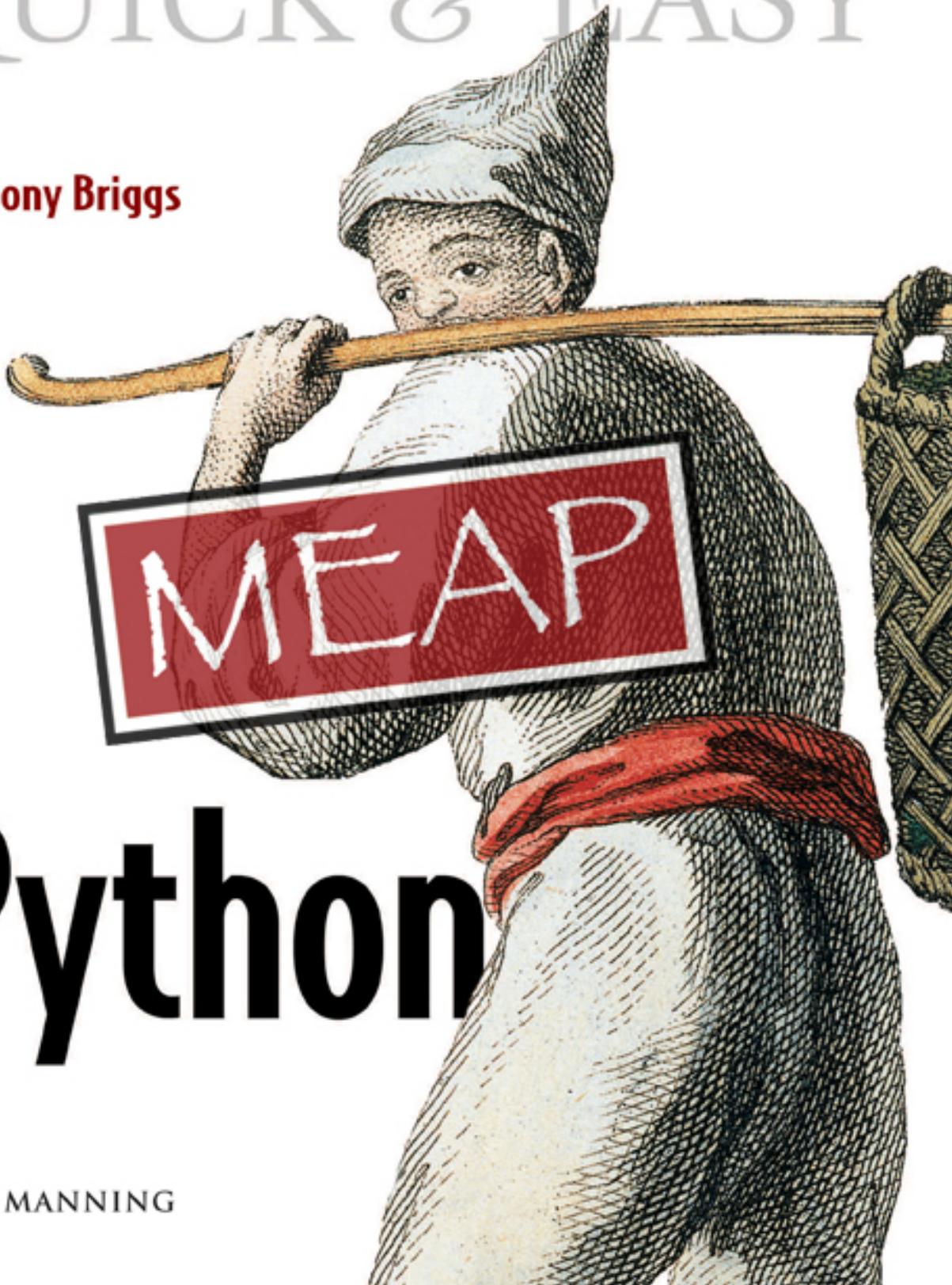


QUICK & EASY

Anthony Briggs

Python



MANNING



**MEAP Edition
Manning Early Access Program
Quick & Easy Python Final Version**

Copyright 2011 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Table of Contents

- 1. Why Python?**
- 2. Hunt the Wumpus**
- 3. Interacting with the world**
- 4. Getting organized**
- 5. Business-oriented programming**
- 6. Classes and object-oriented programming**
- 7. Sufficiently advanced technology**
- 8. Django!**
- 9. Gaming with Pyglet**
- 10. Twisted networking**
- 11. Django revisited**
- 12. Where to from here?**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Preface

When I was first asked to write Quick & Easy Python, I didn't want to write just another introductory book - I wanted to write something different. The programming books that I've read in the past, have often been just a laundry list of features: a list can have things in it, you can call `len(mylist)` to find out exactly how many things, `.pop()` to chop an element from the end, `.append()` to add... There you go, that's all you need to know about lists, now on to the next feature. If there is a case where you're shown a program, it's usually either a trivial few lines, or a couple of chapters tacked on to the end of the book as an afterthought.

Then I thought back to how I first learned to program. I didn't read an entire programming book from cover to cover and then get stuck in once I knew everything there was to know. Instead I started with a goal, something that I wanted to do, and worked towards it, figuring things out as I went. I read programming books from time to time, but really only to figure out the bits that I was stuck on. When I was done, my programs weren't always particularly elegant or fast, but they were mine - I knew how they worked, and they solved a real problem that I was having.

Fast forward to today, and my programs *are* elegant and fast, for the most part. And most of the really good programmers that I know have also learned to program in the same way. In Quick and Easy Python, I've tried to recreate that process, but sped up, with all of the things that I've learned about programming and the pitfalls that I've encountered. Every chapter, (except for the first and last) has a practical program at its core to illustrate either a particular Python feature or library - often several. Some of them are fun, some of them are useful, but there are no boring initial chapters where you learn, in excruciating detail, every feature of a list or dictionary. Or worse, learn how Python adds numbers together.

Instead, you'll watch a program being written and learn about Python features as you need them, not before. Several of the chapters build on previous ones, so you'll learn how to extend existing programs to add new features and keep their design under control - essential if you're going to be writing programs of any scope. There are also several different styles of program, from simple scripts, to object oriented programs, to event-based games.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

The whole idea is to provide something different, where you can start writing programs from the very first chapter and really understand how to use Python's features - by seeing them used in action. I hope this is the sort of book which will help people to really understand how to use Python.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

About this book

Quick & Easy Python is written for people who'd like to learn more about Python and how to program. You might be completely new to programming, or you might have some prior experience, but either way Quick & Easy Python will take you from your very first steps through to writing networked games and web applications.

The style of this book is a bit different to most programming books. Rather than take you through a laundry list of every possible feature, I've chosen to show a more "Real World" picture. Starting with chapter 2, you'll be following along as we write real, useful programs - warts and all. All programming language features have a purpose, and it's hard to see that purpose if don't see all of the bugs, broken code, and badly written programs that the feature is supposed to help with.

Some of the programs in Quick & Easy Python are improved and expanded as the book progresses, so you'll see how Python features such as functions, classes and modules can help keep your code under control as it expands. They'll also reduce the amount of work that you have to do when new parts need to be added.

I think of Quick & Easy Python as being split into three rough sections, although that's not explicitly mentioned in the book. The first chapters cover the basic syntax of Python, how to use libraries, some common concepts and all the other pieces that you'll need to know to understand how things work. The middle section covers some more advanced features and introduces common libraries which will help you get more done without having to reinvent the wheel. In the final section we write complete programs using frameworks, which will help you even more.

The fun doesn't stop when you've finished the book, either. All of the programs in Quick & Python are intended to be extended and reused when you write your own programs. Most experienced programmers tend to have a library of code that they've previously written, and the code in this book will give you a head start on your own projects.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Acknowledgements

Firstly I'd like to thank Lyndall, my beautiful wife, for being supportive and giving me the time that I needed to write this book. It took much longer than we originally thought, but her enthusiasm was unwavering, despite the many weekends that I spent cloistered in the study.

Secondly I'd like to thank the team at Manning: My editor Sebastian Stirling for his suggestions and experience, Karen Tegtmeyer for organizing the reviews, and Michael Stephens and Marjan Bace for helping me to develop the initial concept of the book.

Finally I'd like to thank all of my beta testers who helped to find errors - Daniel Hadson, Eldar Marcussen, William Taylor, David Hepworth and Tony Haig, as well as everyone in the MEAP program who offered advice and criticism or discovered errors.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

1

Why Python?

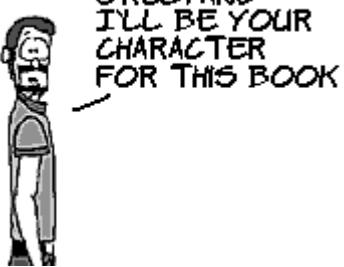
If you've picked up this book, you're probably trying to learn how to program. Congratulations! Not very many people do, but it's one of the most interesting and rewarding subjects that you can learn. Programming is the new literacy - if you're not sure how to write a simple program, whether as a batch file, mail filter or formula in a spreadsheet, you're at a disadvantage compared to those who do. Programming is a lever - using it, you can turn your ideas into reality, often far more effectively than doing it yourself, since computers don't get bored.

I first started to program when I was around 10, on the Commodore 64. Back then there wasn't much in the way of preprogrammed software unless you counted games or simple word processing. Computers like the Commodore came with BASIC built in, and programming was a lot more accessible - you didn't need to learn a great deal to be able to get results quickly.

Since then, computers have moved away from that early ideal. Now you have to go out of your way to install something so that your computer can be programmed. But it's still possible, and once you know how, you can create all sorts of wondrous programs which will

do boring work for you, inform you and entertain you. Most of all that last part - programming is fun, and everybody should try it.

You might notice some cartoons sprinkled through the book. The idea is to use them to give you some background information on what's going on in the chapter, or to cover some common problems while having a bit of fun. While the characters are from User Friendly, the text and jokes are all mine, so if you don't like them you know who to blame.



Let's start by learning the basics of programming.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

1.1 Learning to program

Since this book is about programming, it makes sense to give you some sort of overview before we jump in and start learning the details in chapter 2. What is programming? How does it work? The definition of programming is very easy.

Definition Programming is telling a computer what to do.

Like most definitions though, it's a drastic oversimplification, and there are all sorts of interesting bits which are lost in translation. Much like chess, learning the initial rules of programming is very easy, but putting them together in a useful way and mastering them is much harder. Programming touches on most areas of human endeavor these days, and it's just as much about design and ideas and personal expression as it is about numbers and calculation.

PROGRAMMING IS
ART, MAN.
THAT'S ALL YOU
NEED TO KNOW



1.1.1 Telling a computer what to do

Let's break down the different parts of our definition and look at them individually. In order to understand our definition, we need to know what a computer is, what we mean by 'telling' and what 'what to do' means.

A COMPUTER

A computer is a very fast calculator that can make simple decisions based on instructions that you give it. Computer instructions are very simple, not much more than adding numbers together and comparing things, but they can be combined to make much larger programs which can do complex things, like writing documents, playing games, balancing your accounts and controlling nuclear reactors.

Computers can seem smart, but they're actually very stupid and single minded, with no common sense. At their heart they're still just machines; they'll do exactly what you (or the developers of Python) tell them to do - whatever the consequences. Delete the whole hard drive? That's a bit drastic - most people would probably check to make sure that's what you wanted, but a computer will just go right ahead and destroy all your data.

Note: The great thing about computers is that they do exactly what you tell them. The terrible thing about computers is that they do exactly what you tell them.

So if a program that you're using (or that you've written) is doing something odd or crashes for no reason, it's nothing personal - it's just following the instructions that it was given.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

TELLING

When working with Python, you'll typically instruct it by typing 'program code' into a text file and then telling the Python program to run it. You'll find out how to do that later on in this chapter. The instructions that you type can be very complex, or very simple, and they cover a wide range of tasks - adding numbers, opening other files, placing things on screen and so on. A simple Python program looks like this:

```
number = "42"
print "Guess my number..."
guess = raw_input(">")
if guess == number:
    print "Yes! that's it!"
else:
    print "No - it's", number

raw_input("hit enter to continue")
```

Don't worry too much about trying to understand this program just yet. It's just to give you some background.

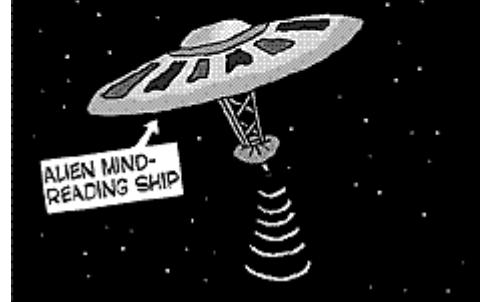
WHAT TO DO

This is where the fun starts. Most modern computers are 'Turing Complete', which means that they can do anything that's possible. Anything that you can think of, a computer can do. At least in theory - it might take longer or be more complicated than you first expected, or need special hardware if you want to interact in a certain way, but if the computer has access to enough data and you've programmed it properly, the sky's the limit. Here are some of the tasks that computers have been used for:

- Controlling manned and unmanned spacecraft and probes and guiding robots on other planets such as the Mars exploration rovers Spirit and Opportunity.
- Transmitting data around the world via a network of computers - the Internet and World Wide Web! You can transmit or receive information from around the world in just a fraction of a second.
- Building robots, from industrial robot arms, to roomba vacuum cleaners to lifelike human robots that can climb stairs or mimic human emotions.
- Modeling real world processes such as gravity, light or weather. This includes scientific models, but also most games.

MEANWHILE, IN ORBIT ABOVE OUR PLANET...

```
#!/usr/bin/python
import deathraylaser
import mindcontrol
from probes import RectalProbe
mindcontrol.scan(evil=True)
```



You might not have the hardware that's needed to be able to send a robot probe to another planet, but in principle you can still run the same programs. The computers used to drive Spirit and Opportunity, for example, are much less powerful than the computer sitting under your desk.

1.1.2 Programming is made of ideas

It's easy to focus on the concrete aspects of computer programming - instructions, adding numbers, networks, hardware and so on - but the core of programming is about ideas, and capturing those ideas in a program so that other people can use them. Helping other people by discovering new, cool things is something that's been happening since early man started using pointy sticks, and programming is no exception. Computers have helped to develop many new ideas since their invention, including the internet, spreadsheets, interactive games and desktop publishing.

TO BECOME A TRUE
PROGRAMMER YOU MUST
BECOME ONE WITH THE TAO
OF THE COMPUTER, AND
SWIM IN YOUR IDEAS AS A
A FISH SWIMS IN THE STREAM



Unfortunately I can't help you come up with new ideas, but I can show you some of the ideas that other people have come up with as inspiration to develop some of your own.

1.1.3 Programming is design

Most of the aspects of programming that we'll deal with in this book deal with design. Design is typically described as a common solution to a particular problem. For example, Architecture is the design of buildings and the space that they occupy - it addresses some of the problems common to buildings, such as how people get in and out and move around inside a building, how they occupy it, how to make people happy about being in a building, using materials sensibly and so on.

What makes a design good, or better than another, is whether it solves your problems well. This means that a design is never complete; there are always other, potentially better ways to solve a problem. Other factors that you can consider are whether the solution is accurate, or only solves part of your problem, and how easy the design is to put into practice. If a design solves a problem so that it's 10% better in some way, but is twice as hard to put into practice, then you might go with the simpler one.

So if programming is the design of ideas, what are some of the problems that it solves? Well the key thing that programs need to be able to do is express the ideas that we talked about in the previous section. Some of the problems that you're likely to run into:

- Your idea isn't fully formed - there are normally details that need to be worked out.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

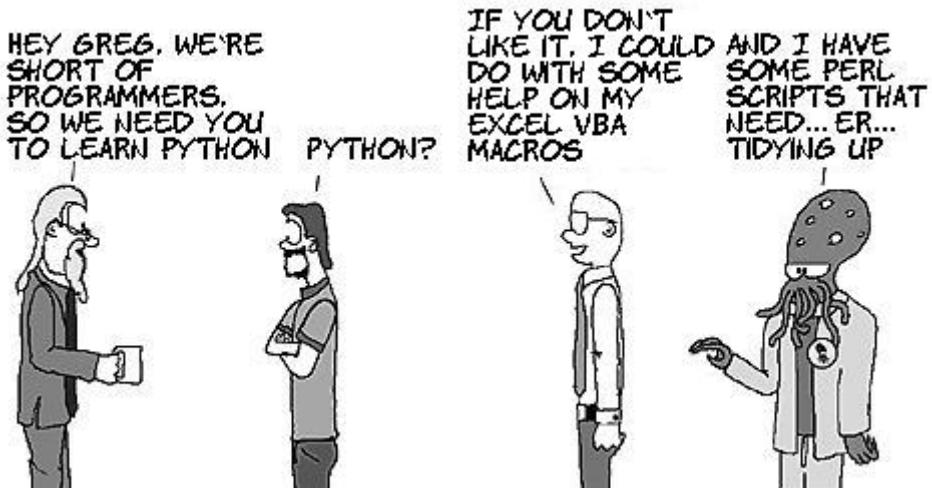
- Most ideas are complicated, and have a lot of details involved once you start writing them down.
- Your ideas need to be clear and easy to follow, so that other people can use them, understand them and build on them.

One of the common themes in the development of computer languages is the management of complexity. Even when working on quite straightforward programs, it's easy to get bogged down in details and lose sight of what you're trying to do. When it comes time to make changes to a program, you can misunderstand the original purpose of the program and introduce errors or inconsistencies. A good programming language will have features to help you work at different levels of detail, moving to more (or less) detailed levels as necessary.

Another important factor is how flexible your programs are when written in a particular language. Exploratory programming is a very useful tool when developing ideas, and we'll be doing a lot of it in this book, but if your programming language doesn't have strong facilities for managing complexity or hiding detail, then they become hard to change and a lot of the benefit is lost.

Now that you have a basic understanding of programming, it's time to check out this book's chosen language, Python.

1.2 *What makes Python so great?*



In this book, you'll be learning Python, which not-so-coincidentally, happens to be my favorite programming language. It's ideal for a beginner to get started, for a number of reasons.

1.2.1 Python is easy

The first thing that you'll notice if you compare Python to other programming languages is that it's very easy to read. Python's syntax is intended to be as clear as possible:

- it avoids the use of punctuation characters like { } \$ / and \
- Python uses whitespace to indent lines for program control, instead of using brackets
- programmers are encouraged to make their programs clear and easy to read
- Python supports a number of different ways to structure your programs, so you can pick the best one for the job

Python's developers try to do things 'right', by making programming as straightforward as possible. There have been several cases where features have been delayed (or even cancelled outright) while the core developers figured out the best way to present a particular feature. Python even has its own philosophy on how programs should look and behave. Try typing 'import this' once you have Python installed later on in the chapter.

1.2.2 Python is a real language

Although Python is an easy to use language, it's also a 'real' language. Typically languages come in two flavors - easy ones with training wheels, to teach people how to program, and harder ones with more features to let you get real work done. When you're learning how to program, you have two choices:

- Jump head first into a 'real' language, but be prepared to be very confused until you figure out the hard language.
- Start with a beginner's language, but be prepared to throw away all of the work that you've done when you need a feature that it doesn't have.

Python manages to combine the best of both of these worlds. It's easy to use and learn, but as your programming skills grow, you'll be able to continue using Python, since it's fast and has lots of useful features. Best of all, jumping in and learning how to do things the 'real' way is often easier than following all of the steps that you need to learn how to program 'properly'.

1.2.3 Python has "batteries included"

There are a large number of libraries that come included with Python, and there are many more which you can download and install. Libraries are program code that other programmers have written which you can easily reuse. They'll let you read files, process data, connect to other computers via the internet, serve web pages, generate random numbers, and pretty much any other sort of basic activity. Python is a good choice for:

- web development

MMM, GAMES...



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

- networking
- graphical interfaces
- scripting operating system tasks
- games
- data processing
- business applications

Often, when it comes time to write a program, most of the hard bits are already done for you, and all you have to do is join together a few libraries to be able to do what you need. You'll read more about Python's libraries and how to use them in chapter 3.

1.2.4 Python has a large community

Python is a popular language, and has a large, friendly community which is happy to help out new Python developers. Questions are always welcome on the main mailing list, and there is also a specialized mailing list set up specifically to help new developers. There are also a lot of introductions, tutorials and a great deal of example code available on the internet.

"Good artists borrow, Great artists steal."

With the size of the Python developer community, there are a lot of programs to beg, borrow and steal, regardless of what type of program you're writing. Once you have some Python experience, reading other people's programs is an excellent way to learn more.

One of the other advantages of having a large community is that Python gets a lot of active development, so bugs are fixed very rapidly and new features are added. Python is going from strength to strength.

Now that you know about programming, and why Python is a good choice, let's install Python on your computer so that you can run your own programs. If you're running Linux, skip ahead a section. If you're running Mac, skip ahead two sections.

1.3 Setting up Python for Windows

Over the next couple of sections we'll go through the installation process step by step, create a simple program to make sure that Python is working on your system and teach you the basic steps involved in running a program. Making sure that Python is working properly now will save you a lot of frustration later on. If you're running Linux or Mac, I'll

1.3.1 Installing Python

We'll be using the latest version of Python 2, since most of the libraries that we'll use in this book don't yet support Python 3. At the time of writing, Python 2.6 is the standard version, but Python 2.7 should be available by the time you read this. To install Python, we need to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

download a program from the Python website and run it. That program includes Python, its libraries and everything you need to run Python programs.

The first step is to go to <http://python.org/> and click on downloads. That should take you to a page listing all of the operating systems that Python can be installed on. Click on the Windows version and save it to your desktop.



Figure 1.1 Python.org's download page

Once it's finished downloading, double click on the program's icon to open and run it. You'll probably be shown a screen like figure 1.2. Click on 'run' to run the Python installer.



Figure 1.2 Are you sure you want to run this strange program from the internet? Yes!

You'll now be given a series of options for installing Python. Typically the defaults (the options that have already been chosen for you) are good enough, unless your computer is low on disk space and needs to install to a different partition. If you're happy with the options at each step, just hit 'Next>' to go to the next screen.



Figure 1.3 Install Python for all users

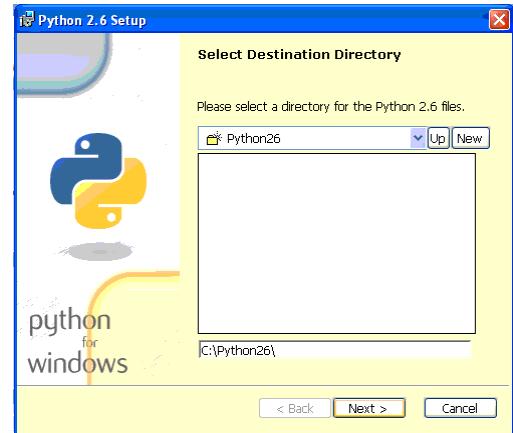


Figure 1.4 Choose Python's location

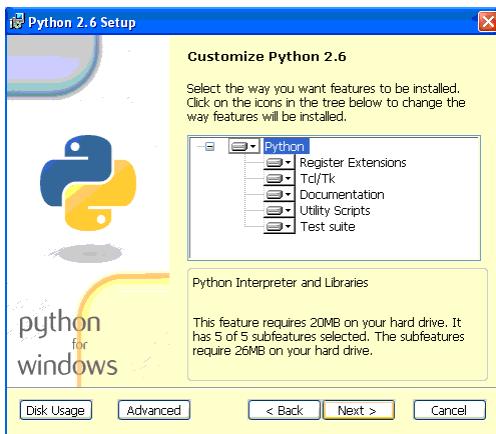


Figure 1.5 Choose which bits of Python you want

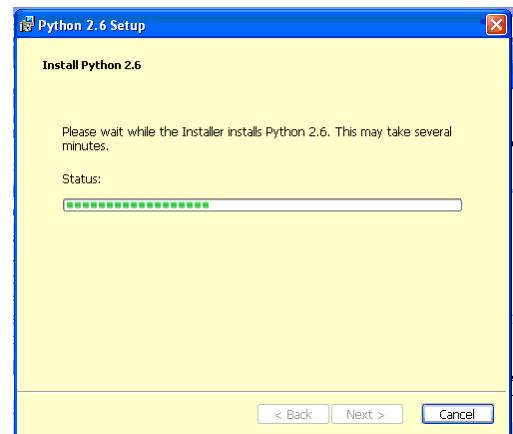


Figure 1.6 Installing Python

The final stage might take a little while depending on the speed of your computer, but once you see the next figure, you're done.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>



Figure 1.7 Hooray! Python's installed!

NEXT...
NEXT...
NEXT...
NEXT...
AM I MISSING
SOMETHING?
THIS IS
SUSPICIOUSLY
EASY...



Congratulations! You've installed Python!

1.3.2 *Running Python programs on Windows*

Now that you have Python installed on your system, let's create a simple program. This will let you know that Python is installed correctly, and also show you how to create a program and run it.

Python programs are normally written into a text file, then run by the Python interpreter. To start with we'll use Notepad to create our file (although if you already have a favorite text editor you can use that). Avoid using Microsoft Word or Wordpad to create your programs - they insert extra characters for formatting which Python won't understand. Notepad is in the 'Programs > Accessories' section of your start menu.

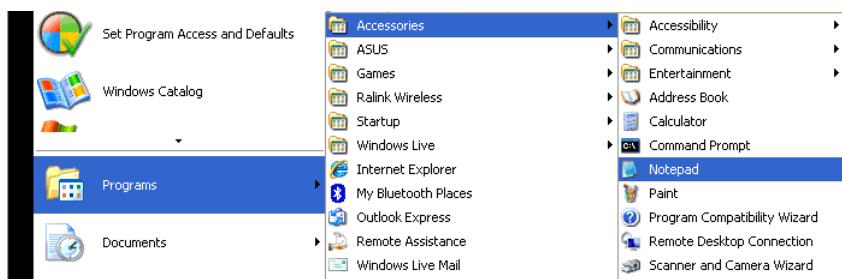


Figure 1.8 Here's where Notepad lives

In the notepad window that opens, type the following code. Don't worry too much about what it does just yet - for now we just want to test out Python and make sure that we can run a program.

```
print "Hello World!"  
raw_input("Hit enter to continue")
```

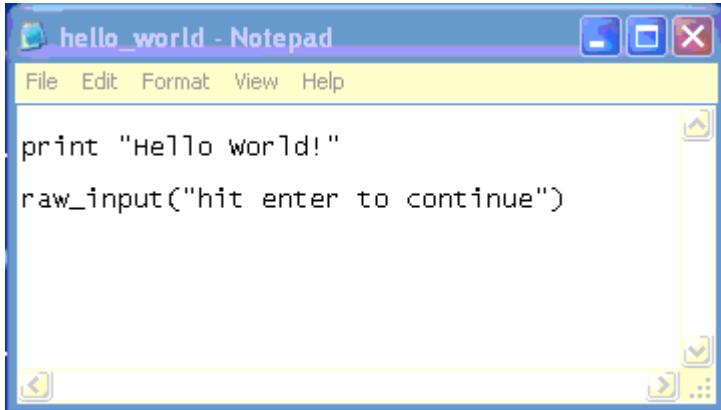
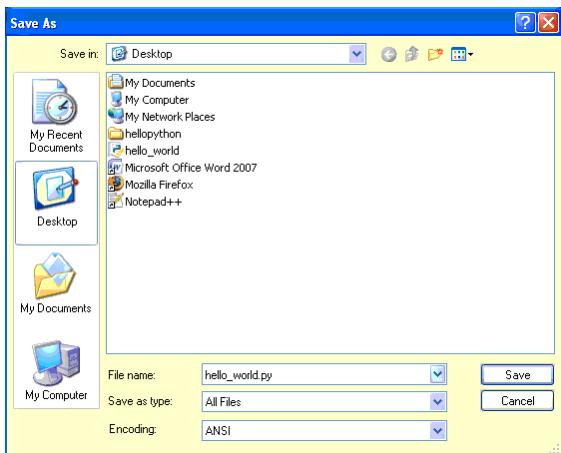


Figure 1.9 The test program for Python

Once you're done, save it to your desktop as `hello_world.py`. The `.py` on the end is important - that's how Windows knows that it's a Python program.



AH, I REMEMBER MY
FIRST HELLO WORLD
PROGRAM LIKE IT
WAS YESTERDAY...



Figure 1.10 Save your test program to the desktop

If you have a look on your desktop, you should be able to see your program, with the blue and yellow Python icon on it. Double click the document icon, and your program should run.

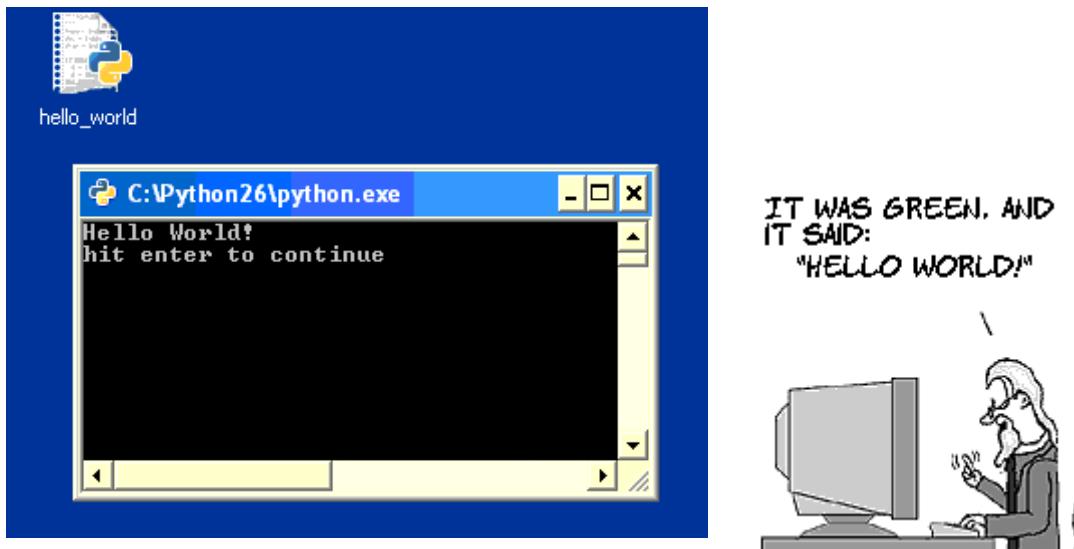


Figure 1.11 Run your script by double-clicking it

Congratulations! Python is installed and working properly on your computer! Read on to find out how to run Python from the command line - it can be an important troubleshooting tool when things go wrong. If you don't see the output, don't worry - the Troubleshooting section has some common problems and their solutions.

1.3.3 *Running Python programs from the command line*

It's also possible to run Python programs from the command line. This is often easier when you have a program which deals mainly with text input and output, or runs as an operating system script or which needs lots of input - using command line options can be easier to program than a custom settings window.

Note: There are many different ways to access and run programs. Double clicking through the GUI is one way; the command line is another. You'll learn several during the course of the book.

Running from the command line is also easier when you have a program that has a bug - you'll see an error message rather than seeing no window or having your window close immediately.

The Windows command line program is available from 'Program Files > Accessories' under the Windows Start menu.

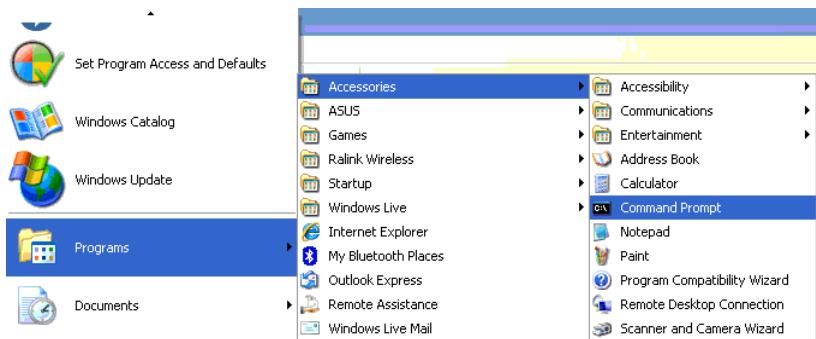


Figure 1.12 Where the Windows command line lives

If you run that program, you should see a black window with some white text. Type `cd Desktop`, to change to the desktop directory, and then `python hello_world.py` to open Python and tell it to run the script file that you created earlier.

When you do this, one of two things will happen - either your program will run, in which case you're done, or you'll see an error message saying that the Python program couldn't be found. If that happens, don't panic - you just need to tell Windows where to find Python.

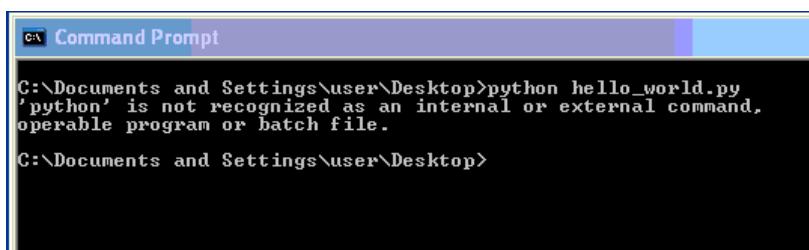


Figure 1.13 Windows doesn't know where Python is!

You need to make some changes to the path settings of Windows. The path is a list of places where Windows looks to find programs that you've asked it to run. To start, right click on your 'My Computer' icon and click 'Properties'.

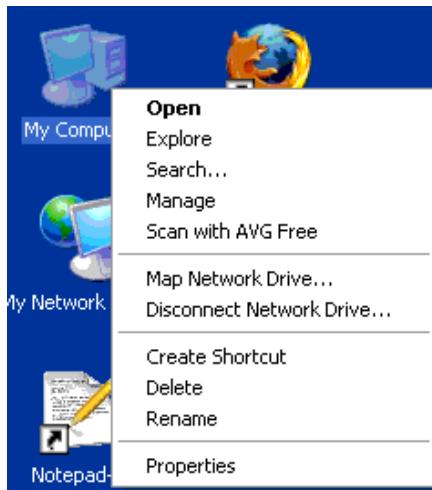


Figure 1.14 Looking in your computer's properties

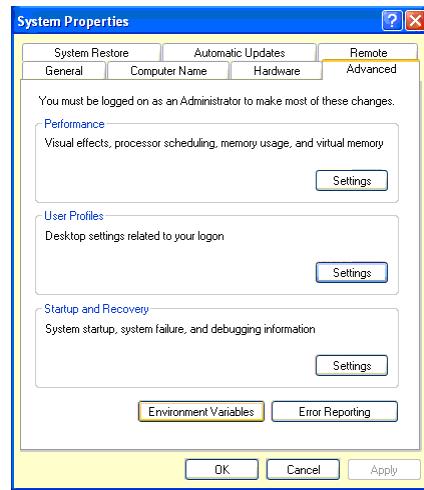


Figure 1.15 Editing your system properties

Then select the 'advanced' tab, and click the 'Environment Variables' button at the bottom. You should see a list of environment variables like the figure below.

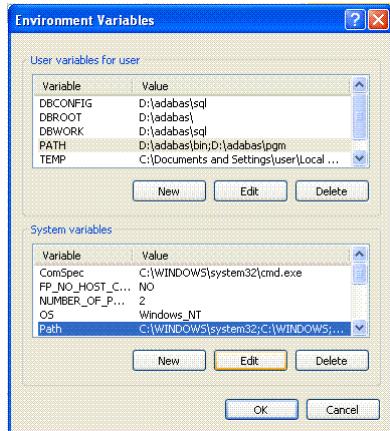


Figure 1.16 Opening the PATH variable

In the bottom half, look for the line named 'path' and double click it. In the edit box that appears, you need to add ';c:\python26' at the end of the line and hit OK.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Note: Paths are what Windows uses to find files. Each individual file on your computer has a path. You'll learn more about paths and how to use them in chapter 3.

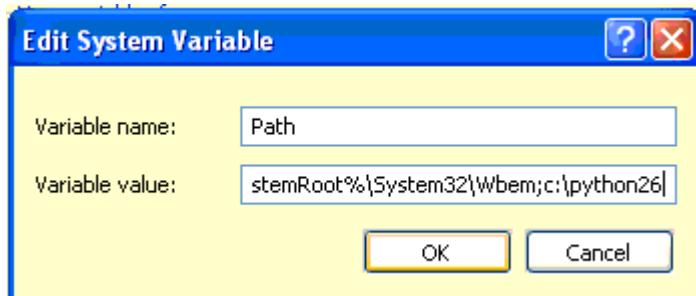


Figure 1.17 Adding Python to your PATH variable

Once you've done that, hit OK in all of the windows that you've opened until you're back at your desktop. Open another command prompt window (the old one will still have the old path settings) and type `python hello_world.py` again. You should see the output from your program.

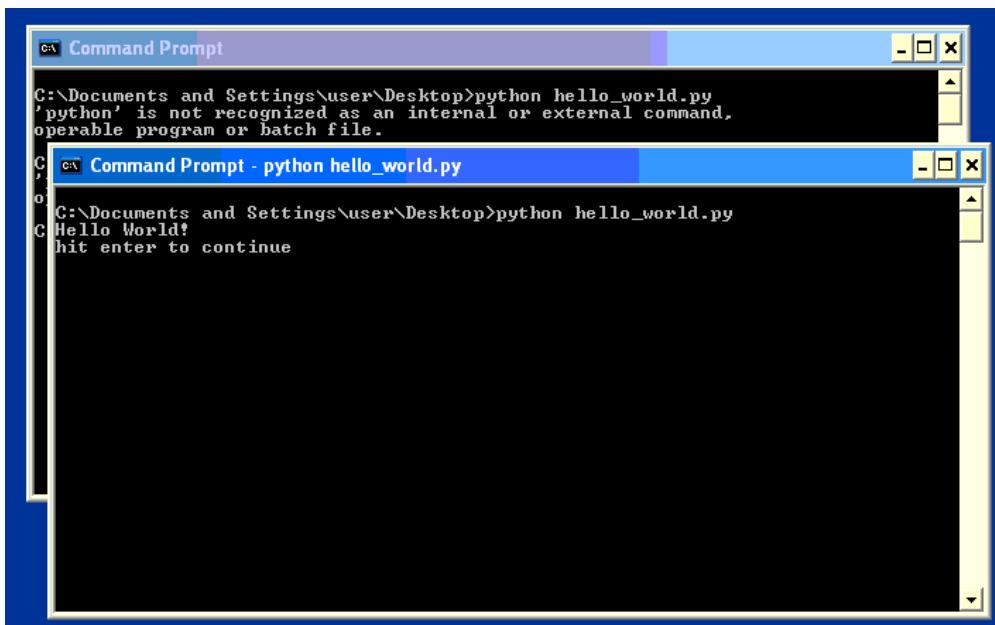


Figure 1.18 Success! Now Windows knows where Python is

Congratulations! You're now ready to start programming. You might want to read the Troubleshooting section first though, to find a better program to edit your Python programs. Next we'll find out how to install Python on Linux machines.

1.4 Linux

Using Python with Linux is harder to describe exactly, because there are a large number of Linux distributions available and they all do things in a slightly different way. I've chosen to use Gnome and Ubuntu as an example - other Linux distributions will be similar.

1.4.1 Installing under Linux

Installing Python for Linux is often not necessary, depending on which distribution you're running. Most will have some version of Python installed by default, although often one which is a few revisions out of date. You can use `python -V` to find out which one you have.

There are two main methods of installation under Linux - you can use a package or compile from source.



Package managers are straightforward to use and handle most of the dependency and compilation issues for you. Under Debian's apt-get system you can type something like `sudo apt-get install python` and have the most up to date version of Python installed automatically. You can also use `apt-cache search python` to find out what else is available, since there are usually a number of other packages (`python-dev` or `python-docs`) which you'll probably want to install as well.

Compiling from source is also an option, but somewhat outside the scope of this book. It can be a complicated process, and you'll need several other libraries (like gnu-readlines and OpenSSL) installed if you want all of Python's features. It's usually easier to install via package, but you can find more information on compiling Python at www.python.org/download/source/ if you want to go down this route.

1.4.2 Linux GUI

In general, Linux users will be more comfortable with the command line, which we cover next, but you can also run Python programs from a GUI such as Gnome, although it's a little more involved than the Windows version. Type the following program into a text editor and save it:

```
#!/usr/bin/python
print "hello world!"
ignored = raw_input("Hit enter to continue")
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

You'll also need to edit the permissions for the file to set it executable, so that you can run it directly, as shown in the next figure.



Figure 1.18 The permissions window for hello_world.py



Once you've done that, you can double click the program file and select 'run in terminal' to run your program.



Figure 1.19 Choosing what to do with our program

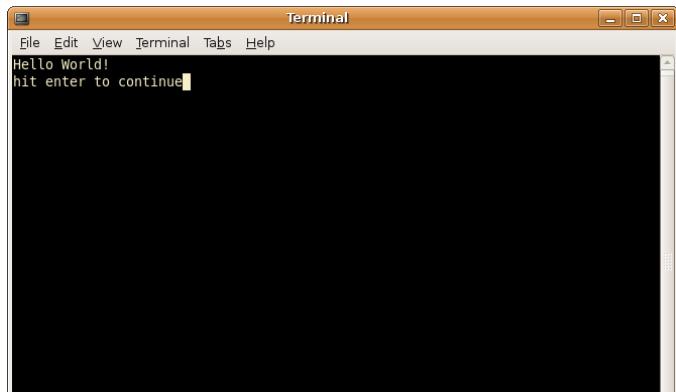


Figure 1.20 Our test program running in a terminal window under Ubuntu Linux

Once you can see this window, you're done. Although this is the easiest method of running python programs from the GUI, there are other options for running scripts which don't involve choosing whether to run or display your program. Under Gnome you can set up a program launcher. The permissions window is displayed below.



Figure 1.21 Setting the command in a launcher

Bear in mind that for a terminal-based program such as our test script, you'll need to run it within a terminal window, by issuing something like the following command:

```
gnome-terminal -e '/usr/bin/python
/home/anthony/Desktop/hello_world.py'
```

Although these examples are Gnome-specific, there are similar options for other distributions and window managers.



1.4.3 Linux command line

A lot of Linux programs are run from the command line, and Python is no exception. You'll need to be able to open a terminal window - if you're using Gnome then this is available under the Applications >> Accessories menu.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Once you've opened the terminal window, you'll see a command prompt. To execute your script, just type:

```
python path/to/your/script
```

If you've saved your script to the desktop, this can be shortened to
`python ~/Desktop/hello_world.py`

If you want to make your script look more like a system command, you can omit the `.py` on the end of the file, save it somewhere on your path (most systems support a `~/bin` folder) and make it executable with a command like `chmod 755 path/to/script.py`. Now, as long as you've kept the `#!/usr/bin/python` line as the first line of your file, you should be able to type your script's name from anywhere and have it run.

Now that Windows and Linux users have been covered, let's see how to install Python on the Mac.

1.5 Macintosh

Using Python on the Mac is pretty much like running under Linux, with the obvious exception of the graphical parts. Mac OS 10.5 comes with Python 2.5 preinstalled, and Snow Leopard (Mac OS 10.6) comes with Python 2.6. Either version should work with the code that you'll be using in Hello Python.

If you need to install a later version of Python you can also download it from the Python website and install it via a standard `.dmg` image file, but there are a few fiddly details to take care of to get things running properly.

1.5.1 Updating the Shell Profile

The first is that you'll need to tell Mac OS X to use the new version of Python if you're running things from the Terminal, otherwise it'll continue to use the built-in version. Fortunately, Python includes a script to set this for you. If you navigate to the Python folder within Applications and run the application called 'Update Shell Profile', future shell windows should use the right version.

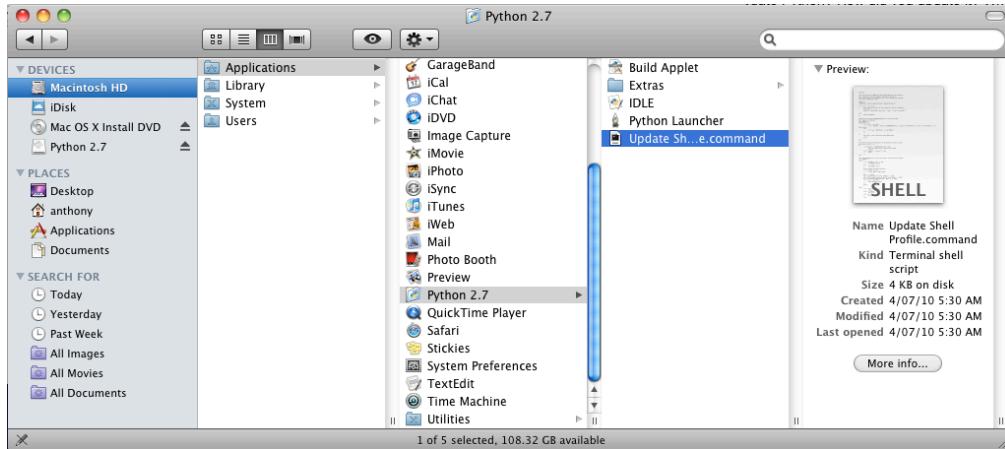


Figure 1.22 Setting the new Python path properly

1.5.2 Setting the default application

The second step is to set what Python programs do when you double click on them. By default they'll open in IDLE, the editor which comes with Python, but I prefer to have them run the Python program instead, so they behave more like a real application. If you right-click (or control-click) on a .py Python file, you should see this pop-up menu.

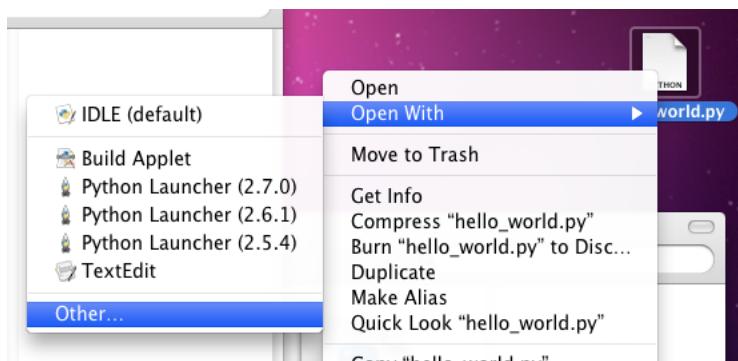


Figure 1.23 Setting the default action for Python files

This lets you choose which program to run your Python script this time, but if you select 'Other', you can pick which program will run each time.

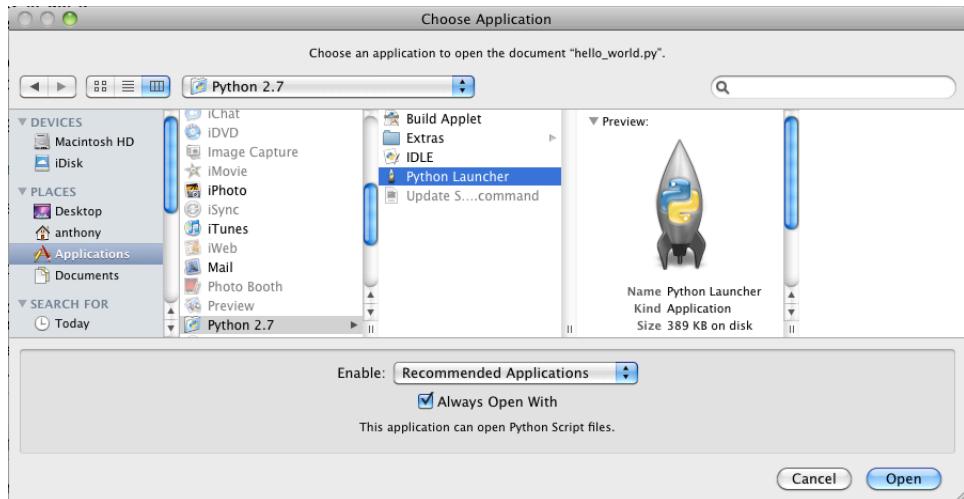


Figure 1.24 Setting the Python Launcher as the default app

Select the Python Launcher within the Python folder in the Applications directory, and tick the 'Always Open With' checkbox and click 'Open'. Now each time you double click on a .py script, it'll run it instead of opening it in IDLE. If you want to test that the command line is working properly, you can open the Terminal application and try out all of the commands above in the Linux section.

Now that you have Python installed on your chosen operating system, it's time to figure out any hiccups you may have had.



1.6 Troubleshooting

If you don't see a window when you run your Python program, there could be a few things wrong. You will potentially face a lot of errors like this as you learn to program. A good source of information is to do a web search for the exact error message or symptoms that you're getting when you try to run a program. Also, don't be afraid to ask for help (eg. on one of the Python mailing lists) if you get stuck. Here are some of the more common problems:

1.6.1 A syntax error

If you made a mistake in typing your program, you might see a window flash on and off briefly. Double check that you've typed everything correctly and then rerun your

program. If it's still not working, try running it from the command line - that'll tell you what Python is doing, and if there are any errors.

1.6.2 An incorrect file extension (Windows)

If you don't see the blue and yellow icon on your document, it means that Windows isn't recognizing that it's a Python program. Double check that your file ends in '.py'. If that doesn't work, it's possible that Python isn't installed properly - try uninstalling and reinstalling it.

1.6.3 Python is installed in a different place (Linux)

Under Linux, the `#!` line which you put at the start of your program tells the shell which program to use to run your script. If that program doesn't exist, then your command line program will fail with something like the following error:

```
bash: ./hello_world.py: /usr/local/bin/python: bad interpreter:  
No such file or directory
```

To fix this, you need to find out where Python is really installed, and update the line. The easiest way is to type `which python` at the command line, which should respond with the current location of Python. Another option is to use `#!/usr/bin/env python`, which will use the `env` program to look for `python` instead of referring to it directly.

Finally, let's see how text editors and IDEs can make programming easier.

1.7 Text editors and IDEs

To create your programs, you'll need to use a text editor to edit the files which Python reads. Programs like Microsoft Word or Wordpad are a bad choice, since they use a more complicated format that won't work with Python (or other programming languages). Instead, you'll want to use a program which edits text directly, and doesn't support formatting like bold text or pages.

If you're using a Windows PC you can always use Notepad, and there are similar applications available under Linux and Mac OS X, but it's extremely basic and won't help you to catch many common programming errors, such as indenting your code properly or not closing quotes in strings.

A better option is to use the IDLE editor that comes with Python, or else download one of the editors below, which are specifically designed for programming. Programming editors often have extra features which make programming much easier:

- they automatically indent your code
- they can color in different instructions to make your program easier to read
- they can run your program and send you back to the exact line that an error occurred, making it faster to write your programs

There is a large list of editors which are usable for Python editing available on Python's website at <http://wiki.python.org/moin/PythonEditors>. Some of the more commonly used ones are:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

- IDLE, which is installed with Python
- Emacs and Vim are used by a great many developers and are very powerful, but have a fairly steep learning curve. Cream is a variant of Vim which has more normal keybindings.
- Notepad++ is a Windows specific editor with lots of features.

Some editors are also Integrated Development Environments, or IDEs. IDEs provide extra services, above and beyond just text editing, to save you time when programming. Typically they'll give you access to a Python interpreter, some sort of auto completion, and more advanced code navigation (eg. Jumping directly to the source of an error in your program), as well as interactive debugging tools so that you can run your code step by step and look at variables while your program is running. There's also a list of Python IDEs on the Python wiki at <http://wiki.python.org/moin/IntegratedDevelopmentEnvironments>.

- IDLE is a simple IDE - it has a python interpreter included, as well as pop up completion and taking you directly to errors.
- WingIDE is a commercial IDE with integrated unit testing, source browsing and auto-completion. Wingware offer a free license to developers working on open source projects.
- PyDev is an open source plugin for Eclipse.
- SPE is also open source, and offers a wide range of features, including a code checker which tests for common programming mistakes and rates the quality of your code.
- Komodo is available in a number of forms, including an open source editor called OpenKomodo.

Ultimately, whether you use an IDE or an editor, and which one you use tends to be a decision based on personal preference and the scope of your project. As you start building larger programming projects, the investment in learning a more featured editor or an IDE will pay off. The best advice is to try a number of editors and see which ones you prefer.



1.8 Summary

In this chapter you learned the basics that you'll need to know in order to get started programming in Python.

You learned some high level details - what programming is, the philosophy of programming and the sorts of problems that programmers tend to face, and also some low level details - how to install and run Python, how to create programs and how to run them from both a graphical user interface and the command line.

One of the most important long term skills to learn when you're programming is how to deal with errors that might occur. When this happens, tracing them back to their source and ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

fixing the root cause of the problem can require some persistence and detective work, so being aware of the resources that are available to you is important. We'll be learning the details of how to deal with errors in your program in later chapters.

The background that you've learned in this chapter, particularly the part where you run a Python program, will help you in the chapters to come, where we'll be taking a look at Python's basic statements and using them to write a game called Hunt the Wumpus.

2

Hunt the Wumpus

Now that you have Python set up and installed, and know how to enter and run a test program, let's get started writing a real one. I'll start out by explaining a few of Python's basic features and then we'll create a simple text-based adventure game, called "Hunt the Wumpus".

As you progress through the chapter, you'll add features to your game, building on the initial version. This is how most programmers (including the author) learned to program: learn just enough about the language to be able to write a simple program, then build up from there. In order to do that you'll need to learn more, but you only need a little bit more to be able to add a little bit to your program. Repeat a few more times, and you'll have a program that you couldn't have created in one sitting. In the process you'll have learned a lot about the programming language.

In this chapter you'll experience the early days of programming first hand, as you write your own version of Hunt the Wumpus. The text based interface is ideal for your first program since you only need

to know two simple statements to handle all of your input and output. All of your input will be strings, so the logic of your program is straightforward and you don't need to learn a lot in order to start being productive.

HAIRY, SMELLY
BEAST? SOUNDS
LIKE PITR!



A brief history of Hunt the Wumpus

Hunt the Wumpus was a popular early computer game written by Gregory Yob in 1976. It puts you in the shoes of an intrepid explorer, delving into a network of caves in search of the hairy, smelly, mysterious beast known only as the wumpus. Many hazards faced the player, including bats, bottomless pits, and of course the wumpus. Since the original

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

game was released with source code, it allowed many other people to create their own versions of Hunt the Wumpus with different caves and hazards, ultimately leading to the development of an entire genre of first person "adventure" games such as Adventure and Zork.

By the end of this chapter, you'll know how to add features to your fully functioning version of "Hunt the Wumpus", and even tweak it to create your own version.

Before we get to the cave adventures, let's figure out the basics.

2.1 What's a program?

As you learned in chapter 1, a program consists of statements that tell the computer how to do something. Programs can execute simple tasks such as printing a string to the screen, and can be combined to execute complex tasks like balancing accounts or editing a document.

Definition Program—a series of instructions, which tell your computer how to do certain things, usually called *statements*.

The basic mechanics are straightforward; Python starts at the first line of your program and does what it says, then moves to the next and does what that says. For example, enter this simple Python program:

```
print "Hello world!"  
print "This is the next line of my program"
```

The code outputs output the following text to the screen:

```
Hello world!  
This is the next line of my program
```

There are many different types of things that Python can do. So that you can get started on your program as soon as possible, this chapter will give you a brief idea of the statements that you can use to tell Python what to do. We won't be going into extensive detail, but you'll get everything that you need to be able to follow what's going on.

There's a lot to take in, so don't worry too much if you don't understand it all at once. Perhaps think of it as painting a picture, and this is a light pencil sketch before we get started properly. Some parts will be hazy at first, but it's important to get a sense of the whole before individual details will make sense.

You might want to read this part at your computer, so that you can experiment with different statements to see what works and try out your own ideas.

We'll start by investigating that `print` statement we just tried out.

WHAT'S UP GREG? HOW'S
THE PYTHON COMING ALONG?

I HEARD ABOUT THIS
"HUNT THE WUMPUS"
GAME SO I THOUGHT I'D
WRITE MY OWN VERSION



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

2.1.1 Writing to the screen

The `print` statement is used to tell the player things that happen in our game, like which cave they're in or whether there's a wumpus nearby. You've already seen the `print` statement in our Hello World program, but there are some extra things that it can do, and you're not just limited to printing out words; pretty much anything in Python can be printed:

```
print "Hello world!"  
print 42  
print 3.141592
```

You can print out lots of things at once just by putting a comma between them:

```
print "Hello", "world!"  
print "The answer to life, the universe and everything is", 42  
print 3.141592, "is pi!"
```

Just printing statements wouldn't make for a very interactive game. Let's see how we can add options.

2.1.2 Remembering things with variables

Python also needs some way to know what's happening. In our Hunt the Wumpus game, for example, it needs to be able to tell which cave the wumpus is hiding in, so that we'll know when the player has found it. In programming, we call this memory *data*, and it's stored using a type of object called a *variable*. Variables have names so that we can refer to them later in the program.

To tell Python to set a variable, we choose a name for our variable, and then use the equals sign to tell Python what the variable should be. Variables can be letters, numbers, words or sentences, as well as some other things that we'll learn about later:

```
variable = 42  
x = 123.2  
abc_123 = "A string!"
```

In practice, your program can get quite complex, so it helps if you choose names which tell you what the variable actually means or how it's supposed to be used. In our Hunt the Wumpus program, we'll use variable names like this:

```
player_name = "Bob"  
wumpus_location = 2
```

I THINK PITR WAS SAYING THAT HE'D WRITTEN A VERSION. HIS CODE CAN BE A BIT CRUFTY SOMETIMES, BUT WHY DON'T YOU TAKE A LOOK?



NOTE There are some restrictions on what your variable names can be - they can't start with a number, have spaces in them or conflict with some of the names which Python uses for its own purposes - but in practice you won't run into these limitations if you're using meaningful names.

Table 2.1 Types of variable used in Hunt the Wumpus

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Type	Overview
Numbers	Whole numbers like 3 or 527, or floating point numbers, like 2.0 or 3.14159. Python won't switch between them, so you'll need to be careful in some cases: for instance, 3 / 2 is 1 instead of 1.5. 3.0 / 2 will give the right answer.
Strings	A sequence of characters, including a-z, numbers and punctuation. They can be used for storing words and sentences. Python has a few different ways of representing strings: you can use both single or double quotes, 'foo' or "foo", as well as special versions with triple quotes that can run over multiple lines.
Lists	A collection of other variables, which can include other lists. Lists begin and end with a square bracket, and the items inside are separated with commas: ["foo", "bar", 1, 2, [3, 4, 5]]

Now that we have variables working, how do we get the player involved?

2.1.3 Asking the player what to do

The program also needs some way of asking the player what to do in certain situations. For Hunt the Wumpus, we'll use the `raw_input` command. When Python runs that command, it will prompt the player to type something in, and then whatever they typed can be stored in a variable.

```
player_input = raw_input(">")
```

Next we need to figure out what to do with our user input.

2.1.4 Making decisions

If that was all there was to programming, it would be kind of boring. All of the interesting stuff happens when the player has to make a choice in the game. Will they pick cave 2? or cave 8? Is the wumpus hiding in there? Will the player be eaten? To tell Python what we want to happen in certain situations we use the `if` statement, which takes a *condition*, such as two variables being equal or a variable being equal to something else, and something to do if the condition is met.

```
if x == y:
    print "x is equal to y!"
if a_variable > 2:
    print "The variable is greater than 2"
if player_name == "Bob":
    print "Hello Bob!"
```

You can also use an `else` command, which tells Python what to do if the condition doesn't match.

```
if player_name == "Bob":
    print "Hello Bob!"
else:
```



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```
print "Hey! You're not Bob!"
```

So that Python can tell the body of the `if` statement from the rest of your program, the lines which are part of it are indented. If you put an `if` statement within another `if` statement, then you need to indent again, for a total of eight spaces. Normally you'll use four spaces for each level of indentation.

Table 2.2 Common expressions

Expression	Overview
<code>name == "bob"</code>	True if the variable <code>name</code> stores the string "bob". Python uses two equal signs to distinguish it from assignment - <code>name = "bob"</code> means something completely different.
<code>name != "bob"</code>	True if the variable <code>name</code> is something other than the string "bob". <code>!=</code> is generally read as "not equals".
<code>a > 0</code>	True if the variable <code>a</code> stores a number which is greater than 0.
<code>0 <= a <= 10</code>	True if <code>a</code> is a number between 0 and 10, inclusive.
<code>"ab" in "abcde"</code>	You can also tell whether a string is part of another string, by using <code>in</code>
<code>not "bob" in "ab"</code>	Python also has the <code>not</code> and <code>not in</code> commands, which reverse the sense of "bob" not in "ab" an expression

Now that you have a handle on decision-making statements, let's see what we can do to keep the program going.

2.1.5 Loops

One of the great things about computers is not just that they can do things, but they can do things over and over and over and not get bored. Big lists of numbers to add? No problem. Hundreds of lines of files? Ditto. The program just needs to know what it's going to be repeating, and when it should stop. In our Hunt the Wumpus program, we'll be using a structure called a *while loop*, which loops while a condition that you specify is true, and a `break` statement which allows us to control when it stops.

```
while True:
    print "What word am I thinking of?"
    answer = raw_input(">")
    if answer == "cheese":
        print "You guessed it!"
        break
    else:
        print "No, not that word..."
```

We're almost to the end of the tour of Python's basic features - our last one is functions.



2.1.6 Functions

There are also a few statements called *functions* in our wumpus program. They usually tell us useful things about our program, the player or the variables, and they look like this:

```
range(1,21)
len(cave_numbers)
```

Normally functions will tell you things by *returning* a value, which you can store in a different variable, or use directly:

```
cave_numbers = range(1,21)
print "You can see", len(cave_numbers), "caves"
```

Now that we've covered some of the basics, let's see how you can use them to build a simple program. This doesn't do everything that the original wumpus program did, but for now we just want to get something off the ground to see how it all fits together.



Incremental programming

In later sections of this chapter we'll build on this program by adding features or refining ones which are already there, and tidy up as we go. This is how most programmers tend to work - start simply and build as you go. You can download this program from <http://www.manning.com/briggs/>, but I'd suggest following along and typing it in as you read it. That'll help you remember the individual statements more easily, but you'll also be establishing a key habit which will help you as you write larger programs - start simply and build up.

Table 2.3 recaps the basic features you've learned in this section.

Table 2.3 Basic Python features

Feature	Overview
Statements	Usually one line in a program (but can be more) which tell Python to do something
Variables	Used to refer to information so that a program can use it later. There are many different types of information that Python can refer to.
If-then-else	This is how you tell Python to make a decision. An if statement consists of at least an expression, such as <code>x == 2</code> or <code>some_function() < 42</code> and something for Python to do if that expression is true. You can also include an <code>else</code> clause, which tells Python what to do if the expression is false.
Loops	Used to repeat certain statements multiple times. They can be either while loops, which are based on a condition like an if statement, or for loops, which run once for each element of a list. From within a loop, you can use the

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

	continue statement, which jumps to the next iteration of the loop, or a break statement, which breaks out of the loop entirely.
Functions	A series of statements which can be run to return a value to a separate part of your program. They can take input if necessary, or they can read (and sometimes write) to other variables in your program.
Indenting	Since you can nest functions, loops and if statements within each other, Python uses white space (normally four spaces per level) at the start of a line to tell which statements belong where.
Comments	Whenever Python encounters a # character that's at the start of a line, it'll ignore that line and not run it. As well, if there's a # character that's not inside a string, it will ignore the rest of the line. Comments are used to explain parts of your program, either to other programmers, or to yourself in a few weeks, when you've forgotten most of the details of what you were doing. You won't see too many in the book, since we can use

You've learned a lot in this section, but in the next section you'll put this knowledge to good use and write your first program.

2.2 Your first program

Now that you have an understanding of the basics of Python, let's take a look at our program. It's difficult to see how a program works by just reading about individual features, since in a working program they all depend on each other. In this section we'll explore our first version of Hunt the Wumpus and solve the first problem that comes up.

Note: Experimentation is critical to developing an intuition for how Python works, and how all of the parts fit together. Without it, you'll be stuck cut and pasting other people's programs, and when you have a bug, it'll be impossible to fix.

RUN!



2.2.1 The first version of Hunt the Wumpus

If you don't understand the next listing straight away, don't worry. A good way to figure out what it does is to experiment with it - change a few statements, run it again and see what the differences are. Or copy a few statements into another file so that you can run them in isolation.

Listing 2.1 - Your first version of Hunt the Wumpus

```
from random import choice          #1
                                #1
cave_numbers = range(1,21)          #1
wumpus_location = choice(cave_numbers) #1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

player_location = choice(cave_numbers)           #1
while player_location == wumpus_location:       #1
    player_location = choice(cave_numbers)       #1

print "Welcome to Hunt the Wumpus!"             #2
print "You can see", len(caves), "caves"         #2
print "To play, just type the number"            #2
print "of the cave you wish to enter next"       #2

while True:                                     #3
    print "You are in cave", player_location     #4
    if (player_location == wumpus_location - 1 or
        player_location == wumpus_location + 1):   #4
        print "I smell a wumpus!"                 #4

    print "Which cave next?"                      #5
    player_input = raw_input(">")                 #5
    if (not player_input.isdigit() or
        int(player_input) not in cave_numbers):    #5
        print player_input, "is not a cave!"        #5

    else:                                         #6
        player_location = int(player_input)         #6
        if player_location == wumpus_location:      #6
            print "Aargh! You got eaten by a wumpus!" #6
            break                                     #6

```

We start with the "set up" part of our program **(1)**. We're storing a list of numbers in our program, each of which represents a cave. Don't worry too much about the first line - we'll learn more about the import statement in chapter 3. The choice function will return one of the caves, picked at random, and we use it to place the wumpus and the player in their starting positions. Note the loop at the end that we use to tell if the player and the wumpus are in the same spot - it wouldn't be a very fun game if the player got eaten straight away!

The introductory text **(2)** tells the player how the game works. We use the len() function to tell how many caves there are. This is useful because you may want to change the number of caves at a later point, and using a function like this means that you only have to change things in one place when you define the list of caves.

Our main game loop **(3)** is where the game starts. When playing the game, the program gives you details of what you can see, asks you to enter a cave, checks to see whether you've been eaten, then starts over at the beginning. while loops will loop as long as their condition is true, so while True: means 'loop over and over again without stopping' (we'll handle the stopping part in a minute).

The first if statement **(4)** tells the player where they are, and warns them if the wumpus is only one room away ("I smell a wumpus!"). Note how we're using the player_location and wumpus_location variables. Since they're



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

numbers, we can add and subtract numbers from them. If you're in cave 3, and the wumpus is in cave 4, then the `player_location == wumpus_location - 1` condition will be true, and Python will display the message.

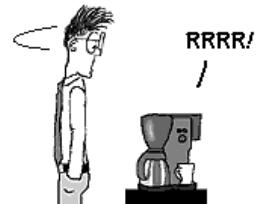
We then ask the player which cave they want next (5). We do some checking to see that they've put in the right sort of input. It has to be a number, and it has to be one of our caves. Note also that the input will be a string, not a number, so we have to convert it using the `int()` function. If it doesn't match what we need, we display a message to the player.

If our input does match a cave number, we'll trigger this `else` clause instead (6). It updates our `player_location` variable with the new value, and then checks to see if the player's location is the same as the wumpus'. If it is ... "Aargh! You got eaten by a wumpus!". Once you've been eaten, the game should stop, so we use the `break` command to stop our main loop. Python has no more statements to execute, and so the game ends.

2.2.2 Debugging

If you've typed in listing 2.1 exactly as written and run it, you'll notice that it doesn't quite work as advertised. In fact it refuses to run at all. The exact results will depend on your computer's operating system and how you're running your Python program, but you should see something like Listing 2.2. If you don't, try running your program from the command line by typing `python wumpus-1.py`.

NO ONE'S QUITE SURE -
THEY'RE MASTERS OF
DISGUISE!



Listing 2.2 BANG! Your program explodes:

```
Welcome to Hunt the Wumpus!
You can see
Traceback (most recent call last):
  File "wumpus-1.py", line 10, in ?
    print "You can see", len(caves), "caves"
NameError: name 'caves' is not defined
```

What's happened is that there's a *bug* in the program. There's a statement in listing 2.1 that Python doesn't know how to run, and rather than guess what you meant, it'll stop and refuse to go any further until you've fixed it.

Luckily the problem is easy to fix - Python tells you what line is at fault, the type of error that's been triggered and a rough description of the problem. In this case it's line 10, and the error is `NameError: name 'caves' is not defined`. Oops - our program tried to access the variable `caves` instead of `cave_numbers`. If you change line 10 so that it reads:

```
print "You can see", len(cave_numbers), "caves"
```

I AM TELLING NOT TO BE
TOUCHING CODE! NOW
EXPERIMENT IS RUINED!
YOU LEFT YOUR
EXPERIMENTAL CODE
ON THE SERVER WHERE
ANYONE COULD
FIND IT?!



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

then the program should run.

Congratulations - your first real Python program! Next let's see what we else we can do to improve Hunt the Wumpus.

2.3 Experimenting with your program

Experimenting with programs is the main way that most programmers learn how to deal with new programming problems and find solutions. You can also experiment with your new program and see what else you can make it do. You're the one typing it in, so the wumpus program is yours. You can make it do whatever you want it to. If you're feeling brave, here are some ideas:

2.3.1 More (or fewer) caves

You might find 20 caves to be too many, or too few. Luckily it's your program now, so you can change the line where we define `cave_numbers` to be smaller or larger. Question: What happens if you have only one cave?

2.3.2 A nicer wumpus

We haven't given the player a bow and arrow yet, so all they can do is wander aimlessly around the caves until they blunder into the wumpus and get eaten. Not a very fun game.

How about if we change the line where the player finds the wumpus to read:

```
print "You got hugged by a wumpus!"
```

Aww, what a nice wumpus? Of course, the author and publisher disclaim any and all responsibility for the dry cleaning of your clothes to get the wumpus smell out should you choose this option.

2.3.3 More than one wumpus

The wumpus must be awfully lonely down in the caves. How about giving it a friend? A bit trickier, but you already have the existing wumpus code to work from. Just add a `wumpus_friend_location` variable, and check that wherever you check the first `wumpus_location`:

Listing 2.3 - Adding a friend for the wumpus

```
wumpus_location = choice(cave_numbers)
wumpus_friend_location = choice(cave_numbers)
player_location = choice(cave_numbers)
while (player_location == wumpus_location or
       player_location == wumpus_friend_location):
    player_location = choice(cave_numbers)
...
if (player_location == wumpus_location - 1 or
    player_location == wumpus_location + 1):
    print "I smell a wumpus!"
if (player_location == wumpus_friend_location - 1 or
    player_location == wumpus_friend_location + 1):
    print "I smell an even stinkier wumpus!"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

    ...
    if player_location == wumpus_location:
        print "Aargh! You got eaten by a wumpus!"
        break
    if player_location == wumpus_friend_location:
        print "Aargh! You got eaten by the wumpus' friend!"
        break

```

Now that's a more interesting game!

