Python for Professionals

Learning Python as a Second Language

MATT TELLES

bpb

Python for Professionals

Learning Python as a Second Language

by

Matt Telles

FIRST EDITION 2020

ISBN: 978-93-89423-754

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

My wife Teresa

About the Author

Matt Telles is a 35-year veteran in the software industry. He has worked with virtually all programming languages, and has been a developer, manager, tester, and designer. He's been working on Python for several years and is constantly extending his knowledge in the field.

Matt is married with three children, living in New York, the United States. He has a menagerie of cats, dogs and a turtle, and loves reading books on his Microsoft Surface on the train to work every morning.

About the Reviewer

Tarun Behal is a computer scientist and software engineer with almost 8 years of experience in software development. He completed his graduation in computer science and software engineering from UPTU, India, in 2012. Currently pursuing masters from BITS Pilani, India. Since then, he has worked with several technologies and languages, including Odoo, Django, JavaScript, Ruby, PHP and Python. He currently resides in Noida, India and works for Adobe Inc.

Some of the applications covered by Tarun during his career include RESTful APIs, ERPs, billing platforms, hotel management, financial systems and e-commerce websites. He has been using Python on both personal and professional projects since 2012, and he is passionate about software design, software quality, and coding standards.

I would like to thank my parents for their love, good advice, and continuous support. I would also like to thank all my friends that I met along the way, who enriched my life, for motivating me and helping me progress.

Acknowledgement

First and foremost, I would like to thank all of the employers who have allowed me to learn my skills while working for them. Thank you for that opportunity.

Secondly, I would like to thank my family, who have put up with me while writing this book.

Finally, I'd like to thank everyone who reads the book, and finds anything in it to be useful in their daily lives. I've spend a lot of time over the years learning from others, this is my chance to give back to the community.

– Matt Telles

Preface

Python has become very popular among programmers, testers, and web developers. The majority of Python is easy to pick up and run with right away, but the details of the language are sometimes a bit difficult to understand without good examples. The purpose of this book is to help a professional programmer get started quickly with the language. The target audience of this book is someone who has written programs in the past, not necessarily in Python though. This book is divided into 10 chapters and it provides a detailed description of the core concepts of Python programming.

Chapter 1 introduces the history and installation of Python, as well as the differences between the current versions to help you determine which one to install.

Chapter 2 addresses the basic data types in Python, as well as the collections and iterables.

Chapter 3 discusses the nuts and bolts of Pythons, with a focus on conditional statements, loops, and objects, with their attributes.

Chapter 4 discusses how to work with Python, from the immediate mode of the interpreter to the importing of pre-built packages and modules for use in your application.

Chapter 5 is involved with object-oriented programming as it applies to Python. You will learn about the pillars of object orientation and how they are implemented using the Python programming language.

Chapter 6 addresses the concepts of advanced manipulations in Python. You will learn about comprehensions, generators and slices.

Chapter 7 is all about input and output in Python, from writing to the console to reading and writing files. You'll learn about JSON and text files, as well as working with binary file formats.

Chapter 8 describes imports and exports, how to use other people's code and make your own code available for the community or your own company.

Chapter 9 addresses a variety of miscellaneous topics, from decorators and properties to documentation and metaclasses.

Chapter 10 will help you to focus on not reinventing the wheel by exploring some of the more popular Python packages available to you, and looking at how to use them in your own programs.

Chapter 11 will present a series of tips and tricks that you can use to immediately upgrade your Python programming skills.

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

Chapter 1
The History and Installation of Python

Introduction

Every professional programmer who has been in the industry for any length of time has seen changes. Whether it is a new job, or a new boss, or simply a new approach at work, change is the only constant in the programming world. New operating systems, new frameworks, new devices, and, of course, new programming languages. For those of us that came up in the Unix world, or using MS-DOS, change has been dramatic. If you've worked mostly in the Linux world, the change may have been less dramatic. Changes in your development language, however, are always both exiting and traumatic. Moving to a new language is like moving to a new house with all new things to get used to and to accept. If you are coming from the Java, C# or C++ world, transitioning to Python might be confusing. Not only has the syntax changed, but the very way of thinking has changed. The purpose of this book is to ease that transition, and to help you think like a Pythonista. A Pythonista, of course, is one that has adopted Python as their primary language, and struggles to master new concepts in Python at all times.

Structure

- History
- Selecting a Python version

- The Zen of Python
- Keeping it simple
- Using virtual environments
- Using pip
- Python IDE's and command line work

Objectives

By the end of this chapter, you should understand how to select a Python version to use, how to install the Python system, and how to use some of the general tools that Python developers need in their day to day experience. You will learn about the history of Python, the Zen of Python, and how to keep it simple.

Python: the language

One of the most frustrating things for professional developers is finding a good transition methodology. Let's be honest, you don't want to learn about what a memory location is, or how the compiler/interpreter works to do your job. You want to start out with that nice simple *Hello world* example, see how the types and statement work, and what the gotcha's are in the language. You already know what loops are, what assignment statements are, and how to use functions and classes from your previous work. What you don't know is how that new language implements all these things, and that's what you will learn here.

We will look at the basics of Python in this chapter, as well as introducing some basic concepts that you'll need to think about when writing in Python. You'll find out a little bit of trivia, because what developer doesn't love to know trivia about his or her language? You'll find out the philosophy behind Python and why things are the way they are. Then we'll get into the nuts and bolts of getting you a functional development environment, and show you how to create your first Python program.

Welcome to Python! May your journey be fruitful and your code concise.

History

If you've never seen Python before, it might surprise you to know that it has been around for a very long time. Older than Java and C#, Python was first created in the late 1980s. It was created by Guido Van *Rossum* at CWI in the Netherlands. The first cut at the language didn't get much traction in the *real world* of software development though. It wasn't until the turn of the millennium, the year 2000, when Python 2.0 came out and people began to use it in earnest.

Where the original Python language was a very simple interpreted language best suited for UNIX scripting, the second version of the language was much more

robust. Supporting memory management and garbage collection, not to mention full **object-oriented programming (OOP)** concepts like classes and inheritance, it made writing complex tasks very easy. Much like Visual Basic did for Windows, Python did for the early days of Linux, making it easy for non-developers to get quickly into programming. Unlike Visual Basic, however, Python was well thought out, enough so that advanced programming was done quickly. Because it was interpreted, the development cycle for Python was rapid, and thus was adopted in many companies for that reason alone.

As the language adoption increased, so did the *packages*, code libraries that accomplished complex tasks. We'll look at those later in the book, but Python today is used for such diverse tasks as writing web servers and doing artificial intelligence. It has excellent mathematical libraries that make it ideal for doing statistical work as well. Because of its small footprint, Python is available on virtually every platform in existence. With the second release came support for Unicode, giving it international acceptance as well.

Python is a *byte-code* language, like Java or C#. That means that the interpreter will *compile* your scripts into a simplistic byte code that can be quickly and easily interpreted. This is good, in that it is fast and easy to use. At the same time, there is a cost, as there always is. Python interprets things as it reaches them, meaning that syntax errors aren't caught at compile time, but at run-time.

The biggest reason that people gravitate to Python is its simplicity. Rather than some languages which have a dozen different constructs for looping, Python has but two; the `for` statement and the while statement. A language like Java requires a dozen lines to open a file and write to it, Python requires three. In C# or C++, you have to structure your code in a specific manner that the compiler accepts, in Python; you can implement a full program in a single line in a single file. Functional programming used to require hideously complicated languages like Scala, which in turn required the Java system; Python can do it in a few lines of code. That's not to say that Python can replace all existing programming languages, it most certainly can't. Being an interpreted language, it is naturally slower than a compiled language. It lacks the GUI libraries that many languages come with, and the security built into other byte-code languages like C# or Java.

Oh, and finally, the fun facts. Python isn't named after a snake. Rather, it is named after the British comedy show *Monty Python*. It has become so popular that it is the de facto standard programming language at Google, a somewhat large multi-national software company. Python is open source, meaning that it isn't controlled by a single company or a single person. Anyone can contribute to the Python system, there is a committee that approves language changes, but if you want your own personal version, you can do so. Finally, in a study done by *Ocado Technology*, Python is a more popular language than French in schools. Take that Napoleon!

Selecting a Python version

Jumping right in to the topic, as professionals do, let's get started with Python. Of course, first we need to install the system on our devices. Python will run on almost anything, although it isn't really recommended that you write code on your phone. You can, though, which speaks to the simplicity of the language and the compactness of the interpreter. Still, we'll stick with writing code on a computer. For the purposes of this book, it doesn't matter whether you use a PC, a Mac, or a Linux box, the language is exactly the same. The environment you choose to develop with may vary, but the reality is, you can use any editor you want, either an integrated (IDE) system or a simple text editor and command-line execution. Before we get to install the system, we need to make a big decision. Should we use Python 2 or Python 3?

For the purposes of this book, we will be using Python 3.x. almost all of the code and libraries will work just fine in the latest Python 2.7 release, but for consistency sake, we'll choose Python 3.

Why not use 2.7?

Python 2 is coming to the end of its useful lifetime. That doesn't mean that it will go away anytime soon; there are literally millions of lines of Python code out there that use the 2.x environments. However, the Python organization (see: **https://www.python.org/**) has decided that January of 2020 will be the last date for which new updates will be made to Python 2. It isn't like the language will implode, or that all of your scripts and applications will suddenly stop working, but there are regular bug fixes and language feature updates that will no longer be applied to the Python 2.x branch. For this reason, we will be using Python 3.x in this book, and you should in your own code.

Migrating from Python 2.7 to 3.0 is not a major endeavor, and there are many guides to doing so. The biggest obvious change, for those of you that worry about this sort of thing, is that the print statement, which looked like this in Python 1.0 and 2.0:

```
print "something"
```

Will be converted into a full-fledged function:

```
print("something")
```

This change can be annoying if you get used to writing code without the parentheses. In addition, it works fine with parentheses in Python 2.x, so you might as well just get used to it. To be honest, there was never a good reason to omit the parentheses around statements; it was a holdover from older languages like BASIC.

Which 3.x to use?

Once we have made the decision to go with Python 3.x, the next question is which number to fill in after the x. The answer is that it really doesn't matter. As of the writing of this book, Python 3.7 is the most current version, but 3.6 is the most commonly used version. There are a few packages out there that have not yet been updated to support some changes in 3.7, so a number of people are not yet using it. For our purposes, any version will do, as we are not going to get into the internals of the language and interpreter. In the code, it may assume you are using Python 3.7, but if you have a slightly older or newer version, don't worry, it is almost certain that the code here will work as advertised. If a newer version comes out after the book is published, the code on the associated website will be updated to work with it, and so you are golden.

Installing

All versions of Python can be found on the main Python downloads page at: **https://www.python.org/downloads/**. You will find installers here for all major operating systems and devices, as well. While the actual mechanics of installing Python vary slightly depending on the system you are installing it on, the basics are always the same. The distribution is available in several flavors; depending on your familiarity with each you can choose the one that best fits your skill levels. For example, all of the operating systems distribute Python as a compressed ZIP file. For Windows, you could select that and install it on your local environment, or you could use the MSI installer that is native to Microsoft Windows. Likewise, for the Macintosh, there is a native zip file as well as a native Mac installer file (PKGfile). The official Linux/UNIX distribution is done by a compressed `tarball` file (TGZ) which can be expanded on the system of your choice. Individual Linux distributions normally come with Python pre-installed, although you can add your own.

A note about installing: Unlike many languages, Python permits you to install multiple versions on the same machine and choose between them when building your projects. We'll talk about this more in a little bit when we discuss virtual environments.

For the Mac, the default OSX installation comes with Python 2.7 installed. Unfortunately, this version is not only old; but it is not the general distribution too. As a result, you will want to install a newer version whether or not your system already contains Python.

In this case, we'll go through the installation process for Windows and Mac, since they are similar but slightly different.

For Mac OSX, first, download the package from the above link at python.org. While you can select any of the possible downloads, it is recommended that you use the

64 bit stable installer. The Mac is moving away from 32-bit programming, as most operating systems are, and it is simply easier and more forward-looking to work with 64-bit code now, instead of porting it later. Download the `python-3.7.<revision>-macosx<os-version>.pkg` file to your Mac downloads directory (`~/Downloads`). In this case:

Revision is the current released version of Python. As of the writing of this book, that number is 4, it is most likely that this will change, possibly even to the minor version (.7). Python is normally completely backwardly compatible, so this shouldn't present a problem for you.

Launch the installer by either opening it in Mac Finder and double-clicking it, or selecting **`Open`** in the download bar of your browser. You should see the installer window pop up as shown in *Figure 1.1:*

Figure 1.1: *The Python installer running on Mac OS10.9*

Click Continue will lead you through the process of installation. Note that you will be presented with a screen informing you that you absolutely, positively must read the license agreement before continuing *(Figure 1.2)*.

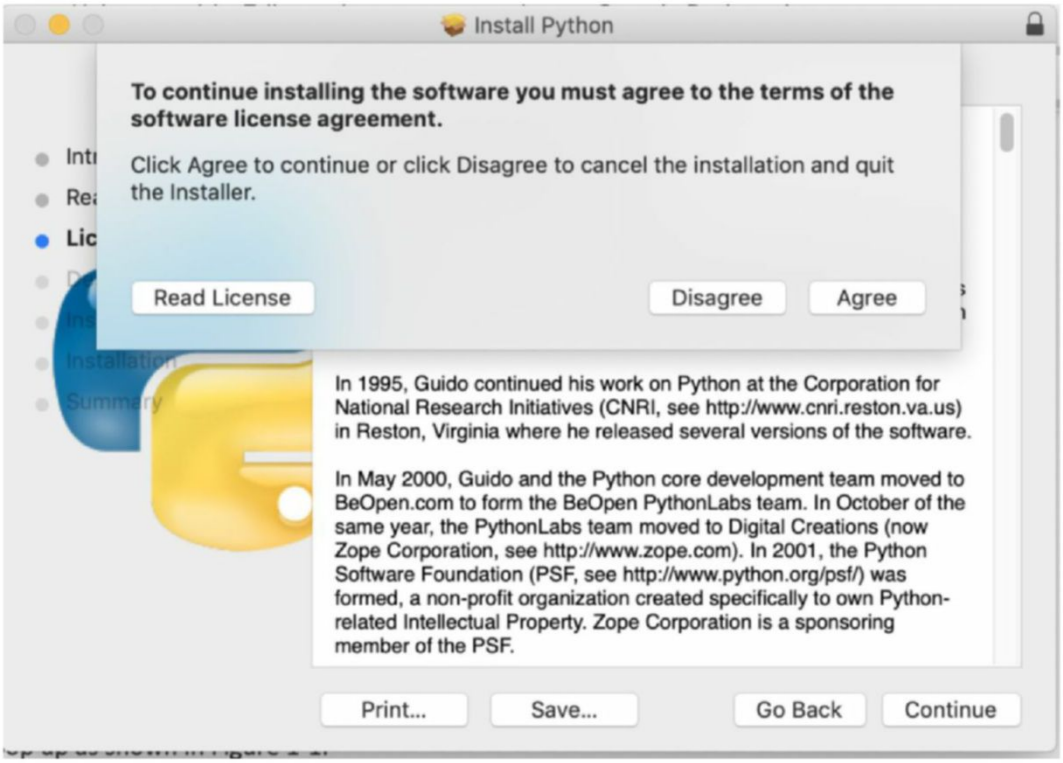

Figure 1.2: *Read license nag screen*

Scroll through the license screen using the scroll bar on the right and click agree, and it will stop nagging at you. Your next selection is where to install it. Normally, unless you are on a corporate machine where you cannot install things outside your own personal directory, you can just accept the defaults here. Click on through until you reach the **`Install`** button and click it, the installer will do its work and you will have Python 3.x installed on your Macintosh.

For Windows, the process is very similar. Launch the installer in Windows as shown in *Figure 1.3*. Click the install button. You may wish to customize your installation to place the system where you want it on your computer. Depending on your

configuration, you may be asked to allow the installation to proceed. Go ahead and do so. It will chug along for a while and then finish:

Figure 1.3: *The Windows installer.*

In the Windows environment, this is all that is necessary. The installer will do the rest of the work, and you can proceed with testing your installation.

Testing your installation

Python has no visual element, aside from an included package called **IDLE** (as in Eric Idle, one of the stars of Monty Python), which differs from environment to environment. Instead, Python is run from the command line or shell. To verify that your Python installation went swimmingly, open up a terminal (Mac) or command window (Windows) and type "`python`" and hit enter. You should see the following:

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

To exit the Python interactive shell, type `quit()` and press return.

For the Windows world, click on the Start button and scroll down to the Python entry (for my system, it was Python 3.7, but yours might be slightly different). It should have a *new* entry below it. Notice that there is an entry called *Python 3.x* where *x* is the version you installed. Click on that and you will see a new Command Prompt launched with the entire environment variables set for the Windows world. You can then follow the instructions above to verify that the system is working properly.

Congratulations! You have just installed Python on your system. You are well on your way to becoming a Pythonista.

The Zen of Python

Python is more than simply a programming language, at least to those that use it. It is a way of creating software that is simple and straightforward, leading to better applications that are easier to maintain and debug. The Agile movement drew a lot of its inspiration from the Python philosophies and many of the things we take for granted in software development are here because of the push of the Python community.

The philosophy of Python is so important that it is actually baked right into the language. To see it, start your Python interpreter and type the following:

```
import this
```

We will talk about the import statement and its usage a bit later, that's not the point here. This is an *easter* egg in the interpreter itself. If you've typed it correctly, hitting return should show you the following:

```
>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
```

```
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Understanding the Zen understands what Python is all about. Put simply, the Zen can be broken down into three basic groups. First, keep it simple. Second, keep it readable. Finally, if you can't explain it in a one line comment, it is probably too difficult to implement, much less support.

Keeping it simple

In most programming languages, developers strive for elegance and *efficiency*.Python will never be the most efficient language in the world, since it is interpreted, but it is generally fast enough for anything but systems level programming. Elegance, to the average developer, seems to have become synonymous with complicated. For example, in Java, we might write something like this to use the streams library:

```
Integer sum = list.stream().map(Employee::getSalary)
                .reduce(0, (Integer a, Integer b) -> Integer.sum(a, b));
```

To a Java programmer, something like this makes sense. If you aren't a Java programmer, this code is intended to sum up the values in a list of integers. We aren't going to get into serious Python programming in this chapter, but let's take a quick look at what the same code would look like in Python.

Assuming one has a list of integers:

```
a = [1,2,3]
```

And you wanted to sum them, you'd write:

```
the_sum = sum(a)
```

Where `sum` is a Python function that accepts an *iterable*, that is a container of values that can be iterated over, and returns the total of them. If you don't happen to know that the function exists, you could write:

```
sum = 0
for x in l:
    sum += x
```

Both approaches do the same thing, the second one is easier to understand if you are coming from another programming language like C++ or Java. The first is more intuitive to the Python programmer. Both are simple and easy to read. Given a choice, choose simplicity over elegance.

Keeping it readable

Many programming languages do not enforce any sort of constraints about how the code can look. For example, this is perfectly valid C++:

```
int x=0; for (int i=0;i<10;++i) x = x+i; printf("x = %d\n", x);
```

It works, simply because the only delimiter that C++ cares about is the semi-colon. Any statement terminated by a semi-colon will be parsed as a separate bit of code, no matter how cluttered it looks, or how hard it is to read.

Python uses a different approach to statements. Indentation is the key to Python code. Each block of code has to be indented to the same or a greater level than the block above it. For example, to write an `if` statement in Python, we use the following:

```
if (some_condition):

do_something

else:

do_something_else()

then_do_something_cool()
```

You can see from the above code clearly that the `do_something` line is only called if the if statement above it evaluates to `true`, as the `do_something_else` line is only called if the statement is `false`. The `then_do_something_cool` function is called after the `if` is evaluated, no matter what the `if` statement evaluates to. By simply glancing at the indentation level, we know exactly where a given piece of code falls. Indentation is not only nice, it is required in Python.

Never is better than right now

This particular statement might seem odd if you are coming from another programming language. Python encourages you to do things right, or not do them at all. The language hasn't changed radically over the years because the designers and implementors of Python long ago decided that a feature that just *made it easier* wasn't necessary. Instead, they selected more generalized features, as we will see, that made it possible to do things the way you need them done. When you are writing Python code, don't look for the *coolest* way to do it, select the one that will be the easiest for people to understand and modify.

Most of the Python tools and extensions that are used today are written in Python. The language did not require changes to make it more powerful, Python developers

managed that with the minimal language they were given. As we will see when we discuss things like virtual environments and pip, Python allows for you to extend its power without changing the language for everyone else.

Using pip

Not quite time to start coding yet, but it is time to explore a little of the environment. Let's begin with the pip command. The **pip** command is short-hand for **package installer for Python**, and is used to download and install packages. Packages are like libraries in C++ or Java, and are similar to the C# NuGet components. It extends the language by adding new bits of functionality that is available to your code.

For some environments, pip is automatically installed. For others, such as Linux and MacOS, you need to install it before you can use it. For Mac OSX, you install pip using the `easy_install` program:

```
sudo easy_install pip
```

The `sudo` part is necessary because pip becomes part of the system functionality, available through `/usr/bin`, and thus requires elevated permissions to install. Once you have pip installed, it is simple to use.

For right now, there are four commands that you should concern yourself with for pip:

Command	Arguments	Purpose
`install`	<package-list>	Installs one or more packages from the command line into the Python environment
`uninstall`	<package-list>	Removes one or more packages from the command line from the Python environment
`list`	n/a	Lists all installed packages.
`search`	Package-name	To find a given package given a full or partial name.

For example, running the pip list command in my current environment returns the following (this list is partial, and will vary by your environment):

Package	Version
asn1crypto	0.24.0
certifi	2019.6.16
cffi	1.11.5
chardet	3.0.4
Click	7.0
cryptography	2.4.2

decorator	4.4.0
EasyProcess	0.2.7
enum34	1.1.6
Flask	1.0.3
gevent	1.4.0
greenlet	0.4.15
idna	2.6

As you can see, the Python environment comes with a number of pre-installed packages. The list above does show some that have been installed by hand on my system. The name of the package is the name that is listed in the package repository. For Python, the main package repository is called **PyPI**. This is short for **Python Package Index**. The package name is the first column, and the currently installed version of the package is the second column. All Python packages have versions baked into them, so that you can find the proper version for the version of Python and other packages you are using. We will discuss versioning in `Chapter <TODO>` as well as talking about how you can define the exact set of packages and versions that you need for a given project.

Suppose, for example, one of your fellow developers tells you about this wonderful package called `Flask`. As we will see later on, `Flask` is a package that helps you to write web services. We could search for the flask package using the pip search command:

```
pip search flask
```

The return from this command is a list of packages that match the search string, including those in which the name is contained within it. Here is a very partial list of what would be printed out.

```
pip search flask
Flask-SimpleMDE (0.3.0)     -    Flask-SimpleMDE - a Flask extension for
                                 SimpleMDE
Flask-Pure (0.5)            -    Flask-Pure - a Flask extension for Pure.
                                 css
Flask-OrientDB (0.1)        -    A Flask extension for using OrientDB
                                 with Flask
Flask-ElasticUtils (0.1.7) -     ElasticUtils for Flask
Flask-Waitress (0.0.1)      -    Flask Waitress
flask-zs (0.0.17)           -    A helpers for Flask.
flask-ws (0.0.1.0)          -    Websocket for flask.
```

```
Flask-PubSub (0.1.0)       -    Flask-PubSub
flask-helloworld (1.0.0)   -    Flask Helloworld
sockjs-flask (0.3)         -    SockJs for Flask
Flask-Stripe (0.1.0)       -    Flask-Stripe
Flask-Quik (0.1.1)         -    Quik for Flask
Flask-BDEA (0.1.0)         -    Flask-BDEA
Flask-Helper (0.19)        -    Flask Helper
Flask-GripControl (0.0.1)  -    Flask GripControl
Flask-SRI (0.1.0)          -    Flask-SRI
```

As you can see, there are a lot of pieces to the Flask system. This is in keeping with the keep it simple philosophy of Python. Someone wrote a good general purpose package that did something. Other people came along and added very specific extensions to that package, but rather than force everyone to use a massive package that contained much more than they might want, they created a separate package to add to the base one. For example, the `flask-ws` package requires the basic `Flask` package, and then adds functionality allowing you to use web sockets (internet protocol connections) along with your `Flask` functionality.

The output from search is the package name, with the version in parentheses, followed by a description of the package. The simplest way to install a package is to use the basic form of the install command:

```
pip install Flask
```

This command will install the current version of the base `Flask` library. If `Flask` has multiple versions, the latest will be used. What if you don't want the latest version? The pip command supports this variant:

```
pip install Flask==1.04
```

Where `1.04` is the specific version you want. It even allows you to do this:

```
pip install Flask>=1.04
```

Where `1.04` is the minimum version you want.

What if you have the package installed, but what to upgrade to a more current version? Typing `pip install` package will not do anything if the package is already installed. Adding the `–upgrade` flag to the command `pip install –upgrade` package however, will upgrade the package in place for all users.

Likewise, you can remove packages that are installed. This can free up space, as well as removing packages that might be interfering with things, such as the oddball case where you have a local package of the same name as one installed via pip.

```
pip uninstall Flask
```

This command will remove the `Flask` package from the global environment.

Another interesting thing about the pip command, the creators of Python, and pip, understood that using Python in a corporate environment and in build environments means that you don't always have permission to install to the centralized location for packages. For these cases, the `–user` flag will install the package only for the current user.

Finally, we come to the subject of dependencies. It is not at all uncommon, as seen in the Flask case above, for a given package to rely on having other packages already installed. The pip command automatically handles dependent packages by installing all of the dependencies first. This is done via the information that is provided in the package definition, which will be discussed later in the book.

Oh, one last thing about pip. It is written in Python! You can prove this to yourself by running this command:

```
python –m pip
```

We'll talk about what this means, but essentially it means run the pip module through the Python interpreter. Any module can be run directly if it has a main entry point.

Using virtual environments

Our next *non-programming* programming topic is that of a unique Pythonesque concept, virtual environments. A virtual environment is a self-contained *sandbox* of sorts for a Python project. It is easier to explain why the thing exists than how it works, so let's just talk about that. Virtual environments might sound like the latest in game technology, but they are anything but.

One of the biggest problems in software development is collisions between different systems. In the Windows world, we called it *DLL Hell*, in which you had multiple versions of a dynamic link library. In the Mac world, this wasn't an issue until OSX, in which everything was moved into a centralized area, causing problems with version collisions.

Let's say that you are developing a brand new version of your application. This application, written in Python, requires a certain package. That package, let's called it Foo, was created just for your application. It was version 1.0 of the package and did things in a very specific way. Sadly, the developers who wrote Foo took it with them when they got fired from the company and came out with a version 2.0 that isn't at all backwardly compatible. Even more sadly, your new application requires version 2.0 of the Foo package to run, since it has all kinds of sparkly new functionality. In the majority of programming languages that used any sort of centralized library system, you would be out of luck. There would be no way to have both versions of the package installed on your system. This is the problem that virtual environments solve.

Like pip, virtual environments are written in Python. You can create one using the `venv` package (there are others, the virtual environment structure is standard, anyone can create amodule to implement it). Where Python is different from other systems is that not only can you define the versions of packages, even system packages that you want for your project, you can also define exactly what versions of Python and its tools you want to use for your project. You can even have multiple virtual environments for a single project, and switch between them.

It is a standard process to define a virtual environment for each and every project you create. The overhead is fairly minimal, and you gain a lot of safety and control. If someone accidentally upgrades your Python interpreter as part of a corporate-wide initiative, it will not affect your individual projects, since they refer to a specific version of Python and packages.

The reason that virtual environments came to be is due to one of the rare poor choices by Python's designers. There are two types of packages that can be installed for Python, system packages and site packages. System packages are never a big problem because they are part of the release version and will always work with a given release of the Python interpreter. You can re-download a system package for a given version and be assured that it will continue to work without interruption.

Site packages, on the other hand, are developed and distributed by third-party developers. The package we discussed above, `Flask`, is an example of a site package. While a system package is always guaranteed to be compatible with the Python version it is written for, there are no such guarantees for site packages. As in our example above, you might have three different versions of `Flask` for three different applications that you are maintaining. The poor choice made by the designers was that site packages are all, by default, stored in the same place and without regard for versioning. Let's say that we have two projects that use a site package, called *Package A*. One of these projects uses *Package A* version 1.0, while the second project uses version 2.0 If you install version 2.0 of the package, it will overwrite all (or worse, part) of version 1.0. This clearly is not an optimal solution.

Site packages are normally stored in a specific single directory, which is pointed to by an environment variable in the operating system. Python uses that environment variable to determine where to load its site packages. The virtual environment resets that environment variable, but only for the lifetime of the virtual environment. This is confusing, because the notion of overriding the environment is confusing. It is easier to show you how all this works from the user perspective.

For the moment, we are going to assume that our project is on the Mac, and that it exists in the `~/projects/project_1 directory`. To create a new virtual environment within the directory, we'd issue the following command in the Terminal (on Mac, a Command Prompt in Windows):

```
cd ~/projects/project_1
python3 -m venv env
```

This command loads the `venv` module into the Python interpreter, and passes along the argument `env` to that module. It will create a new subdirectory within the current directory (which we have changed to be `projects/project1`) called `env`. Note that since we are using Python 3.6, we use the "`python3`" command to run it. On the Mac, using the `python` command will launch the default system version, which is 2.7. You cannot remove the system version on Macs which is annoying.

If we look at the contents of the `env` directory, we will see the following structure:

```
├── bin
├── activate
├── activate.csh
├── activate.fish
├── easy_install
├── easy_install-3.6
├── pip
├── pip
├── pip3.6
├── python -> python3.6
├── python3 -> python3.6
└── python3.6 ->
    /Library/Frameworks/Python.framework/Versions/3.5/bin/python3.6
├── include
├── lib
   └── python3.6
   └── site-packages
└── pyvenv.cfg
```

The bin directory contains a bunch of the applications that are used by Python, like `pip` and `easy_install`, as well as links to the actual Python executables. In this case, the `python3` executable is pointing at the system python3.6 executable. We could modify this to point at anything we want, and `venv` actually gives us a way to do this. For example, if you wanted to create a virtual environment using Python 3.7, you'd run:

