument) which the `FileInfo` does not use.

- ③ Python supports data attributes (called "instance variables" in Java and Powerbuilder, and "member variables" in C++). Data attributes are pieces of data held by a specific instance of a class. In this case, each instance of `UserDict` will have a data attribute `data`. To reference this attribute from code outside the class, you qualify it with the instance name, `instance.data`, in the same way that you qualify a function with its module name. To reference a data attribute from within the class, you use `self` as the qualifier. By convention, all data attributes are initialized to reasonable values in the `__init__` method. However, this is not required, since data attributes, like local variables, spring into existence when they are first assigned a value.
- ④ The `update` method is a dictionary duplicator: it copies all the keys and values from one dictionary to another. This does *not* clear the target dictionary first; if the target dictionary already has some keys, the ones from the source dictionary will be overwritten, but others will be left untouched. Think of `update` as a merge function, not a copy function.
- ⑤ This is a syntax you may not have seen before (I haven't used it in the examples in this book). It's an `if` statement, but instead of having an indented block starting on the next line, there is just a single statement on the same line, after the colon. This is perfectly legal syntax, which is just a shortcut you can use when you have only one statement in a block. (It's like specifying a single statement without braces in C++.) You can use this syntax, or you can have indented code on subsequent lines, but you can't do both for the same block.

Java and Powerbuilder support function overloading by argument list, *i.e.* one class can have multiple methods with the same name but a different number of arguments, or arguments of different types. Other languages (most notably PL/SQL) even support function overloading by argument name; *i.e.* one class can have multiple methods with the same name and the same number of arguments of the same type but different argument names. Python supports neither of these; it has no form of function overloading whatsoever. Methods are defined solely by their name, and there can be only one method per class with a given name. So if a descendant class has an `__init__` method, it *always* overrides the ancestor `__init__` method, even if the descendant defines it with a different argument list. And the same rule applies to any other method.

Guido, the original author of Python, explains method overriding this way: "Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class, may in fact end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)" If that doesn't make sense to you (it confuses the hell out of me), feel free to ignore it. I just thought I'd pass it along.

Always assign an initial value to all of an instance's data attributes in the `__init__` method. It will save you hours of debugging later, tracking down `AttributeError` exceptions because you're referencing uninitialized (and therefore non-existent) attributes.

### Example 5.10. `UserDict` Normal Methods

```
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` is a normal class method; it is publicly available to be called by anyone at any time. Notice that `clear`, like all class methods, has `self` as its first argument. (Remember that you don't include `self` when you call the method; it's something that Python adds for you.) Also note the basic technique of this wrapper class: store a real dictionary (`data`) as a data attribute, define all the methods that a real dictionary has, and have each class method redirect to the corresponding method on the real dictionary. (In case you'd forgotten, a dictionary's

`clear` method deletes all of its keys and their associated values.)

- ❷ The `copy` method of a real dictionary returns a new dictionary that is an exact duplicate of the original (all the same key–value pairs). But `UserDict` can't simply redirect to `self.data.copy`, because that method returns a real dictionary, and what you want is to return a new instance that is the same class as `self`.
- ❸ You use the `__class__` attribute to see if `self` is a `UserDict`; if so, you're golden, because you know how to copy a `UserDict`: just create a new `UserDict` and give it the real dictionary that you've squirreled away in `self.data`. Then you immediately return the new `UserDict` you don't even get to the `import copy` on the next line.
- ❹ If `self.__class__` is not `UserDict`, then `self` must be some subclass of `UserDict` (like maybe `FileInfo`), in which case life gets trickier. `UserDict` doesn't know how to make an exact copy of one of its descendants; there could, for instance, be other data attributes defined in the subclass, so you would need to iterate through them and make sure to copy all of them. Luckily, Python comes with a module to do exactly this, and it's called `copy`. I won't go into the details here (though it's a wicked cool module, if you're ever inclined to dive into it on your own). Suffice it to say that `copy` can copy arbitrary Python objects, and that's how you're using it here.
- ❺ The rest of the methods are straightforward, redirecting the calls to the built-in methods on `self.data`.

In versions of Python prior to 2.2, you could not directly subclass built-in datatypes like strings, lists, and dictionaries. To compensate for this, Python comes with wrapper classes that mimic the behavior of these built-in datatypes: `UserString`, `UserList`, and `UserDict`. Using a combination of normal and special methods, the `UserDict` class does an excellent imitation of a dictionary. In Python 2.2 and later, you can inherit classes directly from built-in datatypes like `dict`. An example of this is given in the examples that come with this book, in `fileinfo_fromdict.py`.

In Python, you can inherit directly from the `dict` built-in datatype, as shown in this example. There are three differences here compared to the `UserDict` version.

### Example 5.11. Inheriting Directly from Built-In Datatype `dict`

```
class FileInfo(dict):  
    "store file metadata"  
    def __init__(self, filename=None):  
        self["name"] = filename
```

- ❶ The first difference is that you don't need to import the `UserDict` module, since `dict` is a built-in datatype and is always available. The second is that you are inheriting from `dict` directly, instead of from `UserDict.UserDict`.
- ❷ The third difference is subtle but important. Because of the way `UserDict` works internally, it requires you to manually call its `__init__` method to properly initialize its internal data structures. `dict` does not work like this; it is not a wrapper, and it requires no explicit initialization.

### Further Reading on `UserDict`

- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `UserDict` module (<http://www.python.org/doc/current/lib/module-UserDict.html>) and the `copy` module (<http://www.python.org/doc/current/lib/module-copy.html>).

## 5.6. Special Class Methods

In addition to normal class methods, there are a number of special methods that Python classes can define. Instead of being called directly by your code (like normal methods), special methods are called for you by Python in particular circumstances or when specific syntax is used.

As you saw in the previous section, normal methods go a long way towards wrapping a dictionary in a class. But normal methods alone are not enough, because there are a lot of things you can do with dictionaries besides call methods on them. For starters, you can get and set items with a syntax that doesn't include explicitly invoking methods. This is where special class methods come in: they provide a way to map non-method-calling syntax into method calls.

### 5.6.1. Getting and Setting Items

#### Example 5.12. The `__getitem__` Special Method

```
def __getitem__(self, key): return self.data[key]

>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3")
>>> f
{'name': '/music/_singles/kairo.mp3'}
>>> f.__getitem__("name") ❶
'/music/_singles/kairo.mp3'
>>> f["name"] ❷
'/music/_singles/kairo.mp3'
```

- ❶ The `__getitem__` special method looks simple enough. Like the normal methods `clear`, `keys`, and `values`, it just redirects to the dictionary to return its value. But how does it get called? Well, you can call `__getitem__` directly, but in practice you wouldn't actually do that; I'm just doing it here to show you how it works. The right way to use `__getitem__` is to get Python to call it for you.
- ❷ This looks just like the syntax you would use to get a dictionary value, and in fact it returns the value you would expect. But here's the missing link: under the covers, Python has converted this syntax to the method call `f.__getitem__("name")`. That's why `__getitem__` is a special class method; not only can you call it yourself, you can get Python to call it for you by using the right syntax.

Of course, Python has a `__setitem__` special method to go along with `__getitem__`, as shown in the next example.

#### Example 5.13. The `__setitem__` Special Method

```
def __setitem__(self, key, item): self.data[key] = item

>>> f
{'name': '/music/_singles/kairo.mp3'}
>>> f.__setitem__("genre", 31) ❶
>>> f
{'name': '/music/_singles/kairo.mp3', 'genre': 31}
>>> f["genre"] = 32 ❷
>>> f
{'name': '/music/_singles/kairo.mp3', 'genre': 32}
```

- ❶ Like the `__getitem__` method, `__setitem__` simply redirects to the real dictionary `self.data` to do its work. And like `__getitem__`, you wouldn't ordinarily call it directly like this; Python calls `__setitem__` for you when you use the right syntax.
- ❷ This looks like regular dictionary syntax, except of course that `f` is really a class that's trying very hard to masquerade as a dictionary, and `__setitem__` is an essential part of that masquerade. This line of code actually calls `f.__setitem__("genre", 32)` under the covers.

`__setitem__` is a special class method because it gets called for you, but it's still a class method. Just as easily as the `__setitem__` method was defined in `UserDict`, you can redefine it in the descendant class to override the ancestor method. This allows you to define classes that act like dictionaries in some ways but define their own behavior above and beyond the built-in dictionary.

This concept is the basis of the entire framework you're studying in this chapter. Each file type can have a handler class that knows how to get metadata from a particular type of file. Once some attributes (like the file's name and location) are known, the handler class knows how to derive other attributes automatically. This is done by overriding the `__setitem__` method, checking for particular keys, and adding additional processing when they are found.

For example, `MP3FileInfo` is a descendant of `FileInfo`. When an `MP3FileInfo`'s name is set, it doesn't just set the name key (like the ancestor `FileInfo` does); it also looks in the file itself for MP3 tags and populates a whole set of keys. The next example shows how this works.

#### Example 5.14. Overriding `__setitem__` in `MP3FileInfo`

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

❶  
❷  
❸  
❹

- ❶ Notice that this `__setitem__` method is defined exactly the same way as the ancestor method. This is important, since Python will be calling the method for you, and it expects it to be defined with a certain number of arguments. (Technically speaking, the names of the arguments don't matter; only the number of arguments is important.)
- ❷ Here's the crux of the entire `MP3FileInfo` class: if you're assigning a value to the name key, you want to do something extra.
- ❸ The extra processing you do for names is encapsulated in the `__parse` method. This is another class method defined in `MP3FileInfo`, and when you call it, you qualify it with `self`. Just calling `__parse` would look for a normal function defined outside the class, which is not what you want. Calling `self.__parse` will look for a class method defined within the class. This isn't anything new; you reference data attributes the same way.
- ❹ After doing this extra processing, you want to call the ancestor method. Remember that this is never done for you in Python; you must do it manually. Note that you're calling the immediate ancestor, `FileInfo`, even though it doesn't have a `__setitem__` method. That's okay, because Python will walk up the ancestor tree until it finds a class with the method you're calling, so this line of code will eventually find and call the `__setitem__` defined in `UserDict`.

When accessing data attributes within a class, you need to qualify the attribute name: `self.attribute`. When calling other methods within a class, you need to qualify the method name: `self.method`.

#### Example 5.15. Setting an `MP3FileInfo`'s name

```
>>> import fileinfo  
>>> mp3file = fileinfo.MP3FileInfo()  
>>> mp3file  
{'name': None}  
>>> mp3file["name"] = "/music/_singles/kairo.mp3"  
>>> mp3file  
{'album': 'Rave Mix', 'artist': '***DJ MARY-JANE***', 'genre': 31,  
'title': 'KAIRO***THE BEST GOA', 'name': '/music/_singles/kairo.mp3',  
'year': '2000', 'comment': 'http://mp3.com/DJMARYJANE'}  
>>> mp3file["name"] = "/music/_singles/sidewinder.mp3"  
>>> mp3file  
{'album': '', 'artist': 'The Cynic Project', 'genre': 18, 'title': 'Sidewinder',
```

❶

❷

❸

```
'name': '/music/_singles/sidewinder.mp3', 'year': '2000',
'comment': 'http://mp3.com/cynicproject'}
```

- ❶ First, you create an instance of `MP3FileInfo`, without passing it a filename. (You can get away with this because the filename argument of the `__init__` method is optional.) Since `MP3FileInfo` has no `__init__` method of its own, Python walks up the ancestor tree and finds the `__init__` method of `FileInfo`. This `__init__` method manually calls the `__init__` method of `UserDict` and then sets the name key to filename, which is `None`, since you didn't pass a filename. Thus, `mp3file` initially looks like a dictionary with one key, `name`, whose value is `None`.
- ❷ Now the real fun begins. Setting the name key of `mp3file` triggers the `__setitem__` method on `MP3FileInfo` (not `UserDict`), which notices that you're setting the name key with a real value and calls `self.__parse`. Although you haven't traced through the `__parse` method yet, you can see from the output that it sets several other keys: `album`, `artist`, `genre`, `title`, `year`, and `comment`.
- ❸ Modifying the name key will go through the same process again: Python calls `__setitem__`, which calls `self.__parse`, which sets all the other keys.

## 5.7. Advanced Special Class Methods

Python has more special methods than just `__getitem__` and `__setitem__`. Some of them let you emulate functionality that you may not even know about.

This example shows some of the other special methods in `UserDict`.

### Example 5.16. More Special Methods in `UserDict`

```
def __repr__(self): return repr(self.data)           ❶
def __cmp__(self, dict):                             ❷
    if isinstance(dict, UserDict):
        return cmp(self.data, dict.data)
    else:
        return cmp(self.data, dict)
def __len__(self): return len(self.data)             ❸
def __delitem__(self, key): del self.data[key]       ❹
```

- ❶ `__repr__` is a special method that is called when you call `repr(instance)`. The `repr` function is a built-in function that returns a string representation of an object. It works on any object, not just class instances. You're already intimately familiar with `repr` and you don't even know it. In the interactive window, when you type just a variable name and press the **ENTER** key, Python uses `repr` to display the variable's value. Go create a dictionary `d` with some data and then print `repr(d)` to see for yourself.
- ❷ `__cmp__` is called when you compare class instances. In general, you can compare any two Python objects, not just class instances, by using `==`. There are rules that define when built-in datatypes are considered equal; for instance, dictionaries are equal when they have all the same keys and values, and strings are equal when they are the same length and contain the same sequence of characters. For class instances, you can define the `__cmp__` method and code the comparison logic yourself, and then you can use `==` to compare instances of your class and Python will call your `__cmp__` special method for you.
- ❸ `__len__` is called when you call `len(instance)`. The `len` function is a built-in function that returns the length of an object. It works on any object that could reasonably be thought of as having a length. The `len` of a string is its number of characters; the `len` of a dictionary is its number of keys; the `len` of a list or tuple is its number of elements. For class instances, define the `__len__` method



and code the length calculation yourself, and then call `len(instance)` and Python will call your `__len__` special method for you.

- ④ `__delitem__` is called when you call `del instance[key]`, which you may remember as the way to delete individual items from a dictionary. When you use `del` on a class instance, Python calls the `__delitem__` special method for you.

In Java, you determine whether two string variables reference the same physical memory location by using `str1 == str2`. This is called *object identity*, and it is written in Python as `str1 is str2`. To compare string values in Java, you would use `str1.equals(str2)`; in Python, you would use `str1 == str2`. Java programmers who have been taught to believe that the world is a better place because `==` in Java compares by identity instead of by value may have a difficult time adjusting to Python's lack of such "gotchas".

At this point, you may be thinking, "All this work just to do something in a class that I can do with a built-in datatype." And it's true that life would be easier (and the entire `UserDict` class would be unnecessary) if you could inherit from built-in datatypes like a dictionary. But even if you could, special methods would still be useful, because they can be used in any class, not just wrapper classes like `UserDict`.

Special methods mean that *any class* can store key/value pairs like a dictionary, just by defining the `__setitem__` method. *Any class* can act like a sequence, just by defining the `__getitem__` method. Any class that defines the `__cmp__` method can be compared with `==`. And if your class represents something that has a length, don't define a `GetLength` method; define the `__len__` method and use `len(instance)`.

While other object-oriented languages only let you define the physical model of an object ("this object has a `GetLength` method"), Python's special class methods like `__len__` allow you to define the logical model of an object ("this object has a length").

Python has a lot of other special methods. There's a whole set of them that let classes act like numbers, allowing you to add, subtract, and do other arithmetic operations on class instances. (The canonical example of this is a class that represents complex numbers, numbers with both real and imaginary components.) The `__call__` method lets a class act like a function, allowing you to call a class instance directly. And there are other special methods that allow classes to have read-only and write-only data attributes; you'll talk more about those in later chapters.

### Further Reading on Special Class Methods

- *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) documents all the special class methods (<http://www.python.org/doc/current/ref/specialnames.html>).

## 5.8. Introducing Class Attributes

You already know about data attributes, which are variables owned by a specific instance of a class. Python also supports class attributes, which are variables owned by the class itself.

### Example 5.17. Introducing Class Attributes

```
class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title" : ( 3, 33, stripnulls),
                  "artist" : ( 33, 63, stripnulls),
                  "album" : ( 63, 93, stripnulls),
                  "year" : ( 93, 97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre" : (127, 128, ord)}
```