There's still more we can do to improve our Hunt the Wumpus game, starting with the cave structure.

2.4 Making the caves

The first thing that you might have noticed with listing 2.1 is that the "maze of caves" isn't really a maze. In fact, it's more like a corridor, with each cave neatly placed in a line, one after the other. It's easy to figure out where the Wumpus is - just move into the next cave in sequence until you smell it. Since figuring out the location of the Wumpus is such an integral part of the game, that'll be the first thing to fix. In the process, we'll learn a bit more about Python's *lists*, and *for loops*.

AND THOSE WUMPUSES
EXTERMINATORS ARE
EXPENSIVE!
AT LEAST WE'LL BE BACK
UP AND RUNNING SOON. \



2.4.1 Lists

Assume for a second that you wanted to write a program to help you do your shopping. The first thing that you'd need is some way to keep track of what you wanted to buy. Python has a built in mechanism for exactly this sort of thing, called a *list*. You can create and use it just like any other variable.

```
shopping_list = ['Milk', 'Bread', 'Cheese', 'Bow and Arrow']
```

If you want to find out what's in your shopping list, you can just print it out, or you can use an *index* to find out what's in a specific place. Lists will keep everything in the order that you defined it. The only catch is that the index of an array starts at 0, rather than 1.

```

print shopping_list
['Milk', 'Bread', 'Cheese', 'Bow and Arrow']
print shopping_list[0]
'Milk'

```

IS GREATER SIBERIAN
WUMPUSES - MUCH
TOUGHER THAN PUNY
AMERICAN WUMPUSES!

A clever trick if you need it, is that an index of -1 gets the last item in your array:

```

print shopping_list[-1]
'Bow and Arrow'

```



You can also check whether a particular thing is in your list:

```

if 'Milk' in shopping_list:
    print "Oh good, you remembered the milk!"

```

The other cool thing about lists is that they're an all purpose bag. You're not limited to just strings or numbers - you can put anything at all in there, including other lists. If

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

you had lists for two stores (say, the supermarket and Wumpus R Us - "for all your Wumpus hunting needs!"), you could store them in their own lists, and then store those lists in one big list:

```
supermarket_list = ['Milk', 'Bread', 'Cheese']
wumpus_r_us_list = ['Bow and Arrow', 'Lantern', 'Wumpus B Gone']
my_shopping_lists = [supermarket_list, wumpus_r_us_list]
```

You can also put things into a list and take them out again. If you forget to put rope on your list, that's easily fixed:

```
wumpus_r_us_list.append('Rope')
print wumpus_r_us_list
['Bow and Arrow', 'Lantern', 'Wumpus B Gone', 'Rope']
```

and we actually want to catch a Wumpus instead of scaring it away, so perhaps the "Wumpus B Gone" isn't such a good idea:

```
wumpus_r_us_list.remove('Wumpus B Gone')
print wumpus_r_us_list
['Bow and Arrow', 'Lantern', 'Rope']
```

You can also cut parts of a list out if you need to, by giving two values separated with a colon. This is called slicing a list. Python will return another list starting at the first index, up to but not including the second index. Remember that list indexes starts at zero.

```
first_three = wumpus_r_us_list[0:2]
```

If you give a negative value, then Python will measure from the end instead of the front:

```
last_three = wumpus_r_us_list[-2:-1]
```

And if you leave a value out of a slice, Python will use the start or end of the list. These two slices are exactly the same as the previous two:

```
first_three = wumpus_r_us_list[:2]
last_three = wumpus_r_us_list[-2:]
```

Finally, once you've taken everything out of a list, you'll end up with an *empty list*, which is represented with two square brackets by themselves: []

List Gotchas One difference between Python and some other programs such as C is that Python's variables aren't really variables in the classic sense. For the most part they behave as if they are, but they're actually more like a label or a pointer to an object in memory. When you issue a command like `a = []`, Python creates a new list object and makes the `a` variable point to it. If you then issue a command like `b = a`, `b` will point to the same list object, and anything that you do via `a` will also appear to happen to `b`.

Now that you know about lists, let's tackle for loops.

2.4.2 For loops

Once you have all of your things in a list, a common way to use the list is to do something to each item in it. The easiest way to do this is to use a type of loop called a *for loop*. A for loop works by repeating some statements for every item in a list, and assigns that item to a variable so that you can do something with it:

```
print "Wumpus hunting checklist:"
for each_item in wumpus_r_us_list:
    print each_item
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

if each_item == "Lantern":
    print "Don't forget to light your lantern"
    print "once you're down there."

```

Except for the variable, for loops are much the same as while loops. The break statement which we used in our while loop in listing 2.1 will also work in for loops.

NOTE This is a common pattern in programming - get a bunch of stuff, and do something to everything in your bunch.

2.4.3 Coding your caves

In Hunt the Wumpus, each cave is only supposed to connect to a small number of other caves. For example, cave #1 might only have tunnels to caves 5, 7 and 12, then #5 has tunnels to 10, 14 and 17. This limits the number of caves that the player can visit at once, and navigating their way through the cave system to try and find the wumpus becomes the central challenge of the game.

In your first version of Hunt the Wumpus, you were already using a list of cave numbers to tell Python where the wumpus and player were. In our new version we'll use a similar sort of list, but changed so that it can tell us which caves can be visited from a particular place. For each cave, we'll need a list of other caves, so what we're after is a list of lists. In Python, it looks like this:

```

caves = [ [2, 3, 7],
          [5, 6, 12],
          ...
        ]

```

What this tells us is that cave #0 (don't forget that lists start with their index at 0) links to caves 2, 3 and 7, cave #1 links to caves 5, 6 and 12, and so on. Since the caves are generated randomly, your numbers will be different, but the overall structure will be the same. The number of the cave is the same as its index in the list so that Python can easily find the exits later on. Let's replace section 1 of listing 2.1 so that it sets up our new and improved cave system.

Listing 2.4 Setting up our caves

```

import random

cave_numbers = range(0,20)                      #1
caves = []                                       #1
for i in cave_numbers:                           #1
    caves.append([])                            #1

for i in cave_numbers:                           #2
    for j in range(3):                         #2
        passage_to = choice(cave_numbers)      #3
        caves[i].append(passage_to)            #3
print caves                                      #4

```

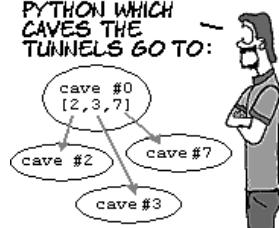
©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

A LIST OF LISTS? HOW DOES THAT WORK?

AH, I SEE - [2, 3, 7] IS A CAVE...

AND THE NUMBERS TELL PYTHON WHICH CAVES THE TUNNELS GO TO:



#1 A new list of caves
#2 Creating tunnels
#3 Linking caves
#4 Debugging

We're still using a range function to generate our list of caves, but we've changed the range so that it starts at 0 instead of 1, to match the indexes of our list **(1)**. Then we make an empty list for each of the caves that we're supposed to have. At this point it's just a list of unconnected caves.

For each unconnected cave in our list, we pick three other caves at random, and append them onto this cave's list of tunnels **(2)**. To make our life easier, we use another for loop inside the first one, so that if we need to change the number of tunnels later on, we only need to change the number 3 to whatever we'd like.

When we're picking a cave to link to, we use a *temporary variable* to store it **(3)**. The main advantage of this is that we can use a meaningful name to make the code much easier to read, since you know what that variable does. Note that we could have joined these two lines together by writing `caves[i].append(choice(cave_numbers))` instead, (using the `choice(cave_numbers)` function directly), but it's much harder to read.

Just so that we can check the program is working properly, we print out the list of caves **(4)**. This is usually referred to as a *debug string*, since it's a very handy technique when you're trying to debug a program. You can remove this line once the program is running properly, since the player shouldn't know the caves ahead of time.

Now when you run your program, it should print out a list of caves, like this:

```
[[8, 7, 14], [1, 18, 4], [4, 8, 15], [6, 6, 0],
 [5, 3, 6], [15, 9, 10], [2, 13, 5], [17, 18, 3],
 [4, 8, 15], [18, 17, 2], [1, 9, 15], [11, 4, 16],
 [16, 10, 6], [2, 10, 5], [13, 4, 6], [8, 14, 11],
 [16, 4, 10], [3, 12, 17], [18, 18, 0], [2, 8, 5]]
```

Which is exactly what we're expecting. In this one, cave #0 links to caves 8, 7 and 14, cave #1 links to caves 1, 18 and 4, and so on. Now that we have our list, all that we have to do is alter the rest of our program to use it. Sections 4 and 5 of listing 2.1 should be replaced with listing 2.5.



Listing 2.5 Altering our program to use the new cave system

```
print "You are in cave", player_location
print "From here, you can see caves:", caves[player_location] #1
if wumpus_location in caves[player_location]: #1
    print "I smell a wumpus!"

print "Which cave next?"
player_input = raw_input(">")
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

if (not player_input.isdigit() or
    int(player_input) not in caves[player_location]):           #1
    print player_input + "?"
    print "That's not a direction that I can see!"
    continue

```

Cue balls in code and text

#1 Changes to the code

All we're using our cave list for is finding out which caves the player can enter next, so the changes to the code (1) are pretty straightforward. Instead of checking whether the player's input is within the list of cave numbers, we just check the list for the specific cave that we're in.

There's a bug in the code we used to set up our caves. You may not believe me, especially if you've played a few games already, but there is. Let's get back into debugging mode.

2.5 Fixing a more subtle bug

What makes this hard to spot is that the code runs properly, but sometimes the game is impossible to win. In this section we'll look at why the game can be unwinnable and how to fix it.

NOTE These are the worst kind of bugs to hunt down - your program doesn't crash or spit out any obvious errors, but it's definitely wrong.

We'll start by examining how the caves are linked.

2.5.1 The problem

The trick is that all of the cave tunnels are generated randomly, so they can be linked in any possible way. Let's think about an easier case, with a small cave system. Suppose that the tunnels happened to link like this?

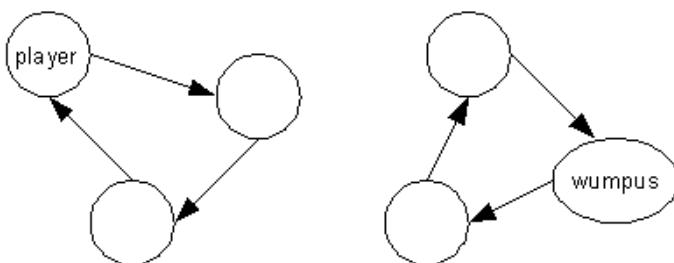


Figure 2.1 Not a very fun game.

The player wouldn't ever be able to catch the wumpus.

With lots of caves it's less likely that we'll strand the player in an isolated corner of the map, but ideally we'd like our program to be as *bulletproof* as we can make it, so that it's impossible, rather than just unlikely.

2.5.2 The solution

There are two changes that we'd need to make to our map generation to solve the problem. The first is to make our tunnels two way. In other words, if we can go from cave 1 to cave 2, then we should be able to move back from cave 2 to cave 1.

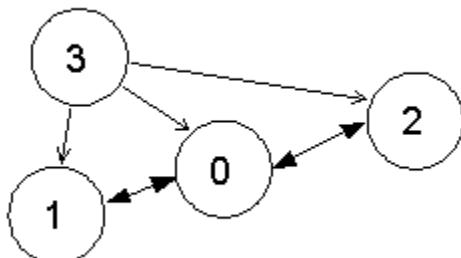


Figure 2.2 Adding cave 3 to our network

The second is to make sure that every cave is linked together, and that there are no isolated caves (or networks of caves). What we're looking for is called a *connected* structure:



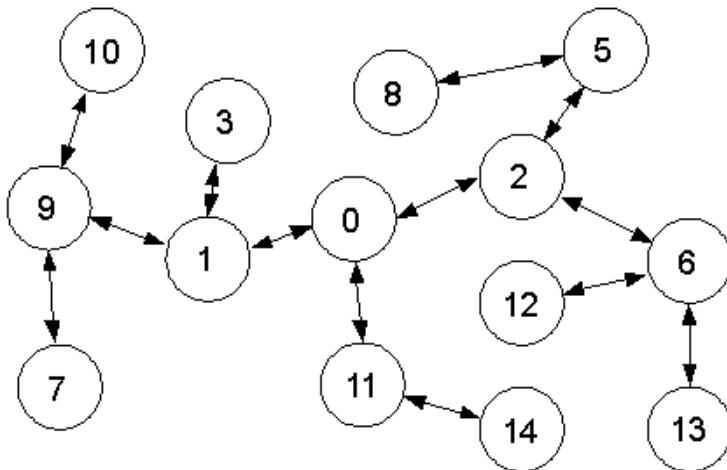


Figure 2.3 That's much better!

Now, no matter how we join up the rest of the passages, we can be sure that the player can reach every cave, since they can just go back the way that they came and choose a different passage. Of course, if they forget which way they came then they can still get lost, but that's their fault rather than ours.

So, how do we link our tunnels like this in Python?

2.5.3 Coding connected caves

The way we do it is straightforward - when we create a one way tunnel, we just add another one way tunnel back the way we came. Every time we say `caves[a].append[b]` we also say `caves[b].append[a]`. The program looks something like listing 2.6:

Listing 2.6 Creating a linked cave network

```

import random

cave_numbers = range(0,20)
caves = []
for i in cave_numbers:
    caves.append([])

unvisited_caves = range(0,20)      #1
visited_caves = [0]                #1
unvisited_caves.remove(0)          #1

while unvisited_caves != []:
    #2
  
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

i = choice(visited_caves)           #3
if len(caves[i]) >= 3:             #3
    continue                         #3

next_cave = choice(unvisited_caves)  #4
caves[i].append(next_cave)          #4
caves[next_cave].append(i)          #4

visited_caves.append(next_cave)     #5
unvisited_caves.remove(next_cave)   #5

for number in cave_numbers:         #6
    print number, ":", caves[number] #6
print '-----'                      #6

for i in cave_numbers:              #7
    while len(caves[i]) < 3:        #7
        passage_to = choice(cave_numbers) #7
        caves[i].append(passage_to)      #7

    for number in cave_numbers:      #6
        print number, ":", caves[number] #6
    print '-----'                  #6

#1 Set up
#2 Main loop
#3 Pick a random visited cave
#4 Link it to an unvisited one
#5 Mark the cave as visited
#6 Progress report
#7 Dig out the rest of the tunnels

```

First, create a list of caves which we haven't visited, and visit cave 0 (**1**). We loop until `unvisited_caves` is empty (**2**), i.e. there are no unvisited caves left. We pick one that has fewer than 3 tunnels to other caves (**3**). If we link one cave to 10 others the game will be too hard, since it'll be difficult or impossible to work out which tunnel leads to the wumpus.

(**4**) is where we're actually building our cave. We pick a random unvisited cave, and put a tunnel in the old cave to our new one, and then a link from our new one back to the old one. This way we know that the player can find their way back.

Once we're done with the cave, we can move it from the unvisited list to the visited list (**5**). Steps (**3**), (**4**) and (**5**) get repeated until we run out of caves (`unvisited_caves == []`)

The progress report lines (**6**) are optional, but if you include them you'll be able to see your caves in the process of being built, since every time Python goes through the loop it'll print out the current cave structure. It also looks a bit nicer than just a `print caves`.

Now that all of our caves are linked, the rest is just adding some more one-way tunnels (**7**). It's exactly the same as our previous example, except that we'll already have at least one tunnel in each cave already. So that we don't add more than 3 tunnels, we change our for loop into a while loop.

Now that our cave problem has been solved, let's see how functions can improve the readability of our code.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

2.6 Clean up your code with functions!

If you've been following along with the examples (you should!), you'll notice that our program is growing longer and longer. It's a relatively short example, but even so it's becoming hard to understand what's happening in the program. If you wanted to give a copy of your program to a friend for them to use, they might have a hard time figuring out what all of the bits do.

NOTE Remember how we were talking about hiding complexity in chapter 1? Functions are one of the critical ways that Python can hide the complex parts of your program.

It's time for a spring cleaning, and we're going to do that by designing our program to use some functions. We've been using a few functions so far - they're the `choice()`, `len()`, `raw_input()` parts of your code - so you have a rough idea of how they work. What you don't know (yet) is what they really are, or how to create your own.

2.6.1 Function basics

Functions are a way of making a section of your program self contained, often referred to as *encapsulation*. It's an important way of breaking down a program into easily understood parts. A good rule of thumb is that each function "should do one thing and do it well." In other words, there should be as little overlap between your functions as possible. Much like the parts of the engine of your car - if a fan belt breaks, you should just have to replace the fan belt. It wouldn't make much sense to have to change your tyres or spark plugs as well.

There are several advantages to using functions in your program:

- You only have to write that part of the program once, and then you can use it wherever you like. Later, if you don't like the way that your program works or you find a bug, you only have to change your code in one place.
- In much the same way that you can choose nice variable names that tell you what's going on in your program, you can also choose nice function names that describe what the function does.
- One of the reasons that your code is hard to understand now is that it's all in one big piece and it's hard to tell where one part begins and ends. If it were broken up into smaller parts, with a part for setting up the caves, a part for making a tunnel, a part for moving the player and so on, you only need to read (and understand) one small piece of the program instead of a large chunk.



Functions are one of the main units of encapsulation in Python. Even advanced structures such as classes, which we cover in chapter 6, are composed of functions. Python also has

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

what are called *first class functions*, which means that you can assign functions to variables and pass them to other functions. You'll learn more about how to use functions like this in chapter 7.

Functions have input and output, which you've seen already - when you use a function, you send it some data and then get back some more data as an answer. Some functions will do things themselves, while other functions will just return a value after performing some calculations. Below is a simple function which will add two numbers together.

```
def add_two_numbers(a, b):
    """ This function adds two numbers """
    return a + b
```

Let's look at the initial line of the function declaration. It starts with the reserved word `def`, followed by a name for your function, then the parameters that the function will expect within brackets. When you call the function later on in your program, you specify what these parameters are - they can be explicit values or variables.

The second line is called a *docstring*, and is another very useful way of making your programs easier to read when combined with good variable and function names. It should be a short description of the function and what it does - anything that someone might need to know in order to use the function properly. We've also used a special version of a Python string with three quotes, so that we can extend the docstring over more than one line if we need to.

The third line is where the function does its work. In this case it's easy - just add `a` and `b` together. The `return` statement tells Python that the function has finished, and to send the result of `a + b` back to whoever called it.

2.6.2 Variable scope

Python places some limits on functions, so that they can only affect a small part of your program, normally just the function itself. Most variables that are set inside your functions are known as *local variables*, and you won't be able to use them outside of the function.

```
def create_a():
    a = 42

create_a()
print a
```

When you try and run this program, you'll get an error like this one:

```
Traceback (most recent call last):
  File "<stdin>", line 5, in test.py
NameError: name 'a' is not defined
```

What happened? You set the `a` variable inside the `create_a()` function, didn't you? Actually, it was only created inside the function. You can think of it as

ACCORDING TO MY OLD PROFESSOR, THIS ONE'S BEEN HUNTING WUMPUSES IN THE ANDES FOR 20 YEARS...



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

'belonging' to `create_a`. As soon as Python has finished with a variable it gets thrown away, in this case as soon as the function exits.

As well as that, you also won't be able to change most variables which have been defined outside the function. Instead, when you create a variable you'll actually be creating a new one. The following code won't work:

```
a = 42
def add_to_a(b):
    a = a + b
add_to_a(42)
```

Python assumes that the `a` variable is supposed to be within the `add_one_to_a` function unless you tell it otherwise. Trying to access a variable inside of a function produces an error like this:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    File "<stdin>", line 2, in add_to_a
UnboundLocalError: local variable 'a' referenced before assignment
```

The rule of thumb to remember is that the variables used in functions and the variables used in the rest of your program are different. Within a function you should only use the variables that are passed into it as parameters, and once back in the main part of your program you should only use the variables which are returned from the function.

Like most rules of thumb though, there are exceptions. In our program we're making one exception, when we're modifying our list of caves. In Python, our lists of caves and cave networks are actually a special type of variable called an *object*, and behind the scenes we're sending messages to these objects instead of modifying them directly. We'll learn more about how that works in chapter 6, but for now just think of lists as being a special exception to the rule that you can't modify external variables.

2.6.3 Shared state

When functions (or objects) work on a single copy of something, it's referred to as *shared state*. We can make use of shared state by making our functions work on a list of caves, but generally shared state is a bad thing to have in your programs. If you have a bug in one of your functions, Python may just *corrupt* your data (perhaps truncate it, or replace it with something odd). You won't notice this until a completely separate part of your program tries to read the garbled data and displays odd results. When that happens, your program will become much harder to fix, depending on the number of functions which access your shared state.

NOTE Shared data is a two edged sword. You need to have some, but it's also a source of bugs - particularly if you have a lot of functions sharing the data.

In chapter 6, we'll learn how to limit the number of functions which have access to shared state, by using another Python structure, called a *class*. For now though, we'll just have to be careful, by only modifying our caves when we set them up, and leaving them alone once we're playing our game.

Data, and operations on data

Most programs can be thought of as a collection of information or data, and ways to interact with that data, and our Hunt the Wumpus program is no exception. We have a cave structure and locations for the wumpus and the player, functions which make changes to that data and then a main program which ties it all together using the functions.

Designing your programs this way makes them much easier to write and debug, and gives you more opportunities to reuse your code than if you had thrown everything into one big program or function.

If you have a *data structure* which fits everything that your program needs, and makes it easy to retrieve the data that you need, that's normally half the battle toward writing your program.

Now that you know what functions are and why you'd want to use them, let's go ahead and see how to break up our wumpus game into individual functions.

2.7 Fixing the wumpus

In principle encapsulating a program into functions isn't too hard - just look for parts of your program that fit some of the following criteria and try to pull them out into functions where they:

- do one particular thing (*self contained*)
- are repeated several times
- if the statements for a section are hard to understand

Thinking about our Hunt the Wumpus game, you should be able to see that it has three main sections to it. We'll start with the simplest functions first and then use them to build the rest of our program.

2.7.1 Interacting with the caves

When dealing with cave related tasks, there are several simple actions which we perform quite often:

- create a tunnel from one cave to another
- mark a cave as visited
- pick a cave at random, preferably one which is ok to dig a tunnel to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

To make our lives easier when working with the list of caves, we can create what are known as *convenience functions*. These are functions which perform a (potentially complicated) series of actions, but hide that complexity when you're using the function in your program. The benefit is that you can perform them in one step in your main program, and you don't have to worry about the details once you've created it. That makes your program easier to understand and helps to reduce bugs in your programs.

Listing 2.7 - Adding convenience functions

```
def create_tunnel(cave_from, cave_to):                      #1
    """ Create a tunnel between cave_from
    and cave_to """
    caves[cave_from].append(cave_to)
    caves[cave_to].append(cave_from)

def visit_cave(cave_number):                                #1
    """ Mark a cave as visited """
    visited_caves.append(cave_number)
    unvisited_caves.remove(cave_number)

def choose_cave(cave_list):                                 #2
    """ Pick a cave from a list, provided
    that the cave has less than 3 tunnels."""
    cave_number = choice(cave_list)
    while len(caves[cave_number]) >= 3:
        cave_number = choice(cave_list)
    return cave_number

def print_caves():                                         #3
    """ Print out the current cave structure """
    for number in cave_numbers:
        print number, ":", caves[number]
    print '-----'
#1 Creating tunnels and visiting caves
#2 Choosing a cave
#3 Printing caves
```

Creating tunnels and visiting caves are both obvious candidates for functions **(1)**. It's easy to make an error by using the wrong variable to refer to a cave, and using code like `create_tunnel(cave1, cave2)` makes your program much easier to read.

In the `choose_cave` function **(2)** we can hide even more detail. When we choose a cave, we're normally only interested in caves which have less than 3 tunnels. Adding that check into the function will remove a lot of duplicated code from our main program. Note also that `choose_cave` accepts a list of caves as input, so that we can use it to pick a cave from either the visited or unvisited cave list.

It's not just the 'final' versions of your code which can have convenience functions. You can also create them to help you out while programming, too. If you wanted to debug your code at a later point, a function to print all of your caves **(3)** comes in very handy.

Next let's turn our attention to how we create our caves.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

2.7.2 Creating the caves

We've already talked about the data that a program uses. One good rule of thumb is to create functions which do particular things to your data, or which tell you about your data, and then use only those functions to "talk" to your data. In programming terminology, this is normally referred to as an *interface*. With an interface to guide you, it's much harder to make a mistake, or get confused about what the data means. To some extent we've already started on that process

In Hunt the Wumpus, there are three tasks that we need to perform when creating our caves that are ideal candidates for functions:

- set up our cave list
- make sure all of our caves are linked
- make sure that there are three tunnels per cave

In listing 2.13, there are three functions which do exactly that. These functions are the essential core of our program, so it will pay off to try and get them right. There are no hard and fast rules, but some signs that your program is well written include:

- it's easy to read and understand
- it's easy to find and fix bugs
- you only have to change limited parts of your program when you add new features
- you can reuse some of your functions when modifying it

Ultimately though, what "right" means will vary from program to program depending on the design and what that design is trying to achieve.

Listing 2.8 - Cave creation functions

```
def setup_caves(cave_numbers):                                #1
    """ Create the starting list of caves """
    caves = []
    for cave in cave_numbers:
        caves.append([])
    return caves

def link_caves():                                              #2
    """ Make sure all of the caves are connected
    with two-way tunnels """
    while unvisited_caves != []:
        this_cave = choose_cave(unvisited_caves)
        next_cave = choose_cave(unvisited_caves)
        create_tunnel(this_cave, next_cave)
        visit_cave(next_cave)

def finish_caves():                                            #3
    """ Link the rest of the caves with
    one-way tunnels """
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

for cave in cave_numbers:
    while len(caves[cave]) < 3:
        passage_to = choose_cave(cave_numbers)
        caves[cave].append(passage_to)

#1 Creating the list of caves
#2 Connecting the caves
#3 Three tunnels per cave

```

Creating the list of caves (**1**) hasn't changed much from the previous listing, but it's still a good idea to put well defined sections of code in their own functions for readability.

All of the hard work of connecting the caves and tunneling is done in `link_caves` (**2**). Notice how the convenience functions which we defined in the previous listing help to tidy things up even further? Even if you didn't know what this function was doing, it'd be pretty easy to guess.

With `finish_caves` we haven't created a convenience function (**3**). It's the only section of code where we create a one-way tunnel, so the benefit is a bit more limited than in the other cases. Whether you create a function in cases like this might depend on whether you were planning on adding more functionality later on. Decisions like this can be something of a stylistic issue, so pick the option that feels best for you. You can always change it later if you need to repeat some code.

Finally, let's bring functions to how Hunt the Wumpus interacts with the player.

2.7.3 *Interacting with the player*

When running our program, there are two tasks which we perform regularly to find out what the player wants to do next:

- tell the player about where they are
- get some input from the player

Since the appearance of a program is likely to change substantially, either due to the feedback of the people using it or from adding new features, it often makes sense to keep the interface separated from the rest of the program, and interact with the player through well defined mechanisms.

Listing 2.9 - Player interaction functions

```

def print_location(player_location):
    """ Tell the player about where they are """
    print "You are in cave", player_location
    print "From here, you can see caves:"
    print caves[player_location]
    if wumpus_location in caves[player_location]:
        print "I smell a wumpus!"

def get_next_location():                      #1
    """ Get the player's next location """
    print "Which cave next?"

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

player_input = raw_input(">")
if (not player_input.isdigit() or
    int(player_input) not in
        caves[player_location]):
    print player_input + "?"
    print "That's not a direction that I can see!"
    return None
else:
    return int(player_input)

```

#1 Get next location

Here's the mechanism that I was talking about. It doesn't matter what the player enters, this function will always return either a special value of `None` (Python's version of `null`) if the input wasn't right, or the number of the cave that they want to enter. We can check this very easily in the main part of our program.

2.7.4 The rest of the program

Once you have all of these functions, it doesn't leave much of your program that isn't a function, but that's a good thing, as you'll see shortly.

Here's the final installment of our updated *Hunt the Wumpus* game. It behaves exactly the same way as the program in listing 2.6 as far as the player is concerned, but the structure has completely changed. All of our tasks are now stored within functions, and the main program uses those functions to do everything in the game - display the current cave, get input, moving the player and so on.

Listing 2.10 The 'refactored' wumpus game

```

from random import choice

...function definitions...

cave_numbers = range(0,20)
unvisited_caves = range(0,20)
visited_caves = [ ]
caves = setup_caves(cave_numbers)

visit_cave(0)
print_caves()
link_caves()
print_caves()
finish_caves()

wumpus_location = choice(cave_numbers)
player_location = choice(cave_numbers)
while player_location == wumpus_location:
    player_location = choice(cave_numbers)

while True:
    print_location(player_location)
    new_location = get_next_location()
    if new_location is not None:

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

player_location = new_location
if player_location == wumpus_location:
    print "Aargh! You got eaten by a wumpus!"
    break

```

The thing to notice is how short and easy to follow the main part of the program is now. It's only 20 lines, and since we've chosen useful function names you could probably figure out what it does even if you didn't know anything about Python. That's the ideal that you should be aiming for. Clear, easy to understand code will save you a lot of time when reading and modifying it later on.

Simplify You've seen how we refined and simplified our program as we went along, including going back and changing parts completely when necessary. If you can simplify your code, there's normally no reason not to. The simpler a program is, the easier it is to write, understand, debug and modify. The refining process is typically along the lines that you've seen so far in this chapter:

- use meaningful names for both variables and functions
- use white space to separate sections of program
- store values in intermediate variables
- break functions up so that they do one thing well
- limit the amount of shared state that functions use, and be clear about what that shared state is

"Perfection is achieved not when there is nothing left to add, but when there is nothing left to take away." -- Antoine de Saint-Exupery

Caves ... check. Wumpus ... check. Running around in the caves ... check. A way to win the game ... Hmm. No way to win the game. Better do something about that.

2.8 Bows and arrows

In the traditional wumpus game, you had a bow and one arrow, and when you thought that you knew which cave the wumpus was in, you could choose to fire an arrow into that cave. If you guessed wrong - too bad!



NOTE One of the golden rules of game design: **The player has to be able to enjoy your game.** Without a bow and arrow, you can still explore and have fun, but firing your bow and arrow is where you find out whether your exploration is correct.

It should be easy to see how to add this sort of feature by now, since it's similar in style to our `get_next_location()` function. We'll add a total of three more functions:

- ask whether the player wants to move or shoot
- find out where to move
- find out where to fire an arrow

We'll also modify our `get_next_location()` function into a general function `ask_for_cave()`. That's what it is already, and we can call it from both our movement and firing functions. By writing it this way, our two input functions will be short, which helps keep our program manageable. If you add another feature later which needs to ask for a cave, then you'll already have a useful function to call on, which makes programming easier and faster.

Listing 2.11 Adding arrows

```
def ask_for_cave():                                     #1
    """ Ask the player to choose a cave from
    their current_location. """
    player_input = raw_input("Which cave?")
    if (not player_input.isdigit() or
        int(player_input) not in caves[player_location]):
        print player_input + "?"
        print "That's not a direction that I can see!"
        return None
    else:
        return int(player_input)

def get_action():                                       #2
    """ Find out what the player wants to do next. """
    print "What do you do next?"
    print "  m) move"
    print "  a) fire an arrow"
    action = raw_input("> ")
    if action == "m" or action == "a":
        return action
    else:
        print action + "?"
        print "That's not an action that I know about"
        return None

def do_movement():                                      #3
    print "Moving..."
    new_location = ask_for_cave()
    if new_location is None:
        return player_location
    else:
        return new_location

def do_shooting():                                     #3
    print "Firing..."
    shoot_at = ask_for_cave()
    if shoot_at is None:
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        return False

    if shoot_at == wumpus_location:
        print "Twang ... Aargh! You shot the wumpus!"
        print "Well done, mighty wumpus hunter!"
    else:
        print "Twang ... clatter, clatter!"
        print "You wasted your arrow!"
        print "Empty handed, you begin the "
        print "long trek back to your village..."

    return True

    ...

while 1:
    print_location(player_location)

    action = get_action()
    if action is None:
        continue

    if action == "m":
        player_location = do_movement() #4
        if player_location == wumpus_location:
            print "Aargh! You got eaten by a wumpus!"
            break

    if action == "a":
        game_over = do_shooting() #4
        if game_over:
            break

#1 Redefine player_input()
#2 Get player action
#3 Functions for program actions, too
#4 The main program is clear

```

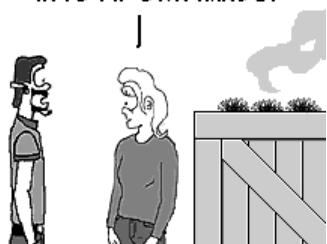
We don't need to make too many changes to our earlier `get_next_location` function - just a name change to make its intention clear and some cosmetic changes to how the program asks for input **(1)**. This is normally a good sign that a function is designed properly. If we had to significantly modify our function, it could be a sign that the original was trying to do too much at once.

`get_action()` **(2)** is quite similar to the `ask_for_cave()` function, except that the valid input differs. Hmm - perhaps there's the possibility that we can create a clearer function which both of these can call? In chapter 6, we'll learn about a good way to do that.

It's not just input that can be made into its own function. Actions within the game can be functions too **(3)**. Perhaps 'actions' is too strong a word - notice how the action functions

WHAT'S WITH THE CRATE?

I'M GETTING TIRED OF THIS WUMPUS SMELL.
SO I'M TAKING MATTERS INTO MY OWN HANDS.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

don't actually **do** anything (ie. set any variables), they just return what should happen, and then the main program takes action based on what the functions tell it to do.

The main part of our program is still just as clear as it was previously (**4**), even though we've just added a major new piece of functionality. If it's much more complicated, that's usually a sign that you might need to create a new function for some parts of your program and simplify the core of what you're doing.

2.9 More atmosphere

Congratulations! You now have a fully functional *Hunt the Wumpus* program, which you can play over and over again and use to impress your friends. Well, sort of. It works, but a number for each cave isn't very atmospheric or impressive. It makes your program easier to think about, but it needs that extra bit of polish. How about we change the program so that instead of numbers, it uses descriptive names for each cave?

NOTE The core game mechanics are what make Hunt the Wumpus fun, but the final bits of polish like this are what distinguish good games from **great** games.

One way to do that is to reference a list of cave names stored in your program based on the cave number. Instead of displaying the raw cave number, display `cave_names[cave_number]`. When you ask the player for a cave, they should instead pick a number from 1 to 3, with the name of the cave after the number. We're aiming for something like this:

Listing 2.12 An interface for Hunt the Wumpus

```
Black pit
From here, you can see:
  1 - Winding steps
  2 - Old firepit
  3 - Icy underground river
I smell a wumpus!
```

```
What do you do next?
  m) move
  a) fire an arrow
>
```

Well, the list of cave names is relatively easy. You can borrow mine, or create your own. Notice that we can break a list over multiple lines at the commas between items. This is to make the program easier to read and modify.



Listing 2.13 - A list of cave names

```
cave_names = [
    "Arched cavern",
    "Twisty passages",
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

    "Dripping cave",
    "Dusty crawlspace",
    "Underground lake",
    "Black pit",
    "Fallen cave",
    "Shallow pool",
    "Icy underground river",
    "Sandy hollow",
    "Old firepit",
    "Tree root cave",
    "Narrow ledge",
    "Winding steps",
    "Echoing chamber",
    "Musty cave",
    "Gloomy cave",
    "Low ceilinged cave",
    "Wumpus lair",
    "Spooky Chasm",
]

```

The only other changes that we need to make are to what we're displaying, and what input we'll accept.

Listing 2.14 Hunt the Wumpus - now with 40% more atmosphere!

```

def print_location(player_location):
    """ Tell the player about where they are """
    print
    print cave_names[player_location]                                #1
    print "From here, you can see:"                                     #1
    neighbors = caves[player_location]                                 #1
    for tunnel in range(0,3):                                         #1
        next_cave = neighbors[tunnel]                                  #1
        print " ", tunnel+1, "-", cave_names[next_cave]             #1
    if wumpus_location in neighbors:
        print "I smell a wumpus!"                                     #1

def ask_for_cave():
    """ Ask the player to choose a cave from
    their current_location."""
    player_input = raw_input("Which cave? ")
    if player_input in ['1', '2', '3']:                               #2
        index = int(player_input) - 1                                #2
        neighbors = caves[player_location]                            #2
        cave_number = neighbors[index]                             #2
        return cave_number                                         #3
    else:
        print player_input + "?"
        print "That's not a direction that I can see!"
        return False

#1 Changes to the player's view
#2 Simplify player input
#3 Nothing else has changed

```

Here's where we print out the current caves, and the list of caves that the player can see (1). They're all using the 'printable' cave name from our list of cave names, rather than the number. Instead of just printing our cave list, we're using a for loop, with tunnel as an index into the list of tunnels. We're also adding one to it to get 1, 2 or 3 rather than the 0, 1 or 2 indexes, to make it extra friendly.



Now that we know there are only three valid choices, we can just check directly for those (2) rather than needing the user to enter the number of the cave. We're also subtracting one from the result, since we need 0, 1 or 2 for our list index, rather than 1, 2 or 3.

Even though we're using 1, 2 and 3 as choices, we still return the cave number as an index. All of our changes are contained within the `print_location` and `ask_for_cave` functions and use the interface that we talked about earlier, so nothing else in our program needs to be changed at all (3).

2.10 Where to from here?

You don't have to stop with the program as listed. There are a number of features you can add, including some which were in the original version of Hunt the Wumpus. Feel free to invent your own - this is your program now, and you can make it do whatever you like.

2.10.1 Bats and Pits

In the original Hunt the Wumpus, there were other hazards: bats, which carried the player to another cave, and pits - which worked in a similar way to the wumpus ("I feel a draft!").

2.10.2 Make the wumpus move

One wumpus variant made the wumpus move to a different random cave if the player missed with their arrow, instead of losing the game.

2.10.3 Different cave structures

The original Hunt the Wumpus had a static cave structure, in which the caves were vertices of a dodecahedron. You don't necessarily have to follow their lead, but experimenting with different cave structures could make for a more fun game. For example, perhaps you don't like one-way tunnels? That should be easy to fix. Also, in the current version caves can tunnel to themselves. I happen to like that sort of layout, but you may not. Being able to write your own programs means that you're not stuck with my design choices - you're free to make your own.

2.11 Summary

There was a lot to cover in this chapter. Not only did you learn about most of the basics of Python and how you can fit them together to make a program, we also covered some ideas about how to design your programs, and why certain design choices might be good or bad.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

The best way to **start** writing a program is to choose something simple that does part of what you need it to, or which describes the core of your program and build from there. In our case, that was the initial game loop of choosing a cave and allowing the player to move to a different one. From there we were able to develop a proper cave system, make sure that it was connected properly, then turn it into a fully fledged game with a which can be played and won (or lost).

The best way to **continue** to develop your program is to refine it as you go, breaking commonly used parts into functions and trying to develop an *interface* between different sections of your program. It's very easy to lose track of the overall structure in low-level details such as adding items to lists or making sure that caves have three tunnels, so a large part of your interfaces will often be hiding fiddly details, or making sections of your program easier to work with.

3

Interacting with the world

One of the key strengths of Python is its standard library, a large suite of program code which is installed along with Python and which covers things like finding and iterating over files, handling user input, downloading and parsing pages from the web and accessing databases. If you make good use of the standard library, you can often write programs in a fraction of the time that it would take you otherwise, with less typing and far fewer bugs.

Guido's Time Machine The standard library is so extensive that one of the running jokes in the Python community is that Guido owns a time machine. When someone asks for a module that performs a particular task, Guido just hops in his time machine, travels back to the beginning of Python and "poof!" it's already there.

In chapter 2 you used the choice function in Python's random module to pick something from a list, so you've already used a library. In this chapter we'll go in depth and find out more about how to use them, what other libraries exist and how to use Python's documentation to find out more about them. In the process, we'll also pick up a few other missing pieces of Python, such as how we can read files, and another of Python's data types - the dictionary.

The program in this chapter solves a common problem that you've probably faced before - you have two similar directories (perhaps one's a backup of your holiday photos) and you'd like to know which files differ between the two of them. We'll be tackling our program from a different angle than chapter 2 though. Rather than write most of our own code, we'll be using Python to glue together several standard libraries to get the job done.

Let's start by learning more about Python libraries.

3.1 “Batteries Included”: Python’s libraries

What are libraries used for? Normally they’ll be geared towards a single purpose, such as sending data via a network, writing CSV or Excel files, displaying graphics or handling user input, although they can grow to cover a large number of related functions - there’s no hard or fast rule.

Definition A library is program code which is written so that it can be used by other programs.

Python libraries can do anything that Python can, and more. In some (rare) cases like intensive number crunching or graphics processing, Python can be too slow to do what you need, but it’s possible to extend Python to use libraries written in C.

In this section we’ll learn about Python’s standard library, see which other libraries we can add, try them out, and get a handle on exploring a single library.

3.1.1 Python’s standard library

Python installs with a large number of libraries that cover most of the common tasks that you’ll need to handle when programming.

If you find yourself facing a tricky problem, it’s a good habit to read through the modules in Python’s standard library to see if there’s something that covers what you need to do. The Python manuals are installed with the standard Windows installer, and there’s normally a documentation package when installing under Linux. The latest versions are also available at docs.python.org if you’re connected to the internet. Being able to use a good library can save you hours of programming, so five or ten minutes up front can pay big dividends.

The Python standard library is large enough that it can be hard to find what you need. Another way to learn it is to take it one piece at a time. The Python Module of the Week blog (<http://www.doughellmann.com/PyMOTW/>) covers most of Python’s standard library and is an excellent way to familiarize yourself with what’s available, since it often contains far more explanation than the standard Python documentation.

3.1.2 Other libraries

You’re not limited to the libraries that Python installs. It’s very easy to download and install extra libraries to add extra functionality that you need. Most add-on libraries come with their own installers or installation script - those that don’t can normally just be copied into the library folder of your Python directory. You’ll find out how to install libraries in later chapters when we’ll need them. Once the extra libraries are installed, they behave just like Python’s built in ones - there’s no special syntax that you need to know.



© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

3.1.3 Using libraries

Once installed, using a library is very straightforward. Just add an import line at the top of the script. There are several ways to do it, here are the three most common.

INCLUDE EVERYTHING

You can include everything from a library into your script by using a line like:

```
from os import *
```

This will read everything from the os module, and drop it straight into your script. If you want to use the access function from os then you can just use it directly, like `access("myfile.txt")`. This has the advantage of saving some typing, but with serious downsides:

- you now have a lot of strange functions in your script
- worse, if you include more than one module in this way then you run the risk of functions in the later module overwriting the functions from the first module - Ouch!
- finally, it's much harder for you to remember which module a particular function came from, which makes your program difficult to maintain

Fortunately there are much better ways to import modules.

INCLUDE JUST THE MODULE

A better way to handle things is with a line like `import os`. This will import everything in os, but make it available only through an os object. Now if you want to use the access function you need to use it like this: `os.access("myfile.txt")`. It's a bit more typing, but without running the risk of overwriting any other functions.

IT'S AN IMPORTANT REPORT
THAT I'M WORKING ON. I HAD
IT SAVED ON THE SERVER
YESTERDAY, BUT NOW IT'S
GONE!

/ I'LL HAVE A LOOK IN
YOUR SHARED
FOLDER...



INCLUDE ONLY THE BITS THAT YOU WANT

If you're using the functions from a module a lot, you might find that your code becomes a bit hard to read, particularly if the module has a long name. There's a third option in this case - you can use a line like `from os import access`. This will import directly so that you can use `access("myfile.txt")` without the module name, but only include the access function, not the entire os module. You still run the risk of overwriting with a later module, but since you have to specify the functions and there are fewer of them, it's much less likely.

3.1.4 What's in a library, anyway?

Libraries can include anything that's included in standard Python - variables, functions and classes, as well as Python code that should be run when the library is loaded. You're not limited in any way - anything that's legal in Python is fine to put in a library. When using a

library for the first time, it helps to know what's in it, and what it does. There are two main ways to find out.

TIP `dir` and `help` aren't just useful for libraries. You can try it out on all of the Python objects such as classes and functions. It even supports strings and numbers.

READ THE FINE MANUAL

Python comes with a detailed manual on every aspect of its use, syntax, standard libraries, pretty much everything you might need to reference when writing programs. It doesn't cover every possible use, but the majority of the standard library is there. If you have internet access, you can view it at <http://docs.python.org/>, and it's normally installed alongside Python too.

EXPLORATION

One useful function for finding out what a library contains is `dir()`. You can call it on any object to find out what methods it supports, but it's particularly useful with libraries. You can combine it with the `__doc__` special variable, which is set to the docstring defined for a function or method, to get a quick overview of a library or classes' methods and what they do. In fact, this combination is so useful that there's a shortcut called `help()` which is defined as one of Python's built in function.

For the details you're often better off looking at the documentation, but if you only need to jog your memory, or if the documentation is patchy or confusing, `dir`, `__doc__` and `help` are much faster. Here's an example of looking up some information about the `os` library.



Listing 3.1 Finding out more about the `os.path` library

```
>>> import os
>>> dir(os.path)
['__all__', '__builtins__', '__doc__', '__file__',
 '__name__', '__package__', '__getfullpathname',
 'abspath', 'altsep', 'basename', 'commonprefix',
 'curdir', 'defpath', 'devnull', 'dirname', 'exists',
 'expanduser', 'expandvars', 'extsep', 'genericpath',
 'getatime', 'getctime', 'getmtime', 'getsize',
 'isabs', '.isdir', '.isfile', 'islink', 'ismount',
 'join', 'lexists', 'normcase', 'normpath', 'os',
 'pardir', 'pathsep', 'realpath', 'relpath', 'sep',
 'split', 'splitdrive', 'splitext', 'splitunc',
 'stat', 'supports_unicode_filenames', 'sys',
 'walk', 'warnings']
>>> print os.path.__doc__
Common pathname manipulations, WindowsNT/95 version.
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

Instead of importing this module directly, import os           #3
and refer to this module as os.path.                      #3
                                                       #3

>>> print os.path.isdir.__doc__                         #4
Return true if the pathname refers to an existing        #4
directory.                                              #4
>>> print os.path.isdir('c:/')                          #5
True                                                    #5
>>> print os.path.isdir('c:/windows/system.ini')      #5
False                                                   #5

>>> help (os)                                         #6
Help on module os:                                     #6
                                                       #6

NAME                                                 #6
    os - OS routines for Mac, NT, or Posix depending   #6
    on what system we're on.                           #6
                                                       #6

FILE                                               #6
    c:\python26\lib\os.py                            #6
                                                       #6

DESCRIPTION                                         #6
    This exports:                                    #6
        - all functions from posix, nt, os2, or ce,    #6
          e.g. unlink, stat, etc.                     #6
        - os.path is one of the modules posixpath,    #6
          or ntpath                                #6
        - os.name is 'posix', 'nt', 'os2', 'ce' or     #6
          'riscos'                                #6
        - os.curdir is a string representing the     #6
          current directory ('.' or ':')            #6
        - os.pardir is a string representing the     #6
          parent directory ('..' or '::')          #6

#1 import os
#2 explore os.path
#3 docstring for os.path module
#4 docstring for the isdir function
#5 testing the functions
#6 the help() function

```

First, we need to import the `os` module (**1**). We can just import `os.path` directly, but this is the way that it's normally done, so we'll have fewer surprises later. Next we call the `dir()` function on `os.path`, to see what's in it (**2**). It'll return a big list of function and variable names, including some built-in Python ones, like `__doc__` and `__name__`.

Since we can see a `__doc__` variable in `os.path`, let's print it and see what it contains (**3**). It's a general description of the `os.path` module and how it's supposed to be used.

If we look at the `__doc__` variable for a function in `os.path` (**4**), it shows us much the same thing - a short description of what the function is supposed to do.

Once you've found a function which you think does what you need, you can try it out to make sure (**5**). Here we're calling `os.path.isdir()` on a couple of different files and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

directories to see what it returns. For more complicated libraries, you might find it easier to write a short program rather than type it all in at the command line.

Finally, the output of the `help()` function (6) contains all of the same information that `__doc__` and `dir()` do, but printed nicely. It also looks through the whole object and returns all of its variables and methods without you having to look for them.

In practice, it can often take a mixture of these methods before you figure out enough of a library to be useful. A quick overview of the library documentation, followed by some experimenting at the command line and a further read of the documentation once you understand how it fits together to pick up some of the finer points. Also bear in mind that you don't necessarily have to understand the whole of the library in one go if you can just pick off the pieces that you need.

Now that you know the basics of Python libraries, let's see what we can do with them.

3.2 Another way to ask questions

There's one thing that we need to know before we can start putting our program together. Well, there are a couple of other things, but we can pick those up on the way. What we'd like to be able to do to start with is tell the computer which directories we want to compare. If this were a normal program we'd probably have a graphical interface where we could click on the relevant directories, but that sounds hard, so we'll pick something simpler to write - a command line interface.



3.2.1 Using command line arguments

Command line arguments are often used in system-level programs. When you run a program from the command line, you can specify additional parameters by typing them after the program's name. In our case we'll be typing in the names of the two directories that we want to compare - something like:

```
python difference.py directory1 directory2
```

If you have spaces in your directory names, you can surround the parameters with quotation marks, otherwise your operating system will interpret it as two different parameters:

```
python difference.py "My Documents\directory1" "My Documents\directory2"
```

Now that we have our parameters, what are we going to do with them?

3.2.2 Using the sys module

In order to read the parameters that you've fed in, we'll need to use the `sys` module which comes with Python's standard library. `sys` deals with all sorts of system-related functionality, such as finding out which version of Python a script is running on, information about the script, paths, and so on. We'll be making use of `sys.argv`, which is an array

containing the script's name and any parameters that it was called with. Our initial program is listing 3.2, which will be the starting point for our comparison script.

Listing 3.2 - Reading parameters using sys

```
import sys

if len(sys.argv) < 3:                                     #1
    print "You need to specify two directories:"          #1
    print sys.argv[0], "<directory 1> <directory 2>"    #1
    sys.exit()                                            #1

directory1 = sys.argv[1]                                    #2
directory2 = sys.argv[2]                                    #2

print "Comparing:"                                       #3
print directory1                                         #3
print directory2                                         #3
print

#1 checking the parameters
#2 storing the parameter values
#3 debugging strings
```

First we check to make sure that the script has been called with enough parameters **(1)**. If there are too few, then we return an error to the user. Note also that we're using `sys.argv[0]` to find out what the name of our script is, and `sys.exit` to end our program early.

Since we know now that there are at least two other values, we can store them for later use **(2)**. We could use `sys.argv` directly, but this way we've got a nice variable name, which makes our program easier to understand.

Once we have our variables, we can print them out **(3)** to make sure they're what we're expecting. You can test it out by trying the commands from the previous. The script should respond back with whatever you've specified.

NOTE File objects are an important part of Python. Quite a few libraries will use file-like objects to access other things, like web pages, strings or the output returned from other programs.

If you're happy with it the results, it's time to start building our program in the next section.

3.3 *Reading and Writing Files*

The next thing that we'll need to do in our duplicate checker is to find our files and directories and open them to see if they're the same. Python has built-in support for handling files as well as very good cross platform file and directory support via the `os` module. We'll be using both of these in our program.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

3.3.1 Paths and directories (a.k.a., Dude, where's my file?)

Before we open our file though, we need to know where to find it. Actually, we want to find all of the files in a directory and open them, as well as any files in directories within that directory, and so on. That's pretty tricky if you're writing it yourself, but fortunately the `os` module has a function called `walk` which does exactly what we want. The `os.walk()` function returns a list of all of the directories and files for a path. If you append listing 3.3 to the end of listing 3.2, it will call `os.walk()` on the directories that you've specified.

Listing 3.3 - using os.walk()

```
import os

for directory in [directory1, directory2]:                      #1
    if not os.access(directory, os.F_OK):                         #2
        print directory, "is not a valid directory!"             #2
        sys.exit()                                                 #2

    print "Directory", directory                                    #3
    for item in os.walk(directory):                                #3
        print item                                              #3
    print                                         #3

#1 Don't repeat yourself
#2 Input checking
#3 Walking over the directory
```

We're going to be doing the same thing for both `directory1` and `directory2` (1). We could repeat our code over again for `directory2`, but then if we want to change it later, we'll have to change it in two places. Worse, we could accidentally change one but not the other, or change it slightly differently. A better way is to use the directory names in a `for` loop like this, so that we can reuse the code within the loop.

It's good idea to check the input that your script's been given (2). If there's something amiss, then exit with a reasonable error message to let the user know what's gone wrong.

(3) is the part where we walk over the directory. For now we're just printing the raw output that's returned from `os.walk()`, but in a minute we'll actually do something with it.

I've set up two test directories on my computer, with a few directories that I found lying around. It's probably a good idea for you to do the same, so that you can test your program and know you're making progress.

If you run the program so far, you should see something like the following output.



```
D:\code>python difference_engine_2_os.py . test1 test2
Comparing:
test1
test2

Directory test1
('C:\\\\test1', ['31123', 'My Music', 'My Pictures', 'test'], [])
('C:\\\\test1\\\\31123', [], [])
('C:\\\\test1\\\\My Music', [], ['Desktop.ini', 'Sample Music.lnk'])
('C:\\\\test1\\\\My Pictures', [], ['Sample Pictures.lnk'])
('C:\\\\test1\\\\test', [], ['foo1.py', 'foo1.pyc', 'foo2.py', 'foo2.pyc',
'os.walk.py', 'test.py'])

Directory test2
('C:\\\\test2', ['31123', 'My Music', 'My Pictures', 'test'], [])
('C:\\\\test2\\\\31123', [], [])
('C:\\\\test2\\\\My Music', [], ['Desktop.ini', 'Sample Music.lnk'])
('C:\\\\test2\\\\My Pictures', [], ['Sample Pictures.lnk'])
('C:\\\\test2\\\\test', [], ['foo1.py', 'foo1.pyc', 'foo2.py', 'foo2.pyc',
'os.walk.py', 'test.py'])
```

In Python strings, some special characters can be created by using a backslash in front of another character. If you want a tab character, for example, you can put `\t` into your string. When Python prints it, it'll get replaced with a literal tab character. If you actually do want a backslash though, like we do here, then you'll need to use two backslashes, one after the other.

The output for each line gives you the name of a directory within your path, then a list of directories within that directory, then a list of the files. Very handy, and definitely beats writing your own version.

3.3.2 Paths

If you want to use a file or directory, you'll need what's called a path. A path is a string which gives the exact location of a file, including any directories which contain it. For example, the path to Python on my computer is `C:\\python26\\python.exe`, which looks like `"C:\\\\python26\\\\python.exe"` when expressed as a Python string.

If you wanted a path for `foo2.py` in the last line of the listing above, you can use `os.path.join('C:\\\\test2\\\\test', 'foo2.py')`, to get a path which looks like `'C:\\\\test2\\\\test\\\\foo2.py'`. We'll see more of the details of this when we start putting our program together in a minute.



I'M SURE IT'S JUST
YOUR IMAGIN... AH.

TIP One thing to bear in mind when using paths is that the separator will be different depending on which platform you're using. Windows uses a backslash (`\`) character, and Linux and Macintosh use a forward slash (`/`). To make sure that your programs work on all three systems, it's a good idea to get in the habit of using

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

the `os.path.join()` function, which takes a list of strings and joins them with whatever the path separator is on the current computer.

Once you have the location of your file, the next step is opening it.

3.3.3 *File, open!*

To open a file in Python, you can use the `file` or `open` built in functions. They're exactly the same behind the scenes, so it doesn't matter which one you use. If the file exists and you can open it, you'll get back a file object, which you can read using the `read()` or `readlines()` methods. The only difference between `read` and `readlines` is that `readlines` will split the file up into a list of strings, but `read` will return the file as one big string. The code below shows how we can open a file and read its contents.

```
read_file = file(os.path.join("c:\\test1\\test", "foo2.py"))
file_contents = list(read_file.readlines())
print "Read in", len(file_contents), "lines from foo2.py"
print "The first line reads:", file_contents[0]
```

First we create a path using `os.path.join` and then use it to open the file at that location. You'll want to put in the path to a text file that exists on your computer. `read_file` will now be a file object, so we can use the `readlines()` method to read the entire contents of the file. We're also turning the file contents into a list using the `list()` function. We don't normally treat files like this, but it helps to show you what's going on. `file_contents` is a list now, so we can just use the `len()` function to see how many lines it has, and print the first line by using an index of 0.

Although we won't be using it in our program, it's also possible to write text into a file as well as read from it. To do this, you'll need to open the file with a write mode instead of the default read-only mode, and use the `write()` or `writelines()` of the file object. Here's a quick example:

```
write_file = file("C:\\test2\\test\\write_file.txt", "w")           #1
write_file.write("This is the first line of the file\\n")          #2
write_file.writelines([
    "and the second\\n",
    "and the third!\\n"])
write_file.close()                                                 #4
#1 Open the file
#2 Write one line
#3 Write multiple lines
#4 Close the file
```

We're using the same `file` function that we used before, but here we're feeding it an extra parameter, the string "`w`", to tell Python that we want to open it for writing (**1**).

Once we have the file object back, we can write to it by using the `.write()` method, with the string that we want to write as a parameter **(2)**. The "`\n`" at the end is the special character for a new line - without it, all of our output would be on one line. We can also write multiple lines at once, by putting them into a list and using the `.writelines()` method instead **(3)**.

Once we're done with a file, it's normally a good idea to close it **(4)**, particularly if we're writing to it. Files can sometimes be buffered, which means that they're not written onto the disk straight away - if your computer crashes, it might not be saved.

That's not all that you can do with files, but it's enough to get us started. For our difference engine we won't even need to write files, but it'll help for future programs. For now, let's turn our attention to the last major feature we'd like to add to our program.

3.4 Comparing files

We're almost there, but there's one last hurdle. When we're running our program, we need to know whether we've seen a particular file in the other directory, and if so, whether it has the same content too. We could read all of the files in and compare their content line by line, but what if you have a very large directory with big images? That's a lot of storage, which means that Python is likely to run very slowly.

NOTE It's often important to consider how fast your program will run, or how much data it will need to store, particularly if the problem that you're working on is "open ended", that is, if it might be run on a large amount of data.

3.4.1 Fingerprinting a file

Fortunately, there's another library to help us, called `hashlib`, which is used to generate a hash for a particular piece of data. A hash is like a fingerprint for a file - from the data that it's given it will generate a list of numbers and letters which is virtually guaranteed to be unique for that data. If even a small part of the file changes, the hash will be completely different and we'll be able to detect the change. Best of all, the hashes are relatively small, so they won't take up much space. Here's a small script which shows how you might generate a hash for one file.

Listing 3.4 Generating a hash for a file

```
import hashlib
import sys

file_name = sys.argv[1]
read_file = file(file_name)          #1
the_hash = hashlib.md5()            #2
for line in read_file.readlines():   #3
    the_hash.update(line)
print the_hash.hexdigest()          #4

#1 Open the file
#2 Create a hash object
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

#3 Update the hash
#4 Print a digest

After importing our libraries, we read a file name from the command line and open it (1). Next we create a hash object here (2), which will handle all of the hash generation. I'm using md5, but there are many others in hashlib.

Now that we have an open file and a hash object, we feed each line of the file into the hash with the update() method (3).

Once we've fed all of the lines into the hash, we can get the final hash in 'hexdigest' form (4). It uses only numbers and the letters a-f, so it's easy to display on screen or paste into an email.

An easy way to test the script is to run it on itself. After you've run it once, try making a minor change to the script, such as adding an extra blank line at the end of the file. If you run the script again, the output should be completely different.

Here I'm running the hash generating script on itself. For the same content, it'll always generate the same output.

```
D:\test>python hash.py hash.py
df16fd6453cedecdea3dddca83d070d4
D:\test>python hash.py hash.py
df16fd6453cedecdea3dddca83d070d4
```

Here are the results of adding one blank line to the end of the hash.py file. It's a minor change (most people wouldn't notice it), but now the hash is completely different:

```
D:\test>hash.py hash.py
47eeac6e2f3e676933e88f096e457911
```

Now that our hashes are working, let's see how we can use them in our program.



3.4.2 Mugshots: storing your files' fingerprints in a dictionary

Now that we can generate a hash for any given file, we just need somewhere to put it. One option is to put the hashes into a list, but searching over a list every time that you want to find a particular file is going to be very slow, particularly if you have a large directory with lots of files.

There's a better way to do it, by using Python's other main data type - the dictionary.

You can think of dictionaries as a bag of data. You put data in, give it a name and then later when you want the data back you give the dictionary its name it will return the data. In Python's terminology, the name is called a key and the data is called the value for that key. Let's see how we use a dictionary.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Listing 3.5 How to use a dictionary

```

test_dictionary = {}
test_dictionary = {'one' : 1, 'two' : 2}
test_dictionary = {
    'list' : [1,2,3],                                     #1
    'dict' : {'one' : 1, 'two' : 2},                      #1
}
print test_dictionary['list']                           #2
del test_dictionary['list']                           #3
print test_dictionary.keys()                         #4
print test_dictionary.values()                       #4
print test_dictionary.items()                         #4

#1 Put anything you like in a dictionary
#2 Access values
#3 Remove values
#4 Useful dictionary methods

```

Dictionaries are fairly similar to lists, except that you use curly braces instead of square ones, and you separate keys and their values with a colon.

The other similarity to lists is that you can put anything that you like as a value **(1)**, including lists, dictionaries and other objects. You're not limited to storing just simple types like strings or numbers, or just one type of thing. The only constraint is on the key - it can only be something which isn't modifiable, like a string or number.

To get your value back once you've put it in the dictionary, just use the dictionary's name with the key after it in square brackets **(2)**. If you're finished with a value, it's easy to remove it by using `del` followed by the dictionary and the key that you want to delete **(3)**.

Dictionaries are objects, so they have some useful methods **(4)** as well as direct access. `keys` returns all of the keys in a dictionary, `values` will return its values and `items` returns both the keys and values. Typically you'll use it in a for loop, like this: `for key, value in test_dictionary.items():`

...

When deciding what keys and values to use for a dictionary, the best option is to use something unique for the key, and the data that you'll need in your program as the value. You might need to convert the data somehow when building your dictionary, but it normally makes your code easier to write and easier to understand. For our dictionary, we'll use the path to the file as the key, and the checksum that we've generated as the value.

Now that you know about hashes and dictionaries, let's put our program together.



3.5 Sticking it all together

"Measure twice, cut once" is an old adage that often holds true. When programming, you always have your undo key, but you can't undo the time that you've spent writing the code you've just thrown away.

It often helps when developing to have some sort of plan in place as to how you'll proceed. Your plan doesn't have to be terribly detailed, but it can help you to avoid potential roadblocks or trouble spots if you can foresee them. Now that we think we have all of the parts that we'll need, let's plan out the overall design of our program at a high level. It should go something like this:

- Read in and sanity check the directories that we want to compare
- Build a dictionary containing all of the files in the first directory
- For each file in the second directory, compare it to the same file in the first dictionary.

That seems pretty straightforward. As well as this overall structure, it can help to think about the four different possibilities for each file:

3.5.1 Case	The file doesn't exist in directory 2	3.5.2 Case	The file exists, but is different in each directory
1		2	
3.5.3 Case	The files are identical in both	3.5.4 Case	The file exists in directory 2, but not our first directory.
3		4	

Figure 3.1 The four possibilities for differences between files.

Given this rough approach, there are a couple of issues that should stand out. Firstly, our initial plan of building all of the checksums straight away may not be such a good idea after all. If the file isn't in the second directory, then we'll have gone to all of the trouble of building a checksum that we'll never use. For small files and directories it might not make much difference, but for larger ones (eg. photos from a digital camera or mp3s) the extra time might be significant. The solution to this is to put a placeholder into the dictionary that we build, and only generate the checksum once we know we have both files.

Can't we just use a list? If we're just putting a placeholder into our dictionary instead of a checksum, you'd normally just start by using a list. Looking up a value in a dictionary is typically much faster though, since for large lists Python needs to check each value in turn, whereas a dictionary just needs a single lookup. Another good reason is that dictionaries are more flexible and easier to use than lists if you're comparing independent objects.

Secondly, what happens if a file is in the first directory but not the second? Given the rough plan above, we're only comparing the second directory to the first one, not vice-versa. We won't notice a file if it's not in the second directory. One solution to this is to delete the files from the directory as we compare them. Once we've finished the comparisons, we know that anything that's left over is missing from the second directory.



Planning like this can take time, but it's often faster to spend a little time up front working out potential problems. What's better to throw away when you change your mind? Five minutes of design or half an hour of writing code? Listings 3.6 and 3.7 show the last two parts of our program based on the updated plan. You can join these together with listings 3.2 and 3.3 to get a working program.

Listing 3.6 - Utility functions for our difference program

```
import hashlib

def md5(file_path):                                     #1
    """Return an md5 checksum for a file."""           #1
    read_file = file(file_path)                         #1
    the_hash = hashlib.md5()                           #1
    for line in read_file.readlines():                  #1
        the_hash.update(line)                          #1
    return the_hash.hexdigest()                        #1

def directory_listing(dir_name):                      #2
    """Return all of the files in a directory."""      #2
    dir_file_list = {}                                #3
    dir_root = None                                  #3
    dir_trim = 0                                     #3
    for path, dirs, files in os.walk(dir_name):       #3
        if dir_root is None:                          #3
            dir_root = path                         #3
            dir_trim = len(dir_root)                 #3
            print "dir", dir_name,                   #3
            print "root is", dir_root               #3
        trimmed_path = path[dir_trim:]              #4
        if trimmed_path.startswith(os.path.sep):     #4
            trimmed_path = trimmed_path[1:]         #4
        for each_file in files:                     #4
            file_path = os.path.join(                #4
                trimmed_path, each_file)             #4
            dir_file_list[file_path] = True          #4
    return (dir_file_list, dir_root)                  #5

#1 - MD5 function
#2 - Directory listing function
#3 - Finding the 'root' of a directory
#4 - Building our dictionary of files
#5 - Returning multiple values
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

This is just our program from listing 3.5, rolled up into a function. Notice how I've added a docstring as the second line (1), so that it's easy to remember what the function does.

Since we'll be building a list of files for two directories, it makes sense to have a function which returns all of the information that we need about a directory (2), so that we can reuse it each time. The two things which we need are the 'root', or lowest level directory (the one we typed in at the command line), and a list of all of the files relative to that root so that we can compare the two directories easily. For example, C:\test\test_dir\file.txt and C:\test2\test_dir\file.txt should both be entered into their respective dictionaries as \test_dir\file.txt.

Since os.walk starts at the root of a directory by default, all we need to do is remember the first directory that it returns (3). We do that by setting dir_root to None before we enter the for loop. None is a special value in Python which means 'not set' or 'value unknown'. It's what you use if you need to define a variable but don't know its value yet. Inside the loop, if the dir_root is None, we know that it's the first time through the loop and we have to set it. We're setting a dir_trim variable too, so that later we can easily trim the first part of each directory that's returned.

Once we have our directory root, we can chop off the common part of our directories and path separators from the front of the path returned by os.walk (4). We do that by using string slices, which will return a subsection of a string. It works in exactly the same way as a list index, so it starts at 0, and can go up to the length of the string.

Once we're done, we return both the directory listing, and the root of the directory (5) using a special Python data type called a tuple. Tuples are similar to lists, except that they're immutable - you can't change them once they've been created.

Now that we've checked our inputs and set up all of our program's data, we can start making use of it. Just like in chapter 2 when we simplified Hunt the Wumpus, the part of our program that actually does stuff is fairly short, clear and easy to understand. All of the tricky details have been hidden away inside functions.



Listing 3.7 - Finding the differences between directories

```
dir1_file_list, dir1_root = directory_listing(directory1)      #1
dir2_file_list, dir2_root = directory_listing(directory2)      #1

for file_path in dir2_list.keys():
    if file_path not in dir1_file_list:                         #2
        print file_path, "not found in directory 1"             #2
    else:
        print file_path, "found in directory 1 and 2"          #3
        file1 = os.path.join(dir1_root, file_path)                #3
        file2 = os.path.join(dir2_root, file_path)                #3
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

if md5(file1) != md5(file2):          #3
    print file1, "and", file2, "differ!" #3
    del dir1_file_list[file_path]       #4

for key, value in dir1_file_list.items(): #4
    print key, "not found in directory 2" #4
#1 - Using our directory functions
#2 - Files not in directory 1
#3 - Comparing checksums
#4 - Files not in directory 2

```

To assign both of the variables that we get back from our function, we separate them with a comma **(1)**. You've already seen this when using `dictionary.items()` in a for loop.

Here's the first comparison **(2)** - if the file isn't in directory 1, then we warn the user. We can just use `in` in the same way that we would for a list. Python will automatically assume that we want to check the keys.

If the file exists in both directories, then we build a checksum for each file and compare them **(3)**. If they're different, then we know that the files are different and we again warn the user. If the checksums are the same then we keep quiet, since we don't want to overwhelm people with screens and screens of output - they just want to know the differences.

Once we've compared the files in section 3, we delete them from the dictionary. Any that are left over, we know aren't in directory 2 and we tell the user about them **(4)**.

That seems to just about do it for our program, but are you sure that it's working? Time to test it out.

**NO NEED TO PANIC,
CONTINUE ABOUT YOUR
BUSINESS CITIZENS.**



3.6 Testing your program

If you haven't already, now's probably a good time to create some test directories so that you can try out your script and make sure that it's working. It's especially important as you start working on problems that have real world consequences. For example, if you're backing up some family photos and your program doesn't report that a file has changed (or doesn't exist), you won't know to back it up and might lose it if your hard drive crashes. Or it might report two files as the same when they're actually different.

You can test your script on directories that you already have, but specific test directories are a good idea, mainly because you can exercise all of the features that you're expecting. At a minimum, I'd suggest:

- Adding at least two directory levels, to make sure that paths are handled properly
- Creating a directory with at least one space in it
- Using both text and binary files (eg. Images)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

- Setting up all of the cases that you're expecting (files missing, file differences, files that are the same)
- By thinking about all of the possible cases, you can catch bugs in your program before you run it over a real directory and miss something or worse, lose important data.

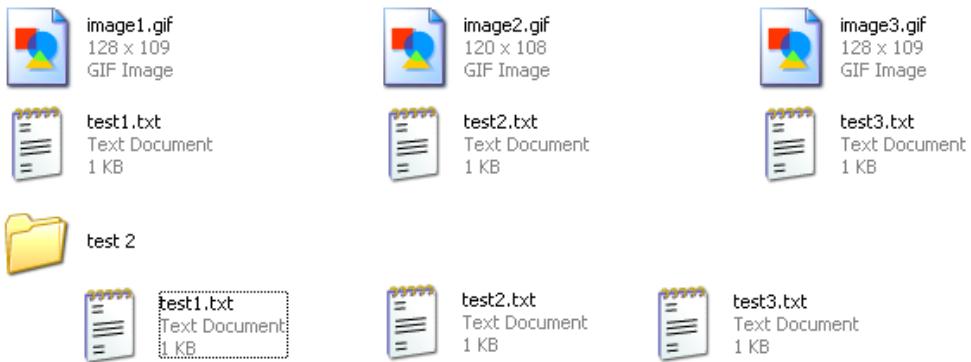


Figure 3.2 A test directory for your difference engine

The above figure shows the initial test directory (called test) that I set up on my computer. It doesn't get all of the possible failures, but it does check for most of them. The next step was to copy that directory (I called it test2) and make some changes for the difference engine to work on. I've used the numbers 1 to 4 within the files to represent each of the possible cases, with 1 and 4 being missing files, 2 for files which have some differences and 3 for files which are identical in both directories.

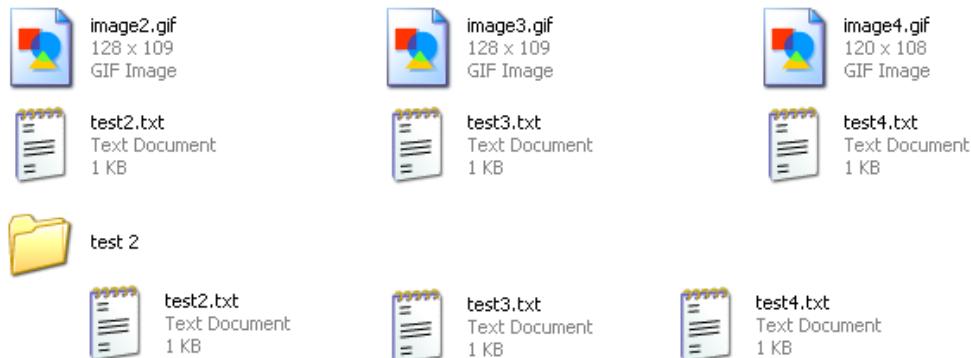


Figure 3.3 test2, an almost identical copy of the first test directory

You can see the output of running our script over these directories below:

```
D:\>python code\difference_engine.py test test2
Comparing:
test
test2
dir test root is test
dir test2 root is test2
test\test 2\test2.txt and test2\test 2\test2.txt differ!
image4.gif not found in directory 1
test 2\test4.txt not found in directory 1
test\image2.gif and test2\image2.gif differ!
test4.txt not found in directory 1
test\test2.txt and test2\test2.txt differ!
test1.txt not found in directory 2
test 2\test1.txt not found in directory 2
image1.gif not found in directory 2
```

That seems to be pretty much what we were expecting. Our script is descending into the test 2 directory in each case and is picking up the differences between the files - 1 and 4 are missing, 2 is different and 3 isn't reported since they're identical.

Now that we've tested out our script, let's see what we can do to improve it.

3.7 *Improving our script*

Our script so far works, but it could do with a few improvements. For a start, the results that it returns are out of order. The files which are missing from the second directory appear right at the end. Ideally we'd have them appear next to the other entries for that directory, to make it easier to see what the differences are.

NOTE Does this strategy look familiar? It's exactly what we did when developing Hunt the Wumpus. We start by writing a program that's as simple as we can make it, and then build on the extra features that we need.

3.7.1 *Putting results in order*

It initially might be difficult to see how we might go about ordering the results, but if you think back to chapter 2, one of the strategies that we used with Hunt the Wumpus was to separate the program from its interface. In our difference engine, we haven't done so much of that so far - now might be a good time to start. We need two parts to our program - one part which does the work and stores the data which it generates, and another to display that data.

Listing 3.8 Separating generating results from display

```
dir1_file_list, dir1_root = directory_listing(directory1)
dir2_file_list, dir2_root = directory_listing(directory2)
results = {}

for file_path in dir2_file_list.keys():
    if file_path not in dir1_file_list:
        #1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        results[file_path] = "not found in directory 1"          #2
else:
    file1 = os.path.join(dir1_root, file_path)
    file2 = os.path.join(dir2_root, file_path)
    if md5(file1) != md5(file2):
        results[file_path] = "is different in directory 2"  #2
    else:
        results[file_path] = "is the same in both"           #2

for file_path, value in dir1_file_list.items():
    if file_path not in results:
        results[file_path] = "not found in directory 2"      #2
#1 Results dictionary
#2 Storing results

```

Here's the 'trick'. Rather than try and display our results as soon as we get them, which means that we're trying to shoehorn our program structure into our display structure, we store our results in a dictionary to display later (**1**).

The result of each comparison is stored in `result (2)`, with the file path as the key and a description of the result of the comparison as the value.

That should take care of storing the results - let's take a look at how we display them.