```
python –m venv --python=/usr/bin/python2.6 venv2_6
```

This command would create a structure virtually identical to the one above, but in place of Python 3.6, it would be using version 2.6. Normally, you wouldn't switch back and forth between versions 2.x and 3.x, but you can.

Once you have created the virtual environment, you have to activate it to use it. To do this, there is a shell script file (batch file in Windows) called activate, in the `bin` directory. In Linux or on macOSX, you'd enter the command:

```
source bin/activate
```

For Windows, you'd run bin/`Scripts/activate.bat`, otherwise the process is the same.

Once you run this command, you will see the following change to your Terminal or Command Prompt:

```
(env) my-machine:bin my-user$
```

Notice the (`env`) part of the Command Prompt, which indicates that we are running inside a virtual environment. Normally, on the Mac, if I were to type `python`, I will get the default 2.7 installation versions. Within the virtual environment, however, typing python gives me:

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

The important directory here, though, isn't the bin one but rather the `lib` directory and the `site-packages` directory beneath it. Thanks to the wonders of environment variables and links, this is where any packages that you install once the virtual environment is activated will be placed. Since each virtual environment has its own `site-packages` directory, each time you install things for a given environment, you will get another copy of it. This isn't wonderful for disk space, but it is for detangling the mess that is site package collision.

It is important to note that only after a virtual environment is created and activated will site packages be stored in it locally. Before that, they will be installed globally. Because Python will follow the `PYTHONPATH` environment variable to find its `site-packages`, this means that within the environment, `site-packages` in the virtual environment are found first, while outside of it, the global `site-package` will be the only one found. We'll talk a little more about this when we discuss imports and modules.

Python IDE's and command line work

One of the nicest things about working with Python is that there are no specific tools that you absolutely have to use. You can choose your own editors, your own source code control, and your own reporting mechanisms. The only thing that is *required* is that you have a version of Python on your system. Most developers, however, prefer one of two approaches to software development.

The first type of developer is what is often referred to as the *bare metal* sort. These are the developers that don't want any sort of fancy integrated system. They have the source code editors that they have been using forever. Most of them work directly from the command line, and look at the output from the compiler or interpreter and then go back into those editors in order to make changes before running again. These developers are very comfortable with the output of their tools and wouldn't change anything for the world. Python works just fine for people like this group.

The second group of developers prefers a modicum of *hand holding*, using integrated environments that do a lot of the work for you. These integrated environments make it simple to create new projects, setting up the environments and virtual settings automatically. They remember your favorite settings for compiling and running code, and show errors in a list that can be used to go directly to the offending line.

For the first group, you already know what you want to do. The process for creating a new Python project is simple. Create a new sub-directory named after your project. Create a new virtual environment by running the `python -m venv env` command from the command line. Create a new Python file within that directory using your favorite text editor. Syntax check and run the file using the command `python my-file.py`, where `my-file.py` is the name of your newly created file. Look at the output, make any changes, and continue.

For the second group, things are a little more interesting. There are numerous integrated environments available for Python. Some of them are free of charge; some of them are commercial and cost money. As of the writing of this book, the two most popular **integrated development environments (IDEs)** are PyCharm, which was created by Jet Brains software, and Visual Code, which was created by Microsoft. Both have their advantages and disadvantages.

PyCharm is solely for the development of Python applications, although it is based on the same core code that is used for IntelliJ for Java and Scala, as well as other languages like C++ (that one is called CLion). PyCharm is distributed in two forms, a community edition that is open-source and free, and a more extensive professional edition. The community edition, which can be downloaded from **https://www.jetbrains.com/pycharm/download/** can be seen in *Figure 1.4*. You can see that it has sections to edit code, output from your running Python script, and a **`Project`** tree that allows you to see all of the files in your project in one go. PyCharm has become something of a standard in Python development in the corporate world,

thanks mostly to the integration with third party tools like source control and build systems. PyCharm is available on Windows, Mac, and many Linux distributions.

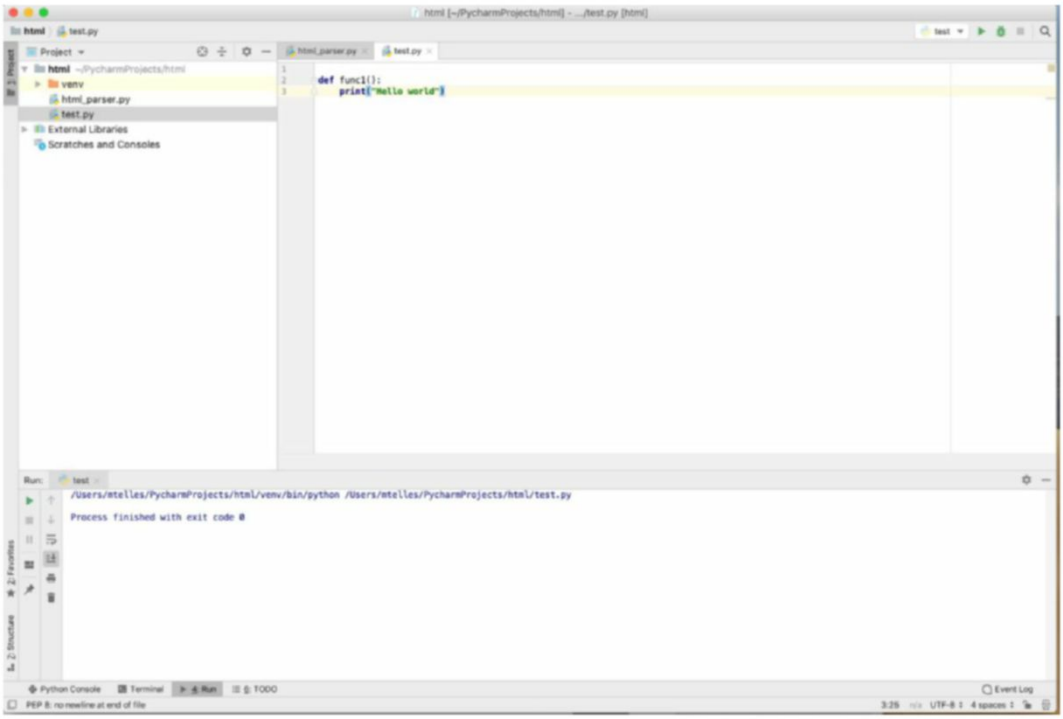

Figure 1.4: *PyCharm IDE*

Visual Code, on the other hand, is a more general purpose code editor that can be extended and configured to use in most development languages. Visual Code is a stripped down version of the popular Visual Studio development environment from Microsoft, but it has been re-written to run on virtually any operating system. The Visual Code environment is almost infinitely extensible and has plug-in modules that allow you to connect to source code control systems, to build systems, to compile and run almost every language known to man. It has extensions that will allow you to syntax check your Python code, run it, debug it, and anything else you might imagine. Visual Code is freely available on most platforms, including Windows, Mac

and Linux. *Figure 1.5* shows the Visual Code environment with a new Python file created with code in it.

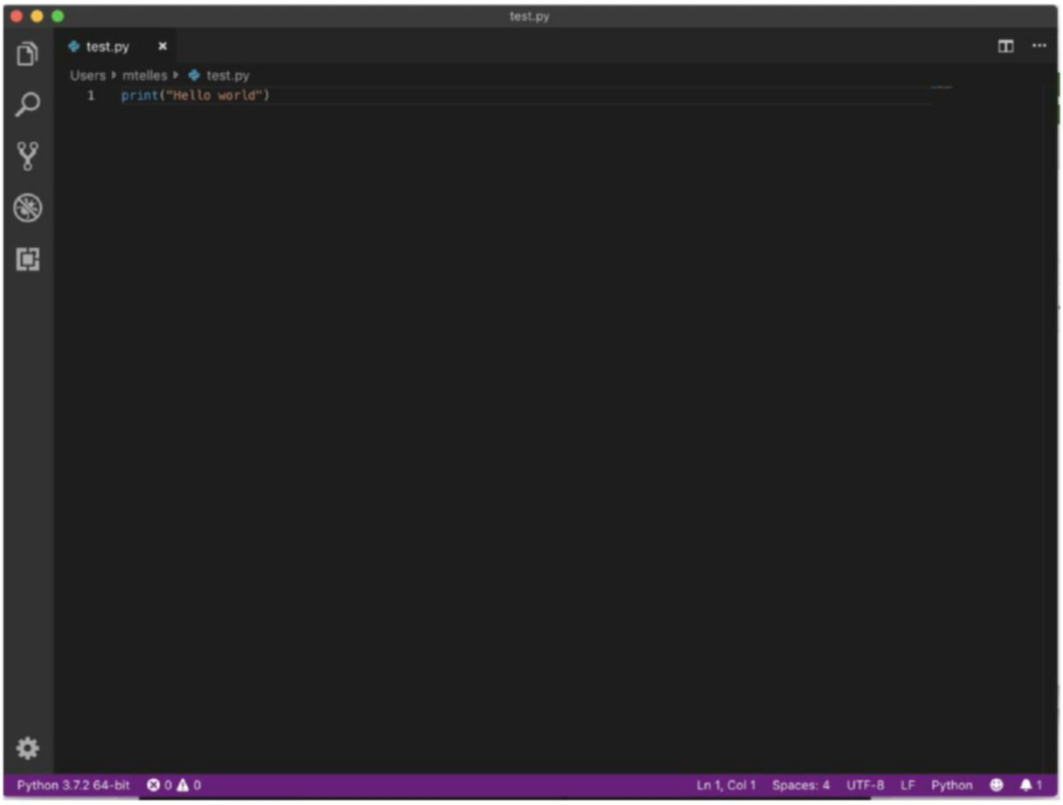

Figure 1.5: *The Visual Code IDE*

We will not specify an editor or IDE in this book; you are free to use anything you like. Individual chapter screen shots may show a particular IDE or editor, simply to illustrate what the code looks like in a nicely formatted and colored manner, but that does not indicate that you are in any way obligated to use that editor. Python is all about freedom for the developer, making it as easy as possible, while not restricting you in anything but syntax.

Hello world

No programming book worth its salt would be complete without an introductory *Hello world* exercise that shows you the simplest possible form of the language in a working example. In the Agile world, we call this a **Minimal Viable Product**, and for Python, it honestly could not be any simpler.

To create an MVP for our Python Hello World program, we first create a new file. In this example, the file is called `hello_world.py`, but you can call it anything you like. The extension `.py` is the default for Python files, but it isn't a real requirement. If you want to call it `hello_world.py3`, to indicate that it is a Python 3.6 file, that's up to you. The Python interpreter doesn't care what the file is called, or what the extension is, it is simply a convention that is used. If you call it something else, just substitute that in for the remainder of the exercise.

Within your new file, place the following line:

```
print("Hello world")
```

The information within the quotation marks is unimportant, it is a literal string, and the interpreter will just display whatever you put there. What does matter is that you enclose the literal string in parentheses. If you have previously used Python 2.x, this was not mandatory; this is a change for Python 3.x. Also important is that the print statement be in lowercase. Python is case sensitive. Finally, there should be no spaces before the print statement on a line in your file. Python is very sensitive to indentation, as we will see in the following chapters.

Run your first Python program by typing the following at a Command Prompt (Windows) or Terminal (Mac), or Shell window (Linux):

```
python3 hello_world.py
```

If all goes well, you should see the disk spin a bit and finally spit out:

```
Hello world
```

On Terminal immediately below your command. The whole session should look something like this:

```
python3 test.py

Hello world
```

Congratulations! You have written your first Python program!

By the way, if something goes wrong, such as seeing an error like `python3 not found`, verify that the path to the Python application is in your path. If you installed Python from this terminal window, it may not have picked up changes to your path. The easiest solution is to open a new terminal window and try it again.

Conclusion

In this chapter, you learned how to install the Python system and how to select a specific version of it to write your application code. You should have learned about pip, the Python package installer, as well as the IDE's available and the immediate mode interpreter. Hopefully, you picked up some of the principles that make up Python development, and some of the *rules* that Pythonistas use when writing code.

In our next chapter we'll begin writing real code, introducing you to the types and constructs that make up the Python programming language.

Questions

1. How long has Python been in existence?
2. What are the two major versions of Python available?
3. What is a virtual environment?
4. How do you run a Python script file from the command line?

Chapter 2
Python Types and Constructs

Introduction

Every discussion of programming begins with looking at the types of values that you can store, along with the ways in which you can work with those variables. Python, and this book, is no exception to the rule. In this chapter, we'll start out by looking at the various types of data that can be used in Python, and how you can use them. We'll explore how to convert from one to another, how to build enumerable types such as lists and sets, and how Python expects things to be created.

Data types, of course, are the building blocks that make up a language. Surprisingly, Python has fewer of them than most, because it does much more with them.

Structure

- Data types
- Iterators and iterables
- Classes and objects
- Booleans and truthyness

Objectives

In this chapter, you will learn about the various data types in Python, from simple types like integers and floats to more complex types like classes and collections. You will explore the concept of iterators and iterables, of booleans and truthyness, a truly Pythonesque concept. We will talk a little about complex numbers, a difficult math concept made simple in Python. Finally, we'll talk about that standard of programming, commenting.

Integers

The most basic numeric type is the integer. Integers are whole numbers with no fractional portion. In most languages, there are numerous types of integers, representing very small numbers, like bytes, up to very large numbers, like long integers, or even long long values. Instead, Python uses a single integer type, which is fixed precision, allowing it to store any value up to available machine memory. That is to say, we can have a single variable, `myInt`, which can be used to store:

```
myInt = 1
```

```
myInt = 12345678901234567890123456789
```

We can show this using the Python interactive tool:

```
>>> myInt = 12345678901234567890123456789
```

```
>>> print(myInt)
```

```
12345678901234567890123456789
```

```
>>> myInt = 12345678901234567890123456789919823198273198273198273982739
81273987129837129837129837981273981273981273981273981723987129837129837
2913
```

```
>>> print(myInt)
```

```
12345678901234567890123456789919823198273198273198273982739812739871298
71298371298379812739812739812739812739817239871298371298372913
```

As you can see, there really are no limits for integer values. There are limits on things like the maximum number of entries in an enumerable, this value can be discovered by using the `sys.maxsize` constant. To get at Python constants like this, we use the `import` statement:

```
import sys
```

```
print(sys.maxsize)
```

Running this on a Mac gives the following output:

```
9223372036854775807
```

As you can see this is a pretty big number. Unlike Java, or C++, there is no concern about converting smaller integer value variables into larger ones, or vice versa:

```
i=1
i=42000000000000
```

As you can see, it works the same way either way.

As with all languages, integers can be added, multiplied, divided and subtracted, and you will mostly get what you expect. Using the interactive interpreter we can see this:

```
>>> x = 1
>>> y = 2
>>> print(x+y)
3
>>> print(y-x)
1
>>> print(x*y)
2
```

There is a single exception, however, and that is in division.

```
>>> print(x/y)
0.5
```

Dividing two integers in most languages will give you the natural integer result. For example, dividing 1 by 2 in C++ or Java will give you `0`, since the result is a fraction less than one, and fractions cannot be represented in integer format. If you want to do integer division, such as finding out whether or not a number is odd, use the `//` operator:

```
>>> print(x//y)
0
```

The `//` operator is also called the *floor division* operator. It works in conjunction with the modulus operator, `%`, which returns the remainder of the division of two integers:

```
>>> x = 42
>>> y = 5
>>> print(x%y)
2
>>> print(x//y)
8
```

Finally, we have the power operator, which raises a given number to a given power. In math, we might write `2^5`, to indicate the value `2` multiplied by itself `5` times:

```
2^5 = 2*2*2*2*2 = 32
```

In Python, we would write this as:

```
>>> print(2**5)
32
```

Naturally, Python supports assignment. You can assign a constant to a variable:

```
x = 1
```

Or you can assign another variable to a given variable:

```
y= 1
x= y
print(x)
print(y)
```

If you run this in the interactive interpreter, you will see that both `x` and `y` have the value of `1`.

In addition to the simple operations Python also supports short-hand notation for most of the operators. For example, we can use the += operator to add a value to a given variable:

```
>>> x = 10
>>> x += 5
>>> print(x)
15
```

In addition, there are versions of the short-hand operators for subtraction (`-=`), division (`/=`), multiplication (`*=`), power, modulus, and floor division (`**=`, `%=`, and `//=` respectively):

```
>>> x = 10
>>> x **= 2
>>> print(x)
100
>>> x /= 10
>>> print(x)
10
>>> x //= 3
>>> print(x)
```

```
3
>>> x %= 2
>>> print(x)
1
```

If you happen to read older Python books, or online documentation, you may see a type called long, which represents an arbitrary fixed point value. The long value has been subsumed into the integer type in Python 3, although you can still use it if you want to:

```
x=long(10000)
```

Aside from the creation statement, the long type is simply an integer under the covers and can be treated like one using all operators listed above. Finally, there is a function to convert a value to integers, called int. You can use it with floating point numbers like this:

```
>>> x=int(5.6)
>>> print(x)
5
```

The `int()` function can also be used to convert a string to an integer:

```
>>> print(int('5'))
5
```

The function, however, cannot be used with non-integer strings:

```
>>> print(int('5.0'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '5.0'
```

Floating point numbers

A float point number is much like an integer in Python, with the exception of using floating point math, rather than integer math. That is, dividing `10` by `0` gives you `2.5` as a floating point value. Any integer value automatically becomes a floating point number if the decimal point is included. We can verify this using the built-in Python type function, which tells you what type of value something is:

```
>>> x = 1
>>> print(type(x))
<type 'int'>
>>> x = 1.0
```

```
>>> print(type(x))
<type 'float'>
```

Floating point numbers, of course, can be added together, multiplied together, divided by each other and so on and so forth. In addition, if you use an integer value in any of the operations, it will be *promoted* to a float and the result will be a floating point value:

```
>>> print(x * 5)
5.0
```

The usual operators (`+`, `-`, `+=`, `-=`, and so forth) can be used with floating point numbers, which is no surprise. What might be a surprise is that you can use the modulus operator with floating point number:

```
>>> x = 5.0
>>> x %= 3
>>> print(x)
2.0
```

This isn't something you see in a lot of languages, although the mathematical world has accepted functions like this for all time. This is another part of the 'no surprises' aspect of Python, things generally work the way you expect them to, even when you aren't thinking about it.

Another thing that might surprise you is that floating point numbers can be in both parts of exponential equations. For example, I can raise a number to the one-half power (which happens to be the square root):

```
>>> print(25**0.5)
5.0
```

Likewise, you can raise a floating point number to an exponential power:

```
>>> print(2.5**2)
6.25
```

It is probably worth mentioning, at this point, that Python follows the standard order of precedence for operators, sometimes called **PEMDAS (Parentheses, Exponents, Multiplication, Division, Addition, and Subtraction).** So, if you write this:

```
>>> x = 5*2**3
```

You might expect `x` to be `1000`, since `5*2` is `10`, and `10` raised to the third power is `1000`. However, when we apply the order of precedence, it is as if we re-wrote this equation as:

```
x=5*(2**3)
```

Raising `2` to the third power gives us `8`, and multiplying that by `5` gives us the expected answer of 40:

```
>>> print(x)
40
```

There is no penalty to putting parentheses around expressions if you aren't sure of what the order is, so feel free to do so. For example:

```
>>> 5/4-4
-3
```

If you re-wrote this as `5/(4-4)`, you'd get ..

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Python is kind enough to point out to us that we tried to divide by zero and generates an error, called an **exception**. We'll learn all about exceptions and exception handling later in the book.

Finally, like integers, there is a `float` function that will convert given inputs into floating point values to be stored in variables:

```
>>> x=float(1)
>>> print(x)
1.0
>>> x=float("1")
>>> print(x)
1.0
>>> x=float("1.5")
>>> print(x)
1.5
```

Unlike the int function, float does accept integers as well as floating points in strings.

Boolean

In Python, as with virtually all programming languages, the notion of Boolean values means true or false. Python does have an actual `Boolean` type, which you can assign `True` or `False` to. Note that in Python, unlike many languages, the values are actually capitalized. Thus, you can write:

```
x = True
```

```
But not:

x = true
```

Writing the former will be accepted by the interpreter, writing the latter will generate an error message.

You don't have to use the `True` and `False` values, if you really don't want to. As we will see in just a little bit, the Python notion of *truthy-ness* and *falsy-ness* is extended to handle numeric and string values, not to mention the infamous None.

To get slightly ahead of ourselves, but to get an idea of how Python handles Booleans:

```
>>> x = True
>>> print(x)
True
>>> if x:
...    print("Yes")
... else:
...    print("no")
```

Don't worry if you don't completely understand this block, it is a conditional statement in Python that uses the shorthand for `True` as if `x`. The output from this block is:

```
Yes
```

Complex values

It is an odd concept, but Python natively supports complex numbers. If you don't know what a complex number is, chances are you have never needed to use one. This wouldn't be terribly surprising, as complex numbers are an odd branch of `math`. Complex numbers have two parts to them, a *real* part and an *imaginary* part.

If you don't have any need to work with complex numbers, feel free to skip this section. If you do use them, such as in electronics work, you should continue reading. Most of the work in complex numbers are straightforward. You create one using the `complex()` operator:

```
>>> x=complex(2,3)
>>> print(x)
(2+3j)
>>> y=complex(3,4)
>>> print(y)
```

```
(3+4j)
>>> print(x+y)
(5+7j)
```

Likewise, one can add, subtract, multiply and divide complex numbers, in accordance with the rules of math. Also in accordance with `math`, you can multiply and divide a complex number by a constant value. For example:

```
>>> x=complex(1.0, 2.0)
>>> print(x)
(1+2j)
>>> x *= 2
>>> print(x)
(2+4j)
>>> print(x/2)
(1+2j)
```

In a nice little touch, Python also allows you to define a complex number without using the `complex()` operator, you can write it as if you were writing a normal `math` equation:

```
>>> x = 3+4j
>>> print(x)
(3+4j)
```

You can extract the real and imaginary portions of a complex number using the .real and `.imag` properties of the complex number:

```
>>> x = 3+5j
>>> print(x.real)
3.0
>>> print(x.imag)
5.0
```

If you are accustomed to languages like C++ or Java, where you must implement your own complex variable type, or use someone else's, it is certainly nice to see it done for you with all of the proper work and testing done before it ever gets to you.

Variable naming

In some languages, naming of variables is a hotly contested thing. Hungarian naming conventions, extended naming, naming for different variables in functions

or classes instead of in global code, all of these things become religious wars in the programming community. Python is a lot less hung up on such things. There are, however, a few conventions and rules that not only make sense, they actually have to be followed.

Python has the following absolute rules for variables:

- Variable names are case sensitive. Thus x and X are different variables
- The first character in a Python variable name must be a letter. Once past the first letter, you can use any combination of letters, upper or lower case, as well as numbers, or the underscore character. A note here: there are special cases where a variable should begin with an underscore.
- Variable names can be almost any length, but a single line in Python should only be 79 characters or less, so that's the practical limit
- Variables may not be reserved words like if, for, range, or while.

In terms of recommended naming conventions:

- Python programmers lean toward meaningful names, instead of short meaningless ones.
- Variables should be in snake case (that is, `this_is_a_variable`) rather than Camel case `thisIsAVariable`, something more common to Java.
- Variable naming should be consistent across modules. If you call something `water_temperature`, then use `water_clarity` in the same module, rather than clarity.

Some notes on variables in Python.

- There is no *declaration* of variables in Python, nor any type definition. A variable exists from the point at which it is assigned a value.
- Variables can be chained together in assignments:

```
a = b = c = 0

f1 = f2 = f3 = 0.0

nil_value = None
```

- The special value `None` indicates a variable without a value, it is akin to the `NULL` type in C++ or null in Java.

Strings

The next logical data type to consider is the string. Strings are one of the most important data types in any programming language, since it is via the string that we normally communicate with the user. Prompts to input, headers for outputs, error messages and generalized reports all require the string type. A string, of course, is simply a collection of characters. In many ways, strings in Python really are

collections, which we will talk about next. However, they also have a few special considerations that make them worth talking about individually.

Probably the most important aspect of strings to the Python programmer is that they are immutable. Immutable, if you are unfamiliar with the term, means that they cannot be changed. In many languages, the string type is mutable. For example, in C++, we can do something like this:

```
std::string my_string;

my_string = "This is a test";

my_string[2] = 'a';
```

Which makes the final version of `my_string` equal to `Thas is a test`. This is perfectly fine and the compiler will not complain in the least. In C# and Java, however, trying to do this will result in an error which explains that the `operator[]` is read-only. Not terribly useful, but it is accurate. These languages support immutable strings. To see what we are talking about, let's look at a Python string.

```
>> s = "Hello world"

>>> s[1] = 'a'

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment
```

As you can see, we can't assign to a single character position of a Python string. Does that mean we can't change them? No, that's not the case at all. Python has what is called *slicing*. In most languages, this is called a *substring*, but in Python it is a bit more powerful. The slicing operator in Python looks like this:

```
s = "Hello world"

print(s[2]) -- This prints 'l'
```

But we can rebuild a string from slices:

```
s=s[:1]+'k'+s[2:]

>>> print(s)

Hkllo world
```

Note that we have *modified* the string s. Of course, we've actually done nothing of the sort, because strings are immutable. Instead, we have created a brand new string out of the pieces of the original and a new character, and then assigned that string to the original variable.

The slicing operator, though, has a lot more power than that. As you can see in the above example, you aren't required to fill in the start and end arguments, letting us use `[:2]` which means 'start at the beginning of the string and go until the second

position'. That's just the start. There's an optional third parameter that allows you to specify a step count. For example, if we do this:

```
>>> s = "Hello world"
>>> print(s[0:len(s):2])
```

The output from this command looks like this:

```
Hlowrd
```

Note that the `len()` operator works with any collection, it returns the number of elements in the collection. A string is just a collection of characters, so the number of elements is the number of characters in the string.

There is, of course, a `str` operator that will convert something to a string:

```
>>> x = 1
>>> s = str(x)
>>> print(s)
1
>>> print(len(s))
1
```

As you can see, the result of the `str()` is a string. If you don't believe it, we can verify this in two ways. First, we can check the type of the variable:

```
>>> print(type(s))
<class 'str'>
```

Secondly, we can verify that you can't find the length of an `int`:

```
>>> print(len(x))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Finding substrings

A string is a kind of collection, and collections need the ability to search for things. Python supports searching strings in two ways. First, there is a method on the string object called `find`. Find is much like the C# or Java string index operator, it returns the position of the string you supply within the string you want to search. For example:

```
s = "This is a test"
>>> print(s.find('is'))
2
```

The find method indicates the index, zero based, where it find the first occurrence of a string. If there are multiple strings that match, only the first will be returned. If no substring requested could be found, the result is `-1`, which is also much like its other language counterparts.

```
>>> print(s.find('xyzzy'))
-1
```

This is not really a Pythonic way of doing things, however, particularly if all you want to know is whether or not the substring exists within the parent. Imagine, for example, that we are parsing some sort of known stock ticker line and all we want to know is whether or not it contains the stock symbol we are looking for. In this example, we'll imagine that the stock symbol is for Apple Computers, AAPL.

We could do this:

```
if s.find('AAPL') != -1:
    do_something_with_the_ticker_line(s)
```

This is complicated though, and fraught with error potential. What if someone decides down the line to change the signal value returned to None? This would actually make a great deal of sense. We don't want to know where it is in the string, just whether or not it is there. For this, Python provides the 'in' operator:

```
>>> s = 'Stock Symbol: AAPL|Ticker Value: 123.45|Volume: 123123123'
>>> print ('AAPL' in s)
True
```

Just to show you that it works properly, consider the following:

```
>>> s = 'Stock Symbol: AAPL|Ticker Value: 123.45|Volume: 123123123'
>>> print ('AAPL' in s)
True
```

The `in` operator works for all sorts of collections, including strings, lists and dictionaries, which we will see shortly. You can also use the `not in` operator to see if a string is not contained within a parent string:

```
>>> print('IBM' in s)
False
>>> print('IBM' not in s)
True
```

This approach is not only more readable, but less error prone and thus more Pythonic. As you see, Python is all about making things easier for developers now and in the future.

Multiple line strings

As you've seen, one of the tenets of Python is to keep lines readable. In fact, one of the Python principles is to never have a line more than 79 characters. This limitation, by the way, comes from the good old days of terminals, which could only display 80 characters across. While modern displays with high resolution monitors can show many more characters than that, it is still a good idea not to make your lines overly long, since people don't always remember to scroll to the right to see the whole thing.

There is no error generated if you create a line more than 80 characters, as there might have been in COBOL or FORTRAN, it is simply frowned upon. Good Python checkers will issue a warning if you have a line that extends beyond the limit.

Sadly, the modern world always intrudes on the best laid plans of programmers. We often have error messages that are more than 79 characters. How are we to put these into our code? We could do something clever, and store them in a file, only to read them and display them, but that causes other issues. Instead, we can create multiple line strings. There are two ways to do this:

```
>>> s = "This is a test of the emergency broadcast system" \
... " and it will display an emergency if you put a line of text" \
... " that is more than 80 characters on a single line"
>>> print(s)
This is a test of the emergency broadcast system and it will display an emergency if you put a line of text that is more than 80 characters on a single line
```

The backslash character \ tells the Python interpreter that you are continuing a line. Normally, a single statement must be contained in a single line, aside from blocks, which we have briefly looked at in if statements.

There's another way to do it, however, which is easier to read and therefore considered more Pythonic. The triple quote construct looks like this:

```
>>> s1 = """ This is a test of the emergency broadcast system.
...  and it will display an emergency if you put a line of text
...  that is more than 80 characters on a single line
... """
>>> print(s1)
 This is a test of the emergency broadcast system.
 and it will display an emergency if you put a line of text
```

```
that is more than 80 characters on a single line
```

As you can see, the triple quote is easier to read than the backslash and is generally preferred.

Concatenating

It is not at all uncommon to need to concatenate two strings together, for output purposes, or to build a message for logging, or simply to check against some other string. Python uses the natural + symbol to concatenate strings:

```
>>> s = "Hello"
>>> s1 = "world"
>>> print(s+s1)
Helloworld
```

Of course, that output is kind of ugly. We need a little white space in the middle:

```
>>> print(s+" "+s1)
Hello world
```

As you can see, a string literal is treated exactly the same way as a string variable, which is consistent with Python treating all kinds of objects equally.