```

>>> import fileinfo
>>> fileinfo.MP3FileInfo ❶
<class fileinfo.MP3FileInfo at 01257FDC>
>>> fileinfo.MP3FileInfo.tagDataMap ❷
{'title': (3, 33, <function stripnulls at 0260C8D4>),
 'genre': (127, 128, <built-in function ord>),
 'artist': (33, 63, <function stripnulls at 0260C8D4>),
 'year': (93, 97, <function stripnulls at 0260C8D4>),
 'comment': (97, 126, <function stripnulls at 0260C8D4>),
 'album': (63, 93, <function stripnulls at 0260C8D4>)}
>>> m = fileinfo.MP3FileInfo() ❸
>>> m.tagDataMap
{'title': (3, 33, <function stripnulls at 0260C8D4>),
 'genre': (127, 128, <built-in function ord>),
 'artist': (33, 63, <function stripnulls at 0260C8D4>),
 'year': (93, 97, <function stripnulls at 0260C8D4>),
 'comment': (97, 126, <function stripnulls at 0260C8D4>),
 'album': (63, 93, <function stripnulls at 0260C8D4>)}

```

- ❶ MP3FileInfo is the class itself, not any particular instance of the class.
- ❷ tagDataMap is a class attribute: literally, an attribute of the class. It is available before creating any instances of the class.
- ❸ Class attributes are available both through direct reference to the class and through any instance of the class.

In Java, both static variables (called class attributes in Python) and instance variables (called data attributes in Python) are defined immediately after the class definition (one with the `static` keyword, one without). In Python, only class attributes can be defined here; data attributes are defined in the `__init__` method.

Class attributes can be used as class-level constants (which is how you use them in `MP3FileInfo`), but they are not really constants. You can also change them.

There are no constants in Python. Everything can be changed if you try hard enough. This fits with one of the core principles of Python: bad behavior should be discouraged but not banned. If you really want to change the value of `None`, you can do it, but don't come running to me when your code is impossible to debug.

### Example 5.18. Modifying Class Attributes

```

>>> class counter:
...     count = 0 ❶
...     def __init__(self):
...         self.__class__.count += 1 ❷
...
>>> counter
<class __main__.counter at 010EAECC>
>>> counter.count ❸
0
>>> c = counter()
>>> c.count ❹
1
>>> counter.count
1
>>> d = counter() ❺
>>> d.count
2
>>> c.count
2
>>> counter.count
2

```

- ❶ `count` is a class attribute of the `counter` class.
- ❷ `__class__` is a built-in attribute of every class instance (of every class). It is a reference to the class that `self` is an instance of (in this case, the `counter` class).
- ❸ Because `count` is a class attribute, it is available through direct reference to the class, before you have created any instances of the class.
- ❹ Creating an instance of the class calls the `__init__` method, which increments the class attribute `count` by 1. This affects the class itself, not just the newly created instance.
- ❺ Creating a second instance will increment the class attribute `count` again. Notice how the class attribute is shared by the class and all instances of the class.

## 5.9. Private Functions

Like most languages, Python has the concept of private elements:

- Private functions, which can't be called from outside their module
- Private class methods, which can't be called from outside their class
- Private attributes, which can't be accessed from outside their class.

Unlike in most languages, whether a Python function, method, or attribute is private or public is determined entirely by its name.

If the name of a Python function, class method, or attribute starts with (but doesn't end with) two underscores, it's private; everything else is public. Python has no concept of *protected* class methods (accessible only in their own class and descendant classes). Class methods are either private (accessible only in their own class) or public (accessible from anywhere).

In `MP3FileInfo`, there are two methods: `__parse` and `__setitem__`. As you have already discussed, `__setitem__` is a special method; normally, you would call it indirectly by using the dictionary syntax on a class instance, but it is public, and you could call it directly (even from outside the `fileinfo` module) if you had a really good reason. However, `__parse` is private, because it has two underscores at the beginning of its name.

In Python, all special methods (like `__setitem__`) and built-in attributes (like `__doc__`) follow a standard naming convention: they both start with and end with two underscores. Don't name your own methods and attributes this way, because it will only confuse you (and others) later.

### Example 5.19. Trying to Call a Private Method

```
>>> import fileinfo
>>> m = fileinfo.MP3FileInfo()
>>> m.__parse("/music/_singles/kairo.mp3") ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'MP3FileInfo' instance has no attribute '__parse'
```

- ❶ If you try to call a private method, Python will raise a slightly misleading exception, saying that the method does not exist. Of course it does exist, but it's private, so it's not accessible outside the class. Strictly speaking, private methods are accessible outside their class, just not *easily* accessible. Nothing in Python is truly private; internally, the names of private methods and attributes are mangled and unmangled on the fly to make them seem inaccessible by their given names. You can access the `__parse` method of the `MP3FileInfo` class by

## Further Reading on Private Functions

- ## 5.10. Summary

Before diving into the next chapter, make sure you're comfortable doing all of these things:

- 63

# Chapter 6. Exceptions and File Handling

In this chapter, you will dive into exceptions, file objects, `for` loops, and the `os` and `sys` modules. If you've used exceptions in another programming language, you can skim the first section to get a sense of Python's syntax. Be sure to tune in again for file handling.

## 6.1. Handling Exceptions

Like many other programming languages, Python has exception handling via `try...except` blocks.

Python uses `try...except` to handle exceptions and `raise` to generate them. Java and C++ use `try...catch` to handle exceptions, and `throw` to generate them.

Exceptions are everywhere in Python. Virtually every module in the standard Python library uses them, and Python itself will raise them in a lot of different circumstances. You've already seen them repeatedly throughout this book.

- Accessing a non-existent dictionary key will raise a `KeyError` exception.
- Searching a list for a non-existent value will raise a `ValueError` exception.
- Calling a non-existent method will raise an `AttributeError` exception.
- Referencing a non-existent variable will raise a `NameError` exception.
- Mixing datatypes without coercion will raise a `TypeError` exception.

In each of these cases, you were simply playing around in the Python IDE: an error occurred, the exception was printed (depending on your IDE, perhaps in an intentionally jarring shade of red), and that was that. This is called an *unhandled* exception. When the exception was raised, there was no code to explicitly notice it and deal with it, so it bubbled its way back to the default behavior built in to Python, which is to spit out some debugging information and give up. In the IDE, that's no big deal, but if that happened while your actual Python program was running, the entire program would come to a screeching halt.

An exception doesn't need result in a complete program crash, though. Exceptions, when raised, can be *handled*. Sometimes an exception is really because you have a bug in your code (like accessing a variable that doesn't exist), but many times, an exception is something you can anticipate. If you're opening a file, it might not exist. If you're connecting to a database, it might be unavailable, or you might not have the correct security credentials to access it. If you know a line of code may raise an exception, you should handle the exception using a `try...except` block.

### Example 6.1. Opening a Non-Existent File

```
>>> fsock = open("/notthere", "r") ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: '/notthere'
>>> try:
...     fsock = open("/notthere") ❷
... except IOError: ❸
...     print "The file does not exist, exiting gracefully"
... print "This line will always print" ❹
The file does not exist, exiting gracefully
This line will always print
```

- ❶ Using the built-in `open` function, you can try to open a file for reading (more on `open` in the next section). But the file doesn't exist, so this raises the `IOError` exception. Since you haven't provided any explicit check

for an `IOError` exception, Python just prints out some debugging information about what happened and then gives up.

- ❷ You're trying to open the same non-existent file, but this time you're doing it within a `try...except` block.
- ❸ When the `open` method raises an `IOError` exception, you're ready for it. The `except IOError:` line catches the exception and executes your own block of code, which in this case just prints a more pleasant error message.
- ❹ Once an exception has been handled, processing continues normally on the first line after the `try...except` block. Note that this line will always print, whether or not an exception occurs. If you really did have a file called `notthere` in your root directory, the call to `open` would succeed, the `except` clause would be ignored, and this line would still be executed.

Exceptions may seem unfriendly (after all, if you don't catch the exception, your entire program will crash), but consider the alternative. Would you rather get back an unusable file object to a non-existent file? You'd need to check its validity somehow anyway, and if you forgot, somewhere down the line, your program would give you strange errors somewhere down the line that you would need to trace back to the source. I'm sure you've experienced this, and you know it's not fun. With exceptions, errors occur immediately, and you can handle them in a standard way at the source of the problem.

### 6.1.1. Using Exceptions For Other Purposes

There are a lot of other uses for exceptions besides handling actual error conditions. A common use in the standard Python library is to try to import a module, and then check whether it worked. Importing a module that does not exist will raise an `ImportError` exception. You can use this to define multiple levels of functionality based on which modules are available at run-time, or to support multiple platforms (where platform-specific code is separated into different modules).

You can also define your own exceptions by creating a class that inherits from the built-in `Exception` class, and then raise your exceptions with the `raise` command. See the further reading section if you're interested in doing this.

The next example demonstrates how to use an exception to support platform-specific functionality. This code comes from the `getpass` module, a wrapper module for getting a password from the user. Getting a password is accomplished differently on UNIX, Windows, and Mac OS platforms, but this code encapsulates all of those differences.

#### Example 6.2. Supporting Platform-Specific Functionality

```
# Bind the name getpass to the appropriate function
try:
    import termios, TERMIOS
except ImportError:
    try:
        import msvcrt
    except ImportError:
        try:
            from EasyDialogs import AskPassword
        except ImportError:
            getpass = default_getpass
        else:
            getpass = AskPassword
    else:
        getpass = win_getpass
else:
    getpass = unix_getpass
```

❶

❷

❸

❹

❺

- ❶ `termios` is a UNIX-specific module that provides low-level control over the input terminal. If this module is not available (because it's not on your system, or your system doesn't support it), the import fails and Python raises an `ImportError`, which you catch.
- ❷ OK, you didn't have `termios`, so let's try `msvcrt`, which is a Windows-specific module that provides an API to many useful functions in the Microsoft Visual C++ runtime services. If this import fails, Python will raise an `ImportError`, which you catch.
- ❸ If the first two didn't work, you try to import a function from `EasyDialogs`, which is a Mac OS-specific module that provides functions to pop up dialog boxes of various types. Once again, if this import fails, Python will raise an `ImportError`, which you catch.
- ❹ None of these platform-specific modules is available (which is possible, since Python has been ported to a lot of different platforms), so you need to fall back on a default password input function (which is defined elsewhere in the `getpass` module). Notice what you're doing here: assigning the function `default_getpass` to the variable `getpass`. If you read the official `getpass` documentation, it tells you that the `getpass` module defines a `getpass` function. It does this by binding `getpass` to the correct function for your platform. Then when you call the `getpass` function, you're really calling a platform-specific function that this code has set up for you. You don't need to know or care which platform your code is running on — just call `getpass`, and it will always do the right thing.
- ❺ A `try...except` block can have an `else` clause, like an `if` statement. If no exception is raised during the `try` block, the `else` clause is executed afterwards. In this case, that means that the `from EasyDialogs import AskPassword` import worked, so you should bind `getpass` to the `AskPassword` function. Each of the other `try...except` blocks has similar `else` clauses to bind `getpass` to the appropriate function when you find an import that works.

### Further Reading on Exception Handling

- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses defining and raising your own exceptions, and handling multiple exceptions at once (<http://www.python.org/doc/current/tut/node10.html#SECTION0010400000000000000000>).
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the built-in exceptions (<http://www.python.org/doc/current/lib/module-exceptions.html>).
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `getpass` (<http://www.python.org/doc/current/lib/module-getpass.html>) module.
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `traceback` module (<http://www.python.org/doc/current/lib/module-traceback.html>), which provides low-level access to exception attributes after an exception is raised.
- *Python Reference Manual* (<http://www.python.org/doc/current/ref/>) discusses the inner workings of the `try...except` block (<http://www.python.org/doc/current/ref/try.html>).

## 6.2. Working with File Objects

Python has a built-in function, `open`, for opening a file on disk. `open` returns a file object, which has methods and attributes for getting information about and manipulating the opened file.

### Example 6.3. Opening a File

```
>>> f = open("/music/_singles/kairo.mp3", "rb") ❶
>>> f ❷
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.mode ❸
'rb'
>>> f.name ❹
'/music/_singles/kairo.mp3'
```

- ❶ The `open` method can take up to three parameters: a filename, a mode, and a buffering parameter. Only the first one, the filename, is required; the other two are optional. If not specified, the file is opened for reading in text mode. Here you are opening the file for reading in binary mode. (`print open.__doc__` displays a great explanation of all the possible modes.)
- ❷ The `open` function returns an object (by now, this should not surprise you). A file object has several useful attributes.
- ❸ The `mode` attribute of a file object tells you in which mode the file was opened.
- ❹ The `name` attribute of a file object tells you the name of the file that the file object has open.

### 6.2.1. Reading Files

After you open a file, the first thing you'll want to do is read from it, as shown in the next example.

#### Example 6.4. Reading a File

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

- ❶ A file object maintains state about the file it has open. The `tell` method of a file object tells you your current position in the open file. Since you haven't done anything with this file yet, the current position is 0, which is the beginning of the file.
- ❷ The `seek` method of a file object moves to another position in the open file. The second parameter specifies what the first one means; 0 means move to an absolute position (counting from the start of the file), 1 means move to a relative position (counting from the current position), and 2 means move to a position relative to the end of the file. Since the MP3 tags you're looking for are stored at the end of the file, you use 2 and tell the file object to move to a position 128 bytes from the end of the file.
- ❸ The `tell` method confirms that the current file position has moved.
- ❹ The `read` method reads a specified number of bytes from the open file and returns a string with the data that was read. The optional parameter specifies the maximum number of bytes to read. If no parameter is specified, `read` will read until the end of the file. (You could have simply said `read()` here, since you know exactly where you are in the file and you are, in fact, reading the last 128 bytes.) The read data is assigned to the `tagData` variable, and the current position is updated based on how many bytes were read.
- ❺ The `tell` method confirms that the current position has moved. If you do the math, you'll see that after reading 128 bytes, the position has been incremented by 128.

### 6.2.2. Closing Files

Open files consume system resources, and depending on the file mode, other programs may not be able to access them. It's important to close files as soon as you're finished with them.



## Example 6.5. Closing a File

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed ❶
False
>>> f.close() ❷
>>> f
<closed file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed ❸
True
>>> f.seek(0) ❹
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.tell()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.read()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.close() ❺
```

- ❶ The `closed` attribute of a file object indicates whether the object has a file open or not. In this case, the file is still open (`closed` is `False`).
- ❷ To close a file, call the `close` method of the file object. This frees the lock (if any) that you were holding on the file, flushes buffered writes (if any) that the system hadn't gotten around to actually writing yet, and releases the system resources.
- ❸ The `closed` attribute confirms that the file is closed.
- ❹ Just because a file is closed doesn't mean that the file object ceases to exist. The variable `f` will continue to exist until it goes out of scope or gets manually deleted. However, none of the methods that manipulate an open file will work once the file has been closed; they all raise an exception.
- ❺ Calling `close` on a file object whose file is already closed does *not* raise an exception; it fails silently.

## 6.2.3. Handling I/O Errors

Now you've seen enough to understand the file handling code in the `fileinfo.py` sample code from the previous chapter. This example shows how to safely open and read from a file and gracefully handle errors.

### Example 6.6. File Objects in `MP3FileInfo`

```
try: ❶
    fsock = open(filename, "rb", 0) ❷
    try:
        fsock.seek(-128, 2) ❸
        tagdata = fsock.read(128) ❹
    finally: ❺
        fsock.close()
    .
    .
    .
except IOError: ❻
    pass
```

- ❶ Because opening and reading files is risky and may raise an exception, all of this code is wrapped in a `try...except` block. (Hey, isn't standardized indentation great? This is where you start to appreciate it.)
- ❷ The `open` function may raise an `IOError`. (Maybe the file doesn't exist.)
- ❸ The `seek` method may raise an `IOError`. (Maybe the file is smaller than 128 bytes.)
- ❹ The `read` method may raise an `IOError`. (Maybe the disk has a bad sector, or it's on a network drive and the network just went down.)
- ❺ This is new: a `try...finally` block. Once the file has been opened successfully by the `open` function, you want to make absolutely sure that you close it, even if an exception is raised by the `seek` or `read` methods. That's what a `try...finally` block is for: code in the `finally` block will *always* be executed, even if something in the `try` block raises an exception. Think of it as code that gets executed on the way out, regardless of what happened before.
- ❻ At last, you handle your `IOError` exception. This could be the `IOError` exception raised by the call to `open`, `seek`, or `read`. Here, you really don't care, because all you're going to do is ignore it silently and continue. (Remember, `pass` is a Python statement that does nothing.) That's perfectly legal; "handling" an exception can mean explicitly doing nothing. It still counts as handled, and processing will continue normally on the next line of code after the `try...except` block.