```

print
for file_path, result in sorted(results.items()):          #1
    if os.path.sep not in file_path and "same" not in result:  #2
        print path, result                                     #2

for path, result in sorted(results.items()):                #3
    if os.path.sep in file_path and "same" not in result:   #3
        print path, result                                     #3
#1 Sorting results
#2 Checking within strings
#3 Other directories

```

`sorted` is a built-in Python function which sorts groups of items (**1**). You can give it lists, dictionary keys, values or items, strings and all sorts of other things. In this case we're using it to sort `result.items()` by `file_path`, the first part of `result.items()`.

Within the body of our loop, we're using `in` to check the contents of our strings (**2**). We want to know whether this path is part of a directory, in which case it'll have `os.path.sep` somewhere within it, and we also want to know whether the result shows that the files are the same.

Now that we've displayed everything within the root of the directory, we can go ahead and show everything

ALL STEF'S *MARKETING MATERIAL* WAS PUT ON A SEPARATE DRIVE AFTER LAST TIME, I JUST ... REMOVED IT.



within the subdirectories (3). We're just reversing the sense of the if statement to show everything that wasn't shown the first time around.

In hindsight that was relatively easy. Following the pattern that we established in Hunt the Wumpus, separating data from its display, is a powerful tactic that can make complicated problems easy to understand and program.

3.7.2 Comparing directories

The other thing that our program should probably handle is the case where we have empty directories. Currently it only looks for files, and any empty directories will be skipped. While not necessary for our initial use case (checking for missing images before we back up), it will almost certainly be useful somewhere down the track. Once we've added it we'll be able to spot any change in the directories, short of permission changes to the files, and it requires surprisingly little code.

Listing 3.9 Comparing directories too

```
def md5(file_path):
    if os.path.isdir(file_path): #1
        return '1' #1
    read_file = file(file_path)

    ...

    for path, dirs, files in os.walk(directory_name):
        ...
        for each_file in files + dirs: #2
            file_path = os.path.join(trimmed_path, each_file)
            dir_file_list[file_path] = True
#1 Don't try and checksum directories
#2 Include directory and file paths
```

The first thing to do is to include directory paths as well as files when generating a listing (2). To do that, we just join the dirs and files lists with the + operator.

If you try and open a directory to read its contents you'll get an error (1), since directories don't have contents in the same way that files do. To get around that, we cheat. We alter our md5 function, and use `os.path.isdir()` to find out whether it's a directory. If it is, we return a dummy value of '1'. It doesn't matter what the contents of a directory are, since the files will be checked anyway when it's their turn. We only care whether a directory exists (or not).

Once you've made those changes, you're done. Since the directories follow the same data structure as the files, you don't need to make any changes to the comparison or

OH I'LL JUST PUT IT IN
WITH THE NEGATIVES FROM
THE LAST OFFICE XMAS
PARTY - I'M PRETTY SURE
I HAVE A PAY REVIEW
COMING UP SOON...



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

display parts of your program. You'll probably want to add some directories to both of your test directories to make sure that it's working properly.

You've improved your script, but that doesn't mean there isn't more we can do.

3.8 Where to from here?

The program as it stands now is feature-complete based on our initial need, but you can use the code that we've written so far for other purposes. Here are some ideas:

- If you're sure that you won't have any different files, you can extend your program to create a merged directory from multiple sources. Given a number of directories, consolidate their contents into a third, separate location.
- A related task would be to find all of the identical copies of a file within a directory - you might have several old backups, and want to know whether there are any sneaky extra files that you've put in one of them.
- You could create a change monitor - a script that notifies you of changes in one directory. One script would look at a directory and store the results in a file. The second script would look at that file and directory and tell you if any of the output has changed. Your storage file doesn't have to be complicated - a text file containing a path and checksum for each file should be all you need.
- You can also use your os.walk functions as a template to do something other than check file contents. A script to check your directory sizes could be useful. Your operating system will probably give you information about how much space a particular directory takes up, but what if you want to graph usage over time? Or break your results down by file type? A script is much more flexible and you can make it do whatever you need.

Of course, you'll need to avoid the temptation of reinventing the wheel. If there's a tool which has already been written that solves your problem, it's generally better to use that, or at least include it in your script if possible. For example, you might consider writing a program which shows you the changes between different versions of files as well as whether they're different, but that program's already been written - it's called `diff`. It's widely available as a command line program under Linux, but it's also available for Windows and comes in graphical versions too.

One of the other programming tricks is knowing when to stop. Gold-plating your program can be fun, but you could always be working on your next project instead!

3.9 Summary

In this chapter, you learned about some of the standard library packages available with every installation of Python, as well as how to include and use them, and how to learn about unfamiliar ones.

We built what would normally be a fairly complex application, but since we made good use of several Python libraries the amount of code that we needed to write was quite minimal.

In the next chapter, we'll look at another way of organizing our programs, as well as other uses for functions and some other Python techniques which can help you to write clearer, more concise code. Our program in this chapter was fairly easy to test, but not all programs will be that straightforward, so we'll also look at another way of testing programs to make sure that they work.

4

Getting organized

Up until now, we've been learning how to use Python and our programming has been "by the seat of our pants." Hunt the Wumpus didn't have much in the way of planning and no tests at all, and in the last chapter we tested - but only fairly lightly. Now we'll change tack and focus on how to plan and test programs more thoroughly. We'll also do some more tricky things with functions and learn about `pickle` and `textwrap`, two more of Python's standard libraries.

The major change for this chapter is that we'll start learning how to test our programs automatically. *Unit tests* are a relatively recent idea and help offload a lot of the grunt work of testing and debugging your programs onto the computer. We'll also be turning our development practice on its head by using Test Driven Development, writing our tests before our program. It sounds odd, but it can be very enlightening to see how unit testing can make tricky problems easy, and how writing tests first can help shape the design of your program for the better.

Since our theme for this chapter is "Getting Organized", the program that we'll be writing is a productivity application to help us manage todo lists. We'll be making it a command line application so that we can focus on the important parts, namely getting the todo list functionality right. Later on in chapter 8, we'll see how we can extend the core of this program and give it a web interface.

Let's start by figuring out what we'd like to accomplish.

4.1 Planning: Specifying your program

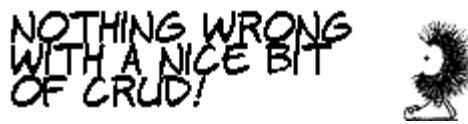
The first thing that we need to do is try to figure out ahead of time what our program needs to do, as well as what would be "nice to have". That way we'll have the advantage of knowing in advance what we're trying to do, and will have had time to think about the best way to approach a problem.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

We'll be using a "top-down" approach to our design, where we break our program down, and describe each part. We'll also want to think about how each part fits together, how they'll communicate and store data - commonly referred to as the *architecture* of your program. If you have enough detail to start programming, then great. If not, you can repeat it by breaking down your parts into other parts until each is detailed enough for you to start work (or, for your customer to sign off). Different projects will require different levels of detail depending on what they are and who the final customer is. The finished product of this process is known as a *specification*, or *spec*, and it's much like a blueprint for a building.

In the case of our to-do list program, we can use the popular computer industry acronym *CRUD* to help guide our spec. *CRUD* isn't a statement on the quality of your program - it stands for Create, Retrieve, Update and Delete, which are the four basic things which you generally need to be able to do with your data.



1. Add a to-do item (Create)
2. View the to-dos that you've already created (Retrieve)
3. Edit the information in a to-do (Update)
4. Delete a to-do from your list (Delete)

Our program will also need to handle input from the user, as well as saving the current to-dos so that they're accessible later on and searching for specific to-dos - or at least displaying a list of the ones which match certain criteria (such as 'due today').

In terms of architecture, we'll reuse the functions plus shared data route that we put together for Hunt the Wumpus, but soup up the user interface so that we can ask the user for more detailed information.

4.2 How do you know your program works?

Before we dive into coding though, let's first talk a little bit about unit tests and how we can test our programs more thoroughly. Testing properly sounds boring, but in fact it's the opposite. If you don't test, you will invariably end up debugging your program instead. If you think testing is boring, debugging is ten times worse. Let's see how automated tests can help make your programming more fun.

4.2.1 Testing manually - boring!

In our Hunt the Wumpus program, we had no automatic testing at all. Any changes that we made to the program were verified manually - whenever we made a change, we'd run our program and type some input, and if everything looked good, then we assumed that our change was good. This can have some downsides, as we saw. Your program can look ok, but have errors that you can't see.

Note: Why the emphasis on testing? The simple answer is that creating a program is much more fun than trying to debug it. Testing thoroughly, particularly with automatic tests, really helps to nip errors in the bud, and will keep your programming fun!

The other problem is that it's boring. That means that as you develop your program further, you're more likely to assume that something's working, but it may in fact be broken, particularly as you ask old parts of your program to do new things. An "easy fix" can turn out to seriously break your program.

HMM, I WONDER IF THIS WILL WORK?



4.2.2 Functional testing

When writing our difference engine, we used some simple *functional testing* to make sure that our program worked properly. We set up two directories, ran the program and checked that it output the right results - that is, we tested its functionality directly. That's a lot better than manually testing, since it's easier and faster and you're not likely to get too bored, but the downside is that it only finds bugs in your program. You still need to go through the arduous process of debugging in order to find out what's causing the error.

The other problem is that if you change how your program works, then you might need to change your tests, which could potentially be a lot of work - you might be tempted to ignore your tests and go back to testing manually.

4.2.3 Unit testing - make the computer do it

Luckily there's an easier way to deal with repetitive, boring tasks - make the computer do it. The mechanism that we'll use in this chapter is called *unit testing*. Unit testing works by testing small parts, or units, of your program. Just like we've been breaking down our programs to make them easier to write, unit testing breaks down your program into units such as functions, and makes sure that they work for a range of inputs. Unit tests also help to isolate the code which you're testing, which means that any errors when running your tests can be quickly tracked down to individual functions and fixed.

4.2.4 Test driven development

The key way to use unit testing when developing your programs is to write your tests first. That seems backwards and odd, but it focuses you towards the higher level design of your code, instead of the nitty-gritty details. The way that it works is this: if you want to add a feature, you write a test and run it. You won't have written the code that the test needs yet, so it'll break. You then add enough code that your test passes and starts to work, then think of another test and repeat the process. As you work, you'll be building a suite of tests which will help keep your program on the right track.

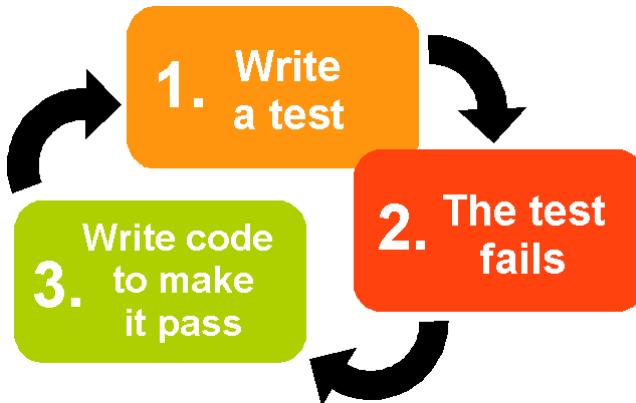


Figure 4.1 - The Test Driven Development cycle.

Your unit tests will also act as a low level specification for how your program should work - exactly what input should Python expect from this function? What should it do when you give it some input that it's not expecting, or if the input is wrong?

4.3 Writing our program

Let's get started writing our first test. There are libraries which can help us to test our programs, but for now we'll keep things simple and just use Python's built-in assert statement. assert takes the following format:

```
assert something == something_else, "Message if
assert is triggered!"
```

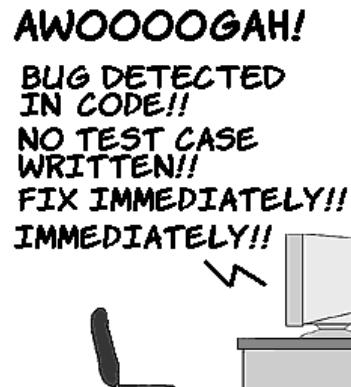
Python will test the first condition, exactly like an if statement, and if it's false then an error is raised with the message that you've specified in the second part. We'll test each part of our program with a function that tries a particular section of your program, then uses assert to make sure that the results are what we expect.

Type the program from Listing 4.1 into a file called `test_todo.py`. This test specifies how one function from our program should behave.

Listing 4.1 - Our first unit test

```
import todo #1

def test_create_todo(): #2
    todo.todos = [] #3
    todo.create_todo(todo.todos, #4
```



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        title="Make some stuff",                      #4
        description="Stuff needs to be programmed",    #4
        level="Important")                           #4

    assert len(todo.todos) == 1, "Todo was not created!" #5
    assert todo.todos[0]['title'] == "Make some stuff"   #5
    assert (todo.todos[0]['description'] ==           #5
            "Stuff needs to be programmed")           #5
    assert todo.todos[0]['level'] == "Important"       #5

    print "ok - create_todo"                         #6

test_create_todo()                                #6

#1 import our todo program
#2 our first test
#3 our list of todos
#4 run a small part of our program
#5 testing the results
#6 running the test

```

This is the program that we'll write, which will be called `todo.py` **(1)**. Don't create the file just yet, we'll do that in the next section.

Our first test is just a simple function **(2)**. Note that you should follow the same rules for your unit tests that you do for the functions in the rest of your program. If they're confusing, then you'll have trouble finding errors or fixing things when your tests fail.

`todos` will be where our program will store its list of todos **(3)**. Note that we're overriding whatever the current list of todos is by making it an empty list. This helps us write our tests faster - if we had a shared todo list, then we'd have to worry about what other tests had done to it before we'd run our tests. The other thing to note is that we're referring to the module's version of `todos`. If we were to just use a local `todos` variable we'd have two versions - one in the module and another that we've created, and we might confuse the two.

Our test runs one small part of the todo program **(4)**, creating a todo. It might be tempting to do more in one test, but the larger your test is, the more difficult it is to track down errors when they happen.

Now we use Python's `assert` command to test that `create_todo` has done the right thing **(5)**. It should have created a todo item with the right details and added it into our todo list.

Once we've set up our test, we can just call it to run it and test our program **(6)**. We've also added a `print` statement so that we know when the test has been run successfully.

Note: It's not just your tests that should be simple either. Unit testing also forces you to make your *code* simple. Large, crufty functions are hard to test - and by extension, hard to understand.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

The main thing to notice about listing 4.1 is that it's short and simple. Unit tests shouldn't be long, complicated and hard to understand - if they are then there's something wrong with either your tests or your code.

4.3.1 Making our tests pass

We have a unit test, but what does it do? Let's run it and see what happens.

```
Traceback (most recent call last):
  File "D:/Documents and
Settings/Anthony/....test_todo.py",
    line 2, in <module>
      import todo
ImportError: No module named todo
```

Uh-oh, what's gone wrong? Well, nothing. That's pretty much what we were expecting. Since we haven't written our program yet, our test doesn't have a `todo` module to work with. From here we'll be adding bits to our program to fix the errors that we'll get from our unit tests, so go ahead and create a file called `todo.py` in the same directory and run the test again.

```
Traceback (most recent call last):
  File "D:/Documents and Settings/Anthony/....test_todo.py",
    line 18, in <module>
      test_create_todo()
  File "D:/Documents and Settings/Anthony/....test_todo.py",
    line 6, in test_create_todo
      todo.create_todo()
AttributeError: 'module' object has no attribute 'create_todo'
```

Another error, but it's different this time and on line 6 rather than line 2, so we're making progress. Our test is now complaining that it can't find the `create_todo` function, so let's go ahead and create that. As input, it'll need to have our `todo` list, plus a title, description and level, since that's what we've specified in our test.

```
def create_todo(todos, title, description, level):
    pass
```

It's a very simple program, that uses Python's `pass` statement to do nothing at all. It doesn't pass our test either, but we're getting further still. We're now starting to test the functionality of our program rather than whether a function exists.

```
Traceback (most recent call last):
  File "D:/Documents and Settings/Anthony/....test_todo.py",
    line 18, in <module>
      test_create_todo()
  File "D:/Documents and Settings/Anthony/....test_todo.py",
    line 10, in test_create_todo
```

COOL. ISN'T IT? THAT'S
"ORWELL". OUR NEW
TESTING AND
PERFORMANCE
FRAMEWORK!



```
assert len(todo.todos) == 1, "Todo was not created!"
AssertionError: Todo was not created!
```

Now our test is complaining that the todo wasn't added to the todo list. The code to make your test pass is pretty obvious now, so let's fix it all in one fell swoop.

```
def create_todo(todos, title, description, level):
    todo = {
        'title' : title,
        'description' : description,
        'level' : level,
    }
    todos.append(todo)
```

Now our test passes. When you run the test against this program, you should see `ok - create_todo` printed to the screen. Fantastic - our test passes, so we know that the function is working. Let's now have a look at how we'll be calling this function within our program.

4.4 Putting our program together

We'll follow the same strategy that we did for Hunt the Wumpus - get something simple up and running quickly, then build from there. The simplest usable program that we can create will be something that's only able to create todos, but that should be enough. To get there, we'll have to think about how we want to be able to input todos, as well as how we get from that input to running the relevant function in our program, and then how we return output to the screen. More importantly, we want to think about an easy way for us to write tests to make sure that it's all working properly.



4.4.1 Testing user interfaces

One of the big problems with unit testing is that it's not so good at testing user interfaces. For example, there's no Python command which will let us type information into `raw_input`. Things get even harder when it comes time to test graphical interfaces, with mouse position and pop up windows.

The solution is to make your user interface as simple as possible, so that it's easy to test. Ideally it should be possible to make sure that your code is correct just by looking at it. In our todo list application, we'll use the following snippet to run everything in our program.

Listing 4.2 - One part of our program that we can't test

```
def main_loop():
    user_input = "" #1
    while 1:
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

print run_command(user_input)                      #1
user_input = raw_input("> ")                      #2
if user_input.lower().startswith("quit"):          #3
    print "Exiting..."                           #3
    break

if __name__ == '__main__':                         #4
    main_loop()                                  #4

#1 Do something with user input
#2 Get new input
#3 Check to see if we should quit
#4 Only run main_loop if we're run directly

```

First we do something with the command that the person using our program has typed in **(1)**. Initially we're not accepting input, which might seem a bit backwards, but it'll let us print a welcome/help screen when the program is first run. `run_command` is the 'meat' of our program, but it just takes whatever input has been typed in - that'll make it easier for us to test in a minute.

Once we've run with the input that we've been given, we tell Python to ask for some more input **(2)**.

The one command which is outside our `run_command` function is 'quit'. We check here for any command which starts with the word 'quit' **(3)**. If we see it, we break out of our while loop straight away, which ends the program.

When we're importing our program as part of our tests, we don't want to run our `main_loop` function, but we do if we run it directly as a program **(4)**. The solution is this `if` statement, which is very common in Python programs. `__name__` is the current namespace, or the name of the module that you're running in. If a program is run directly it'll be called `__main__`, and we can catch it with our `if` statement.

Note: This type of structure is called an event loop - it tells Python to wait for input from the person using the program, or some other source like the network, and then takes action based on what it finds.

The other thing that we need is some way to get multiple lines of input. If we're adding a new todo then we'll need to ask for its title, description and level. That's also fairly hard to test without resorting to drastic measures, like modifying the `raw_input` function.

Listing 4.3 - The other part of our program that we can't test

```

def get_input(fields):
    user_input = []
    for field in fields:
        user_input[field] = raw_input(field + " > ")
    return user_input

```

Again, this is a very straightforward part of our program. We're keeping it as simple as possible, so that we have very little to debug manually. If there is an error that our tests can't pick up, it should be very obvious where that error is.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

4.4.2 What do we do with our input?

Now we can start writing our `run_command` script in earnest. The first thing that we'll want the program to do is to pick a Python function to run based on what the user types in, so let's do that part first. It's fairly easy, but we'll need to use a new Python trick. You should put the next section in `test_todo.py`. All of the testing code will go in that file, and the program code itself will go into `todo.py`.

```
def test_get_function():
    assert todo.get_function('new') ==
    todo.create_todo           #1
#1 First class function
```



We're planning on setting up a function which tells us what to run for a given command. So when we call it with a 'new' command, we expect it to return the `create_todo` function. Not the results of the function, but the function itself. In Python you can assign functions to variables just as you can assign strings, numbers, lists and dictionaries. We'll see how to make use of it shortly.

Now that we have our test, run your tests again. The new test which you've just added should fail. Here's some code that fixes it, and allows room for us to expand to include other functions.

```
commands = {
    'new' : create_todo,          #1
}
#1 - A dictionary of our commands

def get_function(command_name):  #2
    return commands[command_name]
#2 - Find out which function to call
```

(1) is a dictionary with all of our commands in it. The value of the dictionary is just the name of the function.

Given the name of the command that we want to run, this function will return the function that we need to call **(2)**.

4.4.3 Running commands

That's one piece of the puzzle. The next piece is how we get the input from our `get_input` function into our final function. Well, we'll need to know what fields a particular function needs, so let's start with that.

```
def test_get_fields():
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```
assert (todo.get_fields('new') ==
       ['title', 'description', 'level'])
```

That's pretty easy, since it's much the same thing that we did to find the command function. Run your tests and make sure your test fails, then you can write your code. Here's my version:

Listing 4.4 - finding command fields

```
commands = {
    'new' : [create_todo, ['title', 'description', 'level']],
}

def get_function(command_name):
    return commands[command_name][0]

def get_fields(command_name):
    return commands[command_name][1]
```

Notice that we've changed the commands dictionary as well as the code that we wrote to find the function. It makes more sense to keep the command function and the fields that it's expecting in the same place so that they're easier to change and don't get mixed up. That's completely normal, and perfectly ok - as long as your tests still pass.

Now that we've created and tested those two low lying functions, we're ready to try and create our `run_command` function. That'll complete the user interface section of our program, and we can work on the rest of the code that actually does the work. There are a few more unit testing techniques that we'll need to use first though.



Listing 4.5 - testing run_command

```
def test_run_command():
    result = todo.run_command(
        'test',
        {'abcd':'efgh', 'ijkl':'mnop'}
    )
    expected = """Command 'test' returned:
abcd: efgh
ijkl': mnop"""
    assert result == expected, \
           result + " != " + expected
    print "ok - run_command"
#1 creating a 'test' command
#2 design by "wishful thinking"
#3 Our architecture helps us to test
```

When we're unit testing, we'd ideally like to test exactly one aspect of our program. If we have tests which combine results from lots of functions and one of those functions fails, we still have to debug our program. We only want to test `run_command`, so we're creating a dummy test program **(1)** which just returns its input, rather than forcing our tests to use (and then interpret the results from) the `create_todo` function.

The other thing that we need to test is that data is fed into our command function properly, but how do we do that without forcing someone to enter the data every time we test? The solution is to use a Python default variable to mimic the data entry **(2)**. When our program runs normally it'll ask the user via our `get_input` function, but if we feed in a dictionary it'll use that instead.

Since our architecture is just moving text around **(3)**, our function is easy to test - just feed in some input dictionary, and check that you get the right output back.

Let's see what the code which will make our test pass looks like. Again, once you've written the test, the code is relatively straightforward - and you generally don't have to debug functions that you've already written.

Listing 4.6 - writing run_command

```
def test(todos, abcd, ijk1):
    return "Command 'test' returned:\n" + \
        "abcd: " + abcd + "\nijkl: " + ijk1

commands = {
    'new' : [create_todo, ['title', 'description', 'level']],
    'test' : [test, ['abcd', 'ijkl']],
}

def run_command(user_input, data=None): #1
    user_input = user_input.lower() #2
    if user_input not in commands: #2
        return user_input + "?" \
            " I don't know what that command is." #2
    else: #2
        the_func = get_function(user_input) #2

    if data is None: #3
        the_fields = get_fields(user_input) #3
        data = get_input(the_fields) #3
    return the_func(todos, **data) #4

#1 Default variable
#2 Figure out which command function we need to run
#3 Get input from the user if necessary
#4 Call our function
```

First we use our default variable **(1)**. We set `data` to `None` for most cases, but when testing we can feed in `data` as a dictionary to mimic user input.

We use the `lower()` method to make the command lower case, and then look it up in our dictionary **(2)**. If we can't find it, then we return an error.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

When running the program normally, data will be `None`. When we see this, we know that we need to read some input from the user, and we can call `get_fields` so that we know what to ask **(3)**.

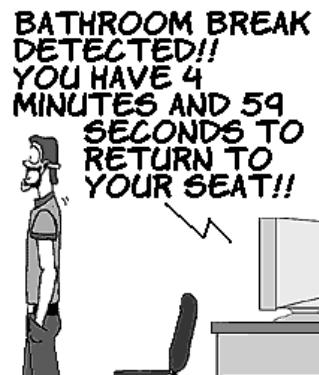
Now we know which function to call and what data to call it with, we can go ahead and pass control over **(4)**. The command function will do whatever it's supposed to, and feed the results back as a string, which we hand back to the user. The `**` in front of our input dictionary looks a bit weird - what it does is pass in the dictionary arguments in as keyword arguments. This way you can see what values a particular function is expecting in the function definition, rather than having one big value.

Great - now we have a straightforward way to assign text input from the person using our program and pass that on to a particular function. The rest of the chapter from here will deal with adding to that framework, by writing other functions which fit into it.

4.4.4 *Running your program*

You might have noticed something odd by this stage. We haven't actually run our program yet to make sure that it works. In previous chapters, we've been writing our program, running it to make sure it works, then writing a bit more. Since we've been unit testing though, we haven't had to do that once - the unit tests pass, so our code must be working, right?

You might be a bit skeptical about that, but our program is actually pretty functional at this stage, and you can run it if you want to make sure. Listing 4.7 has a sample run.



Listing 4.7 - Our program so far

```
D:\Documents and Settings\Anthony>python todo.py
? I don't know what that command is.
> test
abcd > qwer
ijkl > uiop
Command 'test' returned:
abcd: qwer
ijkl: uiop
> new
title > Test Todo
description > This is a test
level > Very Important
None
> quit
Exiting...
```

There are still a few loose ends to tidy up, but we can already create a todo item from the user interface on our program, which means that all of our infrastructure is working. We're on the downhill run now!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

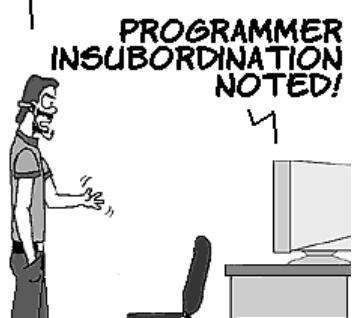
4.5 Taking stock

So far we've made a good start on our program, and we have most of the core of it working. As well as that, we have tests which we can run to make sure that our program stays working. Our unit tests have also had another benefit - since we've only been testing small parts of our program, our program is already broken down into small functions, and there's no need to tidy it up or refactor it. At least, not yet.

4.5.1 What to do next?

The next important part of our application is showing what's in the todo list. There's not much point in adding to your list if you can't see what's in there later on. To get started, we'll write a test for a function which will show us all of our todos, then we'll look at streamlining it to only show ones with a certain importance or higher. After we've done that, we'll look at how we can save our lists and reload them, so that we don't have to re-enter everything when we restart our program.

FIVE MINUTES FOR A
BATHROOM BREAK? YOU'VE
GOT TO BE KIDDING ME!



Listing 4.8 - testing our todo list view

```
def test_show.todos():
    todo.todos = [
        { 'title' : 'test todo',
          'description' : 'This is a test',
          'level' : 'Important'
        }
    ]
    result = todo.show.todos(todo.todos)
    lines = result.split("\n")

    first_line = lines[0]
    assert "Item" in first_line
    assert "Title" in first_line
    assert "Description" in first_line
    assert "Level" in first_line

    second_line = lines[1]
    assert "1" in second_line
    assert "test todo" in second_line
    assert "This is a test" in second_line
    assert "Important" in second_line

    print "ok - show.todos"
#1 - Setting up our data
#2 - Run the show.todos function
#3 - Test the results
```

We first set up a todo list **(1)**. Since our todo list is in a known state, it'll make it easier to test. It's tempting here to reuse our creation test to set up our todo list, but that's a trap. Even though it might save some code, you're creating a dependency between your tests. If later on there was a bug in the creation function, you'd have two (or more) test failures, and the bug would be much harder to track down.

We run our view function over the todo list **(2)**, and get the results back. To make life easier, we split the result up into lines by using the split method of the result string to split on line endings.

Now we test that the words that we're expecting exist in each line **(3)**. Our first line should be the headers for the columns, and the second should have the values that we're expecting. Notice that we're specifying each value individually - we could generate a string for the exact result that we're expecting from the function, but that's another trap. Specifying the results too strictly makes the test very fragile, and the slightest change to how the results are formatted or the order of the columns can make your test fail when it shouldn't. In practice, you should only test what's important, and leave as much of the rest out as you can.

Now that we know what we're expecting of our function, we can go ahead and write it. Python strings have several methods which we can use to format our output - let's see how we can use them.



Listing 4.9 - Displaying todo items

```
def show.todos(todos):
    output = ("Item      Title           "
              "Description          Level\n")
    #1                  #1
    #1                  #1
    for index, todo in enumerate(todos):
        #2                  #2
        line = str(index+1).ljust(8)
        #3                  #3
        for key, length in [('title', 16),
                             ('description', 24),
                             ('level', 16)]:
            #4                  #4
            line += str(todo[key]).ljust(length)
            #4                  #4
        output += line + "\n"
    return output
#1 - Todo list headers
#2 - Adding numbers
#3 - Formatting results
#4 - Formatting each todo item
```

First we initialize our output as a list of headers **(1)**. We'll be copying each line onto the end of the output as we go. Notice how I've put the string on two lines, wrapped within brackets? That's just to make it easier to read on the page. Python automatically joins strings like this, so there'll only be one big string when it's assigned to output.

Next we go through each of our todos and add numbers (2). I've added an index to make it easier to see how many todos you have. enumerate takes a list or iterable and returns the next item along with its index in the list. Very handy for situations like this.

In order to format the results, we start our line by printing the number of the todo (3). So that the rest of the columns line up, we convert it to a string, and use the `.ljust()` string method to space it out to 8 columns. Python strings have many other methods like this, such as `.rjust()` and `.center()`.

Now we print each part of our todo in a column (4). Here I've been a bit tricky and pulled out the key that we're printing and its width into a list, which we're looping over. That way, we can pull out each value from the todo and make it the right width.

The code that we've added is straightforward, but if you were developing this using "code and bugfix" as we did for Hunt the Wumpus, you'd have to go back and forth several times to get the code working. With unit testing, you can specify exactly what your output should be, and then add it directly to your program.

4.5.2 I'm very busy and important

The other thing that we'd like to check is that our view function displays our todos in the right order. Ideally, important things should be displayed differently depending on how important they are. We'll put the important items at the top, and unimportant ones at the bottom. The problem is that so far we've been putting the level of importance as a text field, which would appear to make our list a bit hard to sort. Luckily there are tools in Python to deal with this sort of thing.

But we're getting ahead of ourselves. First we need a test to make sure that our program sorts our todos properly!

**BUG DETECTED!!
THE PROGRAM AS
ENTERED DEVIATES
FROM MANDATE
#8394: "THE SITE
MUST BE BLUE!"
RECTIFY
IMMEDIATELY!!**



**HOW ABOUT WE PUT
EVERYTHING MARKED
AS UNIMPORTANT IN
THE ROUND FILE?**



Listing 4.10 - Testing the order of our view

```
def test_todo_sort_order():
    todo.todos = [
        { 'title' : 'test unimportant todo',           #1
          'description' : 'An unimportant test',       #1
          'level' : 'Unimportant'                      #1
        },
        { 'title' : 'test medium todo',                #1
          'description' : 'A test',                    #1
          'level' : 'Medium'                          #1
        },
        { 'title' : 'test important todo',             #1
          'description' : 'Important test',           #1
          'level' : 'Important'                      #1
        }
    ]
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        'description' : 'An important test',           #1
        'level' : 'Important'                      #1
    },
]
result = todo.show.todos(todo.todos)
lines = result.split("\n")

assert "Important" in lines[1]                  #2
assert "Medium" in lines[2]                     #2
assert "Unimportant" in lines[3]                #2

print "ok - todo sort order"
#1 - Sample list of todos
#2 - Test that they're in the right order

```

Here's our sample list of todos **(1)**. They're in reverse order (unimportant to important), to make sure that the sorting is working.

The todos should be in order, from important to unimportant **(2)**.

So that covers what we're expecting. How do we get there? In practice, we'll still be entering them as text strings, so how about if we put all of the important fields first, all the ones marked 'unimportant' at the bottom, and everything else in between?

The standard way that we'd do that would be with three for loops one after the other, each for a separate case, but I'd like to show you a faster way, which is also clearer once you get used to it.



4.5.3 List Comprehensions

List comprehensions are a powerful built-in Python tool for making sense of lists of things. They're a general solution to a common programming problem - that of handling groups of items. Perhaps you might want to get the total of every item in a list, or filter out the ones which aren't important, or only include the ones which have been open for too long. List comprehensions will let you do all of these things.

What we're trying to ask for when we want to display important todos is something like:

"Python, please give me every todo in our todo list which is marked as 'Important'"

We can use a list comprehension to get exactly that, and more. Let's have a look at some common types of list comprehension and get a feel for what they can do.

Listing 4.11 - Lots of things that you can do with list comprehensions

```
important.todos = [todo for todo in todos           #1
                   if todo['level'].lower() == 'important'] #1
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

def capitalize(todo):                      #2
    todo['level'] = todo['level'].upper()   #2
    return todo                           #2

important_todos = [capitalize(todo) for todo in todos      #2
                  if todo['level'].lower() == 'important'] #2

squares = [x**2 for x in range(10)]          #3
names   = [name.title() for name in list_of_names] #3

coordinates = [(x,y) for x in range(10) for y in range(10)] #4
#1 - the basic version
#2 - you can use functions too
#3 - and numbers
#4 - and you're not limited to using one list

```

Here's a list comprehension that gives us what we're after (1). Python will go through each todo in our list, and collect the ones which match our if statement (i.e. have a level of 'important'). I've added a .lower() call so that the level will get converted to lower case. 'important', 'Important' and 'IMPORTANT' will all match.

That's not all that list comprehensions can do. You can also apply functions to each member of the final result to get a different list (2). Here, we're writing a function to capitalize the level and then calling it on each todo that's marked as important.

If you have a list of numbers, you can perform other operations on them as well (3). If they're an object, you can call any method of that object, and so on. Anything that you can do to the original value, you can do within a list comprehension.

Finally, we have a list comprehension that uses two lists of numbers to generate a list of coordinates (4).

So, given all of that, our final code listing could look something like Listing 4.12.



Listing 4.12 - Code to sort our list of todos

```

def show.todos(todos):
    output = ("Item      Title           "
              "Description           Level\n")
    important = [capitalize(todo) for todo in todos      #1
                 if todo['level'].lower() == 'important']
    unimportant = [todo for todo in todos                #1
                  if todo['level'].lower() == 'unimportant']
    medium = [todo for todo in todos                   #1
              if todo['level'].lower() != 'important' and
                 todo['level'].lower() != 'unimportant']
    sorted.todos = (important +                      #2
                   medium +                      #2
                   unimportant)                 #2

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

for index, todo in enumerate(sorted.todos):
    line = str(index+1).ljust(8)
    for key, length in [ ('title', 16),
                          ('description', 24),
                          ('level', 16)]:
        line += str(todo[key]).ljust(length)
    output += line + "\n"
print output
return output
#3
#1 - Filtering todos
#2 - Joining the todos back up
#3 - Debugging

```

Here are three list comprehensions **(1)**, each for a separate level of todo "Important", "Unimportant" and "everything else". We capitalize the important ones, to make them stand out a bit more.

Once we have the todo lists separated, we can join them back up by using a + **(2)**.

If you want to see what the output looks like, you can print the output here, just before it's sent back. Our program will print exactly what the function returns, so we'll see what the end user will see **(3)**.

So now we can sort our todos into a specific order by using some list comprehensions. They're a very powerful and easy to understand tool. Often you'll find that you can replace complicated for loops with a simple function and a list comprehension.

4.5.4 *Oops, a bug!*

If you've had a look at the output from listing 4.13, you'll notice that the columns don't quite display properly. The `show.todos` test looks ok, but the second one has all of its fields squashed together - where an item is too long it's pushing the other columns out.

Listing 4.13 - Output from our tests

```

C:\Documents and Settings\Anthony>python test_todo.py
ok - create_todo
ok - get_function
ok - get_fields
ok - run_command
Item      Title          Description           Level
1         test todo      This is a test       IMPORTANT

ok - show.todos
Item      Title          Description           Level
1         test important todoThis is an important testIMPORTANT
2         test medium todoThis is a test       Medium
3         test unimportant todoThis is an unimportant testUnimportant

ok - todo sort order

```

That doesn't look very nice. Isn't unit testing supposed to make sure that our code is bug free? Unfortunately, not entirely. You can test for things that you've thought of, but if there's something that you haven't considered, then you might still have bugs in your program. If

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

you're using unit testing and you notice a bug in your program like this, the solution is relatively easy: write a test which covers the behavior that you *do* expect, make sure that it fails, and then fix your program.

It's also possible that you might have made a mistake in one of your tests. Again, unit testing is a useful tool, but not a complete solution. It's still possible to test the wrong thing, or to have bugs in your unit tests. In practice that's a lot less likely than having errors in your program, since the unit tests are easier to follow.

The question still remains though: What do we want the program to do when a line is too long? If we don't have a clear answer for that, then it's hard to write a test! We could trim the string down to a fixed width if it was too long, but we'd like all of the information to still be visible. A better way to do it would be to wrap each todo, so that we'd have something like listing 4.14 for the case above.

Listing 4.14 - A better way to display our todos

Item	Title	Description	Level
1	test important	This is an important test	IMPORTANT
2	test medium	This is a test	Medium
3	test unimportant	This is an unimportant test	Unimportant

From a visual point of view that looks a lot better, but how on earth are we going to program it? The short answer is - exactly the way that we've been programming it so far. Write a test first!

Listing 4.15 - Testing that our todos wrap lines

```
def test_todo_wrap_long_lines():
    todo.todos = [
        { 'title' : 'test important todo',
          'description' : ('This is an important test. We\'d really like '
                          'this line to wrap several times, to imitate what might '
                          'happen in a real program.'),
          'level' : 'Important'
        },
    ]
    result = todo.show.todos(todo.todos)
    lines = result.split("\n")

    assert "test important" in lines[1]
    assert "This is an important" in lines[1]

    assert "todo" in lines[2]
    assert "test. We'd really like" in lines[2]
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

assert "this line to wrap" in lines[3]                      #2
assert "several times, too" in lines[4]                     #2
assert "imitate what might" in lines[5]                    #2
assert "happen in a real" in lines[6]                      #2
assert "program." in lines[7]                             #2

print "ok - todo wrap long lines"
#1 - set up our data
#2 - test that the lines wrap

```

First we set up a todo with long lines that should be wrapped (**1**). We've made it look as close as we can to what we think a real todo might look like, just to make sure that wrapping works when we have to wrap over several lines, as well as just one. Notice that I've broken up the description according to what I'm expecting in the results. That helps when you're writing the test, since you can see what you need to check for.

Then we test that the correct parts of the lines appear when the todo is viewed (**2**). The test for the description goes for several lines, but this way we're sure that our program is wrapping properly for larger descriptions.

Well the test was easy, but I suspect that writing the code might be a bit harder. Fortunately we've been testing thoroughly so far, so if we make a mistake our tests should catch us.

Note: What would you do if the code were too hard to write? In that case, the answer is normally that you're trying to do too much at once, and you need to break the problem down into smaller, easier parts.

The problem is that we're wrapping our lines, but within other lines, so we can't rely on Python's built-in printing mechanisms. Python does have a `textwrap` module available, which doesn't quite do what we'd like, but it's a start. Our overall plan then would be to write a function to generate the lines for each todo. Within that, we can split each section of the todo (the title, description, etc.) into lines using the `textwrap` module, and then somehow knit them together into our final output. Let's have a go at that now. Here's the new function, `show_todo`, and the changes that you'll need to make to `show.todos`.

Listing 4.16 - A function to show a todo

```

import textwrap                                #1

...
def show_todo(todo, index):
    wrapped_title = textwrap.wrap(todo['title'], 16)      #1
    wrapped_descr = textwrap.wrap(todo['description'], 24)  #1

    output = str(index+1).ljust(8) + "  "
    output += wrapped_title[0].ljust(16) + "  "
    output += wrapped_descr[0].ljust(24) + "  "
    output += todo['level'].ljust(16)                  #2

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        output += "\n"                                #2

        max_len = max(len(wrapped_title),
                      len(wrapped_descr))
        for index in range(1, max_len):
            output += " " * 8 + " "
            if index < len(wrapped_title):
                output += wrapped_title[index].ljust(16) + " "
            else:
                output += " " * 16 + " "
            if index < len(wrapped_descr):
                output += wrapped_descr[index].ljust(24) + " "
            else:
                output += " " * 24 + " "
        output += "\n"                                #3

    return output

def sort.todos(todos):                           #4
    important = [capitalize(todo) for todo in todos
                 if todo['level'].lower() == 'important']      #4
    unimportant = [todo for todo in todos
                  if todo['level'].lower() == 'unimportant']  #4
    medium = [todo for todo in todos
              if todo['level'].lower() != 'important' and
                 todo['level'].lower() != 'unimportant']       #4
    todos = important + medium + unimportant      #4
    return todos

def show.todos(todos):                          #5
    output = ("Item      Title           "
              "Description          Level\n")
    sorted.todos = sort.todos(todos)
    for index, todo in enumerate(sorted.todos):
        output += show.todo(todo, index)
    return output

#1 - Wrap title and description
#2 - Output the first line
#3 - Output any remaining lines
#4 - Sorting todos
#5 - New version of show.todos

```

First we use the `textwrap` module's `wrap` function to wrap the title and description to the right number of characters **(1)**. You'll also need to import `textwrap` at the top of your script.

We start by building the first line, with our index and level, which we assume don't wrap, plus the first wrapped line of the title and description **(2)**. We're using the `+=` operator, which is shorthand for `output = output +` I'm also adding two spaces between each column, to make it easier to read.

If there are any lines left in our title or description, we print them here, and put in placeholders for the index and importance **(3)**. We're using a slightly different version of `range`, where we specify the starting index as well as the ending one. If there's only one line,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

`max_len` will be 1 as well, `enumerate` will be empty and no extra lines will be printed. The other catch is that before we print out each line in our title and description, we need to make sure that we still have something to print, otherwise Python will crash with a 'list index out of range' error.

While not strictly necessary, I've broken the sorting of todos out into its own function (4). We can do this since we have unit tests to catch any breakages, and it makes our program nicer to look at.

Our new version of `show.todos` calls on both `show.todo` and `sort.todos` and is much shorter and easier to follow (5). That tells us that we're moving in the right direction - if it were longer and more complicated, we'd be doing the wrong thing.

The last thing that you'll need to do is update the `test_todo_sort_order` test case so that it references the new line numbers in the output. If you run your tests after that change they should all pass, and you now have a much prettier view of your todos. Ching! Next feature!

4.6 Saving your work

The last essential thing that we need to be able to do is save our todo list to a file. Without that, the person using our program would have to re-enter all of their work. Well, they probably wouldn't - they would instead find a program which could. Since we'll be using this program ourselves, that's not really an option.

To save our todo list, we'll be using a Python module called `pickle`, which is designed for writing Python objects to a file. There are some limitations on the sorts of objects that you can pickle, but all of the basic Python types such as strings, lists and dictionaries are supported, so it's ideal for our program. Using `pickle` has the advantage of being quick to implement and easy to test, but it won't be editable in a plain text editor. Writing your own functions to read and write a custom format is possible, but it's harder to program and difficult to get completely right. We'll take the easy option, but you can always write your own format at a later stage if you need it.

So how do we test our saving function? The easiest option is to use what's called a round-trip: create a todo list and save it, then reload it from the same file and compare it to the original. If it's the same, then our test passes, but the downside is that you're testing both the load and save functionality in one go. If your test doesn't pass, then it's hard to tell whether it's the load function or the save function (or both) which is at fault. The way around that is to create a "known good" file from a successful save, but that implies that you've already saved properly.

Let's pick the first option and see how we go - we'll be using a built-in Python module in a pretty straightforward way, so we're not likely to run into any major problems.

CAREFUL NOT TO DO
TOO MANY ROUND
TRIPS - YOUR DATA
MIGHT GET DIZZY!



Listing 4.17 - Testing that our application saves properly

```

def test_save_todo_list():
    todos_original = [
        { 'title' : 'test todo',                               #1
          'description' : 'This is a test',                  #1
          'level' : 'Important'                            #1
        }
    ]
    todo.todos = todos_original                         #1
    assert "todos.pickle" not in os.listdir('..') #1

    todo.save_todo_list()                             #2
    assert "todos.pickle" in os.listdir('..')      #2

    todo.load_todo_list()                           #3
    assert todo.todos == todos_original            #3
    os.unlink("todos.pickle")                      #3

    print "ok - save todo list"
#1 - Create a todo list
#2 - Test saving
#3 - Test loading

```

Here we're creating our todo list **(1)**, exactly the same as we have been in previous tests. The only difference is that we're keeping another copy so that we can refer back to it once we've reloaded the todo list. We also make sure that we don't have an existing todo list, otherwise our tests would fail or overwrite someone's todo list.

First we run our save command **(2)**. While we can't test the contents of the save file directly, we can test that the file has been created by using the `os.listdir()` function. `'.'` is shorthand for whatever the current directory is.

Next we clear our todo list out and then call the `load_todo_list` function to reload it **(3)**. At the end we'll have two lists of dictionaries, which should be exactly the same.

Run your tests, make sure that the new one fails, and then we'll add code to create our save file and reload it.



Listing 4.18 - Loading and saving your todos

```

import pickle
import os

...

def save_todo_list():
    save_file = file("todos.pickle", "w")           #1
    pickle.dump(todos, save_file)                   #2
    save_file.close()                            #3

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

def load_todo_list():
    global todos
    if os.access("todos.pickle", os.F_OK):
        save_file = file("todos.pickle")
        todos = pickle.load(save_file)
#4
#5
#6
#6

#1 - Open our save file
#2 - Dump todos into the file
#3 - Close the save file
#4 - The todos variable needs to be global
#5 - Make sure our save file exists
#6 - Load todos from the file

```

pickle needs an open file to work with, so we first open our save file **(1)**, which we've called "todos.pickle", with a mode of "w", which means open it and overwrite whatever's already there.

The pickle syntax is very straightforward - just call the pickle.dump function **(2)** with the object that you want to pickle, and the file where you want it to be pickled.

Next we close the file **(3)**. You don't have to do this step, since python will close the file once it leaves the save_todo_list function, but it's a good habit to get into and helps to keep things tidy.

Since we're replacing our todo list when we load it, we'll need to declare it as a global variable **(4)**. This means that the changes which we make to the todos variable will be visible outside our function.

One thing that we need to check before we do anything is that the file exists **(5)**. If we try to open a non-existent file then Python will raise an error and our program will crash.

Once we're ready to load from our save file it should be opened in read mode **(6)**. The pickle.load method will then read the todo list which we previously saved. Once we're done we close the save file. We don't need to return the object, since it's a global variable and we've already updated it.

The only question once we've added our load and save functions is where we call them. We could make the user call them explicitly, but they'd have to know they were there and remember to call them. An easier way is to call load automatically when the program starts, and then save when the program exits. You can easily add that by calling load_todo_list and save_todo_list at the start and end of our main loop.



Listing 4.19 - Automatic loading and saving

```

def main_loop():
    user_input = ""
    load_todo_list()
#1

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

while 1:
    print run_command(user_input)
    user_input = raw_input("> ")
    if user_input.lower().startswith("quit"):
        print "Exiting..."
        break
    save_todo_list() #2

#1 - Automatically load
#2 - Automatically save

```

Before we start accepting any user input, we first look for a pre-existing save file, and if it exists we load our todos from that (**1**).

Once the user issues a quit command, we break out of the loop and the program will automatically save its todo list (**2**). If you want to be even more cautious, you can call `save_todo_list` at the end of each function which might cause a change - `create_todo`, `edit_todo` and `delete_todo`.

We can add, view and save our todo lists (that's the C and R in CRUD for those of you who remember the first part of the chapter) and store all of the todos entered to date, so all we have to do now to have completed all of the absolutely essential features is to handle the editing and deleting of our existing todos.

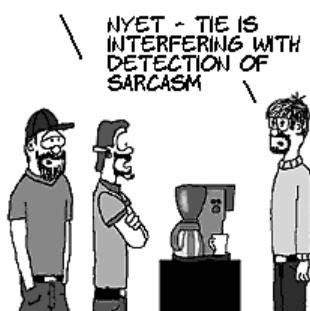
4.7 Editing and deleting

For our particular application they're not quite as essential, which is why we've left them until last, but it'd be pretty annoying to have to do without deletion or editing.

4.7.1 A quick fix

Firstly, there's one problem which we should deal with before we start. When we sorted our todos earlier, we didn't update the stored list, and sorted our list every time that we viewed it. When the user wants to tell us which todo they want to edit or delete, say with the index number, we'll have to rebuild the list again to know which one they mean. It'd be much easier to have the todo list sorted already. Let's do that now. It'll mean calling the `sort.todos` function every time a todo is added to the todo list. The user might've changed the importance of a todo when editing it, so we'll need to call it then too, but we won't need to call it for deletion, since it'll already be in order then.

DO YOU THINK WE
LAID THAT ON A
BIT THICK?



As we've done so far, the first place to start is by writing a unit test.

Listing 4.20 - Adding a todo sorter

```

def test_todo_sort_after_creation():
    todo.todos = [
        { 'title' : 'test unimportant todo', #1
          ...

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        'description' : 'This is an unimportant test',
        'level' : 'Unimportant'
    },
    { 'title' : 'test medium todo',
        'description' : 'This is a test',
        'level' : 'Medium'
    },
]
]

todo.create_todo(todo.todos,
    title="Make some stuff",
    description="Stuff needs to be programmed",
    level="Important")

assert todo.todos[0]['level'] == "IMPORTANT"
assert todo.todos[1]['level'] == "Medium"
assert todo.todos[2]['level'] == "Unimportant"

print "ok - todo sort after creation"
#1 - Set up our initial data
#2 - Create another todo
#3 - Check todo order

```

This should be pretty familiar by now. For this test we're only setting up two todos, in the reverse order (**1**).

(**2**) is where our action takes place. We create an important todo. With the code as it currently stands, this will just append the important todo at the bottom.

Now we check that all of the todos are in the right order (**3**). Important ones come first, unimportant at the bottom and everything else in the middle.

Run your tests now, and the newest one should fail. Time to write some code!

**AWOOGAH!
TIE IS 12
DEGREES FROM
VERTICAL!
INSUFFICIENT
PERSONAL
GROOMING
DETECTED!**



Listing 4.21 - New sort_todos

```

def sort.todos():
    global todos
    #
    important = [capitalize(todo) for todo in todos
                 if todo['level'].lower() == 'important']
    unimportant = [todo for todo in todos
                  if todo['level'].lower() == 'unimportant']
    medium = [todo for todo in todos
              if todo['level'].lower() != 'important' and
                 todo['level'].lower() != 'unimportant']
    todos = important + medium + unimportant
    #
def create_todo(todos, title, description, level):
    todo = {
        'title' : title,
        'description' : description,

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        'level' : level,
    }
todos.append(todo)
sort.todos() #2

def show.todos(todos):
    output = ("Item      Title      "
              "Description      Level\n")
    for index, todo in enumerate(todos):
        output += show.todo(todo, index) #2
    return output
#1 - Alter sort.todos
#2 - Move sort.todos from show.todos to create_todo

```

Ideally, we'd like to be able to call `sort.todos` from anywhere in our program, but that's a bit hard in its current state. The easiest way forward is to make `todos` a global variable **(1)**. Note that once we've done this we don't have to return `todos` from `sort.todos`. Now we can call it from any function that we want to.

Now that `sort.todos` is easier to use, we can remove it from `show.todos` and just put it wherever the order of `todos` is likely to be changed **(2)**. In the next section we'll also call it when changing `todos` in our todo list.

That should be enough to get us going for the next section. What we've done is to ensure that our todo list is always sorted in the same order, whether that's behind the scenes or when displayed on the screen. It's a major change to the way that the program stores its

data, but since we have a suite of unit tests we can be confident that making major changes like this won't have broken anything in our program. Let's press on and put the final pieces in place.



4.7.2 Deleting todos

Now we're ready to start deleting todos from our list. The code to do this is pretty straightforward, but since we're starting on destructive functions that can potentially delete user data, we'll step up the unit testing a notch. Up until now we've mainly been testing the "happy path", by making sure that the code works for normal usage. It's equally important to make sure that your program notices input or data which is wrong, and generates an appropriate error message. Let's take a look now at how we test that.

Listing 4.22 - testing deletion

```

def test_delete_todo():
    todo.todos = [
        { 'title' : 'test important todo',
          'description' : 'This is an important test',
          'level' : 'IMPORTANT'
        },
        { 'title' : 'test medium todo',

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        'description' : 'This is a test',
        'level' : 'Medium'
    },
    { 'title' : 'test unimportant todo',
        'description' : 'This is an unimportant test',
        'level' : 'Unimportant'
    },
]

todo.delete_todo(todo.todos, which="2") #1
#1

assert response == "Deleted todo #2" #1
assert len(todo.todos) == 2 #1
assert todo.todos[0]['level'] = 'IMPORTANT' #1
assert todo.todos[1]['level'] = 'Unimportant' #1

def test_delete_todo_failure():
    todo.todos = [
        { 'title' : 'test important todo',
            'description' : 'This is an important test',
            'level' : 'IMPORTANT'
        },
    ]

    for bad_input in ['', 'foo', '0', '42']: #2
        response = todo.delete_todo( #2
            todo.todos, which=bad_input) #2
        assert response == ("'" + bad_input + #2
            "' needs to be the number of a todo!") #2
        assert len(todo.todos) == 1 #2

    print "ok - test delete todo failures"
#1 - Test the "happy path"
#2 - Test for bad input

```

For our deletion test, we set up three todos and delete the middle one. This tests that we don't delete the wrong todo, as well as that the right one is deleted **(1)**. We're also checking that `delete.todos` gives us back a reasonable message to tell us what it's done.

One of the things that we've been able to skip over so far is checking user input. For our deletion script that's no longer possible, since you might put in a wrong number or something that isn't a number. Here we check that all of the possible types of bad input generate an error message and don't delete any todos **(2)**.

That covers all of the potential things which we can think of which can go wrong, but it's important to note that testing failures like this is an ongoing process. In other words, our failure tests aren't final. Particularly with more complex functions, there may be bad input or data which will cause errors that you haven't considered. When you find input like that you



should consider it a bug in your program, but the fix is easy - either add another unit test or an extra case to your failure test which will cover the failure, and then fix your code so that the test passes.

Listing 4.23 - deleting todos

```
def delete_todo(todos, which):
    if not which.isdigit(): #1
        return ("'" + which +
               "' needs to be the number of a todo!")
    which = int(which) #1
    if which < 1 or which > len(todos): #1
        return ("'" + str(which) +
               "' needs to be the number of a todo!")
    del todos[which-1] #2
    return "Deleted todo #" + str(which) #3

#1 - Check user input
#2 - Delete the todo
#3 - Show user what we've done
```

Here's where we do our checking to make sure that the input that we're fed matches a todo in our list **(1)**. It has to be a number, so we first use the `.isdigit()` method to make sure of that. Then we turn it into a number by using `int()`, and check to see if it corresponds to an entry in our todo list. If our input fails any of these checks, we do nothing except return an informative error message.

Now we can delete our todo **(2)**. Notice that we're converting the number that we're given into a list index by subtracting one from it.

The person using the program probably wants to know what we've done, so we tell them here **(3)**. Whatever string we return will be printed on the screen as a result.

That's all we need to do to make sure that deleting todos works properly. All your tests should pass now, and we're ready to move on to the next section.

4.7.3 Editing todos

Editing our todos is also fairly straightforward. Since editing is a cross between deletion and editing in many ways, we can combine code from our previous unit tests and program code to create an `edit_todo` function. There's nothing in principle that we haven't already covered in this chapter.

The only catch is that we're running into a limitation of Python's `raw_input` function. Since we can't pre-populate the text which is



entered into the function, we can't make it as easy as we'd like to edit an existing entry, so unfortunately we'll need to work around it. The easiest way is to make a blank entry not overwrite an existing field, but for any field which you want to edit, you'll need to either reenter the data or cut and paste it from earlier on in the output. It's annoying, but there's not a lot that we can do about it. In chapter 8, we'll extend our todo list and give it a web interface with Django, so proper editing will have to wait until then.

Let's go ahead and write a test which covers the functionality which we can add.

Listing 4.24 - testing todo editing

```
def test_edit_todo():
    todo.todos = [
        { 'title' : "Make some stuff",
          'description' : 'This is an important test',
          'level' : 'IMPORTANT'
        },
    ]

    response = todo.edit_todo(todo.todos,
        which="1",
        title="",
        description="Stuff needs to be programmed properly",
        level="")

    assert response == "Edited todo #1", response
    assert len(todo.todos) == 1
    assert todo.todos[0]['title'] == "Make some stuff"
    assert (todo.todos[0]['description']) ==
           "Stuff needs to be programmed properly"
    assert todo.todos[0]['level'] == "IMPORTANT"

    print "ok - edit todo"
#1 - Edit a todo
#2 - Test that the right fields were edited
```

Here's a function call which should edit a todo **(1)**. We're simulating blank entries with blank strings in our input arguments.

Now we test that the todo has the right fields **(2)**. Those which were blank should be unchanged and those which weren't should be set to the right values. We're also checking that we still have just one todo, and that we get the right response.

The other thing that we need to test is that editing the level of a todo will result in it being reordered. If a todo suddenly becomes important, we want it to appear at the start of our list, rather than still being halfway down. Listing 4.25 shows how we can test that.

Listing 4.25 - testing sort order after editing

```
def test_edit_importance():
    todo.todos = [
        { 'title' : 'test medium todo',
          'description' : 'This is a medium todo',
          'level' : 'medium'
        }
    ]
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

},
{ 'title' : 'test another medium todo',          #1
  'description' : 'This is another medium todo', #1
  'level' : 'medium'                           #1
},
]                                              #1

response = todo.edit_todo(todo.todos,           #2
    which="2",                                #2
    title="",                                 #2
    description="",                           #2
    level="Important")                      #2

assert todo.todos[0]['level'] == "IMPORTANT"   #3
assert todo.todos[1]['level'] == "medium"       #3

print "ok - edit importance"
#1 - Set up two medium todos
#2 - Edit the last todo in the list
#3 - The important todo should now appear first

```

First we set up two ‘Medium’ level todos (1). We edit the last todo and set its level to ‘Important’, but leave the other fields unchanged (2).

Now that the importance of the second todo has been changed, it should appear first in the list rather than second (3).

That covers the behavior that we’re expecting from editing a todo. Let’s see how we go about implementing it in our program.