Other methods

Strings are one of the most heavily used types in programming, so it is of little surprise that Python provides a wide array of functionality based around them. As of Python3, the string type supports both ASCII and UNICODE strings, so that foreign languages, characters with accent marks and multi-byte strings are all supported. Here is a list of some of the methods that you can use on a string:

Method	Purpose	Example
`capitalize`	Capitalizes the first letter of the string	`print(name.capitalize()):` This prints a string `matt` as `Matt`.
`lower`	Returns the string in lower case. Does not modify the existing string	`s= "THIS IS A TEST` `print(s.lower()):` This prints `this is a test.`
`upper`	Returns the string in upper case. Does not modify the existing string.	`s= "this is a test"` `print(s.upper()):` This prints `THIS IS A TEST.`
`center, ljust, rjust`	Returns a justified string using the formatting requested.	`print('test',10) prints test .`

`startswith, endswith`	Returns true if the string begins or ends with the requested substring	`print("This is a test". startswith("This"))`: This prints `True`
`replace`	Replaces a given substring with a replacement substring	`print('this'.replace('is', 'was'))`: This prints `thwas`.
`split`	Returns a list of words split by a given separator. The default is space	`s="This is a test"` `print(s.split())`: This prints (`'This', 'is', 'a', 'test'`).
`strip, rstrip, lstrip`	Trims spaces from either the front and back of a string, or just the back or front, depending on the version	`s=" . This is a test . "` `print(s.strip())`: This prints `This is a test` with no leading or trailing spaces.

`find, rfind`	Returns the position of the first, or last, position of a substring within a string, or –1 if it can't be found	`s="test the test"`
		`s.find('test') returns 0`
		`s.rfind('test') returns 9`

As you can see, there are a lot of functions built into the string library! Finally, as a note, you can chain different methods together to form a simple function:

```
>>> s = "ThIs Is A tEsT"
>>> s.lower().find('is')
2
```

Python allows you to chain together methods, so long as each one returns an object of the correct type. You cannot chain together a method that returns a different type using methods on the original object. For example, find returns an integer value, so you can't do this:

```
>>> s.find('is').lower()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'lower'
```

When we begin to talk about writing our own types and classes, we'll look at how you can make all of this possible for your own functionality.

Finally, let's discuss the join function for strings. Here's how join works. If you have a string, you can Imagine for a moment that you have an array of strings. Python doesn't have arrays, it has lists, which we will look at next. However, most programmers of any experience understand arrays, so we'll use those for right now.

```
array = ["a", "b", "c"]
```

You want to print the array of strings out as a single string, essentially flattening it into a single string. Here's what we want to see: `abc`. If we print the array, though, we'll see `[a,b,c]`. The join method makes it possible to do this:

```
>>> array = ["a","b","c"]
>>> print(array)
['a', 'b', 'c']
>>> print(''.join(array))
abc
```

But wait! There's more, as the commercials are all so fond of saying. The string at the beginning is used to concatenate all of the pieces of the array, so we can do this:

```
>>> print('+'.join(array))
a+b+c
```

Pretty cool, isn't it? Oh, one final note on strings. You may see, in this book and in code around the Internet, the usage of either the single quote (') or double quote (") surrounding a string. This is perfectly normal, Python considers them the same. The only difference is that if you have a string that contains one of those quote types, you can only encapsulate it in the other type:

```
s= 'This is a "test"'
```

And

```
s= "This ain't a test"
```

Python Collections

As experienced programmers know, variables are only the tip of the iceberg when it comes to writing code. You can't create an applications by assigning things to variables, you have to actually do something with them. The first step to doing something with the data is to arrange it into data structures that can be manipulated in groups. Python has a variety of ways to group data. These groups are referred to as collections, or sometimes iterables. The latter name comes from the notion that you normally need to step through the individual elements of a given collection, a process known as iterating.

Lists

It might surprise you to know that Python has no concept of an *array*, which is a central construct in languages like Java or C++ or C#. Lists are the closest thing that Python has to an array, and you will find that the syntax of a list is very similar to that of an array.

Here's how you define a Python list:

```
my_list = []
```

This statement, using square brackets, creates an empty list. Likewise, you can create a list of elements by placing them inside of the brackets:

```
my_list = [1,2,3]
```

This statement creates a list with three values, the integers 1, 2, and 3. This likely looks normal to you, if you are coming from Java or C++. It is just an array with three elements. But could you do something like this in Java?

```
>>> my_list = [1,2.5,"Hello world"]
>>> print(my_list)
[1, 2.5, 'Hello world']
```

Python does not require that lists be homogeneous. You can put any sort of data into a list, and Python will allow it. In the above example, we have a list that contains an integer value, a float, and a string. You can apply list functionality to this list:

```
len(list)
```

You can even test to see if a list contains a value of a specific type, even if the list is not homogeneous:

```
>>> print(2.5 in my_list)
True
```

You might wonder if you can apply the usual arithmetic operators to lists, and the answer is, of course you can! It just doesn't always work the way you would expect:

```
>>> my_list += 'fred'
>>> print(my_list)
[1, 2.5, 'Hello world', 'f', 'r', 'e', 'd']
```

As you can see, we expected it to add the string `fred` to our list, but instead, it added each of the characters to the list. Why would it do this? Because strings are collections. When you add one collection to another, you add all of the elements individually to the target collections. But wait, you cry! What if you want to add the string to the collection as a single element. Well, you do this by creating a list containing the single element that is the string:

```
>>> my_list = [1,2,3,4]
>>> my_list_1 = ['fred']
>>> print(my_list + my_list_1)
[1, 2, 3, 4, 'fred']
```

You can kind of multiple lists, and while it does exactly what you would expect it to do, it probably isn't as useful as you might think:

```
>>> my_list = [1,2]
>>> print(my_list*3)
[1, 2, 1, 2, 1, 2]
```

You cannot, however, subtract one list from another, at least not by using the minus sign. You cannot divide lists either. You can `remove` elements from a list:

```
>>> my_list = [1,2,3,4]
>>> my_list.remove(3)
>>> print(my_list)
[1, 2, 4]
```

An important point here is that the `remove` operator removes the first element it finds that matches, not all of them:

```
>>> my_list = [1,1,2,2,3,3,4]
>>> my_list.remove(2)
>>> print(my_list)
[1, 1, 2, 3, 3, 4]
```

The remove function removes elements by value. To remove them by their position (index) in the list, use the `del` function:

```
>> my_list = [1,1,2,2,3,3,4]
>>> del my_list[1]
>>> print(my_list)
[1, 2, 2, 3, 3, 4]
```

Lists, unlike strings, are mutable. You can add to them, remove from them, even modify them in place. To add a new entry to a list, use the `append` method:

```
>>> my_list = [1,2,3]
>>> my_list.append(4)
>>> print(my_list)
[1, 2, 3, 4]
```

If you prefer to add something other than at the end, use the method:

```
>>> my_list.insert(0, 5)
>>> print(my_list)
```

```
[5, 1, 2, 3, 4]
```

Again, because lists are mutable, you can change a value in place:

```
>>> my_list[1] = 6
>>> print(my_list)
[5, 6, 2, 3, 4]
```

As always, lists are like Java, C# or C++ arrays, they are zero based. So the first element is at position zero, the second at position `1`, and so forth up to the length of the list minus one. To find the length of a list, use the same `len` operator as for strings.

```
>>> print(len(my_list))
5
```

Finally, before we move on to the next topic, let's take a look at a uniquely Pythonic concept. If you have a list, but what you want is just the values within the list, such as when you are passing values to a function, you can use the unpacking operator (*). This isn't always clear, and we'll examine it in more detail when we look at functions, but take a look at the difference between the two outputs here:

```
>>> print(my_list)
[5, 6, 2, 3, 4]
>>> print(*my_list)
5 6 2 3 4
```

To show you when you'd use this, we'll get a little bit ahead of ourselves, but you'll see how it works. Suppose we enter this into the immediate interpreter:

```
>>> def func(arg1, arg2, arg3):
...     print(arg1)
...     print(arg2)
...     print(arg3)
...
>>> l = [1,2,3]
```

Here, we have created a function that takes three arguments; arg1, arg2, and arg3. We've also created a list that has three elements. You cannot do this:

```
>>> func(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: func() missing 2 required positional arguments: 'arg2' and
'arg3'
>>> func(*l)
```

As the Python interpreter informs you, the function expects three arguments, not a list. But thanks to the wonders of unpacking, you can do this:

```
>>> func(*l)
1
2
3
```

If you run across the unpacking operator, now you will at least understand what it is trying to do.

Dictionaries

A dictionary is simply a container of name and value pairs. Most languages have a variation of the dictionary, although few have it actually built into the language. In C++ and Java, forexample, they are often called **Hash Sets**, because that's how they are implemented. Python prefers to call things what they are, instead of worrying about the underlying implementation.

Although dictionaries are very powerful, the one important thing to remember about them is that they are unordered. That is, if I add a batch of values to a dictionary, there is absolutely no guarantee that when I iterate through the object I will get them back in the same order. Normally, that isn't the use case of a dictionary, so that isn't a problem. The other important thing to remember, which the basic functionality of a dictionary is, is that they cannot contain multiple keys of the same value. Each key must be unique.

The basic setup of a dictionary in Python looks like this:

```
>>> d = {
... "key1": "value1",
... "key2": "value2",
... "key3": 3
... }
>>> print(d)
{'key1': 'value1', 'key2': 'value2', 'key3': 3}
```

You will notice that you don't have to have values of the same type, as shown by `key3`.

There is absolutely no requirement that the key be a string either, although typically, we use them that way. Consider the following:

```
>>> d1 = {
... 1: "This is a test",
... 2: "This is another test",
... 3.5: "This is the last test"
... }
>>> print(d1)
{1: 'This is a test', 2: 'This is another test', 3.5: 'This is the last test'}
```

The only requirement for a key in a dictionary is that the key must be of an immutable type, such as string, numeric, or boolean. It cannot be a list, as that is a mutable type.

Getting the value of a key

Creating a dictionary is all well and good, but clearly, what we want is to be able to get back the value of a given key in the dictionary. This is fairly simply in Python, we just use the `[]` operator with the key name we are interested in. Given the above dictionary, suppose that we want to get back the value of `2` as a key:

```
>>> print(d1[2])
This is another test
```

You don't have to know the type of the value, although if you want to use it that will become a necessity.

Testing if a key is in a dictionary

The problem with a dictionary is that you might think there is a key in it that isn't, in fact, there. For example, consider the following example using the dictionary we defined previously:

```
>>> print(d1[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
```

This error will stop your program at this point with an exception. You can either handle the exception, which is slow if you are doing this over and over, or you can verify that the key you are looking for is in the dictionary to begin with. For this purpose, we use, what else, the in operator:

```
>>> print(3 in d1)
False
```

We will look at how to deal with such cases when we take up the conditional options in Python.

Iterating

The normal use case for a dictionary is to load it with keys that are assigned values, and then look up those values at run time for various scenarios. For example, we might want to look up the value of a given configuration option stored in a dictionary. However, there are times when we want to walk through a dictionary and look at all of the keys and values. We can do this in numerous ways. Let's look at three of them.

First, we can iterate over the keys as the default of the dictionary:

```
>>> d = {
... "key1": "value1",
... "key2": "value2",
... "key3": "value3"
... }
>>> print(d)
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
>>> for x in d:
...     print(x)
key1
key2
key3
```

We can also print out the values of the dictionary, taking advantage of the fact that they dictionary look up uses the key. In fact, we can print out both in the same iteration:

```
>>> for x in d:
...     print("Key: " + x + " Value: " + d[x])
...
Key: key1 Value: value1
Key: key2 Value: value2
Key: key3 Value: value3
```

Finally, we can use the items property of the dictionary, which returns us both the key and value in a single statement:

```
>>> for k, v in d.items():
...    print("Key: " + k + " Value: " + v )
...
Key: key1 Value: value1
Key: key2 Value: value2
Key: key3 Value: value3
```

Length of a dictionary

The length of a dictionary is the number of keys in the dictionary, which is returned by the `len()` function, just like strings and lists. Using the dictionary we defined above, that would be represented as follows:

```
>>> print(len(d))
3
```

Adding new items

Dictionaries are mutable, you can add or remove items, or even modify existing items in the dictionary. Modifying a dictionary is done in exactly the same fashion as retrieving the value:

```
>>> d['key1'] = 'valuenew'
>>> print(d)
{'key1': 'valuenew', 'key2': 'value2', 'key3': 'value3'}
```

Somewhat surprisingly, the exact same method is used for adding a new value to the dictionary.

```
>>> d['key4'] = 'value4'
>>> print(d)
{'key1': 'valuenew', 'key2': 'value2', 'key3': 'value3', 'key4': 'value4'}
```

Of course, you would want to check to see if a value existed if you were worried about overwriting an existing entry, since there is no indication that the entry changed, since assignment and addition are exactly the same.

You can delete entries from a dictionary as well. Much like lists, you can use the `del` function, or you can use the `pop` function. Both work for either dictionaries or lists:

```
>>> del d['key1']
>>> print(d)
```

```
{'key2': 'value2', 'key3': 'value3', 'key4': 'value4'}
Or:
>>> d.pop('key4')
'value4'
>>> print(d)
{'key2': 'value2', 'key3': 'value3'}
```

Note that unlike `del()`, the `pop` function will return you the item you are removing from the dictionary. In either case, you can only delete the entry by key, not by value. To delete items by value, you'd need to iterate over the dictionary and find all of the keys that had that value and then remove them one at a time.

It is a BAD IDEA to delete items in any sort of iteration over a collection in Python.

Nested dictionaries

As mentioned earlier, both the key and value sides of a dictionary can be any valid Python type, so long as it is hashable (which is basically all types). Let's consider a more real world example. Suppose, for example, that we want to store some configuration information for our application. However, our application runs in different environments. We'll call the environments `production_1, production_2` and `production_3`.

In each of the environments, we want to be able to store things like the number of web servers, the name of the environment that we want to display for users, and perhaps the maximum number of users that the environment can support. We could do this by having a different dictionary for each environment, and then selecting the one we want based on the name of the specific environment that we are trying to get information about. Alternatively, we could take advantage of the fact that dictionaries can be nested. Let's look at how that might be done.

We could do it this way:

```
product_1_dictionary = {
    "number_of_web_servers": 3,
    "name_to_display": "Production Server East",
    "maximum_number_of_users": 100
}


product_2_dictionary = {
    "number_of_web_servers": 5,
```

```
    "name_to_display": "Production Server West",
    "maximum_number_of_users": 500
}

product_3_dictionary = {
    "number_of_web_servers": 1,
    "name_to_display": "Demo Server",
    "maximum_number_of_users": 5
}

production_dictionary = {
    "production_1": product_1_dictionary,
    "production_2": product_2_dictionary,
    "production_3": product_3_dictionary
}

environment = "production_1"
print(production_dictionary[environment]['name_to_display'])
```

We assemble each dictionary, then create a master dictionary that stores each one of them, keyed to the name we specified above (`production_1`, and more).

This works the way you expect, the output is `Production Server East` for the `print` statement at the bottom. Alternatively, we could do it all in one place:

```
production_dictionary_a = {
    "production_1": {
        "number_of_web_servers": 3,
        "name_to_display": "Production Server East",
        "maximum_number_of_users": 100
    },
    "production_2": {
        "number_of_web_servers": 5,
        "name_to_display": "Production Server West",
        "maximum_number_of_users": 500
    },
```

```
    "production_3":  {
        "number_of_web_servers": 1,
        "name_to_display": "Demo Server",
        "maximum_number_of_users": 5
    }
}
print(production_dictionary_a[environment]['name_to_display'])
```

This works as well and prints out the same value. It is slightly more compact as well, and easier to read since it is in one place. Please note that you need to have commas after each block of the dictionary, to indicate to the interpreter that this is a separate piece.

Sets

Like dictionaries, sets are unordered collections of data in Python. Unlike dictionaries, they do not contain keys, but instead lists of values. In this, they are very much like the list data type. In fact, a set is a combination of the dictionary type and the list type. It contains a list of values, but the values must be unique. You can have a list that looks like this:

```
list = [1,2,3,4,3,2,1]
print(list)
```

Printing this out gives us what we have come to expect:

```
[1, 2, 3, 4, 3, 2, 1]
```

A set, however, is defined in a syntax that looks more like a dictionary and has a uniqueness property like a dictionary too:

```
set = {1,2,3,4,3,2,1}
print(set)
set([1, 2, 3, 4])
```

Notice the syntax of defining the set is the curly braces, but without the colon denoting keys and values.

You can also use the `set()` function to define a set:

```
s = set([1,2,3,4,5,4,3,2,1])
print(s)
set([1, 2, 3, 4, 5])
```

Notice however, that a set is created using a list of items (actually, any iterable). Also notice that a set is created without displaying an error for the duplicate values.

They are simply filtered out. There's nothing magical about creating a set from a list, either. Consider our previous dictionary example, made into a set:

```
s = set(production_dictionary)

print(s)

set(['production_1', 'production_2', 'production_3'])
```

Why do we care about sets? Let's look at how simple Python's functionality makes it to do really complex tasks. This is the beginning text of the *Declaration of Independence of the United States of America*:

```
text = """

The unanimous Declaration of the thirteen united States of America,

When in the Course of human events, it becomes necessary for one people

to dissolve the political bands which have connected them with another,

and to assume among the powers of the earth, the separate and equal station

to which the Laws of Nature and of Nature's God entitle them, a decent
respect

to the opinions of mankind requires that they should declare the causes
which

impel them to the separation.


We hold these truths to be self-evident, that all men are created equal,

that they are endowed by their Creator with certain unalienable Rights,
that among these are Life,

Liberty and the pursuit of Happiness.

"""
```

If we want to find out all of the individual words in the declaration, we can use the string split function to get back a list of them:

```
list_of_words = text.split()

print(list_of_words)
```

The output of this, clipped for space, looks like this:

```
['The', 'unanimous', 'Declaration', 'of', 'the', 'thirteen', 'united',
'States', 'of', 'America,', 'When', 'in', 'the', 'Course', 'of', 'human' .. ]
```

You will notice that the words are duplicated. There are an awful lot of `of` s in the text. What if we just want the individual words and don't want to see the duplicates? That's where the beauty of sets comes in. We can directly convert the list of duplicate words into a set of unique words in a single line:

```
set_of_words = set(list_of_words)

print(set_of_words)
```

The output of this single line, again abbreviated for space, gives us the unique list:

```
set(['and', 'among', 'all', 'have', 'people', 'pursuit', 'God', 'When',
'it', 'Liberty', 'one', 'America,' …]
```

If you don't like the way it is displayed, and would prefer to just see the words written out, we can do that in a single line too:

```
print(' '.join(set_of_words))

and among all have people pursuit God When it Liberty one America …
```

Now you can begin to see the true power of Python. Just a few more notes on sets. You can add new data to sets with the `add` function:

```
s = set([1,2,3,4,5,4,3,2,1])

print(s)

s.add(99)

print(s)

set([1, 2, 3, 4, 5])

set([1, 2, 99, 4, 5, 3])
```

Observe that the new entry appears randomly inserted into the new set.

You can also remove items from sets. Unlike lists, you needn't worry about whether you are removing all entries of a given value, since there is by definition only a single one. The removal is done via the `discard` function:

```
s = set([1,2,3,4,5])

s.discard(3)

print(s)

set([1, 2, 4, 5])
```

The union and intersection of sets provide the ability to `add` and `subtract` set contents. Union is the unique combination of two sets, and is done via the `|` (pipe) operator. If you are accustomed to working in Java, or C#, this will seem logical, as the `|` operator is the or operator in those languages. Essentially, the union of sets is the logical combination of values in one OR the other:

```
s1 = set([1,2,3,4,5])

s2 = set([6,7,8,9,10])

print(s1 | s2)

set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Likewise, the & (and) operator is used to determine the intersection of two sets. Intersection is really defined as the set of elements that are in both set one AND set two, so again, this makes sense.

```
s1 = set([1,2,3,4,5,6])

s2 = set([3,4,5,6,7,8])

print(s1 & s2)

set([3, 4, 5, 6])
```

The difference between two sets is the set of all values which are in the first set but not in the second. Not surprisingly, it is represented by the subtraction operator, the minus sign:

```
s1 = set([1,2,3,4,5,6])

s2 = set([3,4,5,6,7,8])

print(s1 - s2)

set([1, 2])
```

As with regular math, `s1 – s2` is not equal to `s2 – s1`, nor is it for sets:

```
print(s2-s1)

set([8, 7])
```

The very last set related thing is checking to see whether one set is a subset, or superset, of another set. A set which is a subset of another set contains items which are all in the superset. A set which is a superset contains all of the items of the subset, with potentially other items as well. Two sets which contain the same items are considered to be both subsets and supersets of each other. The subset notation is >=, while the superset notation is <=:

```
s1 = [1,2,3,4,5]

s2 = [2,3,4]

print(s1 <= s2)

print(s2 >= s1)

s3 = [2,3,4]

print(s2 >= s3)

print(s3 >= s2)
```

This snippet prints out:

```
True

True

True
```

```
True
```

Tuples

Our next Python type for your consideration is the tuple. A tuple is a set of data values that can't be modified once created. A tuple looks like this:

```
x=(1,2,3)
```

In spite of its name, which makes it sound as if it should have two values, a tuple can actually have as many values inside of it as you like. Tuples are very much like lists, except that they are immutable (meaning you can't change them, and that they can be the keys in dictionaries). You can only assign to a tuple once, so things like this don't work:

```
>>> x[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

You cannot delete an element from a tuple, nor add a new one to it. You can delete the entire tuple using the delete function, which erases the entire variable. Tuples are ordered, so that the order you define them in is what they will always exist as.

The primary purpose of tuples is to create data structures that can't be modified. You can use most of the list functionality on a tuple:

```
>>> print(x[::-1])
(3, 2, 1)
```

This example reverses a tuple's values when printing them. This doesn't change the actual variable, it simply returns a new tuple in reverse order.

As we will see when we get to functions and methods, tuples are mostly used to return multiple values from a function.

Because a tuple is an iterable sequence, you can get at individual pieces of it:

```
print(x[0])
1
```

And, in keeping with our discussion about sets, you can use a tuple to create a set, since it is a collection that is iterable:

```
>>> x=(1,2,3,4,5,6,5,4,3,2,1)
>>> print(x)
(1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1)
>>> s=set(x)
```

```
>>> print(s)
set([1, 2, 3, 4, 5, 6])
```

One last thing. Earlier, it was stated that a tuple could be created with any number of elements. If you try creating a tuple with a single element, you will find that this doesn't appear to be true:

```
>>> t=(1)
>>> print(t)
1
>>> print(type(t))
<type 'int'>
```

It can be done, however, there's just a slight trick to it:

```
>>> t=(1,)
>>> print(t)
(1,)
>>> print(type(t))
<type 'tuple'>
```

Iterators and iterables

As we have seen, Python's collection types all have similar functionality. The differences tend to be in the sort of data they contain. For example, a tuple and a list are the exact same thing from the outside, with the exception that you cannot modify a tuple (it is immutable). This means that all methods for the tuple are the same as that for the list, with the exception of the append and modify ones. There are quite a number of functions that can be used for collections in Python. Let's take a look at some of them here.

The `any()` method returns True if any single element in a collection is not `False`. We'll talk a little but about true and false later, but for now, just accept that anything that exists and is not zero is true. This isn't one hundred percent accurate, but it is close enough to understand the problem now.

To see how it works, let's start out with something very basic:

```
>>> x=[True,False,False]
>>> print(any(x))
True
>>> x=[False,False,False]
```

```
>>> print(any(x))
False
```

As you can see, the first list contains one `True` element and two `False` ones. The `any()` function checks all of the elements, stopping as soon as it finds a valid True element. In this case, the very first element is `True` and the function returns a true value. For the second case, none of the elements is true and therefore the function returns a false value.

Why would you use `any()`? Consider the case where you are implementing a web system that requires a user to accept all of the privacy statements. This is a big deal now in the European market, and thus is a big deal for programmers. If you collect all of the user responses to your privacy statements into a list, you can then simply call `any()` to very that they were all accepted.

The `all()` function is the converse of `any()`. It returns true only if all of the items in a list are true. Here's an example:

```
>>> x=[1,2,3,0]
>>> print(all(x))
False
>>> x=[1,2,3,4]
>>> print(all(x))
True
```

Note that a non-zero value is considered `True` in Python, and thus any list that contains all non-zero values is true for the `all()` function. What happens if you have a negative value? Let's try that out, since experimentation is the core of the Pythonista.

```
>>> x=[-1,1,2,-3]
>>> print(all(x))
True
```

Not surprisingly, negative numbers are considered truthy. You will hear the term *truthy* thrown about a lot in Python. Things aren't absolutely true or false, they are mostly true (that is, non-zero) or mostly false (that is, zero or `None` or `False`). Rather than worry about exactly what the programmer uses, Python groups together things that make sense.

The next function to consider is the `enumerate()` function. The enumerate function is one of those handy things that most programming languages require you to handle yourself. For example, let's imagine that you are working in Java, and have an array of elements. You want to print out the elements in a table, with an index

value showing the number of the element and then the value. You'd probably do something like this:

```
for(int i=1; i<11; i++)
{ System.out.println("Element: " + I + " equals " + myarray[I]); }
```

This certainly works, but it is not only error prone, but hard to read. Enter Python and the `enumerate()` function:

```
>>> for idx, value in enumerate(my_array):
...    print(idx, value)
...
0 1
1 2
2 3
3 4
4 5
5 6
```

As you can see, the `enumerate()` function returns an *enumerator*, hence its name, which contains two values. The first is the index of the position you are at in the list or other collection. The second is the value at that position in the collection. Needless to say, you can use an enumerate call on any Python collection:

```
d = {
    "Element1" : 1.0,
    "Element2" : 2.0,
    "Element3" : 3.0
    }

0 Element1
1 Element2
2 Element3
for idx, value in enumerate(d):
    print(idx, value)
```

Of course, dictionaries have both keys and values. How then to print them all out in a simple, Pythonistic way?

```
d = {
    "Element1" : 1.0,
```

```
    "Element2" : 2.0,
    "Element3" : 3.0
    }

for idx, (key, value) in enumerate(d.items()):
     print("Entry {0} has a key of {1} and a value of {2}".format(idx,
key, value))
```

The output from this little program snippet is:

```
Entry 0 has a key of Element1 and a value of 1.0
Entry 1 has a key of Element2 and a value of 2.0
Entry 2 has a key of Element3 and a value of 3.0
```

Which is exactly what we were looking for!

Filtering is something that is almost a standard when working with data sets these days. From doing high-performance filtering of Big Data to simply filtering down lists of items that the user can select from based on a given input, the ability to filter is important. Python, of course, considers filtering to be important enough to make as part of the basic language, rather than bolting it on. You'll see that this is a common theme for Python.

To use filter, we need a function that determines whether or not the data passed to it is valid for the result set. In our example, we'll look at a simple function, since we haven't really discussed functions yet, that checks if a given value is odd:

```
def odd(x):
   return not (x % 2)== 0
```

We can test this function in the immediate window:

```
print(odd(3))
print(odd(4))
True
False
```

As you can see, the function tells us whether or not a given number is odd or not. Now, let's take a list of numbers:

```
x=[1,2,3,4,5,6,7,8,9,10,11]
result = filter(odd, x)
for r in result:
 print(r)
```

```
1
3
5
7
9
11
```

As you can see, the filter function works properly. We'll expand on this a little later, as we learn about lambda functions and generators, and be able to write the last line as a single line of Python.

When we think of maps, we usually think of GPS coordinates, of paper displays of geographic areas, and the like. However, map has another meaning, which is to take a given set of data and project it (*or map it*) onto a given set of information. In Python, we use the `map()` function to apply a specific operation to a collection of data. For example, if you wanted to create a list of the squares of an input list, you might write something like this:

```
input_list = [1,2,3,4,5]
output_list = []
for i in input_list:
    output_list.append(i*i)
print(output_list)
```

As you might expect, the output from this is:

```
[1, 4, 9, 16, 25]
```

In Python, however, we can do this much more easily, using the map() function:

```
def square(x):
    return x*x
output_list = map(square, input_list)
for o in output_list:
    print(o)
```

This is a simplistic example, one can imagine much more complicated ones in which the `input_list` is transformed through multiple processes.

The next stop on the Python express is the `range()` function. This is actually one of the most used, and most versatile functions in the Python arsenal. The `range()` function generates a list of items, based on inputs. If you are a C++ or Java programmer, you probably know the basis of the range function. It is the same thing you are used to

specifying in your `for` loops. For example, in C++, if you want to write a loop that goes from one to ten, you'd write:

```
for(int I=0; I<10; ++I)
```

This loop goes from the starting value, `0`, up to but not including the ending value, `10`. So, you would see the numbers `0` to `9` used within the loop. The equivalent Python `range` loop looks like this:

```
for I in range(0,10):
```

Again, this loop goes forward, starting at zero, and ending at nine (the stop value minus one). In C++, you can also use that final argument in the `for` loop to go backwards, as a simple example:

```
for (int I=10; I>0; --I)
```

This loop starts at the value of ten, and goes backwards until it reaches `1`. The > sign indicates that it should go only to the value before the stop value of zero. So, you would see the numbers `10`, `9`, `8` and so forth, until you reached on. The equivalent Python `range` loop looks similar:

```
for I in range(10,1,-1):
```

The primary difference between range and the Java or C++ for loop is that the range function can be used outside of a loop. The `range()` function produces an iterator. If you ever used Python 2, you may have written something like this:

```
list_of_integers_from_1_to_10 = range(1,11)
```

This will not work in Python 3, it is one of the big changes from the 2.x range of the language. Instead, if you write:

```
list_of_integers = range(1,11)
```

You are actually creating an iterator, a sequence operator, not a collection. So, while you could do things like range(0,10)[5] in Python 2.x, this will not work in Python 3. The developers of Python rarely take away something without giving you something in return, so we can write things like:

```
my_list = list(range(0,10))
```

This produces a list of values from zero to nine, which you can slice, dice, and iterate over to your heart's content.

Likewise, you can apply all of the things we've learned so far to your `range` object:

```
my_list = list(range(0,5))

print(my_list[2])

for s in map(square, my_list):

    print(s)
```

This code produces the following output:

```
2 --> This s the index my_list[2]
0 --> These are the squares of the values in the list
1
4
9
16
```

Clearly, the `range` operator is very powerful and versatile. It is one of those things you will find yourself using again and again. It isn't required that you use constant values in it either, you can do things like this:

```
s = "This is a test"
for i in range(0, len(s), 2):
    print(s[i])
T
i


s
a
t
S
```

A relative of range, with respect to utility and versatility, is the slicing operator in Python. You may not recognize the slicing operator, since it isn't used as a verb like `range` or `any`. In the C++ or Java worlds, the slicing operator is a close relative of the index operator, which looks like `[]`. You can use the slice operator in a variety of ways, and most Python developers use it in more than one in a given application. So, let's take a look at this versatile functionality, in each of its fashions.