## 6.2.4. Writing to Files

As you would expect, you can also write to files in much the same way that you read from them. There are two basic file modes:

- "Append" mode will add data to the end of the file.
- "write" mode will overwrite the file.

Either mode will create the file automatically if it doesn't already exist, so there's never a need for any sort of fiddly "if the log file doesn't exist yet, create a new empty file just so you can open it for the first time" logic. Just open it and start writing.

### Example 6.7. Writing to Files

```
>>> logfile = open('test.log', 'w') ❶
>>> logfile.write('test succeeded') ❷
>>> logfile.close()
>>> print file('test.log').read() ❸
test succeeded
>>> logfile = open('test.log', 'a') ❹
>>> logfile.write('line 2')
>>> logfile.close()
>>> print file('test.log').read() ❺
test succeededline 2
```

- ❶ You start boldly by creating either the new file `test.log` or overwrites the existing file, and opening the file for writing. (The second parameter "w" means open the file for writing.) Yes, that's all as dangerous as it sounds. I hope you didn't care about the previous contents of that file, because it's gone now.
- ❷ You can add data to the newly opened file with the `write` method of the file object returned by `open`.
- ❸ `file` is a synonym for `open`. This one-liner opens the file, reads its contents, and prints them.
- ❹ You happen to know that `test.log` exists (since you just finished writing to it), so you can open it and append to it. (The "a" parameter means open the file for appending.) Actually you could do this even if the file didn't exist, because opening the file for appending will create the file if necessary. But appending

will *never* harm the existing contents of the file.

- ⑤ As you can see, both the original line you wrote and the second line you appended are now in `test.log`. Also note that carriage returns are not included. Since you didn't write them explicitly to the file either time, the file doesn't include them. You can write a carriage return with the `"\n"` character. Since you didn't do this, everything you wrote to the file ended up smooshed together on the same line.

### Further Reading on File Handling

- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses reading and writing files, including how to read a file one line at a time into a list (<http://www.python.org/doc/current/tut/node9.html#SECTION0092100000000000000000>).
- *eff-bot* (<http://www.effbot.org/guides/>) discusses efficiency and performance of various ways of reading a file (<http://www.effbot.org/guides/readline-performance.htm>).
- Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) answers common questions about files (<http://www.faqs.com/knowledge-base/index.phtml/fid/552>).
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes all the file object methods (<http://www.python.org/doc/current/lib/bltin-file-objects.html>).

## 6.3. Iterating with `for` Loops

Like most other languages, Python has `for` loops. The only reason you haven't seen them until now is that Python is good at so many other things that you don't need them as often.

Most other languages don't have a powerful list datatype like Python, so you end up doing a lot of manual work, specifying a start, end, and step to define a range of integers or characters or other iterable entities. But in Python, a `for` loop simply iterates over a list, the same way list comprehensions work.

### Example 6.8. Introducing the `for` Loop

```
>>> li = ['a', 'b', 'e']
>>> for s in li:           ❶
...     print s           ❷
a
b
e
>>> print "\n".join(li)  ❸
a
b
e
```

- ❶ The syntax for a `for` loop is similar to list comprehensions. `li` is a list, and `s` will take the value of each element in turn, starting from the first element.
- ❷ Like an `if` statement or any other indented block, a `for` loop can have any number of lines of code in it.
- ❸ This is the reason you haven't seen the `for` loop yet: you haven't needed it yet. It's amazing how often you use `for` loops in other languages when all you really want is a `join` or a list comprehension.

Doing a "normal" (by Visual Basic standards) counter `for` loop is also simple.

### Example 6.9. Simple Counters

```
>>> for i in range(5):    ❶
...     print i
```

```

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

```

- ❶ As you saw in Example 3.20, `Assigning Consecutive Values`, `range` produces a list of integers, which you then loop through. I know it looks a bit odd, but it is occasionally (and I stress *occasionally*) useful to have a counter loop.
- ❷ Don't ever do this. This is Visual Basic–style thinking. Break out of it. Just iterate through the list, as shown in the previous example.

`for` loops are not just for simple counters. They can iterate through all kinds of things. Here is an example of using a `for` loop to iterate through a dictionary.

### Example 6.10. Iterating Through a Dictionary

```

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

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

[...snip...]

```

- ❶ `os.environ` is a dictionary of the environment variables defined on your system. In Windows, these are your user and system variables accessible from MS–DOS. In UNIX, they are the variables exported in your shell's startup scripts. In Mac OS, there is no concept of environment variables, so this dictionary is empty.
- ❷ `os.environ.items()` returns a list of tuples: `[(key1, value1), (key2, value2), ...]`. The `for` loop iterates through this list. The first round, it assigns `key1` to `k` and `value1` to `v`, so `k = USERPROFILE` and `v = C:\Documents and Settings\mpilgrim`. In the second round, `k` gets the second key, `OS`, and `v` gets the corresponding value, `Windows_NT`.
- ❸ With multi–variable assignment and list comprehensions, you can replace the entire `for` loop with a single statement. Whether you actually do this in real code is a matter of personal coding style. I like it because it makes it clear that what I'm doing is mapping a dictionary into a list, then joining the list into a single string. Other programmers prefer to write this out as a `for` loop. The output is the same in either case, although this version is slightly faster, because there is only one `print` statement instead of many.

Now we can look at the `for` loop in `MP3FileInfo`, from the sample `fileinfo.py` program introduced in

**Example 6.11. for Loop in MP3FileInfo**

```

tagDataMap = {"title"   : ( 3, 33, stripnulls),
              "artist"  : (33, 63, stripnulls),
              "album"   : (63, 93, stripnulls),
              "year"    : (93, 97, stripnulls),
              "comment" : (97, 126, stripnulls),
              "genre"   : (127, 128, ord)}
.
.
.
    if tagdata[:3] == "TAG":
        for tag, (start, end, parseFunc) in self.tagDataMap.items():
            self[tag] = parseFunc(tagdata[start:end])

```

- ❶ tagDataMap is a class attribute that defines the tags you're looking for in an MP3 file. Tags are stored in fixed-length fields. Once you read the last 128 bytes of the file, bytes 3 through 32 of those are always the song title, 33 through 62 are always the artist name, 63 through 92 are the album name, and so forth. Note that tagDataMap is a dictionary of tuples, and each tuple contains two integers and a function reference.
- ❷ This looks complicated, but it's not. The structure of the for variables matches the structure of the elements of the list returned by items. Remember that items returns a list of tuples of the form (key, value). The first element of that list is ("title", (3, 33, <function stripnulls>)), so the first time around the loop, tag gets "title", start gets 3, end gets 33, and parseFunc gets the function stripnulls.
- ❸ Now that you've extracted all the parameters for a single MP3 tag, saving the tag data is easy. You slice tagdata from start to end to get the actual data for this tag, call parseFunc to post-process the data, and assign this as the value for the key tag in the pseudo-dictionary self. After iterating through all the elements in tagDataMap, self has the values for all the tags, and you know what that looks like.

**6.4. Using sys.modules**

Modules, like everything else in Python, are objects. Once imported, you can always get a reference to a module through the global dictionary `sys.modules`.

**Example 6.12. Introducing sys.modules**

```

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

```

- ❶ The `sys` module contains system-level information, such as the version of Python you're

running (`sys.version` or `sys.version_info`), and system-level options such as the maximum allowed recursion depth (`sys.getrecursionlimit()` and `sys.setrecursionlimit()`).

- ❷ `sys.modules` is a dictionary containing all the modules that have ever been imported since Python was started; the key is the module name, the value is the module object. Note that this is more than just the modules *your* program has imported. Python preloads some modules on startup, and if you're using a Python IDE, `sys.modules` contains all the modules imported by all the programs you've run within the IDE.

This example demonstrates how to use `sys.modules`.

### Example 6.13. Using `sys.modules`

```
>>> import fileinfo ❶
>>> 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"] ❷
<module 'fileinfo' from 'fileinfo.pyc'>
```

- ❶ As new modules are imported, they are added to `sys.modules`. This explains why importing the same module twice is very fast: Python has already loaded and cached the module in `sys.modules`, so importing the second time is simply a dictionary lookup.
- ❷ Given the name (as a string) of any previously-imported module, you can get a reference to the module itself through the `sys.modules` dictionary.

The next example shows how to use the `__module__` class attribute with the `sys.modules` dictionary to get a reference to the module in which a class is defined.

### Example 6.14. The `__module__` Class Attribute

```
>>> from fileinfo import MP3FileInfo
>>> MP3FileInfo.__module__ ❶
'fileinfo'
>>> sys.modules[MP3FileInfo.__module__] ❷
<module 'fileinfo' from 'fileinfo.pyc'>
```

- ❶ Every Python class has a built-in class attribute `__module__`, which is the name of the module in which the class is defined.
- ❷ Combining this with the `sys.modules` dictionary, you can get a reference to the module in which a class is defined.

Now you're ready to see how `sys.modules` is used in `fileinfo.py`, the sample program introduced in Chapter 5. This example shows that portion of the code.

### Example 6.15. `sys.modules` in `fileinfo.py`

```
def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]): ❶
    "get file info class from filename extension"
    subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:] ❷
    return hasattr(module, subclass) and getattr(module, subclass) or FileInfo ❸
```

- ❶ This is a function with two arguments; `filename` is required, but `module` is optional and defaults to the module that contains the `FileInfo` class. This looks inefficient, because you might expect Python to evaluate the `sys.modules` expression every time the function is called. In fact, Python evaluates default expressions only once, the first time the module is imported. As you'll see later, you never call this function with a `module` argument, so `module` serves as a function-level constant.
- ❷ You'll plow through this line later, after you dive into the `os` module. For now, take it on faith that `subclass` ends up as the name of a class, like `MP3FileInfo`.
- ❸ You already know about `getattr`, which gets a reference to an object by name. `hasattr` is a complementary function that checks whether an object has a particular attribute; in this case, whether a module has a particular class (although it works for any object and any attribute, just like `getattr`). In English, this line of code says, "If this module has the class named by `subclass` then return it, otherwise return the base class `FileInfo`."

### Further Reading on Modules

- *Python Tutorial* (<http://www.python.org/doc/current/tut/tut.html>) discusses exactly when and how default arguments are evaluated (<http://www.python.org/doc/current/tut/node6.html#SECTION0067100000000000000000>).
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `sys` (<http://www.python.org/doc/current/lib/module-sys.html>) module.

## 6.5. Working with Directories

The `os.path` module has several functions for manipulating files and directories. Here, we're looking at handling pathnames and listing the contents of a directory.

### Example 6.16. Constructing Pathnames

```
>>> import os
>>> os.path.join("c:\\music\\ap\\", "mahadeva.mp3") ❶ ❷
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.join("c:\\music\\ap", "mahadeva.mp3") ❸
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.expanduser("~") ❹
'c:\\Documents and Settings\\mpilgrim\\My Documents'
>>> os.path.join(os.path.expanduser("~"), "Python") ❺
'c:\\Documents and Settings\\mpilgrim\\My Documents\\Python'
```

- ❶ `os.path` is a reference to a module -- which module depends on your platform. Just as `getpass` encapsulates differences between platforms by setting `getpass` to a platform-specific function, `os` encapsulates differences between platforms by setting `path` to a platform-specific module.
- ❷

The `join` function of `os.path` constructs a pathname out of one or more partial pathnames. In this case, it simply concatenates strings. (Note that dealing with pathnames on Windows is annoying because the backslash character must be escaped.)

- ❸ In this slightly less trivial case, `join` will add an extra backslash to the pathname before joining it to the filename. I was overjoyed when I discovered this, since `addSlashIfNecessary` is one of the stupid little functions I always need to write when building up my toolbox in a new language. *Do not* write this stupid little function in Python; smart people have already taken care of it for you.
- ❹ `expanduser` will expand a pathname that uses `~` to represent the current user's home directory. This works on any platform where users have a home directory, like Windows, UNIX, and Mac OS X; it has no effect on Mac OS.
- ❺ Combining these techniques, you can easily construct pathnames for directories and files under the user's home directory.

### Example 6.17. Splitting Pathnames

```
>>> os.path.split("c:\\music\\ap\\mahadeva.mp3")           ❶
('c:\\music\\ap', 'mahadeva.mp3')
>>> (filepath, filename) = os.path.split("c:\\music\\ap\\mahadeva.mp3")  ❷
>>> filepath                                              ❸
'c:\\music\\ap'
>>> filename                                              ❹
'mahadeva.mp3'
>>> (shortname, extension) = os.path.splitext(filename)    ❺
>>> shortname
'mahadeva'
>>> extension
'.mp3'
```

- ❶ The `split` function splits a full pathname and returns a tuple containing the path and filename. Remember when I said you could use multi-variable assignment to return multiple values from a function? Well, `split` is such a function.
- ❷ You assign the return value of the `split` function into a tuple of two variables. Each variable receives the value of the corresponding element of the returned tuple.
- ❸ The first variable, `filepath`, receives the value of the first element of the tuple returned from `split`, the file path.
- ❹ The second variable, `filename`, receives the value of the second element of the tuple returned from `split`, the filename.
- ❺ `os.path` also contains a function `splitext`, which splits a filename and returns a tuple containing the filename and the file extension. You use the same technique to assign each of them to separate variables.

### Example 6.18. Listing Directories

```
>>> os.listdir("c:\\music\\_singles\\")                    ❶
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
>>> dirname = "c:\\\"
>>> os.listdir(dirname)                                    ❷
['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))]      ❸
```



```
['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))] ❹
['cygwin', 'docbook', 'Documents and Settings', 'Incoming',
'Inetpub', 'Music', 'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
```

- ❶ The `listdir` function takes a pathname and returns a list of the contents of the directory.
- ❷ `listdir` returns both files and folders, with no indication of which is which.
- ❸ You can use list filtering and the `isfile` function of the `os.path` module to separate the files from the folders. `isfile` takes a pathname and returns 1 if the path represents a file, and 0 otherwise. Here you're using `os.path.join` to ensure a full pathname, but `isfile` also works with a partial path, relative to the current working directory. You can use `os.getcwd()` to get the current working directory.
- ❹ `os.path` also has a `isdir` function which returns 1 if the path represents a directory, and 0 otherwise. You can use this to get a list of the subdirectories within a directory.

### Example 6.19. Listing Directories in `fileinfo.py`

```
def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                 for f in os.listdir(directory)] ❶ ❷
    fileList = [os.path.join(directory, f)
                 for f in fileList
                 if os.path.splitext(f)[1] in fileExtList] ❸ ❹ ❺
```

- ❶ `os.listdir(directory)` returns a list of all the files and folders in `directory`.
- ❷ Iterating through the list with `f`, you use `os.path.normcase(f)` to normalize the case according to operating system defaults. `normcase` is a useful little function that compensates for case-insensitive operating systems that think that `mahadeva.mp3` and `mahadeva.MP3` are the same file. For instance, on Windows and Mac OS, `normcase` will convert the entire filename to lowercase; on UNIX-compatible systems, it will return the filename unchanged.
- ❸ Iterating through the normalized list with `f` again, you use `os.path.splitext(f)` to split each filename into name and extension.
- ❹ For each file, you see if the extension is in the list of file extensions you care about (`fileExtList`, which was passed to the `listDirectory` function).
- ❺ For each file you care about, you use `os.path.join(directory, f)` to construct the full pathname of the file, and return a list of the full pathnames.

Whenever possible, you should use the functions in `os` and `os.path` for file, directory, and path manipulations. These modules are wrappers for platform-specific modules, so functions like `os.path.split` work on UNIX, Windows, Mac OS, and any other platform supported by Python.

There is one other way to get the contents of a directory. It's very powerful, and it uses the sort of wildcards that you may already be familiar with from working on the command line.