Listing 4.26 - code to edit a todo

```

def edit_todo(todos, which, title, description, level):
    if not which.isdigit():
        return ("'" + which +
               "' needs to be the number of a todo!")
    which = int(which)
    if which < 1 or which > len(todos):
        return ("'" + str(which) +
               "' needs to be the number of a todo!")

    todo = todos[which-1]
    if title != "":
        todo['title'] = title
    if description != "":
        todo['description'] = description
    if level != "":
        todo['level'] = level

    sort_todos()
    return "Edited todo #" + str(which)
#1 - Check user input
#2 - Update the todo
#3 - Tidy up

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

We use exactly the same code that we're using in `delete_todo` to check user input **(1)**. We could probably pull it out and make it a function, but since we're only using it in two places, whether we do so or not is a line call. If you add a third function which uses this code, then it should definitely be separated.

Now we update the todo **(2)**. For any non-blank input, we override the field of the todo.

The final step is to sort our todos (since the level might have changed), and return a message to let the user know what's happened **(3)**.

We're done! All of the essential features that we set out at the start of the chapter have been completed, and we have a usable todo list program.

Note: Of course, the definition of "essential" will vary from person to person, but getting the core of your application in place will definitely help put the finishing touches on the rest of the essential parts.

Better yet, we have a comprehensive test suite which covers all of the major functionality of our application, so if we make any changes further down the track, we can easily check to make sure that our program still works.

4.8 Where to from here?

Like all of the programs in this book, your todo list program is now yours, and you can extend and enhance it to suit your own needs. While it's usable, there are some features that could dramatically improve it, and adding them would be a useful exercise. Here are some ideas for features which you could try to add.

4.8.1 Undo

When deleting or editing todo items, there's no way out if you make a mistake. You're only human, and it makes sense to try and allow for errors as much as possible, especially when deleting todo items. Once you've deleted one, there's no way to get it back.

One way to get around this would be to instead mark deleted todos instead of actually removing them from the list, and just don't display them under normal circumstances. If necessary you could use another command to display the todos which have been deleted (perhaps `showdeleted?`) and restore them (`restore`).

4.8.2 Different interface

You might find the prompt > response, prompt > response nature of the interface to be a bit annoying. It was designed to be as easy to program as possible, but that doesn't mean that it's as easy to use as possible. One alternative is to allow arguments after commands that the user types in. For example, instead of typing `delete <enter> 3 <enter>`, you could instead type `delete 3 <enter>` and have the program do the same thing. How essential this is will depend on whether you prefer the existing interface or not, but if you decide to add this type of interface the `shlex` module will be extremely useful.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

4.8.3 Time management and estimation

Another useful feature would be to record an estimate of how long you think a task will take to complete, then later marking items as done and recording the time that you've spent on them. Later you could generate a report showing where you've spent your time, and how accurate your initial estimates were. Being able to estimate the time it'll take to complete a task can be a useful skill, but it only improves if you practice and get feedback on how accurate your estimates were.

4.8.4 Study one of the unit testing frameworks

Unit testing in itself isn't particularly difficult, which is why we "rolled our own" in this chapter, but there are a number of unit testing modules which you can use. There are two key advantages in using them. The first is that they can help you organize your tests into test suites and classes and run them all automatically from multiple files, as well as run set up and tear down code before and after each test. The second is that they allow you to test a lot more than you can with just simple assert statements and will give you more detailed information when things go wrong. The three unit testing modules that you'll initially want to look at are `unittest` and `doctest`, both included with Python, and `py.test` which is a lighter weight version of `unittest`, available from <http://codespeak.net/py/dist/>.

4.9 Summary

In this chapter we learned about unit testing, saw first-hand how to use it to write programs and developed a large suite of unit tests so that we could extend our application without worrying about how we might break it if we change something. We also learned about some aspects of our program (mainly user input) that were harder to unit test, and how to work around that by keeping the untested sections of code as small and simple as possible.

We also learned that Python has first-class functions which can be assigned to variables in the same way as more basic types such as integers and strings, and one good way to make use of functions by assigning them as values in a dictionary. We'll learn more about first class functions in chapter 7. We used two more Python libraries, `pickle` and `textwrap`, and also discovered how we could filter our todo lists using list comprehensions, which are a simple but powerful way of filtering and processing lists.

The final thing that we covered indirectly was how we can work around problems that arise in development. Sometimes, as with editing our todo items, there's not much that can be done except to find a reasonable workaround. In other cases, like wrapping the text of our todos, some patience and persistence (and a decent suite of tests) can really pay off.

5

Business-oriented programming

In this chapter we're going to take a look at how Python can be used in the real world, to help you to do your job better and faster. As a sample project we'll take some stock data from a web page on the internet, extract the figures that we're interested in, and then see how we can make use of that data, report on it and send those reports to interested parties. To make your life easier all of this will be written so that it's easy to automate.

One of the critical tasks facing many programmers and system administrators is to make many different systems talk to each other. You might need to:

- read some data from one system
- compare it with the results from a second
- check that both of them make sense (often referred to as a 'sanity check')
- save the results for later use and then
- email relevant people with a report about what you found or
- any problems that you encountered.

Of course, people are depending on the information from these systems, so whatever you write it has to be robust. You can't just try something and hope for the best. Sound daunting? Don't worry - Python is at its heart a very practical language, and has a number of features and library modules to make interfacing with the real world and all of its quirks much easier.

Why Automate? The more selfish reason for wanting to automate is that once you've set your program up, you don't have to worry about it anymore, which frees you up to think about more important and interesting things.

We'll start by building our reporting program and then we'll look at what steps we can take to anticipate errors and make it bulletproof.

5.1 Making programs talk to each other

So how do we make programs talk to each other? Typically programs will have some sort of data input and output, so integrating two programs is normally a question of taking the output of one program, reading its data and then presenting that data in a format which the second program will understand. Ultimately you can chain lots of different programs together with Python acting as an interpreter. The system that we'll be building looks something like Figure 5.1.

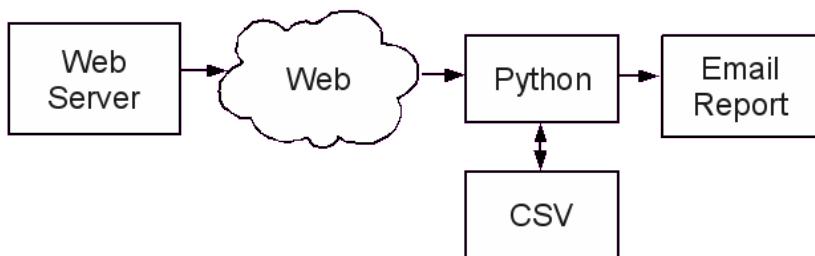


Figure 5.1 Python as a glue language

Programs for tasks like this are normally referred to as 'glue code', since you're gluing two or more programs together into one system.

5.1.1 CSV to the rescue!

The process of gluing programs together is made much easier if you have a common data format - a data 'language' that all of the programs in question speak. The format which is closest to being a lingua franca of data exchange is the humble CSV (comma separated value) file, which is just a simple spreadsheet format, consisting of a header line and a number of rows afterwards. The items on the rows are separated by commas, hence the term comma separated value. Some CSV files will use other character values, such as tabs, to separate their values, but the principle is the same.

There are many advantages to using CSV files. CSV is a simple and straightforward format, which is important when developing or debugging your system. If you run into problems, you can just read the file in a text editor. Most programming languages will have a

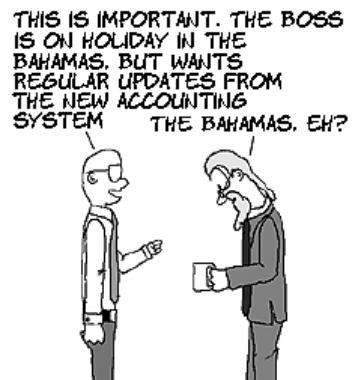


©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

library to read and write CSV, and a lot of programs use CSV as an import or export format too, so you can reuse all of the routines that you write for one program on the next one. Finally, it also maps reasonably well onto most data - you can even think of it as an SQL table - so it's generally useful in most cases.

A nice feature of using CSV is that most spreadsheet programs such as Excel can import it easily and you can then use them to generate graphs or color coded charts. An important warning though: many spreadsheet programs will convert data into their internal formats when they import, which means that your data may get silently corrupted. This is particularly important for anything that looks like a date, or a string which looks like a number. An employee id of 00073261, for example, would get converted to the number 73,261. Where ever possible, it's best to just use Excel to view data and consider any data which it outputs as 'tainted'. Don't use it for any further work - just the original CSV file.



TIP If you need to get work done quickly, it's often quicker to build on systems that already exist. Python will let you email a report from one system to your program, along with some data from a web page, and dump it into a .csv file or database.

5.1.2 Other formats

As well as CSV, Python has libraries to read many other formats, all of which can be used in data exchange in some way. Here's a quick list of the most common ones, but there are many others available either in Python's standard library or as add on packages which you can install.

HTML

You might not think it, but HTML is a data format, and most programming languages have libraries which let you write programs that behave as if they were web browsers, reading HTML and posting data back via HTTP POST or GET requests. Python has several libraries available to download and interpret web pages and send data back in this way. In the next section we'll look at how we can download a web page using Python's built-in `urllib` library and then extract stock prices from it using an add-on module called Beautiful Soup.

JSON, YAML AND MICROFORMATS

If you need more structured data, such as a nested tree or a network of objects, then CSV might not be the best fit. Other formats, such as JSON or YAML are more general

SQLITE

If you might be upgrading your data storage to a database or you need your data access to be fast, then you might want to consider SQLite, which is included in the Python standard

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

library as of version 2.6. It provides a subset of the SQL commands that you would expect to find in databases such as MySQL or PostgreSQL, and saves its data to a local file. Many programs such as Mozilla, Skype and the iPhone use SQLite as a data storage format.

MAILBOXES (MBOX)

Python is also capable of reading most common mailbox formats such as mbox or maildir, and parsing the messages within them, including extracting attachments and reading multi-part MIME messages. Anything that you receive via email can be read, interpreted and acted upon. Python is also capable of acting as a normal mail reader, via add-on libraries like getmail, and can send emails back out via SMTP.

XML

Python supports reading, writing and parsing XML files, as well as XMLRPC (XML Remote Procedure Call) services. The latest version of Python, version 2.6, includes ElementTree, which is a very easy and powerful library for dealing with XML.



Any program that you need to interface with will have its own way of doing things, so it's important to know what libraries are available to output the formats that the program wants, and conversely to read in the formats that it outputs. Fortunately Python can handle a wide variety of formats, very easily. Let's move on and take a look at the tools that we'll be using in this chapter.

5.2 Getting started

Our first task is to have a look at the data that's exported by the program that we want to interface with. In our case, we'd like to interface with Yahoo's stock tracking site, which you can access at this link: <http://finance.yahoo.com/q?s=GOOG>, and report on some of the statistics of the stock price over time. That link gives you the results for Google, but feel free to pick a different one, such as IBM or AAPL (Apple). You might be called upon to interface with an entirely different site, but the general principles here will hold. We'll be using two main tools when parsing - BeautifulSoup, to let Python read HTML and Firebug, to help us inspect the site's HTML and figure out which elements we want to extract.

5.2.1 Installing BeautifulSoup

Beautiful Soup is a Python library which is designed to be easy to use, but also to handle a wide variety of HTML markup, including "pathologically bad" markup. Often you won't have a choice about which page you want to scrape, so it pays to pick a library like BeautifulSoup which isn't too fussy about the HTML that it's given.

Beautiful Soup is available from <http://www.crummy.com/software/BeautifulSoup/>. I would recommend that you download the latest version of the 3.0.x series, since at the time of writing there were some issues with the 3.1 version. To install, download the file to your ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

desktop and unzip it, then from within a command prompt window type cd Desktop\BeautifulSoup-3.0.x followed by python setup.py install. BeautifulSoup will then install itself into Python's site-packages folder so that you can use it from anywhere. To make sure that it's installed properly, open a Python command prompt and type import BeautifulSoup. If there's no error, you're good to go!

5.2.2 *Installing Firefox and Firebug*

The other tool that we need is Firefox, which is a more open and standards compliant browser than Internet Explorer. That'll help us when we're looking at the code of web pages. You can get Firefox from <http://getfirefox.com/>.

Firebug provides a lot of extra development features for the Firefox web browser. It isn't essential for our task, but it does make interacting with the HTML of web pages a lot easier. You can download it by visiting <http://getfirebug.com/> in Firefox and clicking on the big 'install firebug' button. You might need to change your settings to allow Firefox to install from that particular site, but other than that everything should be automatic. When you're done and Firefox has restarted, you'll see a small bug icon in the bottom right hand corner of your Firefox window, and you'll have some extra options when you right click on some elements of your page.

5.2.3 *Examining the page*

Now that we've got Firebug installed, we can have a look at the elements that we'd like to be able to export in our Python script. If you right-click on a section of the page, such as the title of the stock, and select 'Inspect Element', the bottom half of the window should open and show you the HTML corresponding to the title.

It'll be something like:

```
<div class="yfi_quote_summary">
  <div class="hd">
    <h2>Google Inc.</h2>
    <span>(NasdaqGS: GOOG)</span>
  </div>
  ...
</div>
```

HI PITR, SID SAYS THAT I'M
HELPING YOU WITH AN EMAIL
SCRIPT OR SOMETHING.
WHERE DO WE START?

FIRST TO BE LEARNINK
ABOUT OUR FOE



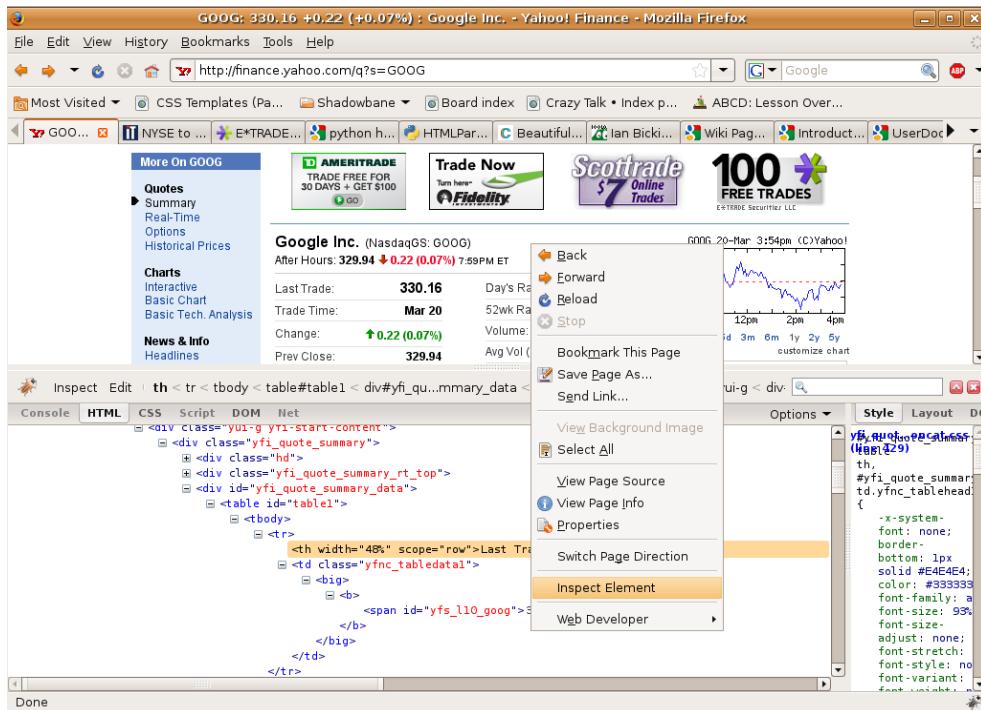


Figure 5.2 - Examining elements using Firebug.

You can use a similar process when examining other elements of the page to find out what their HTML looks like. If you're not sure which parts of the HTML correspond to particular elements of the page, you can hover your mouse over either the HTML or the element that you're interested in. Firebug will highlight the relevant sections of the page, as you can see in figure 5.3.

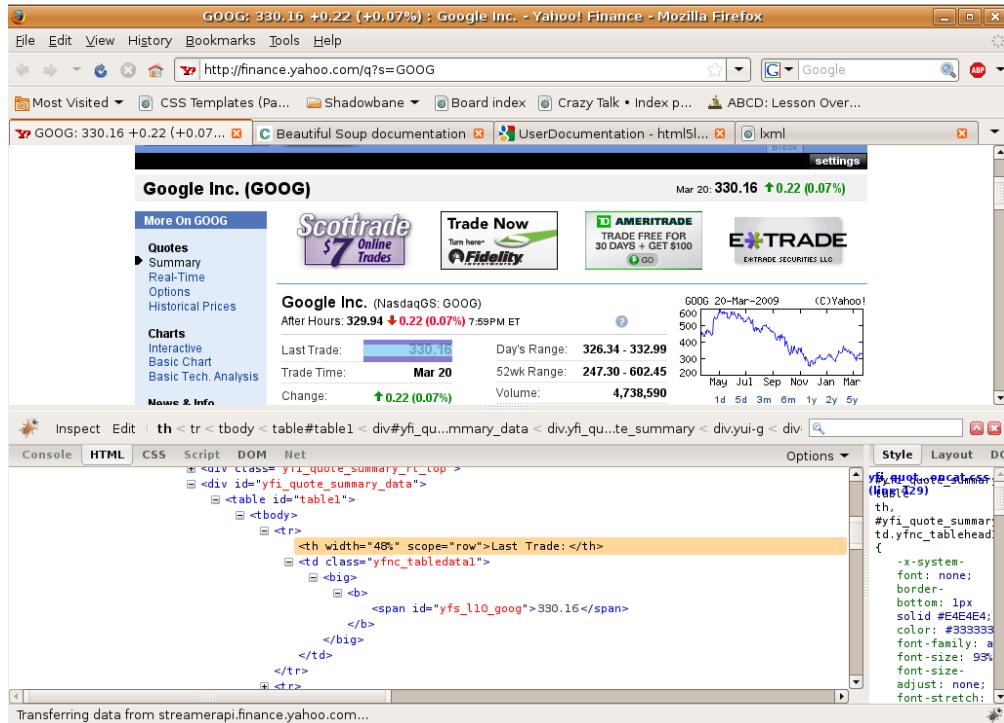


Figure 5.3 - Using Firebug with highlighting

Now that you know how to use Firebug to examine a page and find the elements that you're looking for, extracting data from the HTML is a lot easier.

5.3 Downloading the page with Python

We'll start out by downloading the entire page using Python's `urllib2` module. We'll do this by writing a function which will return the HTML code for any stock page that we name. This will be an easily reusable function which we can paste directly into our final script.

WELL, ACCORDING TO THEIR WEBSITE, THEIR SYSTEM COMBINES SEVERAL BEST-OF-BREED SYSTEMS INTO A DYNAMIC WHOLE

MEANS IS HAVING TO INTEGRATE MANY CRAPPY SCHLOTTKAS AT SAME TIME



Listing 5.1 - Downloading a web page

```
import urllib2

def get_stock_html(ticker_name):
    opener = urllib2.build_opener(
        urllib2.HTTPRedirectHandler(), #1
        urllib2.HTTPHandler(debuglevel=0), #1
        # etc.
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        )
opener.addheaders = [                                #2
    ('User-agent',                                #2
     "Mozilla/4.0 (compatible; MSIE 7.0; "
     "Windows NT 5.1; .NET CLR 2.0.50727; "
     ".NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)")  #2
]

url = "http://finance.yahoo.com/q?s=" + ticker_name      #3
response = opener.open(url)                            #3
return ''.join(response.readlines())                  #3

if __name__ == '__main__':                             #4
    print get_stock_html('GOOG')                      #4

#1 - Creating an opener object
#2 - Adding headers to our request
#3 - Read a web page with our opener
#4 - Calling our function

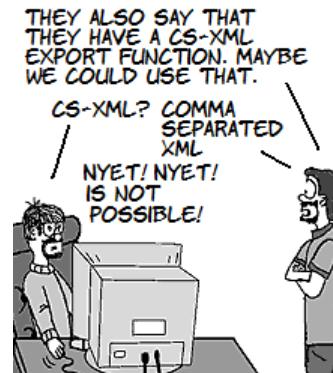
```

`urllib` uses `opener` objects to read web pages. Here we're creating one **(1)** and feeding it two handlers, which are objects that handle certain types of HTTP responses from the web server. `HTTPRedirectHandler` will automatically follow redirects, so if a page has moved temporarily you don't have to worry about writing code to follow it, and `HTTPHandler` will read any web pages that are returned.

Unfortunately, some websites like to block automated agents like ours, so to be on the safe side we're being sneaky here and setting the User-agent that we send to the server so that we appear to be a completely different web browser **(2)**. In this case we're pretending to be Internet Explorer 7 running on Windows XP. You can find other user agent strings by doing a web search for 'user agent strings'.

Now all we need to do is to be able to read a web page is call the `opener`'s `open` method with a URL **(3)**. That method returns a 'file like' object, which responds exactly as if it's an open file, so we can get the text of the web page by calling `readlines` and joining its response together.

It's easy to call our function now **(4)**, and all of the tricky `urllib` parts are hidden away. If you run this script, it'll print out the entire contents of the <http://finance.yahoo.com/q?s=GOOG> page on the screen.



urllib in Python 3.0 In Python 3.0, the `urllib`, `urllib2`, `urlparse` and `robotparse` modules have all been merged into `urllib`, and several improvements have been made. The methods that we're using here have been moved into the `urllib.request` module, but other than that they're the same.

Of course, the entire content of the page is a little bit much for what we're trying to do. We're only interested in the part which has the stock price. We need to limit our result to just the section of the page that we're interested in.

5.3.1 Chopping out the bit that we need

Let's get our feet wet with Beautiful Soup and parse out just the quote element that we're interested in and print it. Once we've done that, we can start pulling out individual elements for our final output.

Most of the time you can simplify your parsing by looking for landmarks in the web page's HTML. Normally there will be 'id' and 'class' attributes which you can use to pinpoint a particular section, and then narrow down your search from there. In our case, it looks like there's a `<div>` element with a class of "yfi_quote_summary" which contains all of the information we need. Here's a function which uses Beautiful Soup to pull just that section out of our stock page.

Listing 5.2 - Finding the quote section

```
from BeautifulSoup import BeautifulSoup
...
def find_quote_section(html):
    soup = BeautifulSoup(html)
    quote = soup.find('div',
                       attrs={'class': 'yfi_quote_summary'})
    return quote

if __name__ == '__main__':
    html = get_stock_html('GOOG')
    print find_quote_section(html)

#1 - Create a parse object
#2 - Find the yfi_quote_summary element
#3 - Print the quote_section
```

The first thing that we need to do when parsing HTML is to create a Beautiful Soup object **(1)**. This object looks at all of the HTML that it's fed and provides lots of methods for you to examine, search through and navigate it.

The soup object provides a `find` method, which can quickly search through the HTML. Here we're finding all of the `<div>` elements which also have a class of 'yfi_quote_summary' **(2)**. The `find` command returns the first element that it finds which matches the criteria, but as another soup object, so you can perform further searching if you need to.

As a shortcut, if you print a soup object **(3)** it'll just return a string containing its HTML. In our case, this is exactly what we're looking for - the HTML of the 'yfi_quote_summary' div.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

If you run this script, it should print out a much shorter piece of HTML, which is the quote section that we're looking for. You should be able to see some sections like the stock name and price, and some of the other `<div>` elements. Let's now add another function which will take the soup object for the summary div and produce more meaningful data.

5.3.2 Adding extra information

Now that we have our smaller section of HTML, we can examine it further and pull out the specific parts that we need. The `find` command will return another soup object, so we don't have to worry about parsing it again - we can just call the `find` method on the results to extract the data that we need. Listing 5.3 shows a function which uses a number of `find` calls to build a dictionary of data from our stock page.

Listing 5.3 - Extracting the data for the stock

```
def parse_stock_html(html, ticker_name):
    quote = find_quote_section(html)
    result = {}
    tick = ticker_name.lower() #3

    # <h2>Google Inc.</h2> #1
    result['stock_name'] = quote.find('h2').contents[0] #2

    ### After hours values
    # <span id="yfs_l91_goog">329.94</span> #3
    result['ah_price'] = quote.find('span', #3
                                    attrs={'id': 'yfs_l91_'+tick}).string #3

    # <span id="yfs_z08_goog">
    #     <span class="yfi-price-change-down">0.22</span> #4
    result['ah_change'] = (quote.find( #4
                                    attrs={'id': 'yfs_z08_'+tick}).contents[1]) #4

    ### Current values
    # <span id="yfs_l10_goog">330.16</span>
    result['last_trade'] = quote.find( #4
                                    'span', attrs={'id': 'yfs_l10_'+tick}).string #4

    # <span id="yfs_c10_goog" class="yfi_quote_price">
    #     <span class="yfi-price-change-down">1.06</span>
    def is_price_change(value): #5
        return (value is not None and #5
                value.strip().lower() #5
                .startswith('yfi-price-change')) #5

    result['change'] = ( #6
        quote.find(attrs={'id': 'yfs_c10_'+tick}) #6
        .find(attrs={'class': is_price_change}) #6
        .string) #6

    return result

if __name__ == '__main__':
    html = get_stock_html('GOOG')
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

print parse_stock_html(html, 'GOOG')

#1 - Including elements as comments
#2 - A simple find
#3 - A more involved find
#4 - Differences between Beautiful Soup and firebug
#5 - A function to locate elements
#6 - Using our function

```

Does this look familiar? It's our old divide and conquer strategy again - write something simple that works, and then refine it a bit at a time until you have the data that you need.

You might find it helpful to include the HTML that you're trying to match as comments, like **(1)**. It saves switching back and forth between your editor window and your web browser to remind yourself what the HTML looks like.

Next we run a simple find over our quote summary to find the first h2 element **(2)**. Once we've done that, we get the first element from the .contents attribute, which in this case will be the name of our stock. The .contents attribute returns all of the sub-elements within a particular element as a list of soup objects.

Notice that in the html, the ids that we're looking for are named after the company. That's not much of a problem, since we can just pass in the ticker name and lower case it **(3)**. We're also using the .string method. If you're certain that there'll only ever be one text node within your search results, you can use the .string shortcut, which will return that node as text.

If you have a close look at the search here and the corresponding HTML in Firebug **(4)**, you might notice that they're different. The code seems to be ignoring the extra span that you can see in the browser. The answer is that sometimes when HTML is invalid, firebug will insert extra elements to make the HTML code valid. That's not a problem for Beautiful Soup though, which just returns both the image and text as two elements. If in doubt, you can always 'view source' and search for the id or class of the element to see the HTML exactly as you received it from the server.

If you need more flexibility in how you search, then another way that you can use Beautiful Soup's find method is to use a function instead of a string **(5)**. Beautiful Soup will feed the function the attribute name - if the function returns True, then the element is included.

Using the function from **(5)** when searching is easy - just use the function **(6)** instead of the string. In this section we're also 'chaining' find calls together. The first find looks for elements with an id of 'yfs_c10_goog' and returns another Beautiful Soup object, which we use to immediately run another find command. The whole set of calls is contained in brackets so that we can wrap it over multiple lines and make it easier to understand.

**SO NOW THAT WE HAVE THAT,
ALL WE NEED TO DO IS PASS
IT TO AN EXCEL FOR
CALCULATING, THEN WRITE IT
OUT TO PLAIN TEXT AND
EMAIL IT OFF TO THE BOSS'
IPHONE**



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

You can continue in this vein until you've extracted all of the data that you need from the page. Just be careful that your parsing doesn't grow too unwieldy. If it does, you may want to consider breaking `parse_stock_html` up into functions, one per data value, and loop over a dictionary of data value names and functions when you're parsing:

```
parse_items = {'stock_name': parse_stock_name,
               'ah_price': parse_ah_price, ... }
```

5.3.3 Caveats for web scraping

While reading data directly from the web is a very useful tool, it's not without its drawbacks. The main issue is that web pages change frequently, and your parsing code may need to change with it. You can mitigate the risk somewhat by focusing on the elements of the page least likely to change, such as id or class variables, but you're still at the mercy of whoever creates the page. If at all possible, it's usually much better long term if you can rely on official channels, such as a published API for accessing data, rather than doing it all yourself. Later we'll look at strategies for dealing with failures in your script and how you can mitigate them.

First, however, we need to add some complexity to our tool.

5.4 Writing out to a CSV file

An individual stock price is not very useful. To make any recommendations about whether it's good to buy or sell, or what the stock is likely to do in the future, we'd like some history of the stock price and its movement. That means that we need to save the data that we've just read so that we can use it again in future. As we said in section 5.1.1, the most common data format is a CSV file, so let's write some code to save our results dictionary to a row in a CSV file.

Listing 5.4 - Writing a CSV file

```
import csv

field_order = ['date', 'last_trade', 'change',
               'ah_price', 'ah_change'] #1
fields = {'date' : 'Date', #1
          'last_trade' : 'Last Trade', #1
          'change' : 'Change', #1
          'ah_price' : 'After Hours Price', #1
          'ah_change' : 'After Hours Change'} #1

def write_row(ticker_name, stock_values):
    file_name = "stocktracker-" + ticker_name + ".csv" #2
    if os.access(file_name, os.F_OK): #2
        file_mode = 'ab' #2
    else: #2
        file_mode = 'wb'

    csv_writer = csv.DictWriter( #3
        open(file_name, file_mode), #3
        fieldnames=field_order, #3
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        extrasaction='ignore' )                                #3

    if file_mode == 'wb':
        csv_writer.writerow(fields)                         #4
        csv_writer.writerow(stock_values)                  #4
        csv_writer.writerow(stock_values)                  #4

if __name__ == '__main__':
    html = get_stock_html('GOOG')
    stock = parse_stock_html(html)
    stock['date'] = time.strftime("%Y-%m-%d %H:%M")      #5
    write_row(ticker_name, stock)
    print stock
#1 - A field to header mapping
#2 - Look for an existing .csv file
#3 - Create a csv.DictWriter object
#4 - Write our rows
#5 - Include a date

```

Before we start creating our csv file, we need to know which key-value pairs in the dictionary correspond to which headers in the csv file (**1**). Storing them in a dictionary means that we can easily access them later on. Dictionaries aren't guaranteed to be in a specific order though, so there's another list to tell us what order the columns should be in.

When we write our file, we'll name it after the stock that we're tracking, so that it's easier to find, and so that any other scripts will be able to easily access it. We also need to know whether we've written to this file before, so that we can add the headers to it if necessary. `os.access` does the trick, and we just need to know whether it exists (**2**).

`csv.DictWriter` is a class which writes dictionaries into a csv file (**3**). It needs two arguments to function - a file opened in binary mode and a list of the fields in the order that they should appear in the csv file. I've also added an `extrasaction` argument, which tells `DictWriter` whether it should ignore extra values in the dictionary, or raise an exception. In our case we have an extra 'stock_name' field which we'd rather not have appear over and over again in the csv file, so we'll just ignore it.

Once the `DictWriter` object is created, using it to write a row is easy (**4**) - just feed it a dictionary to write. If any keys are missing though, you'll get an error raised.

We're also interested in when a particular stock record was retrieved. In Python, here's how you output the current local time and date (**5**). The `%Y %H` parts of the string will be replaced with the current year, hour, etc. You can arrange them in any order you like, as long as you keep the `%` signs together with their corresponding character.

Now you have a csv file which will be updated every time you run your script. If you run it several times, you'll see extra lines being appended to the end. Typically you'd automate this script using cron (if
©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>



you're using Linux or Mac) or Windows Scheduler or similar if you're running on Windows. You can stop at this point for some scripts, but if the results are important, you'll want to make sure that other people know about them.

Next let's figure out how to create an e-mail with our .csv file.

5.5 E-mailing the csv file

If you need to do anything with email, the `email` module is normally the place to start. It contains classes and functions for parsing email and extracting their information as well as tools for creating and encoding emails, even the creation of multipart emails containing HTML, text (if the recipient can't read HTML) and attachments. Normally we'd start with a simple section and build up, but when creating an email it's easier just to remove the sections that you don't need.

Creating an email is straightforward, but it definitely helps if you have some background knowledge of how emails work. Let's take a look at that first, and then we'll see how to put it into practice in our program.

5.5.1 Email structure

Most emails, other than the simplest plain text emails, are composed of containers, with parts within them. These parts can be text, HTML or any other part which can be described with a MIME type. When the email is sent, the email structure will be converted into a plain text format that can be reassembled when it reaches its destination.

Normally there'll be at least two parts, one which contains your email in HTML and another one which contains a text version, but an email can in theory contain as many different parts as you need. The structure which I've found most useful, and which displays the best across a variety of email programs, is in figure 5.4 below.

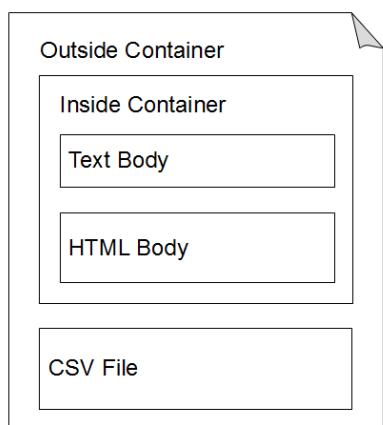


Figure 5.4 The structure of a HTML email

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

This structure has two containers. The outside one contains the message part and any number of attachments, and the inside container has the two versions of your email. If you need extra attachments, just attach them to the outside container.

5.5.2 Creating an email

Now that we know how MIME messages are constructed, let's take a look at the corresponding program. This function will take an email address and a stock ticker name such as 'GOOG', and construct an email ready to be sent.

I SEE. AND THEN?

WELL, THEN WE'LL BE
ABLE TO SEND IT OUT
TO THE SPREADSHEET.
THEN TEXT FILES...
I

AND THEN OFF TO THE
BOSS? OK, LET ME
KNOW HOW IT GOES...



Listing 5.5 - Creating a MIME email

```
from email.mime.multipart import MIME Multipart
from email.mime.text import MIMEText
...
def mail_report(to, ticker_name):
    outer = MIME Multipart()
    outer['Subject'] = "Stock report for " + ticker_name #1
    outer['From'] = "me@example.com" #1
    outer['To'] = to #1

    # Internal text container
    inner = MIME Multipart('alternative') #2
    text = "Here is the stock report for " + ticker_name #2
    html = """\ #2
<html>
<head></head> #2
<body> #2
    <p>Here is the stock report for #2
        <b>"""+ticker_name+"""</b> #2
    </p> #2
</body> #2
</html> #2
"""
    part1 = MIMEText(text, 'plain') #3
    part2 = MIMEText(html, 'html') #3
    inner.attach(part1) #3
    inner.attach(part2) #3
    outer.attach(msg) #3

    filename = 'stocktracker-GOOG.csv' #4
    csv_text = ''.join(file(filename).readlines()) #4
    csv_part = MIMEText(csv_text, 'csv') #4
    csv_part.add_header('Content-Disposition', #4
        'attachment', filename=filename) #4
    outer.attach(csv_part) #4

    # print outer.as_string() #5
    send_message(outer)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

- #1 - Create the external container
- #2 - Create the body of the email
- #3 - Attach the body to the external container
- #4 - Create the CSV part and attach it
- #5 - Send the email

The first thing that we need to do is to create the external container (1) which will contain all of the other parts. It's also where we put all of the message headers - the Subject, To and From lines.

Next up is the body of the email (2), the HTML and text parts. Normally an email program will display the last part of this container as the body, and fall back on the others if it can't handle it, so we put the HTML last.

Now we can create `MIMEText` objects to hold our email body. They'll automatically be of type 'text/something', and the second argument tells it what that something will be. Once we have those objects, we call `attach` to insert them into the inner container, and then attach the inner container to the outer one (3).

Now we do the same thing for the csv section that we did for the body of the email. Once we've read in the csv file, we create a 'text/csv' part and insert it into the outer section (4). The only extra thing that we need to do is to add a 'Content-Disposition' header to say that it's an attachment, and to give it a file name. Without a file name, you'll get a default name like 'Part 1.2', which doesn't look very friendly or professional.

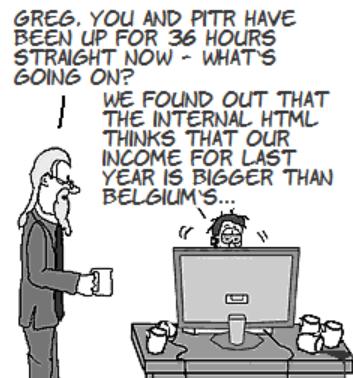
If you want to see what you've created, just use the `as_string` method on your outer message object, and it'll print out the email exactly as it'll be sent (5). We'll be writing the `send_message` function in the next section, so don't worry about that part for now.

That's all that you need to do with the report email - it's ready to be sent. If you'd like to reuse this function there are a number of things you can do to extend it. The first obvious one is to pass in the subject, and text and HTML content as arguments, instead of hard coding them in the body. Another is to be able to pass in more than one attachment, as a list. An important function for this second part is `mimetypes.guess_type`, which will give you a MIME type and an encoding (such as zip, gzip or compress) based on the filename of the attachment. From there, you can create the right type of MIME object such as `MIMEApplication` or `MIMEImage` and attach it to the email.

By the way, if you're attaching images, you can link to them from within the HTML body by using a `cid: url`, like this: ``.

5.5.3 *Sending Email*

The last thing that we need to do with our email is actually send it. This is the most straightforward part of the email sending process, and only needs a 'From' address, a list of 'To' addresses, and the email itself.



Listing 5.6 - Sending an email

```

import smtplib
...
def send_message(message):
    s = smtplib.SMTP('mail.yourisp.com')                      #1
    s.sendmail(                                              #2
        message['From'],
        message['To'],
        message.as_string())                                 #2
    s.close()                                                 #3

if __name__ == '__main__':
    html = get_stock_html('GOOG')
    stock = parse_stock_html(html)
    stock['date'] = time.strftime("%Y-%m-%d %H:%M")
    write_row(ticker_name, stock)
    mail_report('youremail@example.com', ticker_name)      #4
    print stock
#1 - Create an SMTP sender
#2 - Use the sender the send an email
#3 - Close the connection
#4 - Include the mail_report function

```

First we create an SMTP object, which will handle the sending of our email **(1)**. This will normally be enough if you're connecting to your Internet Service Provider's mail server, but if they need a password then you should include a username and password with a line like `s.login('user', 'password')`

Once you have an SMTP object, you can call its `sendmail` method to send email **(2)**. We're pulling the email addresses as well as the body of the email out of the message, that way we don't have to specify them as separate arguments and our code stays neater. You can call the `sendmail` method multiple times if you need to.

Once we're done we can close our connection **(3)**. This saves some load on the SMTP server, since it'll have one less connection to keep track of, but if you have multiple email messages to send though, it's better to reuse the connection.

To start emailing reports whenever your program is run, just include the function here and feed it your email address and the name of the stock that you want to report on **(4)**. If you'd prefer a weekly report then you could include this in a second separate script instead. If you'd like to email multiple people at once then you still feed one string here, but with commas separating the email addresses.

That's all that we need to do for our script. It scrapes data from a web page, posts it into a CSV file, and then emails a report to someone who can make use of the information. A surprisingly large amount of business programming boils down to a roughly similar process: gather some data, process it and then send or store the results somewhere, either to a human who needs the information, or to another program.

5.5.4 Other email modules

Although we didn't need them in this script, there are other email-related modules that you can make use of if you need more flexibility when dealing with email, or if you need to do something outside what this script can do. The two that I most commonly use, other than the modules in this script, are the `mailbox` module and `getmail`.

The `mailbox` module contains classes to read several different types of mailbox, including the two most common, `mbox` and `maildir`, and provides an easy way to loop over each message in the file. Parsing `mbox` files is relatively easy, but there are several catches to it, and it's easier just to use a library. As well as writing emails, the `email` module provides the `email.parser` to read the header lines, body and attachments out of a flat text file. Together they provide everything that you need to be able to handle email.

`Getmail` is an add-on module written by Charles Cazabon and available from <http://pyropus.ca/software/getmail/>. It handles POP and IMAP4, including via SSL, and can save messages to `mbox` or `maildir` storage as well as passing them to another program. It's also very easy to use, only needing one configuration file to work.

Between Python's built-in email modules and `getmail`, you should be able to deal with almost any email programming problem that comes your way, whether you need to read, download, parse or analyze email.

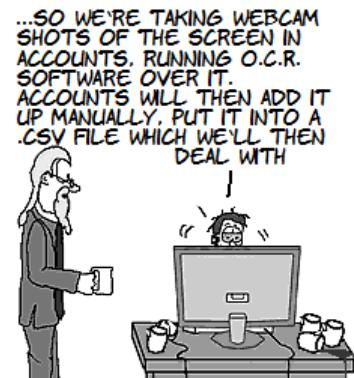
5.6 A simple script - what could possibly go wrong?

There's a very useful question that you can ask yourself when completing a project like this: "Are you done?"

Try it now and see what you think the answer is. Are you done? Would you be confident that you could run this script every day and not have to worry about it? Would it ever break? If the CEO or Director of your company was relying on the results of your script, would you be able to sleep at night? Even if your script isn't vital, how would you know that it was working? Would you have to babysit your script to make sure that it worked, perhaps checking the results every few days to make sure that nothing had gone wrong?

It's possible to write a program that works at first, but needs so much assistance to run that you might as well have not written it in the first place. To keep your sanity, try to analyze your program and find as many potential failure points as you can.

NOTE This is the toughest part of writing programs - anything and everything might fail in some way, and you have to be ready for the potential consequences. If you've ever wondered why a lot of programmers and system admins seem like cagey pessimists, now you know.



I've outlined some possibilities below for issues which might break our script. In the next section we'll look at how we can go about fixing them.

5.6.1 No internet

Obviously, if there's no internet connection, the script won't be able to download the stock page or send email, and there'll be nothing that you can do about it. But what happens exactly? Will your script fail immediately, or will it get halfway and corrupt your data? If you can't connect for a day, what should appear in the CSV file?

5.6.2 Invalid data

If Yahoo decided to change the design of their site, what would happen to your script? If it's expecting a particular id within the HTML and that id is removed, then your script will break. Alternately, there could be a partial outage and you'll see null or zero values. Or if the server is under load you might see timeout errors or only receive half a page. How does your script handle that? Does it try to parse the error page and fail? Or does it recognize what's happened? In the worst case, it'll have data which looks similar to the data that you're expecting, you won't notice that it's changed, and your data will be silently corrupted.

THAT SOUNDS ... COMPLICATED.
WHAT HAPPENS IF -



5.6.3 Data that you haven't thought of

There's another failure mode related to invalid data - sometimes you can be given data which is valid, but just within the range that you're expecting. It might also be formatted or presented differently if it's within a certain range or currently unknown. These sorts of values are generally known as 'edge cases'. They don't happen very often, so they can be harder to predict, but they still have a large effect on the stability of your program. The best way to deal with edge cases is to try and consider the entirety of the range of your data and include any cases which are in doubt into your test suite.

5.6.4 Unable to write data

When we're processing, we're assuming that we'll be able to write to the CSV file. This is normally the case, but there are some circumstances where you might not be able to - if an administrator on the site has set the wrong permissions, or your computer is out of space. You might want to consider rotating your CSV file every so often, zipping the old ones and deleting any which are older than a certain date.

5.6.5 No mail server

You can also run into problems when trying to send your email. Most of the time email is pretty foolproof, but it is possible for a mail server to be down. If that's the case, what happens to your script? It might be enough for it to store the row in the CSV file and resend

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

it the next night, or you might need to check that the mail server is up and try an alternative route if it isn't.

5.6.6 You don't have to fix them

These are by no means the only things that can go wrong with your script, but they're the most likely. Depending on your script, its purpose and the environment that it runs in, these might be more or less of an issue, or you might not need to worry about them at all, but you still need to consider them.

Let's go on now and see how we might solve, or at least mitigate, some of these issues in practice.

5.7 How to deal with breaking scripts

There are a number of strategies for dealing with the weak points that we've seen so far in our script. Which one you choose depends on the nature of your script and its purpose. First, let's look at two factors that affect how you program your script as well as how you look at potential failures and how you solve them.

5.7.1 Communication

When you're building software for other people, communication is vital. It's important to know the overall goals of your project, how it impacts on other aspects of the business, what the likely effects of a failure are and how people in the business will use your final product. While not strictly speaking a bug, a program that doesn't do what's needed might as well not have been written.

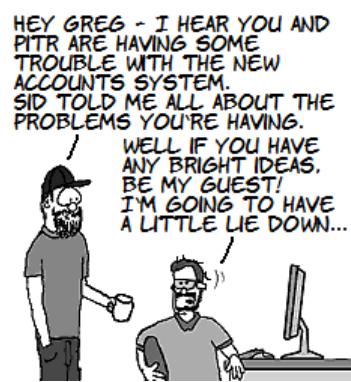
It's also important to keep people informed as you build your program, since the problem that you're solving can change at any point. There's nothing worse than finishing a program only to find that it's no longer required and that weeks of effort have been wasted.

5.7.2 Tolerance of failure

There are many different ways to deal with a potential error, and they all have varying costs. Which ones you choose will depend on the businesses' tolerance of failure.

THE EXPENSIVE CASE

For example, if the business were using our script to buy and sell millions of dollars worth of stock then we'd have a very low tolerance of any possible failure. We might host it on a dedicated server or multiple servers in separate locations, since an extra several thousand dollars would be a small price compared to the risk of losing millions of dollars worth of trades. We'd also want to pay a few hundred dollars a month to access an API specifically provided for the purpose rather than scraping web pages,



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

and have a full suite of functional and unit tests to catch any errors.

THE CHEAPER CASE

If instead we were using our script as a more general business intelligence application, a stock failing here or there or taking a day to propagate through the system might not be so bad. Cost is more of an issue, so you'd run your script on a server with several other applications. This opens you up to the possibility of extra errors, like running out of disk space or having one application use so much CPU that nothing else can get anything done, but the impact of any errors like this is minor compared to the relative cost of separate servers.

5.7.3 *Don't break in the first place*

It might sound obvious, but the easiest way to avoid bugs in your program is not to write them in the first place. It's very easy to throw together a script that looks like it works, but often you'll find that there are all sorts of issues lurking in your code, just waiting for an opportunity to crash your program.

First, consider all of the possible data, including weird pathological cases, when you write your program. Look for edge cases and things which "can't possibly happen", and make sure that they can't. If you're working with numbers, what happens when the number is zero? or negative? or enormous? Should the program throw an error? ignore that particular value? Thinking about this ahead of time is easier than thinking about it when your program crashes and you have to fix it **right now**.

Once you know what data you can and can't handle, you can include it in your tests. Your unit tests and functional tests can verify what happens when you give a program data which is either invalid ("fruit" when you were expecting a number) or likely to be a problem (zero, negative or very large). If you find input which gives you an error while you're testing or when your program goes live, you can add it to your test suite to ensure that it doesn't happen again.

5.7.4 *Fail early and loudly*

If at some point in your program there's an error, normally the best way to deal with it is for your program to stop immediately and start "yelling" (via email or by printing to the screen). This is particularly true during development or if the problem is unexpected. Trying to soldier on in the face of errors is dangerous, since you can overwrite important data with nonsense results.

Wherever possible, check data and any error codes returned from the libraries that you're using. If you're trying to load data from the web, you can check the response code - anything other than 200 (success) means that there's been an error somewhere and you should

HMM - WHAT MAKES YOU THINK
THE CHIEF WILL BE ABLE TO
CONFIGURE HIS IPHONE'S EMAIL
FROM BARBADOS?
I THINK A WEB BASED
INTERFACE MIGHT BE
BETTER...



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

stop. If you have trouble parsing the data returned, it's quite possible that you're seeing a different type of page, or that the data is not what you're expecting. In that case, it's also a good idea to log it somewhere and skip processing. Don't forget to include relevant data in the error so that you can replicate the problem.

5.7.5 Belt and braces

To mitigate the effects of any errors, it can often help to have multiple fallbacks if things go wrong. For example, you might have two copies of your script running on separate servers. If something goes wrong with one script such as the network being unavailable, the other script may still be able to access the data.

Another tip is to save intermediate copies of the data whenever possible. In our script, you might want to consider saving the HTML that you download from the server before you analyze it. If some of the data looks odd, or you have an error when parsing, you can double check your results and see what's gone wrong.

5.7.6 Stress and Performance Testing

A common problem when your program goes live is that it works well on small amounts of data, but fails or runs too slowly when used on real data. Make sure that your program can handle the workload in the volumes that are expected when it goes live, and use real data when testing wherever possible.

5.7.7 Try again later

If your program fails due to an outside source not being available, you can often try again several times before giving up. Perhaps the site that you're trying to load is having some temporary downtime, and will be back up in a few minutes. If you take this route be sure to wait a while between queries, and wait a little longer between them if they're failing. If you want five retries, you might wait 1 minute, then 3, 5, 7 and then finally give up.

If you need to send data via email, a queue can simplify your error handling. Instead of sending emails directly to the server, queue them to a directory on disk instead. A second process reads the files that are saved and tries to send them. If the mail is sent successfully, then we delete the mail file or move it to a separate directory, but if it fails we leave it ready for next time. Listing 5.7 shows how we might add that sort of logic to our stock tracking script.



Listing 5.7 - Queuing email to a temporary file

```

import tempfile
...
def queue_email(message):
    if os.access('mail_queue', os.F_OK) != 1:
        os.mkdir('mail_queue')                                #1
    handle, file_name = tempfile.mkstemp(
        prefix='mail',
        dir='mail_queue',
        text=True)                                         #2
    mail_file = open(file_name, 'w')                      #2
    mail_file.write(message['From'] + '\n')                #3
    mail_file.write(message['To'] + '\n')                  #3
    mail_file.write(message.as_string() + '\n')            #3
#1 - Create the mail queue directory
#2 - Make a temporary file
#3 - Write mail info to mail file

```

First we check that the mail queue directory exists. If it doesn't then we need to create it **(1)**.

Next we use the `tempfile` module to create our mail file **(2)**. By doing it this way instead of figuring out the filename ourselves, we're much less likely to run into a conflict with naming if we're running multiple scripts at once.

Now that we have our file, we can write all of the information that we need when it's time to send our email - To, From and the body of the email itself **(3)**.

Once we have the email queued to disk, we can use a second process to read it and send it out. Listing 5.8 shows how the second process might be written.

Listing 5.8 - Sending email from a mail queue directory

```

import os
import smtplib
import sys

mailserver = smtplib.SMTP('mail.yourisp.com')
mail_files = os.listdir('mail_queue')                         #1

for eachfile in mail_files:
    file_name = 'mail_queue' + os.path.sep + eachfile         #1
    file = open(file_name, 'r')                               #1
    me = file.readline()                                     #1
    them = file.readline()                                    #1
    mail_body = ''.join(file.readlines())                     #1
    file.close()                                            #1

    mailserver.sendmail(me, them, mail_body)                 #2
    os.remove(file_name)                                     #2

mailserver.quit()
#1 - Read in mail files
#2 - Try and send the mail, then delete it

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

This part of the process is the opposite of the one that we've just looked at. Given a directory, we want to read in all of the files in it (1), and for each of them read out the To and From lines, then the mail body.

The `smtplib` server will generate an error for anything that means that the mail can't be sent, so we simply try and send the email (2). If it succeeds then we know that the mail has been sent and we can delete the mail file and continue.

Now you don't have to worry about mail being lost because the mail server is down for maintenance, or you can't reach it via the network. All mail will be queued in the `mail_queue` directory, and will only be deleted once it's been sent.

There are still a few limitations though. The main one being that the first error that our program runs into will abort the whole mail sending process. For our purposes it works well enough, since if one email fails the others are likely to fail as well. However, we'd really like our program to be as robust as we can make it. A malformed email address, for example, can cause the SMTP server to reject your connection request, and then that email will be repeated over and over again, blocking all of the others queued up behind it.

NOTE Gracefully handling errors is even more important if your program takes a while to return, or is a batch process that runs overnight. If you have to wait 6 hours to find out if it ran properly, it can take a week or more to shake out all the errors. Detailed error reports help, but you can also work on smaller data sets until you're confident that your program works.

One error shouldn't bring our whole program grinding to a halt, so what we really need are exceptions, a feature in Python designed to help you react to errors like this as they occur and recover gracefully.

5.8 Exceptions

Whenever Python runs into a problem that it can't handle, it triggers an error called an exception. There are a wide variety of exceptions, and you can even define your own to suit particular types of errors if a built-in one doesn't fit your exact error. Exceptions aren't final though - you can write code to catch them, interpret their results and take whatever action you need.

5.8.1 Why use exceptions?

Exceptions, when used well, make your program much easier to understand. You don't need as much error-checking and handling, particularly for error codes and return results, since

I JUST SET UP A PAGE WITH THE HTML FROM THE INTERNAL REPORTS, AND ADD A BIT TO THE TOTAL EVERY SO OFTEN. BY THE TIME THE CHIEF GETS BACK, WE WON'T NEED IT ANY MORE...

BUT... THAT'S CHEATING!

IT WORKED FOR GMAIL, DIDN'T IT?

GAH!



you can generally assume that anything which goes really wrong will raise an exception. Not having error handling code means that the part of your program which does things stands out a lot more, and is easier to understand since it's not being interrupted by checking return codes.

5.8.2 *What does it all mean when your program goes “bang!”*

Before we get into how to use exceptions, let's take a look at a few examples and see how they work. When you get an error in your program, you'll see what's called a **traceback**. This will give you the whole function 'stack', from the part of your program which triggered the error, through intermediate functions to the one which originally started in the core of your program.

Let's start with a traceback from the mail sender which we wrote in the previous section. It shows the most recent error last, so we'll be working backwards.

Listing 5.9 - A traceback when sending mail

```
anthony@anthony:~/Desktop/stocktracker$ python mailsender.py
Traceback (most recent call last):
  File "mailsender.py", line 5, in <module>
    mailserver = smtplib.SMTP('mail.yourisp.com')                      #1
  File "/usr/lib/python2.5/smtplib.py", line 244, in __init__           #2
    (code, msg) = self.connect(host, port)
  File "/usr/lib/python2.5/smtplib.py", line 296, in connect            #3
    for res in socket.getaddrinfo(host, port, 0, socket.SOCK_STREAM):
socket.gaierror: (-2, 'Name or service not known')                      #4
anthony@anthony:~/Desktop/stocktracker$                                         #5
#1 - Our error
#2 - Traceback
#3 - Where's that line?
#4 - The line which triggered the exception
#5 - The exception
```

We start at the end of the listing with the name of the exception and a short description of the error that occurred (5). If you can't figure out what's going wrong with your program, doing a web search for the exception and description can often give you a clue.

The last line that was executed is the one where the exception was raised (4). Note though, that the bug may actually be earlier on in the program, if a variable wasn't set properly or a function is being called with the wrong parameters.

(3) is the file, line number and function name where the exception in (4) was raised. Notice that this is within Python's standard library, so we probably haven't found our problem yet. If in doubt, always assume that there's a bug in your program, rather than Python's standard library.

The traceback (2) will continue to the function which called our original one. Notice how the function name that we're calling, `self.connect`, is the same as the function listed in #3, the last part of the traceback.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Now that we get into our code, the error is obvious. We've forgotten to take out the exampleisp mail server and substitute it with our own (1). There's no web address at 'mail.yourisp.com', hence the original error "Name or service not known." Often the root cause of an exception might be several layers deep in the traceback, and it's a case of tracing the function calls back until you find the source of the error.

Python handles exceptions by propagating them 'up the stack', that is, it'll first look for an exception handler in the current function. If it can't find one, it'll look in the function which called the current one, then that function's parent, and so on, all the way back to the top of your program. If there's still no handler, Python will halt, and print out the unhandled exception and the stack trace. Bang!

Figure 5.5 shows how the stack trace from Listing 5.9 is generated, with the function calls on the left hand side and the traceback rewinding on the right hand side.

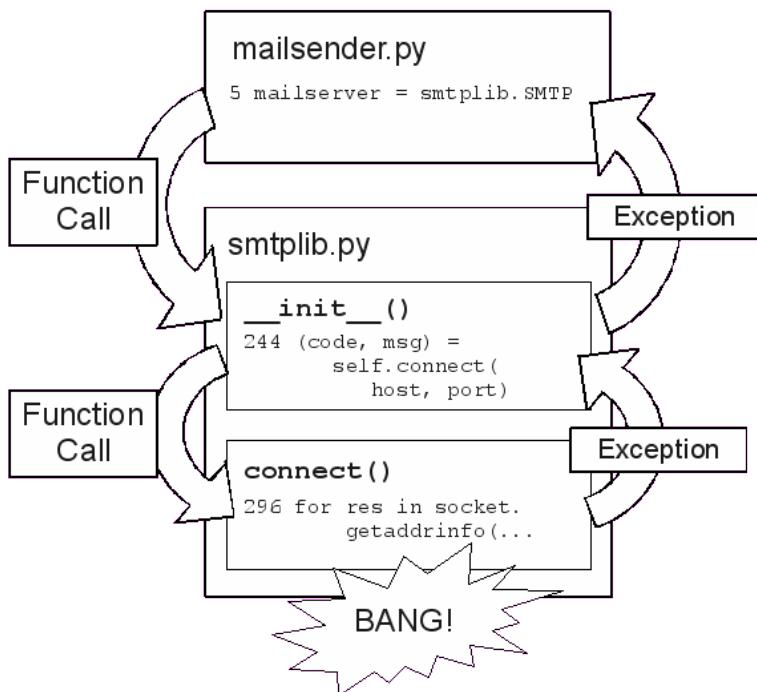


figure 5.5 - A diagram of a stack trace

For a different point of reference, here's another traceback, from `parse_stock_html`.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Listing 5.10 - An exception when parsing a HTML page

```

Traceback (most recent call last):
  File "./stocktracker.py", line 172, in <module>
    stock_values = parse_stock_html(html, ticker_name)
      File "./stocktracker.py", line 61, in parse_stock_html
        result['ah_change'] = (quote.find(attrs={'id': 'yfs_z08_1'+tick}).contents[1])
                                         ^~~~~~
IndexError: list index out of range
#1 - The exception
#2 - The line which triggered it

```

First, **(1)** is the exception which was triggered. It's an `IndexError`, which means that we've tried to access an index of an array that doesn't exist. Something like `['foo'][1]` will trigger a similar exception, since there's no second element of that array.

If we look at the actual line, we can see what might be causing the problem. We're running a `find` in Beautiful Soup, and trying to access the second element **(2)**. There's obviously some data for which the HTML only has one element, and our parsing code doesn't handle it.

The error in this case is due to the up or down arrow images not being shown when the stock price hasn't moved, which is changing the number of elements returned from `.contents()`.



Figure 5.6 - The HTML changes if there's no movement in the stock

It's relatively easily fixed by using a negative index, like this: `.contents[-1]`. Now Python will access either the second element, if there is one, or else the first one. We can be pretty sure that there won't be more than two elements in that particular span.

This should give you some pointers in using tracebacks to help you troubleshoot errors. The main thing to bear in mind is to work carefully backwards looking for potential errors or odd results. If that doesn't help, you might need more traditional checks such as `print` or

assert statements. Of course, once you've tracked down the error, you should add it to your testing suite if possible.

5.8.3 Catching errors

The main reason for having exceptions is to catch them if they are raised and deal with them appropriately. You do this by using a `try..except` block, commonly known as an exception handler. The code within the `try` part is run, and if an exception is raised, Python looks at the `except` part to see how to deal with the error. Here's an example of how we could include one in our mail sending program.

Listing 5.11 - A mail sender with exception handling

```
try:                                            #1
    mailserver.sendmail(me, them, mailouttext)
except smtplib.SMTPAuthenticationError, error:   #2
    print "Bad username or password:", error
except (smtplib.SMTPConnectError,             #3
        smtplib.SMTPHeloError), error:           #3
    print "The server is not responding!", error
except Exception, error:                      #4
    print "An unexpected exception: ", error
else:                                         #5
    os.remove('maildir'+os.path.sep+eachfile)

#1 - Normal program flow
#2 - A Normal Exception handler
#3 - Multiple exception handlers
#4 - A generic exception handler
#5 - If we ran successfully
```

We start with the part of our program that we'd like to wrap **(1)**, to catch any exceptions. It behaves much like any normal indented block, so you can include if statements, loops, function calls and whatever else you need to.

(2) is how we handle a single exception, which is how most exception handling is done. The handler works a little bit like a function in that you give it the type of exception that it needs to handle, and a variable to store the error message. When an exception is raised, we can print an error message and the error message that we've received.

You can also handle multiple types of exception with one handler, by putting the exceptions that you're expecting into a tuple, instead of having one by itself **(3)**.

If you need to catch every exception that's raised, just use `Exception`, which is a generic exception object **(4)**. It's generally considered poor form to use a generic handler like this, since you may end up masking errors that you'd rather have propagate up. If possible you should handle specific errors, but there are occasions where you don't know what exceptions you'll need to handle.

We can also use `else:` to include a section of code to execute if the `try..except` block ran successfully and didn't raise any exception **(5)**. In this case, we're deleting the mail that

we've just sent. There's also a `finally:` option which you can use if you have something which needs to always be run, regardless of whether you had any exceptions.

Exception handling like this should handle most of your needs and allow you to write programs which can recover from error conditions, or at least fail gracefully, and you can use it wherever you need to handle errors. Where that is depends on the nature of the program and the errors that you're trying to catch. A program which handles user input might have one high level exception handler, which wraps your entire program in a `try..except` clause. That way, no matter what the user types, you can handle it and return a reasonable error message. You can also use error handlers around sub-sections of your program, or around specific modules which throw exceptions.

Handling exceptions like this doesn't, however, give you a lot of information about what went wrong. Particularly for critical production systems, it's extremely helpful to know where the error occurred and in what file, pretty much the same as you would for a traceback if you were running the program without exception handling. Fortunately, there's a Python module which can help us to print out our own debugging messages and find out what went wrong - the `traceback` module.

5.8.4 The `traceback` module

The `traceback` module gives you a number of functions for handling exceptions as well as formatting and extracting tracebacks and error messages. The two key ones are `print_exc` and `format_exc`, which print out a traceback and return a traceback as a string, respectively. You can extract this information from the `sys` module, via `sys.exc_type`, `sys.exc_value` and `sys.exc_traceback`, but it's much more straightforward to use `traceback`. Let's extend the error handling in the last section to print out a nice traceback if we get an unknown error.

Listing 5.12 - Using the `traceback` module

```
import traceback
...
try:
    mailserver.sendmail(me, them, mailouttext)
    ...
except:
    traceback.print_exc()                      #1
    traceback_string = traceback.format_exc()   #2
    print traceback_string                     #3
else:
    os.remove('maildir'+os.path.sep+eachfile)
#1 - A shortcut
#2 - Print the exception
#3 - Extract a formatted exception
```

Let's start with a shortcut: You don't have to specify `Exception` for a generic handler. If you omit any exception types at all, it behaves in exactly the same way (**1**).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

The `print_exc` function will print a formatted traceback **(2)**. This is useful if you're running the program interactively, or if you use something like cron, which will email you with the output of any programs that you run.

If you need to log to a file, then you can use the `format_exc` function to return a string with the traceback **(3)**. Other than that difference, it's exactly the same output.

Now you have everything you need to be able to handle any errors that crop up in your program. You can extend the code in this section to handle most situations that you'll run into in practice, and if not then at least leave enough data behind for you to be able to figure out what's gone wrong.

5.9 Where to from here?

The scripts in this chapter are self-contained, so there aren't any specific suggestions as to how you can extend them. Instead, try applying the lessons (and code) from this chapter to automate something that you do frequently, either at work or home. Good candidates are anything which you find dull and boring, or which requires detailed steps and is difficult to do properly.

If you can't automate all of your process, you can at least cover part - for example download required data so that it's all in one place, or send out several emails from a central data store.

Often a good script can take away several hours of work over the course of a month, so with all of the new free time that you have, you can write another script. Eventually you might not have to do any work at all!

5.10 Summary

The first half of this chapter covered some of the basic but important libraries in Python that you can use to connect to the outside world and get 'real work' done. We covered a number of technologies:

- downloading HTML from the web
- parsing HTML using BeautifulSoup
- writing data out to a CSV file
- composing email, including writing HTML email and attaching documents
- sending email via SMTP

In the second half we stepped back and looked at how we can make our programs more reliable - after all, this is the real world, and other people might have lots of money riding on them.

We first looked at how we can find areas of our problem which are at risk, and also thought about whether we should fix them - depending on the costs of doing so and the fallout from any potential failures. Then we covered some simple strategies to make our programs more reliable, as well as reducing the damage done if they did fail.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Finally we covered exceptions and tracebacks, which are Python's way of handling errors when they occur, and how we can catch exceptions, examine them and deal with the problem if we're able to.

In the next chapter, we'll take a break and write our own adventure game, with monsters, treasure, danger and excitement!

6

Classes and Object Oriented Programming

Up until now we've been skimming over one of the fundamental ways that you can organize your program in Python - the class. Classes and Object Oriented Programming are normally seen as big, scary things that real programmers use to write their programs, and you might think that you need a lot of theoretical knowledge in order to use them properly. Nothing could be further from the truth, and in Python it's possible to ease into them gradually.

In this chapter we'll start with the code that we wrote back in chapter 2 to generate our caves for Hunt the Wumpus, and show you how much easier it is to write it using classes. Then we'll build up from there into a fully fledged adventure game along the lines of Adventure or Zork. While we do that, you'll find out all about Python's classes and how to make best use of them.

6.1 What exactly are classes?

If you think all the way back to chapter 2, you might remember that we had a group of functions that dealt with caves for our player and wumpus to live in. There was a function for creating a cave, another for linking two caves together to create a tunnel, one to make sure all of the caves were linked, and so on. When we were writing our program, we dealt with the caves entirely through our functions. One way to create a class is to identify a group of functions like this and make their relationship official.

6.1.1 Containing your data

Another way to think about classes is as a container or a wrapper around data that you want to use in your program. You can group all of the data needed to perform a particular task, and provide functions to deal with it, particularly if the data is complicated, difficult to work

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Licensed to Robert Humphrey <rphumphrey@gmail.com>

with, or needs to be consistent - for example, the program that keeps track of the balance of your account at the bank.

6.1.2 They're a type of their own

Classes are similar to something called an Abstract Data Type, which is a set of data and all of the operations that can be performed on that data. You don't have to specify all of the possible things that you can do with the data inside your class, just what's useful for your particular situation. When designing your class though, it often helps to think of all of the possible things that you might want to use it for, and add those that make sense.

6.1.3 So how do they work?

I think of classes as a big rubber stamp. Once you've created your rubber stamp, you can easily stamp out as many pictures as you like. Classes work the same way - you generally don't work directly with a class, you instead work with *instances* of that class, which you create by asking the class to do it for you.

The main difference when using classes in a program is that if you need a slightly different picture, it's easy to create a copy of the original stamp, change it a bit and work with that.

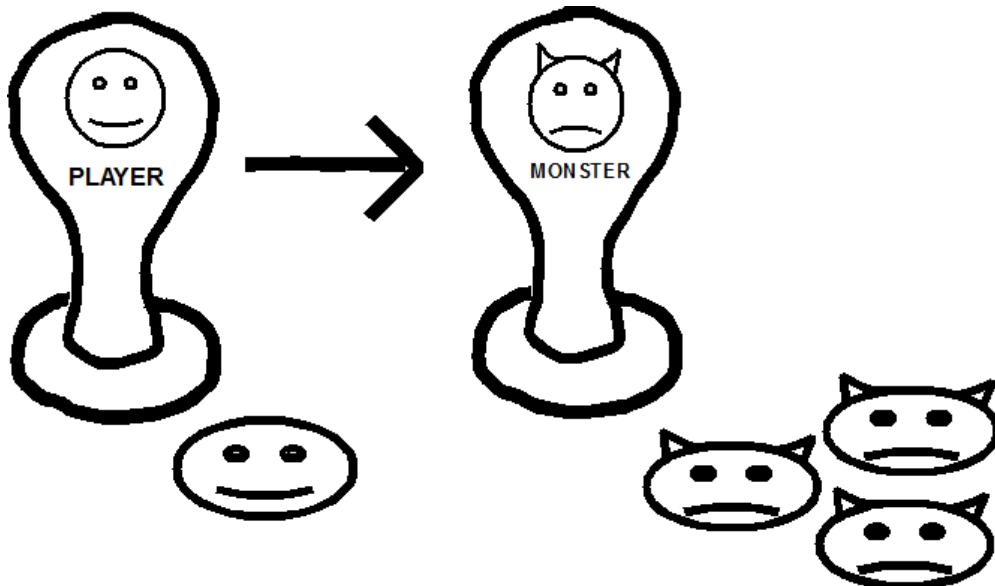


Figure 6.1 Monsters are like the player, except with horns and a frowny face

Both the instance and the class itself can have methods for you to call and data to access. Mostly these will be set when you first create an instance, but Python also allows you to update them on the fly if you need to, even to the point of rebinding methods.

Terminology Object Oriented programming has a great deal of terminology, most apparently designed to confuse the unwary reader. You'll hear people refer to classes, objects, instances, methods, class methods, getters, setters, and so on. If you're unsure of what someone means, just try to figure out whether they're talking about the rubber stamp, or the mark that it's making on your program.

6.1.4 Your first class

The class in listing 6.1 should look very familiar, although there are some parts which are quite different. It contains the cave list and methods that we wrote in chapter 2, but updated so that they're contained within a class.

Listing 6.1 - A object to store caves

```
from random import choice

class Caves(object):
    def __init__(self, number_of_caves): #1
        self.number_of_caves = number_of_caves #2
        self.cave_list = range(number_of_caves) #2
        self.unvisited = range(number_of_caves)[1:] #2
        self.visited = [0] #2
        self.caves = [] #2
        self.setup_caves(number_of_caves) #2
        self.link_caves() #2

    def setup_caves(self, cave_numbers): #3
        """ Create the starting list of caves """
        for cave in range(cave_numbers):
            self.caves.append([])

    def link_caves(self): #3
        """ Make sure all of the caves are connected
        with two-way tunnels """
        while self.unvisited != []:
            this_cave = self.choose_cave(self.visited)
            next_cave = self.choose_cave(self.unvisited)
            self.create_tunnel(this_cave, next_cave)
            self.visit_cave(next_cave)

    def create_tunnel(self, cave_from, cave_to): #3
        """ Create a tunnel between cave_from
        and cave_to """
        self.caves[cave_from].append(cave_to)
        self.caves[cave_to].append(cave_from)

    def visit_cave(self, cave_number): #3
        """ Mark a cave as visited """

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        self.visited.append(cave_number)
        self.unvisited.remove(cave_number)

    def choose_cave(self, cave_list):                                #3
        """ Pick a cave from a list, provided
            that the cave has less than 3 tunnels."""
        cave_number = choice(cave_list)
        while len(self.caves[cave_number]) >= 3:
            cave_number = choice(cave_list)
        return cave_number

    def print_caves(self):                                         #3
        """ Print out the current cave structure """
        for number in self.cave_list:
            print number, ":", self.caves[number]

if __name__ == '__main__':                                         #4
    caves = Caves(20)                                              #4
    caves.print_caves()                                             #4

#1 - Creating a class
#2 - Setting up variables within the class
#3 - Turning functions into methods
#4 - Testing by creating an instance

```

We start with the syntax that Python uses to create a class (**1**). It's similar to the creation of a function, except that by convention a class name starts with a capital (classes are important, after all). The object in brackets is the class that this one inherits from - in this case Python's generic object class, since we're not inheriting anything.

Most Python classes will have the `__init__` method, which is responsible for setting up instances of the class when it's first created (**2**). Notice that we've gained a `self` argument in the method? That's so that a method can access variables and share state. All of the lists that we used in chapter 2 are here, but prefixed with `self` so that they refer to the variables in the instance.

The functions that we used to set up our caves are here, and they've received the `self` treatment too (**3**). Other than that there aren't too many changes to them, which is what we're expecting since they're just functions with an explicit `self`.

Once we've set up our class, we create an instance of it and call its `print_caves` method to test it out (**4**). Python runs the `__init__` method of the class, which in turn calls `setup_caves` and `link_caves` and creates our cave network, which you can see from the results of `caves.print_caves`.

So, what have we gained from putting all of our functions inside a class? The main benefit is that all of the details of the caves are contained within the instance that we've created. We could now create extra cave systems at



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

the same time and not have to worry about them conflicting with each other. From here you can also extend the class, perhaps including the atmospheric cave names and other functions that I've excluded, or adding a method to extend the cave system.

NOTE Classes are another divide-and-conquer mechanism that you can use in your programs. Once you've created an instance, you don't need to think about why it works - just about what you can use it for in your code.

There's just one problem with our new Caves class though. While we've created our class and it works well, it's still not a very Object Oriented design. We've really just taken our existing functional design and pushed it into a class. If in the future we wanted to add extra functionality, such as having treasure to pick up within our caves, more monsters or other features, they'd be very hard to add. Much as adding functions changed the design of our program back in chapter 2, using classes properly will change the emphasis of our design now.

6.2 *Object Oriented Design*

One of the reasons that many people prefer Object Oriented programs is that objects tend to map very well onto the things that we deal with in the real world, and make it easier for us to think about how they interact when we're developing our program. If you were writing a program to manage your finances, you might create classes called Account, Expense, Income or Transaction. If you were writing a program to control a factory, you could have classes called Component, ConveyorBelt, Assembly (as in, multiple components joined together) or AssemblyLine.

Let's take a step back and think about our adventure game a bit more. What sorts of things will it have? Well, if we take a traditional approach, the player will be an intrepid adventurer, searching for treasure, fame and glory in an underground dungeon or cave system filled to the ceiling with monsters.

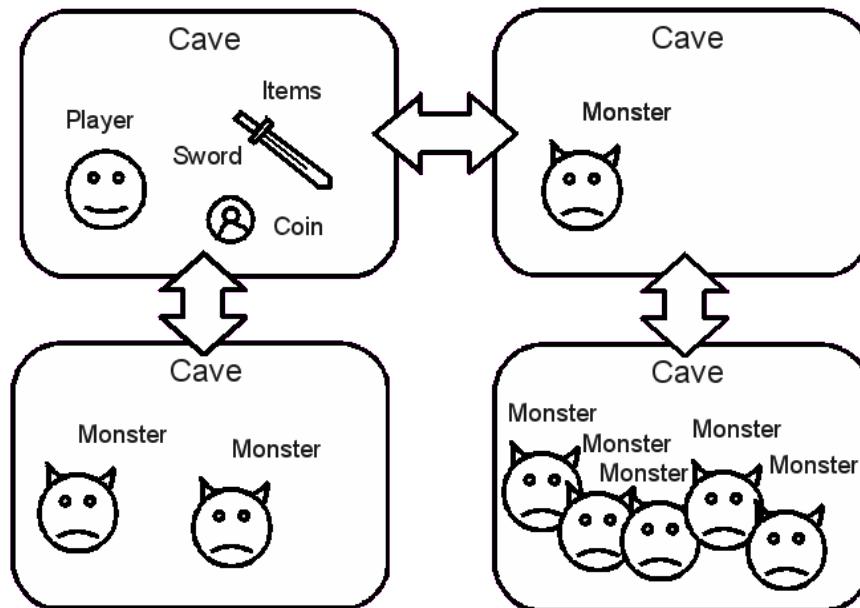


Figure 6.2 A back-of-the-envelope sketch of our game

So, the fundamental thing that we should be dealing with is really the cave, rather than the cave system as a whole. We've been tricked by the mechanics of our lists and functions into thinking that the cave system was the important part, but we can get a much cleaner design by thinking at the right level. That is, of individual caves and what's inside them.

Listing 6.2 A more object oriented design

```
from random import choice, shuffle

class Cave(object):
    def __init__(self, name, description):
        self.name = name
        self.description = description
        self.here = []
        self.tunnels = []

    def tunnel_to(self, cave):
        """Create a two-way tunnel"""
        self.tunnels.append(cave)
        cave.tunnels.append(self)

    def __repr__(self):
        return "<Cave " + self.name + ">"
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

cave_names = [
    "Arched cavern",
    ...
]

def create_caves():                                         #4
    shuffle(cave_names)
    caves = [Cave(cave_names[0])]
    for name in cave_names[1:]:
        new_cave = Cave(name, name)
        eligible_caves = [cave for cave in caves           #4
                            if len(cave.tunnels) < 3]           #4
        new_cave.tunnel_to(choice(eligible_caves))
        caves.append(new_cave)
    return caves

if __name__ == '__main__':
    for cave in create_caves():
        print cave.name, "=>", cave.tunnels               #5
    #1 - Our new Cave object
    #2 - Each cave 'knows' what it's connected to
    #3 - Add a __repr__ method
    #4 - A function to set our caves up
    #5 - Print the list of caves as a test

```

We start with the setup for our new Cave object **(1)**. Rather than set up a list for cave names, another for linking, and a third to tell whether a cave's been visited or not, we store it all within the object itself. When we build lists of caves later on, we can easily filter by these attributes. I've added a `self.here` list to store any other objects (such as the player, monsters or treasure) that might be in the cave, and also a `description` string, which will describe the cave when the player enters it. We'll ignore these two new values for now.

Since you can easily tell what a cave is linked to **(2)**, adding a tunnel to another cave is easy. Just add them into our list of tunnels and add ourself (`self`, the current Cave instance) into their list of tunnels. Notice too, how we're dealing with the caves at the instance level, which makes our program a lot clearer.

One last thing that we'll do is add a `__repr__` method to our class **(3)**. The one which is built in to the base `object` is a little unreadable (it'll be something like `<__main__.Cave object at 0x00B38EF0>`), and this makes our program's output look much nicer when we have to print out a cave.

Now all we need to do is to figure out how to link the caves up. Borrowing our list of cave names from chapter 2, we can assign each one to a new cave instance, link that instance to an existing cave and then add it into our caves list **(4)**. The only even slightly tricky bit is how we find caves to link to, but we can easily figure that out by checking the length of each cave's tunnel list inside a list comprehension. Also notice that Python doesn't constrain

WELL, WHAT'S WORLD OF
WARCRAFT THEN, EXCEPT AN
ADVENTURE GAME WITH
PRETTY PICTURES?

I'LL GIVE IT A GO.
I USED TO PLAY
ADVENTURE BACK
BEFORE A.J. WAS BORN



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

how you solve a problem - you're free to use functions, classes and bare code wherever you need to.

If you're not convinced that it works, (5) prints out the complete cave list.

The main thing to notice is how much of Listing 6.1 is replaced. The original version of Caves had 6 different methods calling each other, and we've replaced that with one class and an external function. Like we learned in Chapter 2 - shorter, simpler code is often a sign that you're on the right track, and an Object Oriented design maps onto our adventure game very well. Let's go on now and tackle the next part of our program - handling input from the player.

6.3 Player input

Most adventure games are played by typing instructions at a prompt, things like "GO NORTH", "GET SWORD", "KILL MONSTER" and "GET TREASURE". The game then responds with the results of your action, as well as a description of the room you're in and things which are in the room. We'll take the same approach, and use some of the properties of objects to make our program easy to extend. Bear in mind too that we'll want to make our code easy to test, so we'll break out the user input into a separate function.

6.3.1 First steps - verbing nouns

We'll start by trying to find a good way to write the "verb noun" interface into our class structure. Normally an object will be a noun, and the methods on that object will be verbs, so a command like "get sword" should try and find the sword object in the current room and call its 'get' interface. Designing this way means that rather than having one massive Player class which knows how to do everything possible in the game, you can instead have more, smaller classes, which are easier to understand (and change and extend).

Listing 6.3 has the code for the core of our application - the player object. It's responsible for reading input from the player, as well as finding the right object to call to interpret the command.



Listing 6.3 - A Player object

```
import shlex

class Player(object):
    def __init__(self, location):
        self.location = location
        self.location.here.append(self)
        self.playing = True
        #1
        #1
        #1
        #1

    def get_input(self):
        return raw_input(">")
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

def process_input(self, input):                      #2
    parts = shlex.split(input)                      #2
    if len(parts) == 0:                            #2
        return []                                 #2
    if len(parts) == 1:                            #2
        parts.append("")                          #2
    verb = parts[0]                             #2
    noun = " ".join(parts[1:])                   #2

    handler = self.find_handler(verb, noun)         #3
    if handler is None:                           #3
        return [input +                         #3
                  "? I don't know how to do that!"] #3
    return handler(self, noun)                    #3

def find_handler(self, verb, noun):                #4
    if noun != "":                                #4
        object = [x for x in self.location.here   #4
                   if x is not self and           #4
                      x.name == noun and       #4
                      verb in x.actions]      #4
    if len(object) > 0:                           #4
        return getattr(object[0], verb)            #4

    if verb.lower() in self.actions:               #5
        return getattr(self, verb)                 #5
    elif verb.lower() in self.location.actions:    #5
        return getattr(self.location, verb)        #5

def look(self, player, noun):                     #6
    return [self.location.name,                  #6
            self.location.description]

def quit(self, player, noun):                    #6
    self.playing = False                         #6
    return ["bye bye!"]

actions = ['look', 'quit']                       #6

#1 Player variables
#2 Handling input from the player
#3 What do we do with our command?
#4 Finding a handler on objects
#5 Finding a handler in the location or ourselves
#6 Some simple commands

```

The variables that we'll initially need within our `Player` class are pretty straightforward (**1**). We just add them to a location, and tell the game that they're playing.

We split up our command here, and make sure that there's always something in the verb and noun variables (**2**). We're using `shlex.split()` to split our command because it handles quotes much better than the normal `split`. If, for example, the player types `GET 'GOLD KEY'`, then `shlex.split()` will read `GOLD KEY` as one part. We join up anything after the verb and assume that it's part of the noun, so `GET GOLD KEY` will work just as well.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Once we have our command in an easy to process format, we try and find a method to call **(3)**. If the method finder returns None, it means that there wasn't a method to handle the command, and we return an error (If you've ever played an adventure game, it'll be all too familiar).

Once we have our command, we try and find a method to handle it **(4)**. If we have a noun, we look for an object which matches it, for example the SWORD in GET SWORD, and see if it can respond. If not, then we pass the command through to either the location or the player (this will help us provide a better interface in the next few sections).

If the player hasn't given us a noun, then it might be a more generic command, like LOOK or QUIT, so we look for it in our current location and our Player object **(5)**. If neither of those work then we haven't found one and we 'fall off' the end of the method, which means that we return None, which results in an error.

I've added two basic commands **(6)** to the Player, LOOK and QUIT, so you can get a feel for how they'll work in the finished game.

Now that you have a player, you just need to be able to read input from them and use that input to run the game. A sample framework is listed below. We make a simple cave, put the player into it and then loop, reading input until the player is finished playing. Later on, we'll probably incorporate this into a Game object, but since Python is flexible we can leave it as a function while we write and test our other classes.

Listing 6.4 - Running our player class

```
def test():
    import cave
    empty_cave = cave.Cave(
        "Empty Cave",
        "A desolate, empty cave, "
        "waiting for someone to fill it.")
    player = Player(empty_cave)

    print player.location.name          #1
    print player.location.description  #1
    while player.playing:             #1
        input = player.get_input()     #2
        result = player.process_input(input) #2
        print "\n".join(result)         #2

    if __name__ == '__main__':
        test()
#1 - Set up a test environment
#2 - Our main loop
```

Our player class needs a location to work properly, so we set up a test cave here and put the player into it **(1)**. A simple description and name is all we need to get going.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Once we've done that, we get input from the player and pass it to the `process_input` method, which will run our code and return the results as a list of strings (2). We then print them one to a line using the `join` method of the "\n" string. Once the player has issued the quit command, the `player.playing` variable will be false, and we stop the program.

If you run your adventure program now, you should be able to give it commands such as LOOK and QUIT. It's simple stuff, but we'll see how to extend our interface in the next section.

6.4 Treasure!

Let's start adding some more exciting things to our game. First of all, we'd like to be able to give the player some equipment or treasure early on, to draw them into the game and get them involved. It's not really an adventure without treasure or a sword, so let's add those first. Before we do though, we'll need to think a bit more about our design.

6.4.1 Where should our methods go?

We'll obviously need to interact with our items, which means that we'll at least want to be able to do things like GET SWORD, LOOK SWORD and DROP SWORD. With our current way of doing things that means that there'll need to be a method somewhere to handle the GET, LOOK and DROP.

One option is to store it in the `Player` class - after all, it's the player doing the getting and dropping. It's tempting to think along these lines, but when doing any Object Oriented programming, you'll want to delegate as much responsibility as possible. For example, later on you'll probably want to add objects which the player can't pick up, like a heavy chest or a statue. That's fine, just add a check to see if the object has an immovable flag set. What if they can pick the chest up if they have the gilded girdle of strength? Hmm, another check. You can see where this is going - by the time you finish your game, you might have five or six (or twenty) conditions in the `get` method on your player.

NOTE Class design can be a tricky thing when you're first starting out. The main thing to remember is that experience counts, so you'll get better with practice. Also, don't forget that you can experiment with different designs, and pick the best one.

A better way is to make the objects themselves responsible for judging whether they can be picked up. A chest 'knows' that it's heavy, and can check to make sure that the player has the right items in their inventory before allowing itself to be picked up. It sounds odd, but really the `Player` object shouldn't be responsible for how heavy objects are or how monsters fight, and adding things like that to the `Player` object will make it too complicated. Let's have a look at how we would program some objects which can be looked at, then we'll modify them so that they can be picked up.

Listing 6.5 - An object which can be looked at

```
class Item(object):
    def __init__(self, name, description, location):      #1
        self.name = name
        self.description = description
        self.location = location
        location.here.append(self)

    actions = ['look']                                     #2

    def look(self, player, noun):                         #2
        return [self.description]                         #2
```

#1 - initializing the Item

#2 - Let the item be looked at

The things which the `Item` needs to know are pretty much the same as for the `Player` and `Cave` objects - what its name is and where it is (1). We can feed all of those in when we create an `Item` instance.

Initially our item responds to one command: `LOOK` (2). When the player issues a `LOOK ITEM` command with the item's name as a noun, this is the method which will be called. All it does is return the item's description.

AH, WELL YOU NEED TO
GET THE HAMMER FIRST.
THAT'S A PRETTY GOOD
WEAPON.
ONCE YOU HAVE THAT, YOU
SHOULD BE OK, BUT LET
ME KNOW IF YOU FIGURE
OUT HOW TO BEAT THE
TROLL.



6.4.2 Finding the treasure

As well as this, we'll want to modify the description of the cave so that the player knows what items are in a particular location. While we're at it we'll follow the lead set so far and move the `look` method out of the `Player` class. Just delete the method from `Player`, and add this one into the `Cave` class.

Listing 6.6 - Modifying the look command

```
def look(self, player, noun):
    if noun == "":
        result = [self.name,
                  self.description]
    if len(self.here) > 0:                      #1
        result += ["Items here:"]
        result += [x.name for x in self.here]      #1
                                                #1
                                                #1
    else:
        result = [noun + "? I can't see that."]   #2
    return result

    actions = ['look']                           #3

#1 - List items
#2 - Error handling
#3 - Update actions
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

The main change to this method is to list all of the items in the location and place them under the description **(1)**. Without this, the player won't know that there's a sword in the cave, unless they happen to guess that there might be.