First of all, the slice operator is used to retrieve a specific index of an interable. For strings, for example, we can look at the 3rd character in a string directly, using slicing:

```
s = "This is a test"
print(s[2])
```

As always, any iterable in Python, like Java or C++ or C#, is zero-based. Thus, the third character is character number three, whereas the first character is number zero. Printing out the third character is using the string as an indexable collection, in this case of characters, and retrieving the third one. As with all languages, trying to access a character outside of the range of the string:

```
print(s[len(s)])
```

```
IndexError: string index out of range
```

This error occurs because the `len()` function returns the total number of characters in a string, and the indexing is zero-based. So, for the string above, we are trying to print out one character more than the number we are allowed, and so we get an error.

Likewise, you would assume that doing the following would also generate an error:

```
s = "This is a test"
print(s[-1])
```

Running this in interactive mode gives you the following output:

```
T
```

Wait, you say, how can that be? Minus one is clearly outside the range of zero to the number of characters in the string minus one, right? Well, kind of. In Python, the indexing operator can be used directly in two ways. Positive values count from the left side of the string or array. That is, `array[1]` is the second value, and `array[4]` is the fifth. However, if you user a negative index, you can retrieve values from the right side of the array. For a string, that means the end of the string moving backwards. In fact, we can print out a string using the same methodology we've used before, and then print it out backwards simply by changing the sign of the index:

```
a = "test"
for i in range(0, len(a)):
   print(a[i])
print("Backwards:")
for i in range(0, len(a)):
   print(a[-(i+1)])
```

```
The output of this snippet looks like this:
t
e
s
t
```

Backwards:

```
t
s
e
T
```

As you can see, we are printing out the string using the exact same methodology, but moving from the back forward. The only reason we need to do the (`I+1`) part is because a negative zero is still zero.

The slicing operator isn't restricted to a single index, however, it can have more than one. Unlike functions, however, the indices are separated by the colon (:) rather than the comma:

```
a = "test"

print(a[1:3])
```

The slice operator used this way returns a *slice* of the iterable beginning with the first index and terminating with the second index. So, in the example above, we would expect it to print out the characters from the second to the fourth. In fact, if you run the snippet, you see the following output:

```
es
```

This, of course, is the middle of the string we defined. Given what you've seen so far, it shouldn't surprise you at all to know that you can also use negative indices in the exact same fashion:

```
print(a[-3:-1])

Es
```

An important point about these indices. You can omit one or more of the indices by simply leaving it out in the code. If you do this, the outermost limit of that index will be used. For example, if we write:

```
a = "test"

print(a[:3])
```

Python will interpret this as if we had written [`0 : 3`] since the omitted index is on the left hand side. That side would naturally begin with `0`. Similarly, if we were to write:

```
a = "test"

print(a[2:])
```

In this case, Python assumes that the code really means: [`2:len(a)-1`] since the right-most index will always be the length of the iterable. Running this code, not surprisingly, gives us an output of:

```
st
```

Which is exactly what we would expect, again. With the exception of negative numbers, this really isn't much different than any other language, aside from the fact that you can't do it this directly. In Java, or C++, at least for strings, there is some variant of the substring operator that will return you a string made up of characters in a given range. Unless you roll your own, though, it won't work for negative, or missing, values.

Python allows for one more entry in the slicing operator, also separated by a colon. This is the *step count* you wish to use for getting from the start to the end. It is a smidge confusing, because it doesn't do exactly what you might think if you were using Java. For example, consider these statements:

```
a = "test"

print(a[0:4:2])
```

The output from this is:

```
ts
```

Not surprising, we told Python to output things starting from the left, ending on the right, and incrementing by two.

But what about this?

```
a = "test"

print(a[4:0:-2])
```

This one prints out:

```
te
```

Surprised? You shouldn't be, it starts out at the left-hand end, goes backward two characters at a time, and ends when it goes past the right-hand end.

But this one is likely a surprise:

```
a = "test"

print(a[::-1])
```

This snippet prints out the following:

```
tset
```

This is the string backwards. In fact, this is a very common use of the slicing operator, to reverse a string, or other iterable in a string line. It might not be obvious with non-strings, but take a look what happens to a normal list:

```
x=[1,2,3,4,5]

print(x[::-1])

[5, 4, 3, 2, 1]
```

This is the exact same thing. Strings are printed as runs of characters, while lists are printed as individual elements, otherwise you are looking at the same process.

Go back through and work through all of the slicing operator examples, because they are going to be a common usage pattern in your coding of Python.

Sorted

If filtering is the number two most common thing in the software development world, then sorting is certainly number one. We often need to present data in sorted order for the user, whether by name or date or some other bit of information. Naturally, Python was built with this need in mind. There are several ways to sort items in Python, let's look at the most common one, the sorted function.

Here's an example of sorting a simple list of words:

```
x = ["This", "is", "a", "test", "of", "beta", "gamma", "zeta", "alpha 2"]

xs = sorted(x)

print(xs)
```

```
['This', 'a', 'alpha 2', 'beta', 'gamma', 'is', 'of', 'test', 'zeta']
```

The result of calling the `sorted` function on a collection is a new collection, in sorted order. When sorting strings, as you can see above, the sorting is case sensitive, thus the capital letters come first, and the lower case letters second. Within a single letter, of course, each subsequent letter in the string is examined and compared to produce the sort order, as you can see with the `a` and `alpha` entries.

You might wonder, can I simply sort a string? The answer, of course, is yes, since a string is just a collection of characters. The result may or may not be useful, depending on what you are trying to do:

```
x = "This is a test of beta gamma zeta alpha2"

xs = sorted(x)

print(xs)
```

```
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '2', 'T', 'a', 'a', 'a', 'a',
'a', 'a', 'a', 'b', 'e', 'e', 'e', 'f', 'g', 'h', 'h', 'i', 'i', 'l', 'm',
'm', 'o', 'p', 's', 's', 's', 't', 't', 't', 't', 'z']
```

As you can see, the result of sorting a string is again a list, this time of characters. This makes sense, since each individual element in the collection is sorted, and in this case, the individual elements are characters.

What if you wanted to sort a string by words? We've already looked at all of the pieces you need to do this. To begin with, we take the sentence string and split it into *words*:

```
xw = x.split()

print(xw)
```

```
['This', 'is', 'a', 'test', 'of', 'beta', 'gamma', 'zeta', 'alpha2']
```

Next, we sort the resulting list of words:

```
xws = sorted(xw)
print(xws)
['This', 'a', 'alpha2', 'beta', 'gamma', 'is', 'of', 'test', 'zeta']
```

As you can see, the result is the same as the original list that we sorted above. Sorted, of course, accepts any sort of iterable, so we can sort dictionaries too:

```
d = {
    1: "zeta",
    2: "xylem",
    3: "wisconsin",
    4: "violent"
}
ds = sorted(d)
print(ds)
[1, 2, 3, 4]
```

Of course, the result of sorting a dictionary is a sorted list of keys, so this is one way that you could make a dictionary into an ordered collection, rather than an unordered one. Sorted also works for tuples, because it does not modify the original collection. You may remember that tuples are immutable, they cannot be modified in order or contents. The `sorted` function returns a copy of the tuple:

```
t = (1,5,2,4,3)
ts = sorted(t)
print(ts)
print(t)
[1, 2, 3, 4, 5]
(1, 5, 2, 4, 3)
```

Note that the second tuple shown is the original, and it is not modified in the least. As a note, there is a `sort` method that exists only for lists, and sorts elements in place. When you have a list, and you don't mind modifying its order, you can call sort instead:

```
t = [5,3,1,2,4]
print(t)
t.sort()
```

```
print(t)
```

```
[5, 3, 1, 2, 4]
```

```
[1, 2, 3, 4, 5]
```

Naturally, this won't work for tuples or other immutables, since you can't change their values or orders.

You probably noticed in our string sorting example that the capital letter beginning a word sorted earlier than the lower case letters. This is because capital A comes before lower-case `a` in the ASCII encoding scheme. What if you wanted to sort things based on lower case letters only? There are two alternatives here. If you are sorting the string by itself, you can just call the lower function on it:

```
x = "This is a test of beta gamma zeta alpha2"
```

```
xs = sorted(x.lower())
```

```
print(xs)
```

```
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '2', 'a', 'a', 'a', 'a', 'a',
'a', 'a', 'b', 'e', 'e', 'e', 'f', 'g', 'h', 'h', 'i', 'i', 'l', 'm', 'm',
'o', 'p', 's', 's', 's', 't', 't', 't', 't', 't', 'z']
```

The designers of Python, however, recognized that sometime you would need to be able to sort on things that they had not anticipated. For this reason, the `sorted()` function permits you to specify a third argument, called **key**, which is a function that returns the key for a given element to sort. This function can be one you write, or it can be a built in function. For example, in our lower case key above, we could write:

```
x = "This is a test of beta gamma zeta alpha2"
```

```
xs = sorted(xw, key=str.lower)
```

And it will work exactly the same way. It works for all collections, so you could split the string into a list of words and apply the same logic:

```
x = "This is a test of beta gamma zeta alpha2"
```

```
xw = x.split()
```

```
xs = sorted(xw, key=str.lower)
```

```
print(xs)
```

```
['a', 'alpha2', 'beta', 'gamma', 'is', 'of', 'test', 'This', 'zeta']
```

Going back to our dictionary sorting example, you can see that the `sorted()` method only appears to sort on the keys to the dictionary. Likewise, for a list of structured items, such as tuples, you can only sort on the first element:

```
marks = [
```

```
    ('Fred', 'Science', 90),
```

```
    ('George', 'Math', 95),
```

```
    ('Albert', 'English', 90)
]
print(sorted(marks))
[('Albert', 'English', 90), ('Fred', 'Science', 90), ('George', 'Math', 95)]
```

Does it surprise you that you can sort a list of tuples? You can't sort an individual tuple, because it is immutable, but the list containing them is not. In any case, the `sorted()` function automatically sorts on the first element of the tuple, in this case the name. What if we wanted to sort on the class for the student, which is the second element of the tuple? The answer lies in the key argument to the `sorted` function, which can return a piece of its caller:

```
def get_class(t):
   return t[1]
print(sorted(marks, key=get_class))
```

In this case, we've created a function that just returns the second element of the tuple. When we talk about lambda functions, which are almost exactly the same as they are in Java or C#, you'll see that we can substitute one of those for an in-place extraction.

For beginners in Python, this is often more than enough to get going with. However, the professional is already thinking ahead, wondering about how to accomplish other tasks. In a school assignment, you'd be asked to sort a list of grades. But in the real world, the program manager is harassing you to sort the list by subject, and within the subject, by grade, so that he can see things the way a teacher would want to. Surprise! Python is way ahead of you. To get there, though, we have to introduce a slightly new concept, imports. We'll talk about these considerably more further in the book, so for now, just accept that an import in Python is very much akin to the include statement in C++ or the self-same import statement in Java. Here's how you would do it. First, the code, since it is easier to discuss code with a programmer than to lecture to them. To begin with, we are going to increase the size of our list of grades, so you can see the actual impact of the code:

```
marks = [
    ('Fred', 'Science', 90),
    ('George', 'Math', 95),
    ('Albert', 'English', 90),
    ('Zelda', 'English', 96),
    ('Alvin', 'Math', 92)
```

Now, we'll duplicate the `sorted()` function call above, but use a slightly different method for extracting the element we want to sort by:

```
from operator import itemgetter, attrgetter, methodcaller

from operator import itemgetter, attrgetter, methodcaller

print(sorted(marks, key=itemgetter(1)))

[('Albert', 'English', 90), ('Zelda', 'English', 96), ('George', 'Math',
95), ('Alvin', 'Math', 92), ('Fred', 'Science', 90)]
```

As you can see, the output still sorts by the subject (the second element in the tuple), but leaves all of the other pieces in the order in which they were defined. Now, let's add a little complexity to the problem, sorting by the grade within the subject. As we can see from the data, both the `Math` and `English` subjects have multiple entries, and at least one of them is not already sorted.

Here's the modification to our code to sort by a second data element:

```
from operator import itemgetter, attrgetter, methodcaller

print(sorted(marks, key=itemgetter(1, 2)))
```

That's the only change needed, adding a second parameter to the `itemgetter` function call! Looking at the output:

```
[('Albert', 'English', 90), ('Zelda', 'English', 96), ('Alvin', 'Math',
92), ('George', 'Math', 95), ('Fred', 'Science', 90)]
```

You can see that the Math scores have been updated to be in the requested sort order.

Finally, we would be remiss if we didn't point out that there is another optional argument to the `sorted()` function called *reverse*. As you might guess, reverse is used to indicate that you want to sort the elements in reverse order, in other words in descending order. The default, of course, is ascending.

```
a_sorted_list = [1,2,3,4,5,6,7,8,9,10]

print(sorted(a_sorted_list, reverse=True))

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

The reverse and key argument can be used together, but you cannot specify a different sort order (ascending or descending) for each key you select. If you want to do something like that, you'll have to write your own sort routine.

The zip function

If you have worked in the software world for any length of time, as most of you have, you have encountered the zip program or one of its combinations. The zip program compresses files down into an archive which can then be more easily copied or transmitted. You might think, therefore, that the `zip()` function in Python might be related to compression and archiving. You would be wrong, they have absolutely nothing in common.

The `zip()` function name in Python comes from a shortening of *zipper*. The `zip` function takes a set of iterables, and weaves them together to form a list of tuples that contain the elements of each. In its absolutely simplest form, the `zip()` function accepts no arguments and returns an empty list:

```
print(zip())
```

```
[]
```

In the next most complicated case is applying zip to a list of elements:

```
string_list = ["Alpha", "Beta", "Gamma"]
```

```
print(zip(string_list))
```

```
[('Alpha',), ('Beta',), ('Gamma',)]
```

As you can see, this case produces a list of tuples, with a single element each. You can guess, therefore, that if we add a second list to the equation:

```
number_list = [1,2,3]
```

```
print(zip(string_list, number_list))
```

```
[('Alpha', 1), ('Beta', 2), ('Gamma', 3)]
```

Hopefully, you can see the pattern at this point. Each element is *zipped* into a tuple with the other list elements in the same position. Do you wonder what happens if you have a list that is shorter than the rest? You might think that you get a tuple with less elements in the output list, but that isn't the case. The `zip()` function uses the shortest list given as the length of the output list:

```
misc_list = [2.5, 5.0]
```

```
print(zip(string_list, number_list, misc_list))
```

```
[('Alpha', 1, 2.5), ('Beta', 2, 5.0)]
```

This can be a useful way to build 'sets' of data to work with, but it is limited to the smallest possible list that covers all of the inputs.

Booleans and truthiness

We have briefly touched on Boolean and the idea of true and false in Python earlier, but it is worth exploring the concept just a bit. Most languages have the concept of true or false. In C++ and Java, there are pre-defined constants for true and false that you use with the Boolean type. Python also has the notion of `True` and `False` (note the capitalization, Python does not support `true` and `false`), but applies the idea a little more broadly.

In Python, there are three basic truth types. For integer values, the value of zero is false. All other values, whether positive or negative, are true. With this said, however, that doesn't mean that the value equates to `True`. For example, if we write this:

```
for i in range(-1, 2):
    print(i, i == True)
```

The output might surprise you a little:

```
(-1, False)
(0, False)
(1, True)
```

However, if we do something that expects a true or false result:

```
for i in range(-1, 2):
    if i:
   print("True!")
    else:
   print("FALSE")
True!
FALSE
True!
```

As you can see, any non-zero number is *truish*, or *truthy*. It does not mean, however, that a number is equal to `True`, which is where the notion of truthiness comes from. Here is a simple chart for each type of data in Python that shows you the values of `False` for that data type.

Type	False Value	True Values
int	0	Anything but 0
sequence	Empty sequence	Any sequence with values
None	Always False	Never false

A sequence, of course, is a list, dictionary, tuple, or string. That means that the empty string is a false value:

```
s1 = ""
s2 = " "
if s1:
   print("S1 = TRUE")
else:
   print("S1 = FALSE")
if s2:
```

```
    print("S2 = TRUE")
else:
    print("S2 = FALSE")
```

As you can see, an empty string, that is one with zero characters in it, is `False`, whereas a string with any characters in it, even blanks, is `True`. Similarly a list with zero elements is `False`, and all collections with at least one element is `False`.

It is very important to remember that comparing *truthiness*, that is if `<value>` is not the same as comparing to `True`, if `<value> == True`. Those starting out with the language from other languages often want to be as concise as possible, and make the mistake of comparing to the `True` value. Don't make this mistake.

Comments

For beginning programmers, this is enough to get them started writing code. You've been doing this for a while, though, and know that sooner or later you will have to explain to others exactly what your code is doing. Explaining the behavior of code is done through either documentation, electronic or paper, and in code comments. Hopefully, we all fully comment our code!

Python supports two methods of commenting code. The first is to use the pound (`#`) symbol for a single line comment. Anything that follows the pound symbol is considered to be comment, it will not be read or executed by the interpreter. Oh, for those of you younger than the concept of pound signs and phone keyboards, this is also called the hash sign, as in hashtag `#Python!`

You might want to comment a block of code that you are writing to indicate where the values come from, for example:

```
# This is Planck's Constant, which relates the energy of a photon to its
frequency

planks_constant = 6.62607015*(10**-34)
```

This allows people to understand what your line of code is meant to do. You can also use the single line comment to remove a piece of code from executing at the moment. For example, we might use it to comment out statements that we use for debugging purposes, but don't want in the production level code:

```
x= y*z/j+l

#print(x)
```

In this case, we are removing the `print` statement from executing during normal production. This is a common way of leaving in things that are useful for the developer during the development and maintenance phases of a project.

Python also supports multi-line comments, which are usually used to indicate the purpose of a block of code, or to embed things like copyright statements in your code for corporate purposes. The multi-line comment in Python is done using the triple quote format, which you may remember is also the way in which to create a string that is longer than a single line. By now assigning the block to a variable, Python will not allocate space for the string, so it becomes a simple human-readable comment:

```
"""
This is a multi-line comment.
It doesn't do anything.
"""
```

Conclusion

Programming is both difficult and easy. It is difficult because, unlike math and the hard sciences, there is no single *right answer* to how to program something. At the same time, it is easy because the number of things you really need to know is limited. In this chapter, we have reviewed most of the pieces that you will need to write Python programs. From this point on, we are going to focus on the nuts and bolts of the language, how to take the little pieces we've learned here and turn them into full-fledged Python applications.

In our next chapter, we will explore some of the nuts and bolts of the Python programming language, tackling the basic programming building blocks like conditionals and loops.

Questions

1. What is the difference between an integer and a float?
2. How do you iterate over a collection?
3. What is the difference between a mutable and immutable collection?
4. How do you create multi-line comments in Python

Chapter 3
The Nuts and Bolts

Introduction

To this point, we've looked at the building blocks of the Python language, getting it running, the data types available, and some of the basic functionality built into the language. As a professional, you know that the building blocks are just the groceries in a meal, the meat and potatoes that have to be assembled into something delicious and wonderful. In this chapter, we'll start looking at those recipe components, the ways in which the language works. We'll look at the various kinds of conditional statements, as well as the shorthands that those statements allow. We'll look at looping constructs, so that you can repeat yourself in code ad infinitum. Then we'll start looking at extending the language, creating your own groupings of code into functions, building your own types using classes, and how Python works with those classes to create objects that can themselves be extended and modified in your code.

Structure

- Conditionals
- Indentation
- Functions
- Classes
- Objects

Objectives

By the end of this chapter, you should be able to put together a complete application in Python. It might not be efficient or ideal, but it will work and do a bit more than the classical 'Hello world' program that most beginning books start with.

Conditionals

Most modern programming languages contain one or more conditional statements. The main conditional is the `if` statement, which executes code only if a statement evaluates to a true value. Some languages support a ternary operator, which is simply a short-hand `if` statement. Yet other languages contain a `switch` statement, which allows you to evaluate a given value and execute a block of code based on its value in a list of possible values.

Python has the classic `if` statement and a few variants of this simply statement. The basic language has no `switch` statement, although we'll look at ways to do this later on in the book in the *Chapter 11:Tips and Tricks* section. Python does not have a case statement because the `if` statement makes it unnecessary. Let's start out with the simplest form of the `if` statement, the simple if. An if statement is made up of a condition to evaluate and one or more lines of code that should be executed if the condition evaluates to any truthy expression. For example:

```
x = 100
if x == 100:
   print("X = 100")
```

As you can see, we have an `if` statement, followed by a condition to evaluate (x equal to `100`). If `x` is less than `100` or greater than `100`, the code block within the `if` statement will not be evaluated. If the value of `x` is exactly 100, then the program will print out the string `X = 100`. There are a number of things worth mentioning here.

First, notice that Python uses the C# style == operator to make an exact comparison to a value. For numeric values, simple comparison is done. For other values, a comparison is done using the rules for that data type. Strings are compared for both case and content, so `HELLO WORLD` is not equal to `hello world`, nor is `This Is A Test` equal to `Test is a This`.

The comparison operators are not surprising, but we'll list them out for completeness anyway:

Operator	Meaning
==	Two values are equal
>	Left hand value is greater than the right hand value

>=	Left hand value is greater than or equal to the right hand value
<	Left hand value is less than the right hand value
<=	Left hand value is less than or equal to the right hand value
<>	The left hand value is not equal to the right hand value
!=	The left hand value is not equal to the right hand value

A few things worth noting. Python, like C++, C# and Java, differentiates between the assignment operator (=) and the comparison operator (==). In C++, for example, writing something like if x = 99 is permitted in some versions of the language, and does what you said to do but probably not what you meant to do. There are legitimate reasons to do things like this, but the developers of Python decided that it would be in the best interests of programmers everywhere not to permit it. As a result;

```
if x = 100:
   print("X is 100")
    if x = 100:
         ^
SyntaxError: invalid syntax
```

You will also notice that Python accepts both the <> and != versions of not equal to. Why? There's really no reason given for it, except that Python attempts to appeal to as wide a base as possible, and selecting a single method for this, in an environment in which about half the languages accept each version, seems like a problem in the making. Either way, you can do either of these:

```
x = 100
if x <> 99:
   print("Not 99")
if x != 99:
   print("Not 99")
Not 99
Not 99
```

The "walrus" operator

Although this piece of functionality does not exist in Python 3.7, it will in Python 3.8. The so-called **walrus operator** which is the := operator, does an assignment and a test at the same time. So, instead of writing:

```
x = dictionary_1['something']
If x == nil:
```

```
    Do_something()
```

We can now write:

```
If x := dictionary['something']:
Do_something()
```

It isn't a major change, for all of the vociferous comments about it in the Python community, but it is something that will likely be there as you program in Python in the real world, so it is worth knowing about.

ANDs, ORs, NOTs, and logicals

Languages like Java, C++ and C# use a confusing array of logical operators. The `&&` symbol means *and* from a logical viewpoint, as in this `&&` that means this and that where both must be true for the expression to evaluate to true.

Python doesn't try to trick you into remembering whether there is one or two ampersands in the *and* operator. In most other languages, the single ampers and is a bitwise operator. For example, you might want to know if the low bit of a number is set for some reason. In Python, and most languages, we can do this:

```
for x in range(0,3):
print("Value: {0}".format(x))
  if x & 1:
 print("Low bit set!")
```

```
Value: 0
Value: 1
Low bit set!
Value: 2
```

The equivalent bitwise operator for or is the | (pipe sign). Neither of these operators can be used in an *if* statement to test for a combination of factors. Instead, Python has the AND and NOT operators:

```
x = 10
if x < 20 and x > 5:
print("In range")
```

```
In range
```

Python does have an operator that you won't find in most other languages, the `is` operator. You can use the `is` operator in a variety of ways. You can use it as the equivalent of an equal statement:

```
if x is 10:
   print("X is 10")
```

You can use the `is` statement to test for the type of a variable:

```
if type(x) is int:
   print("X is an integer")
```

Finally, you can test to see if two variables point to the same location or value in memory. Consider the following snippets:

```
x = "True"
y = "True"
z = "False"
if x is y:
 print("X = Y")
if x is z:
print("X = Z")

x = 1
y = 1
z = 2
if x is y:
   print("Yes")
if z is x:
   print("No")
```

These lines print out the following:

```
X = Y
Yes
```

This is because `x` and `y` point at the same string `True` in the first example, while `z` does not. In the second block, `x` and `y` both point at the integer value `1`, so they are the same. The value of `z` points at `2`, however, so it isn't the same.

Finally, you can combine the is with the NOT operator to see if something isn't the same as something else:

```
if z is not x:
print"It is not!")
It is not!
```

One more thing, you might notice that comparing a single element to a list or other collection doesn't work (aside from comparing two strings for content equality). In most languages, to check to see if a given value is in a collection, you'd have to write some sort of looping construct to go through the various elements of the collection and compare them to the value you are trying to find. Python makes this considerably easier with the in and not in operators:

```
list_of_commands = ['help', 'quit', 'file']

command = 'help'

if command in list_of_commands:

   print("Valid command")

bad_command = "fred"

if bad_command not in list_of_commands:

   print("The bad command was not found!")

Valid command

The bad command was not found!
```

For lists and sets, the value is checked against each list element. For dictionaries, the value is checked against the keys of the dictionary. For strings, the operator checks to see if the given sub-string is found anywhere within the given string.

Returning to the `if` statement, you are probably accustomed to the various forms of it in other languages. You can test for a condition and do something if it is true. If it is not true, there is usually an else clause that you can use to do something else. Python has the same structure, but adds something of a twist. Remember that Python doesn't have a switch (or case) statement? There's a reason for this, Python has the `elif` and else statements instead. Here's what it looks like in code:

```
cmd = "quit"

if cmd == "help":

   print("I'd be happy to help you!")

elif cmd == "stop":

   print("We can stop whenever you like!")

elif cmd == "drop":

   print("Consider the matter dropped")

else:

   print"I have no idea what you are asking")
```

The block is evaluated in the following fashion. If the first condition (`cmd == "Help"`) is not met, the Python interpreter moves to the next statement, which is an "`elif`".

Elif is short for `else...if` and does another comparison. At any point in the stack of `if's` and `elif's`, any condition that matches terminates the check, and control drops to the next statement below the block. If none of the `elif` statements match, control passes to the `else` statement. In the example above, since the command string given doesn't match, we see the `else` statement printed out.

You've probably noticed that the if and `elif` and else statements terminate with a colon. This indicates to the interpreter that what follows this is a command block to be executed for the if. If you leave off the colon, the interpreter will generate an annoying error:

```
cmd = "quit"
if cmd == "help"
    if cmd == "help"
                   ^
SyntaxError: invalid syntax
```

Obviously, there's nothing wrong with your *syntax*, it is telling you that it expected to find a terminating colon. Watch out for this, since the colon is easy to forget, and most IDE systems don't display a clear enough signal when the condition is detected.

Indentation

Indentation is the amount of white space that a coding line begins with. Up to this point, we've spent a lot of time looking at how the code looks without really talking about some of the details of the syntax. The most important detail is indentation, so let's take a bit of a look at that now. For many programming languages defined after COBOL and FORTRAN, indentation is unimportant. C++ does not care which column you begin coding in, Java doesn't care whether or not a line within a block is indented a certain amount. C# has no issues with deciding what block a given line of code belongs to, it is defined by the curly brace characters that define blocks.

Python is an older language, and one designed around simplicity. For this reason, the indentation level is critical to the interpreter deciding whether a given piece of code belongs to a block or not. For example, these two snippets of code have very different outputs:

```
# Block 1
x = True
if not x:
   print("The value is true")
   print("So we will do what we need to do")
```

```
# Block 2:
if not x:
   print("The value is true")
print("So we will do what we need to do")
```

In the case of the first block, where both of the `print` statements are indented, the interpreter will recognize them as a single block and will not output either statement, since x is not `True`. In the case of the second block, where the first line is indented, and the second is not, the second line is not considered part of the `if` block, and will be output regardless of the value of `x`.

Indentation is most important when you have nested blocks of code. As an example, we might have something like this:

```
for i in range(0,10):
if i % 2 != 0:
  print("{0} is odd".format(i))
```

Without the indentation, the Python interpreter has no way of knowing that the `print` statement is contained within the `if` statement.

The pass statement

Back in the early days of programming, when assembly language was the only option for writing code on a computer, there was a statement called `noop`. This statement, pronounced *no op*, was short for no operation. In short, it did nothing. Why would you want a statement that did nothing? It turns out, doing nothing is often quite valuable. For one thing, you could use it in place of a wait statement to kill time. For another, you could put it in branching statements. Many assembly languages had a the equivalent of the if statement, but required an else. What if you didn't want to do anything in the `else` case? You put in the `noop` command.

Python has the equivalent of the `noop` command, it is called `pass`. There are a few cases when you want to use it, and most of those will be covered when we discuss classes and functions. For now, however, if you see something like this:

```
if somecondition:
    Do_this()
else:
    pass
```

Realize that this statement is likely there temporarily, to address the fact that the programmer knows that something needs to be done if the condition is not true, but isn't entirely sure what. It is kind of like a comment, but in code.

Now that we have all of the pieces in place and understood, it's time to look at the important things in any language, the flow constructs.

Loops

If you happen to be old enough to remember the original BASIC programming language, you will recall that it had no looping structure. A loop was done by the GOTO statement:

```
10 DO_SOMETHING
      .. doing something
      IF NOT DONE GOTO 10
```

Seems awfully primitive and error prone, doesn't it? If the value of `DONE`, whatever that is, never got set properly it would keep going back and forth across the statements forever. Fortunately, we all know that we don't do things like that anyway. For example, nobody would ever do something like anymore in modern languages like Java or C++.

In Java, for example, you couldn't possibly do something like this:

```
class MainClass
{
  public static void main(String[] args)
  {
    int count = 1;
    while (count < 10)
    {
      System.out.println("Count is: " + count);
    }
  }
}
```

Yes, this loop will continue forever. Yes, it is exactly the same as BASIC. It is called an infinite loop, and there are few languages in the world that you cannot define such a construct in. Python is not one of those languages, you can create an infinite loop in Python just as easily as in any other language.

In spite of this little affront to programmers everywhere, Python does support loops. In fact, it supports two different sorts of loops, the `for` and `while` loops. Would it surprise you to know that there is little difference between them? This is true of most languages. Python has some things that are a little easier to do in one form

or another, but the truth is, you can replace a while loop with a for loop anywhere, just as you can replace a `for loop` with a `while` anywhere. Keep that in mind when designing your applications.

The loop types are:

- `for`
- `while`

In general, the form of the for loop is for `<variable> in <iterable>`. There are quite a number of ways to create an interable, we'll look at two specific ones here. First, there is the `range()` function that we've looked at before. This makes the `for` loop most like the Java or C++ `for loop`. These languages have for loops of the form `for <variable = initial state; <variable> != terminal state; <variable>modification function`. Let's say that you have a loop to count the odd numbers from zero to ten. We could write this in C++ as:

```
for ( int I=0; I<10; ++I ) {
if ( I % 2 != 0 )
    cout << I << endl;
}
```

The equivalent Python for loop to accomplish the same thing looks like this:

```
for i in range(0, 10):
if i % 2 != 0:
  print(i)
```

As you can see, while there is a minor difference in syntax, the basic loop is the same. C++ and Java are a bit more direct about how the loop variable is modified, whereas Python hides it within the `range()` function, but this takes almost no time to get used to. Likewise, if we have a list (or array in C++) and want to iterate over it, the syntax is similar but different:

```
for ( int I=0; I<array.length(); ++I ) {
  cout << array[I] << endl;
}
```

In Python, we'd write:

```
for a in array:
    print(a)
```

Once again, ignoring the slight syntactical differences, these are quite similar.

The second sort of loop is also found in Java, C++ and so forth, and is called the `while` loop. As its name implies, the loop executes while a given condition is `true`. The general form of the `while` loop is:

```
while condition:
    do_some_stuff
```

Obviously, the most dangerous thing about the `while` loop is that it requires you to think about the condition that terminates it, and make sure that the condition is set at some point. Of course, there are times when you really do want an infinite loop. For example, suppose that you are writing an application that is meant to monitor the state of your system. You might want it to never end, unless it was physically killed. In this case, you could create an infinite loop by writing:

```
while (True):
```

The while loop is important because it evaluates its condition at the top of the loop, meaning that your code may not execute at all. For example, if we write a loop like this:

```
done = True
while not done:
print("Not done!")
```

You will discover that it doesn't print anything at all. If you are accustomed to the `do…while` loop in other languages, which always executes at least once, this can be a little disconcerting. Whatever you choose to do, it is important to understand that the while loop can be exited from any point in the code, using the `break` statement.

The `break` statement shouldn't be unfamiliar; it exists in most modern programming languages. The essential function of the statement is to drop out of whatever loop you are in. Normally, this is done in response to a specific condition being reached. For example, consider the following code:

```
done = False
while not done:
print("Enter a command: ")
 cmd = get_a_command()
if cmd.lower() == 'quit':
   break
 print("Processing command: " + cmd)
```

This code snippet will process commands, presumably from some sort of input function. Perhaps it is reading them from a file, or from the user console, or maybe it is just playing back a list of commands to do a test. Whatever the case, we want to continue doing so until the command is `quit`. When this command is encountered,

the break statement is triggered and control passes to the first statement following the end of the loop.

The important part of understanding that last statement is to realize that if there are nested loops, a break statement exits the loop in which it is executed. Let's look at an example of a nested loop that requires the innermost loop to exit on a given condition:

```
i = 0
j = 0
while i < 3:
  while j < 3:
  if i == j:
   break
print("i = {0} and j = {1}".format(i,j))
j = j + 1
i = i + 1
```

This set of loop's purpose appears to be to find the combinations of values where the two indices are not the same. In fact, the output from this loop is as follows:

```
i = 1 and j = 0
i = 2 and j = 1
```

As you can see, the break statement here does not exit all the way out of the nested loop construct. It only exits out of the while `k < 3` loop, and picks up with the `I = I + 1` statement in the outer loop. The `break` statement is often the final alternative for getting out of a sticky situation in a loop, so remember it when writing complex looping code.

Like the `break` statement, the continue statement also alters the flow of a loop. Rather than dropping out of the loop, however, the continue statement goes to the end of the loop in which it is executed, and starts from the beginning again. This can be useful if an error is encountered, or if the code cannot handle a specific use case. For example, suppose you want to go between two values, dividing the current setting of the value into a given constant. You might write code that looks something like this:

```
start = -3
end = 3
idx = start
while idx <= end:
  val= 12345 / idx
```

```
   print(val)
  idx = idx + 1
```

This code seems all fine and dandy, but it has a rather nasty flaw. If you run the code, you will see that flaw:

```
Traceback (most recent call last):
    val = 12345 / idx
    ZeroDivisionError: integer division or modulo by zero
```

Oh dear, we don't want to be dividing by zero. But we do want all of the other cases to run. This is an excellent case for the continue statement, right? Well, not exactly. Suppose we did this:

```
start = -3
end = 3
idx = start
while idx <= end:
   if idx == 0:
  continue
 val = 12345 / idx
   print(val)
 idx = idx + 1
```

This code appears to work, the loop will find the zero value and drop to the bottom, executing from the next iteration, right? No, not quite. As mentioned earlier, the continue statement goes to the very end, starting at the top. So the idx value is never incremented and the loop will continue forever. An infinite loop!

Instead of creating an infinite loop and spending days figuring out why our program never terminates, let's fix it. The continue statement can be placed anywhere, all code will be executed normally until it is hit. So let's modify our code to look like this:

```
start = -3
end = 3
idx = start
while idx <= end:
  if idx == 0:
   idx = idx + 1
 continue
val = 12345 / idx
```

```
printval)
idx = idx + 1
```

Running this code results in the following output:

```
-4115
-6173
-12345
12345
6172
4115
```

This output is exactly what we were looking for. One last thing before we move on to the next subject. Python has a very strange construct that doesn't really appear useful until you need it. This is the else statement in a loop. This definitely sounds strange, doesn't it? Why would a loop contain an else statement? The else statements are for if statements, not loops. Yet, when you see how it works, you immediately understand it and wonder why nobody else thought of this.

Here's how it happens. If a loop terminates normally, that is, runs through the condition for which it was set up, the else statement is called. It is a kind of *all went well* extension to the loop. On the other hand, if a break statement is encountered in the loop, the `else` statement is not called. You can use the else in both `for` and `while` loops. Let's look at an example because this really does seem very strange.

```
signal_values = [1,5,9,99]


test_value = 3
for v in signal_values:
if v == test_value:
  break
   else:
     print("The value was not found!")
```

If you run this code, you will see the output `The value was not found!`. However, if you modify the value of the `test_value` variable to be `5`, you'll see no output at all. This is because the value was found in the `signal_values` array. Yes, you could simply test for this using `test_value in signal_values`, that isn't really the point to this exercise. The idea is, you either exited the loop due to an error condition, or to finding something before you reached the end. Normally, we would write something like this in another language:

```
signal_values = [1,5,9,99]

test_value = 3
found_it = False
for v in signal_values:
 if v == test_value:
    found_it = True
   break

if not found_it:
 print("The value was not found!")
```

In Python, this isn't necessary, and leads to less error prone coding. If we forgot to set the flag before exiting the loop, we might think we found the signal value, and proceed accordingly and wrongly.

The else statement can also be used in while loops. To see how, let's create our first *real* Python program of the book, a very simple guessing game. We won't talk about the input parts quite yet, just accept that `raw_input` inputs a string from the user console:

```
import random


real_value = random.randint(1, 100)
guesses = 0
while guesses < 10:
  guess = int(raw_input('Enter your guess: '))
 if guess == real_value:
   print("You got it!")
  break
 elif guess < real_value:
   print("Too low!")
  elif guess > real_value:
     print("Too high!")
   guesses = guesses + 1
else:
   print("Better luck next time!")
```

Here are two output runs of the little program. The first is a successful guess, the second a failure:

```
Enter your guess: 50
Too high!
Enter your guess: 30
Too low!
Enter your guess: 40
Too low!
Enter your guess: 45
Too low!
Enter your guess: 48
Too low!
Enter your guess: 49
You got it!
And the second:
Enter your guess: 5
Too low!
Enter your guess: 90
Too low!
Enter your guess: 35
Too low!
Enter your guess: 45
Too low!
Enter your guess: 55
Too low!
Enter your guess: 65
Too low!
Enter your guess: 75
Too low!
Enter your guess: 86
Too low!
Enter your guess: 1
```

```
Too low!
Enter your guess: 2
Too low!
Better luck next time!
```

As you can see, the else part only triggered when the guess count exceeded our limit. It also shows that someone is a terrible guesser.

That concludes our discussion of loops. Between the basic loop structure and the basic conditional structure, you can see that you have the vast majority of the basic functionality handled for writing Python code. In fact, you could, and we did, write a complete application using nothing but these processing constructs.

Python, however, was meant to not only be useful, but be reusable and maintainable. Being reusable means not to repeat yourself in your own code. There is a programming acronym called **DRY** that says just that. **Don't Repeat Yourself**. To make code easily portable, and easy to reuse, we need something that encapsulates that code into pieces that can be moved around. Which brings us to our next subject, functions.

Functions

Every modern programming language has the notion of a function. A function is simply an encapsulation of a block of code. It can take input in the form of parameters, and it can produce output, in the form of returned values. Python supports functions as first-class citizens, which means that you can assign a function to a variable, and execute it from that variable. To begin with, let's take a look at the basic form of a function.

In Python, a function looks like this:

```
def function_name(arguments):
    # One or more statements to accomplish something
    return (optional)
```

The `function_name` is a user-defined name for this function. Unlike some programming languages, such as Java or C++, you cannot have multiple functions of the same name with different arguments in Python. It isn't an error; it just absolutely won't do what you expect it to do. For example:

```
def func1(a, b, c):
 sum = c
 for i in range(a,b):
    sum = sum + i
   return sum
```

```
def func1(a,b):
   sum = 0
  for i in range(a,b):
sum = sum + i
  return sum

print(func1(1, 2, 3))
```

You would expect that this would call the first function, which accepts three arguments, but it does not work. Rather, running this snippet results in the following error from the Python interpreter:

```
Traceback (most recent call last):
File "/Users/mtelles/PycharmProjects/html/func_test.py", line 14, in <module>
    print(func1(1, 2, 3))
TypeError: func1() takes exactly 2 arguments (3 given)
```

On the other hand, calling this function:

```
print(func1(1, 2))
```

Works fine and returns the expected value of `1`. There are ways around this, as we will see shortly.

Parameters

Parameters are arguments to the function. As we'll see in a bit, they can be required or optional, and can be referred to by position or by name. For the moment, we'll assume that all of the parameters are required and passed by position, as they are in most languages. Most compiled languages require you to pass arguments by position, meaning that the order of the arguments passed in has to match the order that are processed by the function. In other words, if you have a function in Java or C++ or C# that is defined as `func(a,b,c,d)` then the parameters you pass to the function have to match `a,b,c,` and d in that order.

Parameters can be of any valid Python type, including user defined types like classes.

Return values

A return value is the value that is returned to the caller of the function or method. By default, a Python function returns `None`. That means, if you have a function like this:

```
def a_function():
```

```
print"This is a function")
```

And you call this function like this:

```
x = a_function()
print(x)
```

You will see output like this:

```
This is a function
None
```

The first line is output by the function itself, the second line is the value of `x`. None is the default value for anything, so it is what you get back if the function returns no values. Returning a value is done by, you guessed it, the `return` statement!

We could modify the above function to return a Boolean value:

```
def a_function():
print("This is a function")
 return True


x = a_function()
print(x)
```

The output from this function is, as you expect.

```
This is a function
True
```

Functions can return multiple values. The return value is a tuple containing the values returned from the code in the function:

```
def function_returning_multiple_values(a,b,c):
  return a*2,b*3,c*4

print(function_returning_multiple_values(1,2,3))
The output here is: (2, 6, 12)
```

Python passes parameters by reference. That means, if you change what a variable refers to (or, as you might say in Java or C++, points at), it will be modified in the calling code. You cannot change what an ordinary type, such as int or float, points to, so it will remain the same:

```
def function_that_changes_input(a):
    print("Before, a = "+str(a))
```

```
    a = a * 2
    print("After, a = "+str(a))

a = 10
print(function_that_changes_input(a))
print("Outside, a = " + str(a))
Before, a = 10
After, a = 20
None
Outside, a = 10
```

If you think about it, this makes perfect sense. Imagine that we passed the value 1 to the `function_that_changes_a_value` function. Would the value of `1` now be `2` throughout the system? That wouldn't make a great deal of sense, and Python strives to always make sense.

If the parameter can change what it refers to, however, the output is quite different:

```
def add_a_value(list_to_add_to, value):
list_to_add_to.append(value)
return list_to_add_to

x = []
print(add_a_value(x, 12))
print(x)
This, not surprisingly, returns:
[12]
[12]
```

Once again, this makes sense, since an array is a pointer to a block of memory. You can't change the memory address itself, but you can easily change what that memory address contains.

As mentioned, you cannot change a simple variable inside of a function directly. But you can return a value and assign that value to the variable:

```
def set_x_times_2(x):
return x*2

x = 1.0
```

```
x = set_x_times_2(x)
print(x)
```

Required vs optional arguments

Remember that we said that you can't do function overloading in Python, because creating two functions of the same name simply overwrites the first function with the second. It is untrue, however, that you can't have a function that takes differing numbers of arguments, and the method to accomplish this is with required and optional parameters.

A required parameter is one that has no default value. An optional parameter is one that is given a default value in the function header, and can either be set or not set by the caller of the function. Let's look at an example:

```
def a_function_that_takes_variable_arguments(a, b='hello', c=2.0):
    s = b + ' ' + str(c) + ' ' + str(a)
    return s
```

Once we have this function defined, you will notice that one argument ('a') does not have a value assigned to it, while the other two ('b' and 'c') have values. These values are called default values, and will be assigned to the variables if the caller does not specify them. We can now call this function three different ways.

```
print(a_function_that_takes_variable_arguments(9))
print(a_function_that_takes_variable_arguments(10, 'goodbye'))
print(a_function_that_takes_variable_arguments(11, 'why', 3.0))
```

The output from these three calls is:

```
hello 2.0 9
goodbye 2.0 10
why 3.0 11
```

This shows that our default arguments were set properly, and that we can override those defaults if we want to. The a parameter is called required, because you must specify a value for it. The other parameters are optional, because if you do not specify values for them, the interpreter will use the default values.

Keyword arguments

Python allows something that most other languages do not. For the majority of programming languages, when you define a function with three arguments; `a,b,c` you have to pass in the values in that order for them to match up within the function code. In Python, you can pass arguments that way, but you have another option. You

can also pass them by the name they have in the function, and when you do, you can pass them in any order you like. For example:

```
def print_name_and_address( name, address, city, state, zip):
    print("Name: " + name)
    print("Address: " + address)
    print("City, State, Zip: {0}, {1}, {2}".format(city, state, zip))
```

Now, if we want to call this function, we can call it with the arguments in the order they are defined:

```
print("Calling it in order:")
print_name_and_address('Matt Telles', '1313 Mockingbird Lane', 'New York', 'NY', '10012')
```

But, there is another way to do it:

```
my_name = "Matt Telles"
my_address = "1212 Fleming Circle"
my_city = "Trenton"
my_state = "NJ"
my_zip = "20123"
print_name_and_address(address=my_address, name=my_name, zip=my_zip, city=my_city)
```

Note that the order we are passing the arguments in is completely different from the order that the function defines, yet we are getting the correct output:

```
Name: Matt Telles
Address: 1212 Fleming Circle
City, State, Zip: Trenton, NJ, 20123
```

How is this possible? The answer lies in the way that Python actually uses parameters. For all Python functions, the arguments are passed as a dictionary, which is unpacked to pass the arguments in order. If you specify names for the parameters, those names go into the dictionary in the correct positions, and then the unpacking proceeds as expected. It is actually somewhat safer to pass values by name, because no matter how the function is modified to add new parameters, your arguments will remain the same.

Variable length arguments

Way back in the early days of C and C++, macros were used to introduce the concept of variable length argument lists. You could allow the user to pass one, two, or a

multitude of arguments to your function. Originally, this was designed for things like the print function in Python, a way of outputting a random number of things at once. It was much nicer to be able to write a printf statement like `printf("%d, %d = %s\n", 1, 2, "This is a test")` rather than having to output each piece of it as once. As people realized the advantages of variable lists, it became a more common (some would say it became too common) way of manipulating lists of information. Python supports variable numbers of arguments via the `*<argument>` syntax:

```
def variable_length_argument_function(fixed_string, *list_of_args):
   print("The fixed string is " + fixed_string)
   for var in list_of_args:
      print"Argument: " + str(var))
```

There is no set requirement for the types of the variable arguments, as you can see from this function call and its result:

```
variable_length_argument_function ('This is a fixed string', 1, 2.0, 'an optional string', complex(2, 3))
The fixed string is This is a fixed string
Argument: 1
Argument: 2.0
Argument: an optional string
Argument: (2+3j)
```

Now, you might be asking yourself, is it possible to combine the variable arguments with the keyword arguments and have a variable list of keywords and values? Hey, this is Python, your wish is the language's desire. The functionality you are looking for is encapsulated in the `**kwargs` parameter:

```
def keyword_variable_arguments(**kwargs):
 for kw, kv in kwargs.items():
   print("Keyword {0} = {1}".format(kw, kv))

keyword_variable_arguments(a=1, f=2.0, c='Hello world')
Keyword a = 1
Keyword c = Hello world
Keyword f = 2.0
```

Lambdas

Lambda functions are fairly new to most of the programming world. Java has had them for a few years; C# and C++ have added them in very recent versions. In

Python, however, lambdas have been there since Python 2.x. A lambda function is an *anonymous* function that is a function that has no name. A lambda is normally implemented in a single line of code. Lambda functions can accept arguments, and return values. They really are just functions, but allow for some interesting shortcuts in Python.

There is absolutely nothing that you can do in a lambda that you cannot do in a function, although there is some syntactic sugar that makes it easier to do strange and complicated things with lambdas. On the other hand, there are severe restrictions on what you can do in a lambda. You cannot have multiple statements. You cannot define variables, and the return value of the lambda is always implicit. With all of that said, let's look at why you would want to use a lambda, and what other use cases there might be.

The basic form of a lambda is `lambda (args): (manipulation)` where `args` is a set of variable names to use as arguments to the thing, and `manipulation` is a single line that somehow manipulates the arguments. The return value from the lambda is the last evaluated value within it. Python always sets the value of the last evaluation on the stack, so essentially this is just popping the stack. It is important to note that your lambda manipulation must result in a value, so you can't use it for printing or other non-value manipulations.

Here is the world's simplest `lamda` function:

```
lambda x,y: x+y
```

This lambda does nothing except to add the two arguments. However, creating a lambda like this doesn't make it possible to use it. It is an anonymous function, after all. So, how do we use lambdas?

There are two direct ways to use a lambda function. The first is to use it as an argument to a function call. We saw an example of this when we looked the `sorted()` function last chapter. This chapter allows you to assign a lambda value to the *key* argument of the function. The second way is that we can assign a variable to the lambda. Because functions, and thus lambdas, are first class citizens in Python, you can use it as if it were another kind of variable

```
foo = lambda x,y: x+y

print(foo(1,2))
```

You can then use the foo variable anywhere you would use a function, as shown in the print statement.

To be fair, lambdas are generally not worth the time. They are hard to read, confusing to debug, and difficult to maintain and document. With that said, they do serve a purpose, such as with the `sorted()` function example. If you need to do anything beyond very simple manipulations, however, writing your own function is the way to go almost every time. A function can be plugged in wherever you have a lambda anyway.

Classes

Python is an object oriented language. By this we mean that we think in terms of modeling the real world using code based *objects*. These objects are usually implemented in the form of a class, and Python is no exception. You are certainly familiar with the object oriented paradigm by now, with its focus on encapsulation and extension. The Python class is an excellent example of how to do **object-oriented programming (OOP)** well. Let's take a look at how Python classes work, how to use them, and how to extend them.

First of all, a class definition looks like this:

```
class <className>:

def <method>:
```

From a syntactical point of view, that's about it. Anything defined between the class name and any given defined method is called a **class level variable**. Most OOP languages have some form of the class level variable, as well as local and global variables. We'll take a look at the options you have in Python in a little bit when we talk about scoping.

Classes generally have two important methods that the system needs to know about, the initialization and finalization (sometimes called a destructor in C++) methods. Python has these as well. The initialization method, not surprisingly, is called `__init__` while the finalization or destructor method is called `__del__`. In general, the rules for naming of Python methods are to use a double underscore character for internal (not intended for direct access by the outside world) methods. There are no true private methods in Python classes, but convention says that a method that begins with a single underscore is considered private, and a double underscore indicates it is internal, only to be used by the system.

In C++ and Java, the object that you are operating upon in a class method is referred to by the `this` name. In Python, the `this` name is replaced by `self`. In fact, for an object method, `self` is always the first parameter passed to the method. That is to say, when you have an object called `foo` of class `Foo`, like this:

```
foo=Foo()
```

And the class `Foo` has a method called bar, which you call like this:

```
foo.bar()
```

This statement indicates that your object `foo` is calling the class method `bar`. The actual implementation of the bar method looks like this:

```
class Foo:

      .. some other methods ..

     def bar(self):
```

```
        .. do something with the object via the self parameter.
```

If your method takes parameters, you just add them after the self parameter, as you would with any other function:

```
def bar_with_parameters(self, a, b, c):
    .. do something with the object and parameters ..
```

In most programming languages, like C++ or Java, you have two sections to your class. First, you have a definition of the class variables. In Java, you might write this:

```
class Car {
  private int wheels;
  private float max_speed_mph
};
Likewise, in C++, you might have something like this:
class Car {
private:
    Int wheels;
    Float max_speed_mph;
};
```

Python has a similar concept, but does it in a different way. You can actually define class variables the same way if you want to:

```
class Foo:
wheels = 3
  max_speed_mph = 75.0

 def __init__(self):
    pass
 def printme(self):
  print(self.wheels)
print(self.max_speed_mph)
```

In this example, we are defining two class variables, called `wheels` and `max_speed_mph` which represent the number of wheels and the maximum speed in miles per hour, respectively. We are assigning them starting values of `3` and `75.0`, again respectively. We have defined two methods, the initialization method and a method

to print out the values within the `Foo` object. Clearly, a `Foo` is some kind of motorized vehicle. Bet you always wondered about that.

Note the use of the `self.wheels` syntax to access the variable within the method. If we write code to create a `Foo`, and call the method, we see the following:

```
foo = Foo()

foo.printme()

3

75.0
```

Note in the class definition, we implement the constructor, or initialization function, `__init__`, but the only statement in the method is the `noop` statement, pass. It is conventional to implement the `__init__` method even if you aren't doing anything with it, and if you don't want a method to do anything, the pass statement is ideal. As mentioned, though, it is not normal to implement the variables in this way. Why? Well, to understand what the two variables really are, let's do this:

```
print Foo.max_speed_mph

75.0
```

Note that the `Foo` in this case is the class `Foo`, not the instance. This is where the scope of the variable lies, at the class. However, these are not quite like static class variables in C++ or Java. For example:

```
foo = Foo()

foo.printme()

foo2 = Foo()

foo2.max_speed_mph = 80

foo.printme()

foo2.printme()

3

75.0

3

80
```

As you can see, the `max_speed_mph` is set for a specific instance of the object, and only that object picks up the new value. If we print it out again at the class level:

```
print Foo.max_speed_mph

75.0
```

We see that the value of the class variable hasn't changed. This is due to the way in which Python manages class and object variables. They are stored in a dictionary associated with the object, while class level items are in a separate dictionary associated with the class. If you don't assign the value in the class, it will inherit the class level value.

It is worth mentioning that in Python, class variables are normally called **attributes**.

We can actually look at the dictionary for a given class, it is stored in the `__dict__` internal variable for the class. You can alsolook at `__dict__` for an instance of the class (`object`):

```
print Foo.__dict__
```

```
{'__module__': '__main__', 'printme': <function printme at 0x10b8e6410>,
'wheels': 3, 'max_speed_mph': 75.0, '__doc__': None, '__init__': <function
__init__ at 0x10b8e6398>}
```

```
 print foo.__dict__
```

```
{}
```

```
print foo2.__dict__
```

```
{'max_speed_mph': 80}
```

Why does the `foo2` instance have a dictionary entry while the `foo` instance does not? Looking at the code, you will realize that there is never anything assigned to the `foo` instance. It only inherits the settings at the class level. The `foo2` also inherits the values from the class level, such as wheels, but overrides one of them, so it is set in the instance. Understanding how the internals works makes it possible to use Python much more efficiently, and to understand why things happen the way they do.

What if you don't want the variables set at the class level, but rather to have each instance have its complete own set of them? This is what the self value is all about. If you assign a variable at the self level, it is set for that instance and that instance only. Let's look at an example:

```
class Foo2:

   def __init__(self):
      self.wheels = 3
      self.max_speed_mph = 75.0

   def set_wheels(self, w):
      self.wheels = w
      return elf
```

```
    def set_max_speed_mph(self, mx):
        self.max_speed_mph = mx
        return self

    def get_wheels(self):
        return self.wheels

    def get_max_speed_mph(self):
      return self.max_speed_mph

    def printme(self):
      print(self.wheels)
    print(self.get_max_speed_mph())
```

As you can see, we are assigning to the `self` variable within both the construction/initializer and the various attribute setting methods of the class. Notice also how we return `self` from the set methods. This allows us to chain calls:

```
f = Foo2()
f.set_max_speed_mph(99.0).set_wheels(5)
f2 = Foo2()
f.printme()
f2.printme()
5
99.0
3
75.0
```

The chaining thing allows you to do more in a single line and with less typing. It is also clearer that we are assigning to a specific instance. Also notice that the initialization and set functions both assign to the self variable attributes. As a result:

```
print(f.__dict__)
print(f2.__dict__)
{'wheels': 5, 'max_speed_mph': 99.0}
{'wheels': 3, 'max_speed_mph': 75.0}
```

This shows you that each instance has its own complete copy of the variables. We can prove this even further:

```
print(Foo2.__dict__)
```

```
{'get_wheels': <function get_wheels at 0x10475a320>, '__module__': '__
main__', 'set_max_speed_mph': <function set_max_speed_mph at 0x10475a2a8>,
'printme': <function printme at 0x10475a410>, 'set_wheels': <function
set_wheels at 0x10475a1b8>, 'get_max_speed_mph': <function get_max_speed_
mph at 0x10475a398>, '__doc__': None, '__init__': <function __init__ at
0x104758f50>}
```

As you can see, there are no attributes defined at the class level for the `Foo2` class, they are all at the instance level. This is the more Pythonic way of doing things and the way we will do things from this point out in the book.

A few more things about classes, some of which you should never really use. First of all, because attributes are simply dictionary entries, you can remove an attribute from an instance by using the `del` operator:

```
del f2.max_speed_mph
```

```
print(f2.__dict__)
```

```
{'get_wheels': <function get_wheels at 0x10a44b320>, '__module__': '__
main__', 'set_max_speed_mph': <function set_max_speed_mph at 0x10a44b2a8>,
'printme': <function printme at 0x10a44b410>, 'set_wheels': <function
set_wheels at 0x10a44b1b8>, 'get_max_speed_mph': <function get_max_speed_
mph at 0x10a44b398>, '__doc__': None, '__init__': <function __init__ at
0x10a449f50>}
```

```
{'wheels': 3}
```

This is a very dangerous operation, and one you probably should never do. Why? Consider that the methods that access this property are still there. Depending on the order you call them, you might or might not get an error, but you will certainly not get what you expect:

```
del f2.max_speed_mph
```

```
print(f2.__dict__)
```

```
print(f2.get_max_speed_mph())
```

```
func_test.py", line 19, in get_max_speed_mph
```

```
    return self.max_speed_mph
```

```
AttributeError: Foo2 instance has no attribute 'max_speed_mph'
```

On the other hand, if you called the set method, all would be good, because you Python attributes are dynamic. You can add one at any time, retrieving one that doesn't exist is an error, because there is no dictionary entry for it.

Simple rule of thumb: Don't delete attributes from classes.

You can also delete objects, just as you can in C++, using the del operator:

```
del f2
```

This will remove the instance of the `Foo2` class called `f2` from the system. Referring to it will be an error and all memory for the object will be reclaimed. Using del is not normally necessary, because Python is a garbage collected language. This means that the interpreter periodically goes through and cleans up memory. It looks at all variables that no longer have anything accessing them and reclaims their memory. Deleting an object in C++ is a necessity, in Python and Java it is generally over-kill.

Finally, all attributes (sometimes called **properties**) in Python are public. That is, in the Foo2 example, we can do either of these and it will result in the same thing:

```
f2.max_speed_mph = 75.0
f2.printme()
f2.set_max_speed_mph(80.0)
f2.printme()
3
75.0
3
80.0
```

Scoping

In Python, variables have scope. Scope is the level at which the variable exists and when it disappears (or goes out of scope, in programming parlance). The scope of a variable is defined as the region of code space in which the name space requested is directly available. Basically, it means that the scope of a variable is where it can be found by the interpreter, and the programmer.

Python realistically has two levels of scope, those inside a function or method definition and those outside of it. There is a third level, which is a class level variable, as we've seen before.

Local variables are those defined within a method or function. These variables are only valid between the beginning and end of the method, and cannot be accessed outside the method.

Global variables are those defined completely outside of a class or function. Global variables can be accessed from anywhere in scope they are defined. For example, without importing another namespace, a global variable is accessible anywhere within the file in which it appears.

Class variables, of course, are accessible by themselves only within the class. They can be reached externally by prepending them with the name of the class, and then become a specialized form of the global variable.

Let's look at any example, because that makes things easier.