### Example 6.20. Listing Directories with `glob`

```
>>> os.listdir("c:\\music\\_singles\\") ❶
['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')
['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')
['c:\\music\\_singles\\sidewinder.mp3',
 'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\*\\*.mp3')

```

- ❶ As you saw earlier, `os.listdir` simply takes a directory path and lists all files and directories in that directory.
- ❷ The `glob` module, on the other hand, takes a wildcard and returns the full path of all files and directories matching the wildcard. Here the wildcard is a directory path plus `"*.mp3"`, which will match all `.mp3` files. Note that each element of the returned list already includes the full path of the file.
- ❸ If you want to find all the files in a specific directory that start with `"s"` and end with `".mp3"`, you can do that too.
- ❹ Now consider this scenario: you have a `music` directory, with several subdirectories within it, with `.mp3` files within each subdirectory. You can get a list of all of those with a single call to `glob`, by using two wildcards at once. One wildcard is the `"*.mp3"` (to match `.mp3` files), and one wildcard is *within the directory path itself*, to match any subdirectory within `c:\\music`. That's a crazy amount of power packed into one deceptively simple-looking function!

### Further Reading on the `os` Module

- Python Knowledge Base (<http://www.faqs.com/knowledge-base/index.phtml/fid/199/>) answers questions about the `os` module (<http://www.faqs.com/knowledge-base/index.phtml/fid/240/>).
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) documents the `os` (<http://www.python.org/doc/current/lib/module-os.html>) module and the `os.path` (<http://www.python.org/doc/current/lib/module-os.path.html>) module.

## 6.6. Putting It All Together

Once again, all the dominoes are in place. You've seen how each line of code works. Now let's step back and see how it all fits together.

### Example 6.21. `listDirectory`

```

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    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__]):
        "get file info class from filename extension"
        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]

```

❶

`listDirectory` is the main attraction of this entire module. It takes a directory (like `c:\music\_singles\` in my case) and a list of interesting file extensions (like `[ '.mp3' ]`), and it returns a list of class instances that act like dictionaries that contain metadata about each interesting file in that directory. And it does it in just a few straightforward lines of code.

- ❷ As you saw in the previous section, this line of code gets a list of the full pathnames of all the files in directory that have an interesting file extension (as specified by `fileExtList`).
- ❸ Old-school Pascal programmers may be familiar with them, but most people give me a blank stare when I tell them that Python supports *nested functions* — literally, a function within a function. The nested function `getFileInfoClass` can be called only from the function in which it is defined, `listDirectory`. As with any other function, you don't need an interface declaration or anything fancy; just define the function and code it.
- ❹ Now that you've seen the `os` module, this line should make more sense. It gets the extension of the file (`os.path.splitext(filename)[1]`), forces it to uppercase (`.upper()`), slices off the dot (`[1:]`), and constructs a class name out of it with string formatting. So `c:\music\ap\mahadeva.mp3` becomes `.mp3` becomes `.MP3` becomes `MP3` becomes `MP3FileInfo`.
- ❺ Having constructed the name of the handler class that would handle this file, you check to see if that handler class actually exists in this module. If it does, you return the class, otherwise you return the base class `FileInfo`. This is a very important point: *this function returns a class*. Not an instance of a class, but the class itself.
- ❻ For each file in the "interesting files" list (`fileList`), you call `getFileInfoClass` with the filename (`f`). Calling `getFileInfoClass(f)` returns a class; you don't know exactly which class, but you don't care. You then create an instance of this class (whatever it is) and pass the filename (`f` again), to the `__init__` method. As you saw earlier in this chapter, the `__init__` method of `FileInfo` sets `self["name"]`, which triggers `__setitem__`, which is overridden in the descendant (`MP3FileInfo`) to parse the file appropriately to pull out the file's metadata. You do all that for each interesting file and return a list of the resulting instances.

Note that `listDirectory` is completely generic. It doesn't know ahead of time which types of files it will be getting, or which classes are defined that could potentially handle those files. It inspects the directory for the files to process, and then introspects its own module to see what special handler classes (like `MP3FileInfo`) are defined. You can extend this program to handle other types of files simply by defining an appropriately-named class: `HTMLFileInfo` for HTML files, `DOCFileInfo` for Word `.doc` files, and so forth. `listDirectory` will handle them all, without modification, by handing off the real work to the appropriate classes and collating the results.

## 6.7. Summary

The `fileinfo.py` program introduced in Chapter 5 should now make perfect sense.

```
"""Framework for getting filetype-specific metadata.
```

```
Instantiate appropriate class with filename. Returned object acts like a
dictionary, with key-value pairs for each piece of metadata.
```

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

```
Or use listDirectory function to get info on all files in a directory.
```

```
for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
    ...
```

```
Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo. Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
```

```
"""
import os
```

```

import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = { "title"      : ( 3, 33, stripnulls),
                   "artist"    : (33, 63, stripnulls),
                   "album"     : (63, 93, stripnulls),
                   "year"      : (93, 97, stripnulls),
                   "comment"   : (97, 126, stripnulls),
                   "genre"     : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        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 == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    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__]):
        "get file info class from filename extension"
        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

```

Before diving into the next chapter, make sure you're comfortable doing the following things:

- Catching exceptions with `try...except`
- Protecting external resources with `try...finally`
- Reading from files
- Assigning multiple values at once in a `for` loop
- Using the `os` module for all your cross-platform file manipulation needs
- Dynamically instantiating classes of unknown type by treating classes as objects and passing them around

# Chapter 7. Regular Expressions

Regular expressions are a powerful and standardized way of searching, replacing, and parsing text with complex patterns of characters. If you've used regular expressions in other languages (like Perl), the syntax will be very familiar, and you get by just reading the summary of the `re` module (<http://www.python.org/doc/current/lib/module-re.html>) to get an overview of the available functions and their arguments.

## 7.1. Diving In

Strings have methods for searching (`index`, `find`, and `count`), replacing (`replace`), and parsing (`split`), but they are limited to the simplest of cases. The search methods look for a single, hard-coded substring, and they are always case-sensitive. To do case-insensitive searches of a string `s`, you must call `s.lower()` or `s.upper()` and make sure your search strings are the appropriate case to match. The `replace` and `split` methods have the same limitations.

If what you're trying to do can be accomplished with string functions, you should use them. They're fast and simple and easy to read, and there's a lot to be said for fast, simple, readable code. But if you find yourself using a lot of different string functions with `if` statements to handle special cases, or if you're combining them with `split` and `join` and list comprehensions in weird unreadable ways, you may need to move up to regular expressions.

Although the regular expression syntax is tight and unlike normal code, the result can end up being *more* readable than a hand-rolled solution that uses a long chain of string functions. There are even ways of embedding comments within regular expressions to make them practically self-documenting.

## 7.2. Case Study: Street Addresses

This series of examples was inspired by a real-life problem I had in my day job several years ago, when I needed to scrub and standardize street addresses exported from a legacy system before importing them into a newer system. (See, I don't just make this stuff up; it's actually useful.) This example shows how I approached the problem.

### Example 7.1. Matching at the End of a String

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.') ❶
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.') ❷
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.') ❸
'100 NORTH BROAD RD.'
>>> import re
>>> re.sub('ROAD$', 'RD.', s) ❹ ❺ ❻
'100 NORTH BROAD RD.'
```

- ❶ My goal is to standardize a street address so that 'ROAD' is always abbreviated as 'RD.'. At first glance, I thought this was simple enough that I could just use the string method `replace`. After all, all the data was already uppercase, so case mismatches would not be a problem. And the search string, 'ROAD', was a constant. And in this deceptively simple example, `s.replace` does indeed work.
- ❷ Life, unfortunately, is full of counterexamples, and I quickly discovered this one. The problem here is that 'ROAD' appears twice in the address, once as part of the street name 'BROAD' and once as its own word. The

replace method sees these two occurrences and blindly replaces both of them; meanwhile, I see my addresses getting destroyed.

- ❸ To solve the problem of addresses with more than one 'ROAD' substring, you could resort to something like this: only search and replace 'ROAD' in the last four characters of the address (`s[-4:]`), and leave the string alone (`s[:-4]`). But you can see that this is already getting unwieldy. For example, the pattern is dependent on the length of the string you're replacing (if you were replacing 'STREET' with 'ST.', you would need to use `s[:-6]` and `s[-6:].replace(...)`). Would you like to come back in six months and debug this? I know I wouldn't.
- ❹ It's time to move up to regular expressions. In Python, all functionality related to regular expressions is contained in the `re` module.
- ❺ Take a look at the first parameter: 'ROAD\$'. This is a simple regular expression that matches 'ROAD' only when it occurs at the end of a string. The `$` means "end of the string". (There is a corresponding character, the caret `^`, which means "beginning of the string".)
- ❻ Using the `re.sub` function, you search the string `s` for the regular expression 'ROAD\$' and replace it with 'RD.'. This matches the ROAD at the end of the string `s`, but does *not* match the ROAD that's part of the word BROAD, because that's in the middle of `s`.

Continuing with my story of scrubbing addresses, I soon discovered that the previous example, matching 'ROAD' at the end of the address, was not good enough, because not all addresses included a street designation at all; some just ended with the street name. Most of the time, I got away with it, but if the street name was 'BROAD', then the regular expression would match 'ROAD' at the end of the string as part of the word 'BROAD', which is not what I wanted.

## Example 7.2. Matching Whole Words

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s) ❶
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s) ❷
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s) ❸
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s) ❹
'100 BROAD RD. APT 3'
```

- ❶ What I *really* wanted was to match 'ROAD' when it was at the end of the string *and* it was its own whole word, not a part of some larger word. To express this in a regular expression, you use `\b`, which means "a word boundary must occur right here". In Python, this is complicated by the fact that the `'\'` character in a string must itself be escaped. This is sometimes referred to as the backslash plague, and it is one reason why regular expressions are easier in Perl than in Python. On the down side, Perl mixes regular expressions with other syntax, so if you have a bug, it may be hard to tell whether it's a bug in syntax or a bug in your regular expression.
- ❷ To work around the backslash plague, you can use what is called a raw string, by prefixing the string with the letter `r`. This tells Python that nothing in this string should be escaped; `'\t'` is a tab character, but `r'\t'` is really the backslash character `\` followed by the letter `t`. I recommend always using raw strings when dealing with regular expressions; otherwise, things get too confusing too quickly (and regular expressions get confusing quickly enough all by themselves).
- ❸ *\*sigh\** Unfortunately, I soon found more cases that contradicted my logic. In this case, the street address contained the word 'ROAD' as a whole word by itself, but it wasn't at the end, because the address had an apartment number after the street designation. Because 'ROAD' isn't at the very end of

the string, it doesn't match, so the entire call to `re.sub` ends up replacing nothing at all, and you get the original string back, which is not what you want.

- ④ To solve this problem, I removed the `$` character and added another `\b`. Now the regular expression reads "match 'ROAD' when it's a whole word by itself anywhere in the string," whether at the end, the beginning, or somewhere in the middle.

## 7.3. Case Study: Roman Numerals

You've most likely seen Roman numerals, even if you didn't recognize them. You may have seen them in copyrights of old movies and television shows ("Copyright MCMXLVI" instead of "Copyright 1946"), or on the dedication walls of libraries or universities ("established MDCCCLXXXVIII" instead of "established 1888"). You may also have seen them in outlines and bibliographical references. It's a system of representing numbers that really does date back to the ancient Roman empire (hence the name).

In Roman numerals, there are seven characters that are repeated and combined in various ways to represent numbers.

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

The following are some general rules for constructing Roman numerals:

- Characters are additive. I is 1, II is 2, and III is 3. VI is 6 (literally, "5 and 1"), VII is 7, and VIII is 8.
- The tens characters (I, X, C, and M) can be repeated up to three times. At 4, you need to subtract from the next highest fives character. You can't represent 4 as IIII; instead, it is represented as IV ("1 less than 5"). The number 40 is written as XL (10 less than 50), 41 as XLI, 42 as XLII, 43 as XLIII, and then 44 as XLIV (10 less than 50, then 1 less than 5).
- Similarly, at 9, you need to subtract from the next highest tens character: 8 is VIII, but 9 is IX (1 less than 10), not VIIII (since the I character can not be repeated four times). The number 90 is XC, 900 is CM.
- The fives characters can not be repeated. The number 10 is always represented as X, never as VV. The number 100 is always C, never LL.
- Roman numerals are always written highest to lowest, and read left to right, so the order the of characters matters very much. DC is 600; CD is a completely different number (400, 100 less than 500). CI is 101; IC is not even a valid Roman numeral (because you can't subtract 1 directly from 100; you would need to write it as XCIX, for 10 less than 100, then 1 less than 10).

### 7.3.1. Checking for Thousands

What would it take to validate that an arbitrary string is a valid Roman numeral? Let's take it one digit at a time. Since Roman numerals are always written highest to lowest, let's start with the highest: the thousands place. For numbers 1000 and higher, the thousands are represented by a series of M characters.

#### Example 7.3. Checking for Thousands

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')
<SRE_Match object at 0106FB58>
```

①  
②



```
>>> re.search(pattern, 'MM') ❸
<SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM') ❹
<SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM') ❺
>>> re.search(pattern, '') ❻
<SRE_Match object at 0106F4A8>
```

❶ This pattern has three parts:

- `^` to match what follows only at the beginning of the string. If this were not specified, the pattern would match no matter where the M characters were, which is not what you want. You want to make sure that the M characters, if they're there, are at the beginning of the string.
- `M?` to optionally match a single M character. Since this is repeated three times, you're matching anywhere from zero to three M characters in a row.
- `$` to match what precedes only at the end of the string. When combined with the `^` character at the beginning, this means that the pattern must match the entire string, with no other characters before or after the M characters.

❷ The essence of the `re` module is the `search` function, that takes a regular expression (`pattern`) and a string (`'M'`) to try to match against the regular expression. If a match is found, `search` returns an object which has various methods to describe the match; if no match is found, `search` returns `None`, the Python null value. All you care about at the moment is whether the pattern matches, which you can tell by just looking at the return value of `search`. `'M'` matches this regular expression, because the first optional M matches and the second and third optional M characters are ignored.

❸ `'MM'` matches because the first and second optional M characters match and the third M is ignored.

❹ `'MMM'` matches because all three M characters match.

❺ `'MMMM'` does not match. All three M characters match, but then the regular expression insists on the string ending (because of the `$` character), and the string doesn't end yet (because of the fourth M). So `search` returns `None`.

❻ Interestingly, an empty string also matches this regular expression, since all the M characters are optional.

### 7.3.2. Checking for Hundreds

The hundreds place is more difficult than the thousands, because there are several mutually exclusive ways it could be expressed, depending on its value.

- 100 = C
- 200 = CC
- 300 = CCC
- 400 = CD
- 500 = D
- 600 = DC
- 700 = DCC
- 800 = DCCC
- 900 = CM

So there are four possible patterns:

- CM
- CD
- Zero to three C characters (zero if the hundreds place is 0)
- D, followed by zero to three C characters

The last two patterns can be combined:

- an optional D, followed by zero to three C characters

This example shows how to validate the hundreds place of a Roman numeral.

#### Example 7.4. Checking for Hundreds

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$' ❶
>>> re.search(pattern, 'MCM') ❷
<SRE_Match object at 01070390>
>>> re.search(pattern, 'MD') ❸
<SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC') ❹
<SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC') ❺
>>> re.search(pattern, '') ❻
<SRE_Match object at 01071D98>
```

- ❶ This pattern starts out the same as the previous one, checking for the beginning of the string (^), then the thousands place (M?M?M?). Then it has the new part, in parentheses, which defines a set of three mutually exclusive patterns, separated by vertical bars: CM, CD, and D?C?C?C? (which is an optional D followed by zero to three optional C characters). The regular expression parser checks for each of these patterns in order (from left to right), takes the first one that matches, and ignores the rest.
- ❷ 'MCM' matches because the first M matches, the second and third M characters are ignored, and the CM matches (so the CD and D?C?C?C? patterns are never even considered). MCM is the Roman numeral representation of 1900.
- ❸ 'MD' matches because the first M matches, the second and third M characters are ignored, and the D?C?C?C? pattern matches D (each of the three C characters are optional and are ignored). MD is the Roman numeral representation of 1500.
- ❹ 'MMMCCC' matches because all three M characters match, and the D?C?C?C? pattern matches CCC (the D is optional and is ignored). MMMCCC is the Roman numeral representation of 3300.
- ❺ 'MCMC' does not match. The first M matches, the second and third M characters are ignored, and the CM matches, but then the \$ does not match because you're not at the end of the string yet (you still have an unmatched C character). The C does *not* match as part of the D?C?C?C? pattern, because the mutually exclusive CM pattern has already matched.
- ❻ Interestingly, an empty string still matches this pattern, because all the M characters are optional and ignored, and the empty string matches the D?C?C?C? pattern where all the characters are optional and ignored.