This function will also get called if the player tries to refer to something that doesn't exist (such as LOOK AARDVARK). If there's no aardvark in the room, we need to return an error **(2)**.

Don't forget to update the actions that the Cave object can handle **(3)**, and remove 'look' from the Player class, otherwise it'll continue to try and handle the LOOK command.

A TROLL TOO? GREG'S INDULGING ALL OF HIS GRUBBY FANTASY CLICHES...

OH YEAH - DON'T FORGET TO GET THE SUPERFLUOUS APOSTROPHE OF ANTIARC FROM THE ANCIENT WIZARD.



Listing 6.7 - Updating your set up

```
...
"A desolate, empty cave,
"waiting for someone to fill it.")

import item
sword = item.Item("sword",
    "A pointy sword.", empty_cave) #1
coin = item.Item("coin", "A shiny gold coin. " #1
    "Your first piece of treasure!", empty_cave) #1

player = Player(empty_cave)
...
#1 - Add items
```

Add items into your adventure! Just set their name, description and location, and the Item object will take care of the rest.

If you run your adventure now, you should be able to look at your treasure, and a shiny sword, but you can't reach them. So... tantalizingly... close...

6.4.3 Picking up the treasure

All we need now is for the objects to respond to being picked up. There are two updates that we need to make, the first being to the objects, to give them get and drop methods, and the second being to update the Player class so that it can carry things.

Listing 6.8 - Items which will let themselves be picked up

```
actions = ['look', 'get', 'drop']

def get(self, player, noun):
    if self.location is player: #2
        return ["You already have the " + self.name"] #2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

    self.location.here.remove(self)                      #1
    self.location = player                            #1
    player.inventory.append(self)                      #1
    return ["You get the " + self.name]                #1

def drop(self, player, noun):                         #3
    if self not in player.inventory:                  #3
        return ["You don't have the " + self.name]    #3
    player.inventory.remove(self)                     #3
    player.location.here.append(self)                 #3
    self.location = player.location                  #3
    return ["You drop the " + self.name]              #3

#1 - A get method
#2 - Check whether we've already been added
#3 - A drop method

```

The get method itself is straightforward **(1)**. The object needs to remove itself from the here array in the current location, and put itself into the player's inventory (we'll make that list in a minute) and set its current location. Once we've done that, we return a message to let the player know what we've done.

If you run the code above without this check **(2)** it'll work fine, but if the player tries to pick the item up again your program will crash, since it can't remove an item from a list if it isn't there.

The drop method is pretty much the same as the get method - just a reversal **(3)**. Just like the get method above, we need to check that the object is in the player's inventory before we move it back to the location that we're in.

Now let's update the Player class so that it can hold objects, and add some commands to tell us what we're carrying, and output error messages, as well as looking in our inventory when trying to find handlers for commands.

Listing 6.9 - updates to our Player class

```

class Player(object):
    def __init__(self, location):
        self.location = location
        self.location.here.append(self)
        self.playing = True
        self.inventory = []                                #1

    ...

    actions = ['quit', 'inv', 'get', 'drop']

    def get(self, player, noun):                         #2
        return [noun + "? I can't see that here."]      #2

```

You are in a maze of passages.
/ \

go north
/ \

You are in a dank,
gloomy cave.
You see an orc.
The orc hits you.



```

def drop(self, player, noun):                      #2
    return [noun + "? I don't have that!"]          #2

def inv(self, player, noun):                       #3
    result = ["You have:"]
    if self.inventory:
        result += [x.name for x in self.inventory] #3
    else:
        result += ["nothing!"]                      #3
    return result                                    #3

...

def find_handler(self, verb, noun):                 #4
    # Try and find the object
    if noun != "":
        object = [x for x in
                    self.location.here + self.inventory
                    if x is not self and
                    ...
#1 - The player's inventory
#2 - Add error handlers
#3 - An inventory command
#4 - Check our inventory when finding handlers

```

To start with, the player needs to be able to carry things around with an inventory (**1**). The easiest option is just to add a list to our class. When the player picks up objects, they'll be appended to this list.

These error handlers (**2**) are similar to the error handler that we wrote for our Cave class. If we try and get something that's not in the current location, then these methods will be called to handle the GET and DROP commands.

The player should be able to remind themselves of what they've found so far, so (**3**) is a command that will list everything that they're carrying.

The final change is to check the items in our inventory when looking for handlers (**4**). This way, the player can LOOK at or DROP things in their inventory.

Now you can pick up the sword and shiny coin, as well as just look at them. You can also put them back down again, although that's much less adventurous. Your trusty sword and first piece of treasure in hand, it's time to venture further into the caves.

Nice try Sid, but your
spell is too old.
The orc hits you.
You die.



GREG!!!



6.5 Further into the caves

An adventure isn't an adventure without some sort of exploration. In most games you move around by issuing commands like GO NORTH, or just NORTH or N for short. As you travel, the game will update the descriptions, to tell you about the area that you've just moved through. We'll set up our movement commands in the same way that we set up our other commands, but we'll add some shortcuts too so that typing our movement is easier.

First we'll look at how we add the directions themselves into the Cave class, and then we'll create the commands that let our player move around.

Listing 6.10 - Adding movement to our Cave class

```
class Cave(object):
    directions = {
        'north' : 'south',                      #1
        'east'   : 'west',                       #1
        'south'  : 'north',                      #1
        'west'   : 'east' }                      #1

    def __init__(self, name="Cave", description=""):
        ...
        self.tunnels = {}                      #2
        for direction in self.directions.keys():
            self.tunnels[direction] = None       #2

    def exits(self):
        return [direction for direction, cave
                in self.tunnels.items()           #3
                if cave is not None]            #3

    def look(self, player, noun):
        if noun == "":
            ...
            if len(self.exits()) > 0:          #4
                result += ['Exits:']          #4
                for direction in self.exits(): #4
                    result += [direction + ": " +
                                   self.tunnels[direction].name] #4
            else:                           #4
                result += ['Exits:', 'none.']

    def tunnel_to(self, direction, cave):      #5
        """Create a two-way tunnel"""
        if direction not in self.directions:   #5
            raise ValueError(direction +
                                " is not a valid direction!")
        reverse_direction = self.directions[direction]
        if cave.tunnels[reverse_direction] is not None: #6
            raise ValueError("Cave " + str(cave) +
                                " already has a cave to the " +
                                reverse_direction + "!")
        self.tunnels[direction] = cave           #5
        cave.tunnels[reverse_direction] = self  #5
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

- #1 - A list of directions
- #2 - Adding tunnels to the cave
- #3 - Listing all of the valid exits
- #4 - Adding the exits to the look command
- #5 - Tunneling from one cave to another
- #6 - Exceptions to handle bad behavior

The first part of our changes adds a list of valid directions into the `Cave` class (1), so that we know all of the directions that we can travel. We can also use this to find the opposite direction (we'll need that in a minute).

When we're creating each cave, we'll need to set up the basic data structure that we'll use to store the caves that we can reach in each direction (2). For now they're all `None`, which means that there's no other cave that way.

(3) is a convenience method to list the directions of all of the exits for a particular cave. It's pretty simple - just a list comprehension over `self.tunnels`, but it makes our code much easier to follow when we can just access `cave.exits()`. Of course, don't let the order that the functions are written in fool you - the code for this method was pulled from `look` once that started to look ugly.

The player will want to know which way he can go from cave to cave, so (4) lists all of the valid exits. If there aren't any, then we let them know that, too.

We also need a way to create tunnels between our caves. Linking one way is easy - we just put the cave in `self.tunnels` in the right direction, but we'd like our tunnels to be two way, so we look up the reverse direction in our list and add a link to ourselves from the target cave (5).

If the above method looked a little strange, it's because we added some exceptions to catch error cases when linking tunnels (6). The `raise` command will create similar errors to the ones that you've been seeing so far, say when you've mistyped something. In our case we're effectively creating our own type of object, much like an integer or a string, so it's much better to behave like one and raise an exception rather than print an error or ignore the bad input. We'll be crashing very close to the source of the problem, and giving a very clear error message, which makes it much easier to troubleshoot your programs.

TIP When you're designing classes like `Cave` (which are effectively library classes) it's always a good idea to catch cases like this and raise an exception where possible. That way when you're using the class later on, it's obvious when you've made a mistake.



So now we have a Cave class which can store directions and links to other caves, as well as describe those directions to the player. Let's now add some commands to let the player move between caves.

Listing 6.11 - Commands to move between caves

```
def go(self, player, noun):                      #1
    if noun not in self.directions:               #1
        return [noun + "? "
                  "I don't know that direction!"]  #1
    if self.tunnels[noun] is None:                 #1
        return ["Can't go " + noun + " from here!"] #1
    self.here.remove(player)                      #2
    self.tunnels[noun].here.append(player)         #2
    player.location = self.tunnels[noun]          #2
    return ['You go ' + noun +
            self.tunnels[noun].look(player, '')]     #2

def north(self, player, noun):                   #3
    return self.go(player, 'north')              #3
n = north                                       #3
def east(self, player, noun):                   #3
    return self.go(player, 'east')               #3
e = east                                         #3
def south(self, player, noun):                  #3
    return self.go(player, 'south')              #3
s = south                                         #3
def west(self, player, noun):                   #3
    return self.go(player, 'west')               #3
w = west                                         #3

l = look                                         #4

actions = ['look', 'l', 'go',
           'north', 'east', 'south', 'west',
           'n', 'e', 's', 'w']                         #4
#4
#4
#4

#1 - Check player input
#2 - Move the player
#3 - Add some shortcuts
#4 - Update the actions for the Cave class
```

The basic command that we're adding is called GO (as in, GO NORTH). First we need to check the direction that the player's entered, both to make sure that it's valid, and that there's a cave in that direction (**1**).

Once we're sure that it's valid, we can go ahead and move the player (**2**). The mechanics are straightforward - remove the player from the current cave, add them to the new one and update their location. We also append the new cave's description onto the results of the command, to make life easier on the player and save wear and tear on their keyboard.

We can also make life easier by providing shortcuts for common commands (**3**). Typing GO NORTH over and over again gets tedious, so we allow the player to use 'NORTH' or just

'N', and similarly for each of the other cardinal directions. Behind the scenes they just call the original go method, so there's no difference in how they behave.

The last thing that we need to do is update the list of valid actions for the Cave class (4). We also add a shortcut for the look command while we're at it.

There, we're done. Notice how we've split the functionality between the player and the location that they're in? This is a normal feature of a good object oriented design, where the objects have well separated responsibilities. In our case, the cave object is responsible for keeping track of its exits and where they go, and the player object can use that information from inside its go command. If later on there's something else which might make use of directions, we don't have to extract the code from the player object, or wherever we've hidden it, to make our new functionality work.

The player still needs somewhere to move to though, so let's extend our previous cave generating function to help out.



Listing 6.11 - Creating a cave network

```
Class Cave(object):
    ...
    def can_tunnel_to(self):
        return [v for v in self.tunnels.values()
                if v is None] != []
    #1
    #1
    #1

    cave_names = [
        "Arched cavern",
        ...
        "Spooky Chasm",
    ]

    def create_caves():
        shuffle(cave_names)
        caves = [Cave(cave_names[0])]
        for name in cave_names[1:]:
            new_cave = Cave(name)
            print caves
            eligible_caves = [cave for cave in caves
                               if cave.can_tunnel_to()]
            #3
            #3
            old_cave = choice(eligible_caves)
            #3
            directions = [direction for direction, cave
                          in old_cave.tunnels.items()
                          if cave is None]
            #4
            #4
            direction = choice(directions)
            #4
            old_cave.tunnel_to(direction, new_cave)
            #5
            caves.append(new_cave)
            #5
        return caves
    #2
```

```

player.py:
if __name__ == '__main__':
    import cave
    caves = cave.create_caves()

    cavel = caves[0]
    import item
    sword = item.Item("sword", "A pointy sword.", cavel)
    coin = item.Item("coin", "A shiny gold coin. "
                      "Your first piece of treasure!", cavel)

    player = Player(cavel)
    print '\n'.join(player.location.look(player, ''))
    while player.playing:
        input = player.get_input()
        result = player.process_input(input)
        print "\n".join(result)

#1 - Another convenience method
#2 - We're modifying our existing function
#3 - Pick a cave from our list
#4 - Pick a direction to link it to
#5 - Link in the new cave
#6 - Update the game setup in players.py

```

We start with another convenience method **(1)**. In a minute we'll see how it's used, but it's just to tell you whether a cave can be linked to (or not, if all four directions are occupied).

This function **(2)** is a modification of the previous `create_caves` which we wrote all the way back in listing 6.2. The main difference is that this one picks a direction as well as a cave, but other than that it's our standard connected cave structure.

We pick the next cave to link to by using our `can_tunnel_to()` convenience method in a list comprehension **(3)**.

We also need to pick an empty direction to link our cave against **(4)**. The choice function will fail if we don't have any directions, but we're not too worried about that happening since `can_tunnel_to()` method has already told us that it has at least one.

Once we have our cave with a spare slot, it's easy to link the new cave in **(5)**. We also add our new cave to the list so that other new caves can be linked to it too.

And finally, update the game setup (in `player.py`) to use the new cave system **(6)**. Rather than have just one empty desolate cave, we now put everything into the first cave in our list, including the player. Other than that, it's pretty much the same.

Now when you run `players.py`, you should see some



exits from your starting position, as well as the normal description and items. You can pick up your sword, move around, drop it in another location and come back to it.

Congratulations - you've created a world! Feel free to go and explore it. When you come back we'll add some more parts.

6.6 Here there be monsters!

So we now have a player, items and treasure to collect - all that's left to put in our adventure is sudden, painful death, also known as danger and excitement. We'll add some monsters into our game that will move around the map and which might attack the player if they're in the same cave and feeling nasty, or else pick up any treasure lying around. The player can attack the monsters too, and loot their treasure.

6.6.1 Creating our monsters

Let's just think about that for a second. Don't the monsters sound awfully familiar? Let's draw up a chart to help:

Monster

- Moves around the map
- Collects treasure
- Attacks the player

Player

- Moves around the map
- Collects treasure
- Attacks monsters

The monsters and the player seem to share an awful lot in common. In our function based program we'd look at this and recognize that we need to avoid duplication, but how do we do that with an object oriented program? The answer is to subclass Player.

Subclassing is really just a fancy way of saying "make a slightly different copy of my class that does this and this differently". In the case of the monsters in our game, they'll behave much the same way, but instead of the player telling them what to do next, they'll figure it out for themselves. They'll also need to have a name and description so that the player can look at them. That means that their `__init__` and `get_input` functions need to be different, but we'll keep most of the rest of the Player class intact.



Listing 6.12 - Adding monsters to the game

```
import random
import player

class Monster(player.Player):
    def __init__(self, location, name, description):
        player.Player.__init__(self, location)
        self.name = name
        self.description = description
    #1
    #2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

def get_input(self):                      #3
    if random.choice((0, 1)):             #3
        return ("go " +
                  random.choice(self.location.exits())) #3
    else:                                #3
        return ""                         #3

def look(self, player, noun):            #4
    return [self.name, self.description] #4

def get(self, player, noun):            #4
    return ["The " + self.name +
            " growls at you."]          #4

#1 - Include the player module and subclass Player
#2 - You need to call the parent classes __init__
#3 - The monster's brain
#4 - Functions for player interaction

```

The first thing that we need to do is to import our player module (**1**). Once that's done, we can use `player.Player` instead of `object` when we create our class. Now instead of using the base object, Python will look in `player.Player` when it tries to find an attribute or method that isn't defined directly in the `Monster` class.

When you're initializing your monster class, you'll also need to initialize the parent class to make sure that everything's set up properly (**2**). In our case, the main thing that needs to be set is the location of the monster.

Here's where you can really see the similarities between the player and monster classes. Our monster AI is a different version of `get_input` which generates a command rather than asking the player to provide one (**3**). To start with, we'll keep things simple, just returning either a blank string to do nothing, or else a random direction to move around (making good use of the `cave` classes `exits()` function).

The player will want to try and interact with the monster, so we need to provide mechanisms for that to occur (**4**). `look` is just copied from the `Item` class and returns the monsters description, and `get` gives an amusing error message.

Now you have a monster class which is fully capable of interacting with the world in the same way that the player can, and which will see all of the same information. This is important for a few reasons. Let's have a deeper look at those reasons and how they tie in to object oriented design.

You are at Wall Street.
A capitalist oppressor is here.
\\ attack capitalist
You smite the capitalist |
with your hammer.
You smite the capitalist
with your sickle.
The capitalist dies.



6.6.2 Some object oriented design tips

The first reason to use inheritance is that you can rely on having the common functionality of the base class, which reduces the amount of 'special casing' that your program needs in order to run. You don't need two separate game loops, one for the player and another for the monsters, or

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=598>

program code which looks like if player then: ... else if monster then: ... you can just treat monsters and players identically.

The second is that it makes your program easier to extend; this effectively builds an interface which monsters, players and whatever else you can think of can interact with the world. If you needed to add a third type of actor to your world you only need to write the parts specific to that actor.

The final reason is that using inheritance greatly reduces the amount of code that you have to write and makes your program much easier to understand, which is always good whether your program is object oriented or not.

The other point to note is that this isn't the only way that you can design your classes. A different, possibly better, way is to create a third class (called something like Mobile, or Actor) with all of the common functionality between the player and the monster, and then have both the player and monster classes inherit from that. In object oriented design, this is normally referred to as an abstract class. You're not supposed to create instances of Actor, but instead subclass it, add the bits which are missing and then create an instance from your subclass.

NOTE Object Oriented terminology can be confusing, but once you've seen a few examples, you'll find it pretty straightforward. Just relate it back to something that you know well, like the Cave, Player and Monster objects in this chapter.

In our case, the advantages of specifying an abstract class aren't immediately clear since we only have two classes, but it's an option in the future if you find that there's functionality which the player class needs, but which monsters shouldn't have access to, or vice versa.

Another design point is that up until now we've favored composition over inheritance. Inheritance is normally referred to as an 'is-a' relationship: A player is-an actor, a monster is-an actor too. Composition, on the other hand is a 'has-a' relationship. A cave has-a player in it, a player has-a number of items. Composition tends to couple your objects less tightly - they have to interact via method calls and inspecting each other's values, as opposed to inheritance, which automatically inserts the methods of one object into the other. Most of the time you'll want to use composition, but when used in the right situation, inheritance can make a big difference.

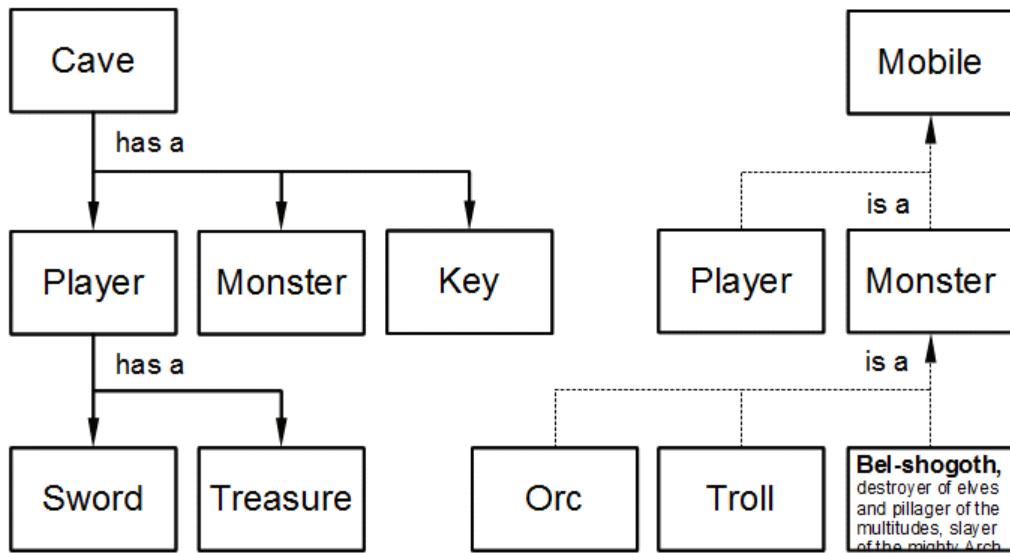


Figure 6.3 - Some of the inheritance and composition in our game

6.6.3 Tying it all together

Now that we have the player and monster classes, we need to make some changes to how the game handles the player when it runs. We no longer want the special case of getting input just for the player, instead our monsters need equal rights too! We'd also like all of the functions that we've been using up until now enclosed within a class, so that it's easier for them to interact properly. Here's a class which does just that - you can use it to set up a game, build caves and gather input until the player has finished.

Listing 6.13 - A Game class

```

import random
import item, player, monster, cave

class Game(object):
    def __init__(self):
        self.caves = self.create_caves() #1
        cavel = self.caves[0]
        sword = item.Item("sword", "A pointy sword.", cavel)
        coin = item.Item("coin", "A shiny gold coin. "
                         "Your first piece of treasure!", cavel)
        orc = monster.Monster(cavel, 'orc',
                             'A generic dungeon monster')
        self.player = player.Player(cavel)
  
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

cave_names = [
    "Arched cavern",
    ...
    "Spooky Chasm",
]
#1

def create_caves(self):
    random.shuffle(self.cave_names)
    caves = [cave.Cave(self.cave_names[0])]
    for name in self.cave_names[1:]:
        new_cave = cave.Cave(name)
        eligible_caves = [each_cave for each_cave in caves
                           if each_cave.can_tunnel_to()]
        old_cave = random.choice(eligible_caves)
        directions = [direction for direction, each_cave
                      in old_cave.tunnels.items()
                      if each_cave is None]
        direction = random.choice(directions)
        old_cave.tunnel_to(direction, new_cave)
        caves.append(new_cave)
    return caves
#1

def do_input(self):
    get_input_from = [thing for cave in self.caves
                      for thing in cave.here
                      if 'get_input' in dir(thing)]
    for thing in get_input_from:
        thing.events = []
        thing.input = thing.get_input()
        # print str(thing) + " input is: " + thing.input
#2

def do_update(self):
    things_to_update = [thing for cave in self.caves
                        for thing in cave.here
                        if 'update' in dir(thing)]
    for thing in things_to_update:
        thing.update()
        # print str(thing) + " result is: " + '\n'.join(thing.result)
#3

def run(self):
    print "\n".join(self.player.location.look(player, ''))
    while self.player.playing:
        self.do_input()
        self.do_update()
        print "\n".join(self.player.events)
        print "\n".join(self.player.result)
#4

if __name__ == '__main__':
    game = Game()
    game.run()
#5

#1 Move game initialization to __init__
#2 Get input from the player and the monsters
#3 Act on the input gathered
#4 Our new game loop

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

#5 Our main() section gets much simpler

This `__init__` function contains all of the setup code that we've been running from the player class so far, creating items, monsters and the player. Within the `__init__` script is a much more sensible place to put it (1).

Next we ask each actor in the game what they're going to do next (2). Note that the `dir()` function applies equally well to your own classes as it does to Python's built-in objects. Splitting the input away from the processing like this means that one actor can't make decisions based on what other actors are about to do, which makes your game easier to understand for both the player and yourself.

Once you've gathered all of the input, each actor will act in turn (3). The mechanism is much the same - build a list of actors and then iterate over them. If you really wanted to be fair you should probably shuffle this list to determine who acts first, but the monsters don't care about fairness.

Here's our main game loop (4). It's basically the same as the one we had previously, except that it calls out to `do_input` and `do_update`, and the player and monsters store their output in a result list. We've also created a separate events list, to store things which happen during each turn.

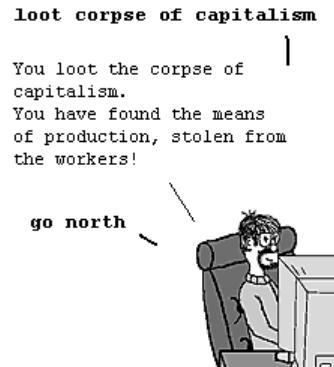
The final benefit is that we now have a nice clean `__main__` loop (5). All of the tricky code which used to be here has been broken up into bits and contained within methods.

The last thing that we need to do is to add an update function to the `Player` class. When the code above checks all of the objects in `world`, it'll see the player, and everything derived from the `Player` class, as things which need to be updated.

```
Class Player(object):
    ...
    def update(self):
        self.result = self.process_input(self.input)
```

All this does is call `process_input` with whatever the player's input has been, and store the return value, which will be a list of strings, in `self.result`.

If you run your program now, you should see an orc in the room with you. Hit the enter key a few times to simulate waiting a while, and the orc should leave for another room. If you have a search around, you should be able to find the orc aimlessly wandering the caves. Unless you want your game to resemble a European art house piece exploring the futility of existence though, we'd better start adding some more interesting game elements.



6.7 Danger and excitement

The final part of our game will be to allow the players and monsters to attack each other. Some sort of element of competition is essential in games, whether you're competing on combat, speed, who can build the biggest city, explore the most or build the best house. In our case we're writing a dungeon adventure, so combat is pretty much essential - anyone who's played Dungeons and Dragons will be expecting to be able to hit orcs. Since combat will be between the player and monsters, we'll start with the Player class and add an attack method.

You are in a factory.
There are workers here.

**give means of production
to workers**

You give the means of
production to the workers.
The workers rejoice!
Congratulations!
You have won
the game!



Listing 6.14 - Attacking other objects

```
class Player(object):

    def __init__(self, location):
        self.name = "Player"
        self.description = "The Player"
        self.hit_points = 3
        self.events = []
        ...

    def attack(self, player, noun):
        hit_chance = 2
        has_sword = [i for i in player.inventory
                     if i.name == 'sword']
        if has_sword:
            hit_chance += 2

        roll = random.choice([1,2,3,4,5,6])
        if roll > hit_chance:
            self.events.append("The " +
                               player.name + " misses you!")
            return ["You miss the " + self.name]

        self.hit_points -= 1
        if self.hit_points <= 0:
            return_value = ["You kill the " + self.name]
            self.events.append("The " +
                               player.name + " has killed you!")
            self.die()
            return return_value

        self.events.append("The " +
                           player.name + " hits you!")
        return ["You hit the " + self.name]

    def die(self):
        self.playing = False
        self.input = ""
        self.name = "A dead " + self.name
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

- #1 - Extra player attributes
- #2 - Calculate a player's chance to hit
- #3 - Roll to hit
- #4 - Report death
- #5 - Report any attacks
- #6 - Handle death

We start by adding some extra attributes that we'll need for the attack method (1). `hit_points` is pretty obvious, `events` is for storing things that happened to the player (or which they saw) during the turn, and we've added a name and description so that we can handle combat easily whether the target is a monster or a player.

We'll use a simple combat mechanism - calculate a to-hit number, roll a die and if the number rolled is under or equal to the to-hit roll, then the player or monster will be hit (2). The to-hit number will normally be 2, but if you have a sword then it'll be 4. Remember that the attack command will be called on the object which is being attacked, rather than the one doing the attacking.

Next we roll to see if we hit (3) - it's just a random choice from 1 to 6. If the number rolled is greater than the to-hit number, then we miss. Before we exit though, we tell both the attacker and the attacked what's happened.

If we are hit, then we lose a hit point. If our hit points are reduced to zero or below, then we die (4). If it's the player that has died, this will trigger the end of the game. Either way we report it, but generate our messages before we call out to the `die()` function, since that may modify it ("You kill the A dead orc!").

If we're not dead yet, then the situation is much the same as a miss - just report it to the attacker and the target, and move on (5).

Since there may potentially be quite a lot of bookkeeping to be done when a monster dies, I've pulled that out into its own method (6). We mark ourselves as not playing, cancel any outstanding orders, and change our name to reflect our newly deceased status.

That's all that you need to do to enable combat in your game! Since your classes are all nicely encapsulated, there's no need to make any changes to the Cave, Item or Game classes at all. Well, not entirely. If you think back to the last chapter or run the code you'll see an obvious problem - the monsters don't fight back. Worse, after you've killed them they still run around! Both of these problems are easy to fix with a simple upgrade to the monster's AI.



Listing 6.15 - Updating your monster's AI

```
def get_input(self):
    if not self.playing:                                #1
        return ""                                       #1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

player_present = [x for x in self.location.here
                  if x.name == "Player"]
#2
if player_present:
    return "attack " + player_present[0].name
#2
if random.choice((0, 1)):
    return "go " + random.choice(self.location.exits())
else:
    return ""

#1 - The monster doesn't know that it's dead
#2 - The monster doesn't like the player very much

```

Dead monsters tell no tales. If we're dead, then we shouldn't be generating any input at all (**1**). Without this section, you'll have an undead orc running around your caves after you kill it.

Next we let the monster get its own back (**2**). If we can see the player in our cave, then we issue an attack order, exactly as the player would if he can see us. If we can't see them, then we fall back to our random wandering around the caves.

So now you have all of the elements of an adventure game - a network of rooms to explore, monsters to attack you and items and treasure to collect to help you in your quest. Armed with the code in this chapter and your imagination, you should be able to create pretty much any type of adventure game that you like.

6.8 *Where to from here?*

The classes and methods that have been introduced in this chapter have really only scratched the surface of what you can add to your game. Depending on the flavor of game that you prefer, you can take your development in any direction. Here are some ideas on how you could extend the game that you've written so far.

6.8.1 *Add more monsters and treasure*

Currently there's only one orc, and a couple of different items. You could add more types of monsters and treasure (or more powerful weapons or weapons which affect monsters differently). You'd also want a score which the player can access with a score method on the player, and which should also be printed out when the player quits, or dies.

6.8.2 *Extend combat and items*

You could extend the Item class or Player.attack method, to add other items which might be useful, such as armor or rope, and something to use them on. If you're adding lots of weapons or armor with different to-hit modifiers, you might want to think about ways to simplify how to find the amount that you add or subtract from the to-hit roll or damage done.

6.8.3 *Add more adventure*

Some adventure games are more about exploring atmospheric locations than killing monsters. You could add proper descriptions during the setup phase, or have a pre-

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

generated cave system instead of a randomly generated one, and add specific methods to handle particular events, such as a boat that sets sail, or a castle drawbridge that you can raise or lower.

6.8.4 Experiment with verbs and nouns

You might want to play around with adding different methods to the `Item` class, to see what you can do when you override built-in methods. For example, you could add the movement methods 'go', 'north', etc. to an item, and make a door that's impassable until the player has the right key. That's also possible with a static monster that needs the right magic sword or a secret password in order to pass. There's also scope for allowing other items to handle particular commands if the game can't find the original object.

6.8.5 Investigate some more advanced features of classes

It's important to note that we haven't dealt with all of the functionality of classes, just the common parts that you'll deal with in 95% of your programs. There are other advanced class features, such as missing method and attribute handling, properties and mix-in classes, which we'll introduce in later chapters as they become relevant. If you're already familiar with classes from other languages though, you might want to have a look through Python's class documentation to see what else you can do with them.

6.9 Summary

We've covered a number of object oriented topics and design issues in this chapter and looked at how classes can help make your programs clearer and easier to understand. In particular, we saw:

- How classes can encapsulate data and functions and initialize them to make instances, which we can reason about and understand much more easily than separate data and functions.
- The ways that classes can interact, calling each other's methods and looking at data to make decisions about what to do.
- That classes can be combined using composition, where instances can contain other instances, and inheritance, where classes can be declared to be particular sub-types of another class.

We haven't covered all of the features of Python's class system yet - in fact far from it - but you now have a firm grasp of the fundamentals of how classes are used and, more importantly, how to use them to solve problems in your programs. In future chapters we'll be making more use of classes and their more advanced features. In the next chapter though, we'll be taking a look at another Python feature which is closely related to the function - the generator.

7

Sufficiently advanced technology...

In this chapter, we're going to be looking at some of the more advanced things that Python can do. In chapter 1, you learned that Python is known as a multi-paradigm language, which means that it doesn't confine you to just one way of doing things. There are three main styles of programming - imperative, object oriented and functional. Python lets you work with all three, and even mix and match them where necessary.

We've already covered imperative and most of object oriented programming in the chapters so far, so this chapter will focus mostly on functional programming and the more advanced parts of object oriented programming in Python.

7.1 *Object Orientation*

Let's start by taking a second look at how our objection orientation classes should be organized, using two separate methods: mixin classes and the `super()` method.

7.1.1 *Mixin classes*

Sometimes you don't need an entire class to be able to do something. Perhaps you only need to add logging, or the ability to save the state of your class to disk. In these cases, you could add the functionality to a base class, or to each class that needs it, but that can get pretty repetitive. There's an easier way, called a mixin class.

The idea is that a mixin class contains only a small, self contained piece of functionality, usually just a few methods or variables, which are unlikely to conflict with anything in the child class.

Listing 7.1 - A logging mixin

```
class Loggable(object):
    """Mixin class to add logging."""
    log_file_name = 'log.txt'
    def log(self, log_line):
        #1
        #1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

file(self.log_file_name).write(log_line)           #1

class MyClass(Loggable):                      #2
    """A class that you've written."""
    log_file_name = "myclass_log.txt"          #3
    def do_something(self):
        self.log("I did something!")          #3

#1 Define the mixin
#2 Inherit
#3 Use the mixin's methods

```

The mixin class is defined **(1)** in exactly the same way as a regular class. Here I've added a class variable for the file name, and a method to write a line to that file. If you want to use the mixin class, all you need to do is inherit from it **(2)**.

Once you're in the child class, all of the mixin's methods and variables become available **(3)**, and you can override them if you need to.

Using simple file logging like this works well, but here's a slightly more involved version which uses Python's built in logging module. The advantage of this version is that as your program grows, you can take advantage of some of the different logging methods - you could send it to your system's logs, or automatically roll over to a new file if the old one gets too big.

Listing 7.2 - Using Python's logging module

```

import logging

class Loggable(object):
    """Mixin class to add logging."""

    def __init__(self,
                 log_file_name = 'log.txt',
                 log_level = logging.INFO,
                 log_name = 'MyApp'):
        self.log_file_name = log_file_name
        self.log_level = log_level
        self.log_name = log_name
        self.logger = self._get_logger()

    def _get_logger(self):
        logger = logging.getLogger(self.log_name)
        logger.setLevel(self.log_level)

        handler = logging.FileHandler(
            self.log_file_name)
        logger.addHandler(handler)

        formatter = logging.Formatter(
            "%(asctime)s: %(name)s - "
            "%(levelname)s - %(message)s")

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        handler.setFormatter(formatter)          #2
        return logger                         #2

    def log(self, log_line, severity=None):    #3
        self.logger.log(severity or self.log_level,   #3
                        log_line)                  #3
        #3

    def warn(self, log_line):                 #3
        self.logger.warn(log_line)            #3
    ...

class MyClass(Loggable):                   #3
    """A class that you've written."""

    def __init__(self):                    #4
        Loggable.__init__(self,           #4
                           log_file_name = 'log2.txt')      #4
        #super(MyClass, self).__init__(           #4
        #                           log_file_name = 'log2.txt')      #4

    def do_something(self):                #5
        print "Doing something!"          #5
        self.log("I did something!")      #5
        self.log("Some debugging info", logging.DEBUG)  #5
        self.warn("Something bad happened!") #5

test = MyClass()                          #5
test.do_something()                      #5

```

#1 - Initialize Loggable
#2 - Create a Logger object
#3 - Logging methods
#4 - How do we call Loggable.__init__()?
#5 - Creating our class and its logging methods

GREG, PITR - WE'VE DECIDED
THAT YOU PROGRAMMERS
NEED A NEW, BIGGER OFFICE.
ONE WITH A DOOR!

Rather than rely on class methods, it's better to instantiate them properly, from an `__init__` method **(1)**. This way we can take care of any extra initialization that we need to do, or require that variables are specified on creation.

(2) is all of the setup that we need to do when creating a Logger from Python's logging module. First we create a Logger instance, then we can add a handler to it, to specify what happens to log entries, and a Formatter to that handler, to tell it how to write out our log lines.



Our mixin class also needs methods so that we can log **(3)**. One option is to use a generic log method where you can give it a severity when you call it, but a cleaner way is to use the logger's methods like `debug`, `info`, `warn`, `error` and `critical`.

Now that we're using `__init__` in Loggable, we'll need to find a way to call it (4). There are two ways. The first is to call each parent class explicitly, by using its name and method directly, but passing in `self`. The second is to use Python's `super()` method, which finds the method in the next parent class. In this case they do much the same thing, but `super()` properly handles the case where you have a common grandparent class. See the next section for the catches when using it, and the sample code in `super_test.py` in the code tarball for this book.

Once all that's done, we can use our logging class in exactly the same way that we did in the previous version (5). As well as this, we've exposed the logger object itself, so that if we need to, we can call its methods directly.

7.1.2 `super()` and friends

Using the `super()` method with "diamond inheritance" (see figure 7.1) can be fraught with peril - the main reason being that when you use it with common methods such as `__init__`, you're not guaranteed which classes' `__init__` method you'll be calling. Each will get called, but they could be in any order. To cover for these cases, it helps to remember the following things when using `super()`:

- First, use `**kwargs`, avoid using plain arguments, and always pass all of the arguments that you receive to any parent methods. Other parent methods might not have the same number or type of arguments as the subclass, particularly when calling `__init__`.
- Secondly, if one of your classes uses `super()`, then they all should. Being inconsistent means that an `__init__` method might not be called, or might be called twice.
- Finally, you don't necessarily *need* to use `super()` if you can design your programs to avoid diamond inheritance, ie. without parent classes sharing a grandparent.

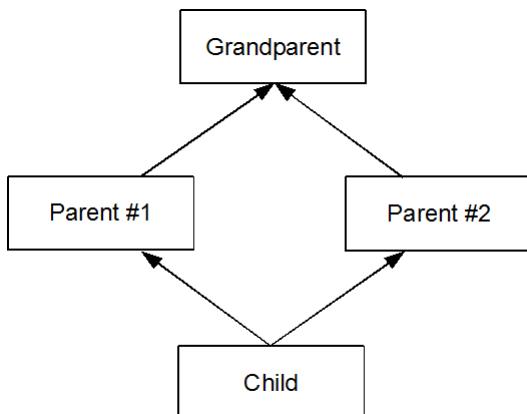


Figure 7.1 - A diamond inheritance structure

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Now you have more of an idea how classes should be organized, and what to watch out for when using multiple inheritance, let's take a look at some of the other things that we can do with classes.

7.2 Customizing classes

Python gives you a great deal of power when it comes to defining how the methods in your class work, and which methods are called. Not only do you have access to all of Python's introspection power, you can also decide to use different methods at runtime, even methods which don't exist.

When Python looks up an attribute or method on a class (for example `self.log_file_name` or `test.do_something()` in the previous example), it'll look up that value in a dictionary called `dict`. `dict` stores all of the user defined values for a class, and uses it for most lookups, but it's possible to override it at several points.

Python provides a number of possible ways to customize attribute access, by overriding some built in methods - in the same way that you've been using `__init__` to initialize your classes.

7.2.1 `__getattr__`

`__getattr__` is used to provide methods or attributes when they're not found in the class or a parent class. You can use this to catch missing methods, or write wrappers around other classes or programs.

Listing 7.3 - using `__getattr__`

```
class TestGetAttr(object):

    def __getattr__(self, name):
        print "Attribute '%s' not found!" % name
        return 42

    test_class = TestGetAttr()
    print test_class.something

    test_class.something = 43
    print test_class.something

#1 - __getattr__ method
#2 - using the method
#3 - normal attributes
```

The `__getattr__` method (**1**) takes one argument - the attribute name, and returns what the value should be. In our case, we just print the name, and then return a default value, but you could do anything - log to a file, call an API or hand over the responsibility to another class or function.

Now when we try and access a value that doesn't exist in our class, `__getattr__` will step in and return our default value (**2**).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Since `__getattr__` is only called when the attribute isn't found, setting an attribute first (3) means that `__getattr__` won't be run.

Now that we can get our attributes, let's learn how to set them as well.

7.2.2 `__setattr__`

`__setattr__` is used to change the way that Python alters attributes or methods. You can intercept calls to your class, log them or do whatever you need to. Listing 7.4 shows a simple way to catch attribute access and redirect it to a different dictionary instead of inserting it into the default `__dict__`.

Listing 7.4 - using `__setattr__`

```
class TestSetAttr(object):

    def __init__(self):
        self.__dict__['things'] = {}

    def __setattr__(self, name, value):
        print "Setting '%s' to '%s'" % (name, value)
        self.things[name] = value

    def __getattr__(self, name):
        try:
            return self.things[name]
        except KeyError:
            raise AttributeError(
                "'%s' object has no attribute '%s'" %
                (self.__class__.__name__, name))

test_class2 = TestSetAttr()
test_class2.something = 42
print test_class2.something
print test_class2.things
print test_class2.something_else

#1 - set up the replacement dictionary
#2 - __setattr__ inserts into things
#3 - __getattr__ reads from things
#4 - using the class
```

(1) is where we're setting `things`, which will store all of our attributes that we'll set. One catch when using `__setattr__` is that you can't directly set something in the class, since that will result in `__setattr__` calling itself and looping until Python runs out of recursion room. You'll need to set the value in the classes' `__dict__` attribute directly, like we do here.

Once `things` is set in `__dict__` though, we can read from it normally, since `__getattr__` won't be



called when we access `self.things`. `__setattr__` takes a name and a value, and in our case we're inserting the value into the `things` dictionary (**2**) instead of into the class.

This version of `__getattr__` just looks in the `self.things` dictionary for our value (**3**). If it's not there, we raise an `AttributeError`, to mimic Python's normal handling.

The class that we've written behaves exactly like a normal class, except that we have close to complete control over how its methods and attributes are read (**4**). If we want to override everything though, we'll need to use `__getattribute__`.

7.2.3 `__getattribute__`

Another way is to override all method access entirely. If `__getattribute__` exists in your class, it'll be called for all method and attribute access.

Well, that's sort of true. Strictly speaking, even `__getattribute__` doesn't override everything. There are still a number of methods, such as `__len__` or `__init__`, which are accessed directly by Python and won't be overridden. But everything else, even `__dict__`, goes through `__getattribute__`. This works, but in practice means that you'll have a hard time getting to any of your actual attributes. If you try something like `self.thing`, then you'll end up in an infinite `__getattribute__` loop.

So how do you fix this? `__getattribute__` won't be much use if we can't access the real variables. The answer is to use a different version of `__getattribute__`; the one that you would normally be using if you hadn't just overridden it. The easiest way to get to a fresh `__getattribute__` is via the base object class, and feed in `self` as the instance. Listing 7.5 shows you how.

Listing 7.5 - using `__getattribute__`

```
class TestGetAttribute(object):
    def __init__(self, things=None):
        my_dict = object.__getattribute__(self, '__dict__')
        if not things:
            my_dict['things'] = {}
        else:
            my_dict['things'] = things

    def __setattr__(self, name, value):
        print "Setting '%s' to '%s'" % (name, value)
        my_dict = get_real_attr(self, '__dict__')
        my_dict[name] = value

    def __getattribute__(self, name):
        try:
            my_dict = get_real_attr(self, '__dict__')
            return my_dict['things'][name]
        except KeyError:
            my_class = get_real_attr(self, '__class__')
            raise AttributeError(
                "'%s' object has no attribute '%s'" %
                (my_class.__name__, name))
#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
#2
#2
#2
#2
#2
#3
#3
#3
#3
#3
#3
#3
#3
#3
#3
#3
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        (my_class.__name__, name)) #3

def get_real_attr(instance, name): #2
    return object.__getattribute__(instance, name) #2

test_class3 = TestGetAttribute({'foo': 'bar'}) #4
print object.__getattribute__(test_class3, '__dict__') #4
test_class3.something = 43 #4
print object.__getattribute__(test_class3, '__dict__') #4
print test_class3.foo #4

#1 - Borrowing object's getattribute
#2 - An easier way
#3 - __getattribute__
#4 - Using our class

```

Python methods are really just functions, so it's relatively easy to call back to `object`. The only thing we need to do is to pass it `self` as the instance, and the name of the attribute that we want **(1)**. I've also updated `__init__` so that you can pass in values to set up the internal things dictionary.

To tidy up our calls to `object`, we can define a helper function to make the call for us. **(2)** is a version of `__setattr__` which uses it.

Other than needing to use `object` to get the dictionary that we're editing, the call to `__getattribute__` **(3)** is much like the one to `__getattr__` - it receives a name, and returns a value, converting `KeyError` to `AttributeError` along the way.

After we've been through all that, our class is ready to be used **(4)**. It follows the same usage pattern, but we can now hide the `things` dictionary from casual inspection. It's still visible if we use the old `object.__getattribute__` though.

If using `__getattribute__` seems like a lot of work, don't worry. Most of the time you won't need to use it. But many third party libraries make use of it as well as `__getattr__` and `__setattr__` - if you need to use them, they can save a lot of work and make your classes' interfaces a lot more Pythonic and easy to use.

7.2.4 Properties

A more specific method for customizing your attributes is to use Python's `property` function. Where `__getattr__` and `__getattribute__` work across the entire class, `property` allows you to specify functions, commonly known as getters and setters, which are responsible for controlling access to an attribute or method.

Properties solve a common programming problem, namely how to customize attribute access without altering your classes' external interface. Without properties, it's standard practice to use getters and setters for every attribute, even if you don't need them



yet, because of the difficulty in switching from attribute access to using a function, but Python allows you to do this without having to change everything that uses your class.

Listing 7.6 - using properties

```
class TestProperty(object):

    def __init__(self, x):                      #1
        self._x = x

    def get_x(self):                           #2
        return self._x

    def set_x(self, value):                   #3
        if not (type(value) == int and 0 < value < 32):
            raise ValueError("TestProperty.x "
                               "must be an integer between 0 and 32")
        self._x = value

    x = property(get_x, set_x)                 #4

test = TestProperty(10)                         #5
print test.x
test.x = 11                                     #5
test.x += 1                                     #5
assert test.x == 12                            #5
print test.x                                     #5

try:                                              #6
    test2 = TestProperty(42)
except ValueError:                                #6
    # ValueError: TestProperty.x must be          #6
    # an integer between 0 and 32               #6
    print "test2 not set to 42"                  #6

#1 - class setup
#2 - x is really _x
#3 - setting _x
#4 - defining the property
#5 - the interface
#6 - bounds checking
```

Our initial setup (**1**) looks much like any classes `__init__` function. Some introductions set the hidden variable directly, but I prefer it this way, since it means that we can't have `x` set to something out of bounds.

(**2**) is our getter, which just returns the value of `_x`, although we could convert it to whatever we liked, or even just return `None`.

(**3**) is our setter, which first checks to make sure that our value is an integer from 0 to 32. If it isn't, then we raise a `ValueError`.

Finally, we set `x` on the class to be a property (**4**), and pass it the getter and setter functions, `get_x` and `set_x`. Note that you can also define a read only property if you only

pass a getter. If you then try and set `x`, you'll get `AttributeError: can't set attribute`.

If you didn't know that it was a property, you wouldn't be able to tell just by using the class. The interface (5) for our defined `x` is exactly the same as if it were a regular attribute.

The only exception to the interface is the one that we've included. If you try and set the value of `test2.x` to something out of bounds, you'll get an exception (6).

In practice, you'll want to use the methods which are most suited for your use case. Some situations, such as logging, wrapping a library or security checking, call for `__getattribute__` or `__getattr__`, but if all you need to do is customize a few specific methods, then properties are normally the best way to do it.

7.2.5 Emulating other types

One other common practice is to write classes to emulate certain types, such as lists or dictionaries. If you have a class which is supposed to behave similarly to a list or a number, it helps when the class behaves in exactly the same way, supporting the same methods and raising the same exceptions when you misuse it.

There are a number of methods that you can define which Python will use when you use your class in certain ways. For example, if you need two instances of your class to compare as equal you can define an `__eq__` method which takes two objects, and returns `True` if they should be treated as equal.

As an example, look at listing 7.7, which adds two methods to our previous class so that we can compare them to each other. I've also renamed the class to `LittleNumber`, to make its purpose clearer (you'll also want to rename the classname in the exception).



Listing 7.7 - extending our properties

```
class LittleNumber(object):
    ...
    def __eq__(self, other):
        return self.x == other.x
    #1
    #1

    def __lt__(self, other):
        return self.x < other.x
    #2
    #2

    def __add__(self, other):
        try:
            if type(other) == int:
                return LittleNumber(self.x + other)
            #3
            elif type(other) == LittleNumber:
                return LittleNumber(self.x + other.x)
            #3
            else:
                #3
        
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        return NotImplemented                         #3
    except ValueError:                            #3
        raise ValueError(                         #3
            "Sum of %d and %d is out of bounds "   #3
            "for LittleNumber!" % (self.x, other.x))  #3
    def __str__(self):                           #3
        return "<LittleNumber: %d>" % self.x      #3

one = LittleNumber(1)                          #4
two = LittleNumber(2)                          #4
print one == one                             #4
print not one == two                         #4
print one != two                            #4
print one < two                             #4
print one > two                            #4
print not one > two                         #4
print two >= one                            #4
print two >= two                            #4

onetoo = LittleNumber(1)                      #4
print onetoo == one                          #4
print not onetoo == two                      #4

print onetoo + one                          #4
print one                                #4
print onetoo + one == two                  #4

#1 - __eq__
#2 - __lt__
#3 - adding values
#4 - using our class

```

This method **(1)** checks our value against the other one that we're given. Whenever Python encounters `a == b`, it will call `a.__eq__(b)` to figure out what the real value should be.

In the same way as `__eq__`, `__lt__` **(2)** will compare two values and return `True` if the current instance is less than the one passed in.

`__add__` is also useful, and should return the result of adding something to our class **(3)**. This case is somewhat more complex - we return a new `LittleNumber` if we're passed an integer or another `LittleNumber`, but we need to catch two cases: where the value goes out of bounds and where someone passes us a different type, such as a string. If we can't (or won't) handle a particular case, we can just return `NotImplemented`, and Python will raise a `TypeError`. Again, a more understandable error message here will save us a lot of debugging further down the track.

Believe it or not, that's all we need to get our class to behave something like an integer **(4)**. Note that we don't necessarily need to implement all of the mirror functions like `__gt__` or `__ne__`, since Python will try their opposites if they're not defined. All of the expressions here should return `True`.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Here's a table of some of the most common methods that you'll want to override if you're providing a class similar to some of the built-in types.

Table 7.1 Common methods you may want to override

Type	Methods	
Most types	<code>__eq__(self, other)</code> <code>__ne__(self, other)</code> <code>__gt__(self, other)</code> <code>__lt__(self, other)</code> <code>__le__(self, other)</code> <code>__ge__(self, other)</code> <code>__str__(self)</code> <code>__repr__(self)</code>	Tests for equality and relative value, <code>==</code> , <code>!=</code> , <code>></code> , <code><</code> , <code><=</code> and <code>>=</code> . Return a printable version of the class and a printable representation of the class.
Dictionary, List or other container	<code>__getitem__(self, key)</code> <code>__setitem__(self, key, value)</code> <code>__delitem__(self, key)</code> <code>keys(self)</code> <code>__len__(self)</code> <code>__iter__(self)</code> and <code>iterkeys(self)</code>	Get, set and delete an entry. Return a list of keys (dictionaries only). Return the number of entries. If your object is large, you might want to consider using these methods to return an iterator (see next section for details).
Numbers	<code>__contains__(self, value)</code> <code>__add__(self, other)</code> <code>__sub__(self, other)</code> <code>__mul__(self, other)</code> <code>__floordiv__(self, other)</code> <code>__pow__(self, other)</code> <code>__int__(self)</code> <code>__float__(self)</code>	Is the value one of our entries (if we're a list or set), or one of our keys (if we're a dictionary). Return the result of adding, multiplying, dividing and raising to a power. Used to convert an instance of your class into an integer or float.

These are by no means the only methods that you can set, but they're by far the most common unless you're doing something really exotic. Let's have a look at a practical example of how these methods are used in practice, by looking at Python's generators and iterators.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

7.3 Generators

Generators are one of Python's best kept secrets after list comprehensions, and worth looking into in more detail. They're intended to solve the problem of storing state in between function calls, but they're also very useful for cases where you need to deal with large amounts of data, perhaps too large to fit easily in memory.

First though we'll look at iterators, Python's method for dealing with looping objects, before we look at how we can make use of generators to quickly deal with large amounts of data in log files.

7.3.1 Iterators

You've been using iterators throughout the book so far, right from chapter 2, because every time you use a `for` loop or a list comprehension, iterators have been acting behind the scenes. You don't need to know how iterators work in order to make use of them, but they're useful for understanding how generators operate.

Note Iterators are a common solution to a frequent programming task: I have a bunch of things - now how do I do something to each one of them? The trick is that most Python collections, be they lists or files or sets, can be used as iterators.

The interface of an iterator is very straightforward. It has a `.next()` method, which you call over and over again to get each value in the sequence. Once all of the values are gone, then the iterator raises a `StopIteration` exception, which tells Python to stop looping. In practice, using an iterator looks something like figure 7.1.

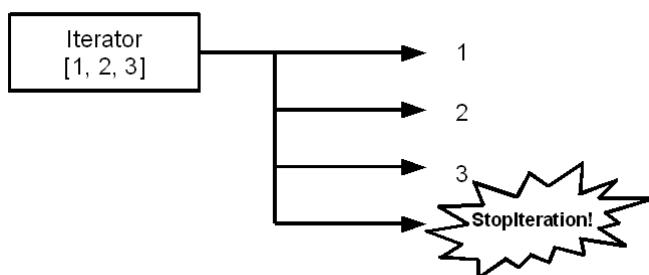


Figure 7.1 - The iterator protocol – once we run through three iterations, it stops.

When you ask Python to iterate over an object such as a list, the first thing that it does is call `iter(object)`, which calls that object's `__iter__` method and expects to get an iterator object. You don't need to use the `__iter__` call directly unless you're creating your

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

own custom iterator. In that case, you'll need to implement both the `__iter__` and the `next()` methods yourself.

Listing 7.8 - Using an iterator the hard way

```
>>> my_list = [1, 2, 3]
>>> foo = iter(my_list)           #1
>>> foo                         #1
<listiterator object at 0x8b3fbec> #1
>>> foo.next()                  #2
1
>>> foo.next()                  #2
2
>>> foo.next()                  #2
3
>>> foo.next()                  #3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration                   #3
>>>
#1 - Create an iterator
#2 - Iterate over it
#3 - No more values
```

We use the `iter()` function to create an iterator object for `my_list` (**1**). If you print it, you can see that's it's a new type of object - a `listiterator` - not a list. As we call the `next()` method of our iterator (**2**), it returns the values from the list: 1, 2 and 3.

Once we've run out of values (**3**), the iterator will raise a `StopIteration` exception to signal that the iterator has finished.

The iterator protocol is simple, but it's a fundamental underpinning of the looping and iteration mechanisms in Python. Let's have a look at how generators use this protocol to make your programming life easier.

7.3.2 Generators

Generators are similar to iterators. They use exactly the same `.next()` method, but they're easier to create and to understand. Generators are defined exactly like functions, except that they use a `yield` statement instead of a `return`. Here's a very simple example of a generator, which counts down to zero from a specified value.

```
def counter(value):           #1
    print "Starting at", value #1
    while value > 0:          #2
        yield value           #3
        value -= 1
```

HERE YOU GO. REARRANGE
IT HOWEVER YOU LIKE -
~~JUST DON'T LOSE ANY~~
OF THE BOSS' EMAILS.