```
x = 3 # X is a global variable, accessible anywhere within this file

class Foo :

   def __init__(self):
      self.x = 10 # Class attribute x


   def print_stuff(self):
      print("Class X = {0}".format(self.x))
      print("Global x = {0}", x)


   def local_stuff(self):
      x = 5
      print("In local_stuff, x = {0}".format(x))

def function_1():
   print("In function, x = {0}".format(x))
   x = 10
   print("But locally it is: {0}".format(x))

print(x)
f = Foo()
f.print_stuff()
function_1()
```

Going by the rules we have established, what would you expect to see printed out? First, we are printing out the global variable `x`, so we'd see a `3`. Next, we print out the class attribute `x` and the global `x` in the `print_stuff` method of the `Foo` class. So, we'll see the class value, which is `10`, followed by the global variable, which again is `3`.

Next up, we print out values within the function, `function_1`. You would expect to see it print the global variable `x`, which is `3`, followed by the local value we just assigned, which is `10`. When we run this little program we don't quite get what we expect:

```
3
```

```
Traceback (most recent call last):
Class X = 10
Global x = 3
    File "/Users/mtelles/PycharmProjects/html/scope.py", line 26, in
<module>
    function_1()
    File "/Users/mtelles/PycharmProjects/html/scope.py", line 19, in
function_1
    print("In function, x = {0}".format(x))
UnboundLocalError: local variable 'x' referenced before assignment
```

Why the error? Python is smarter than we give it credit for. It realizes that x is defined in function function_1 and won't let you print it out until you have assigned it a value, which happens after the global print statement. Does this mean that the global variable is not accessible within a function that uses the same name? Actually, it doesn't mean that. You just have to be a little sneaky when you use it. Here's how you do it:

```
def function_1():
    print("In function, x = {0}".format(globals()['x']))
    x = 10
    print("But locally it is: {0}".format(x))
```

Running the program again, we see:

```
3
Class X = 10
Global x = 3
In function, x = 3
But locally it is: 10
```

What does that `globals()` thing mean? It is the method you use to access program global objects, returning a reference to the dictionary of all global entries. You can print it out:

```
print(globals())
{'f': <__main__.Foo instance at 0x107c3e488>, '__builtins__': <module
'__builtin__' (built-in)>, '__file__': '/Users/mtelles/PycharmProjects/
html/scope.py', 'function_1': <function function_1 at 0x107c3aed8>, '__
package__': None, 'x': 3, '__name__': '__main__', 'Foo': <class __main__.
Foo at 0x107bc3c18>, '__doc__': None}
```

Most of those things are housekeeping objects maintained by the Python interpreter, but you can see our x variable is in there too!

That's really all there is to scoping. If you are interested, there is a `locals()` function that returns all local variables as well. For a single file, it will essentially be the same as `globals()`.

Objects

An object is an instance of a class. You can think of it this way, as you would in any OOPs language, a class is a template for building something, like a blueprint. An object is the think you built using that template.

To see the difference, use the `type()` function in Python:

```
f = Foo()
print(type(f))
print(type(Foo))
<type 'instance'>
<type 'classobj'>
```

If you want to know if a given object is of a certain type, you can use the `is instance` function in Python:

```
class Foo:
def __init__(self):
   self.var = 1


class Foo2:
  def __init__(self):
   self.var_2 = 2


f=Foo()
if isinstance(f, Foo2):
  print("f is a Foo2")
elif isinstance(f, Foo):
 print("f is a Foo")
else:
 print("I don't know what f is")
```

Not surprisingly, the output is:

```
f is a Foo
```

That sums up classes in Python. We'll talk a little more about it when we talk about OOP later in the book, where you'll learn about things like inheritance, but for now you have enough knowledge to be dangerous.

Conclusion

In this chapter, you learned a lot about the *nuts and bolts* of Python, including the syntax of the language, a summary of the types of statements and constructs that can be created, and some of the functionality of the language itself. By now, you should be able to easily read a Python script, and write a simple script for yourself. In our next chapter, we will look at how to organize Python files into applications and packages.

Questions

1. What is the difference between a for loop and a while loop?
2. How does indentation work in Python?
3. What is the difference between a class and an instance?
4. How would I implement a set of conditional statements to check if a number was between 1 and 10, or 11 and 20, or 21 and 30?

Chapter 4
Organizational Skills

Introduction

In a minor departure from the learning of the individual pieces of Python, let's take a chapter off and look at some of the things you are going to need to be able to write Python programs. This isn't really about code, per se, it is about organization and tools that you will be using in the Python development model. Python has some very definite ideas about how things should be structured and used, and provides some very nice tools that are great if you do things right. To be a true Pythonista, you want to do things right. So, let's go!

Structure

- Immediate mode
- Modules
- Packages
- Importing
- Paths
- Requirements
- Dot notation in naming

Objectives

By the end of this chapter you should understand how to construct a Python application, how to use existing packages and create your own, and how to import existing packages and projects into your own application.

You should be able to write code in immediate mode, which we will look at, and how to create a simple module of your own.

Immediate mode

Let's talk about immediate mode. Immediate mode is the actual interaction directly with the Python interpreter. The immediate mode of Python allows you to either enter single lines and run them to test them, or to load (or enter, if you are a masochist) Python files and run them. It isn't a good substitute for a debugger or IDE, but there are some nice advantages to using it.

To begin with, to launch the interpreter in immediate mode slightly varies by the operating system you are using, but in general, just type python3 at your version of the Terminal/Command Prompt. You should see the Python interpreter running as in *Figure 4.1:*

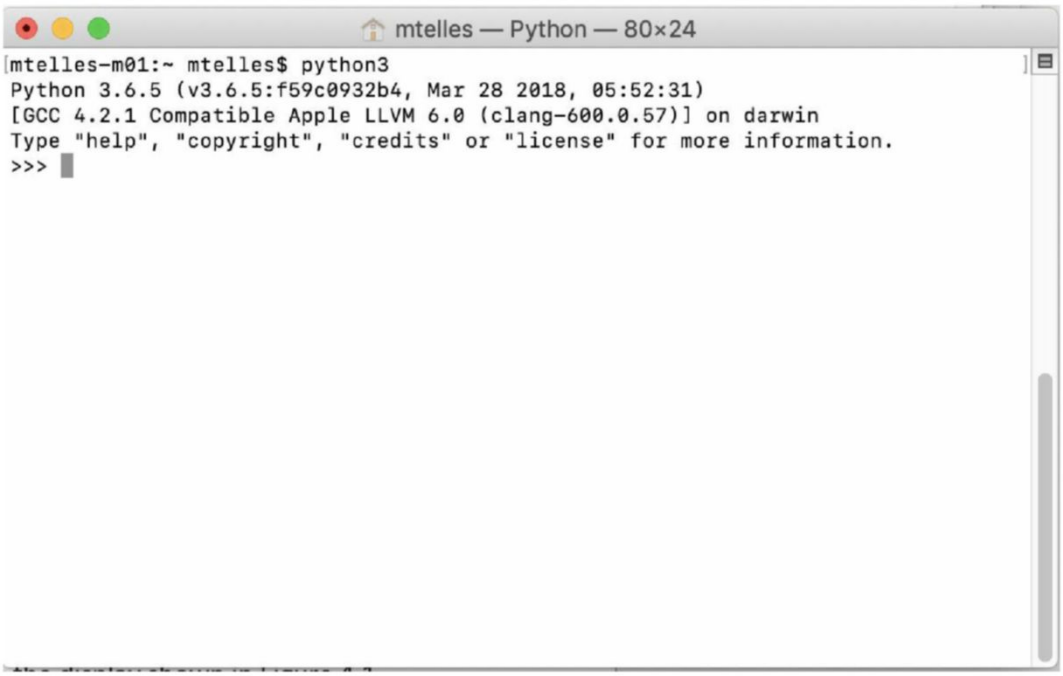

Figure 4.1: *The Python Interpreter in Immediate Mode*

You can type any valid Python statement at the prompt. For example, try typing:

```
>>> print(1+2+3)
6
```

Of course, the interpreter is capable of considerably more than this. You can enter an entire function into the interpreter:

```
>>> def func(x):
...     return x*x
...
```

In this case, the interpreter knows that your function isn't complete until you hit a blank line, so it shows the continuation prompt'.... If you make a mistake, you do have to go back and type the whole thing again, but on most systems, you can use the up and down arrows you bring up the lines you typed before and edit them if necessary. Then hit the return or enter key to re-add them to the memory of the interpreter.

Once you have entered the above function with no errors, you can call it in immediate mode:

```
>>> def func(x):
...     return x*x
...
>>> func(5)
25
>>>
```

As you can see, the interpreter prints out the last return value from the current expression. To see how this works, try just typing in an assignment statement:

```
>>> x = 10
>>>
>>> x
10
>>>
```

We've already looked at one of the Easter eggs in the Python interpreter, typing import this into the interpreter and hitting return. That one displays the Zen of Python. There's another one that's almost as fun. Type import antigravity and hit return. Rather than spoil the surprise, just try it. You do need to be connected to the Internet for it to work, though.

And one more. You can type `import __hello` into the interpreter, but only once:

```
>>> import __hello__
Hello world!
>>> import __hello__
```

One more thing you can do in the interpreter which really marks Python as different from all other languages. If you are trying to move from one version to another, or worry about the changes that are coming with a certain feature, there is a module called `__future__` that allows you to experiment with new things without breaking your code by upgrading. For example, type the following into your interpreter:

```
>>> from __future__ import annotations
>>> def func(x:'int') -> 'int':
...    return x*x
```

For Python 3.7, which is what this book is written using, the only future feature available is annotations. Annotations are a future feature that may or may not be in the next release. They allow you to give *hints* about the arguments and return values for functions. In our example above, you see that the parameter x is hinted to be an integer, as is the return of the function. Note that the interpreter doesn't enforce these hints. You can do something like this right after the above:

```
>>> print(func(5.2))
27.040000000000003
```

The point is to give the programmer an idea of what is expected, since Python by default does not have type information in function definitions.

The Python interpreter immediately executes code as it is written (aside from function or class definitions) as well as when it is loaded into the interpreter. You can load a given Python file into the interpreter to experiment with it using the `import` statement.

Create a simple file called `test3.py` and place the following lines into it:

```
print("This is a new file")
x = "Hello"
y = "world"
print(x + ' ' + y )
```

Now, launch the interpreter and type:

```
>>> import test3
This is a new file
Hello world
```

Note that the code in the file is immediately executed. If we modify the file to contain a function:

```
def my_function(x):
    print(x[1:])
```

Then we can load the file into the interpreter and run the function:

```
>>> import test3
This is a new file
Hello world
>>> test3.my_function("This is a test")
his is a test
```

Notice that we have to prefix the function name with the name of the module. This is the result of Python name space resolution. This prevents you from having issues if you have multiple files that contain a function with the same name.

Modules

In general, when you read *module* in Python, you can substitute *file*. A module is usually a file with a Python extension (`.py`) that contains code that you want to use in your application. A module can be the main entry point, or it can be a repository for code that you want to use, utilities and such. Aside from operating system limitations, there are no naming restrictions for a Python module.

A module can be reused in your code via the import statement. There's a lot you can do with importing, which we'll explore in just a moment, but for now, let's look at the basic form of the import statement with regards to using a file as a module.

```
import <namespace>.<module>
```

You don't use the `.py` part of the file name when you are importing it into another file, Python figures that out on its own. Let's create two separate files. We'll call them `main.py` and `utilities.py`. The main file will contain our main application, and the `utilities` file will contain some useful functionality that we need in our application.

1. First, create a new file in your favorite editor or IDE called `main.py`.
2. Add the following code to your `main.py`:

```
import utilities


# Input some data from the user, store each set in a dictionary.
done = False
```

```
d = {}
while not done:
    key = raw_input("Enter a key value: ")
    if len(key) == 0:
      done = True
  else:
value = raw_input("Enter a value for the key: ")
   if len(value):
      value = utilities.reverse_a_string(value)
        d[key] = value


utilities.print_a_dictionary(d)
```

3. Create a new file called `utilities.py`, and add the following code to it:

```
def print_a_dictionary(d):
   for k, v in d.items():
      print"Key: {0} = {1}".format(k, v))


def reverse_a_string(s):
   return s[::-1]
```

4. You have created a new Python project!

The important things to take away from a cursory inspection of the two files are as follows. The `main.py` file contains all of the immediate execution code. Immediate code is code that the interpreter will run as soon as it encounters it. The `utilities.py` file, on the other hand, contains no immediate execution code. All of the code in the utilities file is functions, which aren't executed until they are called in normal code.

The `main.py` code is made up of three sections. First, we have a loop that processes all of the user input until the user is done. This is done via the `while` loop. Next, we have the user input section, which reads some data from the user using the `raw_input` function. This section adds the data input from the user into our dictionary after running it through our `reverse_a_string` function, which we imported in the import line at the top of the file. Finally, we print out the dictionary via the `utilities` class `print_a_dictionary` function also found in the imported `utilities` file.

This sort of layout is quite common in the Python world. You will have one or more files that contain modules of code that is intended to be executed as it is loaded.

Normally, this is a single file, but it isn't required to be. Then you will have one or more files that contain functionality that is needed by the application. This might be a set of functions, or classes, or a combination of the two. The `main` file will import these files, and then use the functionality in them to accomplish its tasks, whatever they might be.

The next thing to notice is that once we have imported the `utilities` file, we then have to preface all of the functionality in the file with the name of the module. This is called the **namespace** of the functionality. The reason that it exists is simple, if you had multiple files with functions that had the same name, which is easy to do with very simple names, you would have name collision. If I have two files:

```
file1:

    Function_to_call

file2:

    Function_to_call
```

Now, in my `main` file, I do this:

```
import file1

import file2

Function_to_call()
```

Which one of the functions will the interpreter call? It can't be the first one, since that's non-deterministic. What if someone comes along and flips the two `import` statements around. It can't exactly ask, since this is a run-time decision. So, you as the programmer are required to tell the interpreter which one of the functions you want. This is done via the namespace. So, we could call:

```
file1.Function_to_call()

#Or

File2.Function_to_call()
```

This indicates clearly to the interpreter which one of the files you want to call the function out of, and the proper mapping is done to call it.

You might ask, what happens if your directory structure is not flat? For example, consider the following setup for files:

```
Main.py

util:

    Utilities.py

Formatting:

    Formatting.py
```

If you created this structure, and still used the `utilities.reverse_a_string` function, you would find that it created an error. In fact, the import statement itself would fail:

```
Traceback (most recent call last):
  File "/Users/mtelles/PycharmProjects/html/main.py", line 1, in <module>
    from utilities import utilities
ImportError: No module named utilities
```

Why do we receive the above error? Because there is no module named `utilities` in the same directory as the main file. You can fix this by modifying the `import` statement:

```
from util import utilities
```

Now, the interpreter knows that it should be looking in the `util` subdirectory for the `utilities` file. There are more complex scenarios here that we will talk about when we discuss things like packages and writing your own in Python.

You might wonder whether you can reuse modules that are stored in other places on your system. The answer is, yes, you can. In most systems, the Python interpreter uses an environment variable named `PYTHONPATH` to locate modules. So, if you had a previous project that stored all of its data in `/users/me/myprojects/project1`, and you were in `../project2`, you could add the path to `project1` in your `PYTHONPATH` and reuse the code in the previous project without having to move things around. Python is all about simplicity, remember?

Reuse is an important aspect of Python. Don't reinvent the wheel is the hallmark of a Python programmer, so once you have written something, it is important to make it generic enough to reuse in other projects. This is one reason that Python, unlike Java or C++ does not have the concept of a *namespace directive*, which often causes issues in multiple projects due to collisions and long identification strings.

Oh, finally, a module can contain more than just code, whether it is a function or a class. It can also contain data. So, for example, suppose that we wanted to include a module in our code that contained all of our test data in a series of dictionaries. That's very easy to do in Python.

Let's create a module in the `utils` directory that contains our test data and call it `test_dictionaries.py`, the file will contain the following entries:

```
test1 = {
    "data_value_1": "test",
    "data_value_2": "my name",
    "data_value_3": "my address"
}
```

```
test2 = {
    "data_value_1": "test2",
    "data_value_2": "my name 2",
    "data_value_3": "my address 2"
}

test3 = {
    "data_value_1": "test3",
    "data_value_2": "my name 3",
    "data_value_3": "my address 3"
}
```

Now, let's modify our main file, `main.py`, to read as follows:

```
from util import utilities
# Input some data from the user, store each set in a dictionary.

done = False
d = {}
while not done:
   key = raw_input("Enter a key value: ")
   if en(key) == 0:
      done = True
   else:
      value = raw_input("Enter a value for the key: ")
   if len(value):
      value = utilities.reverse_a_string(value)
    d[key] = value
utilities.print_a_dictionary(d)

from util import test_dictionaries

for k, v in test_dictionaries.test1.items():
   if k in d:
      print("You used test data!")
```

If we run this code, and enter the correct data, we'll see that it properly checks the test data from the other module:

```
Enter a key value: data_value_1

Enter a value for the key: fred

Enter a key value:

Key: data_value_1 = derf

You used test data!
```

A few things worth noting about this example. First of all, notice that you can place an import statement anywhere in your file and use things from it from that point downward. You cannot use something that is imported before your `import` statement.

Secondly, notice that we are using data from the imported module, rather than code. This can be very useful for using internationalized strings or, as we do here, test data.

You might wonder how you can find out what functions and classes exist within a module, if you are trying to use it in your own code. Documentation, after all, is something of an anathema to most developers, so the chances that they wrote extensive documentation for the modules they wrote, particularly in house, are small. Fortunately, Python comes to the rescue with the `dir()` function.

For our `utilities` package, for example, we could do this in the interactive mode (or in our application code):

```
Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 24 2018, 02:44:43)

[Clang 6.0 (clang-600.0.57)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>> from util import utilities

>>> dir(utilities)

['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__
name__', '__package__', '__spec__', 'print_a_dictionary', 'reverse_a_
string']
```

We can see that there are the usual system definitions, like name, package, doc, file and such, but also our two functions, `print_a_dictionary` and `reverse_a_string`.

A word about those other system definitions. Let's take a look at what values are stored in them. For example, let's look at the `__file__` variable:

```
>>> print(utilities.__file__)

<project-director>/util/utilities.py
```

In this case, `<project-directory>` isn't what you would see, that's simply where the file is stored on my system. On yours it will be the base directory of the project in which you stored the source code for this project. Likewise, we can look at the `__name__` variable:

```
>>> print(utilities.__name__)
util.utilities
```

This is the actual namespace name of the imported module. If we wanted to print out error messages based on the name, we could do so in our code. Now, what about `__package__`?

```
>>> print(utilities.__package__)
util
```

We are going to talk about packages next, but you can see that even though we haven't specifically made any statements about the `utilities` code being in a package, it is given one when it is loaded.

You might wonder if you can print out information about the file you are working in, such as to log errors and the like. The answer is, of course you can! The information shown for the `utilities` package above is defined for every module in Python. So, for example, place these lines at the top of the `main.py` file:

```
print(__file__)
print(__package__)
```

The output will be something like:

```
<project-dir>/main.py
None
```

The packages name is displayed as none because it isn't in a package directory of its own. We'll see how you accomplish defining and using packages right now. The important thing to remember about modules, for right now, is that if your files are in the same directory, use the syntax:

```
import <file>
```

Whereas if the file is in a subdirectory, or another directory loaded in via the `PYTHONPATH` environment variable, use the syntax:

```
from <package-name> import <file>
```

Packages

A package in Python is simply a directory that contains files, and potentially other subdirectories (and thus packages) of its own. The name of the package is the

name of the subdirectory in which the code lives. Unlike modules, packages can be bundled together and distributed for use by other developers as full-blown entities, not simply files to be added to a project.

The single biggest difference between a package and a module is that a package is in a subdirectory and contains a file called `__init__.py`. This file, which must have that exact name, defines the directory as containing a package. The file does not have to contain anything, although it can contain things like global data or initialization for the package.

If the `__init__.py` file is not empty, it may contain one special variable, which is the `__all__` variable. This variable contains a list of all symbols within the directory that are `exported`, which means they are available for use by other applications. If there is an `__all__` variable, and it does not contain the name of a module found in that directory, that module will not be available via the import statement:

```
__all__ = ["bar"]
```

If you are accustomed to working with libraries in Java or C++, packages will seem a little strange, but the concepts are exactly the same.

Let's begin by building a package. Our package is going to contain three files. First, there will be the `__init__.py` file which defines our directory as containing a package. Next, we'll create a file called `funcs.py` that will contain a few functions we can use in our application. Finally, we'll have a `classes.py` file that contains the classes we are going to create. This will be a very simple package, so there will only be a couple of functions and a single class.

Our class, for this package, will be a Card, which models a playing card:

```
class PlayingCard :

    def __init__(self):
        self.suit = 0
        self.card_value = 0


    def set_suit(self, s):
        self.suit = s
        return self


    def set_rank(self, r):
        self.card_value = r
        return self
```

```
    def suit_name(self):
        suits = ['spades', 'hearts', 'clubs', 'diamonds']
        return suits[self.suit]

    def display(self):
        if self.card_value > 1 and self.card_value <= 10:
            print("the {0} of ".format(self.card_value))
        elif self.card_value == 1:
            print("the ace of ")
        elif self.card_value == 11:
            print("the jack of ")
        elif self.card_value == 12:
            print("the queen of ")
        elif self.card_value == 13:
            print("the king of ")
        print(self.suit_name())
```

There is nothing mysterious or magical here. Our class models a playing card with a suit (spades, hearts, diamonds, or clubs), and a rank. The rank models the number on the card, or for face cards, the value of the jack, queen, king, or ace.

Our `funcs.py` file, on the other hand, will contain a single function, which generates an unshuffled deck of cards. Place the following code in the `funcs.py` file:

```
import classes

def generate_deck():
   deck = []
   for i in range(0,4):
      for j in range(1,14):
         c = classes.PlayingCard()
      c.set_suit(i)
   c.set_rank(j)
deck.append(c)

   return deck
```

Again, nothing surprising. We just generate a deck of fifty-two cards, with thirteen per suit. If you happen to enjoy playing cards with fewer cards, feel free to modify the code, but your users may not thank you for it.

Now, create a file at the root level of your project called `card_game.py`. Place the following code in the file:

```
from cards import funcs


deck = funcs.generate_deck()


for card in deck:
    card.display()
```

Running this main program will result in a display that looks like this:

```
the ace of
spades
the 2 of
spades
the 3 of
Spades
<omitted for space>
the jack of
diamonds
the queen of
diamonds
the king of
diamonds
```

As you can see, we have successfully created a new package, imported it into our application, and used it to create a very basic card game!

One last comment about packages. You will notice that in the `card_game.py` file, we import the `funcs` from the package. We might have to import numerous modules from the package and it can be tedious to write each one on its own line. There is a solution to this. We can put them all on a single line (up until we hit 79 characters!):

```
from cards import funcs, classes
```

Note that when importing from a module, rather than a package, we can import all of the functionality in that file by writing:

```
from my_module import *
```

This, however, is frowned upon by Pythonistas, as it clutters up the code and isn't clear about what you are trying to load. Finally, it is slightly memory wasteful.

Finally, a module is just a set of Python scripts. Within a module directory, you can have executable scripts and run them, just like they were anywhere else in any other file. So, for example, it is often desirable to have a test driver in your package directory to test your code.

Python has a module caching system for loading modules. You rarely need to know about it, but it is important when trying to optimize code. The first thing the interpreter does when it encounters any sort of import request is to look at the `sys.modules` list. This list is updated as new modules and packages are imported. If the module or package was previously imported, it will be found in the cache and can be very quickly loaded back in for execution.

If a module is not found in the cache, the Python finders are used to discover where a given module might be found. The finder system is not quite as simple as looking through the paths and directories in the `PYTHONPATH` environment variable. In fact, the system will first look through the sys.modules list, and then through the `sys.meta_paths` list. This is the functionality that was originally used to create virtual environments. Because the finder system is quite extensible, it is possible to write your own finder which returns enough information for the loader system to retrieve the code and parse it.

Writing a custom module loader is well beyond the scope of this book, but knowing that it exists can often aid you in creating extensible programs.

Importing

Importing a package or module makes it available for use in the file into which it is imported. Import a file specific, you cannot import something *globally* except by importing something that imports something else. For example, consider the card example that we used earlier. In the `card_game.py` file, we imported `funcs` from our `card` package. Yet, we have code that looks like this:

```
for card in deck:
   card.display()
```

The display method of the `Card` class isn't defined in the `funcs.py` file in the package. So how does Python know about it? The answer is that `funcs.py` knows about it because it imports the `classes.py` file at the top:

```
import classes

def generate_deck():
```

Importing a file essentially copies that file and all of the file included by that file into your own source code file (this isn't what happens, but you can think of it that way). That means that you get the advantage of having all of the code that was needed by whatever you called.

You can actually duplicate what the Python interpreter is doing when it imports a module in your own code. After all, Python is just Python, and there's not much in the interpreter that you can't call. In this case, there is a special package in Python 3.x called importlib.

Take a look at this snippet of code first, then we can break it down to understand what is going on. This is fairly advanced stuff, and not something you are likely to do on a daily basis, but as a professional, you want to understand what is going on without blindly accepting it.

```
import importlib

module_obj = importlib.import_module('cards.classes', 'cards')

cls = getattr(module_obj, 'Card')

c = cls()

c.set_suit(1)

c.set_rank(10)

c.display()

the 10 of

Hearts
```

Let's take this apart, piece by piece to understand it. First, we import the `importlib` module. This module is part of the Python system library, so no additional work needs to be done to tell the interpreter where to find things.

Next up, we call the `import_module` method of the `importlib` package to retrieve an object representing the module itself. You can think of this as a handle to the cards package, and specifically to the `classes` module within that package.

The `getattr` function retrieves an object based on a string and a handle to the module in which that object definition resides. In this case, we are using the dynamically loaded handle to the user-defined packages `cards.classes`. The specific class we want to instantiate is called `Card`, which is found in the `classes` module. The return from this function call is an object. We can't call it a `Card` object in the code, because Python doesn't really know what a `Card` object is here. It is just a `class` object.

Next up, we instantiate a specific instance of the class with the line that reads `c = cls()`. This calls the constructor (`__init__`) for the class. For our Card class, there are no arguments needed, but if there were, you would pass them in the `cls()` call. At this point, we have a `Card` object and can call `Card` methods on it.

The next three calls are to the `Card` class and simply set some internal pieces and then invoke the `display()` method to show the output to the user.

A note here. If you are viewing this code within an IDE, it may or may not accept the lines that call specific parts of the Card class. This is because the IDE doesn't know what the object is until runtime. The interpreter, however, will have no problem with it and it will work properly.

This really shows off the power of the Python environment and the sorts of things that you can do with it. In this case, we could retrieve an object from a given package name, and create an object of a specific class name, all without putting the names into the module we are writing. It is all done dynamically!

Paths

As we have mentioned, the python interpreter looks for things using environment variables. There are really three variables that matter when you are working with Python.

First, we have the `PYTHONPATH` variable. This is the environment variable that is used by the Python interpreter to find modules for the project you are working with. The `PYTHONPATH` variable can be set to different places for different projects. The standard is to set it to the root of the project, but you can also pull in modules from different places by appending your paths to the environment variable. You can actually modify this from within your own code, if you like, as we'll see shortly.

Python packages, on the other hand, follow a slightly more complicated path to locating for Python. In its simplest form, Python imports packages by following the value stored in the `sys.path` variable. You can look at the current value of `sys.path` using the following simple code:

```
>>> import sys
>>> print(sys.path())
```

Running this code will show you something that probably looks a lot like your system path (for Windows) or profile path (for Mac/Linux). In addition to the basic stuff in your path, however, you'll see things like:

```
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python37.zip',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages']
```

These paths are added to the location path during the installation of Python, or by the virtual environment system when it is started up. As mentioned, though, we can change this at runtime if we want to. Let's imagine that you want to change the path to include some third-party directory that we read from a configuration file. We won't go through the work of reading the configuration file here, let's just assume that it has been read and stored in the `additional_path` variable. We can now modify the system path by doing this:

```
import sys

additional_path='/usr/local/bin'

print(sys.path)

sys.path.append(additional_path)

print(sys.path)
```

In this example, we are printing out the path before and after appending to it. Your output will vary, but will look something like this:

```
['/System/Library/Frameworks/Python.framework/Versions/2.7/lib/
python2.7', '/System/Library/Frameworks/Python.framework/Versions/2.7/
lib/python2.7/plat-darwin', '/System/Library/Frameworks/Python.
framework/Versions/2.7/lib/python2.7/lib-tk', '/System/Library/
Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac', '/
System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/
plat-mac/lib-scriptpackages', '/Users/mtelles/PycharmProjects/html/venv/
lib/python2.7/site-packages']

['/System/Library/Frameworks/Python.framework/Versions/2.7/lib/
python2.7', '/System/Library/Frameworks/Python.framework/Versions/2.7/
lib/python2.7/plat-darwin', '/System/Library/Frameworks/Python.
framework/Versions/2.7/lib/python2.7/lib-tk', '/System/Library/
Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac', '/
System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/
plat-mac/lib-scriptpackages', '/Users/mtelles/PycharmProjects/html/venv/
lib/python2.7/site-packages', '/usr/local/bin']
```

As you can see, we've added to the path, and it will be followed by the interpreter when trying to load packages. It is very important to note that modifying the path will only affect the current run of the program. Once you exit your current Python script, that path will revert back to what it was beforehand. In addition, if you launch a separate Python program from your own (we'll look at how to do this using `exec()`) it will not pick up these changes.

Dot notation in naming

As you may have noticed, Python has a consistent strategy for naming imports. You have the directory structure in front of the module that you want. So, if you have a directory structure that looks like this:

```
A
|_ B
      |_C
           |_ module.py
```

Then the way that you reference this in Python, presuming that the A directory is within your Python path is to use:

```
A.B.C.module
```

If you keep this in mind going forward, things will be considerably easier. This is really not unlike the Java directory structure, where you have your project followed by `src` and company names underneath. C# and C++ don't have such things, they refer to their `modules` using an include path.

Installing packages using pip

You have probably heard, or read, that one of the main reasons that programmers flock to Python is the wide array of pre-built packages and libraries that can be used with the language to accomplish an ever growing list of tasks. From text processing to big data to artificial intelligence to installing systems in the cloud, Python has functionality to do almost anything you might need. This makes it ideal for creating corporate applications with little investment into the core functionality and more investment into the design and flow of the system for corporate needs. The most likely question to ask yourself is, how does all of this functionality get found and incorporated into my applications. The answer lies in two pieces, the pip program and the PyPI website and services. Let's look at them in somewhat reverse order.

In your browser, you can visit **https://pypi.org/**, as shown in *Figure 4.2*. Here, you will find all of the packages that have been developed and shared by the Python community. These packages all are available for download and use in your programs, many with no costs associated at all. Python is very community oriented, and most packages are distributed in an open-source licensing model. To find a package you are interested in, use the search bar in the middle of the page to enter search terms, or the name of the package you want. For example, you may have found a package in an online discussion board that does what you need, and want to find out more

about it. Let's say, for argument's sake, that you are interested in a package you found called *requests*, which does HTTP gets and posts easily:

Figure 4.2: *The PyPI website*

Entering the search term requests will return a result list like that one shown in *Figure 4.2*. This result list may contain things other than what you are looking for, the search terms you use will determine how specific the results are, as with any search system. In this case, looking at the figure below, you'll see that there are two that are likely candidates, **`requests 2.2`** and **`requests 2.1.6`**. Your own specific results may vary; these packages are updated all of the time.

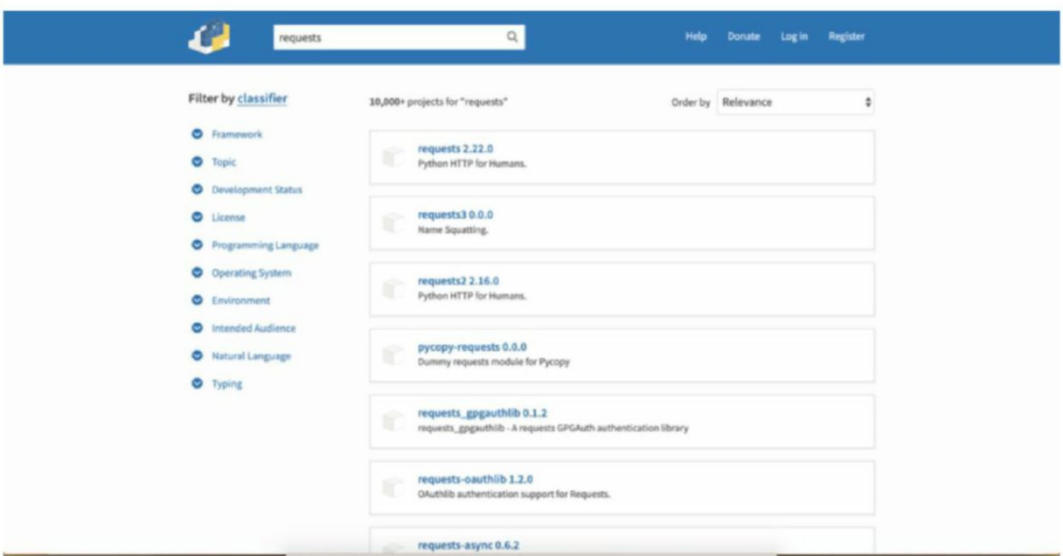

Figure 4.3: *Result list for requests package*

If you click on one of the results, say the first one, you will be taken to a details page which will tell you things about that package. On the left hand side of that page are two bits of information you should carefully check out. First, the license agreement for the package. In the case of requests, the license package is the OSI Apache Software License. The details can be found online, but in general, this means you can use the product freely in your own applications so long as you don't try to sell it by itself.

The second bit of information is a little lower down on the left-hand side, and tells you what versions of Python the package supports. In general, Python will be backwardly compatible with most versions. Exceptions are when the package uses something in a newer version, or when a breaking change (such as changing print to a function instead of a statement) are made in the language. This doesn't happen often.

Of course, once you have the name of your package and the version you want (if it is other than the default version), you need to somehow get that package installed on your system so that you can use it. This is where the PIP program comes into play.

Scanning further down the description page, you will run across a section called *Installation*. All packages in the PyPI.org system contain an installation section, although the vast majority of the time, you will only need to know the name, and perhaps the version, of the package you want installed.

The pip program does all the hard work of installing packages on your Python system. The general usage of the pip program is as follows:

```
pip <command><package-name>
```

Where:

- `command` is the thing you want to accomplish. Normally, this is one of list, install or uninstall, although there are a few others we'll talk about in a few moments.
- `package-name` is the name of the package you want to install on your system. In the case of the requests package we looked at above, that would be *requests*. As a note, if you wish to install a specific version of the package, as when you might want to try a beta release, or perhaps need an older version to get around a bug, you can specify it as well.
- `pip install requests==2.22.0`, for example, will install the specific version `2.22.0`. You can also specify a minimum and maximum version, just in case you want to make sure that the release is at least up to date, or perhaps anything before one that is causing you problems. To do this, use the syntax:

  `pip install package >= version` for a version of the package that is at least version level and pip install package <= version to make sure you don't get one beyond a certain level.

Oh, one more note here. For Python versions 3.x and above, `pip` normally ships with the entire environment. However, for older versions of Python, you may need to manually install it. To do so, follow these instructions:

- On the Mac, use `sudo easy_install pip`, or `brew install pip` if you use Homebrew.
- On Linux, use `sudo apt-get install python-pip or sudo yum install python-pip.`
- On Windows, most Python installations have a pip installer, or you can use the `get-pip.py` script.

If you just want to upgrade your version of `pip`, such as on the Mac which ships with a much older version, you can use this command:

```
pip install -U pip
```

If you are wondering what packages you already have installed, use the `pip list` command to find out:

```
pip list
```

```
DEPRECATION: Python 2.7 will reach the end of its life on January 1st,
2020. Please upgrade your Python as Python 2.7 won't be maintained after
that date. A future version of pip will drop support for Python 2.7.
Package                Version
---------------------- ---------
asn1crypto             0.24.0
certifi                2019.6.16
cffi                   1.11.5
chardet                3.0.4
Click                  7.0
cryptography           2.4.2
decorator              4.4.0
EasyProcess            0.2.7
enum34                 1.1.6
Flask                  1.0.3
gevent                 1.4.0
greenlet               0.4.15
hashids                1.2.0
idna                   2.6
```

```
ipaddress              1.0.22
itsdangerous           1.1.0
<list abbreviated for space>
```

Don't have a browser handy, and want to find packages without having to go to PyPI.org? Use the `pip search` command:

```
pip search requests
```

```
DEPRECATION: Python 2.7 will reach the end of its life on January 1st,
2020. Please upgrade your Python as Python 2.7 won't be maintained after
that date. A future version of pip will drop support for Python 2.7.
requests-hawk (1.0.0)          - requests-hawk
requests-dump (0.1.3)          - `requests-dump` provides hook functions
                                 for requests.
pydantic-requests (0.1.2)      - A pydantic integration with requests.
requests-foauth (0.1.1)        - Requests TransportAdapter for foauth.
                                 org!
requests-aws4auth (0.9)        - AWS4 authentication for Requests
requests-auth (4.0.1)          - Easy Authentication for Requests
Requests-OpenTracing (0.0.1)   - OpenTracing support for Requests
yamlsettings-requests (1.0.0)  - YamlSettings Request Extension
```

If you want to see information about a given package, use the `pip` show command. Note that this command will also show you information about packages you have installed that require the package you are inquiring about:

```
pip show requests
```

```
DEPRECATION: Python 2.7 will reach the end of its life on January 1st,
2020. Please upgrade your Python as Python 2.7 won't be maintained after
that date. A future version of pip will drop support for Python 2.7.
Name: requests
Version: 2.18.4
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
License: Apache 2.0
Location: /usr/local/lib/python2.7/site-packages
```

```
Requires: idna, urllib3, certifi, chardet
Required-by: locustio
```

In this example, you see that the requests package is required by `locustio`, something that is installed on my system. You also see that requests itself has requirements, the `urllib3`, `chardet`, `idna`, and `certifi` packages, all of which are part of the standard Python install.

To remove a package, type `pip uninstall <package-name>`. Note that if other packages were installed as prerequisites for the package

Finally, if you are feeling brave, and want to join the latest and greatest tools users, you can use `pipenv` instead of `pip`. The differences are minor, but the `pipenv` application will also automatically create a virtual environment and show you a graph of all of the dependencies for your project. It is not a standard yet, but it is definitely getting there, and learning about it can be useful. You can use pip and `pipenv` on the same system, so you can test it if you want.

To install `pipenv`, just use Homebrew on Mac: `brew install pipenv`. For Windows, consult the up to date documentation on **https://docs.pipenv.org/**.

One more important aspect of `pip` which we have to talk about before moving on to the next chapter, and that is setting up project requirements and installing a basic set of packages for a project.

Insuring requirements with pip

One of the more annoying things about multi-person projects is keeping everything in sync. As experienced developers, you've probably used tools like Git and Visual Studio to define projects and keep the source code up to date for multiple developers. In Java, for example, keeping track of all of the libraries that are used by your project can be a nightmare, even with tools like maven and the like. One of the strengths of Python is just how well it works with multi-developer projects. One of the reasons for this strength is the pip application.

How, then, do you keep all of the package requirements in sync when multiple developers may be using multiple packages in their pieces of the pie? The answer lies, again, in `pip`. This time, there are two commands that you will find useful; `freeze` and `install -r`.

The pip `freeze` command is used in two ways. First of all, it can show you exactly what packages you are using a given machine. In this, it is a lot like the `pip list` command. However, the freeze command goes a little bit further. It is capable of having its output redirected, so that you can store it in a file. The convention in pip is to use a `requirements.txt` file which can then be loaded by pip on another machine. The `pipfreeze` command exports the list in the exact format that the installer uses to re-load them. Here's how it works.

```
pip freeze > requirements.txt
```

This exports the requirements for the system into a text file called `requirements.txt`. You can use `pip freeze` to create it, or you can make one by hand, Python and the `pip module` do not care. As an example, here's one for a project:

```
cat requirements.txt

requests==2.18.4

pyvirtualdisplay==0.2.1

retry==0.9.2

unittest-xml-reporting==2.2.0

psycopg2-binary==2.7.4

selenium==3.11.0

paramiko==2.4.2

sshtunnel==0.1.3

websocket-client==0.53.0
```

As you can see, the format of the file is a package and the version (minimum or exact) that you are using. Typically, this file is then updated as you add packages, and is stored in a source code control system such as Git.

Now, a new developer on the project can retrieve all of the source code for the project from GitHub or whatever private Git repository you are using. They can look at the source, and install all of the required packages using the following command with the `requirements.txt` file listed above:

```
pip install –r requirements.txt
```

Assuming that everything was done correctly, and that the requirements file was kept up to date, the new developer will have a working environment in a few moments. A note, this is particularly important to do when using CI/CD (continuous integration/continuous development) environments like Jenkins or Team City, since you cannot in any way assume that the packages are installed on a build environment.

User installs vs system installs

For the majority of hobby developers, installing using pip works fine right out of the box. For professionals working in regulated environments, the IT department often locks down what can be `installed` on the system. You may need permission, for example, to install a new IDE or a new system library. When it comes to installing packages, by default they are installed at the system level. This can lead to some very frustrating experiences for corporate developers. For this reason, the Python creators realized that being able to install things locally at the user level was important.

The pip application supports installing at either the system or the local (user) level. For system packages, you will normally find them under the Python `install` directory. In Linux/Mac, this is found in `/usr/local/lib/python-<Version>/site-packages`. If you are using a virtual environment, you are already set, since things will be installed in the site-packages directory within your virtual environment directory. However, if you do not use `venv` or one of its ilks, you may need to install things locally. To do this, use the following syntax for pip:

```
pip install --user <package name>
```

This will install new packages into your user directory, rather than the system directory, which will eliminate the problem of having to go beg the IT department for permission to put new things in the system directory. It will also minimize the effort in finding new packages that have been installed in your system.

So, there you are a whirlwind tour of the Python packaging system and organizational structure. From this point on, we'll assume that you have things organized the way that you want them on your system.

Conclusion

At this point, you should feel comfortable with the notions of modules and packages. You should have learned about installing packages and importing them. You should have learned how to find out what is already installed on your system and how to install it in either the full operating system or your local environment.

Finally, you should have learned how to create your own packages, and to set it up so that Python recognizes it as an addition to the system.

Questions

1. What is immediate mode in Python and how do I use it?
2. What is the difference between a package and a module?
3. How do I import an existing system package into my module?
4. What file is necessary to define a new package?
5. What program do I use to install new packages?

Chapter 5
Object-Oriented Programming

Introduction

The idea of object-oriented programming isn't new, although it was when the original Python language was introduced in 1985. Today, we take **object-oriented programming(OOP)**, for granted, and rarely talk about what it is and what it means to us as developers. In this chapter, we will explore the world of OOP, the foundational ideas upon which it was based and how those ideas are exposed in the Python programming language.

Structure

- Classes and inheritance
- Polymorphism, the Python way
- Overloading of methods
- Overloading of operators
- Chaining of operations
- Initialization
- Document strings

Objective

By the end of this chapter, you should understand the foundational concepts of object oriented programming and how they are implemented in the Python programming language. You'll get an idea of classes, along with methods and operators, and the lifecycle of an object in Python. You'll get a peek at how to do class inheritance, how to document your classes, and how to set up rational defaults within your code.

Object-oriented programming with Python

To a certain age of programming, object-oriented programming was all the rage. It was all about object this and method that. Object oriented programming, often shortened to OOP, is all about four major pieces of design and development, which are usually referred to as the pillars of OOP. These pillars consist of:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

We aren't going to go into great detail about these; you can pick up any decent book on OOP and learn them to your delight. Instead, let's focus on how Python implements the various pillars of OOP and how you can take advantage of them in your own programming exploration.

Abstraction

In OOP parlance, abstraction just means modeling of the underlying data into real world constructs. The Python class is the only real abstraction type you get in the language but is very similar to that of C++ or Java, as you have seen in the previous chapter.

Encapsulation

Encapsulation is a combination of binding data and methods together, as well as data hiding. To be fair, Python does a very poor job of the latter. You can't easily *hide* data in Python by building accessors to it within the class structure. The interpreter will allow you to mark a class data element as *internal* by prefacing it with the double underscore prefix.

Here's a sample of how you can do this:

```
class InternalTest:
```

```
   def __init__(self, x):
     self.__internal_variable = x

   def get_x(self):
      return self.__internal_variable

   def set_x(self, x):
     self.__internal_variable = x
      return self

it = InternalTest(4)
print(it.get_x())
print(it.__internal_variable)
```

If you run this little program in Python, you'll see the following output:

```
4
Traceback (most recent call last):
  File "G:/Projects/DataGenSvc/numpy/internal.py", line 15, in <module>
    print(it.__internal_variable)
AttributeError: 'InternalTest' object has no attribute '__internal_
variable'
```

This shows you that the interpreter will not allow you to retrieve that variable. This is somewhat equivalent to the C++ or Java private access declaration, but not exactly. As we will see when we talk about inheritance, derived classes do not inherit access to these private variables.

The example above, however, does illustrate some of the ways in which Python helps you to do the OOP concept of abstraction. By *binding* the data (in this case, the internal variable) together with the methods (the get and set methods of the class) we abstract away the user from worrying about how the underlying data structure is built. Python takes the idea of abstraction and trusts the end developer a bit to worry about how it is done. They don't have to ignore that they can see the data structure within the class they are using but it is the Pythonic way to do so. Trust those that went before you.

Inheritance

Inheritance is the process of creating a new class from the basics of an existing class. Like genetics, the new class *inherits* all of its innate abilities from its parent but also adds its own unique abilities and attributes.

Python classes may be slightly different from the ones you've used in other languages but the basic concepts are there. You create an instance of a class using the name of the class as if it were a function that generated an object. For our `InternalTest` class above, we created an instance of it by using:

```
it = InternalTest(4)
```

As you see, there is a constructor, alas Java or C++, called `__init__`. There is a destructor `__del__` as well, although it is not usually implemented for the majority of Python code. Memory management is all done by the Python interpreter, freeing the developer to worry more about the implementation of features and less about the underlying OS. This also makes Python more flexible and portable.

Most OOP languages have some sort of inheritance methodology. Basically, you can create a class that is *based upon* another class, which *inherits* the functionality of that class. In Python, you do this in the class declaration part of the system:

As an example, let's take our internal class above and derive something from it.

```
class DerivedInternalTest(InternalTest):


   def __init__(self):
     self.set_x(10)
```

Instantiating a `DerivedInternalTest` object automatically inherits all the functionality of the base class:

```
dit = DerivedInternalTest()
print(dit.get_x())
```

Note that we didn't write the `get_x` method for our derived class, it was picked up from the base `InternalTest` class. Besides, we didn't set the internal variable in our call to the constructor of the base class, instead that was done in the derived constructor. Now, the difference between Python and other languages is this. If we modify the derived class constructor to look like this:

```
class DerivedInternalTest(InternalTest):

   def __init__(self):
      self.__internal_variable = 10
```

it doesn't work, and the interpreter generates an error:

```
Traceback (most recent call last):
  File "internal.py", line 22, in <module>
    print(dit.get_x())
  File "internal.py", line 7, in get_x
```

```
    return self.__internal_variable
AttributeError: 'DerivedInternalTest' object has no attribute '_
InternalTest__internal_variable'
```

To Python, internal means internal always. Not just to anything derived from that class. You are stuck with it as if it were a private variable in C++ or Java. There is no notion of protected in Python. However, if you have a variable defined in the base class, you can use it in the derived class. Let's modify our system to show this:

```
class DerivedInternalTest(InternalTest):


   def __init__(self):
     self.set_x(10)
   self.a_variable = 20


class DerivedInternalTest2(DerivedInternalTest):


   def __init__(self):
     DerivedInternalTest.__init__(self)
   self.b_variable = self.a_variable * 2


   def printme(self):
     print(self.b_variable)
dit2 = DerivedInternalTest2()
dit2.printme()
40
```

Let's check this out a little bit at a time. First of all, we have the definition of the base class before this, called `InternalClass`. This is our class with an internal attribute. Next, we derive a class from this, called `DerivedInternalClass`. This class adds a new variable and sets both it and the internal variable (via the accessor) to values. Finally, we have the `DerivedInternalTest2` class. This is really where all the action is in this example. Deriving is done by enclosing the name of the class from which we are deriving in the parentheses after the class name. This looks like the following line:

```
class DerivedInternalTest2(DerivedInternalTest):
```

Within the constructor (`__init__` method), we want to be able to multiply the *inherited* attribute `a_variable` by two and set our own attribute equal to that result.

If you simply try to refer to the `a_variable` within the class, you'll find it doesn't work. Let's see what is going on here.

Modify the code as follows, just for the moment, so we can see what happens:

```
class DerivedInternalTest2(DerivedInternalTest):

   def __init__(self):
     #DerivedInternalTest.__init__(self)
   self.b_variable = self.a_variable * 2

   def printme(self):
     print(self.b_variable)

dit2 = DerivedInternalTest2()
dit2.printme()
```

The output looks like this:

```
File "internal.py", line 34, in <module>
    dit2 = DerivedInternalTest2()
File "internal.py", line 23, in __init__
    self.b_variable = self.a_variable * 2
AttributeError: 'DerivedInternalTest2' object has no attribute 'a_
variable'
```

Why do you suppose that is? If you've been working in Java, C++, or C# this would appear to make no sense. After all, you've inherited from the class that contains the proper attribute definition for `a_variable`. You instantiated an object of the right class, so why didn't it work?

The answer lies in the simplicity of Python. When you call a Python class method that creates any change to the object state, such as adding a new attribute in the `__init__` method, you are modifying the dictionary that holds all the methods and attributes for that object. In other languages that you are most likely accustomed to, the constructor for the base class is automatically called when you call the constructor for the derived class. This isn't true in Python.

By adding this single line in the constructor for our derived class `DerivedInternalTest2`:

```
DerivedInternalTest.__init__(self)
```

we fix this error shown above about a missing attribute. Why? Let's look at what happens here. In the constructor for the `DerivedInternalTest` class, we accept one

argument, the selfargument. As mentioned earlier, this is the actual object pointer that is used for this variable. The constructor for DerivedInternalTest then sets the base class for itself (`InternalTest`) private attribute `x`, to a value, and it adds its own variable to the dictionary for the object. By passing our own `self` to this method, we allow it to add the variable to our own dictionary, and thus it is available in the derived subclass:

Now when we uncomment that line and run the code, we get

```
40
```

This output is correct since we set `a_variable` to `20` in the `DerivedInternalTest` class, and then we retrieved that value and multiplied it by two to get `40` in the `DerivedInternalTest2` class object. Unlike most object oriented programming languages, Python is amazingly straightforward in how it works. If you are interested, C++ works in a very similar fashion by creating blocks of data and function pointers that represent each level of inheritance.

To get an idea of what the inheritance looks like in Python, please look at *Figure 5.1:*

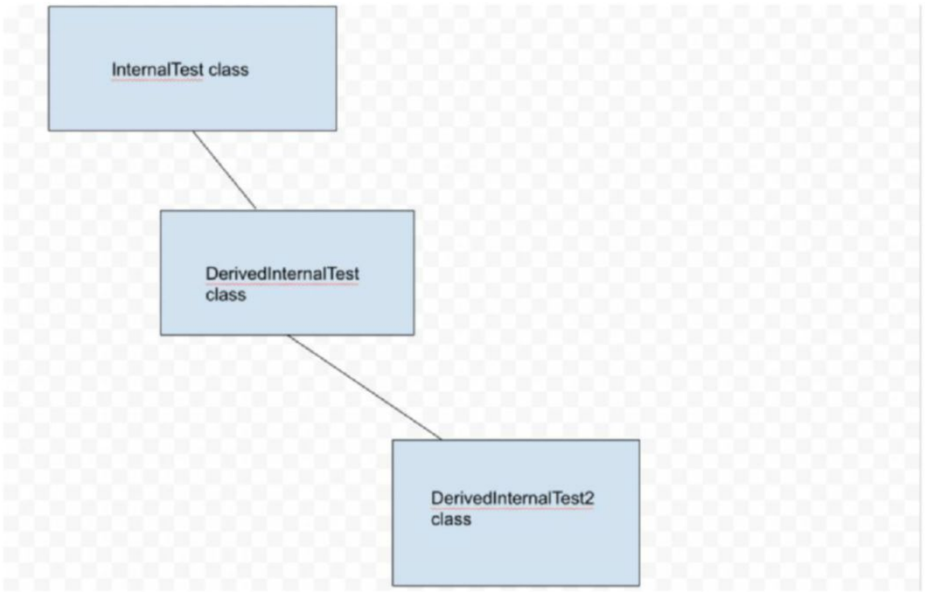

Figure 5.1: *Inheritance tree in Python*

Multiple inheritance

There is something of a debate in the programming world about the concept of multiple inheritance. Some languages, such as C# have decided that it is a terrible idea because it leads to all sorts of unexpected consequences that can cause some grief for the programmer. The Python answer is if you want to do it and you believe you know what you are doing, knock yourself out.

The basic concept of multiple inheritance is that you have two or more classes from which a subclass inherits. Here's a simple example:

```
class A:
   def __init__(self):
     self.x = 10

   def multiply_me(self, value):
     return value * self.x

   def set_x(self, value):
     self.x = value
   return self

   def get_x(self):
     return self.x

class B:
   def __init__(self):
     self.y = 20

   def multiply_me(self, value):
     return value * self.y

   def set_y(self, value):
     self.y = value
   return self

   def get_y(self):
     return self.y

class MyClass(A, B):

   def __init__(self):
     A.__init__(self)
   B.__init__(self)

   def do_a_a_thing(self, v):
```

```
    return self.multiply_me(v)

def do_a_b_thing(self, v):
  return self.multiply_me(v)

mc = MyClass()
print(mc.do_a_a_thing(5))
print(mc.do_a_b_thing(6))
```

The above program will give the following output:

```
50
60
```

The diagram for this class structure is shown in *Figure 5.2*:

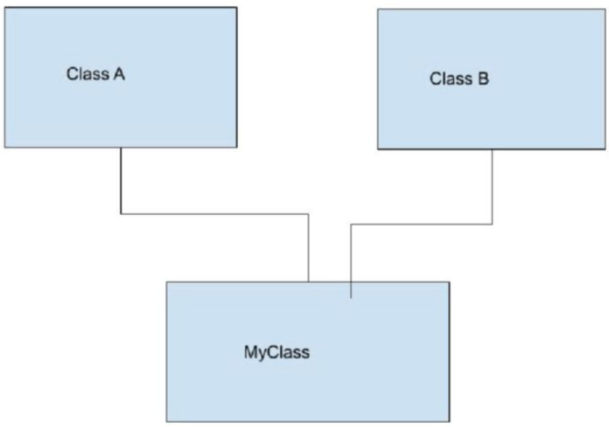

Figure 5.2: *Class diagram for multiple inheritance.*

In this example, we have two base classes, `A` and `B`, and a derived class `MyClass` which derives from both `A` and `B`. A has its attribute `X`, while `B` has its attribute `Y`. What we want to do here is have two methods in `MyClass` that call the individual `A` or `B` methods of the same name, `multiply_me`. So, we create two methods and call the derived method. Does it work?

```
50

60
```

The answer is no, we get only the A method. Why is this? Python has a multiple inheritance rule called the method resolution order. It says that when a method of the same name exists in multiple places, look in the current class. If it is not found there, look in the parent classes left to right as defined in the class line. In our case, that's `A` to `B`, because the class definition says that `MyClass` is derived in that order.

So, can we call the correct methods in the derived class? Well, of course,we can. It isn't quite as clean as straightforward inheritance, but it will work. You have to tell Python specifically which one you want to call:

```
class MyClass(A, B):

   def __init__(self):
     A.__init__(self)
   B.__init__(self)

   def do_a_a_thing(self, v):
     return A.multiply_me(self, v)

def do_a_b_thing(self, v):
  return B.multiply_me(self, v)

mc = MyClass()
print(mc.do_a_a_thing(5))
print(mc.do_a_b_thing(6))
The output here is:
50
120
```

This output, as you can see, is correct. Here's a final note on inheritance in Python. Back in the `DerivedInternalInstance2` class, we had a constructor that looked like this:

```
class DerivedInternalTest2(DerivedInternalTest):

   def __init__(self):
     DerivedInternalTest.__init__(self)
   self.b_variable = self.a_variable * 2
```

Python provides a simpler method for initializing and calling methods of a single derived class, using the `super()` method. We could re-write the constructor as follows:

```
def __init__(self):
   super().__init__()
   self.b_variable = self.a_variable * 2
```

There's nothing different here, the `super()` method acts as if you were calling the parent class with the `self` variable as its argument, otherwise it is the same. Most prefer the `super()` syntax but others like the directness of naming the class. The advantage to `super()` is that you don't have to worry about what happens if you rename the base class or change it.

Polymorphism, the Python way

Polymorphism comes from the Greek, with poly meaning many and morph meaning change. In the object oriented world, it refers to a single class or object being able to take many forms or be used in many ways. In traditional object oriented languages such as C++, polymorphism is obtained through virtual functions and method overloading. Python, obviously, has no such concept as virtual functions, at least not directly. It does, as we will see shortly, have a form of method overloading. But that's not really polymorphism. Instead, Python implements this pillar of the methodology through **duck-typing**.

The notion of duck typing dates back quite some time. In fact, it was first mentioned in a *Marx Brothers* movie in the 1930s. *Groucho Marx* says, *If it walks like a duck and talks like a duck, it must be a duck.* Python uses a similar concept. If something has a method that you call and that method accepts the argument that you call it with, then it really doesn't matter whether it is of type `A` or `B`, as long as it works. This is a slightly different concept of polymorphism than you might be accustomed to, but you will find that it works out quite well overall. If you are accustomed to C# or Java and its idea of interfaces, you will probably see how Python does what it does. The mechanism isn't the same, but the functional equivalency is there.

Let's take a look at how ducktyping works. The odds are, this example will confuse you at first, but we'll explain it after you look through the code:

```
class Vehicle:
   def __init__(self):
     pass

def move(self):
  print("Basic vehicle move")


class Car(Vehicle):
def __init__(self):
  pass
```

```
def move(self):
  print("Car moving!")

class Motorcycle(Vehicle):

def __init__(self):
  pass

def move(self):
  print("Motorcycle go vroom vroom")

class Person:

def __init__(self):
    pass

def move(self):
      print("people walk")

def func_move_it(o):
   o.move()

car = Car()
motorcycle=Motorcycle()
person=Person()

func_move_it(car)
func_move_it(motorcycle)
func_move_it(person)
func_move_it(Vehicle())
```

The output from this little program is not unexpected:

```
Car moving!
Motorcycle go vroom vroom
people walk
Basic vehicle move
```

The first thing you should notice is that while the `Car` and `Motorcycle` classes derive from the base `Vehicle` class, the Person class does not. Yet, each one of the classes contains a move method. When we invoke the `func_move_it` function, it expects to be passed some sort of thing that contains a move method. It doesn't care that it might be derived from the `Vehicle` class. This is what duck typing is all about. We can overload the methods in derived classes to accomplish the task, or we can simply implement a pseudo-interface that contains the methods we want to invoke on the object.

What if we wanted to only call the move method for objects that are derived from the `Vehicle` class? In this case, Python provides a way to accomplish this task as well. We can re-write the `func_move_it` function like this:

```
def func_move_it(o):
   if isinstance(o, Vehicle):
     o.move()
else:
  print("You didn't give me a vehicle!")

car = Car()
motorcycle = Motorcycle()
person = Person()

func_move_it(car)
func_move_it(motorcycle)
func_move_it(person)
func_move_it(Vehicle())
```

The output here is:

```
Car moving!
Motorcycle go vroom vroom
You didn't give me a vehicle!
Basic vehicle move
```

As you can see, the isinstance function will determine whether or not a given object is of the type or a type derived from the type that is passed to it. If we are looking at anything that is somehow derived from a `Vehicle`, the move method is called.

Another thing to note is the use of the pass statement in the constructors for the base and derived classes. We want to have a constructor so that you can create an instance of the class, but we don't need to do anything in that constructor.

Finally, note that overloading in Python works exactly the way you would expect it to. For a derived class that implements a method of the same name as the base class, the derived class method is called, even if you don't know what the type of the object is.

What happens if you don't implement the method move in your derived class?