Whew! See how quickly regular expressions can get nasty? And you've only covered the thousands and hundreds places of Roman numerals. But if you followed all that, the tens and ones places are easy, because they're exactly the same pattern. But let's look at another way to express the pattern.

## 7.4. Using the {n,m} Syntax

In the previous section, you were dealing with a pattern where the same character could be repeated up to three times. There is another way to express this in regular expressions, which some people find more readable. First look at the method we already used in the previous example.

#### Example 7.5. The Old Way: Every Character Optional

```

>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M') ❶
<_sre.SRE_Match object at 0x008EE090>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MM') ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MMM') ❸
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM') ❹
>>>

```

- ❶ This matches the start of the string, and then the first optional M, but not the second and third M (but that's okay because they're optional), and then the end of the string.
- ❷ This matches the start of the string, and then the first and second optional M, but not the third M (but that's okay because it's optional), and then the end of the string.
- ❸ This matches the start of the string, and then all three optional M, and then the end of the string.
- ❹ This matches the start of the string, and then all three optional M, but then does not match the the end of the string (because there is still one unmatched M), so the pattern does not match and returns None.

### Example 7.6. The New Way: From `n o m`

```

>>> pattern = '^M{0,3}$' ❶
>>> re.search(pattern, 'M') ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM') ❸
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM') ❹
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM') ❺
>>>

```

- ❶ This pattern says: "Match the start of the string, then anywhere from zero to three M characters, then the end of the string." The 0 and 3 can be any numbers; if you want to match at least one but no more than three M characters, you could say `M{1,3}`.
- ❷ This matches the start of the string, then one M out of a possible three, then the end of the string.
- ❸ This matches the start of the string, then two M out of a possible three, then the end of the string.
- ❹ This matches the start of the string, then three M out of a possible three, then the end of the string.
- ❺ This matches the start of the string, then three M out of a possible three, but then *does not match* the end of the string. The regular expression allows for up to only three M characters before the end of the string, but you have four, so the pattern does not match and returns None.

There is no way to programmatically determine that two regular expressions are equivalent. The best you can do is write a lot of test cases to make sure they behave the same way on all relevant inputs. You'll talk more about writing test cases later in this book.

## 7.4.1. Checking for Tens and Ones

Now let's expand the Roman numeral regular expression to cover the tens and ones place. This example shows the check for tens.

### Example 7.7. Checking for Tens

```
>>> pattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL') ❶
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML') ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX') ❸
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX') ❹
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX') ❺
>>>
```

- ❶ This matches the start of the string, then the first optional M, then CM, then XL, then the end of the string. Remember, the (A|B|C) syntax means "match exactly one of A, B, or C". You match XL, so you ignore the XC and L?X?X?X? choices, and then move on to the end of the string. MCML is the Roman numeral representation of 1940.
- ❷ This matches the start of the string, then the first optional M, then CM, then L?X?X?X?. Of the L?X?X?X?, it matches the L and skips all three optional X characters. Then you move to the end of the string. MCML is the Roman numeral representation of 1950.
- ❸ This matches the start of the string, then the first optional M, then CM, then the optional L and the first optional X, skips the second and third optional X, then the end of the string. MCMLX is the Roman numeral representation of 1960.
- ❹ This matches the start of the string, then the first optional M, then CM, then the optional L and all three optional X characters, then the end of the string. MCMLXXX is the Roman numeral representation of 1980.
- ❺ This matches the start of the string, then the first optional M, then CM, then the optional L and all three optional X characters, then *fails to match* the end of the string because there is still one more X unaccounted for. So the entire pattern fails to match, and returns None. MCMLXXXX is not a valid Roman numeral.

The expression for the ones place follows the same pattern. I'll spare you the details and show you the end result.

```
>>> pattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

So what does that look like using this alternate {n,m} syntax? This example shows the new syntax.

### Example 7.8. Validating Roman Numerals with {n,m}

```
>>> pattern = '^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV') ❶
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMDCLXVI') ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMMDCCCLXXXVIII') ❸
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'I') ❹
<_sre.SRE_Match object at 0x008EEB48>
```

- ❶ This matches the start of the string, then one of a possible four M characters, then D?C{0,3}. Of that, it matches the optional D and zero of three possible C characters. Moving on, it matches L?X{0,3} by matching the optional L and zero of three possible X characters. Then it matches V?I{0,3} by matching the optional V and zero of three possible I characters, and finally the end of the string. MDLV is the Roman numeral representation of 1555.
- ❷ This matches the start of the string, then two of a possible four M characters, then the D?C{0,3} with a D and one of three possible C characters; then L?X{0,3} with an L and one of three possible X characters; then V?I{0,3} with a V and one of three possible I characters; then the end of the string. MMDCLXVI is the

Roman numeral representation of 2666.

- ③ This matches the start of the string, then four out of four M characters, then `D?C{0,3}` with a D and three out of three C characters; then `L?X{0,3}` with an L and three out of three X characters; then `V?I{0,3}` with a V and three out of three I characters; then the end of the string. MMMDCCCLXXXVIIII is the Roman numeral representation of 3888, and it's the longest Roman numeral you can write without extended syntax.
- ④ Watch closely. (I feel like a magician. "Watch closely, kids, I'm going to pull a rabbit out of my hat.") This matches the start of the string, then zero out of four M, then matches `D?C{0,3}` by skipping the optional D and matching zero out of three C, then matches `L?X{0,3}` by skipping the optional L and matching zero out of three X, then matches `V?I{0,3}` by skipping the optional V and matching one out of three I. Then the end of the string. Whoa.

If you followed all that and understood it on the first try, you're doing better than I did. Now imagine trying to understand someone else's regular expressions, in the middle of a critical function of a large program. Or even imagine coming back to your own regular expressions a few months later. I've done it, and it's not a pretty sight.

In the next section you'll explore an alternate syntax that can help keep your expressions maintainable.

## 7.5. Verbose Regular Expressions

So far you've just been dealing with what I'll call "compact" regular expressions. As you've seen, they are difficult to read, and even if you figure out what one does, that's no guarantee that you'll be able to understand it six months later. What you really need is inline documentation.

Python allows you to do this with something called *verbose regular expressions*. A verbose regular expression is different from a compact regular expression in two ways:

- Whitespace is ignored. Spaces, tabs, and carriage returns are not matched as spaces, tabs, and carriage returns. They're not matched at all. (If you want to match a space in a verbose regular expression, you'll need to escape it by putting a backslash in front of it.)
- Comments are ignored. A comment in a verbose regular expression is just like a comment in Python code: it starts with a # character and goes until the end of the line. In this case it's a comment within a multi-line string instead of within your source code, but it works the same way.

This will be more clear with an example. Let's revisit the compact regular expression you've been working with, and make it a verbose regular expression. This example shows how.

### Example 7.9. Regular Expressions with Inline Comments

```
>>> pattern = """
^                # 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.search(pattern, 'M', re.VERBOSE)           ❶
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)  ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMMDCCCLXXXVIII', re.VERBOSE)  ❸
```

```
<_sre.SRE_Match object at 0x008EEB48>  
>>> re.search(pattern, 'M')
```

❹

- ❶ The most important thing to remember when using verbose regular expressions is that you need to pass an extra argument when working with them: `re.VERBOSE` is a constant defined in the `re` module that signals that the pattern should be treated as a verbose regular expression. As you can see, this pattern has quite a bit of whitespace (all of which is ignored), and several comments (all of which are ignored). Once you ignore the whitespace and the comments, this is exactly the same regular expression as you saw in the previous section, but it's a lot more readable.
- ❷ This matches the start of the string, then one of a possible four M, then CM, then L and three of a possible three X, then IX, then the end of the string.
- ❸ This matches the start of the string, then four of a possible four M, then D and three of a possible three C, then L and three of a possible three X, then V and three of a possible three I, then the end of the string.
- ❹ This does not match. Why? Because it doesn't have the `re.VERBOSE` flag, so the `re.search` function is treating the pattern as a compact regular expression, with significant whitespace and literal hash marks. Python can't auto-detect whether a regular expression is verbose or not. Python assumes every regular expression is compact unless you explicitly state that it is verbose.

## 7.6. Case study: Parsing Phone Numbers

So far you've concentrated on matching whole patterns. Either the pattern matches, or it doesn't. But regular expressions are much more powerful than that. When a regular expression *does* match, you can pick out specific pieces of it. You can find out what matched where.

This example came from another real-world problem I encountered, again from a previous day job. The problem: parsing an American phone number. The client wanted to be able to enter the number free-form (in a single field), but then wanted to store the area code, trunk, number, and optionally an extension separately in the company's database. I scoured the Web and found many examples of regular expressions that purported to do this, but none of them were permissive enough.

Here are the phone numbers I needed to be able to accept:

- 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

Quite a variety! In each of these cases, I need to know that the area code was 800, the trunk was 555, and the rest of the phone number was 1212. For those with an extension, I need to know that the extension was 1234.

Let's work through developing a solution for phone number parsing. This example shows the first step.

### Example 7.10. Finding Numbers

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$') ❶  
>>> phonePattern.search('800-555-1212').groups() ❷  
( '800', '555', '1212' )
```

```
>>> phonePattern.search('800-555-1212-1234')
>>>
```

❸

- ❶ Always read regular expressions from left to right. This one matches the beginning of the string, and then `(\d{3})`. What's `\d{3}`? Well, the `{3}` means "match exactly three numeric digits"; it's a variation on the `{n,m}` syntax you saw earlier. `\d` means "any numeric digit" (0 through 9). Putting it in parentheses means "match exactly three numeric digits, *and then remember them as a group that I can ask for later*". Then match a literal hyphen. Then match another group of exactly three digits. Then another literal hyphen. Then another group of exactly four digits. Then match the end of the string.
- ❷ To get access to the groups that the regular expression parser remembered along the way, use the `groups()` method on the object that the `search` function returns. It will return a tuple of however many groups were defined in the regular expression. In this case, you defined three groups, one with three digits, one with three digits, and one with four digits.
- ❸ This regular expression is not the final answer, because it doesn't handle a phone number with an extension on the end. For that, you'll need to expand the regular expression.

### Example 7.11. Finding the Extension

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$')
>>> phonePattern.search('800-555-1212-1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')
>>>
>>> phonePattern.search('800-555-1212')
>>>
```

❶

❷

❸

❹

- ❶ This regular expression is almost identical to the previous one. Just as before, you match the beginning of the string, then a remembered group of three digits, then a hyphen, then a remembered group of three digits, then a hyphen, then a remembered group of four digits. What's new is that you then match another hyphen, and a remembered group of one or more digits, then the end of the string.
- ❷ The `groups()` method now returns a tuple of four elements, since the regular expression now defines four groups to remember.
- ❸ Unfortunately, this regular expression is not the final answer either, because it assumes that the different parts of the phone number are separated by hyphens. What if they're separated by spaces, or commas, or dots? You need a more general solution to match several different types of separators.
- ❹ Oops! Not only does this regular expression not do everything you want, it's actually a step backwards, because now you can't parse phone numbers *without* an extension. That's not what you wanted at all; if the extension is there, you want to know what it is, but if it's not there, you still want to know what the different parts of the main number are.

The next example shows the regular expression to handle separators between the different parts of the phone number.

### Example 7.12. Handling Different Separators

```
>>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$')
>>> phonePattern.search('800 555 1212 1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('80055512121234')
>>>
>>> phonePattern.search('800-555-1212')
>>>
```

❶

❷

❸

❹

❺



- ❶ Hang on to your hat. You're matching the beginning of the string, then a group of three digits, then `\D+`. What the heck is that? Well, `\D` matches any character *except* a numeric digit, and `+` means "1 or more". So `\D+` matches one or more characters that are not digits. This is what you're using instead of a literal hyphen, to try to match different separators.
- ❷ Using `\D+` instead of `-` means you can now match phone numbers where the parts are separated by spaces instead of hyphens.
- ❸ Of course, phone numbers separated by hyphens still work too.
- ❹ Unfortunately, this is still not the final answer, because it assumes that there is a separator at all. What if the phone number is entered without any spaces or hyphens at all?
- ❺ Oops! This still hasn't fixed the problem of requiring extensions. Now you have two problems, but you can solve both of them with the same technique.

The next example shows the regular expression for handling phone numbers *without* separators.

### Example 7.13. Handling Numbers Without Separators

```
>>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ❶
>>> phonePattern.search('80055512121234').groups() ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups() ❸
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ❹
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234') ❺
>>>
```

- ❶ The only change you've made since that last step is changing all the `+` to `*`. Instead of `\D+` between the parts of the phone number, you now match on `\D*`. Remember that `+` means "1 or more"? Well, `*` means "zero or more". So now you should be able to parse phone numbers even when there is no separator character at all.
- ❷ Lo and behold, it actually works. Why? You matched the beginning of the string, then a remembered group of three digits (800), then zero non-numeric characters, then a remembered group of three digits (555), then zero non-numeric characters, then a remembered group of four digits (1212), then zero non-numeric characters, then a remembered group of an arbitrary number of digits (1234), then the end of the string.
- ❸ Other variations work now too: dots instead of hyphens, and both a space and an `x` before the extension.
- ❹ Finally, you've solved the other long-standing problem: extensions are optional again. If no extension is found, the `groups()` method still returns a tuple of four elements, but the fourth element is just an empty string.
- ❺ I hate to be the bearer of bad news, but you're not finished yet. What's the problem here? There's an extra character before the area code, but the regular expression assumes that the area code is the first thing at the beginning of the string. No problem, you can use the same technique of "zero or more non-numeric characters" to skip over the leading characters before the area code.

The next example shows how to handle leading characters in phone numbers.

### Example 7.14. Handling Leading Characters

```
>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ❶
>>> phonePattern.search('(800)5551212 ext. 1234').groups() ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ❸
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234') ❹
>>>
```



- ❶ This is the same as in the previous example, except now you're matching `\D*`, zero or more non-numeric characters, before the first remembered group (the area code). Notice that you're not remembering these non-numeric characters (they're not in parentheses). If you find them, you'll just skip over them and then start remembering the area code whenever you get to it.
- ❷ You can successfully parse the phone number, even with the leading left parenthesis before the area code. (The right parenthesis after the area code is already handled; it's treated as a non-numeric separator and matched by the `\D*` after the first remembered group.)
- ❸ Just a sanity check to make sure you haven't broken anything that used to work. Since the leading characters are entirely optional, this matches the beginning of the string, then zero non-numeric characters, then a remembered group of three digits (800), then one non-numeric character (the hyphen), then a remembered group of three digits (555), then one non-numeric character (the hyphen), then a remembered group of four digits (1212), then zero non-numeric characters, then a remembered group of zero digits, then the end of the string.
- ❹ This is where regular expressions make me want to gouge my eyes out with a blunt object. Why doesn't this phone number match? Because there's a 1 before the area code, but you assumed that all the leading characters before the area code were non-numeric characters (`\D*`). Aargh.

Let's back up for a second. So far the regular expressions have all matched from the beginning of the string. But now you see that there may be an indeterminate amount of stuff at the beginning of the string that you want to ignore. Rather than trying to match it all just so you can skip over it, let's take a different approach: don't explicitly match the beginning of the string at all. This approach is shown in the next example.

### Example 7.15. Phone Number, Wherever I May Find Ye

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ❶
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups() ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212') ❸
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234') ❹
('800', '555', '1212', '1234')
```

- ❶ Note the lack of `^` in this regular expression. You are not matching the beginning of the string anymore. There's nothing that says you need to match the entire input with your regular expression. The regular expression engine will do the hard work of figuring out where the input string starts to match, and go from there.
- ❷ Now you can successfully parse a phone number that includes leading characters and a leading digit, plus any number of any kind of separators around each part of the phone number.
- ❸ Sanity check. this still works.
- ❹ That still works too.

See how quickly a regular expression can get out of control? Take a quick glance at any of the previous iterations. Can you tell the difference between one and the next?

While you still understand the final answer (and it is the final answer; if you've discovered a case it doesn't handle, I don't want to know about it), let's write it out as a verbose regular expression, before you forget why you made the choices you made.