```
#1 - Definition
#2 - Looping
#3 - Yield
```

Let's look at this one step at a time.

Generators start out just like functions, including the way that you give them arguments **(1)**. We're also including a debugging string here, so we can follow how the generator is called.

Generators need some way to return values repeatedly, so usually you'll see a loop **(2)** within the body.

The `yield` statement **(3)** will stop our function and return the value that we give it.

Finally, Python will return to after each call to the generator's `next()` method. We subtract 1 from it, and then the loop will pass through again.

That's only half of the puzzle though - we still need to be able to call our generator. Listing 7.9 shows how you can do that, by creating a counter and then using it in a for loop. You can also call it directly, with a line like `for x in counter(5)`.

Listing 7.9 - Using our counter generator

```
>>> foo = counter(5)                                #1
>>> foo                                         #1
<generator object at 0x896054c>                  #1
>>> for x in foo:                               #2
...     print x                                    #2
...
counting down from 5                                #3
5                                         #3
4                                         #3
3                                         #3
2                                         #3
1                                         #3
>>>
#1 - Create our counter generator
#2 - Use our generator in a loop
#3 - The output from our generator
```

First we create our counter **(1)**. Even though it looks like a function, it doesn't start straight away, instead it returns a generator object.

We can use our generator object in a loop like **(2)**. Python will call the generator repeatedly until it runs out of values, or the generator exits normally.

(3) is what our loop prints out. The first line is the initial debug from the generator, and the other lines are the results that it returns.

There's one last mechanism that you should know, which can save you a lot of time setting up generator functions.

7.3.3 Generator Expressions

Generator expressions are much like list comprehensions, but behind the scenes they use generators rather than building a whole list. Try running the following expressions in a Python prompt:

```
foo = [x**2 for x in range(1000000)]      #1
bar = (x**2 for x in range(1000000))      #2

#1 - A list comprehension
#2 - A generator
```

Depending on your computer, the list comprehension **(1)** will either take a long time to return, or else raise a `MemoryError`. That's because it's creating a million results and inserting them into a list.

This generator **(2)**, on the other hand, will return immediately. It hasn't created any results at all - it'll only do that if you try to iterate over `bar`. If you break out of that loop after 10 results, then the other 999,990 values won't need to be calculated.

So if generators and iterators are so great, why would you ever use lists or list comprehensions? They're still useful if you want to do anything else with your data other than loop over it. If you want to access your values in a random order, say the fourth value, then the second, then the eighteenth, then your generators won't help since they access values linearly from the first through to the one millionth. Similarly, if you need to add extra values to your list, or modify them in some way then generators won't help - you'll need a list.

So now that we know how generators work, let's look at where they can be useful when we write our programs.

7.4 Using generators

As we said at the start of the chapter, we can use generators in cases where reading in a large amount of data would slow our program down, or make it run out of memory and crash.

Note In case you haven't realized it yet, Python is an intensely pragmatic language.

Every feature has gone through a rigorous community-based vetting process known as a PEP, or Python Enhancement Proposal, so there will be strong use cases for each feature.

You can read more about PEPs at: <http://www.python.org/dev/peps/pep-0001/>

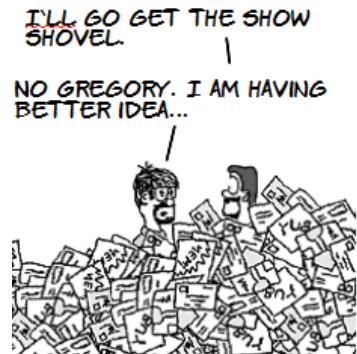
A common problem is the processing of large files. If you have a few hundred log files, and need to find out which ones have a certain string in them, or collate data across several website directories, then it can be hard to make sense of what's happening within a reasonable amount of time. Let's take a look at a few simple generators that can make your life easier if you run into this sort of problem.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

7.4.1 Reading files

One of the areas where Python makes use of generators is in the file processing sections of the `os` module. `os.walk` is a good example - it allows you to iterate recursively through directories, building lists of the files and subdirectories within them, but it builds the list as it goes, so it's nice and fast. We've already encountered `os.walk` in chapter 3, when we were building a program to compare files. A typical use is in listing 7.10, which is a program to read a directory and return the files which are of a certain type - in this case '`.log`' files.



Listing 7.10 - `os.walk` revisited

```
import os

dir_name = '/var/log'                      #1
file_type = '.log'                         #1

for path, dirs, files in os.walk(dir_name):    #2
    print path
    print dirs
    print [f for f in files if f.endswith(file_type)]   #3
    print '-' * 42
#1 - Directory and file type
#2 - Iterate using os.walk
#3 - Log files
```

First we specify our directory and the file type that we want to search for (1). `os.walk` returns a generator which we can use to iterate over the directory (2). It'll give us the path of the directory, as well as any subdirectories and files within it.

We assume that anything which ends in '`.log`' is a log file (3). Depending on your specific situation, an assumption like this may or may not be warranted, but since we will in practice have control of the web server, we can add the `.log` part if we need to.

When you run the program in listing 7.10, it will output something like listing 7.11. Each section contains the directory that you're iterating over, then its subdirectories and the log files within the current directory.

Listing 7.11 - The output from `os.walk`

```
/var/log
['landscape', 'lighttpd', 'dist-upgrade', 'apparmor', ... ]
['wpa_supplicant.log', 'lpr.log', 'user.log', ... ]
-----
/var/log/landscape
[]
['sysinfo.log']
-----
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```
/var/log/dist-upgrade
[]
['main.log', 'apt-term.log', 'xorg_fix_intrepid.log', ... ]
-----
...
```

We can use some generators to make our code a little bit easier to work with. As an example, let's say that we're monitoring our web server for errors, so we want to find out which of our log files have the word 'error' in them. We'd also like to print out the line itself, so that we can track down what's going on if there are any errors.

Listing 7.12 - Generators to work through a directory

```
import os

def log_files(dir_name, file_type):                      #1
    if not os.path.exists(dir_name):                      #1
        raise ValueError(dir_name + " not found!")         #1
    if not os.path.isdir(dir_name):                       #1
        raise ValueError(dir_name + " is not a directory!") #1
    for path, dirs, files in os.walk(dir_name):           #1
        log_files = [f for f in files                   #1
                      if f.endswith(file_type)]            #1
        for each_file in log_files:                      #1
            yield os.path.join(path, each_file)           #1

def log_lines(dir_name, file_type):                      #2
    for each_file in log_files(dir_name, file_type):     #2
        for each_line in file(each_file).readlines():     #2
            yield (each_file, each_line.strip())

def list_errors(dir_name, file_type):                    #3
    return (each_file + ': ' + each_line.strip()          #3
            for each_file, each_line in                  #3
                log_lines(dir_name, file_type)             #3
            if 'error' in each_line.lower())              #3

if __name__ == '__main__':
    dir_name = '/var/log'
    file_type = '.log'
    for each_file in log_files(dir_name, file_type):      #4
        print each_file
    print
    for each_error in list_errors(dir_name, file_type):   #4
        print each_error

#1 - Wrap os.walk in a generator
#2 - Generator for each line of our files
#3 - Filter out non-error lines
#4 - Creating our generators
```

This is the same code that we saw in listing 7.10, but



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

we've wrapped it in a generator function **(1)**. One issue with `os.walk` is that it won't raise an exception if you give it a nonexistent directory or something that's not a directory, so we catch both of those cases before we start.

Now that we have `log_files` as a generator, we can use it to build further generators. `log_lines` reads each file in turn and yields successive lines of each log file along with the name of the file **(2)**.

This generator builds on the `log_lines` generator to return only those lines which have the word 'error' in them **(3)**. Notice also that we're returning a generator comprehension instead of using `yield`. This is an alternative way of doing things which can make sense for small generators, or where the values that you're returning fit the generator comprehension style well.

Once we've done all the hard work of creating our generators **(4)**, calling them is very easy - just give them the directory and file type, and do what you need to with each result.

So now we can find all of the error lines in all of the log files in a certain directory. Returning just the lines with 'error' in them isn't particularly useful though. What if we had an error which didn't contain the word 'error'? Something like 'Process #3456 out of memory!' There are all sorts of conditions that we'd like to check in our log files, so we'll need something a little more powerful.

7.4.2 Getting to grips with our log lines

We'd like to have a lot more control over the data in our log files, including being able to filter by any field or combination of fields. In practice, this means that we'll need to break the data from each line in our log file up and interpret the bits. Listing 7.13 shows some examples from an old Apache access log that I had lying around.

Listing 7.13 - Apache log lines

```
124.150.110.226 -- [26/Jun/2008:06:48:29 +0000] "GET / HTTP/1.1" 200 99 "--
" "Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.14) Gecko/20080419
Ubuntu/8.04 (hardy) Firefox/2.0.0.14"

66.249.70.40 -- [26/Jun/2008:08:41:18 +0000] "GET /robots.txt HTTP/1.1"
404 148 "-- "Mozilla/5.0 (compatible; Googlebot/2.1;
+http://www.google.com/bot.html)"

65.55.211.90 -- [27/Jun/2008:23:33:52 +0000] "GET /robots.txt HTTP/1.1"
404 148 "-- "msnbot/1.1 (+http://search.msn.com/msnbot.htm)"
```

These lines are in Apache's Combined Log Format, and the fields have the following meaning:

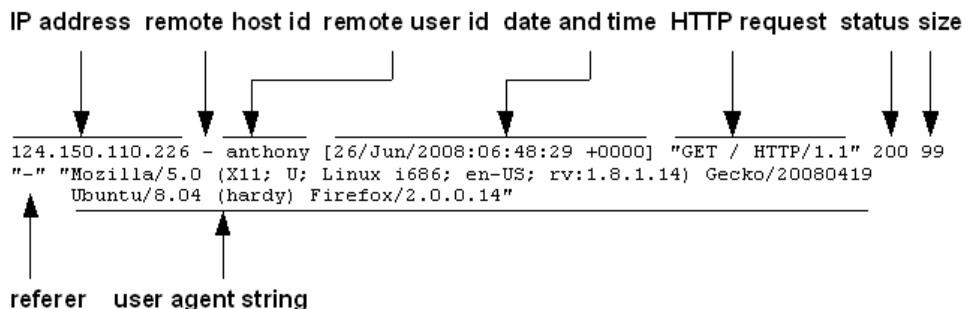


Figure 7.2 - An Apache log line

Most of the fields are self explanatory - IP address is the computer making the request, referer will be the URL of the page (if any) which was used to reach the page, HTTP request contains the path which the user was requesting, size is the number of bytes transferred as a result of the request, and so on.

So in listing 7.13 you should be able to see three separate requests, one from Firefox running under Linux, one from Google's search spider, and one from Microsoft's MSN. As you learned in chapter 5, the user agent string is supplied by the client and so can't really be trusted, but in most cases it's accurate. The HTTP request part is the full command sent to the web server, so it includes the type of request (usually GET or POST) and the HTTP version as well as the path requested.



7.4.3 Pulling out the bits

So that explains what the separate fields mean, but how are we going to get them? We could split on quotes or spaces, but it's possible that they'll appear in odd places and throw the split function off track. For example, there are spaces in both the user agent and the date time string. It's also possible to have quotes within the user agent string and the URL, although they're supposed to be URL encoded.

Tip My rule of thumb is to use Python's string methods like `.endswith()` and `.split()` when looking for simple things, but I find that they can get unwieldy when you're trying to match against more complicated patterns, like the Apache log line.

In cases like this, it's usually easier to break out the "big guns" right at the start, rather than experiment with splitting the fields on various characters and trying to make sure that

it'll work for everything. In our case, the fastest solution is probably to break out a parsing tool called a regular expression, which is useful for reading single lines like this and breaking it down into chunks.

Regular expressions work by using special matching characters to designate particular types of character, such as spaces, digits, the letters a to z, A to Z and so on. A full description of the gory details of regular expressions is out of the scope of this book, but there's a handy quick reference in table 7.2 which will get you started.

Table 7.2 - A regular expression cheat sheet

Expression	Definition
\	Regular expressions use a backslash for special characters. If you need to match an actual backslash, then you can use two backslashes together, \\
\w	A 'word' character, a-zA-Z0-9 and a few others, such as underscore
\W	A non-word character - the opposite of \w
\s	A white space character, such as space or tab
\S	A non-white space character
\d	A digit character, 0-9
.	Matches any character at all
+	Extend a special character to match one or more times. \w+ will match at least one word character, but could match 20
*	Like +, but match zero or more instead of one or more.
?	You can use this after a * or + wildcard search to make them less 'greedy'. .*? will match the minimum that it can, rather than as much as possible.
()	You can put brackets around a set of characters and pull them out later using the .groups() method of the match object.
[]	You can put characters between square brackets to match just those. [aeiou] for example, matches vowels.
r'''	A string preceded with r is a 'raw' string, and Python won't escape any backslashes within it. For example, "line 1\nline 2" will be normally be split over multiple lines, since Python will interpret \n as a return, but r"line 1\nline 2" won't.
match vs. search	There are two main methods used on the regular expression object. match will try and match from the start of a line, but search will look at the whole string. Normally you'll want to use search, unless you know that you want to match at the start.

The regular expression string that we're going to use to match our Apache log lines looks like this:

```
log_format = (r'(\S+) (\S+) (\S+) \[(.*?)\] '
              r'"(\S+) (\S+) (\S+)" (\S+) (\S+) '
              r'"(.+)" "(.+)")')
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

It looks complicated, but it's not so hard if we break it down and look at the individual parts.

- Most of the fields are groups of alphanumeric characters separated by spaces, so we can just use `(\S+)` to match them. They're surrounded by brackets so that we can access the fields after they've been matched. Each part above corresponds to one field in the Apache log line.
- The date and time field is the only one with square brackets around it, so we can match that easily too and pull out everything, including spaces, with a wildcard match. Notice that we've escaped the [and] by putting a backslash in front of them so that the regular expression treats them as normal characters.
- Finally, the referer and the user agent are matched using wildcards too, since they might have quotes or spaces in them.
- The whole string is wrapped in brackets so that we can break it over multiple strings but still have Python consider them as a single string.

Now that you have a rough idea of how we can use regular expressions to match the fields in a log line, let's look at how we write the Python functions and generators to make sense of the overall scope of our log files. Listing 7.16 extends listing 7.7 to add new Apache related functions and generators.



Listing 7.16 - Parsing Apache log lines

```
import re #1
...
apache_log_headers = ['host', 'client_id', 'user_id', #1
                      'datetime', 'method', 'request', 'http_proto', #1
                      'status', 'size', 'referrer', 'user_agent'] #1
log_format = (r'(\S+) (\S+) (\S+) \[(.*?)\] ' #1
              r'"(\S+) (\S+) (\S+)" (\S+) (\S+) ' #1
              r'"(.+)" "(.+)")') #1
log_regex = re.compile(log_format) #1

def parse_apache(line): #2
    log_split = log_regex.match(line) #2
    if not log_split: #2
        print "Line didn't match!", line #2
        return {} #2
    log_split = log_split.groups() #2

    result = dict(zip(apache_log_headers, log_split)) #3
    result['status'] = int(result['status']) #3
    if result['size'].isdigit(): #3
        result['size'] = int(result['size']) #3
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

else:                                     #3
    result['size'] = 0                     #3
    return result                         #3

def apache_lines(dir_name, file_type):      #4
    return (parse_apache(line)              #4
            for line in log_lines(dir_name, file_type)) #4

...
if __name__ == '__main__':                  :
    for each_line in log_lines('/var/log/apache2', '.log'):
        print each_line
        print parse_apache(each_line)

    print sum((each_line['size']))          #5
    for each_line in                   #5
        apache_lines('/var/log/apache2', '.log') #5
        if line.get('status', 0) == 200):       #5
#1 - Set up
#2 - Parse the apache line
#3 - Convert the parsed line into a dictionary
#4 - A generator for our line parser
#5 - Using our new generator

```

Before we get into the functions proper, it's a good idea to set up some of the variables that we'll need for our regular expressions **(1)**. `apache_log_headers` is a list of names for all of the fields that we'll see in our log file and `log_format` is the regular expression string that we looked at earlier. We also 'compile' `log_format` into `log_regexp` so that our matching is faster when we're parsing the log line.

First we set up a function which is responsible for parsing a single line **(2)**. Here's where we use the compiled regular expression object against the line which we've been passed, using the `match` method. If it matches, `log_split` will be a match object, and we can call the `.groups()` method to extract the parts which we matched with brackets. If there's no match, `log_split` will be `None`, which means that we have a line which is probably illegal. There's not much that we can do in this case, so we'll just return an empty dictionary.

If our function is going to be widely useful, we'll need to easily access different parts of the log line. The easiest way to do that is to put all of the fields into a dictionary, so that we can type `line['user_agent']` to access the user agent string. A fast way to do that is by using Python's builtin `zip` function, which joins the fields together with our list of headers. It creates a sequence of tuples (identical to the results of an `.items()` call on a dictionary) and then we can cast that to a dictionary with the `dict()` function. Finally, we turn some of the results into integers to make them easier to deal with later on.

Now that we have our line parsing function, we can add a generator to call it for each line of our log file **(4)**.

If we have more information about what's in the line, we can search for more detail in our logs **(5)**. Here we're adding up the total size of the requests, but only where the request is successful (a status code of 200). You could also do things like exclude the Google, MSN and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Yahoo spiders to get 'real' web traffic, see what volume of traffic is referred to you by Google, or add up individual IP addresses to get an idea of how many unique visitors you have.

When you run the program in Listing 7.16 you should see a list of lines and their parsed representation, with a number at the end. That's the number of bytes which were transferred in successful transactions. Our program is complete, and you can start adding to it if there are particular features that you'd like to add.

7.5 Functional programming

As you become more familiar with programming, you'll start to find that certain features are more or less error prone than others. For example, if your program makes use of a lot of shared state or global variables, then you might find that a lot of your errors will tend to be around managing that state and tracking down why that `!$%@%` function is replacing all your values with `None`.

So it makes sense to try and find ways to design your program which don't involve error prone features, and which are clearer and easier to understand. In turn, you'll be able to write larger, more featureful programs, and write them faster than you could before.

One of those strategies is called functional programming. Its main criteria is that it uses functions which have no side effects – their output is entirely determined by their input, and they don't modify anything outside the function. If a function is written like this, it makes it much easier to reason about and test, since you only need to consider the function by itself, not anything else.

Another feature of functional programming is that functions are objects in their own right – you can pass them as arguments to other functions, and store them in variables. This might not seem particularly important, but it enables a lot of functionality that would otherwise be quite difficult to implement.

7.5.1 Side effects

Side effects refer to anything that a function does which is outside its sphere of control, or which doesn't relate to the value it returns. Modifying a global variable, or one that's been passed in, writing to a file or posting values to a URL are all examples of side effects. Functions should also only rely on values that are passed in, not on anything outside the function.

NEXT MORNING...

BY NOW EMAILS
SHOULD BE ALL
SHREDDED AND IN
BAGS



7.5.2 Map and filter

Once you know that functions are safe to run, and aren't going to do anything weird, you can use them much more frequently – and for situations where you might not normally use functions.

Two common examples are `map` and `filter`. `map` takes a function and an iterable object, like a list or generator, and returns a list with the result of that function applied to each item in the iterable. `filter`, on the other hand, takes an iterable function and returns just those items for which the function returns `True`.

In the case of our log files, you might have code like this:

```
errors = map(extract_error, filter(is_error, log_file.readlines()))
```

Where `extract_error` pulls the error text from a log line, and `is_error` tells you whether the line is an error line. The result will be a new list of the error messages from your log file, and the original list will be untouched.

However, in practice `map` and `filter` tend to make your programs less readable than just using something like a list comprehension:

```
errors = [extract_error(line) for line in log_file.readlines()
          if is_error(line)]
```

A better use of functional programming is to use functions to change the behavior of other functions and classes. A good example is the use of decorators to change how functions and methods behave.

7.5.3 Passing and returning functions

Decorators are essentially wrappers around other functions. They take a function as an argument, potentially with other arguments, and return another function to call in its place. To use a decorator, you place its name above the function that you're decorating, preceded by an @ symbol and any arguments that you need afterwards - just like a function.

A real world example is Django's `user_passes_test` function, used to create decorators like `login_required`. `login_required` checks to see whether the user is logged in, and then returns either the regular web page if they are (Django calls them a view), or redirects them to the site's login page if they aren't. It's fairly complex, but it uses most of the functional programming techniques that we've described so far, plus a few others. I think you're ready to handle it, and we'll take it step by step.

Listing 7.17 - Django's user_passes_test decorator

```
from functools import wraps #3

def user_passes_test(test_func, login_url=None, #1
                     redirect_field_name=REDIRECT_FIELD_NAME): #1

    def decorator(view_func): #2
        @wraps(view_func, #3
               assigned=available_attrs(view_func)) #3
        def _wrapped_view(request, *args, **kwargs): #3
            if test_func(request.user): #4
                return view_func(request, #4
                                  *args, **kwargs) #4
            ...
            from django.contrib.auth.views \ #5
                 import redirect_to_login #5
            return redirect_to_login() #5
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        path, login_url, redirect_field_name)      #5
    return _wrapped_view
return decorator                                #1

def login_required(function=None,                  #6
                   redirect_field_name=REDIRECT_FIELD_NAME, #6
                   login_url=None):                      #6
    actual_decorator = user_passes_test(          #6
        lambda u: u.is_authenticated(),           #6
        login_url=login_url,                     #6
        redirect_field_name=redirect_field_name)  #6
)
if function:                                     #6
    return actual_decorator(function)            #6
return actual_decorator                          #6

@login_required                                    #7
def top_secret_view(request, bunker_id, document_id): #7
    ...
    ...
@login_required(login_url="/super_secret/login") #7
def super_top_secret_view(request, bunker_id, document_id): #7
    ...
#1 - user_passes_test returns a decorator
#2 - the decorator function
#3 - functools
#4 - if the user is logged in
#5 - if the user isn't logged in
#6 - the login_required decorator
#7 - using the decorator

```

The first thing to notice is that `user_passes_test` isn't a decorator itself, it's actually a function which returns a function for you to use as a decorator (**1**). This is a common trick if you need a few similar functions - just pass in the bits which are different and have the function return something which you can use.

(**2**) is the actual decorator itself. Remember, all it has to do is return another function to use in place of `view_func`.

If you're planning on writing a few decorators, it's worth looking into `functools` (**3**), a Python module which provides functional programming related classes and functions. `wrap` makes sure that the original meta information, such as the docstring and function name, are preserved in the final decorator. Notice also that we're using `*args` and `**kwargs` in our function, so that the request's arguments can be passed through to the real view.

(**4**) is the first part of our test. If `test_func` returns `True`, then the user is logged in, and the decorator just returns the results of calling the real view with the same arguments and keyword arguments.

If they're not logged in (**5**), then we return a redirect



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

to the login page instead. Note that I've snipped out some extra code just above here which figures out path based on some Django internals, but that's not necessary to understand how the decorator works.

Next we define the actual decorator (6). We call `user_passes_test` with the relevant arguments, and get back a function which we can use in place of the real view. They're also using `lambda`, which is a Python keyword that you can use to define small, one line functions. If your function is much more complex than this though, it's usually better to define a separate function, so that you can name it and make it clearer.

Python will use the function returned by `login_required` in place of the real view (7), so our `top_secret_view` function will first check to make sure that the user is logged in before it returns any secret documents from one of our bunkers. You can also include arguments if you want the decorator to behave differently, in this case by redirecting to a separate login system at `/super_secret/login`.

The emphasis in most programming is on objects and how they interact, but there's still a place for well written functional programs. Anywhere that you need some extra configuration, have common functionality that can be extracted or need to wrap something (without wanting the overhead of a whole class), you can use functional programming.

7.6 Where to from here?

From here you can extend your log parsing script to capture different sorts of traffic. You could categorize log entries by type (visitor, logged in user, search engine 'bot'), which section of your site they use or what time of day they arrive. It's also possible to track individuals by IP address as they use your site, to work out how people make use of your site, or what they're looking for.

You can use Python's generators in other types of programs too. If you were reading information from web pages rather than log files, you could still use the same strategies to help you reduce the amount of complexity in your code, or the number of downloads that you needed. Any program which needs to reduce the amount of data that it has to read in, or which needs to call something repeatedly but still maintain state can benefit from using generators.

You should also keep an eye out for areas in your programs which might benefit from using some of the advanced functionality that we looked at in this chapter. The secret is that when you use it, it should make your programs simpler to understand, by hiding the difficult or repetitive parts in a module or function. When you're writing an application in Django, you just need to include `@login_required` above each view that you want protected - you don't need to explicitly check the request's user or redirect to a login page yourself.

7.7 Summary

In this mini chapter, you learned about the more advanced Python features, like generators and decorators, and how you can modify classes' behavior and bend them to your will.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

You saw how to alter the way that a classes' methods are looked up, catch a missing method, even swap out the normal method lookups and use your own criteria. You saw how to transparently swap out attributes for functions by using properties, and make your class behave like an integer, list or dictionary by defining special methods.

We also looked at how to use generators to help you organize the data in your programs, and how they can reduce the memory required in your program by only loading data as it's needed, rather than ahead of time in one big chunk. We looked at how to link generators together to help write more complicated programs, using an example where we parsed information from an Apache log file. We also covered some of the regular expression module, when we needed a good way to match or extract information from some text.

Finally, we looked at functional programming, and saw how Python supports it with `map` and `filter`, as well as having functions that can be assigned to a variable. Then we looked at decorators and how they work in practice, by defining and returning different functions.

We've covered most of Python's features, so from here on, we're going to take a slightly different tack and look at some common libraries that are used with Python. The next chapter we're going to take a look at Django, the main web framework used with Python.

8

Django!

In Chapter 3 we looked at building a simple todo list to help us track what we were working on. Now we're going to look at expanding our application and making it available through a web browser, so that we can see what we need to do next regardless of where we are (as long as we have an internet connection, obviously). To make our lives easier, we're going to use a web framework called Django.

What's a web-framework, you ask? When you're developing for the web, there are a lot of details that you need to keep track of. As well as displaying the HTML and handling form input, there are lots of extra bits and pieces, like:

- handling cookies, sessions and logins
- detecting errors and displaying them
- storing data in a database
- separating your page design from the rest of the application (so that your web designers can design the pages without having to bother you)

And so on. With a framework like Django you can make use of code to handle all of these things and get your web application built more quickly.

8.1 Writing web-based applications with Django

Django is the main Python web framework, and has a large developer following. It's not the only Python framework by a long stretch, but it's the most commonly used and best documented one. It mostly follows the model-view-controller style of programming, but sometimes bends it a little. There's a lot of built in functionality to make your life much easier when developing web applications.

Table 8.1 - Django-ese to MVC-ese

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Django	MVC	Purpose
Model	Model	Stores data
Template	View	Presents a user interface
View	Controller	Does 'stuff'

Model-view-controller

Model-view-controller (MVC) is a method of design which separates out your data from how it's presented. The Model stores your data and has functions for manipulating it. The View presents your data and a user interface to the end user, and the Controller does everything in between.

It's often used as a catch-all term, but different people, applications and web frameworks will interpret and use it in different ways, so it's hard to give a definition which will work in every situation. See figure 8.0 for the differences between Django's terminology and 'classic' Model-View-Controller. The most important thing to take away is that it separates your data, the presentation of that data, and the business logic 'glue' that exists in between.

8.1.1 *Installing Django*

Django is straightforward to install, since we already have Python. Just download the latest release from <http://www.djangoproject.com/download/>, decompress it and run `python setup.py install` from within the `install` directory. On Linux and Mac OS X you'll need to prefix it with `sudo`, and on Windows you'll need to run it from a command shell with administrator privileges. Once the install process has finished, just type `import django` from within Python to make sure that it's working. You shouldn't see any errors.

If you run into any issues, or if you think you might need to install a more complicated setup (if you want to run a PostgreSQL or MySQL database server, or use a separate web server such as Apache or Nginx) then there are more detailed installation instructions available at <http://docs.djangoproject.com/en/dev/topics/install/>.

a long time ago, in a meeting room far, far away, Greg and Pitr battle the Dark Lord of Sleep...

8.1.2 *Setting up Django*

Now that we've installed Django, we can start work on our project. Django sets most things up for us, all we need to do is to plug our code into the right places and change a few

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

settings. Pick a directory on your computer where you'd like to store and run your project, and type the code in Listing 8.1 into the command line. This should work as-is under Linux, but for Windows you'll need a few extra steps.

EXTRA STEPS FOR WINDOWS

If you're running under Windows, you'll need to do two extra things. The first is to add C:\Python26\Scripts to your PATH environment variable, like we did in chapter 1, and then restart your terminal for django-admin to work.

The second step is to include your project path in a second environment variable called PYTHONPATH. This variable lets you add extra paths for Python to check when importing modules. Django's settings are imported as a Python module, so Python needs to know where to find them.

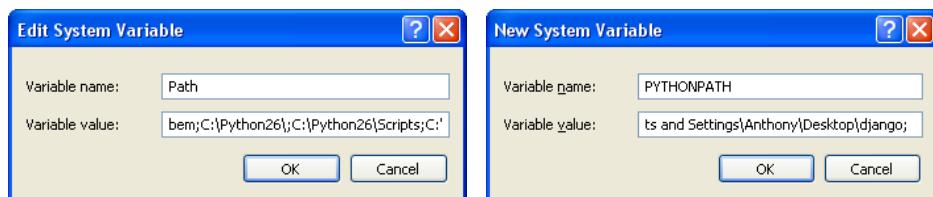


Figure 8.1 - Setting system paths for Django

Once you've made the necessary adjustments, you're ready to go!

Listing 8.1 - Django first run

```

anthony:~$ django-admin startproject todos          #1
anthony:~$ cd todos/
anthony:~/todos$ ls
__init__.py manage.py settings.py urls.py        #2
anthony:~/todos$ python manage.py runserver      #3
Validating models...
0 errors found

Django version 1.0.2 final, using settings 'todos.settings'
Development server is running at http://127.0.0.1:8000/    #4
Quit the server with CONTROL-C.                         #5
[06/Jun/2009 22:59:35] "GET / HTTP/1.1" 200 2051       #6

#1 - Start a project
#2 - A basic Django project
#3 - Start up Django
#4 - Server location
#5 - The server won't quit
#6 - The first request

```

Most of the setup and later interaction with your server is done through `django-admin` (1), which is a script that comes included with Django.

Once we've created our project, you can take a look around and see what Django's created (2). There's not much to see right now, but we'll be building on it as we write our application.

While we're at it, let's start up Django and see what happens. From within the `todos` folder, type `python manage.py runserver` (3).

When the server starts, it'll tell you what IP address and port number it's running on (4). If you'd like to run on a different IP address or port, just specify it when you run `manage.py`. For example, `python manage.py runserver 0.0.0.0:80` will connect to every interface on your PC on port 80, so you can impress your friends by allowing them to connect to the fancy new web server running on your computer.

Django will keep running until you stop it manually (5). `runserver` is the development server, which will automatically detect changes to your files and restart if necessary, so in most cases you don't even need to restart to refresh your application.

As requests come in, Django will print log lines out so that you can see what it's doing (6) - this can be useful for debugging.

If you go to the url listed in the output above, `http://127.0.0.1:8000/`, you should see something like figure 8.2.

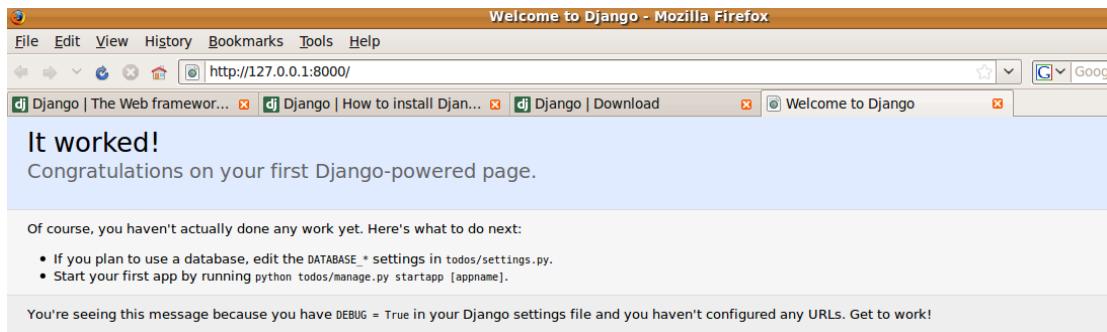


Figure 8.2 - Django's starting screen

If you look carefully, you'll see that Django even tells you what steps to take next, but we'll ignore the database part for now and just install an application. An application is where we'll do all of our work, and the project will set up and coordinate between our applications.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

You can leave the development server running - it'll automatically detect and reimport most changes that we make.

Projects vs. Applications The Django developers encourage you to reuse code, and one of the ways that they do that is to split your project up into applications. Ideally you'll have a number of applications within any one project, which all contribute their own part. You might have one application for storing your todos, another for email handling, a third for Paypal sign ups, and so on. The next time that you create a site with Django, you'll be able to reuse some of these applications to help you build it.

Now we can add an application and create our first simple page. This will make sure that everything's working, and give us our first taste of creating a Django application. There are two files that you need to edit - `urls.py` and `todo/views.py`.

Listing 8.2 - First steps

```

anthony:~/todos$ python manage.py startapp todo          #1
anthony:~/todos$ ls                                     #1
__init__.py __init__.pyc manage.py settings.py          #1
settings.pyc todo urls.py urls.pyc                      #1

todo/views.py:
# Create your views here.                                #2
from django.http import HttpResponseRedirect               #3
def hello_world(request):                               #4
    return HttpResponseRedirect("Hello world!")          #4

urls.py:
...
import todo.views                                         #5

urlpatterns = patterns('',
    (r'^.*$', todo.views.hello_world),                   #5
    ...
)
#1 - Create an app
#2 - Placeholder/signpost
#3 - Django's HttpResponseRedirect
#4 - Our view function
#5 - Use our simple view for everything

```

The first step is to start your application with `manage.py` **(1)**. Django will create a `todo` folder to store all of your application-specific code.

Next we create our view. Django helpfully puts a comment in so that we know we're in the right place **(2)**.

When we create a page to send back to the requester, there's a lot of detail involved, setting mime-types, status codes and so on. Django's `HttpResponse` takes care of all that **(3)**, and stores all the possible variables in one place if we need to alter them.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

(4) is a function which Django can call to display a web page. We're not doing anything fancy for now - just returning a response of "Hello World!".

`urls.py` is where we tie everything together **(5)** - you could think of it as Django's traffic controller, making sure that each request goes to the right place. The `patterns` function takes a number of tuples containing regular expressions and functions. If a regexp matches, Django will call the corresponding function and display whatever it returns. For now we'll use `.*`, which will match everything.

TIP Save time by keeping a directory somewhere with everything that you need to set up a new project - in our case the Django installer, our initial "Hello World" setup and anything that we find as we go which is useful. It makes setting up your next project much faster.

If you refresh your page now, you should see the not-quite-so-helpful "Hello world!" message, something like Figure 8.3.

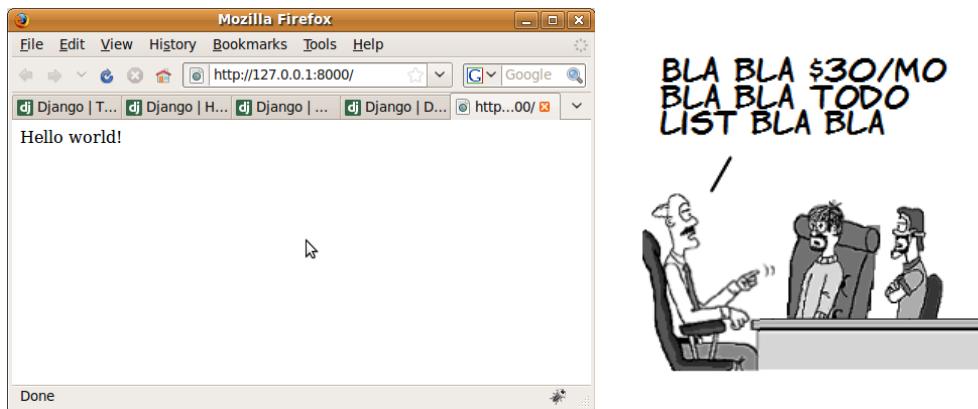


Figure 8.3 - Hello World!

It's not much, but it's *our* Hello World. Now that we know a bit more about Django and how it works, let's start working on our todo list!

8.2 Writing our application

Since we have a pretty good idea of what our todo application should look like, we'll start with the front view and add back end functionality as we need it, or else create a simple stand-in. As we need it, we'll gradually introduce more and more of Django's functionality.

Development Strategy That's only one way of building a program with Django. You can use other methods, such as creating the models that you'll be working with first, or the

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

business logic that you'll need to control everything. It all depends on your application, where the technical risks lie and what makes the most sense.

8.2.1 The simplest possible todo list

Here's our new todo list! It's very simple, but we can already use it to keep track of our todos. You just need to return the HTML for your todo page from your view, instead of the simple "Hello world!". Listing 8.3 shows you how:

Listing 8.3 - A simple todo list

```
def hello_world(request):
    return HttpResponseRedirect('' <html>
<head>
<title>My Todo list!</title>
</head>
<body>
<h1>Todos:</h1>
<p>Mow the lawn</p>
<p>Backup your PC</p>
<p>Buy some Milk</p>
</body>
</html>''' )
```

If you need a new todo, just add it into your view.

When you've finished something, just delete it. Perhaps that's not very useful, and it's annoying having to enter all of the HTML by hand, but bear in mind that this is what Django does for us with any web page that we ask it for. We just need a better way of generating our HTML markup.

I'VE HEARD GOOD THINGS ABOUT THIS "RUBY ON TOP OF RAILS" THING TOO. YOU SHOULD DEFINITELY LOOK INTO THAT.



8.2.2 Using a template

Django and most other web frameworks solve this problem by using a template. Rather than typing all of the HTML up front, you can use a simple programming language to generate it from variables and lists that you have access to. Listing 8.4 gives a simple example.

Listing 8.4 - Using a template

```
from django.template import Context, loader
from django.http import HttpResponseRedirect

def hello_world(request):
    todos = [ {'title': 'Mow the lawn', #1
              'importance': 'Minor'}, #1
              {'title': 'Backup your PC', #1
              'importance': 'High'}, #1
              {'title': 'Buy some Milk', #1
              'importance': 'Medium'} ] #1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

t = loader.get_template('index tmpl')           #2
c = Context({                                #3
    'todos': todos,                          #3
})
return HttpResponseRedirect(t.render(c))       #4

#1 - Our data
#2 - Get a template
#3 - Create a context
#4 - Return a response

```

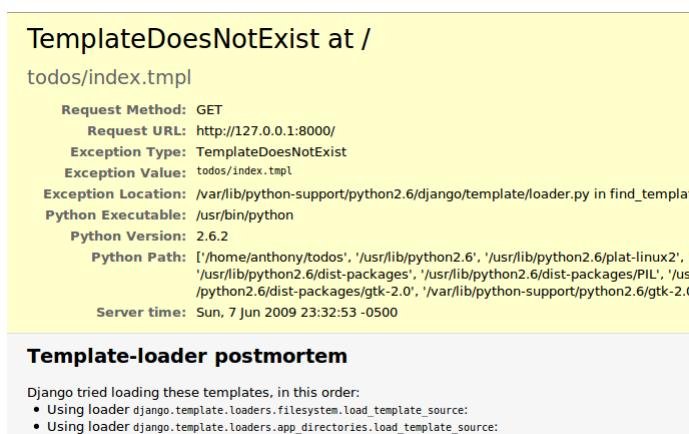
Here's our todo list. Nothing fancy for the moment, just remembering to mow the lawn, back up our PC and buy some milk (**1**).

The template handles the display of our page (**2**), all the fiddly HTML bits. We'll see what a template looks like and how to create it in a minute.

A context is a way of passing in variables to a template (**3**). In this case, we're only really interested in our list of todos, so that's all the template needs to know about.

(**4**) is where all of the actual work gets done. We call our templates `.render()` method with a particular context, then create an `HttpResponse` object with it and send it back.

If you run this code as-is though, you'll get a `TemplateNotFound` error, since we haven't told Django about the `index tmpl` template.



Template-loader postmortem

Django tried loading these templates, in this order:

- Using loader `django.template.loaders.filesystem.load_template_source`:
- Using loader `django.template.loaders.app_directories.load_template_source`:

Figure 8.4 - Where's my template?

You'll need to create your `index tmpl` template, either within your `todos` application or within a separate template directory. My version of the template is in Listing 8.5:

Listing 8.5 - A simple template

```

<html>
<head>
<title>My Todo List</title>
<style type="text/css">

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

body { font-family: Arial, Helvetica, sans-serif;
       color: black;
       background: #ffffff; }
</style>
</head>
<body>
{%- if todos %}                                     #2
<table border="1">
<tr><td>Todo</td><td>Importance</td></tr>
{%- for todo in todos %}                         #3
    <tr><td>{{todo.title}}</td>
        <td>{{todo.importance}}</td></tr>
{%- endfor %}                                    #3
</table>
{%- else %}                                      #2
<p>You have nothing to do!</p>
{%- endif %}                                    #4

</table>
</body>
</html>
#1 - HTML template
#2 - if ... else
#3 - for loops
#4 - endfor and endif

```

Our Django template is really just an HTML page (**1**), but with a few special commands to create some extra HTML on the fly. If you're familiar with HTML, templates won't be too much of a stretch.

Our first bit of Python code is an if statement (**2**). Python within the templates is wrapped within either `{% %}` brackets for program code, or `{{ }}` brackets for variables. The variables are sourced from the context that the template is passed when it's rendered by your application.

You can also use for loops in your templates, to loop over a list or iterator (**3**)

Normal Python programs rely on indentation to tell where a for loop or if statement begins and ends, but that's impossible when embedding code in a HTML template. To get around this problem, you need to include explicit end tags when closing your if statements or for loops (**4**).

Last of all, we need to edit the `settings.py` file within our todos project so that Django knows where to find the templates for our todo application.

Listing 8.6 - editing settings.py

```
...
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates"

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

BUT WE DON'T
KNOW RUBY... OR
RAILS!



```

# or "C:/www.djangoproject/templates".
# Always use forward slashes, even on Windows.
# Don't forget to use absolute paths, not relative paths.
'./home/anthony/todos/todo', #1
)
#1 - Template directory

```

Here's where our todo templates are stored (1). Django will search subfolders, so if you find the application directory getting cluttered, you can store your templates within a subdirectory such as todo/templates/.

Now if you refresh the page within your browser, you should see a nicely formatted table listing the tasks that we have to do. If you think of another task, then just add another item to the todos dictionary and the template will take care of the rest.

But we still have a similar problem - we've just gone from editing our HTML directly to editing a dictionary directly. Separating the presentation and data is an improvement though, since now we're able to store our tasks in a database.

8.3 Using a model/admin

Before we do that though, we need to start a database, tell Django where it is and populate it with our initial data. That's quite a lot of work, but Django will do most of it for us. We're using the built-in SQLite database, which is fine for our needs, but if you're writing a larger application you might want a more industrial strength database, like MySQL or PostgreSQL.

8.3.1 Setting up the database

First of all, we need to edit our `settings.py` file to tell Django what database file to use. Open it up in your favorite editor and change the database lines to look like this:

```
DATABASE_ENGINE = 'sqlite'
DATABASE_NAME = 'todo.db'
```

The rest of the database lines you can leave as they are. There are some other settings to do with admin emails, and time zones which you can edit if you want to, but they're not necessary straight away. The one exception is that your timezone needs to be set to the same one as your system if you're running under Windows. See the documentation at <http://docs.djangoproject.com/en/dev/ref/settings/#time-zone> for more details, including a link to a list of valid timezone settings.

Now just type:

```
python manage.py syncdb
```

and Django will set up your database. During the process it'll also ask you to create an admin user (a good idea). If you don't set an admin user up at this point, you can do so later on by running `python manage.py createsuperuser`.



8.3.2 Creating a model

Now we're ready to create model to store our data. Since we've already done a todo application back in chapter 4, we'll build on that data structure. You'll need to open up the `models.py` file within the `todo` directory and type something like Listing 8.7.

Listing 8.7 - A todo model

```
from django.db import models #1

importance_choices = ( #3
    ('A', 'Very Important'), #3
    ('B', 'Important'), #3
    ('C', 'Medium'), #3
    ('D', 'Unimportant'), #3
)

class Todo(models.Model): #1
    title = models.CharField(max_length=200) #2
    description = models.TextField() #2
    importance = models.CharField( #3
        max_length=1, #3
        choices=importance_choices) #3

#1 - Django's model module
#2 - Creating Fields
#3 - Constraining choices
```

All of the database interaction code is stored within Django's `db` module (**1**). To declare our todos in a format which Django can understand, we create a `Todo` class as a subclass of `models.Model`.

Fields in the database (**2**) are similar to variables in Python - they store the values that we need. Django has a number of different field types, and you can even create your own.

One of the important parts of using a database is that you can restrict the values that get entered into it (**3**). This way, you can't enter nonsense information into your application - Django will catch it and refuse to add or edit your todo. We've placed `importance_choices` outside the model so that we can access it in other contexts.

Once you've created your model, you need to let Django know that it exists. We'll update your `settings.py` to tell Django about the `todo` application, and then sync your database, which tells Django to look for new models or fields and create them.

Listing 8.8 - Adding the todo application to your project

```
settings.py:
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'todos.todo', #1
)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```
Command line:
anthony:~/todos$ python manage.py syncdb          #2
Creating table todo_todo                         #2
anthony:~/todos$

#1 - Add our application
#2 - Generate database tables
```

(1) is the line that we need to add. These lines will be converted to import lines by Django, so to be safe you'll generally want to include the project name and the application.

Once you've done that, you can use `manage.py` to create your tables (2).

If you have some database experience and want to look at what Django's doing behind the scenes, you can use `manage.py`'s `sql` command to inspect your models.

Listing 8.9 - Showing your model's SQL

```
anthony:~/todos$ python manage.py sql todo
BEGIN;
CREATE TABLE "todo_todo" (
    "id" integer NOT NULL PRIMARY KEY,
    "title" varchar(200) NOT NULL,
    "description" text NOT NULL,
    "importance" varchar(1) NOT NULL
)
;
COMMIT;
```

So we have a database, now all we need to do is add in some data so that we can test our application. Django has an easy way to do that too.

NOTE When picking a framework like Django, one of the things to look out for is how many time saving libraries it offers. Most web frameworks offer features like model-view-controller and routing URLs to views, so it's the extra bits like the admin module which will make your life easier.

8.3.3 Django's admin module

One of Django's strengths is it's built in admin system, which will let you view and edit your data without having to write a lot of data handling and checking yourself. We just need to make a few changes to `settings.py` and `urls.py`, and sync our database and we'll be ready to define an admin interface.

Listing 8.10 - Activating Django's admin system

```
settings.py:
INSTALLED_APPS =
...
'django.contrib.sites',
'django.contrib.admin',           #1
'todos.todo',
)
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

urls.py:
from django.contrib import admin
admin.autodiscover() #2 #2

urlpatterns = patterns('',
    (r'^admin/(.*)', admin.site.root), #3
    ...
)
#1 - Add the contrib.admin application
#2 - Autodiscover admin interfaces
#3 - Pick a URL for the admin app

```

All of the admin functionality is stored in the admin application, and it needs to create some database tables, so we include `django.contrib.admin` as an application (**1**).

The autodiscover function looks through the admin interfaces that you've written and automatically generates the configuration that Django needs (**2**). You'll also find that the admin functionality is already added to your `urls.py` - you just need to remove the comment characters from the start of the relevant lines.

The convention is for `^admin/(.*)`, but you can add whatever path you'd like if you'd prefer to keep things secret (**3**). Anything that matches the path gets sent off to the admin module's root function.

Now all you need to do is to sync your db again, using `python manage.py syncdb`, and Django will create the admin tables and indexes that it needs to function. Then visit `http://127.0.0.1:8000/admin/`, and you should see the login page for your server. Use the username and password that you entered earlier, and you should be able to get access to the admin page.



Figure 8.5 - Logging into Django's admin system

We're in, and there are some website-looking things, but where do we edit our todos? First we need to register our model with the Django admin interface, so that it will know to include the model, which fields to display, and so on.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

8.3.4 Adding an admin interface

Django gives you a lot of flexibility in how your admin interfaces are designed, but for now we're going to keep things simple. Listing 8.11 gives you the bare minimum that you need to be able to see your todos in the admin interface. There's not a lot - just import todo and admin, and register the Todo class as something that should be in the admin interface. Django will take care of the rest.

Listing 8.11 - Registering our model - admin.py

```
admin.py:
from todos.todo.models import Todo
from django.contrib import admin
admin.site.register(Todo)

model.py:
class Todo(object):
    ...
    def __unicode__(self):
        return self.title
```

The only change that you'll need to make to your model is to add a `__unicode__` method. Without this, Django won't know how to refer to any particular todo, and you'll just get the model name: `Todo`. You'll need to restart the server for the `admin.autodiscover` function to pick up your admin changes, but once you do you should see a `Todos` link appear in the interface. If you click on `Todos` and then 'Add todo', you'll see something like Figure 8.5. Also notice that if you don't enter a value, or enter something that doesn't fit in your model, Django will notice and give you an error.

Figure 8.6 - Editing a todo in the Django admin interface

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

You can go ahead now and enter some todo items for yourself, then go back and have a look at the Todos page. You'll see that Django's admin page gives you a table of all of the todos, and you can click through and edit any that you need to.

The admin interface is also easy to customize if you need to show or hide some of the fields, or sort by a certain field. Let's add a column to show us the importance of each todo.

```
class TodoAdmin(admin.ModelAdmin):
    list_display = ['title', 'importance'] #1
    search_fields = ['title', 'description']
admin.site.register(Todo, TodoAdmin)
#1 - A custom admin class
```

Normally, the Django admin will create this class for us automatically, based on some simple defaults. To create our own custom version, we just subclass `ModelAdmin` and override the parts that we'd like to change, such as which items to display in the list view.

There are many other attributes which can affect the admin display. For example, we can include searching of title and description by adding this one line.

Now when we register our class, we include the custom admin class instead of letting Django pick its own one.

If you restart your server and look at the todo list now, you'll see a second column. When we created our list of priorities earlier, you might remember that it used the values A, B, C and D. The secret reason for that is that you can sort your todos according to priority by clicking at the top of the column.

Alright, I think our admin interface is done. Of course, you don't normally want to give everyone access to the admin interface, so we'll see how to provide a front end suitable for general consumption in the next section.

8.4 Making use of our data

Now that we have data to work with, we should provide an interface so that other people can see what we're up to. It's often a good idea to have tangible output from your programs as soon as possible - that way you can see what needs to be done, and people can give you feedback and ideas as early as possible.

8.4.1 Using the model

Let's first find out how we can get at the data in our database and make use of it. We'll update our previous view so that it makes use of the database instead of our dictionary.



Listing 8.12 - Altering our view

```

...
from todos.todo.models import Todo                      #1

def todo_index(request):
    todos = Todo.objects.all().order_by(                 #2
        'importance', 'title')
    t = loader.get_template('index tmpl')                #3
    c = Context({                                       #3
        'todos': todos,
    })
    return HttpResponseRedirect(t.render(c))           #3

#1 - Import our model
#2 - Search our models
#3 - Nothing else changes

```

Our model class is the main interface between our view and the database, so we need to import it **(1)**.

(2) is we're looking for all of our todos - nothing too fancy yet. `Todo.objects.all()` will return a `QuerySet` object which contains all of our Todos, which we can then order with the `.order_by` method. There are other `QuerySet` methods which will help you to search through your model - a short list is in table 8.1 below.

Other than that, we don't need to make any changes to our view **(3)** - Django's templates are smart enough to adjust when we feed it a set of objects instead of a list of dictionaries.

Table 8.1 - Some common Django Queryset methods

Method	Description
<code>.all()[0]</code>	A <code>QuerySet</code> object won't trigger a query until it absolutely has to, so you can use slices like these to filter just the first few results in a query.
<code>.filter(criteria)</code> <code>.exclude(criteria)</code> <code>.get(criteria)</code>	<code>.filter()</code> , <code>.exclude()</code> and <code>.get()</code> will return results according to criteria that you specify as keyword arguments.
<code>.get(id__exact=14)</code> <code>.filter(importance__lte='B')</code> <code>.exclude(</code> <code>title__contains=</code> <code>'[blocked]')</code>	The keywords are specified as: <code><field>__<type of match></code> and Django will convert them to the relevant SQL. You can also chain <code>QuerySets</code> together to further restrict the results that you return.

There's one thing that we need to change about our display though. Our priorities are shown using the underlying letter, rather than the human-readable one. Fortunately that's a simple change to the model and template, so let's do it now.

Listing 8.13 - Readable priorities

```
model.py:
class Todo(models.Model):
    ...
    def text_importance(self):
        choices = dict(importance_choices) #1
        return choices[self.importance] #1

template.py:
...
{%
    for todo in todos %}
    <tr>
        <td>{{todo.title}}</td>
        <td>{{todo.text_importance}}</td> #2
    </tr>
{%
    endfor %}
...
#1 - A convenience function
#2 - Update the template
```

The first step is to update our model, so that it can provide us with the human readable version of the priority without having to jump through too many hoops. The easy way to do that is to convert the `importance_choices` variable to a dictionary (**1**), and then use `self.importance` to access the right one.

Now just use the `text_importance` method within the template to display the importance of our todo (**2**). Notice that Django's templates are once again smart enough to do the right thing regardless of what we feed them.

That's the basic functionality that we need to display our todo items. You can, of course, extend your template to make the page look prettier, perhaps color-code the items according to how important they are and so on. The `for` and `if` elements and the methods of your models should be enough to create most of your application.

Table 8.2 shows some other template syntax elements which might be useful, but you're normally better off trying to include the functionality within the model or controller rather than the template if you need anything more complicated.



Table 8.2 - Django template syntax cheat sheet

Syntax	Usage
<pre>{% for variable in iterable %} <tr class="{% cycle 'row1' 'row2' %}"> {{ variable }} {% endfor %} {% comment %} ... {% endcomment %}</pre>	You've already seen the <code>for</code> loop in action. One handy extra though is <code>cycle</code> - it'll swap between the values that you give it on each pass through the loop.
<pre>{% filter force_escape lower %} HTML-escaped lower case text. {% endfilter %} {{ variable urlencode}}</pre>	If you need to add comments to your template, this is the way to do it. Neither the <code>comment</code> tags or the code between them will appear in the final output.
	You can apply various filters to the output of the template, by either wrapping <code>filter</code> tags around what you need to escape, or else using the pipe character <code> </code> and filter names. There are many different filters such as <code>upper</code> , <code>lower</code> and <code>urlencode</code> that you can use - you can even write your own!

So now we have a way of getting data from the database out to the end user - we just need to be able to get data back again. To do that, we'll need to be able to submit forms.

8.4.2 Setting up our urls

First though, we'll need to think a little bit about how our application will be laid out. One good way which Django easily supports is called Representational State Transfer, or REST for short. In a nutshell, it's a style which you can use to represent resources on the web, and the actions which can be performed on them.

REST works very well for typical data-based applications, such as our todo application. In our case we have a number of todos, and we'd like to be able to view, add, edit or delete each individual todo. A typical REST design might look like listing 8.14.

Listing 8.14 - A RESTful url design

```
http://localhost:8080/todos          #1
http://localhost:8080/todos/add       #1
http://localhost:8080/todos/1          #2
http://localhost:8080/todos/1/edit     #3
http://localhost:8080/todos/1/delete   #3
#1 - View our todos
#2 - View a todo
#3 - Modify todos
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

If you think of the list of todos as a resource, this is the root level of our application **(1)**. Adding a todo won't be linked to a todo, so it's best to make 'add' a method of our root resource.

Once you've created a todo, viewing it just means appending its id onto the end of the url **(2)**.

Generally the default method for any resource is to view it, but when you need to edit or delete a todo you'll sometimes want to append the method like **(3)**.

NOTE One of the advantages of using a RESTful interface is that it encourages you to do one piece at a time. It's not normally a big deal if you haven't got deletion working yet - you can still test the other parts, since they're independent.

So now that we've mapped out how our application should work, how do we put it into practice? All of your URL handling is stored within `urls.py`. It's there that you can specify which urls will work, which views should handle them and also extract the ids. To help encapsulate our todo application a little better though, we're going to create our own `urls.py` within `todo` and then include it from the standard `urls.py`.

Listing 8.15 - Setting URLs and views

```
urls.py:
urlpatterns = patterns('',
    ...
    (r'^todos/', include('todos.todo.urls')),      #1
    ...
)

todo/urls.py:
from django.conf.urls.defaults import *          #2
import todo.views                                #2

urlpatterns = patterns('todo.views',
    (r'^$', 'todo_index'),                         #3
    (r'^add$', 'add_todo'),                        #4
    #4
    (r'^(\d+)$', 'view_todo'),                     #5
    (r'^(\d+)/edit$', 'edit_todo'),
    #5
    (r'^(\d+)/delete$', 'delete_todo'),
)
#1 - Add an include() line
#2 - todo/urls.py
#3 - Patterns definition
#4 - Todo urls
#5 - Individual todos
```



First, we include a separate `urls.py` file from

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

within the root `urls.py` (1). Note that the path is relative to the root of the project. You can also remove `import todo.views`.

Next, create `urls.py` within `todo` and add these two lines to it (2). The first just includes the default Django url handling functions, and the second imports our views.

Our pattern definition (3) is exactly the same function definition as we've been using in the root `urls.py`. One time saver here is that you can include the starting part of the function here, rather than call `todo.views.some_function` several times.

Next we define the urls for viewing our todo list (4), and adding a todo. Note that I've renamed our `hello_world` view to `todo_index`, which is a bit more sensible. The `include` function will also snip the previous `todos/` part off the front, so we're matching a blank url.

Finally, (5) is the urls for individual todos. There's one for viewing, one for editing, and another for deleting. Note that the url regular expression has a group defined with brackets. The number that it matches will be fed in as an extra argument to the view - you'll see how to make use of it when we look at how to handle individual todo items.

Note that in our `todo/urls.py` file we're not specifying the absolute path to any of our views, or anything outside of our area of responsibility. This will help us later, particularly if we're trying to combine several different applications, or we need to use our application somewhere else.

That's all we need to do to our urls for now. Let's get on with writing some views which can handle input from the user.

8.4.3 Submitting forms

The first thing that we'll do is create the form and view necessary to handle adding a new todo. There's not much point in writing a todo editing form if there aren't any todos to start with. We'll add it to the root page - it makes the most sense to include it there. When it's submitted, it'll go to `http://localhost:8080/todos/add`, which will take care of the rest. Listing 8.16 shows how we can add a form to our template.



Listing 8.16 - A submission form

```
index.html:
...
<p>Add a todo:<br>
<form action="add" method="POST"> #A
    Todo:<input type="text" name="title"><br>
    Importance:<select name="importance" />
    {% for value, importance in choices %} #1
        <option value="{{value}}> #1
            {{importance}}</option> #1
    {% endfor %}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        </select><br>
        <textarea name="description"></textarea>
        <input type="submit" value="Add">
    </form>
</p>
...

views.py:
from todos.todo.models import Todo, importance_choices #2

def todo_index(request):
    todos = Todo.objects.all().order_by(
        'importance', 'title')
    t = loader.get_template('index.templ')
    c = Context({
        'todos': todos,
        'choices': importance_choices, #2
    })
    return HttpResponseRedirect(t.render(c))

#A - Our form's action
#1 - Importance
#2 - Feed choices into the template

```

Don't forget that we'd like our todo application to be portable. It's tempting to use a hard coded path like /todos/add, but that would mean that we'd need to edit our template whenever we reused our application.

So that we don't have to repeatedly add separate instances of `importance_choices`, which could get out of sync, we include the version from our model **(1)**.

The choices will need to be passed through to the template from the model **(2)**.

Now if you refresh your root todos page, you should see a form at the bottom, under the list of todos.

Todo	Importance
Test Important Todo	Very Important
Test Todo	Medium
Not very important	Unimportant

Add a todo:

Todo:

Importance:

Figure 8.7 - A form to add a todo

But if you try and submit the form, you'll get an error. We haven't written our handler yet, so while Django in theory knows what to do with it, it can't find the function.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Listing 8.17 - A view to handle adding a todo

```
...
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse

...
def add_todo(request):
    t = Todo(
        title = request.POST['title'], #2
        description = request.POST['description'], #2
        importance = request.POST['importance']) #2
    t.save() #3
    return HttpResponseRedirect( #4
        reverse(todo_index)) #4

def view_todo(request, todo_id): #5
    pass #5
def edit_todo(request, todo_id): #5
    pass #5
def delete_todo(request, todo_id): #5
    pass #5
```

#1 - Our addition view

#2 - Create a new todo

#3 - Don't forget to save your new todo!

#4 - Redirect to our todo_index view

#5 - Stubs for other views

Handling post requests is no different to any other view. Just define a function which takes a request parameter **(1)**.

Next we create a new Todo instance, based on the values fed in via HTTP POST **(2)**. There aren't any restrictions on what we can enter, so we just feed the parameters straight in.

Once you've created the todo, it's `.save()` method writes it out to the database **(3)**.

We're finished, so we return to the index page with `HttpResponseRedirect` **(4)**. So that we're not hard coding the url with something like `/todos/`, we use the `reverse()` function, which takes either a view or the name of a view and returns its url.

The `reverse()` function doesn't like having unimplemented views, so we'll add some now. They're all related to a particular todo, so we make sure that the id for the todo is included **(5)**.

That should be all you need to do. Now if you enter a todo in the form and submit it, you should see it appear in the list. Congratulations! You now have a functioning web application. You can display data to the user from a database and accept input back, which your application can use to make additions to that data.

GREG - GREAT NEWS!
WE'VE JUST SIGNED
A DEAL WITH A MAJOR
PET FOOD
DISTRIBUTOR! SO
WE'LL NEED TO
KNOW WHAT SORTS
OF PETS OUR
CUSTOMERS HAVE...



Security in DJANGO If you've had some previous experience in web development, you're probably gritting your teeth over the code above. Normally, blindly accepting input from the people using your site is a major security hole, leading to SQL injection and XSS (cross site scripting) attacks. However, Django's database layer and templating layer will automatically escape any input and data being displayed, unless you tell it otherwise.

Of course, pretty much any web framework, or even a CGI application, will let you do show data and accept requests. The nice thing about Django is that you can do it with only a small amount of simple, straightforward code, and build more advanced things on it easily.

We still need to take care of the individual todos though - editing and deleting are just as important as being able to create new ones. We'll also look at some other ways that we can make our development even easier, using some of Django's more advanced functionality.

8.4.4 Handling individual todos

One of the main advantages in using a web framework is that a lot of the simple, boilerplate stuff is already written for you. In Django's case, two of the most useful parts are generic views and model forms.

- Generic views are implementations of common types of views, for example displaying a list of items from a particular model in the database, and all of the editing, updating and deleting associated with it.
- Model forms are forms which are built directly from your model. Since they know the structure and type of your data, they can take care of the parsing and sanitizing of form data automatically, making it very easy to get information back from the user of your application.

We'll mainly be using generic views for our application, but we'll touch on some of the generic forms. Since our models and views are already written, we'll start with the urls for our application and move on to templates in a minute.

Listing 8.18 - Updating our urls.py

```
from django.conf.urls.defaults import *
from django.views.generic.create_update \
    import update_object, delete_object          #1

import views
from models import Todo

urlpatterns = patterns('',                                #2
    (r'^$', views.todo_index),                      #2
    (r'^add$', views.add_todo),                      #2

    (r'^^(?P<object_id>\d+)$',                    #4
        update_object,                            #3
        {'model': Todo,                          #3
         'template_name': 'templates/todo_form.html', #3
         'post_save_redirect': '/todos/%(id)s',      #3
    )
)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

}),
#3

(r'^(?P<object_id>\d+)/delete$',          #5
 delete_object,                            #5
 {'model': Todo,                           #5
  'template_name':                      #5
   'templates/todo_confirm_delete.html', #5
  'post_delete_redirect': '...',        #5
 }),                                     #5
)
#1 - Import generic views
#2 - Define patterns
#3 - Generic update
#4 - Match the object id
#5 - Generic delete

```

`django.views.generic.create_update` contains the two views that we need to use **(1)**. There are other views within `create_update`, but these are the only two that we need.

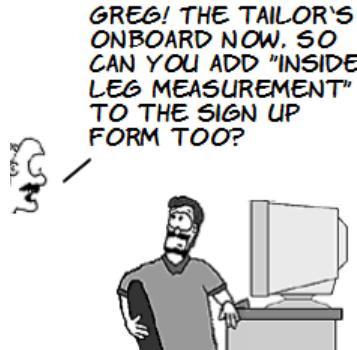
Now that we're using generic views, the sources of our views differ and we can't use the '`todo.views`' shortcut. Instead we'll just feed in the functions directly **(2)**.

(3) is a more advanced way of linking to a view, broken into multiple lines to make it easier to follow. The first line is the url regular expression as usual, and the second is the view function. The third line onwards is a dictionary of arguments to the view, which I've also put one per line. The only mandatory one is `model`, but I'm overriding `template_name` too (the default is `todo/todo_form.html`). The final parameter, `post_save_redirect` tells Django where to go next. `%{id}`s is interpreted in the context of whatever object we're editing, so it'll evaluate to `/todos/1` if you're editing a `Todo` model with id 1.

If you read **(3)** above and wondered how the view knew which object to edit, the `?P<object_id>` part here is how it's done **(4)**. If you have a named match like this in your url, Django will add it to the dictionary of arguments which is passed to the view, so we don't need to specify it explicitly. I normally name all of the parameters in my urls, since if you don't they'll be fed in as arguments, and might be in the wrong order.

Finally, **(5)** is the generic delete function. It's pretty much the same as the update one, except that once we've deleted our `todo` we won't be able to redirect back to it, so we just jump back one directory to our index page with the `post_delete_redirect` argument.

That's all that we need to do within our urls. Now we just need to add two templates - one for updating and the other for confirming deletion. These have basically the same HTML as our previous index template, so I've omitted everything that's the same.



Listing 8.19 - A Todo editing template

```
...
<title>Edit Todo #{{object.id}}</title>                      #1
...
<form action="" method="POST">                                #2
  <table>
    <tr><td valign="top">
      {{ form.title.label_tag }}                                #3
      <td>{{ form.title }}                                #3
    <tr><td valign="top">
      {{ form.importance.label_tag }}                         #3
      <td>{{ form.importance }}                            #3
    <tr><td valign="top">
      {{ form.description.label_tag }}                        #3
      <td>{{ form.description }}                           #3
    <tr><td colspan="2">
      <input type="submit" value="Save">
    </table>
  </form>
<p><a href=".">Return to todo list</a></p>                  #4
...
#1 - Our todo is called object
#2 - Form action
#3 - We have a form that we can use
#4 - Redirect back to the index page
```

Our generic edit template is fed two variables automatically. The first one is 'object' (**1**), which is the object that we're editing - in this case our todo.

The view handles both the display and editing, so we don't need to do anything with the form's action attribute (**2**) - our input will just be passed to the current url.

The second variable that we're given is `form`, which is one of Django's `ModelForm` objects that we touched on at the start of this section (**3**). It has fields which match the ones defined in your model, along with a `label_tag` attribute so that you don't have to repeat the field names in your template. The `ModelForm` field will automatically output the right input element for the fields: text for the title, a dropdown for the importance and a `textarea` element for the description.

If we don't want to edit a todo, or we've finished editing, we can use this link to go back to the main index page (**4**). The `1` in `todos/1` doesn't count as a directory, so we just want to go to our current directory, which is `".."`.

Let's see what our edit form looks like. For now you'll need to type the url to your todo manually. `http://localhost:8080/todos/1` should look like figure 8.7 below.

The screenshot shows a Django application's edit view for a todo item. The form includes fields for 'Title' (containing 'Test Todo'), 'Importance' (set to 'Medium'), and 'Description' (containing 'blah blah blah'). A 'Save' button is located at the bottom left, and a link to 'Return to todo list' is at the bottom right.

Figure 8.8 - Our edit view

If you enter new values for the todo and hit save, they should be saved to the database. You might want to have a separate window open with the index page, and refresh when you save, just to double check.

Last but not least, we'll need to be able to delete a todo once we're done with it.

Listing 8.20 - A deletion template

```
...
<form action="" method="POST"> #1
    <p>Are you sure you want to delete
        todo #{{object.id}}:
        "{{object.title}}"?<br>
    <input type="submit" value="Delete!"><br> #1
    <a href="#">Back...</a> #2
</form>
...
#1 - POST vs GET
#2 - Return to the index page
```

It's a good idea to force destructive behavior, such as editing or deleting a todo, to be done with a POST rather than a GET request (**1**), so that if someone accidentally browses to that page, or Google tries to index your site, nothing is damaged. Django follows this behavior - typing `http://localhost:8000/todos/1/delete` into the address bar of your browser, is a GET request, so Django will prompt you to confirm your action via a POST.

We might not want to delete the todo (perhaps we clicked the wrong thing or mistyped) so we include a way to go back to the index page (**2**). `/todos/1/delete` is one level deeper than `/todos/1` since the `/1/` is counted as a directory, so this back link goes one directory up.

Now if you go to `http://localhost:8000/todos/1/delete` you should be prompted to delete todo #1. If you click the "Delete!" button, your todo should be deleted from the system.

8.5 Final polishing

We're almost done with adding, editing and deleting - the last thing that we need to do now is tie it all together and make it easy to click through to edit and delete an entry. We'll improve a few other things in the interface while we're at it.

Listing 8.21 - Editing our index page

```
index tmpl:
...
<table border="1">
<tr><td>del.</td> #1
    <td>Todo</td>
    <td>Importance</td>
    <td>Description</td></tr> #1
{%
  for todo in todos %
}
<tr> #2
    <td><a href="{{todo.id}}/delete"><b>X</b></td>
    <td><a href="{{todo.id}}"/>{{todo.title}}</a></td> #2
    <td>{{todo.text_importance}}</td>
    <td>{{todo.short_description}}</td> #3
</tr>
{%
  endfor %
}
</table>
...

model.py:
def short_description(self): #4
    return self.description.split('\n')[0][:80] #4

#1 - Extra columns
#2 - Links to edit and delete
#3 - A shorter description
#4 - and it's method
```

We've added two extra columns to our listing (1). The first is for a link to delete the todo, and the second is a snippet of the description. We also added links to the edit and delete functionality that we've just written (2).

A description might be several lines long and mess up our beautiful page, so I've included `todo.short_description` (3) instead, which is a convenience method (4) that returns at most 80 characters from the first line of the description.

Now we can add, edit and delete our todos just by clicking through from the index page. Polish isn't so important while you're developing your application, but once you start to use it you'll appreciate the extra effort that you put in to make your application more usable.

OK GUYS - WE NEED
TO FIND OUT WHY
NOBODY'S SIGNING UP
FOR OUR TODO LIST
APPLICATION!



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

8.6 Where to from here?

Our application is pretty much feature complete, although the design of the front page is just bare "programmer HTML" and might need a little work. You can also add more functionality from here. Some ideas:

- Color code todos based on their importance.
- Include some javascript to sort the columns by clicking the header of the table.
- Allow todos to be assigned to a group. Of course, you'll need to be able to add and remove groups, and you'll want to create a foreign key link from the todo model to a particular group.
- Assign optional due dates to the todos, and sort by those instead of the importance.

We're not done with Django yet. In chapter 11, we extend the todo application further, allowing your friends to log in and create their own todo lists. We also look at some of Django's more advanced functionality, such as built-in unit testing, and look at some advanced database manipulation.

8.7 Summary

Now you can create your own web applications in Python using Django. We've covered all of the basics in this chapter, including designing the urls for your site and setting up views, models and templates.

We also touched on several design issues when writing web applications, such as separating the design and data models from each other, limiting destructive edits to POST requests, and some simple design strategies. These suggestions are based on common practices and experience - working 'with the grain' like this can save you lots of time and energy fighting with your development environment.

Let's take a break from the web for now and try our hand at writing a desktop application. In this case we'll be using a graphics library called Pyglet to create our own arcade game.

9

Gaming with Pyglet

In this chapter, we'll be writing our very own arcade game using a library called Pyglet. Pyglet bills itself as a "cross-platform windowing and multimedia library for Python", but we'll be using it for its *real* purpose - writing games!

If you're familiar with various arcade games, ours will be sort of a cross between SpaceWar, Asteroids and Space Invaders - it'll have a spaceship, evil aliens to shoot, and a planet to run into. To make our game more interesting, we'll give the planet some gravity, so that it draws the ship gradually.

First though, we'll need to get Pyglet installed and working on our computer.

9.1 *Installing Pyglet*

The first thing that we'll need to do is to download and install Pyglet. There's a Windows installer and source code available from <http://www.pyglet.org/download.html>, and pyglet is available as a package for several Linux distributions. Pyglet installation is straightforward under Windows - just download the installer program and run it. Mac users can download a .dmg image with an installer on it.

OpenGL Pyglet uses OpenGL under the hood, so you'll need an OpenGL capable graphics card. This normally isn't a problem unless you're running a very old computer - most cards released in the past five years or so support OpenGL automatically.

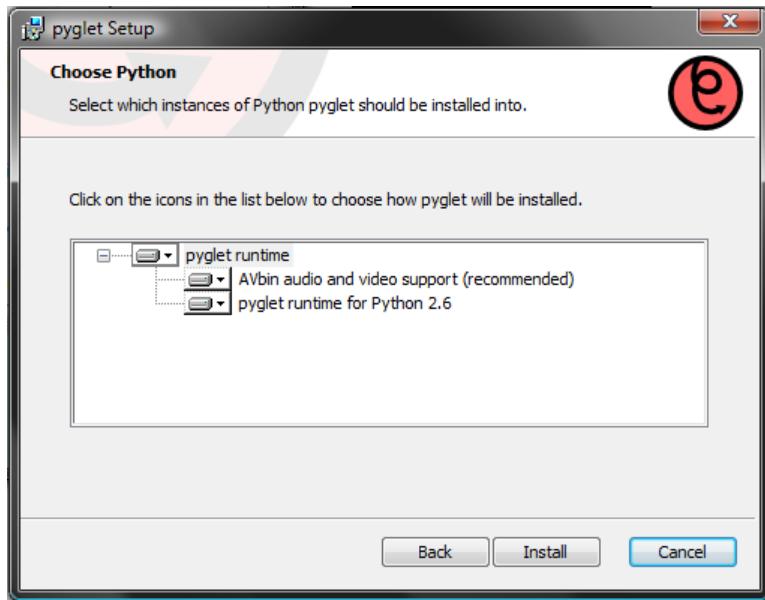


Figure 9.1 - Installing Pyglet

If none of those options work for you, you can always download the source package and run `python setup.py install`, though you'll also need to install AVBin separately if you take this route.

Let's start with a simple Pyglet program, breaking it down line by line:

```
import pyglet
window = pyglet.window.Window(fullscreen=True)
pyglet.app.run()
```

All of the submodules of pyglet are stored within the `pyglet` module. You can access the `window` module, for example, with `pyglet.window`. This saves you importing several modules at the top of your program, and makes your code easier to read.

Pyglet's `Window` object handles all of the screen initialization and rendering. You'll generally need one in every Pyglet application that you write. We're passing in `fullscreen=True` as an argument so that the window takes up the whole screen.

Pyglet is a framework, so after you've set everything up you just need to call its main application loop.

If you type this program in and run it, you should see a screen like Figure 9.2.

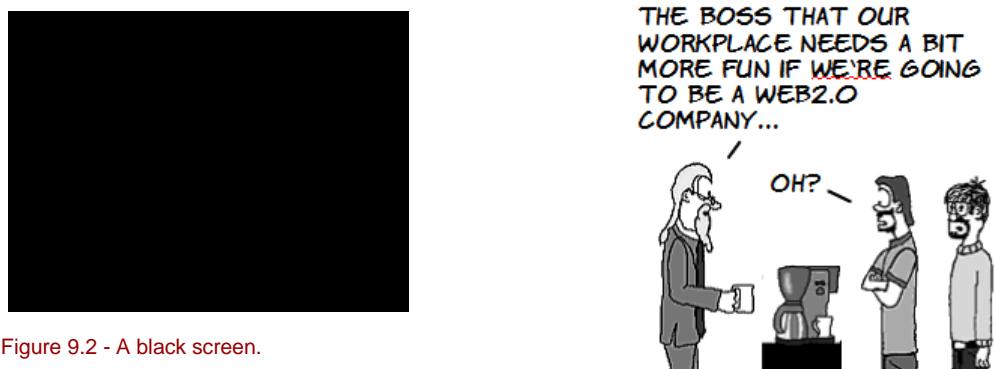


Figure 9.2 - A black screen.