### Example 7.16. Parsing Phone Numbers (Final Version)

```
>>> phonePattern = re.compile(r'''
    # don't match beginning of string, number can start anywhere
    (\d{3})    # area code is 3 digits (e.g. '800')
```

```

\D*      # optional separator is any number of non-digits
(\d{3})  # trunk is 3 digits (e.g. '555')
\D*      # optional separator
(\d{4})  # rest of number is 4 digits (e.g. '1212')
\D*      # optional separator
(\d*)    # extension is optional and can be any number of digits
$        # end of string
''' , re.VERBOSE)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')
('800', '555', '1212', '')

```

❶

❷

- ❶ Other than being spread out over multiple lines, this is exactly the same regular expression as the last step, so it's no surprise that it parses the same inputs.
- ❷ Final sanity check. Yes, this still works. You're done.

### Further Reading on Regular Expressions

- Regular Expression HOWTO (<http://py-howto.sourceforge.net/regex/regex.html>) teaches about regular expressions and how to use them in Python.
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) summarizes the `re` module (<http://www.python.org/doc/current/lib/module-re.html>).

## 7.7. Summary

This is just the tiniest tip of the iceberg of what regular expressions can do. In other words, even though you're completely overwhelmed by them now, believe me, you ain't seen nothing yet.

You should now be familiar with the following techniques:

- `^` matches the beginning of a string.
- `$` matches the end of a string.
- `\b` matches a word boundary.
- `\d` matches any numeric digit.
- `\D` matches any non-numeric character.
- `x?` matches an optional `x` character (in other words, it matches an `x` zero or one times).
- `x*` matches `x` zero or more times.
- `x+` matches `x` one or more times.
- `x{n,m}` matches an `x` character at least `n` times, but not more than `m` times.
- `(a|b|c)` matches either `a` or `b` or `c`.
- `(x)` in general is a *remembered group*. You can get the value of what matched by using the `groups()` method of the object returned by `re.search`.

Regular expressions are extremely powerful, but they are not the correct solution for every problem. You should learn enough about them to know when they are appropriate, when they will solve your problems, and when they will cause more problems than they solve.

Some people, when confronted with a problem, think "I know, I'll use regular expressions."  
Now they have two problems.

—Jamie Zawinski, in comp.emacs.xemacs  
(<http://groups.google.com/groups?selm=33F0C496.370D7C45%40netscape.com>)

# Chapter 8. HTML Processing

## 8.1. Diving in

I often see questions on `comp.lang.python` (<http://groups.google.com/groups?group=comp.lang.python>) like "How can I list all the [headers|images|links] in my HTML document?" "How do I parse/translate/munge the text of my HTML document but leave the tags alone?" "How can I add/remove/quote attributes of all my HTML tags at once?" This chapter will answer all of these questions.

Here is a complete, working Python program in two parts. The first part, `BaseHTMLProcessor.py`, is a generic tool to help you process HTML files by walking through the tags and text blocks. The second part, `dialect.py`, is an example of how to use `BaseHTMLProcessor.py` to translate the text of an HTML document but leave the tags alone. Read the `doc strings` and comments to get an overview of what's going on. Most of it will seem like black magic, because it's not obvious how any of these class methods ever get called. Don't worry, all will be revealed in due time.

### Example 8.1. `BaseHTMLProcessor.py`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
from sgmllib import SGMLParser
import htmlentitydefs

class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        # extend (called by SGMLParser.__init__)
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs):
        # called for each start tag
        # attrs is a list of (attr, value) tuples
        # e.g. for <pre class="screen">, tag="pre", attrs=[("class", "screen")]
        # Ideally we would like to reconstruct original tag and attributes, but
        # we may end up quoting attribute values that weren't quoted in the source
        # document, or we may change the type of quotes around the attribute value
        # (single to double quotes).
        # Note that improperly embedded non-HTML code (like client-side Javascript)
        # may be parsed incorrectly by the ancestor, causing runtime script errors.
        # All non-HTML code must be enclosed in HTML comment tags (<!-- code -->)
        # to ensure that it will pass through this parser unaltered (in 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):
        # called for each end tag, e.g. for </pre>, tag will be "pre"
        # Reconstruct the original end tag.
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):
        # called for each character reference, e.g. for "&#160;", ref will be "160"
        # Reconstruct the original character reference.
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):
```

```

        # called for each entity reference, e.g. for "&copy;", ref will be "copy"
        # Reconstruct the original entity reference.
        self.pieces.append("&%(ref)s" % locals())
        # standard HTML entities are closed with a semicolon; other entities are not
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

def handle_data(self, text):
    # called for each block of plain text, i.e. outside of any tag and
    # not containing any character or entity references
    # Store the original text verbatim.
    self.pieces.append(text)

def handle_comment(self, text):
    # called for each HTML comment, e.g. <!-- insert Javascript code here -->
    # Reconstruct the original comment.
    # It is especially important that the source document enclose client-side
    # code (like Javascript) within comments so it can pass through this
    # processor undisturbed; see comments in unknown_starttag for details.
    self.pieces.append("<!--%(text)s-->" % locals())

def handle_pi(self, text):
    # called for each processing instruction, e.g. <?instruction>
    # Reconstruct original processing instruction.
    self.pieces.append("<?%(text)s>" % locals())

def handle_decl(self, text):
    # called for the DOCTYPE, if present, e.g.
    # <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    # "http://www.w3.org/TR/html4/loose.dtd">
    # Reconstruct original DOCTYPE
    self.pieces.append("<!(text)s>" % locals())

def output(self):
    """Return processed HTML as a single string"""
    return "".join(self.pieces)

```

## Example 8.2. dialect.py

```

import re
from BaseHTMLProcessor import BaseHTMLProcessor

class Dialectizer(BaseHTMLProcessor):
    subs = ()

    def reset(self):
        # extend (called from __init__ in ancestor)
        # Reset all data attributes
        self.verbatim = 0
        BaseHTMLProcessor.reset(self)

    def start_pre(self, attrs):
        # called for every <pre> tag in HTML source
        # Increment verbatim mode count, then handle tag like normal
        self.verbatim += 1
        self.unknown_starttag("pre", attrs)

    def end_pre(self):
        # called for every </pre> tag in HTML source
        # Decrement verbatim mode count
        self.unknown_endtag("pre")

```

```

        self.verbatim -= 1

def handle_data(self, text):
    # override
    # called for every block of text in HTML source
    # If in verbatim mode, save text unaltered;
    # otherwise process the text with a series of substitutions
    self.pieces.append(self.verbatim and text or self.process(text))

def process(self, text):
    # called from handle_data
    # Process text block by performing series of regular expression
    # substitutions (actual substitutions are defined in descendant)
    for fromPattern, toPattern in self.subs:
        text = re.sub(fromPattern, toPattern, text)
    return text

class ChefDialectizer(Dialectizer):
    """convert HTML to Swedish Chef-speak

    based on the classic 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'\lee\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):
    """convert HTML to Elmer Fudd-speak"""
    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):
    """convert HTML to mock Middle English"""
    subs = ((r'i([bcdfghjklmnpqrstvwxyz])e\b', r'y\1'),
            (r'i([bcdfghjklmnpqrstvwxyz])e', r'y\1\le'),
            (r'ick\b', r'yk'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'e\le'),

```

```
(r'e[ea]([bcdfghjklmnpqrstvwxyz])', r'e\le'),
(r'([bcdfghjklmnpqrstvwxyz])y', r'\lee'),
(r'([bcdfghjklmnpqrstvwxyz])er', r'\lre'),
(r'([aeiou])re\b', r'\lr'),
(r'ia([bcdfghjklmnpqrstvwxyz])', r'i\le'),
(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'\le'),
(r'([rnt])\b', r'\l\le'),
(r'from', r'fro'),
(r'when', r'whan'))
```

```
def translate(url, dialectName="chef"):
    """fetch URL and translate using dialect

    dialect 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):
    """test all dialects against URL"""
    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/odbcelper_list.html")
```

### Example 8.3. Output of `dialect.py`

Running this script will translate Section 3.2, *Introducing Lists* into mock Swedish Chef-speak (`../native_data_types/chef.html`) (from *The Muppets*), mock Elmer Fudd-speak (`../native_data_types/fudd.html`) (from *Bugs Bunny* cartoons), and mock Middle English (`../native_data_types/olde.html`) (loosely based on Chaucer's *The Canterbury Tales*). If you look at the HTML source of the output pages, you'll see that all the HTML tags and attributes are untouched, but the text between the tags has been "translated" into the mock language. If you look closer, you'll see that, in fact, only the titles and paragraphs were translated; the code listings and screen examples were left untouched.

```
<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>
```

## 8.2. Introducing `sgmllib.py`

HTML processing is broken into three steps: breaking down the HTML into its constituent pieces, fiddling with the pieces, and reconstructing the pieces into HTML again. The first step is done by `sgmllib.py`, a part of the standard Python library.

The key to understanding this chapter is to realize that HTML is not just text, it is structured text. The structure is derived from the more-or-less-hierarchical sequence of start tags and end tags. Usually you don't work with HTML this way; you work with it *textually* in a text editor, or *visually* in a web browser or web authoring tool. `sgmllib.py` presents HTML *structurally*.

`sgmllib.py` contains one important class: `SGMLParser`. `SGMLParser` parses HTML into useful pieces, like start tags and end tags. As soon as it succeeds in breaking down some data into a useful piece, it calls a method on itself based on what it found. In order to use the parser, you subclass the `SGMLParser` class and override these methods. This is what I meant when I said that it presents HTML *structurally*: the structure of the HTML determines the sequence of method calls and the arguments passed to each method.

`SGMLParser` parses HTML into 8 kinds of data, and calls a separate method for each of them:

#### *Start tag*

An HTML tag that starts a block, like `<html>`, `<head>`, `<body>`, or `<pre>`, or a standalone tag like `<br>` or `<img>`. When it finds a start tag *tagname*, `SGMLParser` will look for a method called `start_tagname` or `do_tagname`. For instance, when it finds a `<pre>` tag, it will look for a `start_pre` or `do_pre` method. If found, `SGMLParser` calls this method with a list of the tag's attributes; otherwise, it calls `unknown_starttag` with the tag name and list of attributes.

#### *End tag*

An HTML tag that ends a block, like `</html>`, `</head>`, `</body>`, or `</pre>`. When it finds an end tag, `SGMLParser` will look for a method called `end_tagname`. If found, `SGMLParser` calls this method, otherwise it calls `unknown_endtag` with the tag name.

### *Character reference*

An escaped character referenced by its decimal or hexadecimal equivalent, like `&#160;`. When found, SGMLParser calls `handle_charref` with the text of the decimal or hexadecimal character equivalent.

### *Entity reference*

An HTML entity, like `&copy;`. When found, SGMLParser calls `handle_entityref` with the name of the HTML entity.

### *Comment*

An HTML comment, enclosed in `<!-- ... -->`. When found, SGMLParser calls `handle_comment` with the body of the comment.

### *Processing instruction*

An HTML processing instruction, enclosed in `<? ... >`. When found, SGMLParser calls `handle_pi` with the body of the processing instruction.

### *Declaration*

An HTML declaration, such as a DOCTYPE, enclosed in `<! ... >`. When found, SGMLParser calls `handle_decl` with the body of the declaration.

### *Text data*

A block of text. Anything that doesn't fit into the other 7 categories. When found, SGMLParser calls `handle_data` with the text.

Python 2.0 had a bug where SGMLParser would not recognize declarations at all (`handle_decl` would never be called), which meant that DOCTYPEs were silently ignored. This is fixed in Python 2.1.

`sgmlib.py` comes with a test suite to illustrate this. You can run `sgmlib.py`, passing the name of an HTML file on the command line, and it will print out the tags and other elements as it parses them. It does this by subclassing the SGMLParser class and defining `unknown_starttag`, `unknown_endtag`, `handle_data` and other methods which simply print their arguments.

In the ActivePython IDE on Windows, you can specify command line arguments in the "Run script" dialog. Separate multiple arguments with spaces.

## **Example 8.4. Sample test of `sgmlib.py`**

Here is a snippet from the table of contents of the HTML version of this book. Of course your paths may vary. (If you haven't downloaded the HTML version of the book, you can do so at <http://diveintopython.org/>.)

```
c:\python23\lib> type "c:\downloads\diveintopython\html\toc\index.html"
<!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">
```

... rest of file omitted for brevity ...

Running this through the test suite of `sgmlib.py` yields this output:

```
c:\python23\lib> python sgmlib.py "c:\downloads\diveintopython\html\toc\index.html"
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      '

```

... rest of output omitted for brevity ...

Here's the roadmap for the rest of the chapter:

- Subclass `SGMLParser` to create classes that extract interesting data out of HTML documents.
- Subclass `SGMLParser` to create `BaseHTMLProcessor`, which overrides all 8 handler methods and uses them to reconstruct the original HTML from the pieces.
- Subclass `BaseHTMLProcessor` to create `Dialectizer`, which adds some methods to process specific HTML tags specially, and overrides the `handle_data` method to provide a framework for processing the text blocks between the HTML tags.
- Subclass `Dialectizer` to create classes that define text processing rules used by `Dialectizer.handle_data`.
- Write a test suite that grabs a real web page from `http://diveintopython.org/` and processes it.

Along the way, you'll also learn about locals, globals, and dictionary-based string formatting.

## 8.3. Extracting data from HTML documents

To extract data from HTML documents, subclass the `SGMLParser` class and define methods for each tag or entity you want to capture.

The first step to extracting data from an HTML document is getting some HTML. If you have some HTML lying around on your hard drive, you can use file functions to read it, but the real fun begins when you get HTML from live web pages.

### Example 8.5. Introducing `urllib`

```

>>> import urllib
>>> sock = urllib.urlopen("http://diveintopython.org/")
>>> htmlSource = sock.read()
>>> sock.close()
>>> print htmlSource
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
  <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:mark@diveintopython.org'>
<meta name='keywords' content='Python, Dive Into Python, tutorial, object-oriented, programming, document'>
<meta name='description' content='a free Python tutorial for experienced programmers'>
</head>
<body bgcolor='white' text='black' link='#0000FF' vlink='#840084' alink='#0000FF'>
<table cellpadding='0' cellspacing='0' border='0' width='100%'>
<tr><td class='header' width='1%' valign='top'>diveintopython.org</td>
<td width='99%' align='right'><hr size='1' noshade></td></tr>
<tr><td class='tagline' colspan='2'>Python&nbsp;for&nbsp;experienced&nbsp;programmers</td></tr>

```

```
[...snip...]
```

- ❶ The `urllib` module is part of the standard Python library. It contains functions for getting information about and actually retrieving data from Internet-based URLs (mainly web pages).
- ❷ The simplest use of `urllib` is to retrieve the entire text of a web page using the `urlopen` function. Opening a URL is similar to opening a file. The return value of `urlopen` is a file-like object, which has some of the same methods as a file object.
- ❸ The simplest thing to do with the file-like object returned by `urlopen` is `read`, which reads the entire HTML of the web page into a single string. The object also supports `readlines`, which reads the text line by line into a list.
- ❹ When you're done with the object, make sure to `close` it, just like a normal file object.
- ❺ You now have the complete HTML of the home page of `http://diveintopython.org/` in a string, and you're ready to parse it.

### Example 8.6. Introducing `urllister.py`

If you have not already done so, you can download this and other examples (<http://diveintopython.org/download/diveintopython-examples-5.4.zip>) used in this book.