That's right - a big black screen. Not very impressive, but it's *our* black screen; the blank canvas on which we'll write our masterpiece. As an added bonus, you know that Pyglet's working properly. To exit Pyglet, just hit the Escape key.

Next, we'll figure out how to make that black screen more impressive.

9.2 First Steps

Let's get started! The first thing that we'd like to do is display an image on the screen. Since we're writing a space game, let's make it a nice big planet. I've used an image of Mars that I downloaded from NASA's website at <http://www.nasa.gov/multimedia/imagegallery/>, but feel free to create your own if you're feeling artistic.

Listing 9.1 - Drawing on the screen

```
import pyglet

window = pyglet.window.Window(fullscreen=True)

pyglet.resource.path.append('./images')                      #1
pyglet.resource.reindex()                                     #1

def center_anchor(img):                                     #2
    img.anchor_x = img.width // 2                            #2
    img.anchor_y = img.height // 2                           #2

planet_image = pyglet.resource.image('mars.png')            #2
center_anchor(planet_image)                                 #2

class Planet(pyglet.sprite.Sprite):                          #3
    def __init__(self, image, x=0, y=0, batch=None):       #3
        super(Planet, self).__init__(#3
            image, x, y, batch=batch)                       #3
        self.x = x                                         #3
        self.y = y                                         #3

    center_x = int(window.width/2)                         #3
    center_y = int(window.height/2)                        #3
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

planet = Planet(planet_image, center_x, center_y, None) #3

@window.event
def on_draw():
    window.clear() #4
    planet.draw() #4

pyglet.app.run()
#1 - Image resource folder
#2 - Load the planet image
#3 - Create a sprite class
#4 - Handle image drawing

```

Before we display images, we need to tell Pyglet where to find them. To do that, we append the path to an images folder onto Pyglet's resource path **(1)**, and ask it to reindex its resources. You'll also need to create the folder manually, and save your planet image within it.

Once you have your image source, all you need to do is call the `pyglet.resource.image` function, which will read the image from your resource directory **(2)**. By default, an image will have an anchor at the lower left corner, but we'd prefer it in the center. I've created a function which will do that for you.

Pyglet is capable of drawing images directly to the screen, but a faster and cleaner way is to use a Sprite class **(3)**. Sprites track their position and image, and have their own optimised drawing routines, which make your program run faster. We'll create one instance of our planet, and stick it right in the middle of the screen. One thing to note is that we're calling `super(Planet, self)` to get the parent class of our sprite - so that we don't have to worry about manually updating it.

TIP Games are an area where a class-based design often makes a lot of sense, since there are usually a number of entities which behave similarly.

Once we've created our sprite, we just need to tell Pyglet to draw it every frame. To do this, we create an `on_draw` event handler **(4)** for the window (we'll cover event handlers in more detail in the next section). We'll do more later, but for now we just clear the screen and draw our planet.

Now you should see a nice big planet in the middle of your screen.

SO HE AGREED TO
INSTALL AN ARCADE
TABLE.

WHAT? REALLY?



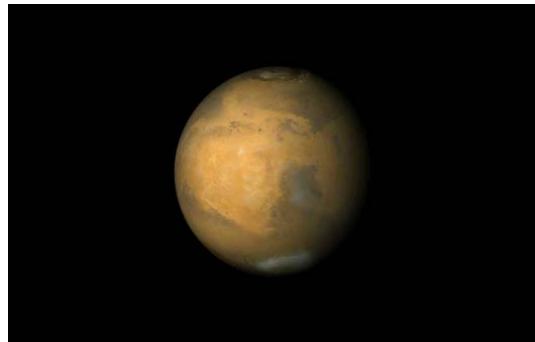


Figure 9.3 - Our planet. Ideal for running into with your spaceship! (Image courtesy of NASA/JPL/Malin Space Science Systems)

The planet will be a hazard for our spaceship, but first we need a spaceship. Let's do that part next. In the process, we'll introduce some important concepts when writing games, or any event-based program.

9.3 Starship piloting 101

Our ship follows much of the same process as the planet, with one main exception - it will move around the screen in response to the player pressing keys. If you've ever played Asteroids you'll be familiar with the control method that we'll use. The up arrow will fire your engines, and left and right will turn your ship. If you want to slow down or go backwards, you need to turn your ship around completely and fire your engines in the opposite direction.

Listing 9.2 shows the start of our Ship class. We'll be adding features to it through the rest of this section.

**ALL THE COOL STARTUPS
ARE HAVING THEM. I
PERSONALLY WAS SPACE
INVADER CHAMPION AT
EVIL TECHNICAL COLLEGE
IN MOSCOW.**



Listing 9.2 - Our Ship class

```

ship_image = pyglet.resource.image('ship.png')           #1
center_anchor(ship_image)                            #1
...
class Ship(pyglet.sprite.Sprite):                      #2
    def __init__(self, image, x=0, y=0,
                 dx=0, dy=0, rotv=0, batch=None):      #2
        super(Ship, self).__init__(                #2
            image, x, y, batch=batch)               #2
        self.x = x                                #2
        self.y = y                                #2
        self.dx = dx                             #2
        self.dy = dy                             #2

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

    self.rotation = rotv                      #2
    self.thrust = 200.0                        #2
    self.rot_spd = 100.0                       #2

...
ship = Ship(ship_image,                      #3
            x=center_x + 300, y=center_y,      #3
            dx=0, dy=150, rot=-90)             #3

...
@window.event
def on_draw():                                #4
    window.clear()                            #4
    planet.draw()                            #4
    ship.draw()                             #4

#1 - Load the ship image
#2 - Our Sprite class
#3 - Create a Ship instance
#4 - Don't forget to draw the ship!

```

First we load the image for our ship (**1**), in the same way that we did for the planet. Our Ship class looks similar to Planet, but we have some extra information (**2**) - .dx and .dy are the ship's speed in the x and y directions, and .rotation is how far left or right we've turned. I've also put in .thrust and .rot_spd to determine how fast the ship should accelerate and turn. The higher these numbers are, the faster the ship will go.

Now we can create an instance of our ship (**3**). We've fed in the ship's speed here as dx and dy, but it won't have any effect until we start updating the ship's position in the next section.

Once you have your ship, you can add `ship.draw()` to the `on_draw` event handler, and your ship will appear on the screen (**4**).

Now we can see where our ship will start, and what it looks like.

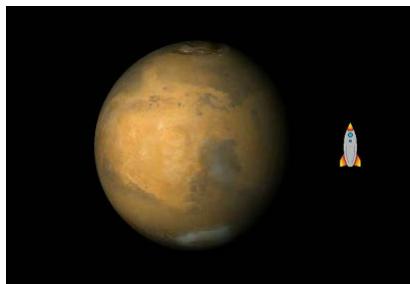


Figure 9.4 - Our space ship

So far it's no different to the planet that we're drawing, but now that we've set up our sprite we can start making it do things.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

9.3.1 Making things happen

In most games you have control over some aspect such as the main character, and can give input to tell them what to do next. Push the left arrow and move left, push the right arrow and move right. In this section we'll see how games accomplish this.

Pyglet uses an *event based* programming model, and it's how most interactive programs like games and graphical user interfaces are written. Rather than checking or waiting for input at certain sections of your program, we instead register functions to be called when something interesting happens. Pyglet refers to these functions as *event handlers*. If you're used to a standard imperative design ("do this, then this..."), an event based structure can seem odd, but it's a much cleaner way of writing some sorts of programs.

**WELL, WE'LL SEE. I'LL
GIVE YOU A FEW DAYS TO
PRACTICE BEFORE I KICK
YOU TOO HARD...**



Listing 9.3 - Handling events

```
from pyglet.window import key
...
@window.event
def on_key_press(symbol, modifiers):
    if symbol == key.LEFT:
        ship.rot_left = True
    if symbol == key.RIGHT:
        ship.rot_right = True
    if symbol == key.UP:
        ship.engines = True

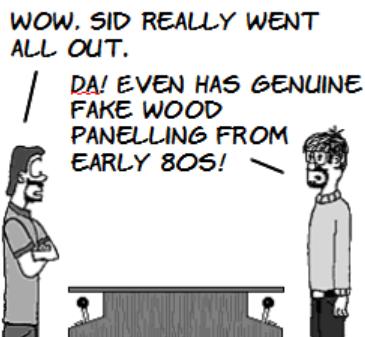
@window.event
def on_key_release(symbol, modifiers):
    if symbol == key.LEFT:
        ship.rot_left = False
    if symbol == key.RIGHT:
        ship.rot_right = False
    if symbol == key.UP:
        ship.engines = False

#1 - Define an event handler
#2 - Key symbols
#3 - Update our Sprite
```

To respond to keys, Pyglet defines two event handlers, `on_key_press` and `on_key_release` (1). They're defined in much the same way that the `on_draw` function is, but they have two arguments - the key which is pressed, and any extra keys which are held down, such as shift or control.

The symbol argument is actually an integer, but Pyglet defines a large number of keys which you can use (2), without having to worry about how to represent non-printable keys like left arrow or the escape key. To use them, just import key from pyglet.window.

If we have arrow keys pressed, then we need to make some change to the game's state. In this case, they correspond directly to the ship, so we'll just make a change to the ship's state, and let the ship handle the changes during its update method (3).



TIP Events are a powerful technique which make your programs simpler and easier to write. The alternative is to write one big loop, which checks everything in your game. And of course, it has to run as quickly as possible, otherwise your game will be laggy and unplayable.

Once you've done that, pressing the arrow keys will trigger an `on_key_press` event, and update your ship's status, but you won't see anything happen on the screen. That's because we haven't told the ship how to respond to changes in its status. For that we'll need to write an update method to change the rotation of the ship according to its status.

Listing 9.4 - Updating our ship

```
class Ship(pyglet.sprite.Sprite):
    def __init__(...):
        ...
        self.rot_left = False          #1
        self.rot_right = False         #1
        self.engines = False           #1

    def update(self, dt):           #2
        if self.rot_left:            #3
            self.rotation -= self.rot_spd * dt #3
        if self.rot_right:           #3
            self.rotation += self.rot_spd * dt #3
    ...
    def update(dt):                #4
        ship.update(dt)             #4

pyglet.clock.schedule_interval(update, 1/60.0)          #5
#1 - Set our initial state
#2 - The ship's update function
#3 - Rotating our ship
#4 - A main update function
#5 - Call the update function
```

When the ship is first created, it won't be turning left or right, or firing its engines. We set the ship's state here **(1)** so that our update function later on won't throw an exception.

By convention, most Pyglet classes will have an update method which gets called on each 'tick' of the game engine **(2)**. This is where our sprites change their position, create new objects in the game and update their internal state. An update method takes one argument, `dt`, which tells us how much time has passed since the last time update was called.

We're starting out simply, so we're just rotating our ship left and right for now **(3)**. If we're turning, then we update our `.rotation` attribute (a Pyglet built-in which rotates the sprite) by multiplying our rotation speed by `dt`.

Later on we'll have other objects with update methods, so it's a good idea to collect all of the method calls in one place **(4)**.

Finally, we set Pyglet's built-in scheduler to call our main update method 60 times per second **(5)**. This is the maximum speed that Pyglet will run our game at. If it's slower then we'll get different values for `dt`, but our game will still run.

Now our feedback loop is finished, and we can see the results of all of our hard work. If you run our program you should be able to rotate your ship left and right by pushing the left and right arrow keys.

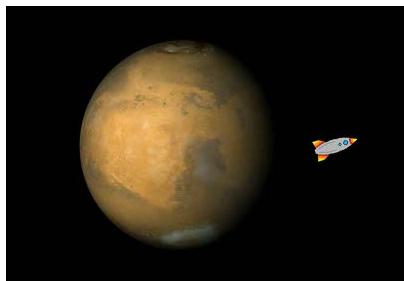


Figure 9.5 - Turning the ship



The next step now is to make our ship move. To do that properly though, we'll need to learn a bit about how to specify directions and distances.

9.3.2 Back to school - Newton's first law (and vectors)

In order to make our ship move consistently, we'll need to apply a little bit of theory. You may remember some of this from school if you did any Math or Physics. If you didn't, don't worry - we'll be taking things one step at a time. The first thing to know is that `x` represents values that go left to right, and `y` represents values that go up and down.

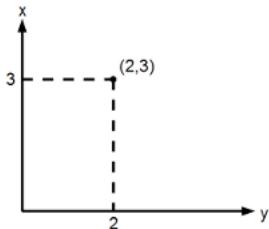


Figure 9.6 - x and y coordinates

NEWTON'S FIRST LAW

If you think back to your Physics classes, you might remember Newton's first law. Briefly, this states: "A body in motion will continue that motion unless acted on by an external force." In other words, our ship will move in a straight line unless we fire our engines. We already have a velocity - that's the `.dx` and `.dy` attributes of our `Ship` class.

VECTORS

The second thing that we need is a way to convert the angle of the ship and its acceleration into values that we can add to the ship's velocity.



Figure 9.7 - The ship's angle can have x and y parts

Whenever our ship's engines are firing, we'll need to break up its angle like this to work out the effect on our velocity in the x and y directions. The direction in figure 9.7 means that when the ship's engines fire, we'll need to add 2 to our y velocity and 3 to our x velocity.

We'll need a few math modules to do this in Python, but the principle isn't any different from Figure 9.7 - figure out the x and y parts of the acceleration, and add those to our x and y velocities. Each update, just add our velocity to our position.

Listing 9.5 - Moving the ship

```
import math #1
...
ship_image_on = pyglet.resource.image('ship_on.png') #6
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

center_anchor(ship_image_on)                                #6
...
def wrap(value, width):                                    #2
    if width == 0:                                         #2
        return 0                                           #2
    if value > width:                                      #2
        value -= width                                     #2
    if value < 0:                                         #2
        value += width                                     #2
    return value                                         #2

def to_radians(degrees):                                 #3
    return math.pi * degrees / 180.0                      #3
...
class Ship(...):
...
    def update(self, dt):                               #6
        self.image = ship_image
        if self.rot_left:
            self.rotation -= self.rot_spd * dt
        if self.rot_right:
            self.rotation += self.rot_spd * dt
        self.rotation = wrap(self.rotation, 360.)          #2

        if self.engines:
            self.image = ship_image_on
            rotation_x = math.cos(
                to_radians(self.rotation))                  #3
            rotation_y = math.sin(
                to_radians(-self.rotation))                  #3
            self.dx += self.thrust * rotation_x * dt       #4
            self.dy += self.thrust * rotation_y * dt       #4

            self.x += self.dx * dt                         #5
            self.y += self.dy * dt                         #5

            self.x = wrap(self.x, window.width)           #2
            self.y = wrap(self.y, window.height)          #2

#1 - Import Python's math module
#2 - A 'wrap' function
#3 - Find the x and y components of rotation
#4 - Alter our velocity
#5 - Update our position
#6 - One last thing

```

All of the trigonometric functions that we need are stored in Python's `.math` module, so we start by importing it (**1**).

Thinking ahead a little bit, we'll also want to be able to handle the case where the ship moves off the edge of the screen. We'll take the easy option

A FEW HOURS LATER...



and just wrap the game up and down and left to right (2). `wrap` is a function which does that - given the value and the amount you'd like it to be constrained to.

Next we break our angle down into x and y parts (3). Note that these might be negative if the angle points left or down. Also, Pyglet and the `.math` module use different representations of angles (degrees vs. radians), so we need a function to convert from Pyglet's version into something that the `math` module can use. We also need to flip our rotation around to get the right values in the y direction.

Once we have our two components, the rest is relatively straightforward. We multiply each part by the ship's acceleration and how long it's been since our last update, and add each one to our velocity (4).

The last step is to update our position on the screen (5). We also check to make sure that we can't go over the edge of the screen by wrapping our x and y positions based on the height and width of the window.

Finally, it looks a bit odd for our ship to be flying around without any visual feedback, so I created an extra image with some flame shooting out of the back. We swap it over whenever the ship's engines are on (6).



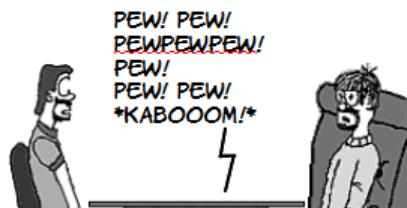
Figure 9.6 - Now we can drive our spaceship around. Brrm! Brrm!

Now you can drive your ship around the screen, accelerating and turning around to decelerate. Wheee! It's fun for a while, but ultimately there's not much to do, and the mechanics are easy to understand. What we'd like is to have something more complex, so that you have more opportunities for different sorts of interaction with the game.

9.4 Gravity

We'll add to our game by making the planet have gravity, so that it pulls on the ship. If the ship collides with our planet, then BOOM! No more ship! Fending off aliens while trying to keep clear of the planet should add enough difficulty to keep the player occupied and entertained.

How exactly do we go about doing that though?



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Well, the obvious place to put the functionality is within the `Planet` class. It makes sense since it's the planet that's affecting the ship, and if we want anything else to be pulled by the planet's gravity, it won't be too hard to add. At the core, we're just adding another force to the ship, just like we did when firing its engines.

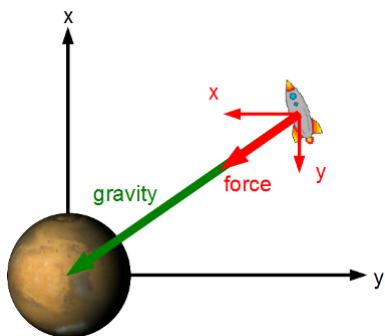


Figure 9.7 - Gravity applies a force to our ship

Figure 9.7 shows you what our problem looks like. The green line is the vector from our ship to the planet. We'd like to find that, convert it into a force vector, and then split that vector into an x and y so that we can easily add it to our ship's velocity. Let's deal with the easy bit first - splitting our force vector.

Listing 9.6 - Planet updates

```
class Planet(pygame.sprite.Sprite):
    def __init__(self, image, x=0, y=0, batch=None):
        super(Planet, self).__init__(
            image, x, y, batch=batch)
        self.x = x
        self.y = y

    def update(self, dt):
        force, angle = self.force_on(ship)
        force_x = force * math.cos(angle) * dt
        force_y = force *
            math.sin(angle) * dt
        #1
        ship.dx += force_x
        #1
        ship.dy += force_y
        #1
    ...
def update(dt):
    planet.update(dt)
    #2
    ship.update(dt)
#1 - Apply a force to our ship
```



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

#2 - Updating our planet

The first thing that we need is to find out how much gravitational force the planet will put on the ship. We'll gloss over this part for now - all we know is that we'll create a method in a minute which will tell us the magnitude and direction of the force **(1)**. Other than this, it's the same as when we updated our ship when its engines were firing.

Don't forget to include our update method **(2)** in the main update function.

Now we have a nice, well defined problem to solve - find the distance and angle to the ship. This is the opposite problem to the one that we solved earlier. Back then we had an angle and distance and wanted the x and y parts, now we have the x and y parts and want to know the angle and distance.

Listing 9.7 - Figuring out gravity

```
class Planet(...):

    self.mass = 5000000 # experiment!                      #1

    def dist_vec_to(self, target):
        dx = self.x - target.x                            #2
        dy = self.y - target.y                            #3
        sqr_distance = dx**2 + dy**2                     #4
        distance = math.sqrt(sqr_distance)                #4

        angle = math.acos(float(dx) / distance)           #5
        if dy < 0:                                         #5
            angle = 2*math.pi - angle                   #5
        return (distance, angle)                         #6

    def force_on(self, target):
        distance, angle = self.dist_vec_to(target)        #7
        return ((self.mass)/(distance**2), angle)         #7

#1 - Our planet has a mass
#2 - Which way is the ship?
#3 - Calculate x and y
#4 - Find the distance
#5 - Find the angle
#6 - Return a vector
#7 - Calculating force
```

First, set our planet's mass **(1)**. The heavier our planet is, the more it'll pull on our ship. This is one of the elements of your game which you can tweak to make it easier or harder.

The core of our method is to find out how far away the ship is, and in which direction **(2)**. Based on that, we can calculate everything else we need.

Next we find out the distance to our target (the ship) in the x and y direction **(3)**. These might end up being negative if the ship is to our left or below us - that's normal.

Now we can find the first part of what we need, which is the distance to the ship **(4)**. This is just using Pythagoras' theorem, squaring the two smaller sides and taking the square root.

The angle is a little bit trickier (5). With a horizontal and vertical distance we can use `math.acos` or `math.asin` to find the angle, but we need to take the complete 360 degree range into account. `math.acos` is only valid for the first half of the circle, so we need to add `2*math.pi` (half a circle in radians) to the angle if it's supposed to be greater than that. Figure 9.8 shows this in a little more detail - the two angles are different, even though the x distance and the direct distance are the same.

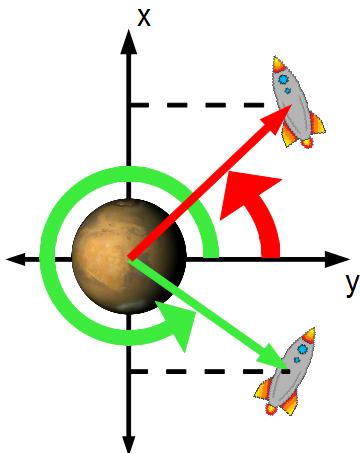


Figure 9.8 - Two different angles, same x position and distance

Once we have the distance and angle, we can just return those vectors (6). I've chosen (distance, angle) as the way a vector is represented, just so that I don't accidentally get them around the wrong way later.

NOTE If all of this math seems a little complicated, don't worry too much. You have easy to use methods for calculating vectors and forces which you can reuse in your next game.

Now that we know the distance and direction to the ship, calculating the force due to gravity is easy (7). It's proportional to the mass of the planet, and diminishes with the square of the distance. The closer the ship, the more force we apply to it.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

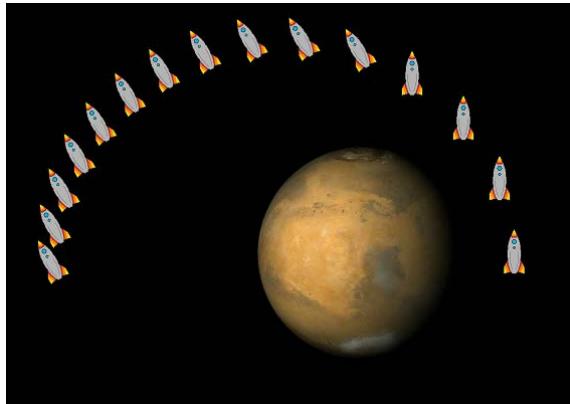


Figure 9.9 - Our ship in orbit around the planet

Now when you run your program, you should see your ship being affected by gravity! Rather than moving in a straight line, it'll have a force applied to it by the planet, and move in a graceful curve. If you're careful, you can even put your ship into an orbit around the planet.

9.4.1 ***Watch out for that planet!***

For our collision detection, we're sticking with circles around the ship, planet and alien. Circles like those in Figure 9.10 make our code simpler and more straightforward, but in trading accuracy for simplicity you might notice a few collisions which should have been near misses. It's possible to get 'pixel perfect' accuracy with Pyglet by comparing the overlap of the images themselves, but that's a little outside the scope of this chapter.

OK. ~~LET'S SEE HOW THE OLD REFLEXES ARE DOING...~~

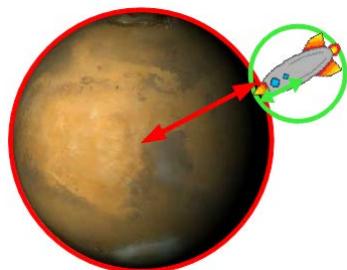


Figure 9.10 - The planet and ship's collision circles

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

In practice though, we don't need to draw circles - we can just compare the distance between the ship and the planet, and compare that to the radius of the planet and the ship.

Listing 9.8 - Crashing into the planet

```

class Planet(...):
    def __init__(...):
        ...
        self.radius = (self.image.height +
                       self.image.width) / 4 #1 #1
    ...
    def update(self, dt):
        distance, angle = self.dist_vec_to(ship) #2
        if distance <= ship.radius + self.radius: #2
            ship.reset() #3
            ship.alive = False #3
            return #3
        ...

class Ship(...):
    def __init__(...):
        ...
        self.alive = True #1 #1
        self.radius = self.image.width / 2

    def reset(self): #4
        self.life_timer = 2.0 # seconds until respawn #4
        self.x = center_x + 300; #4
        self.y = center_y #4
        self.dx = 0; self.dy = 150 #4
        self.rotation = -90 #4

    def update(self, dt):
        ...
        if not self.alive: #5
            print ("Dead! Respawn in %s" %
                  self.life_timer) #5
            self.life_timer -= dt #5
            if self.life_timer > 0: #5
                return #5
            else: #5
                self.reset() #5
                self.alive = True #5
        ...
        ship = Ship(ship_image) #4
        ship.reset() #4

    ...
    @window.event
    def on_draw():
        window.clear()
        planet.draw()
        if ship.alive: #6
            ship.draw() #6

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

- #1 - Adding attributes to our objects
- #2 - How far away is the ship?
- #3 - Crash!
- #4 - Resetting the ship
- #5 - Handling the player's death and respawn
- #6 - Dead players don't draw

We'll need a few attributes on our objects (1). One is to tell the game whether the ship is alive or not, and the others are the radius of the planet, and the ship. To make life easier, we'll calculate the radius of the ship and the planet from the size of their images. If we change the image later on, we won't need to update the object's radius.

SID, ~~YOU'VE BEEN~~
PLAYING FOR AN HOUR.
~~AND YOUR SCORE'S~~
~~ONLY 30.000!?~~



With circles to detect collisions, all we need to do is compare the distance between the ship and the planet, and the sum of their radii (2). If the distance is shorter, then the circles intersect, and we have a collision.

Once our spaceship has crashed (3), we mark the ship as dead and reset the player's position.

The ship's `.reset()` method puts the ship back at the start (4), and sets its velocity to something reasonable. We're also setting a 'life timer', which determines the time until the ship restarts, giving the player a few seconds to think about what went wrong.

To delay the ship's return, we check the `life_timer` attribute that we set in the `.reset()` method (5). If we're dead and the timer is greater than zero then we still have some time left. If it's less than 0, then we can mark the ship as alive, reset its position once more (since gravity still affects it) and we're back to normal.

The last thing that we need to do is make sure that the ship isn't drawn when it's dead (6). A simple `if` statement takes care of that.

Now that our game's starting to take shape, you can see the general form that a game takes. It has a certain state, effectively a simulation of a number of things like planets and ships, and we can have an effect on that simulation in certain ways. With some thought, a bit of luck and experimentation, your simulation will have aspects that are fun.

Next up, let's add some excitement to our game.

9.5 Guns, guns, guns!

What's a space game without aliens to shoot? Even space trading games have guns of some sort, so if we don't have any, we'll look a bit odd. They're easy to do, given the work that we've already done on angles and timers - just set the bullet travelling at high speed at the same angle as the ship, and update it in a similar way. We'll also want to keep track of whether it's run into anything, and after a certain amount of time, remove it from the game.

Listing 9.9 - Shooting

```

bullet_image = pyglet.resource.image('bullet.png')
center_anchor(bullet_image)
...
class Ship(pyglet.sprite.Sprite,
           key.KeyStateHandler): #1
    def __init__(...): #2
        self.shot_timer = 0.1 #2
        self.reload_timer = self.shot_timer #2
        self.bullets = [] #5

    def update(self, dt):
        if self[key.LEFT]: ... #1
        if self[key.RIGHT]: ... #1

        rotation_y = math.sin(to_radians(-self.rotation)) #1
        if self[key.UP]: ... #1
        ...
        if self.reload_timer > 0: #2
            self.reload_timer -= dt #2
        elif self[key.SPACE]: #3
            self.bullets.append( #3
                Bullet(self.x, self.y, #3
                        rotation_x*500+self.dx, #3
                        rotation_y*500+self.dy, #3
                        bullets))
            self.reload_timer = self.shot_timer #2

class Bullet(pyglet.sprite.Sprite): #4
    def __init__(self, x=0, y=0, dx=0, dy=0, batch=None): #4
        super(Bullet, self).__init__( #4
            bullet_image, x, y, batch=batch) #4
        self.x = x #4
        self.y = y #4
        self.dx = dx #4
        self.dy = dy #4
        self.radius = self.image.width / 2 #4
        self.timer = 5.0 #5

    def update(self, dt): #4
        self.x += self.dx * dt #4
        self.y += self.dy * dt #4
        self.x = wrap(self.x, window.width) #4
        self.y = wrap(self.y, window.height) #4

        self.timer -= dt #5
        # collide with planet, or remove after 5 seconds #5
        distance, angle = planet.dist_vec_to(self) #5
        if distance <= planet.radius or self.timer < 0: #5
            ship.bullets.remove(self) #5

    ...
bullets = pyglet.graphics.Batch() #6

@window.event
def on_draw():

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

window.clear()
planet.draw()
bullets.draw()
if ship.alive:
    ship.draw()

# Call update 60 times a second
def update(dt):
    planet.update(dt)
    ship.update(dt)
    for bullet in ship.bullets:
        bullet.update(dt) #6
    ...
window.push_handlers(ship) #1

#1 - Use a KeyStateHandler
#2 - Limit the number of shots
#3 - FIRE!
#4 - A bullet class
#5 - Deleting bullets
#6 - Handling bullet updates

```

An easier way to manage key presses is to use pyglet's `KeyStateHandler` class (**1**). This class keeps track of which keys have been pressed and makes them available with a dictionary syntax. If you push the left arrow key, then `self[key.LEFT]` will be set to True. The only tricky part to remember is that the ship instance is now a key handler, so you need to do a `window.push_handlers(ship)` to register it with Pyglet.

If we just let the player fire whenever the space key is pressed, then they'll get a bullet per frame, or 60 shots a second! Even if your computer is fast enough to handle hundreds of bullets onscreen, it makes the game a bit easy - just fill the screen with bullets until there's nowhere for the alien to hide. We'll limit the number of shots by setting a timer whenever the ship fires a bullet (**2**). Every update we'll subtract `dt` from the timer, until it's 0 and the player is ready to fire again.

Firing is straightforward - we just create an instance of the `Bullet` class going in the right direction (**3**). I'm giving the bullets a speed of 500, with the ship's velocity added in (otherwise you get weird effects when you're travelling very fast or shooting side on to the direction that you're travelling). We store the bullet instance in `ship.bullets`, since if we don't have a reference to them somewhere, Python's garbage collector will remove them, and you'll wonder why your bullets aren't appearing on the screen.

(**4**) is the class that we use whenever we fire a bullet. Bullet updates are easy, since they're not affected by gravity and move in a straight line.

**THIS BEHAVES EXACTLY
LIKE THE OLD ARCADE
MACHINES. REMEMBER?
ONCE YOU GET TO
999,999, IT'LL WRAP
AND START AGAIN
FROM ZERO.**



We don't want bullets hanging around forever, so they have their own timer too. Once they've been around for 5 seconds, we delete them from `ship.bullets` and let Python handle the rest (5). We're also checking for collisions with the planet, in the same way that we did for the ship.

Since there are potentially so many bullets, it makes sense to use Pyglet's Batch class, which makes sprite rendering much faster if you have lots of sprites to draw. To use our bullets batch, we just pass it in when creating the bullet sprite, and then call `bullets.draw()` to draw all of the bullets at once (6).

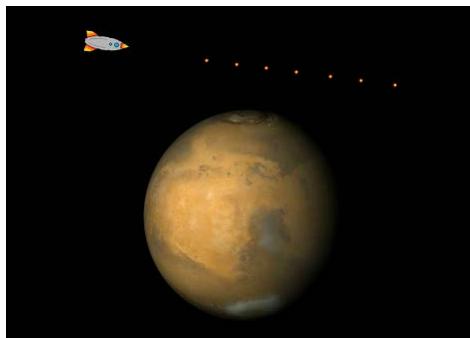


Figure 9.11 - Our ship firing - ready to take on the alien armada

Now we can fly around the galaxy, doing good deeds and destroying alien scum... hang on - we don't have any alien scum to shoot yet. Let's fix that next.

9.6 Evil aliens

What good are bullets without aliens to try them out on? In this section we'll add an alien space ship, whose sole purpose in life is to ram the evil Earthling intruder and destroy him. To make life easier on ourselves, we'll assume that he has advanced alien technology which isn't influenced by gravity, and he can enter and leave the planet's atmosphere at will. In other words, we're going to be a little bit lazy and not worry about all those vectors and collisions with the planet - just whether it's hit the player or not.



Listing 9.12 - A random alien

```
import random #3
...
alien_image = pyglet.resource.image('alien.png') #2
center_anchor(alien_image) #2
...
def make_vec((x1, y1), (x2, y2)): #1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

"""distance and angle from (x1,y1) to (x2,y2)"""
dx = x1 - x2
dy = y1 - y2
distance = math.sqrt(dx**2 + dy**2)
if distance == 0:
    return (0,0)
angle = math.acos(float(dx) / distance)
if dy < 0:
    angle = 2*math.pi - angle
return (distance, angle)

def vec_to_xy(distance, angle):
    x = distance * math.cos(angle)
    y = distance * math.sin(angle)
    return (x,y)

def dist_vec_to(source, target):
    return make_vec(
        (source.x, source.y),
        (target.x, target.y))
...

class Alien(pyglet.sprite.Sprite):
    def __init__(self, image, x=0, y=0,
                 dx=0, dy=0, batch=None):
        super(Alien, self).__init__(
            image, x, y, batch=batch)
        self.x = x
        self.y = y
        self.dx = dx
        self.dy = dy
        self.radius = self.image.width / 2
        self.life_timer = 2.0
        self.accel_spd = 200.0
        self.max_spd = 400.0
        self.alive = True

    def reset(self):
        self.alive = True
        self.life_timer = 2.0 # seconds until respawn
        self.x = random.random() * window.width
        self.y = random.random() * window.height
        self.dx = random.random() * (self.max_spd/2)
        self.dy = random.random() * (self.max_spd/2)

    def update(self, dt):
        if not self.alive:
            self.life_timer -= dt
            if self.life_timer > 0:
                return
            else:
                self.reset()

        if random.random() < 0.2:
            accel_dir = random.random() * math.pi*2
            accel_amt = random.random() * self.accel_spd
            accel_x, accel_y = vec_to_xy(accel_amt, accel_dir)

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        self.dx += accel_x          #3
        self.dy += accel_y          #3

        self.dx = min(self.dx, self.max_spd)    #4
        self.dx = max(self.dx, -self.max_spd)   #4
        self.dy = min(self.dy, self.max_spd)    #4
        self.dy = max(self.dy, -self.max_spd)   #4

        self.x += self.dx * dt           #2
        self.y += self.dy * dt           #2
        self.x = wrap(self.x, window.width) #2
        self.y = wrap(self.y, window.height) #2

...
alien = Alien(alien_image)
alien.reset()

@window.event
def on_draw():
...
    if alien.alive:
        alien.draw()

def update(dt):
...
    alien.update(dt)

#1 - Factor out vector functions
#2 - Our Alien class
#3 - Accelerate in a random direction
#4 - Don't go too fast

```

One thing that I've done in this section of code is pull the vector functions out of the class and made them more standalone **(1)**. I've left them as is here, but ultimately you'll probably want to put them into their own module, or else find a vector library which you can reuse.

Our alien class ends up being quite similar to the Ship class except for its update method **(2)**, so there shouldn't be any major surprises in this part. We have all the same concepts - speed, acceleration, wrapping the x and y position, death and respawn after a countdown.

Our alien has a very simple AI - every so often it accelerates in a random direction **(3)**. The frequency of the acceleration and the parameters which I've set in `__init__` give the alien enough changes in direction that shooting it can be a bit of a challenge.

Finally, I noticed while play testing that it's possible for the alien to accelerate to ridiculous speeds **(4)**, which makes it a bit hard to shoot. To stop it doing that, we just check that the x and y speeds are within the alien's maximum speeds and reduce them if they aren't.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

NOTE The alien is one thing in your game that you definitely want to experiment with. The rough rule of thumb is that the alien should be easy enough for the player to shoot some of the time, but hard enough to be a challenge. Without the right balance, your game won't be fun.

The last thing that we need to do is make the alien interact with the other objects that we have onscreen - it should be killed by bullets and kill the player when it runs into their ship. We'd also like some sort of accomplishment for the player, so we'll add a score. Every time the player does something wrong, like crash into the planet or the alien, we'll subtract 100 points. If they shoot the alien, then we add 100 points.

Listing 9.13 - Making the alien interact

```
class Alien(pyglet.sprite.Sprite):
    def update(self, dt):
        ...
        # check collisions with the player                      #1
        player_dist, player_angle = dist_vec_to(self, ship)      #1
        if player_dist < (ship.radius + self.radius) * 0.75:    #1
            # BANG! got the player
            self.reset()                                         #1
            self.alive = False                                    #1
            ship.reset()                                         #1
            ship.alive = False                                   #1
        ...
class Ship(pyglet.sprite.Sprite, key.KeyStateHandler):
    def __init__(...):
        ...
        self.score = 0                                         #2
    def update(self, dt):
        ...
        score.text = "Score: %d" % self.score                  #4
    if not self.alive:
        self.life_timer -= dt
        if self.life_timer > 0:
            return
        else:
            self.reset()
            self.score -= 100                                  #2
            self.alive = True
    ...
class Bullet(pyglet.sprite.Sprite):
    def update(self, dt):
        ...
        # check collision with Alien                         #3
        dist, angle = dist_vec_to(self, alien)             #3
        if dist < alien.radius:                            #3
            # hit alien
            alien.reset()                                 #3
            alien.alive = False                           #3
            ship.bullets.remove(self)                     #3
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

    ship.score += 100                      #3
    return                                #3

    ...
score = pyglet.text.Label('Speed: 0',           #4
                         font_name='Arial',      #4
                         font_size=36,          #4
                         x=10, y=10,            #4
                         anchor_x='left',       #4
                         anchor_y='bottom')     #4
score.color = (255, 255, 255)                  #4

@window.event
def on_draw():                               #4
    ...
    score.draw()                           #4
#1 - The alien should crash into the player
#2 - Keep score
#3 - We should be able to shoot the alien
#4 - Display the score on screen

```

We'd like the alien to be an extra hazard for the player to avoid, so we check the distance between the ship and the alien (**1**), the same way that we do for the ship and the planet. It's up to you whether you want the alien to disappear or not when it collides with the player.

A score (**2**) is how we let the player know that they've done something right according to the rules of the game, so we give them 100 points for shooting the alien, and subtract 100 points if they run into the alien or the planet.

The bullets should have an effect on the alien (**3**), so for each bullet we check the range to the alien. If it's within the alien's radius then we've shot the alien! Resetting the alien works in pretty much the same way that it does for the player - only draw it if it's alive, and have a short delay between the alien dying and its reappearance.

The player needs to be able to see their score on the screen (**4**), so we add a Label class in the bottom of the screen, 10 pixels from the bottom left corner. Setting the color looks a little odd, since you might be expecting three numbers for red, green and blue. The fourth is the alpha value - 255 is opaque and 0 is completely transparent - useful for fading text in and out.

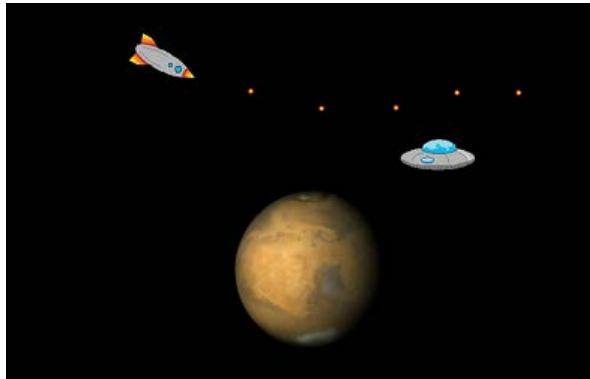


Figure 9.9 - Die, alien scum!

Now you have a full blown space-alien-shooting-get-as-many-points-as-you-can-but-don't-run-into-the-planet game. I'm sure you can think of a catchier title. You can send the game to your friends, compete on scores

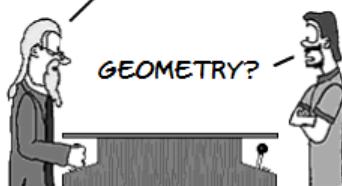
9.7 Where to from here

There are a number of improvements or changes that you can make to the game, either to refine what's already there, expand on the gameplay or turn the game into something completely different. Here are some ideas.

9.7.1 Extend the game play

There are a number of other elements that would normally be in a game like this. A good idea might be to pick your favorite space-shooting game and see how many of its features you can add. You might want to make the alien shoot back, tweak its AI to make it nastier, or just add more aliens. Adding extra, harder levels with each wave of aliens and limited lives for the player would be another extra feature. Sound effects are also good for setting an atmosphere.

WELL, IF YOU
REALLY WANT
TO KNOW -
IT'S ALL IN
THE
GEOMETRY



9.7.2 Alter the game play

Another option is to extend the game in a completely different direction - perhaps you're not a big fan of space alien destruction. If you add a second player, and make your shots affected by gravity, you'll have something pretty close to *Spacewar!*, the original space shooting game, written for the PDP-1 back in 1962.

If shooting at stuff isn't really your thing, you could add extra planets and turn the game into a space trading game, or a '3D' version of Lunar Lander. Limited fuel and lighter or

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

heavier gravity on different planets would add to the challenge of the game, in addition to trading well. Pyglet comes with several examples which you can use to add text input and other features.

9.7.3 Refactoring

Now that you understand the program, there are some areas where the code could be improved. For example, there's quite a bit of duplication in terms of the objects and how their positions are updated - making them derive from a subclass could make your code clearer and easier to extend.

You could also use an external vector class within your objects, so that you don't have to look at (or debug) all of that geometry code. Of course, it helps to know what's behind the vector library before you start.

Unit tests would be a big help in ensuring that your program is functioning properly when you make these changes. It's difficult to test visual and gameplay aspects, but you can still check that collisions are detected properly by manually placing ship, bullet and alien objects and checking whether they overlap. Other game data such as forces and velocities can be tested in the same way.

9.7.4 Get feedback

Another thing to bear in mind is that you're writing a game. You can write the most beautiful code, with all sorts of features, but all that will be for nothing if your game isn't fun to play. One good way to design and develop games is to create a minimal version which includes the elements that you think will be fun, and test it out on a few people, tweaking the various parts as necessary.

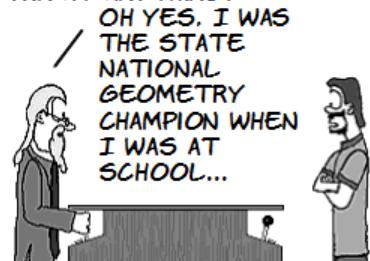
9.8 Summary

In this chapter we learned how to write our own arcade game. We used Pyglet graphics classes to display images onscreen and move them around. To make our objects move realistically, we used some geometry and physics modeling to update their positions onscreen.

We added several types of object and learned how to make them interact with each other - our ship could run into a planet, and fire bullets, then finally we added an alien which could run into the ship and be shot by bullets.

Along the way, we learned about other game elements, such as collision detection and scheduling actions to take place over time.

SURE, GAMES ARE ALL
GEOMETRY BEHIND THE
SCENES. ONCE YOU SEE
THROUGH TO THE MATHS
UNDERNEATH, YOU CAN
WIN AT ANYTHING.



Finally, we covered some aspects of game design - your game needs to be fun, and include some familiar elements to attract people. It's important to get feedback from other people - what's fun for you might not be fun for anybody else.

In the next chapter, we'll be learning more about Django, and how we can make the web applications that we write available on the internet for other people to use.

10

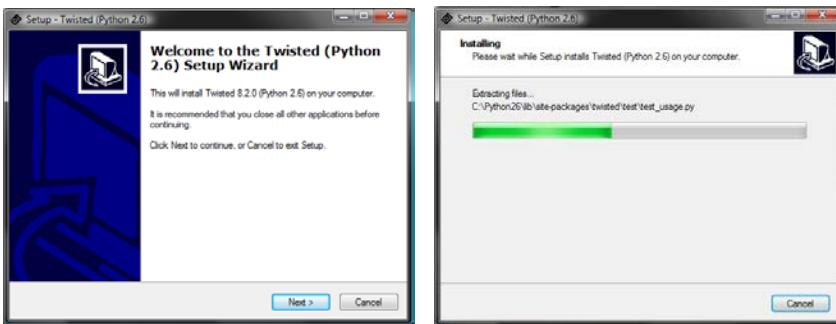
Twisted Networking

In this chapter, we'll be revisiting the adventure game that we wrote in chapter 6 and extending it so that we can log in and play it with other people via an internet connection. Normally these games are referred to as MUDs, which stands for Multi User Dungeon. Depending on the person creating them, MUDs can range from fantasy hack-and-slash to science fiction, and players can compete or cooperate to earn treasure, points or fame.

To get us started quickly, we'll use a framework called Twisted, which contains libraries for working with many different networking protocols and servers.

10.1 *Installing Twisted*

The first step is to install Twisted and get a test application running. Twisted comes with installers for Windows and Macintosh, which are available from the Twisted homepage at <http://twistedmatrix.com/>. If you're using Linux, there should be packages available through your package manager.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Figure 10.1 - Installing Twisted on Windows

The installer will pop up a window as it compiles things, but once you see the rightmost window in Figure 10.1, Twisted is installed!

The only other thing that you need is a telnet application. Most operating systems come with one built-in, and there are many free ones that you can download - I normally use PuTTY, which is available for Windows.

HEY SID - I'M TURNING
MY ADVENTURE GAME
INTO A MUD!



10.2 Our first application

We'll start by writing a simple chat server. The idea is that people will be able to log into it via a program called Telnet and send each other messages. It's a little more complex than hello world, but we can extend this program and use it in our game later on in this chapter. Open a new file and save it as something like `chat_server.py`.

Let's start with the first part of our application, the protocol for our chat server. In Twisted terminology, a protocol refers to the part of your application that handles the low level details - opening connections, receiving data and closing connections when we're finished. You can do this in Twisted by subclassing its existing networking classes.

Listing 10.1 - A simple chat server protocol

```
from twisted.conch.telnet import StatefulTelnetProtocol      #1

class ChatProtocol(StatefulTelnetProtocol):
    def connectionMade(self):                                     #1
        self.ip = self.transport.getPeer().host                  #2
        print "New connection from", self.ip                      #3
        self.msg_all(                                            #3
            "New connection from %s" % self.ip,                   #3
            sender=None)                                         #3
        self.factory.clients.append(self)                         #3

    def lineReceived(self, line):                                #4
        line = line.replace('\r', '')                           #4
        print ("Received line: %s from %s" %                  #4
               (line, self.ip))                                 #4
        self.msg_all(line, sender=self)                         #4

    def connectionLost(self, reason):                            #5
        print "Lost connection to", self.ip                     #5
        self.factory.clients.remove(self)                       #5

    def msg_all(self, message, sender):                         #6
        self.factory.sendToAll(                                #6
            message, sender=sender)                           #6

    def msg_me(self, message):                                #6
        message = message.rstrip() + '\r'                      #6
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

    self.sendLine(message) #6
#1 - ChatProtocol is like Telnet
#2 - Override connectionMade
#3 - A new connection
#4 - Handle data
#5 - Close the connection
#6 - Convenience methods

```

For our chat server we'll be using of Twisted's `StatefulTelnetProtocol` (1). It takes care of the low level line parsing code, which means that we can write our code at the level of individual lines, and not have to worry about whether we have a complete line or not.

We're customizing our protocol by overriding the builtin `connectionMade` method (2). This will get called by Twisted for each connection the first time that it's made.

We're just taking care of a bit of housekeeping here - storing the client's IP address and informing everyone who's already connected of the new connection (3). We also store the new connection so that we can send it broadcast messages in the future.



The Telnet protocol class provides the `lineReceived` method (4), which gets called whenever a complete line is ready for us to use (i.e. whenever the person at the other end hits the return key). In our chat server, all we need to do is send whatever's been typed to everyone else who's connected to the server. The only tricky thing that we need to do is to remove any line feeds, otherwise our lines will overwrite each other when we print them.

If the connection is lost for some reason - either the client disconnects, or is disconnected by us, `connectionLost` will be called so that we can tidy things up (5). In our case we don't really need to do much, just remove the client from our list of connections so that we don't send them any more messages.

To make our code easier to follow, I've created the `msg_all` and `msg_me` methods, which will send out a message to everyone and just ourselves (6). `msg_all` takes a `sender` attribute, which we can use to let people know who the message is coming from.

NOTE A Factory is a programming term for something which creates a class for you to use. It's another way to hide some of the complexity of a library from the programmers who make use of it.

So that takes care of how we want our program to behave - how do we link it in to Twisted? We use what Twisted refers to as a Factory, which is responsible for handling connections and creating new instances of `ChatProtocol` for each one. You can think of it as a switchboard operator - as people connect to your server, the Factory creates new Protocols, and links them together. Something like Figure 10.2 below.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

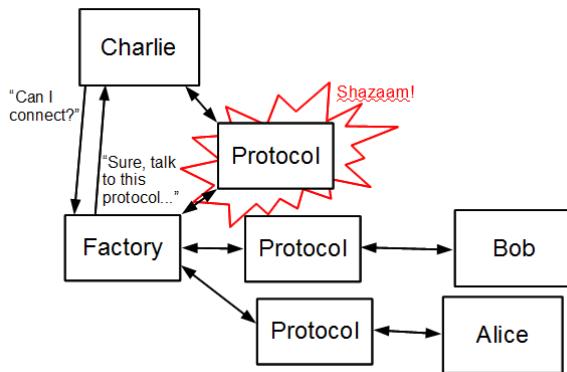


Figure 10.2 - A factory creating protocols

So how do we do this in Twisted? Easy...

Listing 10.2 - Connecting up our protocol

```

from twisted.internet.protocol import ServerFactory           #1
from twisted.internet import reactor                         #3
...
class ChatFactory(ServerFactory):                           #1
    protocol = ChatProtocol                                #1

    def __init__(self):
        self.clients = []                                    #2

    def sendToAll(self, message, sender):                  #2
        message = message.rstrip() + '\r'
        for client in self.clients:
            if sender:
                client.sendLine(                            #2
                    sender.ip + ": " + message)          #2
            else:
                client.sendLine(message)                 #2
        #2

    print "Chat server running!"                          #3
factory = ChatFactory()                                  #3
reactor.listenTCP(4242, factory)                        #3
reactor.run()                                           #3

#1 - A ChatFactory?
#2 - Talking to everyone
#3 - Wiring everything together
  
```

SID?
IT'S LIKE AN
ADVENTURE GAME,
ONLY YOU CAN PLAY IT
WITH OTHER PEOPLE
OVER THE INTERNET



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

A Factory is Object Oriented terminology for something which creates instances of another class **(1)**. In this case, it'll create instances of ChatProtocol.

The ChatFactory is the natural place to store data which is shared between all of the ChatProtocol instances. The sendToAll method is responsible for sending a message to each of the clients specified within the clients list **(2)**. As you saw in listing 10.1 above, the client protocols are responsible for updating this list whenever they connect or disconnect.

The final step is to let Twisted know about our new protocol and factory. We do this by creating an instance of ChatFactory, binding it to a particular port with the listenTCP method and then starting Twisted with a call to its main loop, reactor.run() **(3)**. I've chosen 4242 as the port to listen to - it doesn't matter too much which one you use, as long as it's something above 1024 so that it doesn't interfere with existing network applications.

If you save that program and run it, you should see the message "Chat server running!". If you connect to your computer via telnet on port 4242 (usually by typing telnet localhost 4242), then you should see something like Figure 10.3.

```
C:\> Command Prompt - python chatserver_1.py
C:\>Users\Anthony\Documents\hello_python\bitbucket\hello_python\chapter_10\src>py
thon chatserver_1.py
Chat server running!
New connection from 127.0.0.1
Received line: Hello! from 127.0.0.1
Received line: Hello! from 127.0.0.1
New connection from 127.0.0.1
Received line: from 127.0.0.1
Received line: How's it going? from 127.0.0.1
Received line: Pretty good! from 127.0.0.1

C:\> Telnet localhost
127.0.0.1:
Hello!
127.0.0.1: Hello!
127.0.0.1: New connection from 127.0.0.1
127.0.0.1: 
127.0.0.1: How's it going?
127.0.0.1: Pretty good!
127.0.0.1: Pretty good!

C:\> Telnet localhost
127.0.0.1:
How's it going?
127.0.0.1: How's it going?
127.0.0.1: Pretty good!
```

Figure 10.3 - our chat server is running

It may not seem like much, but we've already got the basic functionality of our MUD server going. If you'd like to explore the chat server further, there's a more fully featured version included with the source code, available from <http://source.manning.com/XXX>, which adds commands to change your name and see who else is connected, as well as limit some common sorts of misbehavior and allow you to kick anyone who's behaving too badly.

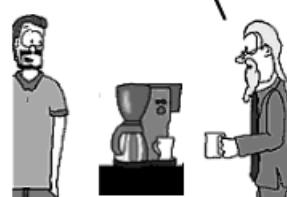
10.3 First steps with our MUD

Now we're ready to start connecting our adventure game to the network. We'll base it on our chat server, but instead of just broadcasting what's typed to everyone who's connected, we'll feed it directly into the adventure game instead. This is a common way to get things done when programming - find two programs (or functions, or libraries) which do separate parts of what you need, then 'glue' them together.

The basic gist is that we'll have multiple players all logged in at once, all trying to execute commands (like 'get sword') at the same time. This could potentially cause problems for our server since we're mixing real-time Twisted code with the one-step-at-a-time of our adventure game - we'll head off most of them by queuing up player commands, and updating our game's state once per second.

Let's get started - you'll want to copy your adventure code from chapter 6 into a new folder, along with the chat server code that we just created. You'll probably also want to rename `chat_server.py` to something like `mud_server.py`, to help keep things straight.

LIM, YEAH.... I
USED TO PLAY
THEM... A WHILE
AGO...



Listing 10.3 - Updating our Chat Protocol

```
from game import Game #1
from player import Player #1
import string #5

class MudProtocol(StatefulTelnetProtocol): #1

    def connectionMade(self): #1
        self.ip = self.transport.getPeer().host
        print "New connection from", self.ip

        self.msg_me("Welcome to the MUD server!") #2
        self.msg_me("") #2
        ...
        self.player = Player(game.start_loc) #3
        self.player.connection = self #3
        game.players.append(self.player) #3

    def connectionLost(self, reason): #4
        ...
        game.players.remove(self.player) #4
        del self.player #4

    def lineReceived(self, line): #5
        line = line.replace('\r', '')
        line = ''.join([ch for ch in line #5
                       if ch in string.printable]) #5
        self.player.input_list.insert(0, line) #5

#1 - Update imports and classes
#2 - Welcome message
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

- #3 - Create a player when connecting
- #4 - Remove players when disconnecting
- #5 - Redirect input to the player class

The first step is to import our Game and Player classes into our code (1). I've also changed the name of the protocol so that it's obvious what we're trying to write.

Next, we give a nice, friendly start when someone first connects to our MUD (2).

Now we'll start to do the real work, but it turns out to not be that hard. I've assumed that the game will keep track of its players somehow, and just added a new player object to the game's list of players (3). So that we can talk to the player from within the game, I've also added the protocol to the player. We'll see how that works in a minute.

We'll still need to handle the case where a player disconnects from the server (4), but again, it's straightforward - just remove them from the game's list of players and delete them.

Once players are connected, they'll want to type commands, like 'go north' or 'attack orc' (5). First, we sanitize the input that we've received (in testing I found that different telnet programs can send different weird characters). Once it's trimmed down to just printable characters, we assume that the player has a list of commands waiting to be executed, and just push this one on to the end.

That's our Protocol done, but what about our Factory and the rest of the bits? It turns out that we don't need to do too much to our protocol - just change a few lines.

**GREAT! I'LL EMAIL
YOU THE DETAILS.
AND YOU CAN LOG
IN AND PLAY!**



Listing 10.4 - Updating our Chat Factory

```
from twisted.internet import reactor, task #3
...
class MudFactory(ServerFactory): #1
    protocol = MudProtocol #1
    ...
    ...

game = Game() #2
# game.run() #2

def run_one_tick(): #3
    game.run_one_tick() #3

print "Prototype MUD server running!" #3
factory = MudFactory() #3
game_runner = task.LoopingCall(run_one_tick) #3
game_runner.start(1.0) #3
reactor.listenTCP(4242, factory) #3
reactor.run()
```

#1 - Update our factory

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

#2 - Create our Game instance
#3 - Run game updates

We don't need to do too much to update our factory (1) - just change its protocol and rename it.

We'll need a Game object too, so we'll create it here (2). We don't want to use the old `run` method though, since it still handles things the old way.

Our design above calls for us to run a game update once per second. Since we're using Twisted's event loop (that's the `reactor.run()` part), we'll need to use Twisted's `task.LoopingCall` to call the game's update method (3), `run_one_tick`, which we'll also create shortly.

That should be all we need to do to our network code for now. We've made a few assumptions about how our game code will work, but often this is easier than jumping back and forth between our Game and MudProtocol classes, trying to fit it all together. Now that our Protocol is written, we just have to make Game and Player play along too.

NO, THAT'S
OK, I'M...
BUSY.



Listing 10.5 - Changing our Game code

```
class Game(object):

    def __init__(self):
        ...
        #self.player = player.Player(cavel)
        self.players = []
        self.start_loc = cavel
        ...

    def run_one_tick(self):
        self.do_input()
        self.do_update()
        self.send_results()

    def send_results(self):
        for player in self.players:
            thing.send_results()

#1 - Don't create just one Player
#2 - Our game's loop
#3 - Sending results
```

The single player version of our adventure game had one player, but we'll potentially have lots, so we'll make it a list instead (1). We're also giving the starting cave a sensible name.

(2) is the main loop that we called from the networking part of our code. You should be able to follow what it's doing just from the names - get input for each Player object (including monsters), run their update and then send the results back.

We already have methods for getting input and processing orders, but we'll need something to send back the results of each player's actions (3). To do that, we'll make another assumption - that each Player object knows how to send results back to the player.

Now we only have two assumptions left to fill in, and they're both in the Player class. The first is that the Player class will have a list of pending commands, and the second is that it will have a way to send the results of any commands or events back to the player. The other thing that we need to do is to make sure that the Player class reads from the list of pending commands, rather than using `raw_input`.

Listing 10.6 - Changing our Player code

```
class Player():
    def __init__(self, location):                      #1
        ...
        self.input_list = []                           #1
        self.result = []                             #1

    def get_input(self):
        #return raw_input(self.name+">")           #2
        if self.input_list:
            return self.input_list.pop()             #2
        else:
            return ''                                #2

    def send_results(self):
        for line in self.result:                   #3
            self.connection.msg_me(line)
        for line in self.events:                  #3
            self.connection.msg_me(line)           #3

    def die(self):
        self.playing = False
        self.input = ""
        self.name = "A dead " + self.name

        for item in self.inventory:               #4
            self.location.here.append(item)
            item.location = self.location
        self.inventory = []                      #4

        self.connection.msg_me("You have died! "
                               "Better luck next time!")   #4
        self.connection.transport.loseConnection() #4

    class Monster(player.Player):
        ...
        def send_results(self):
            pass                                #5

        def get_input(self):
            if not self.playing:
                return ""
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

player_present = [x for x in self.location.here           #6
                  if x.__class__ == player.Player      #6
                  and x.playing]
#1 - Adding input and output buffers to Player
#2 - Update get_input
#3 - Add send_results methods
#4 - What happens when we die?
#5 - Stub out the send_results in Monster
#6 - There might be more than one player

```

We start by creating our list of pending commands, and the result that needs to be sent back to the player (**1**). They're just lists, and when they're in use they'll have a list of strings.

We can't use `raw_input` any more, so we need to read our next command from `self.input_list` (**2**). `pop` removes the command for us so that we don't have to worry about removing it from the list later. `pop` called on an empty list raises an exception, so we check for that case and assume that the command is blank if there's nothing there.

To send the results of a player's actions (**3**), we use the `self.connection` object which we set up in `mudserver.py` above. Note that even if the player isn't doing anything, other players and monsters will, so we have two separate sections - one for the results of your actions and another for events.

In the old version of the game, when we died the game ended. That's no longer the case, so we'll need to gracefully handle the case where a player dies (**4**). To do that, we just make them drop whatever they're carrying, send them a message and drop the connection. If you extend your game, you might want to make the player keep their items. Alternatively you can allow other players to 'get sword from anthony' if you're feeling mean.

Monsters don't connect over the network and don't have the `self.connection` object, so the default `send_results` from the `Player` class won't work. They don't need to know the results of their actions at all, so we'll just 'stub out' their version of `send_results` and return immediately (**5**).

Our previous adventure game just looked at the player's name to figure out whether to attack them or not. Now that we have multiple players, who will probably all have different names, we'll need to be a bit more discerning (**6**). A better way is to examine the class of the object that the monster is looking at, using the `__class__` method. That'll return the class, which we can compare to `player.Player`.

NOTE This worked so well because our game has only one point of communication with the player - the commands that they type and the responses that the game returns.

HEY AJ! WANT TO
SIGN UP FOR MY
MUD?

A MUD? COOL!
KINDA '80S, BUT
STILL COOL...



That should be all you need to do. Now when you run your server and connect via telnet, you'll see your familiar adventure game prompt, and can run around the server collecting loot and slaying monsters. Go ahead and bask in the adventure and glory.

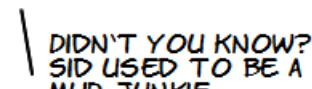
Well, sort of. While the game works, and we can explore and do everything that we need to, there are a few more things that we need to take care of before our game is playable.

10.4 Making the game more fun

I made the code above available to some of my friends online and got feedback from them. There were two major issues that they raised - the first was that the monster was too hard to beat, and the second was that there wasn't enough interaction between the players. Normally in an adventure game like this, you'll be able to change your name and description, talk to other players, look at their description, and so on.

10.4.1 Bad Monster!

The problem with combat is pretty obvious once you run into the orc for the first time. While you're limited to the actions that you type in, the monsters react at computer speed.

```
ca Telnet localhost
Welcome to the MUD server!
look
Mumpus lair
Items here:
sword
coin
Exits:
west: Winding steps
east: Dismal grotto
north: Ledgewind lake
south: Black pit
The orc misses you!
The orc misses you!
The orc hits you!
The orc misses you!
The orc has killed you!
```

Figure 10.4 - Bad monster! No beating on the player!

The solution that most MUDs use is what's known as an 'angry list'. Rather than attacking things directly, the game maintains a list of monsters and other players that you're angry at. If you're not explicitly doing anything else and there's something present that's on your angry list, then you'll attack it. If something attacks you, then they'll go on your angry list too, so you'll at least put up a token defense. Let's look at how we can implement it in our game.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Listing 10.7 - Angry lists

```

class Player(object):
    def __init__(self, game, location):
        ...
        self.angry_list = [] #1

    def update(self):
        self.result = self.process_input(self.input)

        if (self.playing and
            self.input == "" and
            self.angry_list):
            bad_guys = [x for x in self.location.here
                        if 'attack' in dir(x) and
                           x.name in self.angry_list]
            if bad_guys:
                bad_guy = random.choice(bad_guys)
                self.events += bad_guy.do_attack(self) #2

    def die(self):
        ...
        self.angry_list = [] #3

    def stop(self, player, noun): #4
        self.angry_list = [x for x in self.angry_list
                           if x.name != noun]
        return ["Stopped attacking " + noun] #4

    def attack(self, player, noun): #5
        player.angry_list.append(self.name)
        self.angry_list.append(player.name)
        result = ["You attack the " + self.name]
        result += self.do_attack(player)
        return result #5

    def do_attack(self, player): #5
        """Called when <player> is attacking us (self)"""
        hit_chance = 2 #5
        ...
        actions = [..., 'stop'] #6

#1 - Add an angry list
#2 - Attack "bad guys"
#3 - Dead players tell no tales
#4 - Stop attacking
#5 - Modify the attack method
#6 - Add stop to our list of actions

```

Both players and monsters will need a way to remember who they're angry at. We'll just make it a list (**1**), since we're not expecting it to grow too large.

Next we'll modify our update method. If our input attribute is blank, we know that the player (or monster) hasn't entered any commands, and we can go ahead and attack if necessary. All we do is build a list of all the things that we're angry at which are present, and then attack one of them (2).

If a player or monster is dead, they shouldn't keep attacking, so we clear their angry list (3).

The players will also need a way to stop attacking things (maybe they're friends again). The stop command will remove an attacker from the list of things that you're angry at (4).

The final major thing that we'll do is make the attack command modify the angry lists of both the attacker and attacked (5). Now when something gets attacked, it'll automatically fight back. Note how we build our result before we do the attack. That way if the target dies, we won't see "You attack the dead orc". do_attack is just the mechanism from our old attack attribute with a different name.

The final, final thing is to add stop to our list of commands (6), otherwise we won't be able to use it!

Now the player should have at least half a chance against the orc. If the orc beats them now, they'll at least feel like they haven't been completely robbed by the game. If you pick up the sword, you'll find that it helps a lot, which is what we want. There are plenty of other opportunities for improving the combat system, but there's a more pressing problem that we need to deal with instead.

10.4.2 Back to the chat server

The second problem is that players weren't able to interact with each other. This is often a big draw card when writing a multiplayer game - players will come for the game, but stay for the company. Fortunately, making our game more social is easy to do. We'll just add a few extra commands to the Player class.

Listing 10.8 - Social Gaming

```
help_text = """
Welcome to the MUD

This text is intended to help you play the game.
Most of the usual MUD-type commands should work, including:
"""

class Player(object):
    ...
    def help(self, player, noun):
        ...

#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
#1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

A MUD JUNKIE? COME ON. THERE'S NO SUCH THING...

NO, REALLY. HE HAD TO BE HOSPITALISED FOR A WHILE...



```

    return help_text           #1

def name_(self, player, noun):      #2
    self.name = noun               #2
    return ["You changed your name to '%s'" % self.name]   #2

def describe(self, player, noun):    #2
    self.description = noun        #2
    return ["You changed your description to '%s'" %     #2
            self.description]       #2

def look(self, player, noun):       #3
    return ["You see %s." % self.name, self.description]  #3

def say(self, player, noun):        #4
    for object in self.location.here:
        if ('events' in dir(object) and
            object != self):
            object.events.append(
                self.name + " says: " + noun)
    return ["You say: " + noun]      #4

```

#1 - Help!
#2 - Change your name and description
#3 - Look
#4 - Talking to people

If a player's completely new to the game, we need to give them at least half an idea of what they can do. We'll make 'help' output some helpful instructions **(1)**. The full help text that I added is in the source code.

Another easy win is to let players customize their appearance by changing their name and description **(2)**. Rather than being "player #4", you can now be "Grognir, Slayer of Orcs".

Of course, the description's not much good if other players can't see it **(3)**.

We'll also need to add to most important case of all - a say command, so that your players can talk to each other **(4)**. All it needs to do is send what you've typed to every other object in the current room, but it'll allow players to interact on a human level, which will in turn help keep them coming back.

One of the issues that you'll run into is that with the new commands, the old `find_handler` method will sometimes call the wrong thing. For example, both the player and the location have a `look` method, and which one is correct will depend on the context. As well as this, some of the commands that we've just added only really apply to the player themselves, and we shouldn't look for an object to apply them to. Listing 10.9 below has an updated version which is a lot more explicit about which objects it should look at.

HE'S ON SOME SORT
OF COURT-SPONSORED
NETHACK PROGRAM
NOW, I THINK...



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Listing 10.9 - Updating find_handler

```

no_noun_verbs = ['quit', 'inv', 'name_', 'describe',
                  'help', 'say', 'shout', 'go']                      #1
#1

...
def find_handler(self, verb, noun):
    if verb == 'name':                                         #2
        verb = 'name_'
    if verb == "":                                              #2
        verb = 'say'

    if noun in ['me', 'self']:                                    #3
        return getattr(self, verb, None)
#3

    elif noun and verb not in self.no_noun_verbs:               #1
        # Try and find the object
        object = [x for x in self.location.here + self.inventory
                   if x is not self and
                      x.name == noun and
                      verb in x.actions]
        if len(object) > 0:
            return getattr(object[0], verb)
        else:                                                       #4
            return False
#4

    # if that fails, look in location and self
    if verb.lower() in self.location.actions:
        return getattr(self.location, verb)
    elif verb.lower() in self.actions:
        return getattr(self, verb)

def process_input(self, input):
    ...
    handler = self.find_handler(verb, noun)
    if handler is None:
        return [input+"? I don't know how to do that!"]
    elif handler is False:                                     #4
        return ["I can't see the "+noun+"!"]
#4

#1 - Non-noun verbs
#2 - Special cases
#3 - Talking to ourselves
#4 - Can't find that...

```

Let's pay close attention to word choice. Some verbs don't apply to nouns, or else they implicitly apply to the player (**1**).

There are a few special case commands (**2**) which we can't really handle with our current system. We could rewrite our whole handler, but it's easier to catch them and explicitly convert them to something that we *can* handle. Of course, if it becomes more than a handful of conversions then we'll have to have a rethink, but it'll do for now.

So that we can see how we look, we'll add a 'self' object too (3). 'look self' should return your description as it appears to other people.

(4) is another improvement to make things easier for the new player. Rather than just have one error message when things go wrong, we'll have one for a command that we don't understand, and another when we can't find what the player's looking for.

Now our players can chat to each other and compliment each other on their fine threads. Finally, what would social gaming be, without the opportunity to be anti-social too? Most MUDs have the option to shout, which works much like speaking, except that everyone connected can hear you.

THAT WAS A CRAZY
BUG, GREG. HOW DID
PITR MANAGE TO
ATTACK HIMSELF?



Listing 10.10 - Anti-social gaming

```
def shout(self, player, noun):
    noun = noun.upper()
    #1
    for location in self.game.caves:
        #2
        for object in location.here:
            #3
            if ('events' in dir(object) and
                object != self):
                #3
                object.events.append(
                    self.name + " shouts: " + noun)
                #3
    return ["You shout: " + noun]
    #3

class Player(object):
    def __init__(self, game, location):
        self.game = game
        #4
        ...
    #4

class Monster(player.Player):
    def __init__(self, game, location, name, description):
        player.Player.__init__(self, game, location)
        ...
    #5

class Game(object):
    def __init__(self):
        ...
        orc = monster.Monster(self, self.caves[1],
            'orc', 'A generic dungeon monster')
    ...
    #5

class MudProtocol(StatefulTelnetProtocol):
    def connectionMade(self):
        ...
        self.player = Player(game, game.start_loc)
    ...
    #5

#1 -Shouting should look like shouting
#2 - Send it to all the caves
#3 - Find objects which can hear the shout
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

#4 - Update the Player class
#5 - Update all Player and Monster instances

We'll first convert the text to upper case (1), so THAT IT LOOKS A LOT MORE LIKE SHOUTING!

Now we need to visit each cave in turn, but there doesn't seem to be any way to find out what they are. For now, we'll just assume that we have access to the game's list of caves (2).

This is pretty much the same as when players talk to each other (3). Merging the two together, for example by pushing the code into the location class, is left as an exercise for the reader.

Now we need to give our player class access to the caves list from the game (4), by making it a variable that we pass in from the game when we create a player or monster...

..and then update each place where we create an instance of a player or monster (5), so that it now knows about the game object, and can tell where all the caves are.

There! That's a few more rough edges smoothed off of our game. There's plenty left to do, but we won't be writing any new features to the game itself now. Instead we're going to focus on making the infrastructure around our game a bit more robust, so that players won't be put off by having all their hard work disappear.

10.5 Making our lives easier

If you only want to write the game for your friends, you can probably stop here. They can connect and play your game, after all. Currently though, there are still a few issues which will make your life harder than it needs to be. Anyone can log on as anyone else, so it's not particularly secure, and the game doesn't save any progress, so every time you restart your game server they'll have to start over from scratch.

Let's fix that. We'll add usernames and passwords to the game, as well as a mechanism to allow new players to register. Once we know who's logged on, we can save the players' progress every so often and when they quit the game. We'll need to learn a bit more about Twisted though, since we'll be digging into the guts of one of its Telnet classes. But don't worry, it's straightforward once you get the hang of it.

10.5.1 Exploring Unfamiliar Code

Twisted is a large codebase and has a huge number of modules to help you network your application. That's great, because it means that you don't have to write your own code to handle the networking in your application, but it raises a related problem - you have to at least have a basic understanding of how everything fits together before you can make use of all that great code.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

OH, I MESSED UP THE INDEX IN THE ROOM'S LIST OF THINGS.

IS NOT FUNNY!



Ideally the documentation for libraries like Twisted will be 100% up to date and cover everything that you need to do, with a nice, gentle introduction - but this isn't always the case. Often, you'll be able to find something close but then need to piece together how the code works with some guessing, experimentation and detective work.

It sounds hard, but in practice it's usually pretty easy. The trick is not to get too overwhelmed, and to make use of *all* of the resources at your disposal. Here are some ideas on how you can get to grips with a large codebase and make it work in your application.

FIND AN EXAMPLE

Searching for 'twisted tutorial' online gives you a number of starting points, and you can also add 'telnet' or 'telnet protocol' into the mix too. As you learn more about Twisted you'll find other keywords or method names which will help you narrow down what you're looking for. You can also start with a working example which sort of does what you need, and then tweak it until it covers what you need it to do.

THE TWISTED DOCUMENTATION

There's reasonably comprehensive documentation available in the Conch section of the main twisted site, <http://twistedmatrix.com/documents/>, but it doesn't cover all of what we need to do. There are some simple examples of SSH and Telnet servers, which you can skim through to get an idea of how everything fits together.

THE TWISTED API DOCS

There is detailed, automatically generated documentation available for the entire Twisted codebase, which you can see at <http://twistedmatrix.com/documents/current/api/>. Don't let the sheer number of packages put you off - we'll focus on just the Telnet one: <http://twistedmatrix.com/documents/current/api/twisted.conch.telnet.html>.

THE TWISTED CODE

You can also read most of the Twisted code directly. The Windows version of Python stores its libraries at C:\Python26\Lib\site-packages\twisted, while under Linux it'll be somewhere like /usr/lib/python2.6/dist-packages/twisted. All of the Twisted code is stored there, and you can open the files and read the code to find out exactly what a method does.

INTROSPECTION

If a library doesn't have API documentation, all is not lost. You can still create instances of classes and use `dir()`, `help()` and `method.__doc__` to find out what they do. If you have a one-off method which you need to know about, this can often be easier than reading the code or documentation.

NEVER MIND PITR, I'M
SURE YOU'LL GET
THAT VORPAL WABBIT
SOMEDAY...



In practice, none of these sources will cover all of the details that you need to know when writing your program, so you'll end up using a combination of them going back and forth as you learn new parts or run into problems.

10.5.2 Putting it all together

Let's get started putting together our login system. From a quick scan of Twisted's Telnet module, it looks like the best starting point is the `AuthenticatingTelnetProtocol` class. We'll get that working with our code, then make it register new players and finally make the game able to save player data.

To start with, I looked at the Twisted documentation and the API reference for `AuthenticatingTelnetProtocol`. It sort of made sense, but from just the methods and classes it's hard to see how to tie everything together. The protocol needs a `Portal`, which in turn depends on a `Realm`, `Avatar` and `PasswordChecker`. Hmm, confusing. It looks like it's time to try and find an example of how the classes fit together.

There are few different searches that you could try, "twisted telnet", "twisted telnet example", and so on, but I didn't find much until I put in some terms from the code. "twisted telnet TelnetProtocol example" led me to <http://www.mail-archive.com/twisted-python@twistedmatrix.com/msg01490.html> which, if you follow it through to the end, gives you some example code which is enough to see how the classes work together.

The basic gist is something like this: Set up a `Realm` class, along with a `Portal` to get into it. The docs don't say whether it's a *magic* portal, but it should do. A `Portal` controls access to your `Realm`, using one or more password `Checkers`, via a `TelnetTransport`. Of course, the `AuthenticatingTelnetProtocol` only handles authentication, so you'll need to hand off to another protocol like our `MudProtocol` once you're logged in.

Got all that? No, me neither. I had to draw a picture to see how it all worked, and without the example I probably would've been lost. Here's what I came up with:

AFTER ALL, IS ONLY
CUTE LITTLE BUNNY.
HOW TOUGH CAN IT
BE?



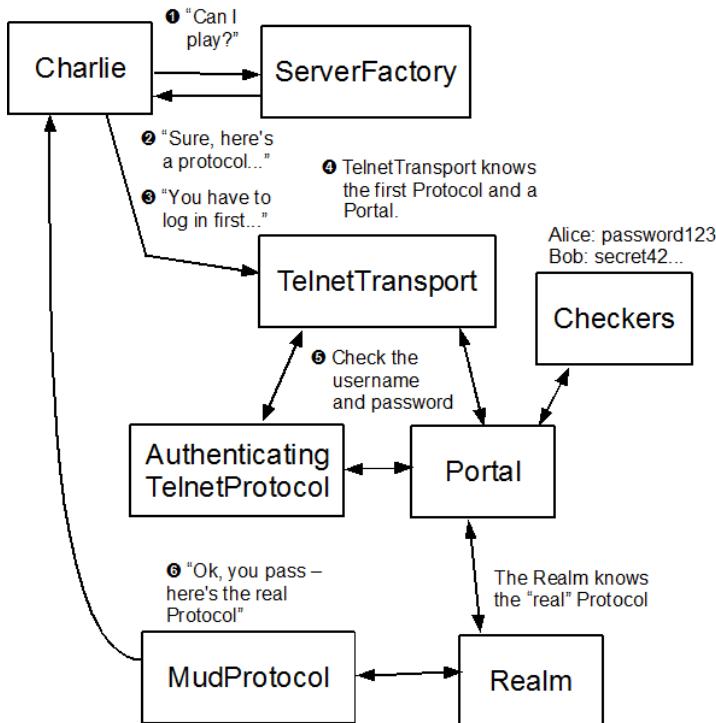


Figure 10.5 - The Twisted class structure

Using our diagram and the example code, we can get a simple login going. Here's how I changed the `mudserver.py` file:

Listing 10.11 - Mudserver.py

```

import sys #1

from zope.interface import implements #1
from twisted.internet import protocol, reactor, task #1
from twisted.python import log #1

from twisted.cred import portal #1
from twisted.cred import checkers #1
from twisted.cred import credentials #1

from twisted.conch.telnet import AuthenticatingTelnetProtocol #1
from twisted.conch.telnet import StatefulTelnetProtocol #1
from twisted.conch.telnet import ITelnetProtocol #1
from twisted.conch.telnet import TelnetTransport #1

...

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

class Realm:
    implements(portal.IRealm)                                #2
    implements(portal.IRealm)                                #2

    def requestAvatar(self, avatarId, mind, *interfaces):      #2
        print "Requesting avatar..."                           #2
        if ITelnetProtocol in interfaces:                      #2
            av = MudProtocol()                               #2
            print "**", avatarId, dir(avatarId)               #3
            print "**", mind, dir(mind)                      #3
            av.name = avatarId                            #2
            av.state = "Command"                          #2
            return ITelnetProtocol, av, lambda:None       #2
            raise NotImplementedError("Not supported by this realm") #2

    ...
class MudProtocol(StatefulTelnetProtocol):

    def connectionMade(self):
        ...
        self.player.name = self.name
        checker = portal_.checkers.values()[0]             #4
        self.player.password = checker.users[self.player.name] #4
        game.players.append(self.player)

    def connectionLost(self, reason):
        print "Lost connection to", self.ip
        if 'player' in dir(self):
            if self.player in game.players:                #4
                game.players.remove(self.player)           #4
                del self.player                            #4

if __name__ == '__main__':
    print "Prototype MUD server running!"

    realm = Realm()
    portal_ = portal.Portal(realm)                         #5
    checker = checkers.InMemoryUsernamePasswordDatabaseDontUse() #5
    checker.addUser("AA", "aa")                            #5
    portal_.registerChecker(checker)                      #5

    game = Game()
    ...
    factory = protocol.ServerFactory()
    factory.protocol = lambda: TelnetTransport(          #6
        AuthenticatingTelnetProtocol, portal_)
    reactor.listenTCP(4242, factory)                      #6
    reactor.run()                                         #7

#1 - What a lot of imports!
#2 - Create a Realm
#3 - Use debugging strings
#4 - Find our player's username and password
#5 - Set up our Realm and Password Checkers

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

#6 - Set up a ServerFactory
#7 - Send logging to the screen

To start off, we'll import all of the bits of Twisted which we need (1). There's a lot, but think of it as code which we don't have to write.

The `Realm` is the core class which represents our game's login (2). We only need to override one method - the one to get an `Avatar`. `Avatars` are instances of the `MudProtocol` and represent the player's login. Notice that we set the player's name so that we have access to it in `MudProtocol`, and set state to "Command", otherwise we'll get logged out straight away.

NOTE The "code that you don't have to write" part is important. It's very easy to overestimate how hard it is to learn how existing code works, and underestimate how hard it is to write new code that's as well tested.

While you're figuring out how everything works, it's perfectly fine to print out things to the screen to try and work out what each object does (3). You can use what you learn to search online, or through the code to find out what else uses these classes.

Most of `MudProtocol` is unchanged, but we'll need to know our player's username and password for later on (4), when we start saving to a file. The `Realm` has already given us the username, so we can use that to get the password from the checker. The other thing that we change is the `connectionLost` method - if we lose the connection to the player, we want to clean up properly.

Now we're into the section where we set our code in motion. The first thing to do is to create a `Realm`, then attach a `Portal` and `Checkers` to it. Once we've done that, we can insert usernames and passwords into our checker (5). The `InMemory..DontUse` is fine for our purposes, even though in theory it's insecure and we're not supposed to use it. There's also a file based checker available, but it doesn't support saving new users back to the file.

Now that we're using `TelnetTransport` and our `Realm` to control things, we don't need a custom Factory (6). The `TelnetTransport` will use `AuthenticatingTelnetProtocol` to handle usernames and passwords, but one that's done it'll hand off to the `Realm` to get the final protocol.

One last thing is that Twisted uses Python's log facility. To see what it's up to, you can add this line, which will redirect the logging to `sys.stdout` - in other words, print it on the screen (7).

What does all of this give you? Well, if you run your server now and then try to connect to it, you should be presented with a login request, instead of a password. Something like figure 10.5 below. If you enter the username and password that are in the script, you should connect to the game.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>



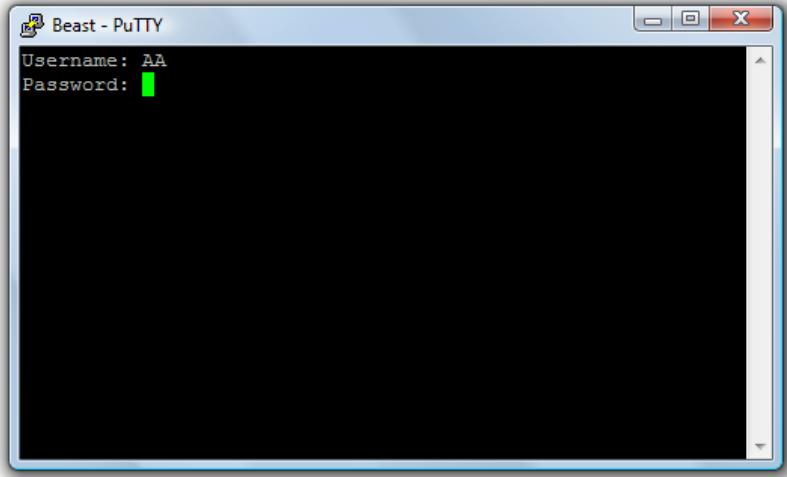


Figure 10.5 - logging into your game

That's not all that we need to do though. Remember that we want to allow players to register their own username and password. For that we'll have to learn a bit more about Twisted.

10.5.3 Write your own state machine

What we're going to do in this section is create a subclass of the class that we've been using so far, which is `AuthenticatingTelnetProtocol`. It's what generates the `Username:` and `Password:` prompts in the login above. What we'd like instead is a prompt that asks you whether you want to login or register a new account. If it's a registration, then it still asks you for a username and password, but creates the account instead of just checking whether it exists.

Let's first take a look at `AuthenticatingTelnetProtocol`, to see how it's done. You can find the telnet module on your computer at `C:\Python26\Lib\site-packages\twisted\conch\telnet.py`, or somewhere like `/usr/lib/python2.6/site-packages/twisted/conch/telnet.py` if you're using Linux or MacOS X. If you open that file and scroll to the bottom, you'll find the class that we're looking for.

AHEM! ER... SO, HOW'S
THE GRAPHIC DESIGN
COMING ALONG, AJ?



Listing 10.12 - Twisted's AuthenticatingTelnetProtocol class

```

class AuthenticatingTelnetProtocol(StatefulTelnetProtocol):      #1
    ...
    def telnet_User(self, line):                                     #2
        self.username = line                                         #3
        self.transport.will(ECHO)                                     #3
        self.transport.write("Password: ")                            #3
        return 'Password'                                           #3

    def telnet_Password(self, line):                                    #4
        username, password = self.username, line
        del self.username
    def login(ignored):
        creds = credentials.UsernamePassword(
            username, password)                                       #5
        d = self.portal.login(creds, None, ITelnetProtocol)          #5
        d.addCallback(self._cbLogin)                                 #5
        d.addErrback(self._ebLogin)                                #5
        self.transport.wont(ECHO).addCallback(login)
        return 'Discard'

    def _cbLogin(self, ial):                                         #6
        interface, protocol, logout = ial
        assert interface is ITelnetProtocol
        self.protocol = protocol
        self.logout = logout
        self.state = 'Command'

        protocol.makeConnection(self.transport)
        self.transport.protocol = protocol

    def _ebLogin(self, failure):                                     #7
        self.transport.write("\nAuthentication failed\n")
        self.transport.write("Username: ")
        self.state = "User"

#1 - AuthenticatingTelnetProtocol uses StatefulTelnetProtocol
#2 - Skip some bits
#3 - The username
#4 - The password
#5 - The fancy Twisted bits
#6 - If everything goes great - the callback
#7 - If everything goes bad - the errorback

```

All of the Telnet classes which we've looked at so far are state machines - there are multiple steps involved in logging in, and the next one depends on the input that we get. We're initially in the "User" state, which means that input is fed to the `telnet_User` method (1). Each method returns a string, which determines the next state.

There are a few other methods; `connectionMade` and `connectionLost`, but we don't need to deal with

*GREG? I NEED TO
BORROW YOUR PC FOR
A MINUTE...*

*NO GREG!
DON'T LISTEN
TO HIM!*



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

them in our case (2).

The first line (after the initial greeting) goes to `telnet_User`, and sets the username within the instance (3). The `transport.will()` call tells the local client that the server (ie. us) will be responsible for echoing anything what the user types, although in our case it's the password, so we don't. Then "Password" is returned, so that the next line goes to `telnet_Password`.

Now we have the password, we can compare it with what we have for that username in the portal's password checker (4).

Twisted has a mechanism called a 'Deferred', which helps to speed up the server (5). A password checker might look at a file on disk, or connect to a different server to see whether the password's correct. If it waits for the result (normally known as "blocking"), nobody else will be able to do anything until the disk or remote server responds. Deferred objects are a way to say: "When we get a response, handle it with this function" and then continue on with other tasks. In fact there are two possibilities - a callback and an errorback.

If the checker responds that the password is right (6), we can go ahead and do the rest of the login, which means storing some values, setting our state to "Command" and switching out our protocol for the final one.

If the checker tells us that the password or the username are wrong (7), then we can tell the user off and switch back to the "User" state. They'll need to type in their username and password again - hopefully they'll get it right this time.

So how can we subclass `AuthenticatingTelnetProtocol`? The answer is to add some new states so that there's a registration branch as well as the normal login one. Something like the flowchart in figure 10.6.

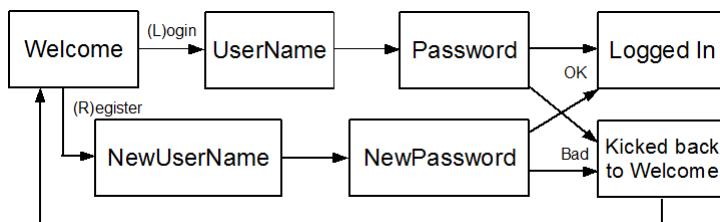


Figure 10.6 - The states in RegisteringTelnetProtocol

In other words, we'll add three new states; "Welcome", "NewUserName" and "New Password" and methods to handle each of them.

Listing 10.13 - RegisteringTelnetProtocol

```

class RegisteringTelnetProtocol(
    AuthenticatingTelnetProtocol):
    #1
    #1
    #1
    state = "Welcome"
    #1

    def connectionMade(self):
        #1
  
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        self.transport.write("Welcome to the server!") #1
        self.transport.write("(L)ogin or (R)egister " #1
                            "a new account? ") #1

    def telnet_Welcome(self, line): #2
        if line.strip().lower() == 'r': #2
            self.transport.write( #2
                "Enter your new username: ") #2
            return "NewUserName" #2
        elif line.strip().lower() == 'l': #2
            self.transport.write('Username: ') #2
            return "User" #2
        self.transport.write( #2
            "I don't understand that option.") #2
        return 'Welcome' #2

    def telnet_NewUserName(self, line): #3
        self.username = line #3
        self.transport.will(ECHO) #3
        self.transport.write( #3
            "Enter your new password: ") #3
        return "NewPassword" #3

    def telnet_NewPassword(self, line): #4
        for checker in self.portal.checkers.values(): #4
            if line.strip() in checker.users: #4
                self.writeline( #4
                    "That account already exists!") #4
                return "Welcome" #4
            self.transport.write( #5
                '\r\nWelcome to the server!\r\n') #5
            self.addNewUser(self.username, line) #5
            return self.telnet_Password(line) #5

    def addNewUser(self, username, password): #6
        for checker in self.portal.checkers.values(): #6
            checker.addUser(username, password) #6
    def _ebLogin(self, failure): #7
        self.transport.write("\nAuthentication failed: " #7
                            " %s (%s)\n" % (failure, dir(failure))) #7
        self.connectionMade() #7
        self.state = "Welcome" #7

#1 - Welcome to the server
#2 - Pick a path
#3 - Register a new name
#4 - As long as it's not taken
#5 - Add the user
#6 - Actually add the user
#7 - Handle errors properly

```

Welcoming the user to the server (**1**) is pretty much the same as the previous example, just with different values. We're prompting the user to enter R to register, or L to login.

Since our previous state was "Welcome", the first method is `telnet_Welcome`. The code is straightforward - R sets the state to "NewUserName", L to "User" and anything else will kick them back to "Welcome" (2).

`telnet_NewUserName` is the same as `telnet_User` too (3). It just prompts slightly differently and passes to a different state, "NewPassword", instead of "Password".

Of course, we can't have two Gandalfs or Conans running around our server, so we need to check that the user name doesn't already exist on our server (4). If it does, we kick them back to "Welcome". Pick something more original!

Now that the player has passed all of the hurdles which we've set for them, we should probably add them to the server (5). To make life easier for them, we also automatically log them in.

The last bit didn't actually add the user, it just pretended to. (6) will do the trick. We're just calling each of our checkers in turn and calling their `addUser` method. Note that this won't work if you use the file-based checker, `twisted.cred.FilePasswordDB`, or at least not permanently, since it won't write the players back to the file.

Finally, if the login raises an error, we should return to the initial "Welcome" state (7), rather than to "User", so that the user can register instead if they can't remember their username (or we've deleted it for some reason).

Awesome! Now we won't have to enter usernames and passwords for everyone who wants to check out our cool new game. In practice, this will mean that we'll get more players, since it sets the bar to entry much lower, and they won't have to wait around for us to check our email. The next step, if you're interested, is to include a password reset or retrieval mechanism, so that the player (if they've set their email address in-game) can be sent their password if they forget it.



10.6 Making our world permanent

We have a few more pressing concerns now - players can register and login, but if we restart the server for some reason (say, adding a new feature) then they lose all of their progress and have to reregister! We don't have to save *everything* though - what we'll do is save just the players and their items and restart all of the monsters from scratch. This is common practice in most MUDs, so that the monsters, puzzles and stories reset each night.

NOTE One of the other reasons to implement saving is that it breaks the player's suspension of disbelief if everything suddenly vanishes. You want the player to believe on

some level that the world you're creating is real, and real worlds don't disappear in a puff of virtual smoke.

Listing 10.14 - Loading Players

```
import os
import pickle

class Game(object):
    ...
    def __init__(self):
        ...
        self.start_loc = cavel
        self.player_store = {}                      #1
        self.load_players()                         #2

    def load_players(self):
        if os.access('players.pickle', os.F_OK) != 1: #3
            return                                #3
        load_file = open('players.pickle', 'rb')      #3
        self.player_store = pickle.load(load_file)   #3

#1 - Create a player store
#2 - Load players
#3 - Actually load the player store
```

We don't actually need to store every player, since we're only interested in their data - what they've called themselves, how they look and which items they're carrying. We'll put that information into the store **(1)** so that we can call it out at will.

The next thing that we'll do is figure out how we're going to call the code that we're going to use to load the player store **(2)**. I think we'll be alright if we just create a method.

The method to load the player store **(3)** turns out to be pretty simple. Check to see if the file exists - if it does, then open it and load the `player_store` from it using pickle.

Easy! Of course, we're not done yet - that only loads the player store. Now we need to work out what goes in the store, and save it to a file.

Listing 10.15 - Saving players

```
class Game(object):
    ...
    def save(self):
        for player in self.players:                  #1
            self.player_store[player.name] = \
                player.save()                        #1
        print "Saving:", self.player_store          #1
        save_file = open('players.pickle', 'wb')     #2
        pickle.dump(self.player_store, save_file)   #2

class Player(object):
    def __init__(self, game, location):
        ...
        self.password = ""                         #A
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

def save(self):
    return {
        'name': self.name, #3
        'description': self.description, #3
        'password': self.password, #3
        'items': [(item.name, item.description) #3
                  for item in self.inventory], }#3
#1 - Save each player
#2 - Save the file
#A - Add a password to the player
#3 - Create the player store

```

We add each player to the player store in typical object oriented fashion - by calling `player.save` to find out what should be stored for each player **(1)**.

Once we've refreshed the store, we can go ahead and save it to disk **(2)**, ready for the next time that we start the game.

All the `player.save` method needs to do is make a dictionary of all of the player's data and return it **(3)**.

Now our game's `save` method should be working, and we can load from it. The last step is to trigger `game.save` at appropriate points, and make sure that the players are loaded with all of their data when they log in.

MAYBE I COULD JUST
LOG ON AND THEN LOG
OFF AGAIN! THAT WON'T
HURT ANYONE!

SID - YOU KNOW
WHAT HAPPENED
LAST TIME...



Listing 10.16 - Updating the server

```

from item import Item

class Player(object):
    ...
    def load(self, config):
        self.name = config['name'] #1
        self.password = config['password'] #1
        self.description = config['description'] #1
        for item in config['items']:
            self.inventory.append[ #1
                Item(item[0], item[1], #1
                     self.location)] #1
    ...
    def quit(self, player, noun): #2
        self.playing = False #2
        self.game.player_store[self.name] = self.save() #2

        # drop all our stuff(?) #2
        for item in self.inventory: #2
            self.location.here.append(item) #2
            item.location = self.location #2
        self.inventory = [] #2

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        return [ "Thanks for playing!" ]           #2
...
class Game(object):
    def connectionMade(self):
        ...
        self.player.password = \
            checker.users[self.player.name]
        if self.player.name in game.player_store:      #A
            self.player.load()                         #A
            game.player_store[self.player.name])       #A
            game.players.append(self.player)

if __name__ == '__main__':
    ...
    def do_save():                                #3
        print "Saving game..."                   #3
        game.save()                            #3
        print "Updating portal passwords..."     #4
        checkers = self.portal.checkers         #4
        for player in game.players:             #4
            for checker in checkers.values():   #4
                checker.users[player.username] = \
                    player.password               #4

    game_saver = task.LoopingCall(do_save)        #5
    game_saver.start(60.0)                        #5

#1 - Load the player
#2 - Update the quit method
#A - Load the player on creation
#3 - Save the game
#4 - Refresh the portal's password list
#5 - Save every minute

```

Loading the player is much the same as saving it **(1)**, only the other way around. Rather than dump our state into a dictionary, we update our state from one.

Rather than have our player die whenever they quit, they'll now save themselves and exit nicely **(2)**. For our game we only have one sword and one coin to share between all of the players, so we'll drop all of our items, but that's not normal practice for an adventure game.

To save everything **(3)**, we'll set up another periodic function using Twisted.

The players can change their passwords in game, so it makes sense to refresh the server's password list along with saving the game **(4)**. Note that there's a bug in this code - when the player changes their name, the old name doesn't get removed. You'll want to either update the name changing code in Player



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

to delete the old name from both the portals and `player_store`, or else disable the name changing code. Disallowing name changes is probably the best option, since it also discourages bad behavior.

Once our function's complete, we just call it every so often (5). I've picked 60 seconds as a reasonable time frame, but you might find that a longer or shorter span works better for you. In practice it'll be a tradeoff between the load on the server when the game is saved, and the risk of losing your players' stuff.

That should be it. Now you have a stable base for your future development, and you don't have to worry about your players not being able to log in, or having to respond to everyone who wants to log in.

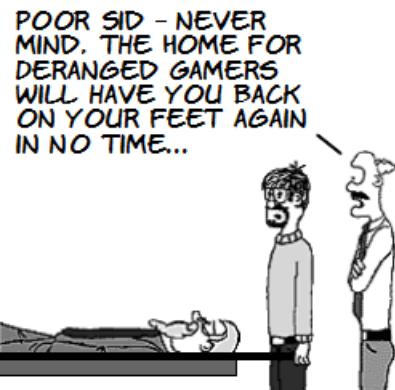
10.7 Where to from here?

Our MUD is working, and feature complete, but we've only scratched the surface of what we could do. One way to find out what needs to be done is to invite some of your friends to play - make sure that they know it's a work in progress - and ask them for suggestions and bug fixes. If you don't have any friends who are into MUDs, here are some ideas which you could try:

- Make the orc respawn once you've killed it (in a different location), or add different monsters. They might have different attacks, take more or fewer hits to kill and drop different sorts of treasure.
- Saving the cave layout as well as the players' info will help your players identify it more strongly as an actual place. Also, most MUDs will let you log in as a 'wizard' and extend the game while you're playing it, adding rooms or monsters.
- Different items, armor or weapons can add an extra level of interest, as players explore or save up their gold for new ones.
- Let the player gain experience and levels - with higher level characters being tougher and more powerful. Different character classes and statistics (strength, intelligence, dexterity and so on) can help players identify with the game and make it more enjoyable.
- There are a number of open source MUDs available, in several languages - so download them and see how they work. Most of the core components will be similar, so you'll know what to look for when you're trying to make sense of them.

10.8 Summary

This chapter, we learned how to add networking to a game, and about the issues that we need to deal with in networked environments. We started with a



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

simple chat server and learned about Twisted's `Protocol` and `Server` classes, before creating a similar setup so that we could play our game over telnet. Since Twisted is asynchronous (does lots of things simultaneously), we also needed to learn how to use Twisted's `task.LoopingCall` for our game loop.

Once we'd done that, we opened our game up for testing, and discovered a few issues with the game play in the new environment. To fix these, we added some new features, such as angry lists, talking to other players and commands to change our name and description.

Finally, we set up a system so that new players could log into our system without us having to add them to a list of users. In this part we learned a bit more about the details of Twisted, particularly its Telnet implementation, but also about how it interfaces with `Protocols`, `Servers` and also `Deferreds` - one of Twisted's more low-level features.

11

Django Revisited!

In Chapter 8 we built a simple todo list with Django, which allowed us to keep track of the things that we needed to do. While useful for us, it's not quite ready if we want to share it with other people. In this chapter we'll look at some of the polishing steps that we need to take to make our Django application useful to more than just ourselves. We'll add authentication so that everyone has their own login and todo list, as well as seeing how to:

- Unit test and functional test your application.
- Update your database when your models change.
- Serve static images and CSS style sheets from both Django and a separate web server.

Let's get started!

11.1 Authentication

Our application was pretty much done from a functionality point of view - we can add, delete and change any of our todos, and add as many as we like. The problem though, is that so can anyone else who has access to our web interface. If they're malicious, then all our todos might be deleted, or our important ones could be tampered with.

In order for our application to be safe, we'll need to restrict who can access our application, and you shouldn't be able to tamper with anyone else's todos. In practice, this means that we'll introduce the following checks:

- You should need to log in to the application.
- Once logged in, you should only see your own todo items.
- Whenever you try to add, change or delete a todo, it should only work if it's your todo.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

Once these three constraints are in place, we should be safe against anyone trying to fiddle with anyone else's todo list.

11.1.1 Logging in

Let's start with logging in to our application. Django provides a built-in application called auth, along with middleware to handle sessions and store user data. It makes user-based applications such as ours much more straightforward, and are a lot more robust and secure than "rolling your own".

SALES ARE STILL DOWN FOR WEB2.OTODO...



Listing 11.1 - Django authentication and login views

```
views.py:
...
from django.shortcuts import render_to_response
from django.contrib.auth import authenticate, \
    login, logout
...
def todo_login(request):
    username = request.POST.get('username', '')          #5
    password = request.POST.get('password', '')          #1
    error_msg = ''                                       #1
    ...
    if (username and password):                         #2
        user = authenticate(                            #2
            username=username,
            password=password)                         #2
        if user is not None:                           #3
            if user.is_active:                         #3
                login(request, user)                  #3
                return HttpResponseRedirect(           #4
                    reverse(todo_index))             #4
            else:
                error_msg = ("Your account has "
                               "been disabled!")       #4
        else:
            error_msg = ("Your username and password "
                           "were incorrect!")      #4
            password = ''                          #4
    ...
    return render_to_response(                          #5
        'templates/todo_login tmpl',                  #5
        {'username': username,                      #5
         'password': password,                     #5
         'error_msg': error_msg,                   #5
        })                                         #5
def todo_logout(request):                           #6
    logout(request)                                #6
    return HttpResponseRedirect(reverse(todo_login)) #6
#1 - Django's auth modules
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```
#2 - Set up common variables
#3 - Authenticate
#4 - Log in and redirect to our todos
#5 - Redirect back to the login page
#6 - Logout
```

We start with the three functions that we need to import to use Django's authentication (**1**).

This view is used in a few different ways, as the initial display of the login form and for checking a username and password. So that we don't run into `KeyError` exceptions, we're setting up the username and password variables (**2**) from the `request.post` dictionary's `.get` method so that if those values aren't set in the request, then they'll default to being blank. We also set `error_msg` to a blank string.

If we have a username and password then someone's trying to log in, and we use Django's `authenticate` function (**3**) to check them against our list of users.

If the username and password check out ok, then `authenticate` will return a `User` object. If not, it'll return `None`. That's easy to test for, but the other thing that we need to look for is where a user has been deactivated. If both of those tests pass, then we can log the user in with `login` and redirect to the todo index (**4**).

If we haven't been redirected to the index page, we'll pass through to this section, where we redisplay the login page (**5**). So that we can repopulate the form if there's an error, we pass back the username and password along with any error messages which we've generated.

Logging out is even easier - just call the `logout` function with the `request`, and it'll remove any session data and cookies that the user's been using (**6**). Once we've done that, we just redirect back to the login page.

Great - now we just need a template to display the login form. That's not too hard to do - here's my version:

Listing 11.2 - login template

```
<html>
<head>
<title>Todo Login</title>
<style type="text/css">
    body { font-family: Arial, Helvetica, sans-serif;
           color: black;
           background: #ffffff; }
    .error { color: red; }                                #1
</style>
</head>
```



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

<body>

{%
  if error_msg %}
    <p class="error">{{ error_msg }}</p>
{%
  endif %
}

<form action="" method="POST">
  <table>
    <tr><td valign="top">Username:          #1
      <td><input type="text" name="username"  #1
           value="{{username}}">            #1
    <tr><td valign="top">Password:          #2
      <td><input type="password" name="password" #2
           value="{{password}}">            #2
    <tr><td colspan="2">                #2
      <input type="submit" value="login">    #2
    </table>
</form>

</body>
</html>
#1 - Show any errors
#2 - The login form

```

If we get an error back from the view, we'd like to display it, so **(1)** is a section of template code that does just that. We've also added an `error` class, which displays the error in red.

Other than including the username and password as values, the form is a standard username and password login **(2)**. We're including the username and password which are fed in, so that if there's an error the user doesn't have to retype everything all over again. Little touches like this go a long way towards making your application look professional.

Last but not least, we'll tell Django about the views so that it can display them. Here's the `urls.py` plumbing to link everything up - `login` and `logout` just go straight to the relevant views.

```
(r'^login$', views.todo_login),
(r'^logout$', views.todo_logout),
```

Now if you go to `http://localhost:8000/todos/login` in your browser, you should be able to type in your username and password and have it redirect you to the index page. It should also give you a nice red error message if you mistype your password or username.

Note If you'd rather not be entering users by hand, there's a Django application called `django-registration` which will let people add their own accounts via email.



11.1.2 Adding users

The other thing that you're probably wondering at this point is how you can add new users. It's easy - just use Django's admin screen to create some. I'm not sure what your friends' names will be, so I'll just call my friend 'Bruce' to save confusion.

The figure consists of three vertically stacked screenshots of the Django admin interface:

- Screenshot 1: Add user**
The 'Add user' page. It has fields for 'Username' (bruce), 'Password' (*****), and 'Password (again)' (*****). Below the fields is a note: 'Enter the same password as above, for verification.' A 'Save' button is at the bottom right.
- Screenshot 2: Change user**
The 'Change user' page for 'bruce'. It shows the 'Username' field with 'bruce' and the 'Password' field with a long hash value. Below is a 'Personal info' section with 'First name: Bruce' and 'Last name:'. A note at the top says: 'The user "bruce" was added successfully. You may edit it again below.'
- Screenshot 3: Select user to change**
The 'Select user to change' page. It lists two users: 'anthony' (with email anthony@example.com) and 'bruce'. The 'bruce' row has a red circular icon next to the 'Staff status' column. Below the table is a note: 'The user "bruce" was changed successfully.'

Figure 11.1 - Adding a user through Django's admin screen

Just click on Users in the admin screen and you should see something like the first screen in figure 11.1. Fill in the username and password and your user will be created, then edit the relevant fields. If you look carefully at the permissions list, you'll see that there are permissions for adding, editing and deleting todos. We're not going to use them in this application, since they apply to every Todo, but they're there if you need them.

Alright, what's next? Now that we have to log in to our application, let's make our todos a bit more secure.

11.2 Listing only our own todos

Now that our users can log in, we can start making changes to how our application displays. Currently our index page still lists every todo in the database, but we'd like it to just show the todos which have been created by the current user. Actually, come to think of it, we don't have any

WOULD MEAN
REWRITING SITE FROM
SCRATCH

PERHAPS WE COULD
STREAMLINE THE
SIGN UP PROCESS
INSTEAD?



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

way to tell which todos belong to which user. We'd better fix that part first before we do anything fancy.

To add a link to the owner of a todo, we just add a foreign key to the Todo class in `models.py`. Something like this:

```
class Todo(models.Model):
    ...
    owner = models.ForeignKey(User)
```

Don't forget to import the `User` model from `django` as well:

```
from django.contrib.auth.models import User
```

The problem though, is that now we can no longer use `python manage.py syncdb` to update our database. For safety reasons Django will only add new tables, not tamper with your existing ones. But we have to do something, since Django will give us an error if we try to use the `Todo` model:

```
OperationalError at /admin/todo/todo/
no such column: todo_todo.owner_id

Request Method: GET
Request URL: http://localhost:8000/admin/todo/todo/
Exception Type: OperationalError
Exception Value: no such column: todo_todo.owner_id
Exception Location: /var/lib/python-support/python2.6/django/db/backen
```

Figure 11.2 - Our database is broken!

Our database and our model are out of sync! You have two choices at this point: either remove the `todo.db` database file and start again from scratch, or install SQLite and use it to update the database.

11.2.1 Fixing our database

We'll take the second option - although it's somewhat harder, you'll need to be able to update the database like this once you start working on applications where you have existing data. Don't worry, it's not too hard to do once you know a few simple commands. SQLite is available from <http://www.sqlite.org/downloads>, and all you need to do is put the executable somewhere where your operating system can find it. Listing 11.5 shows how I updated my database.

**NO - DEFINITELY NOT!
WE NEED ALL OF THAT
VALUABLE MARKETING
INFORMATION!**



Listing 11.3 - Adding a field to the database backend

```
anthony:~/todos$ python manage.py sql todo          #1
BEGIN;
CREATE TABLE "todo_todo" (
    "id" integer NOT NULL PRIMARY KEY,
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

    "title" varchar(200) NOT NULL,
    "description" text NOT NULL,
    "importance" varchar(1) NOT NULL,
    "owner_id" integer NOT NULL REFERENCES \
                "auth_user" ("id")
);
COMMIT;
anthony:~/todos$ sqlite3 todo.db
SQLite version 3.6.10
Enter ".help" for instructions
Enter SQL statements terminated with a ";" #2
sqlite> .tables
auth_group          auth_user_user_permissions
auth_group_permissions  django_admin_log
auth_message         django_content_type
auth_permission       django_session
auth_user             django_site
auth_user_groups     todo_todo
sqlite> .schema todo_todo #2
CREATE TABLE "todo_todo" (
    "id" integer NOT NULL PRIMARY KEY,
    "title" varchar(200) NOT NULL,
    "description" text NOT NULL,
    "importance" varchar(1) NOT NULL
);
sqlite> alter table todo_todo #3
      ...> add column "owner_id" integer NOT NULL #3
      ...> REFERENCES "auth_user" ("id"); #3
SQL error: Cannot add a NOT NULL column with \
           default value
sqlite> select * from auth_user; #4
1|anthony|||anthony@example.com|sha1$7405c$ ...
2|bruce|Bruce|||sha1$01946$ ... #4
sqlite> alter table todo_todo #4
      ...> add column "owner_id" integer NOT NULL #4
      ...> DEFAULT 1 REFERENCES "auth_user" ("id"); #4
sqlite> #4
#1 - What do we need to add?
#2 - Common SQLite commands
#3 - Alter the table, take one
#4 - Alter the table, take two

```

First we need to figure out what to add (**1**). `manage.py` won't make any changes for you, but it still knows what should be there. `python manage.py sql todo` will tell you the exact SQL needed for the database that you're using, which beats having to rack your brain trying to create the right SQL command.

It pays to know common SQLite commands (**2**). If you need to find your way around the database, `.help`, `.tables` and `.schema` are three very useful commands to know.

Next we issue an `alter table` command, with the SQL syntax that we've cribbed from `manage.py` (**3**). Unfortunately, SQLite won't accept it - we've told it that the field shouldn't be `NULL`, but haven't given it a default either, so SQLite won't know what to do with that

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

field for all of our existing todos. To fix the alter command, you can either drop the NOT NULL clause or add a reasonable default.

In my case, I've chosen to make the default be that the existing todos are owned by the admin user (with id=1) (4).

TIP There's a Django application called South, which can automatically alter your database for you based on your models.py file. It doesn't get everything (such as field renames), but it can be a lifesaver if you have a complex application.

Now that we've added our owner column and Django knows about it, our application is a lot more useful, and we can use our owner field to do all sorts of cool stuff. For example, the Django admin system will now let you change the owner of a todo with a convenient drop down box.



Figure 11.3 - Changing the owner of a todo

The Django admin application is very good at reading your models and making appropriate choices about how to display your data. All it needs is for the right relationships to be defined.

11.2.2 Back on track...

Our original plan was to only show todos which are owned by the person who's logged in. That's now easy to do - here are updated todo_index and add_todo views which will filter the todo list that you're shown when you log in.

Listing 11.4 - only show our todos

```

views.py:
def todo_index(request):
    if request.user.id is None:                                #1
        return HttpResponseRedirect(reverse(todo_login))      #1
    todos = Todo.objects.filter(owner=request.user)           #2
        .order_by('importance', 'title')                      #2
    return render_to_response(                                 #3
        'index tmpl')                                         #3
    {'todos': todos,                                         #3
     'choices': importance_choices,                         #3
     'user': request.user,                                  #3
     })
#)

def add_todo(request):
    t = Todo(title = request.POST['title'],
             description = request.POST['description'],
             importance = request.POST['importance'],
             owner=request.user)                                #4
    t.save()
    return HttpResponseRedirect(reverse(todo_index))

#1 - Catch people who haven't logged in
#2 - Filter our todos by owner
#3 - Refactor our response
#4 - Set owner in add_todo

```

If someone hasn't logged in, they could still manually type in the index url, or else bookmark the index page and forget to log in. That'll mess up our application, so if they're anonymous we redirect them to the login page (**1**).

Instead of just returning all of the todos, this database query (**2**) will return the ones where the user is the same as the one currently logged in.

In general, once you've found a clever new way to do things, it's a good idea to go back and clean up the code that you've already written. Here's our old template calling code, but using the new `render_to_response` function which we learned about in the last section (**3**).

The only thing left is to add the owner in to all of our new todos (**4**). Easy peasy!

So now we can view only our todos, and nobody else can see what we're up to. When we create a new todo, it's linked to our user id. Now all we need to do is perform the same checking when editing or deleting our todos. They won't appear in the list of todos, but that won't stop evil people from noticing that your todos are referenced by ids and seeing what happens when they put a different id in. Oops! Just deleted someone else's todo!



11.2.3 Covering all our bases

Here's where Django's simplicity comes into play. Your views are just functions, which get fed certain things depending on the request that comes in, and we can use that simplicity to our advantage. Rather than rewrite the entire view, and throwing all of our work away, we can wrap the view with some of our own code to check the request, and then pass our values to the generic view if the request is ok. Listing 11.7 shows how you can do that.

Listing 11.5 - Wrapping the update and delete views

```

from django.views.generic.create_update \
    import update_object, delete_object          #1
from django.shortcuts import render_to_response, \
    get_object_or_404                           #3
...
def update_todo(request, todo_id):               #2
    todo = get_object_or_404(Todo, id=todo_id)    #3
    if todo.owner.id != request.user.id:          #4
        return HttpResponseRedirect(reverse(todo_index) +      #4
            "?error_msg=That's not your todo!")       #4
    return update_object(                         #5
        request,                                #5
        object_id=todo_id,                      #5
        model=Todo,                            #5
        template_name='templates/todo_form.html', #5
        post_save_redirect='/todos/%(id)s'       #5
    )
def delete_todo(request, todo_id):               #6
    todo = get_object_or_404(Todo, id=todo_id)
    if todo.owner.id != request.user.id:
        return HttpResponseRedirect(reverse(todo_index) +      #4
            "?error_msg=That's not your todo!")
    return delete_object(                         #5
        request,                                #5
        object_id=todo_id,                      #5
        model=Todo,                            #5
        template_name='templates/'              #5
        'todo_confirm_delete.html'),           #5
        post_delete_redirect='...'             #5
    )
#1 - Import the generic views
#2 - Our old view
#3 - Another handy shortcut
#4 - Check that it's our todo
#5 - Call the generic update function
#6 - Deleting todos

```



(1) are the generic views, but we're importing

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

them in views.py, rather than urls.py

You may notice the stub view that we looked at earlier in the chapter (2), but we're expanding it out. We only need the request, and the id of the todo that we're editing.

`get_object_or_404` is another Django shortcut (3). It tries to access the model with that id, and if it can't it triggers a 404 error.

Before we get to the generic view, we want to check that the todo is owned by the person trying to edit it (4). If the id of `todo.owner` doesn't match the one in the request, then we redirect to the index page with an error.

If we get here, then the user owns the todo and we can pass control through to the generic view (5). All of the same arguments that were in `urls.py` previously are here, but they're specified as function arguments, rather than keys and values in a dictionary.

Deleting a todo follows exactly the same steps as updating a todo (6), except that the variables passed to the generic view are slightly different.

Now we need new urls to point to our new views. Here's a much cleaner version of our previous `urls.py`.

Listing 11.6 - New urls.py

```
from django.conf.urls.defaults import *

import views

urlpatterns = patterns('',
    (r'^login$', views.todo_login),
    (r'^logout$', views.todo_logout),

    (r'^$', views.todo_index),
    (r'^add$', views.add_todo),
    (r'^^(?P<todo_id>\d+)/{0,1}$', views.update_todo), #1
    (r'^^(?P<todo_id>\d+)/delete$', views.delete_todo), #1
)
#1 - Updated urls
```

You should be able to follow these updated urls easily by now (1). `(?P<todo_id>\d+)` matches an id and feeds it to the view as an argument, and I've also added an optional forward slash to the edititing url, just in case someone adds one by hand.

11.2.4 Updating our interface

The last thing that we need to do is to update our index page so that it can take an optional error argument. That's very easy to do, and in fact we've already done this for the login form, so we can just cut and paste it into the relevant sections here.

Listing 11.7 - Error messages on the index page

```
views.py:
def todo_index(request):
    ...
    'user': request.user,
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

        'error_msg': request.GET.get('error_msg', ''),
    })

index tmpl:
    .error { color: red; }
...
{%
    if error_msg %}
    <p class="error">{{ error_msg }}</p>
{%
    endif %}

```

#1 - Pass in any error messages
#2 - Show error messages on the index page

The first step is to pass in any error messages from the request into the template **(1)**. We're using a one liner here which is similar to the way that we handle username and password in the login script.

Now we can update the template to show a nice red error message if one is set in the url **(2)**.

With that, our application is done! We can look at, add, edit and delete todos from our todo list. As well as that, our interface is limited to just those people who we choose to give a username and password. In addition to this, if you go back and have a look through the code that you've written, you'll notice that it's nicely broken up - everything to do with display is in your templates, your models contain all of your data and data formatting functions, and your views handle your logins, data extraction and redirecting when something happens.



11.3 Testing!

In our original todo application, we developed it with unit testing, but so far we haven't seen any testing code at all. We can get away without having any testing code while our project is small, but as it grows it'll need some sort of testing to keep it in check. Also, Django's testing infrastructure is cool. Let's take a quick look at how you can test your Django applications.

11.3.1 Unit testing

The first thing that we need to know how to do is how to create unit tests to test our model. You can also use unit tests for other functions and classes that are independent and don't depend on any other infrastructure. Listing 11.10 shows how to create unit tests for your application. You should put it in a file called `tests.py` within your application folder.

Listing 11.8 - Unit testing (`tests.py`)

```

from django.test import TestCase          #1
from django.contrib.auth.models import User #1

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

from todos.todo.models import Todo          #1

class TestTodo(TestCase):
    #2

    def setUp(self):                      #3
        self.password = "IamBruce"
        self.user = User(id=1, username="bruce")
        self.user.set_password(self.password)
        self.user.save()
        self.todo = Todo(
            title='Test Todo',
            description='This is a test todo',
            importance='A',
            owner=self.user)
        self.todo.save()                   #3

    def test_short_model_name(self):       #2
        self.assertEqual(self.todo.short_description(), #4
                         'This is a test todo')           #4

        self.todo.description = "Test\nMultiple\nLines" #4
        self.assertEqual(self.todo.short_description(), #4
                         'Test')                     #4

        self.todo.description = ("A"*50) + ("B"*50)      #4
        self.assertEqual(self.todo.short_description(), #4
                         ("A"*50) + ("B"*30))           #4

#1 - Imports
#2 - Unittest layout
#3 - Set up some sample todos
#4 - Testing short_description

```

We're using Django's `TestCase` class to organize our testing code. It's mainly modeled on the xUnit style of testing, although it's possible to use doctests with Django too. Our `Todo` class needs to link to a `User`, so we're importing that as well as the `Todo` class (**1**).

xUnit tests are structured with a parent class and multiple `test_` methods within that (**2**). The parent class gives you lots of convenience methods to test for equality, truth, exceptions being raised, and so on.

xUnit also has the concept of a `setUp` method, which is called before each test method and is used to do things like set up common data structures. Here we're setting up a user and a `todo` which we need for our tests (**3**). There's also a corresponding `tearDown`, which is called after each test.

(**4**) is one example of what a unit test might look like. We're testing the `short_description` method of the `Todo` class, so we're setting up our test `todo` with different descriptions and making sure that the method returns the right value using `TestCase`'s `assertEqual` method.

You'll typically have a number of unit tests for each of your models, which test all of their functionality, but what about the



©Manning Publications Co. Please post comments or corrections to tl

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

views? They're even more important to test, since they typically define most of your application and link everything together.

11.3.2 Functional testing

For `views.py` and `urls.py`, you'll need to use some functional tests to make sure that everything's working. For functional testing, Django lets you submit 'pretend' forms with the `Client` class, which mimics a browser and the entire web request-response process. Listing 11.11 shows a few simple functional tests.

Listing 11.9 - Functional testing

```
...
from django.test.client import Client          #1
from django.core.urlresolvers import reverse   #1
import views                                  #1

...

class TestTodo(TestCase):
    ...
    def test_login(self):
        """Login should redirect to the index page"""
        client = Client()                      #2
        response = client.post(                #3
            reverse(views.todo_login),         #3
            {'username': self.user.username,   #3
             'password': self.password})      #3

        self.assertEqual(response.status_code, 302)  #4
        self.assertTrue(response['location'].endswith( #4
            reverse(views.todo_index)))       #4

    def test_index(self):
        """Index page should welcome the user"""
        client = Client()
        client.login(username=self.user.username,   #5
                      password=self.password)        #5
        response = client.get('/todos/')

        # print response.content
        self.assertTrue('Welcome, Bruce' in      #6
                        response.content)           #6
        self.assertTrue(self.todo.title in        #6
                        response.content)           #6

#1 - Imports
#2 - Django's Client class
#3 - Simulated POST
#4 - Response object
#5 - Client login
#6 - Accessing HTML content
```

Since we're testing views, the imports **(1)** will be the same as the ones we set up in our views.

Creating an instance of the Client class is easy **(2)** - it doesn't require any arguments.

Once you have a client class, you can use its `.post` method to send data to your application **(3)**. `.post` takes a url and a dictionary of POST arguments, and returns a response object. Notice that we're using reverse here - just as with views, it's important to keep your unit tests independent of where you happen to store your code. There's also a corresponding `.get` method too, which doesn't need the argument dictionary.

The response object **(4)** contains everything that's returned from the POST request that you've just run, such as status code, content, headers, cookies, and so on. Here we're interested in two things - that the response is a redirect, and that the redirect is to the index page (since we've logged in successfully).

We don't want to have to send in a login request every time we test something in our application, so Django provides the `login` method for the client **(5)**. It creates all of the cookies and session variables needed to simulate an actual login.

If you're testing a more normal request and you need to test what it returns, you can access it through `response.content` - it's just a string containing the HTML, just as if you were viewing the source of a page in your browser **(6)**.

So now we have our tests, but that's not much good - we need to be able to run them and make sure that our code tests ok. The beauty is that finding your test code and running your tests is done automatically.

PITR. HAVE YOU
FINISHED THE
ACCOUNT PAGE YET?



11.3.3 Running your tests

Django's `manage.py` contains a 'test' command, which will collect our test code and run it, as well as setting up all of the associated database infrastructure and so on. Listing 11.12 shows a sample test run against our todo application.

Listing 11.10 - Test run

```
anthony:~/todos$ python manage.py test todo
Creating test database...                                #1
Creating table auth_permission                         #2
Creating table auth_group                            #2
Creating table auth_user                             #2
Creating table auth_message                          #2
Creating table django_content_type                  #2
Creating table django_session                        #2
Creating table django_site                           #2
Creating table django_admin_log                     #2
Creating table todo_todo                           #2
Installing index for auth.Permission model          #2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

```

Installing index for auth.Message model          #2
Installing index for admin.LogEntry model       #2
Installing index for todo.Todo model           #2
...
-----
Ran 3 tests in 0.817s                         #3
#3
#3
OK
Destroying test database...                   #3
#3
anthony:~/todos$                                #4

```

- #1 - Running our tests
- #2 - Test database setup
- #3 - The test output
- #4 - Deleting the test database

If you just want to run your tests against one application, then just include the application's name after the test command **(1)**. Otherwise Django will test all of your installed applications, including applications like the admin interface, which might not be what you want.

For each test run, Django will create a test database and connect to that instead of your live database **(2)**. This makes your testing independent of the data you have stored in your application - you can even run tests against a live server if you need to.

Output from the tests is much like you would have for a standard unittest style test run **(3)**. Each test will either get a dot (pass), an E (error) or an F (failure). Once the tests have run, you'll get a report on how many failed, and tracebacks for any errors or failures.

Once the testing is complete, Django will delete the test database **(4)**.

Now you know how to make sure that your applications run according to plan, even if you need to pull them apart and refactor them completely. You can make sure that your releases don't have any known bugs, and that your code doesn't regress when you're developing. All that you need for a robust, healthy, stress-free project.

11.4 Images and styles

One of the last things that we need to do is to configure our image and style sheet serving. Up to this point we've been hard coding our style sheets into the HTML, but if you need to make a change later on, you'll need to edit every template. Django refers to images, stylesheets, javascript and other bits and pieces like that as 'media'.

First let's look at a simple way to serve media directly from Django, and then a more robust method where your media is delivered with a server such as Apache or Nginx.

11.4.1 Serving media from Django

First, a warning: this method is only really suitable for a

ALMOST, GRIGORY.
AM JUST ADDING
DISCLAIMERS TO
SPLASH SCREEN.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

development server. Django is written in Python, and is slow at serving flat files such as images when compared to a server written in C. If you get any significant traffic on your server, it won't be able to handle the load.

TIP Do one thing, and do it well. Django's built more for returning HTML populated with results from a database, so it's best to use it for that, and use something else to serve images.

That said, let's look at how to configure Django to serve static media files using one of Django's built-in views, `django.views.static.serve`. You'll need to make changes to both your `settings.py` and your `urls.py`.

Listing 11.11 - Serving static files with Django

```
settings.py:
MEDIA_ROOT = '/home/anthony/todos/'          #1
MEDIA_ROOT = 'C:/Documents and Settings/      #1
              Anthony/Desktop/todos/media/'    #1
...
MEDIA_URL = '/site_media/'                      #2

urls.py:
from django.conf import settings               #3
...
if settings.DEBUG:
    urlpatterns += patterns('',
        (r'^site_media/(?P<path>.*$)',           #4
         'django.views.static.serve',             #4
         {'document_root': settings.MEDIA_ROOT}),   #4
    )
#1 - Pick a directory to store your files
#2 - Decide where to serve them from
#3 - Check our settings
#4 - Serve the files!
```

First, pick a directory to store your media (**1**). To make it easy, I normally call it 'media' and put it in the root of my project. I've included versions for both Windows and Linux - note that the second and third lines are actually one line, and that Django uses forward slashes, even under Windows.

You'll also want to pick a URL to serve your files from (**2**). You're free to pick any URL that you like, but be aware that if you choose `/media/` you'll interfere with the admin application's media setting. `/site_media/` is the convention for most Django applications.

IF YOU WOULD BE
PROOF READING? IS
HARD TO CHECK WHEN
WORDS ARE BURNT ONTO
RETINAS.



To be doubly sure that we're not going to use Django to serve up images when our site goes live, we check the value of `DEBUG` from `settings.py` (3). If it's set to `True` then we're in development and should be safe.

Finally we set up the `django.views.static.serve` view (4). The two variables that it needs are the path and the document root - other than that it can take care of itself. To save duplicating our setup, we're also including the `MEDIA_ROOT` from `settings.py`.

Once you've made those changes, you can restart the server and start adding your images and stylesheets. Figure 11.4 shows the Django logo displayed on my development server.

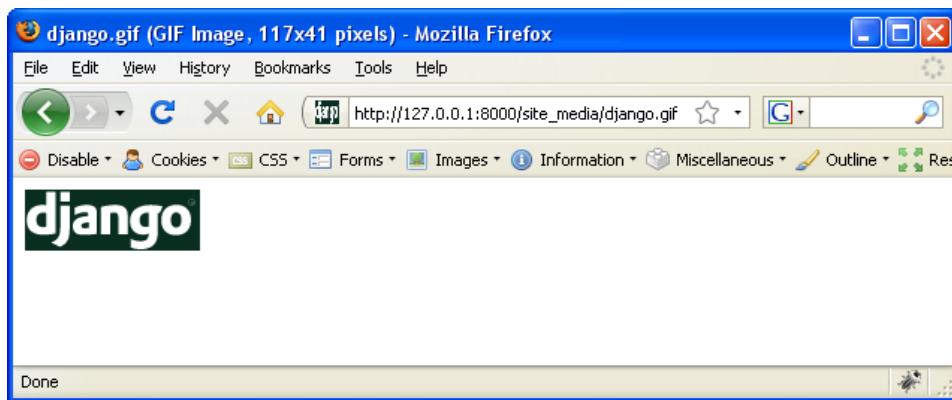


Figure 11.4 - Serving up images with Django

Now you can include images, css files, and javascript in your application templates by referencing `/site_media/` like this:

```
<link rel="stylesheet" type="text/css"
      href="/site_media/style.css">

```

Actually creating a logo for your todo list is left as an exercise for the reader!

11.4.2 Serving media from another server

A better way to serve media though, is to use a program expressly designed to serve static content like images and style sheets and save Django for just the dynamic pages. This is relatively easy to do with most web servers, and there are a number of ways to achieve the same end. Listing 11.14 gives an example of how you might configure Apache with `mod_python` to serve requests for media and images, but pass other page requests on to Django.

Listing 11.12 - A sample Apache mod_python configuration

```

<VirtualHost *>
ServerName www.example.com
DocumentRoot /var/www/www.example.com

<Location "/">                                #1
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE todos.settings
    PythonDebug On
</Location>                                    #1

<Location "/media">                            #2
    SetHandler None
</Location>                                    #2

<LocationMatch "\.(jpg|gif|png)$">              #3
    SetHandler None
</LocationMatch>                                #3

</VirtualHost>
#1 - Configure mod_python to serve Django
#2 - but not for /media
#3 - or anything involving images

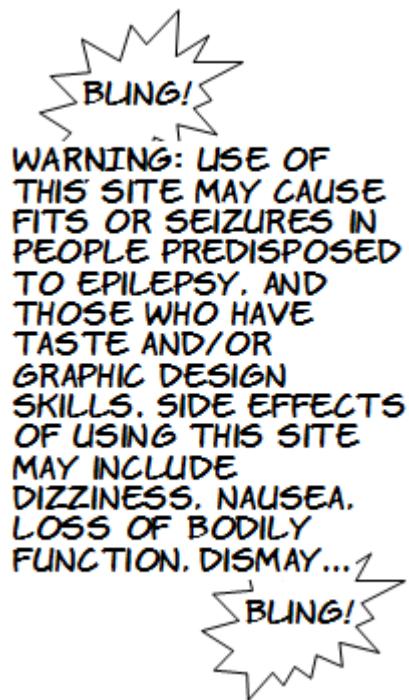
```

Section (1) is pretty much a stock Django-with-mod_python configuration section. We use Django's modpython handler, use our todos.settings as the settings module, and switch debugging on - at least, while we're setting up and testing everything.

For /media though, we'd like Apache to serve files normally, which means that it'll fall back to the normal document root for the server, and display our media which we've stored in /var/www/www.example.com/media (2).

We can do much the same thing with a LocationMatch directive, so that URLs like `http://www.example.com/not_a_media_folder/logo.gif` will fall back to the image stored at `/var/www/www.example.com/not_a_media_folder/logo.gif` (3).

Note also that you're not limited to using subfolders like this - you can use completely separate domains. For example, if you had `www.example.com` serving Django pages, then



it's possible to serve media from `media.example.com` or `images.example.com`. If your application model supports it, arranging your URLs like this can save a lot of configuration.

11.4.3 Last but not least

The final thing that you need to do is to edit your `settings.py` file and find the `DEBUG` setting. With this set to `True`, Django will give you detailed information whenever something goes wrong. It's useful while developing, but this information can be dangerous in the wrong hands. Once you've switched it to `False`, Django will just return a standard 500 error if your site breaks, and keep your application's innards safe.

11.5 Where to from here?

Your application is now not only fully functioning - you can also install it on a server on an intranet or the internet and give out accounts to all your friends. It's still somewhat bare-bones though, so here are some ideas for extending it to make it more useful:

- Now that you can use separate style sheets, you can pretty up your application, add logos and icons to your main page and put your forms within tables. A little bit of beautification can make a big difference to how seriously people take your application.
- Your todos are fairly basic. What about adding some extra data to them, such as deadlines? If you recorded email addresses, the system could also email people when their deadlines are a week or a day away.
- Once you have a few users, you could consider making the application more collaborative. Perhaps you could add a field to mark todo items as public? Other people's public todo items would be included in your list, and you'd be able to view, but not edit them. A list of users in the system and what people are working on might also be useful.
- Or, the ability to add notes to your todos might be useful, perhaps even allowing other people to add comments.

There's also a wealth of information available on the more advanced aspects of Django. The Django documentation, for example is excellent, and freely available from the Django website.

You can also download some of the many Django-based applications and read through them. It's a very good way to learn good ways to structure your project, and learn about new libraries to make your development even easier. A good place to start is the Pinax website, which provides a number of reusable modules such as user registration and pagination (breaking a big list up into smaller pages of 10 or 20). There are also modules for integrating external services like Paypal or Facebook, and entire applications such as content management systems.

11.6 Summary

In this chapter we learned about some of the issues associated with hosting and maintaining Django. We started with adding users and logins to our system, and saw how to update the database when we made changes to our model. Then we took a look through our system so far and secured all of our pages and forms so that different users (or even external attackers) couldn't get access to todo lists that were supposed to be private. In the process we got a lot more hands-on with our views and saw how to wrap some of Django's built-in views so that we could add extra features without having to reimplement the views from scratch.

We then saw how to add tests to our Django application, and how Django allows you to easily add functional tests by providing a Client class which acts like a web browser.

Finally, we looked at how we can serve our static content, such as images and style sheets, both with a built-in view in Django itself, and through a more efficient mechanism like Apache.

That's all for the Python programming that we'll learn in this book, but in the next chapter you can find out what your next steps should be to improve your Python skills even further, and several sources of assistance if you get stuck on your journey.

12

Where to from here?

If you've got to this point in the book you've learned several different Python programs, in a number of different styles. We started out with a straightforward program in Chapter 2 when we wrote Hunt the Wumpus. Since then we've covered libraries, classes, event based programs and interacting with the web. You could think of Quick & Easy Python as a tasting plate, letting you try different styles of Python programming before you read too deeply into any particular topic.

Although we've covered a lot of ground in this book, you're only just beginning to scratch the surface of what you can do with Python. This chapter is intended as a springboard for the next stage of your development as a programmer.

12.1 Read some more code

One of the best ways to learn how to write better programs is to look at how other people have written their programs, and figure out what they've done and why. There's something of an art to reading other people's code well, but experience definitely helps.

I find that the best way to understand new code is to skim its structure first for an idea of the design of the program (so that you won't get *completely* lost), and then dig into the details of how it's written. To remember it more easily, ask yourself questions as you go: Why have they split the program up this way? Why a dictionary instead of a list here? Is there a better way to do this? How could I extend it if I needed to do something differently? Is there a library which will help?

Bear in mind that not all of the program code that you find on the internet is of production quality - some might be throw away prototyping or proof of concept, and some might be for older versions of Python. Asking questions like those above will also help you watch out for these sorts of pitfalls.

Here are some ideas as to where you can find code to read.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

12.1.1 Python Standard Library

A fair chunk of the Python library itself is written in Python. By digging into it, you can find out how common Python features and libraries are implemented, by the people who wrote them. Bear in mind though, that some of the libraries might be somewhat older and use techniques which have been deprecated.

12.1.2 The Python cookbook

Sites like <http://code.activestate.com/recipes/langs/python/> and <http://djangosnippets.org/> provide Python functions or modules, which illustrate a particular technique or solve a specific (small) problem. They're useful when you know exactly what you need to do, say check an ISBN book code or find out how to solve an anagram, but they're also easier to follow when you're first starting out.

12.1.3 Open source projects

Once you've read a few small code samples, you might want to look into larger programs. There are a number of open source projects available on sites like Sourceforge (<http://sf.net/>) and Google Code Hosting (<http://code.google.com/hosting/>), both of which will let you search specifically for Python based projects. Sites like <http://ohloh.net/> will also give you statistics on the age of the project, number of developers and lines of code, so that you can pick established code or smaller projects depending on your comfort levels.

12.2 Join the Python community

Another good way to find out how to improve your programming is to make contact with other people who know Python. Asking questions about what they're up to is a good way to learn more about what's possible.

12.2.1 Sign up to some mailing lists

A good place to ask questions (if you've searched online and not found a solution, or if you're really stuck) is the python-tutor list. You can subscribe to the list at (<http://mail.python.org/mailman/listinfo/tutor>). Don't forget to search the list archives before you ask your question, otherwise you might be asking a question which has been asked a hundred times before.

Also don't forget to "pay it forward" - once you find you've outgrown the tutor list, stick around and help some other people learn how to use Python. Don't worry if there's a question that you don't know how to answer, but if you can help out, jump in. It's run by volunteers, who will be more than happy for the help.

There'll also be mailing lists dedicated to other areas that you might be interested in learning about, such as Django or Pyglet. You'll not only pick up a number of



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=598>

techniques and learn about libraries that work well with that area, you'll also be able to read the discussions and find out why things are written a certain way, or discover limitations before you run into them.

12.2.2 Find a local user group

Python meetups and user groups are an excellent way to find Python programmers who are active near you. They'll often have regular meetings or get togethers, and are a good source of advice and new ideas. It's one thing to read web sites about what Python can do; it's another thing entirely to talk to people about their projects in person.



12.2.3 Help out an open source project

If there's an open source Python project which you use on a regular basis, you might want to consider signing up for the developer mailing list, becoming familiar with its code and contributing patches. Most open source projects will have some sort of tracker which you can use to find bugs which you think you can fix. Even verifying that a bug exists and investigating possible causes is a good start - you don't necessarily have to fix it all in one go. Alternatively, add a minor feature, write documentation or a tutorial, or add a unit test.

12.3 Scratch your own itch

The next time you have a problem, or a cool idea, you can start your own project - whether it's a web site, game, todo list or data processing application (don't laugh - I know a few people who've used Python to manage their fantasy football or office footy tipping, both of which are heavily data-intensive).

Once you've found your feet with Python, the best way to continue learn is by doing. Pick a problem - either something that you're interested in, something which annoys you or something which will be useful - and start trying to solve it. Just don't forget to start with one small chunk, otherwise you'll be overwhelmed.

Another option is to build on the code in this book when starting out. I've covered a lot of ground, so there'll be something close to what you're trying to write, or something which you can use as a scaffold. Writing a program is a lot easier once you have a basic idea of your project and which direction to take it.

When you're ready to share your project with the world, there are plenty of sites like Sourceforge (<http://sourceforge.net/>), Google code (<http://code.google.com/hosting/>), Github (<http://github.com/>) or Bitbucket (<http://bitbucket.org/>) which you can use to publish your code, provide documentation and track bugs and feature requests.

12.4 Look at more Python libraries

As your programs grow in scope, you'll find that you need to be able to do more and more things - perhaps talk to other programs over the internet, load specific data formats or run programs faster. In this book we've already covered several libraries which can help extend what you can do; here are some others which you might otherwise miss. Some of them are included with Python; others you'll need to download and install separately.

12.4.1 Profiling code (hotspot/profile)

If you're writing code that needs to run quickly, or if it's just running a lot slower than you thought it would, Python comes with a profiler called `cProfile`, which can tell you how long the individual parts of your program are taking to run. You can use this information to rewrite just the parts which are slow (or cache or pregenerate them) instead of having to guess why your program's running slowly.

12.4.2 Logging

Python also has built in support for logging - writing status reports to a file so that you can tell what your program's up to. You can log at different levels, only producing some lines if you've configured your program for debugging output, and log to several different destinations, such as a file, printing to the screen or syslog if you're using Linux.

12.4.3 subprocess and multiprocessing

Sometimes you might need to run several processes at the same time - usually if you're running as a system program in the background, or if you're doing something processor intensive and need to make use of all of your system's CPUs. The `subprocess` module is the standard way of running separate processes, and you can use `multiprocessing` if you need to run processes in parallel for extra speed.

12.4.4 Better parsing

We already used `shlex` (along with some custom code) when writing our adventure game, but there are other solutions if you need a more featured parser. `Pyparsing` is easy to get started with and allows you to define more complex types of grammar, rather than just splitting on quotes and spaces, but there are many other types of parser available, depending on your experience and needs.

12.4.5 PIL and image processing

If you're doing any sort of image processing work in Python, `PIL` (the Python Image Library) is essential. With it, you can crop and resize images, merge them,



accept binary image data over the web, check it and save it to disk - even generate images from scratch.

12.4.6 XML, ElementTree and JSON

For XML and XHTML parsing, it's hard to go past ElementTree. It was added to the Python standard library in version 2.5, and has several different models for parsing and inspecting XML data. Python also has `xmlrpclib`; handy if you need to communicate with other programs that use XML-RPC.

If XML seems a bit heavyweight for you, there are a number of other formats that you can try. `json` is ideal if you're working with data on a Javascript-enabled site, but is broadly useful even when just storing data or transmitting it between programs.

12.5 Summary

Now you know not just how to program in Python, but where to find help or further inspiration if you need it. To take your next steps on the path to mastery, you'll want to read code, talk to other programmers, look for new libraries and techniques, and most importantly, experiment and create programs of your own. You won't even need to be hit with a bamboo pole!