```
from sgmlib import SGMLParser

class URLLister(SGMLParser):
    def reset(self):
        SGMLParser.reset(self)
        self.urls = []

    def start_a(self, attrs):
        href = [v for k, v in attrs if k=='href']
        if href:
            self.urls.extend(href)
```

- ❶ `reset` is called by the `__init__` method of `SGMLParser`, and it can also be called manually once an instance of the parser has been created. So if you need to do any initialization, do it in `reset`, not in `__init__`, so that it will be re-initialized properly when someone re-uses a parser instance.
- ❷ `start_a` is called by `SGMLParser` whenever it finds an `<a>` tag. The tag may contain an `href` attribute, and/or other attributes, like `name` or `title`. The `attrs` parameter is a list of tuples, `[(attribute, value), (attribute, value), ...]`. Or it may be just an `<a>`, a valid (if useless) HTML tag, in which case `attrs` would be an empty list.
- ❸ You can find out whether this `<a>` tag has an `href` attribute with a simple multi-variable list comprehension.
- ❹ String comparisons like `k=='href'` are always case-sensitive, but that's safe in this case, because `SGMLParser` converts attribute names to lowercase while building `attrs`.

### Example 8.7. Using `urllister.py`

```
>>> import urllib, urllister
>>> usock = urllib.urlopen("http://diveintopython.org/")
>>> parser = urllister.URLLister()
>>> parser.feed(usock.read())
>>> usock.close()
>>> parser.close()
>>> for url in parser.urls: print url
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
```

... rest of output omitted for brevity ...

- ❶ Call the feed method, defined in SGMLParser, to get HTML into the parser.<sup>[1]</sup> It takes a string, which is what `usock.read()` returns.
- ❷ Like files, you should close your URL objects as soon as you're done with them.
- ❸ You should close your parser object, too, but for a different reason. You've read all the data and fed it to the parser, but the feed method isn't guaranteed to have actually processed all the HTML you give it; it may buffer it, waiting for more. Be sure to call `close` to flush the buffer and force everything to be fully parsed.
- ❹ Once the parser is closed, the parsing is complete, and `parser.urls` contains a list of all the linked URLs in the HTML document. (Your output may look different, if the download links have been updated by the time you read this.)

## 8.4. Introducing BaseHTMLProcessor.py

SGMLParser doesn't produce anything by itself. It parses and parses and parses, and it calls a method for each interesting thing it finds, but the methods don't do anything. SGMLParser is an HTML *consumer*: it takes HTML and breaks it down into small, structured pieces. As you saw in the previous section, you can subclass SGMLParser to define classes that catch specific tags and produce useful things, like a list of all the links on a web page. Now you'll take this one step further by defining a class that catches everything SGMLParser throws at it and reconstructs the complete HTML document. In technical terms, this class will be an HTML *producer*.

BaseHTMLProcessor subclasses SGMLParser and provides all 8 essential handler methods: `unknown_starttag`, `unknown_endtag`, `handle_charref`, `handle_entityref`, `handle_comment`, `handle_pi`, `handle_decl`, and `handle_data`.

### Example 8.8. Introducing BaseHTMLProcessor

```
class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        self.pieces = []
        SGMLParser.reset(self)

    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())

    def unknown_endtag(self, tag):
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):
        self.pieces.append("&%(ref)s" % locals())
```

```

    if htmlentitydefs.entitydefs.has_key(ref):
        self.pieces.append(";")

def handle_data(self, text):
    self.pieces.append(text)

def handle_comment(self, text):
    self.pieces.append("<!--%(text)s-->" % locals())

def handle_pi(self, text):
    self.pieces.append("<?%(text)s>" % locals())

def handle_decl(self, text):
    self.pieces.append("<!%(text)s>" % locals())

```

- ❶ reset, called by `SGMLParser.__init__`, initializes `self.pieces` as an empty list before calling the ancestor method. `self.pieces` is a data attribute which will hold the pieces of the HTML document you're constructing. Each handler method will reconstruct the HTML that `SGMLParser` parsed, and each method will append that string to `self.pieces`. Note that `self.pieces` is a list. You might be tempted to define it as a string and just keep appending each piece to it. That would work, but Python is much more efficient at dealing with lists.<sup>[2]</sup>
- ❷ Since `BaseHTMLProcessor` does not define any methods for specific tags (like the `start_a` method in `URLLister`), `SGMLParser` will call `unknown_starttag` for every start tag. This method takes the tag (`tag`) and the list of attribute name/value pairs (`attrs`), reconstructs the original HTML, and appends it to `self.pieces`. The string formatting here is a little strange; you'll untangle that (and also the odd-looking `locals` function) later in this chapter.
- ❸ Reconstructing end tags is much simpler; just take the tag name and wrap it in the `</ . . . >` brackets.
- ❹ When `SGMLParser` finds a character reference, it calls `handle_charref` with the bare reference. If the HTML document contains the reference `&#160;`, `ref` will be `160`. Reconstructing the original complete character reference just involves wrapping `ref` in `&# . . . ;` characters.
- ❺ Entity references are similar to character references, but without the hash mark. Reconstructing the original entity reference requires wrapping `ref` in `& . . . ;` characters. (Actually, as an erudite reader pointed out to me, it's slightly more complicated than this. Only certain standard HTML entites end in a semicolon; other similar-looking entities do not. Luckily for us, the set of standard HTML entities is defined in a dictionary in a Python module called `htmlentitydefs`. Hence the extra `if` statement.)
- ❻ Blocks of text are simply appended to `self.pieces` unaltered.
- ❼ HTML comments are wrapped in `<!-- . . . -->` characters.
- ❽ Processing instructions are wrapped in `<? . . . >` characters.

The HTML specification requires that all non-HTML (like client-side JavaScript) must be enclosed in HTML comments, but not all web pages do this properly (and all modern web browsers are forgiving if they don't). `BaseHTMLProcessor` is not forgiving; if script is improperly embedded, it will be parsed as if it were HTML. For instance, if the script contains less-than and equals signs, `SGMLParser` may incorrectly think that it has found tags and attributes. `SGMLParser` always converts tags and attribute names to lowercase, which may break the script, and `BaseHTMLProcessor` always encloses attribute values in double quotes (even if the original HTML document used single quotes or no quotes), which will certainly break the script. Always protect your client-side script within HTML comments.

### Example 8.9. `BaseHTMLProcessor` output

```

def output(self):
    """Return processed HTML as a single string"""
    return "".join(self.pieces)

```

- ❶ This is the one method in `BaseHTMLProcessor` that is never called by the ancestor `SGMLParser`. Since the other handler methods store their reconstructed HTML in `self.pieces`, this function is needed to join all those pieces into one string. As noted before, Python is great at lists and mediocre at strings, so you only create the complete string when somebody explicitly asks for it.
- ❷ If you prefer, you could use the `join` method of the `string` module instead:  
`string.join(self.pieces, " ")`

### Further reading

- W3C (<http://www.w3.org/>) discusses character and entity references (<http://www.w3.org/TR/REC-html40/charset.html#entities>).
- *Python Library Reference* (<http://www.python.org/doc/current/lib/>) confirms your suspicions that the `htmlentitydefs` module (<http://www.python.org/doc/current/lib/module-htmlentitydefs.html>) is exactly what it sounds like.

## 8.5. locals and globals

Let's digress from HTML processing for a minute and talk about how Python handles variables. Python has two built-in functions, `locals` and `globals`, which provide dictionary-based access to local and global variables.

Remember `locals`? You first saw it here:

```
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())
```

No, wait, you can't learn about `locals` yet. First, you need to learn about namespaces. This is dry stuff, but it's important, so pay attention.

Python uses what are called namespaces to keep track of variables. A namespace is just like a dictionary where the keys are names of variables and the dictionary values are the values of those variables. In fact, you can access a namespace as a Python dictionary, as you'll see in a minute.

At any particular point in a Python program, there are several namespaces available. Each function has its own namespace, called the local namespace, which keeps track of the function's variables, including function arguments and locally defined variables. Each module has its own namespace, called the global namespace, which keeps track of the module's variables, including functions, classes, any other imported modules, and module-level variables and constants. And there is the built-in namespace, accessible from any module, which holds built-in functions and exceptions.

When a line of code asks for the value of a variable `x`, Python will search for that variable in all the available namespaces, in order:

1. local namespace – specific to the current function or class method. If the function defines a local variable `x`, or has an argument `x`, Python will use this and stop searching.
2. global namespace – specific to the current module. If the module has defined a variable, function, or class called `x`, Python will use that and stop searching.
3. built-in namespace – global to all modules. As a last resort, Python will assume that `x` is the name of built-in function or variable.

If Python doesn't find `x` in any of these namespaces, it gives up and raises a `NameError` with the message `There is no variable named 'x'`, which you saw back in Example 3.18, `Referencing an Unbound Variable`, but

you didn't appreciate how much work Python was doing before giving you that error.

Python 2.2 introduced a subtle but important change that affects the namespace search order: nested scopes. In versions of Python prior to 2.2, when you reference a variable within a nested function or `lambda` function, Python will search for that variable in the current (nested or `lambda`) function's namespace, then in the module's namespace. Python 2.2 will search for the variable in the current (nested or `lambda`) function's namespace, *then in the parent function's namespace*, then in the module's namespace. Python 2.1 can work either way; by default, it works like Python 2.0, but you can add the following line of code at the top of your module to make your module work like Python 2.2:

```
from __future__ import nested_scopes
```

Are you confused yet? Don't despair! This is really cool, I promise. Like many things in Python, namespaces are *directly accessible at run-time*. How? Well, the local namespace is accessible via the built-in `locals` function, and the global (module level) namespace is accessible via the built-in `globals` function.

### Example 8.10. Introducing `locals`

```
>>> def foo(arg): ❶
...     x = 1
...     print locals()
...
>>> foo(7) ❷
{'arg': 7, 'x': 1}
>>> foo('bar') ❸
{'arg': 'bar', 'x': 1}
```

- ❶ The function `foo` has two variables in its local namespace: `arg`, whose value is passed in to the function, and `x`, which is defined within the function.
- ❷ `locals` returns a dictionary of name/value pairs. The keys of this dictionary are the names of the variables as strings; the values of the dictionary are the actual values of the variables. So calling `foo` with 7 prints the dictionary containing the function's two local variables: `arg` (7) and `x` (1).
- ❸ Remember, Python has dynamic typing, so you could just as easily pass a string in for `arg`; the function (and the call to `locals`) would still work just as well. `locals` works with all variables of all datatypes.

What `locals` does for the local (function) namespace, `globals` does for the global (module) namespace. `globals` is more exciting, though, because a module's namespace is more exciting.<sup>[3]</sup> Not only does the module's namespace include module-level variables and constants, it includes all the functions and classes defined in the module. Plus, it includes anything that was imported into the module.

Remember the difference between `from module import` and `import module`? With `import module`, the module itself is imported, but it retains its own namespace, which is why you need to use the module name to access any of its functions or attributes: `module.function`. But with `from module import`, you're actually importing specific functions and attributes from another module into your own namespace, which is why you access them directly without referencing the original module they came from. With the `globals` function, you can actually see this happen.

### Example 8.11. Introducing `globals`

Look at the following block of code at the bottom of `BaseHTMLProcessor.py`:

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

```
for k, v in globals().items():
    print k, "=", v
```

❶

- ❶ Just so you don't get intimidated, remember that you've seen all this before. The `globals` function returns a dictionary, and you're iterating through the dictionary using the `items` method and multi-variable assignment. The only thing new here is the `globals` function.

Now running the script from the command line gives this output (note that your output may be slightly different, depending on your platform and where you installed Python):

```
c:\docbook\dip\py> python BaseHTMLProcessor.py
```

```
SGMLParser = sgmlib.SGMLParser
htmlentitydefs = <module 'htmlentitydefs' from 'C:\Python23\lib\htmlentitydefs.py'>
BaseHTMLProcessor = __main__.BaseHTMLProcessor
__name__ = __main__
... rest of output omitted for brevity...
```

❶

❷

❸

❹

- ❶ `SGMLParser` was imported from `sgmlib`, using `from module import`. That means that it was imported directly into the module's namespace, and here it is.
- ❷ Contrast this with `htmlentitydefs`, which was imported using `import`. That means that the `htmlentitydefs` module itself is in the namespace, but the `entitydefs` variable defined within `htmlentitydefs` is not.
- ❸ This module only defines one class, `BaseHTMLProcessor`, and here it is. Note that the value here is the class itself, not a specific instance of the class.
- ❹ Remember the `if __name__` trick? When running a module (as opposed to importing it from another module), the built-in `__name__` attribute is a special value, `__main__`. Since you ran this module as a script from the command line, `__name__` is `__main__`, which is why the little test code to print the `globals` got executed.

Using the `locals` and `globals` functions, you can get the value of arbitrary variables dynamically, providing the variable name as a string. This mirrors the functionality of the `getattr` function, which allows you to access arbitrary functions dynamically by providing the function name as a string.

There is one other important difference between the `locals` and `globals` functions, which you should learn now before it bites you. It will bite you anyway, but at least then you'll remember learning it.

### Example 8.12. `locals` is read-only, `globals` is not

```
def foo(arg):
    x = 1
    print locals()
    locals()["x"] = 2
    print "x=", x

z = 7
print "z=", z
foo(3)
globals()["z"] = 8
print "z=", z
```

❶

❷

❸

❹

❺

- ❶ Since `foo` is called with 3, this will print `{'arg': 3, 'x': 1}`. This should not be a surprise.
- ❷ `locals` is a function that returns a dictionary, and here you are setting a value in that dictionary. You might think that this would change the value of the local variable `x` to 2, but it doesn't. `locals` does not actually return the local namespace, it returns a copy. So changing it does nothing to the value of the

variables in the local namespace.

- ❸ This prints `x= 1`, not `x= 2`.
- ❹ After being burned by `locals`, you might think that this *wouldn't* change the value of `z`, but it does. Due to internal differences in how Python is implemented (which I'd rather not go into, since I don't fully understand them myself), `globals` returns the actual global namespace, not a copy: the exact opposite behavior of `locals`. So any changes to the dictionary returned by `globals` directly affect your global variables.
- ❺ This prints `z= 8`, not `z= 7`.

## 8.6. Dictionary–based string formatting

Why did you learn about `locals` and `globals`? So you can learn about dictionary–based string formatting. As you recall, regular string formatting provides an easy way to insert values into strings. Values are listed in a tuple and inserted in order into the string in place of each formatting marker. While this is efficient, it is not always the easiest code to read, especially when multiple values are being inserted. You can't simply scan through the string in one pass and understand what the result will be; you're constantly switching between reading the string and reading the tuple of values.

There is an alternative form of string formatting that uses dictionaries instead of tuples of values.

### Example 8.13. Introducing dictionary–based string formatting

```
>>> params = {"server": "mpilgrim", "database": "master", "uid": "sa", "pwd": "secret"}
>>> "%(pwd)s" % params ❶
'secret'
>>> "%(pwd)s is not a good password for %(uid)s" % params ❷
'secret is not a good password for sa'
>>> "%(database)s of mind, %(database)s of body" % params ❸
'master of mind, master of body'
```

- ❶ Instead of a tuple of explicit values, this form of string formatting uses a dictionary, `params`. And instead of a simple `%s` marker in the string, the marker contains a name in parentheses. This name is used as a key in the `params` dictionary and substitutes the corresponding value, `secret`, in place of the `%(pwd)s` marker.
- ❷ Dictionary–based string formatting works with any number of named keys. Each key must exist in the given dictionary, or the formatting will fail with a `KeyError`.
- ❸ You can even specify the same key twice; each occurrence will be replaced with the same value.

So why would you use dictionary–based string formatting? Well, it does seem like overkill to set up a dictionary of keys and values simply to do string formatting in the next line; it's really most useful when you happen to have a dictionary of meaningful keys and values already. Like `locals`.

### Example 8.14. Dictionary–based string formatting in `BaseHTMLProcessor.py`

```
def handle_comment(self, text):
    self.pieces.append("<!--%(text)s-->" % locals()) ❶
```

- ❶ Using the built-in `locals` function is the most common use of dictionary–based string formatting. It means that you can use the names of local variables within your string (in this case, `text`, which was passed to the class method as an argument) and each named variable will be replaced by its value. If `text` is `'Begin page footer'`, the string formatting `"<!--%(text)s-->" % locals()` will resolve to the string `'<!--Begin page footer-->'`.



### Example 8.15. More dictionary-based string formatting

```
def unknown_starttag(self, tag, attrs):
    strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs]) ❶
    self.pieces.append("<%(tag)s%(strattrs)s>" % locals()) ❷
```

- ❶ When this method is called, `attrs` is a list of key/value tuples, just like the items of a dictionary, which means you can use multi-variable assignment to iterate through it. This should be a familiar pattern by now, but there's a lot going on here, so let's break it down:
- a. Suppose `attrs` is `[('href', 'index.html'), ('title', 'Go to home page')]`.
  - b. In the first round of the list comprehension, `key` will get `'href'`, and `value` will get `'index.html'`.
  - c. The string formatting `' %s="%s"' % (key, value)` will resolve to `' href="index.html" '`. This string becomes the first element of the list comprehension's return value.
  - d. In the second round, `key` will get `'title'`, and `value` will get `'Go to home page'`.
  - e. The string formatting will resolve to `' title="Go to home page" '`.
  - f. The list comprehension returns a list of these two resolved strings, and `strattrs` will join both elements of this list together to form `' href="index.html" title="Go to home page" '`.
- ❷ Now, using dictionary-based string formatting, you insert the value of `tag` and `strattrs` into a string. So if `tag` is `'a'`, the final result would be `'<a href="index.html" title="Go to home page">'`, and that is what gets appended to `self.pieces`.

Using dictionary-based string formatting with `locals` is a convenient way of making complex string formatting expressions more readable, but it comes with a price. There is a slight performance hit in making the call to `locals`, since `locals` builds a copy of the local namespace.

## 8.7. Quoting attribute values

A common question on `comp.lang.python` (<http://groups.google.com/groups?group=comp.lang.python>) is "I have a bunch of HTML documents with unquoted attribute values, and I want to properly quote them all. How can I do this?"<sup>[4]</sup> (This is generally precipitated by a project manager who has found the HTML-is-a-standard religion joining a large project and proclaiming that all pages must validate against an HTML validator. Unquoted attribute values are a common violation of the HTML standard.) Whatever the reason, unquoted attribute values are easy to fix by feeding HTML through `BaseHTMLProcessor`.

`BaseHTMLProcessor` consumes HTML (since it's descended from `SGMLParser`) and produces equivalent HTML, but the HTML output is not identical to the input. Tags and attribute names will end up in lowercase, even if they started in uppercase or mixed case, and attribute values will be enclosed in double quotes, even if they started in single quotes or with no quotes at all. It is this last side effect that you can take advantage of.

### Example 8.16. Quoting attribute values

```
>>> htmlSource = """ ❶
...     <html>
...     <head>
...     <title>Test page</title>
...     </head>
...     <body>
...     <ul>
...     <li><a href=index.html>Home</a></li>
...     <li><a href=toc.html>Table of contents</a></li>
...     <li><a href=history.html>Revision history</a></li>
```

```

...     </body>
...     </html>
...     """
>>> from BaseHTMLProcessor import BaseHTMLProcessor
>>> parser = BaseHTMLProcessor()
>>> parser.feed(htmlSource) ❷
>>> print parser.output() ❸
<html>
<head>
<title>Test page</title>
</head>
<body>
<ul>
<li><a href="index.html">Home</a></li>
<li><a href="toc.html">Table of contents</a></li>
<li><a href="history.html">Revision history</a></li>
</body>
</html>

```

- ❶ Note that the attribute values of the href attributes in the <a> tags are not properly quoted. (Also note that you're using triple quotes for something other than a doc string. And directly in the IDE, no less. They're very useful.)
- ❷ Feed the parser.
- ❸ Using the output function defined in BaseHTMLProcessor, you get the output as a single string, complete with quoted attribute values. While this may seem anti-climactic, think about how much has actually happened here: SGMLParser parsed the entire HTML document, breaking it down into tags, refs, data, and so forth; BaseHTMLProcessor used those elements to reconstruct pieces of HTML (which are still stored in parser.pieces, if you want to see them); finally, you called parser.output, which joined all the pieces of HTML into one string.

## 8.8. Introducing dialect.py

Dialectizer is a simple (and silly) descendant of BaseHTMLProcessor. It runs blocks of text through a series of substitutions, but it makes sure that anything within a <pre> . . . </pre> block passes through unaltered.

To handle the <pre> blocks, you define two methods in Dialectizer: start\_pre and end\_pre.

### Example 8.17. Handling specific tags

```

def start_pre(self, attrs): ❶
    self.verbatim += 1 ❷
    self.unknown_starttag("pre", attrs) ❸

def end_pre(self): ❹
    self.unknown_endtag("pre") ❺
    self.verbatim -= 1 ❻

```

- ❶ start\_pre is called every time SGMLParser finds a <pre> tag in the HTML source. (In a minute, you'll see exactly how this happens.) The method takes a single parameter, attrs, which contains the attributes of the tag (if any). attrs is a list of key/value tuples, just like unknown\_starttag takes.
- ❷ In the reset method, you initialize a data attribute that serves as a counter for <pre> tags. Every time you hit a <pre> tag, you increment the counter; every time you hit a </pre> tag, you'll decrement the counter. (You could just use this as a flag and set it to 1 and reset it to 0, but it's just as easy to do it this way, and this handles the odd (but possible) case of nested <pre> tags.) In a minute, you'll see how this counter is put to good use.
- ❸

That's it, that's the only special processing you do for `<pre>` tags. Now you pass the list of attributes along to `unknown_starttag` so it can do the default processing.

- ④ `end_pre` is called every time `SGMLParser` finds a `</pre>` tag. Since end tags can not contain attributes, the method takes no parameters.
- ⑤ First, you want to do the default processing, just like any other end tag.
- ⑥ Second, you decrement your counter to signal that this `<pre>` block has been closed.

At this point, it's worth digging a little further into `SGMLParser`. I've claimed repeatedly (and you've taken it on faith so far) that `SGMLParser` looks for and calls specific methods for each tag, if they exist. For instance, you just saw the definition of `start_pre` and `end_pre` to handle `<pre>` and `</pre>`. But how does this happen? Well, it's not magic, it's just good Python coding.

### Example 8.18. `SGMLParser`

```
def finish_starttag(self, tag, attrs):  
    try:  
        method = getattr(self, 'start_' + tag)  
    except AttributeError:  
        try:  
            method = getattr(self, 'do_' + tag)  
        except AttributeError:  
            self.unknown_starttag(tag, attrs)  
            return -1  
        else:  
            self.handle_starttag(tag, method, attrs)  
            return 0  
    else:  
        self.stack.append(tag)  
        self.handle_starttag(tag, method, attrs)  
        return 1  
  
def handle_starttag(self, tag, method, attrs):  
    method(attrs)
```

- ① At this point, `SGMLParser` has already found a start tag and parsed the attribute list. The only thing left to do is figure out whether there is a specific handler method for this tag, or whether you should fall back on the default method (`unknown_starttag`).
- ② The "magic" of `SGMLParser` is nothing more than your old friend, `getattr`. What you may not have realized before is that `getattr` will find methods defined in descendants of an object as well as the object itself. Here the object is `self`, the current instance. So if `tag` is `'pre'`, this call to `getattr` will look for a `start_pre` method on the current instance, which is an instance of the `Dialectizer` class.
- ③ `getattr` raises an `AttributeError` if the method it's looking for doesn't exist in the object (or any of its descendants), but that's okay, because you wrapped the call to `getattr` inside a `try...except` block and explicitly caught the `AttributeError`.
- ④ Since you didn't find a `start_xxx` method, you'll also look for a `do_xxx` method before giving up. This alternate naming scheme is generally used for standalone tags, like `<br>`, which have no corresponding end tag. But you can use either naming scheme; as you can see, `SGMLParser` tries both for every tag. (You shouldn't define both a `start_xxx` and `do_xxx` handler method for the same tag, though; only the `start_xxx` method will get called.)
- ⑤ Another `AttributeError`, which means that the call to `getattr` failed with `do_xxx`. Since you found neither a `start_xxx` nor a `do_xxx` method for this tag, you catch the

exception and fall back on the default method, `unknown_starttag`.

- ❸ Remember, `try...except` blocks can have an `else` clause, which is called if no exception is raised during the `try...except` block. Logically, that means that you *did* find a `do_xxx` method for this tag, so you're going to call it.
- ❹ By the way, don't worry about these different return values; in theory they mean something, but they're never actually used. Don't worry about the `self.stack.append(tag)` either; `SGMLParser` keeps track internally of whether your start tags are balanced by appropriate end tags, but it doesn't do anything with this information either. In theory, you could use this module to validate that your tags were fully balanced, but it's probably not worth it, and it's beyond the scope of this chapter. You have better things to worry about right now.
- ❺ `start_xxx` and `do_xxx` methods are not called directly; the tag, method, and attributes are passed to this function, `handle_starttag`, so that descendants can override it and change the way *all* start tags are dispatched. You don't need that level of control, so you just let this method do its thing, which is to call the method (`start_xxx` or `do_xxx`) with the list of attributes. Remember, method is a function, returned from `getattr`, and functions are objects. (I know you're getting tired of hearing it, and I promise I'll stop saying it as soon as I run out of ways to use it to my advantage.) Here, the function object is passed into this dispatch method as an argument, and this method turns around and calls the function. At this point, you don't need to know what the function is, what it's named, or where it's defined; the only thing you need to know about the function is that it is called with one argument, `attrs`.

Now back to our regularly scheduled program: `Dialectizer`. When you left, you were in the process of defining specific handler methods for `<pre>` and `</pre>` tags. There's only one thing left to do, and that is to process text blocks with the pre-defined substitutions. For that, you need to override the `handle_data` method.

### Example 8.19. Overriding the `handle_data` method

```
def handle_data(self, text):  
    self.pieces.append(self.verbatim and text or self.process(text))
```

- ❶ `handle_data` is called with only one argument, the text to process.
- ❷ In the ancestor `BaseHTMLProcessor`, the `handle_data` method simply appended the text to the output buffer, `self.pieces`. Here the logic is only slightly more complicated. If you're in the middle of a `<pre>...</pre>` block, `self.verbatim` will be some value greater than 0, and you want to put the text in the output buffer unaltered. Otherwise, you will call a separate method to process the substitutions, then put the result of that into the output buffer. In Python, this is a one-liner, using the `and-or` trick.

You're close to completely understanding `Dialectizer`. The only missing link is the nature of the text substitutions themselves. If you know any Perl, you know that when complex text substitutions are required, the only real solution is regular expressions. The classes later in `dialect.py` define a series of regular expressions that operate on the text between the HTML tags. But you just had a whole chapter on regular expressions. You don't really want to slog through regular expressions again, do you? God knows I don't. I think you've learned enough for one chapter.

## 8.9. Putting it all together

It's time to put everything you've learned so far to good use. I hope you were paying attention.

### Example 8.20. The `translate` function, part 1

```
def translate(url, dialectName="chef"):  
    import urllib
```

```

sock = urllib.urlopen(url)
htmlSource = sock.read()
sock.close()

```

③

- ① The `translate` function has an optional argument `dialectName`, which is a string that specifies the dialect you'll be using. You'll see how this is used in a minute.
- ② Hey, wait a minute, there's an `import` statement in this function! That's perfectly legal in Python. You're used to seeing `import` statements at the top of a program, which means that the imported module is available anywhere in the program. But you can also import modules within a function, which means that the imported module is only available within the function. If you have a module that is only ever used in one function, this is an easy way to make your code more modular. (When you find that your weekend hack has turned into an 800-line work of art and decide to split it up into a dozen reusable modules, you'll appreciate this.)
- ③ Now you get the source of the given URL.

### Example 8.21. The `translate` function, part 2: `curiouser` and `curiouser`

```

parserName = "%sDialectizer" % dialectName.capitalize()
parserClass = globals()[parserName]
parser = parserClass()

```

①  
②  
③

- ① `capitalize` is a string method you haven't seen before; it simply capitalizes the first letter of a string and forces everything else to lowercase. Combined with some string formatting, you've taken the name of a dialect and transformed it into the name of the corresponding `Dialectizer` class. If `dialectName` is the string `'chef'`, `parserName` will be the string `'ChefDialectizer'`.
- ② You have the name of a class as a string (`parserName`), and you have the global namespace as a dictionary (`globals()`). Combined, you can get a reference to the class which the string names. (Remember, classes are objects, and they can be assigned to variables just like any other object.) If `parserName` is the string `'ChefDialectizer'`, `parserClass` will be the class `ChefDialectizer`.
- ③ Finally, you have a class object (`parserClass`), and you want an instance of the class. Well, you already know how to do that: call the class like a function. The fact that the class is being stored in a local variable makes absolutely no difference; you just call the local variable like a function, and out pops an instance of the class. If `parserClass` is the class `ChefDialectizer`, `parser` will be an instance of the class `ChefDialectizer`.

Why bother? After all, there are only 3 `Dialectizer` classes; why not just use a `case` statement? (Well, there's no `case` statement in Python, but why not just use a series of `if` statements?) One reason: extensibility. The `translate` function has absolutely no idea how many `Dialectizer` classes you've defined. Imagine if you defined a new `FooDialectizer` tomorrow; `translate` would work by passing `'foo'` as the `dialectName`.

Even better, imagine putting `FooDialectizer` in a separate module, and importing it with `from module import`. You've already seen that this includes it in `globals()`, so `translate` would still work without modification, even though `FooDialectizer` was in a separate file.

Now imagine that the name of the dialect is coming from somewhere outside the program, maybe from a database or from a user-inputted value on a form. You can use any number of server-side Python scripting architectures to dynamically generate web pages; this function could take a URL and a dialect name (both strings) in the query string of a web page request, and output the "translated" web page.

Finally, imagine a `Dialectizer` framework with a plug-in architecture. You could put each `Dialectizer` class in a separate file, leaving only the `translate` function in `dialect.py`. Assuming a consistent naming scheme, the `translate` function could dynamic import the appropriate class from the appropriate file, given nothing but the dialect name. (You haven't seen dynamic importing yet, but I promise to cover it in a later chapter.) To add a new

dialect, you would simply add an appropriately-named file in the plug-ins directory (like `foodialect.py` which contains the `FoodDialectizer` class). Calling the `translate` function with the dialect name `'foo'` would find the module `foodialect.py`, import the class `FoodDialectizer`, and away you go.

### Example 8.22. The `translate` function, part 3

```
parser.feed(htmlSource) ❶  
parser.close()           ❷  
return parser.output()   ❸
```

- ❶ After all that imagining, this is going to seem pretty boring, but the `feed` function is what does the entire transformation. You had the entire HTML source in a single string, so you only had to call `feed` once. However, you can call `feed` as often as you want, and the parser will just keep parsing. So if you were worried about memory usage (or you knew you were going to be dealing with very large HTML pages), you could set this up in a loop, where you read a few bytes of HTML and fed it to the parser. The result would be the same.
- ❷ Because `feed` maintains an internal buffer, you should always call the parser's `close` method when you're done (even if you fed it all at once, like you did). Otherwise you may find that your output is missing the last few bytes.
- ❸ Remember, `output` is the function you defined on `BaseHTMLProcessor` that joins all the pieces of output you've buffered and returns them in a single string.

And just like that, you've "translated" a web page, given nothing but a URL and the name of a dialect.

### Further reading

- You thought I was kidding about the server-side scripting idea. So did I, until I found this web-based dialectizer (<http://rinkworks.com/dialect/>). Unfortunately, source code does not appear to be available.

## 8.10. Summary

Python provides you with a powerful tool, `sgmllib.py`, to manipulate HTML by turning its structure into an object model. You can use this tool in many different ways.

- parsing the HTML looking for something specific
- aggregating the results, like the URL lister
- altering the structure along the way, like the attribute quoter
- transforming the HTML into something else by manipulating the text while leaving the tags alone, like the `Dialectizer`

Along with these examples, you should be comfortable doing all of the following things:

- Using `locals()` and `globals()` to access namespaces
- Formatting strings using dictionary-based substitutions

---

<sup>[1]</sup> The technical term for a parser like `SGMLParser` is a *consumer*: it consumes HTML and breaks it down. Presumably, the name `feed` was chosen to fit into the whole "consumer" motif. Personally, it makes me think of an exhibit in the zoo where there's just a dark cage with no trees or plants or evidence of life of any kind, but if you stand perfectly still and look really closely you can make out two beady eyes staring back at you from the far left corner, but you convince yourself that that's just your mind playing tricks on you, and the only way you can tell that the whole thing isn't just an empty cage is a small innocuous sign on the railing that reads, "Do not feed the parser." But maybe

that's just me. In any event, it's an interesting mental image.

<sup>[2]</sup> The reason Python is better at lists than strings is that lists are mutable but strings are immutable. This means that appending to a list just adds the element and updates the index. Since strings can not be changed after they are created, code like `s = s + newpiece` will create an entirely new string out of the concatenation of the original and the new piece, then throw away the original string. This involves a lot of expensive memory management, and the amount of effort involved increases as the string gets longer, so doing `s = s + newpiece` in a loop is deadly. In technical terms, appending  $n$  items to a list is  $O(n)$ , while appending  $n$  items to a string is  $O(n^2)$ .

<sup>[3]</sup> I don't get out much.

<sup>[4]</sup> All right, it's not that common a question. It's not up there with "What editor should I use to write Python code?" (answer: Emacs) or "Is Python better or worse than Perl?" (answer: "Perl is worse than Python because people wanted it worse." –Larry Wall, 10/14/1998) But questions about HTML processing pop up in one form or another about once a month, and among those questions, this is a popular one.