```
class DeadVehicle(Vehicle):

   def __init__(self):
     pass
func_move_it(DeadVehicle())
```

The output from this function call is, not surprisingly:

```
Basic vehicle move
```

because the base move method is called for the DeadVehicle class.

One final thing to note here, you see in the calls to the method `func_move_it` that we can either pass an object of the type we want, or simply invoke the constructor for the class (that is,`Vehicle()` in the example) to create an object on the fly and then pass it to the method. If we do this and modify the object within the function, what happens?

The answer is nothing. The object that is passed to the function is modified and returned but since it isn't used again, it simply goes out of scope and disappears. The memory allocated by that object is reclaimed by the garbage processor in Python and it is as if it never existed.

Overloading of methods

In the above example, we showed how you can overload a method from a base class to a derived class. In languages like C++, C#, and Java, you can create multiple methods with the same name and different signatures. For example, in C#, you might do something like this:

```
class Foo {
  string name;
  Foo() {
  }
  void initialize() {
    name = "";
  }
```

```
  void initialize(Foo& aCopy)
  {
name = aCopy.name;
  }
  void initialize(n)
  {
    name = n;
  }
}
```

In this case, we have three different versions of the initialize method that take three different argument sets or signatures. Python does not support this sort of construct. If you were to implement the same thing using valid Python syntax, you'd find that it does not complain in the interpreter, but that only the final version of the method is stored within the class.

Instead, Python allows you to implement overloaded methods by default arguments. For example, let's imagine that we wanted to implement the same sort of `initialize` method that is shown in the above C# code. We might do something like this:

```
class Foo:

   def __init__(self):
     self.name = ""

   def initialize(self, aCopy=None, name=None):
     if aCopy != None:
      self.name = aCopy.name

if name != None:
 self.name = name

return self

   def get_name(self):
     return self.name
```

```
f1 = Foo()
f1.initialize('matt')
f2 = Foo()
f2.initialize(f1)
f3 = Foo()
f3.initialize()

print("F1 name = {0}".format(f1.get_name()))
print("F2 name = {0}".format(f2.get_name()))
print("F3 name = {0}".format(f3.get_name()))
```

You would think that this would work, wouldn't you? It would check if the `aCopy` is there, if not, it'll check if the name was there, and if not, it would simply use its default value. This doesn't work, as you can see when you run the code:

```
Traceback (most recent call last):
  File "overload.py", line 20, in <module>
    f1.initialize('matt')
  File "overload.py", line 9, in initialize
    self.name = aCopy.name
AttributeError: 'str' object has no attribute 'name'
```

Why is this? Remember that Python has no notion of specifying the type of an argument that is passed into a method or function. All types are equal to Python, so it simply assumes that the order of the objects matches up to the order of the arguments in the method. We can fix this in this way by being specific about what our arguments mean:

```
f1 = Foo()
f1.initialize(name='matt')
f2 = Foo()
f2.initialize(aCopy=f1)
f3 = Foo()
f3.initialize()

print("F1 name = {0}".format(f1.get_name()))
print("F2 name = {0}".format(f2.get_name()))
```

```
print("F3 name = {0}".format(f3.get_name()))
```

You might think that we could simply check the type of the values passed in:

```
def initialize(self, aCopy=None, name=None):
   if aCopy != None and isinstance(aCopy, Foo):
     self.name = aCopy.name


if name != None:
  self.name = name


   return self
```

This doesn't work and the reason isn't shocking at all. Remember that Python is passing arguments in the order they are expected. So when we pass in a string to the function, it doesn't just skip over the Foo object argument, it becomes the Foo object argument. We are doing the equivalent of this:

```
f1 = Foo()
f1.initialize(matt', None)
f2 = Foo()
f2.initialize(aCopy, None)
f3 = Foo()
f3.initialize(None, None)
```

You should get accustomed to passing arguments by keyword (often called the `kwarg` or keyword argument) in Python, as this is the Pythonic way of doing things.

Overloading of operators

If you are used to programming in languages like C++ or C#, you have undoubtedly tried your hands at overloading operators. The thrill of being able to write code like:

```
Foo f1 = new Foo()
Foo f2 = new Foo()
Foo f3 = f1 + f2
```

is somewhat overwhelming to the new developer to do operator overloading. There are certainly valid cases for operator overloading. In the case of writing your own mathematical functionality, such as matrices or complex variable types, it is almost expected that you be able to use the standard math operators for the class. In C#,

for example, you can overload any operator except for the dot operator (.) which is reserved by the system. Programmers do often go a bit overboard writing overloaded operators but it is something the professional expects to be able to do.

Python supports overloading operators as well. It might not seem as natural as some of the other languages but it is most certainly there. One of the most common things to do in Python is to overload the string representation operator, `str()`. It is often nice to be able to print out your complex class as a string, for reporting or logging or just to provide information to the end user. It is also quite common to use the `str()` function for debugging.

In Python, you overload the string representation by implementing the `__str__` method in your class. Let's take a look at a very simple example:

```
class MyClass:

   def __init__(self):
     self.x = 1
   selfy = 2.5
 self.z = "This is a test"

 def set_x(self, x):
   self.x = x
 return self

   def set_y(self, y):
     self.y = y
   return self

   def set_z(self, z):
     self.z = z
   return self

   def __str__(self):
     s = "X = {0}, Y = {1}, Z = '{2}'".format(self.x, self.y, self.z)
   return

mc = MyClass()
```

```
mc.set_x(45)
print(mc)
```

If you run this little program, you'll see an output that looks like this:

```
X = 45, Y = 2.5, Z = 'This is a test'
```

This is because the print statement automatically invokes the `str()` for any object that is passed to it. What if we didn't have the `__str__` method overloaded? You'd see this:

```
<__main__.MyClass instance at 0x101834488>
```

This is not very user friendly, which is why we implement the string overload. It is important to note that the goal of `__str__` is to be readable for a human. There are no restrictions placed on the method, and you can print out pretty much anything you want. Python does have a function called `repr()` which will do the same thing in most cases. The `repr()` function calls the overloaded `__repr__` method within your class.

The `__repr__` method will be used for `str()` if you override it, but the `__str__` method will not be used for `repr()` if you override it. In a practical sense, and in keeping with the Python approach, you shouldn't be overriding `__repr__`. Do so at your own peril as too many things rely on it. If you want to change the way your code displays an object when it is printed, use the `__str__` override instead.

Suppose that you had a class like this:

```
class Receipt:

   def __init__(self, t, p):
     self.title = t
   selfprice = p

   def get_title(self):
      return self.title

   def get_price(self):
     return self.price


def set_title(self, t):
  self.title = t
return self
```

```
    def set_price(self, p):
      selfprice = p
    return self


r1 = Receipt('Candy', 1.45)
r2 = Receipt('Dinner', 45.67)
r3 = Receipt('Breakfast', 12.12)
```

Clearly, we are trying to model some sort of expense receipt here, with a title and an amount. It would be nice to be able to sum up the total expenses easily. Python has a wonderful `sum()` function that accepts a list of values and returns the total or sum of those values:

```
print(sum([1.0, 2.0, 3.0]))
```

This prints out the value `6.0`, which is the sum of the three floating point numbers. Wouldn't it be nice to be able to write the following line?

```
print(sum([r1, r2, r3]))
```

Sadly, if we try to do this, we get an error:

```
  File "overload_operator.py", line 55, in <module>
    print(sum([r1, r2, r3]))
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

Here's where things get a little strange when working with Python. Python has a `__add__` operator, which adds two values together. You can implement it for the Receipt class by adding this to your class:

```
def __add__(self, other):
   return self.price + other.price
```

That should fix our problem, right? Adding this to the class and re-running the code results in this:

```
Traceback (most recent call last):
  File "overload_operator.py", line 58, in <module>
    print(sum([r1, r2, r3]))
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

Wait that makes no sense! We added the add operator, right? Take another look at the error, though. It is saying that it couldn't add an int and an object instance. Why is this? Well, the sum function initializes its result to zero and then adds each element

in the iterable to that value. We aren't adding two Receiptitems; we are adding zero to a `Receipt` item! No wonder it doesn't work. But we can tell Python how to do this by taking advantage of something called the commutative property of addition. In `math`, we know that *1+2* is the same as *2+1*. In Python, the `sum` function will first attempt to add the two values using the `__add__` method of the first argument, in this case, an integer. However, if that fails, it will attempt to add the two values calling *reverse add* or `__radd__` of the second value passing in the first argument as the value to add. We can implement this in our Receipt class:

```
def __radd__(self, other):
   return self.price + other
```

Once we do that, we can view the output properly:

```
print(sum([r1, r2, r3]))
```

```
59.24
```

To see what is being called what, let's modify the two operators to tell us which one is being called when:

```
def __add__(self, other):
   print("Add called")
  return self.price + other.price

def __radd__(self, other):
   print("rAdd called")
  return elf.price + other
```

Now, we'll try them both, using both the `sum()` function and standard addition:

```
rAdd called
rAdd called
rAdd called
59.24
Add called
rAdd called
59.24
```

Note that for our total, we call both add and `rAdd`. Why? Because the result of the first `rAdd` is a number and that gets added to the object, so it is just like calling the `sum()` function.

Comparisons and overloading

Most, if not all, functions and operations in a modern programming language rely on being able to compare two values. Sorting, searching, and like operations all need to be able to compare two values in order to properly work. Naturally, we can overload the comparison operators so that we can use the standard functions.

Let's see how this works. Add the following code to our `Receipt` class:

```
def __str__(self):
   s = "Title: {0} Price: {1}".format(self.title, self.price)
  return s

def __lt__(self, other):
   return self.price < other

def ___le__(self, other):
   return self.price <= other

def __eq__(self, other):
   return self.price == other

def __ne__(self, other):
   return self.price != other

def __gt__(self, other):
   return self.price > other

def __ge__(self, other):
   return self.price >= other
```

We can test this easily enough:

```
if r1 > r2:
    print("Receipt 1 ({0}) is greater than Receipt 2 ({1})".format(r1, r2))
else:
     print("Receipt 1 ({0}) is less than or equal to Receipt 2 ({1})".
format(str(r1), str(r2)))
```

The output from this call is:

```
Receipt 1 (Title: Candy Price: 1.45) is less than or equal to Receipt 2
(Title: Dinner Price: 45.67)
```

Since we are comparing the price element of the class to the input value, even this will work:

```
if r1 > 12.0:
   print("Greater than 12!")
else:
   print("Less than 12")
```

This might seem to work:

```
if 12.0 < r2:
   print("Greater than 12")
```

The fact is, however, it does not. What we are comparing here is the memory address of `r2` to the floating point value `12.0`. How can we fix this? Well, remember that we can test what sort of value a given input is. So, for example, we can modify the `__le__` overloaded method to read:

```
def __le__(self, other):
   if isinstance(other, Receipt):
     return self.price <= other.price
elif isinstance(other, (int, float)):
  return self.price <= other
else:
  return None
```

Here, you will also notice a nice feature of the `isinstance()` function. If you pass it a list of possible values, it will test each one of them. So we can compare our price to either an integer or a floating point value. Now, when we do this:

```
if 12.0 < r1:
   print("Greater than 12")
else:
   print("Less than 12")


if 12.0 < r2:
   print("Greater than 12")
```

```
else:
   print("Less than 12")
```

the output from this little code snippet will be as expected, as we see in *Figure 5.3:*

```
Less than 12
Greater than 12
```

Figure 5.3: *Output from code run*

In addition, of course, there is a method to override to return True or False when comparing the object to a Boolean value. This one is called `__bool__`.

Read-only attributes

One of the very nicest features in modern programming languages is the ability to have attributes that are readable, writeable or both. In most languages, there is a `get()` and `set()` function for each attribute or property that allows you to control, for example, the validation of data. You might want to allow a value only to be within a certain range, or perhaps you don't want to allow them to set the value at all. You might have an internal representation of a state that you don't want to be modified from outside the class. Python does allow such things but it isn't quite as clear as you might think or want.

Deep in the bowels of the Python interpreter, the setting and getting of attribute values within a class is done by the `setattr` and `getattr` functions. For class objects, these functions call an internal class method called `__setattr__` and `__getattr__` which is probably intuitive, given the rest of the internal methods of the classes. We can overload these methods, if we like, to modify the behavior of the class. It is slightly dangerous to do this since you are playing with the internals of the system. The `__getattr__` method, particularly, is expected to return certain values, and you can't simply return nothing at all. However, for setting a value, we can do a lot of things. For example, we can make a value read-only. Let's see how this works. First, let's just create a normal class:

```
class ReadOnlyAttributes:

   def __init__(self):
     self.write_or_read = 1.0
   self.read_only = 2.0


   def printme(self):
     print"Write or Read: {0}".format(self.write_or_read))
   print("Read Only: {0}".format(self.read_only))
```

Now, we can call this class the usual way:

```
roa = ReadOnlyAttributes()
roa.read_only = 5
roa.printme()
Write or Read: 1.0
Read Only: 5
```

Notice that the value has been modified in the class when we change the attribute and print it out. Now, let's make that impossible. Add the following code to your class:

```
def __setattr__(self, attr, value):
   print(f'Setting {attr}...')
  if attr != 'read_only' or value == 2.0:
    super().__setattr__(attr, value)
```

Why do we need to check for the value? Well, if you don't, the `set_attribute` method won't ever set the value, and the attribute will never be set in the object. Try it out. Do this:

```
def __setattr__(self, attr, value):
   print(f'Setting {attr}...')
  if attr != 'read_only':
    super().__setattr__(attr, value)
```

Running the same code will result in:

```
  File "overload_operator.py", line 132, in <module>
    roa.printme()
  File "overload_operator.py", line 123, in printme
    print("Read Only: {0}".format(self.read_only))
AttributeError: 'ReadOnlyAttributes' object has no attribute 'read_only'
```

This is due to the fact that `setattr` is called from within our code when we do the assignment in the `__init__` method. But suppose we wanted to validate data instead of just making it read only. We could easily do that with the `__setattr__` method of our class:

```
def __setattr__(self, attr, value):
   print(f'Setting {attr}...')
  if attr != 'read_only':
```

```
    super().__setattr__(attr, value)
else:
  if value > 1.0 and value < 10.0:
    super().__setattr__(attr, value)
```

Now, try setting the value:

```
roa = ReadOnlyAttributes()
roa.read_only = 5
roa.printme()
roa.read_only = 12
roa.printme()
Setting read_only...
Write or Read: 1.0
Read Only: 5
Setting read_only...
Write or Read: 1.0
Read Only: 5
```

You will notice that our `validation` method worked fine and wouldn't allow you to assign a value outside the allowed range of `1` to `10` non-inclusive. You could even write a more generic validation function to use in corporate code:

```
def _validate(self, attr, value):
   if attr != 'read_only':
     return True
if value > 1.0 and value < 10.0:
  return True
return False


def __setattr__(self, attr, value):
print(f'Setting {attr}...')
if self._validate(attr, value):
  super().__setattr__(attr, value)
```

You could extend this easily by creating a list of `validation` objects which looked at the name of the attribute and the allowed range, and use this generically in your own

code. Python allows for amazing customization without relying on the syntactical sugar that most languages use. This will work for most normal access. You can still bypass the setattr method by understanding how things are stored in a Python class. For example, consider this:

```
roa = ReadOnlyAttributes()
roa.read_only = 5
roa.printme()
roa.read_only = 12
roa.printme()
roa.__dict__['read_only'] = 12
roa.printme()
Setting read_only...
Write or Read: 1.0
Read Only: 5
Write or Read: 1.0
Read Only: 12
```

Values in a Python object are stored in an internal dictionary, just like everything else. So if you bypass the class mechanism and simply assign values to the dictionary, you will find that your validation code doesn't work. Fortunately, doing things like this is not only non-Pythonic, it is also a very bad idea. If the internals ever change, your code would break badly, and nobody would have any sympathy for you.

The __new__operator

Do you wonder how Python creates new instances of a class? The interpreter does all the work of allocating a block of memory, and the `__init__` function does the job of initializing the data for a new object. But what is responsible for making sure it all works together? The answer falls in the `__new__` class operator. This operator is called by the interpreter for a given class, which then determines whether or not to create a brand new object, initialize it, and return it to the user. Used normally, the `__new__` operator simply returns a new allocated class which is initialized by the `__init__` method. Every object which is instantiated for a given class will have its own unique memory address and values. But what if you didn't want to do this? To see what is going on, let's start with a simple class:

```
class SingletonClass(object):
   def __init__(self):
     print("Init called")
```

```
e1 = SingletonClass()
e2 = SingletonClass()
print(e1)
print(e2)
```

The output here is:

```
Init called
Init called
<__main__.SingletonClass object at 0x7fa59001aa58>
<__main__.SingletonClass object at 0x7fa59001aa90>
```

The actual values will change for your system, of course, but you will notice that the two addresses are different. What if we wanted our class to have a single object? This might be used for something like interfacing to a given resource in the system. You only want one accessor at a time, because otherwise, you'd end up with potential resource locking issues. You could do this to solve this problem:

```
class SingletonClass(object):
   def __init__(self):
     print("Init called")

def __new__(cls,*args, **kwargs):
  if not hasattr(cls,'_instance'):
    cls._instance = super(SingletonClass, cls).__new__(cls)
return cls._instance

e1 = SingletonClass()
e2 = SingletonClass()
print(e1)
print(e2)
```

Now, if you run the little application, you'll see a very different output:

```
Init called
Init called
<__main__.SingletonClass object at 0x7f9f0009db38>
<__main__.SingletonClass object at 0x7f9f0009db38>
```

Notice that in this case, the two objects are pointing to the same memory address and, in fact, are the same object. It is very rare to override `__new__` and very dangerous.

But there are times when you need it, and for those times, Python provides a good answer.

Classes and Iteratables

We spent some time in the previous chapter talking about iterables. Iterables, of course, are collection-like things that you can iterate over. For example, we can have a list:

```
x= [1,2,3,4]
```

We can use the Python iterator syntax to step through the individual elements of the list:

```
for ele in x:
    print(ele)
```

You might wonder how this is implemented under the covers. In fact, the list, tuple and so forth implement a very specific set of methods (an interface, as we would say in C# or Java) to accomplish the ability to return the individual elements. There's nothing magical about it being elements as you could do virtually anything that stepped through an interable container using the same approach. In fact, let's go ahead and implement a class that is iterable and is not a collection.

In this example, we'll look at a class that returns values that are powers of 10. Yes, you could easily do this in a simple loop, but this way, you don't have to track where you are on the list. All you need to do is call the iterable interface and you'll get back a new one. To implement an iterable, however, you need a stopping point, or you are basically creating an infinite loop. We'll address that too.

The two methods that are of concern to us in this example are the `__iter__` method and the `__next__` method.The former returns a pointer to something that can be used in the iterator you choose, such as a `for` loop. The latter returns each step in the iteration. Here's the whole class, take a look at it and then we can talk about how to use it and how it works.

```
class PowerOfTen:
   def __init__(self, mi = 0):
     self.maximum_iterators = mi

def __iter__(self):
  self.counter = 0
return self

   def __next__(self):
```

```
    if self.counter <= self.maximum_iterators:
   result = 10 ** self.counter
self.counter += 1
return result
else:
  raise StopIteration
```

You would use this the same way that you'd use any iterator.Here's one way of using a simple `for` loop:

```
pt = PowerOfTen(5)
for power in pt:
   print(power)
```

The process is as follows. First, you initialize an iterator. In our case, creating the object itself will provide an iterable thing. Then, the for loop calls the `__iter__` method to retrieve an interface to the actual iterable portion of the object. We just have our object itself, but you could return an iterator in a class that iterated over something else as well. The `for...`in syntax calls the `__next__` method of the iterable until an exception (StopIteration) is raised by the code. That terminates the loop and ends the printing.

Note that the object itself is responsible for maintaining its state. Each time we call the `__next__` method, the object will check if the internal index has reached the maximum allowable number of elements. If so, it raises the exception and the loop terminates.

You might wonder how the for loop is implemented. Let's look at what happens by writing some very trivial `for-like` code:

```
it = iter(PowerOfTen(10000))
print(next(it))
```

The iter function will return an iterator based on the `input` object. It calls the `__iter__` method of the object. If the object does not implement `__iter__`, an error would be generated:

```
class NonIterable:

   def __init__(self):
     pass

def __next__(self):
  return 0
```

```
it2 = iter(NonIterable)
print(next(it2))
  File "overload_operator.py", line 201, in <module>
    it2 = iter(NonIterable)
TypeError: 'type' object is not iterable
```

Once the iterable has validly been returned, the `next()` function will return the next element in the iterable each time it is called. If no exception is raised, calling the `next()` function will continually result in an infinite loop, much like while (`True`).

The real advantage of using an iterable is that we don't have to save all that data in the memory. We can simply generate it on demand and return it to the calling program. You can use this same methodology to generate random numbers or Fibonacci series numbers, prime values, or pretty much anything else you like.

Chaining of operations

One thing that we have used but not really mentioned is the notion of chaining operations. This isn't so much a Python feature as it is a design consideration when writing classes or functions in Python. In general, a chaining operation looks something like this:

```
result = SomeObj().someMethod1().someMethod2().someMethod3()
```

This syntax may seem confusing at first, but it is actually very straightforward and often makes like much easier for the end user of your code. In many cases, all that the developer using your code really wants to do is to make a series of calls. They don't particularly need to know the internal state of your objects or what each little piece does, they simply understand the flow they want to implement. Implementing chaining requires that you return the proper things from your methods.

First of all, why do we do this? The answer lies in simplicity and transparency. By allowing the chaining of method calls, we show the developer the flow from one area to another and allow them to specify in a single place what they want to happen. In addition, we identify, in one place, what can possibly go wrong.

What are the issues with doing this? There aren't a lot of problems with chaining of operations. The biggest one is that if something goes wrong somewhere in the middle of the flow, the process will be aborted and it may be unclear what got done and what did not. Besides, chaining of operations is mostly used for setting information in an object.Returning data from an object normally does not permit the thing to be chained, unless what is returned can be modified.

Let's look at how you chain operations in Python.

In a normal class, you might have accessor methods for your attributes that look something like this:

```
class House:

   def __init__(self):
     self.street = ""
        self.city = ""
        self.number = -1
        self.state = ""
        self.zip_code = ""

 def set_street(self, street_name):
        self.street = street_name

   def set_city(self, city_name):
        self.city = city_name

   def set_street_number(self, street_number):
        self.number = street_number

   def set_state(self, state_abbrev):
        self.state = state_abbrev

 def set_zip_code(self, zip_code_number):
        self.zip_code = zip_code_number
```

You would likely then create a new House object and set all the pieces like this:

```
house = House()
house.set_state(1313)
house.set_street("Mockingbird Lane")
house.set_city("Petaluma")
house.set_street("CA")
house.set_zip_code("94930")
```

Wouldn't it be nicer to be able to do it all in a single line, where it is clear what you are trying to do?

```
house = House().set_state(1313).set_street("Mockingbird Lane")
```

```
.set_city("Petaluma").set_street("CA").set_zip_code("94930")
```

Note that the backslash at the end of the line is simply a continuation so that the line doesn't get too long, it doesn't break anything up.

This is very easy to accomplish. We simply modify our class to look like this:

```
class House:

   def __init__(self):
     self.street = ""
     self.city = ""
   self.number = -1
 self.state = ""
self.zip_code = ""

   def set_street(self, street_name):
     self.street = street_name
   return self

   def set_city(self, city_name):
     self.city = city_name
   return self

   def set_street_number(self, street_number):
     self.number = street_number
   return self

   def set_state(self, state_abbrev):
     self.state = state_abbrev
   return self

   def set_zip_code(self, zip_code_number):
     self.zip_code = zip_code_number
   return self
```

You might be asking yourself, how does this work? Let's look at one little piece of it to get the idea. When you instantiate a `House` object via:

```
house = House()
```

The return from this statement is a House object. You could then call:

```
house.set_street("Mockingbird Lane")
```

Realizing that the return from the constructor (or `__init__` method) is a `House` object already, though, we can call methods on that returned object:

```
House().set_street("Mockingbird Lane")
```

Python then looks at what got returned from that accessor call. Since our set methods all return `self`, we are returning a House object, which can be called again and again. So long as the return from each of the intermediary method calls is a House object (or anything else that can be used to call methods) you can reuse it to call a method.

Now, suppose we had another kind of object involved. For example, let's abstract out the address of the house into its own class.

```
class Address:

   def __init__(self):
     self.street = ""
   self.city = ""
 self.number = -1
    self.state = ""
  self.zip_code = ""

   def set_street(self, street_name):
     self.street = street_name
   return self

   def set_city(self, city_name):
     self.city = city_name
   return self

   def set_street_number(self, street_number):
     self.number = street_number
   return self

   def set_state(self, state_abbrev):
     self.state = state_abbrev
```

```
    return self

    def set_zip_code(self, zip_code_number):
      self.zip_code = zip_code_number
    return self

class NewHouse :

    def __init__(self):
      self.address = None
    self.color = "White"

    def set_address(self, address_object):
      self.address = address_object
    return self

    def set_color(self, the_color):
      self.color = the_color
    return self
```

Thanks to method chaining, we can modify our assignment of all of the attributes very simply:

```
house = NewHouse().set_address(Address().set_state(1313).set_
street("Mockingbird Lane") \
   .set_city("Petaluma").set_street("CA").set_zip_code("94930")).set_
color("Blue")
```

What's going on here?

We can break this statement into two separate lines to make it a little easier to look at:

```
Address().set_state(1313).set_street("Mockingbird Lane") \
      .set_city("Petaluma").set_street("CA").set_zip_code("94930")).set_
color("Blue")
```

And

```
house = NewHouse().set_address(Address()
```

Hopefully, you can understand what is going on. We first create a new *anonymous* `Address` object to which we assign the pieces that we need: city, state, street number, and so forth. Then we take that address object and assign it to the `NewHouse` object as

the `Address` object for it. Finally, taking the result of the address assignment, which is our `Houseobject`, we assign a color to it. This can be a little confusing, so break down the line into smaller pieces until you get it.

This is an important part of the abstraction and encapsulation concepts of object oriented programming. By breaking down all our objects into their component pieces and making those pieces into independent classes, we take the responsibility away from the main object. A `House`, for example, does not really care about how its address is composed, nor should it be responsible for validating the information in the address. The Address class does that work. You should try to do similar things in your own code for readability and maintainability. This way, we can use the `Address` class for other things without having to copy and paste the code from our House class.

Initialization

We've spent a fair amount of time discussing initialization in a class using the `__init__` method, but there are a few issues worth bringing up at this stage. One thing that you must remember is that the only place that you should be creating attributes in a class is in the `__init__` method. You can assign values to an attribute anywhere, but the very first time the attribute name is encountered, it is created. Thus, we can create attributes this way:

```
class AttributesTest:

   my_list = [] # This is bad

  def __init__(self):
    self.my_list_2 = [] # This is good


def some_method(self):
  self.my_list_3 = [] # This is bad
```

In the example above, `my_list` is actually a class level variable. It is shared among all instances of the `AttributesTest` class. If this is what you intended, that's fine, but it is generally a good idea to document this so that some other programmer doesn't come along and think you made a mistake and move it into the `__init__` method.

The `my_list_2` attribute is set, as it should be, in the `__init__` method. This is normal, and the `my_list_2` attribute will be available to anyone in any method in the class, regardless of the order in which the methods are called. This is proper behavior.

The `my_list` attribute is set in a random method. It is an instance level attribute, as it was intended to be (you can tell by the `self` part), but it may or may not be available to a calling function depending on the order in which the calls are made. For example, suppose we had two other methods, `method_1` and `method_2`:

```
def method_2(self):
   for m in self.my_list_3:
  print(m)

def method_3(self):
self.some_method()
self.method_2()
```

If we call the `method_3` entry first, everything works exactly as we expected. If, on the other hand, we call `method_2`, it won't work. Let's try it out and see what happens:

```
at = AttributesTest()
at.method_2()
ile "attributes.py", line 21, in <module>
    at.method_2()
  File "attributes.py", line 13, in method_2
    for m in self.my_list_3:
AttributeError: 'AttributesTest' object has no attribute 'my_list_3'
```

Why does this happen? Because `my_list_3` doesn't exist yet, Python has no idea what you are talking about. It doesn't read the entire class and find all the assignments of attributes, it simply adds them as the code defines them. If we change our code to:

```
at = AttributesTest()
at.method_3()
```

everything works properly. The `my_list3` attribute is assigned, thanks to the call to some_method at the start of `method_3`. Thus, it exists when `method_2` is called to iterate through the list and no error is generated. If we move all of the assignments into the `__init__` methods, we won't ever have this problem.

This can't be stressed enough. Because Python allows you to do things, it doesn't mean you should. Unlike C#, C++, or Java, you can create things on the fly in a Python class. Don't take this as a license to allow yourself to do so in your production code.

```
Document strings
```

Python is one of the very few languages to not only take documentation seriously but to provide a built-in method for creating and using it. Python's *docstrings* are a simple way to document either a function or a class. The docstring is a simple string at the top of the class or function and is incorporated as a part of the object itself. For example, let's document our `AttributesTest` class as shown below:

```
class AttributesTest:
   """
  AttributesTest shows how to do things right, and wrong, in assigning
attributes ina class.
You should always assign attributes in the __init__ method and not
randomly createthem inthe other methods.Note in method_2, you will
get an error if you call it without calling some_method     first!
   """

<remainder of class removed for brevity>
```

Now, if you run the class, you will notice that it continues to be interpreted and run fine. The documentation doesn't appear to do anything. However, that's not the case. Try running this bit of code:

```
print(at.__doc__)
```

You should see the following display on your Python console:

```
    AttributesTest shows how to do things right, and wrong, in assigning
attributes in a class.
    You should always assign attributes in the __init__ method and not
randomly create them in
    the other methods.
   Note in method_2, you will get an error if you call it without calling
some_method first!
```

Okay, I can hear you saying, that's nice. But how does that really help anyone? Well, Python has a built-in help module. You can use it on any built-in method, function or class, as well as on any of your own classes. By default, it will spit out information that it deciphers from the code and signatures of your classes and methods. But it also will include docstrings:

```
help(AttributesTest)
Help on class AttributesTest in module __main__:

class AttributesTest(builtins.object)
```

```
|  AttributesTest shows how to do things right, and wrong, in assigning
attributes in a class.
|  You should always assign attributes in the __init__ method and not
randomly create them in
|  the other methods.
|  Note in method_2, you will get an error if you call it without calling
some_method first!
|
|  Methods defined here:
|
|  __init__(self)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  method_2(self)
|
|  method_3(self)
|
|  some_method(self)
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  my_list = []
```

That's pretty cool, isn't it? As mentioned earlier, Python takes documentation seriously, and so should you. You can also add a doc string to each method or function:

```
def __init__(self):
    """

    This is the initialization method for the AttributesTest class
    """

    self.my_list_2 = [] # This is good
```

As you can see, the documentation shows us exactly how to use the code. By using the documentation string feature, we make it easy for the next developer to use the code properly.

Conclusion

In this section, we explored the wide world of object oriented programming. We looked at how the four pillars of OOP are implemented in Python, and how you can use them to make your code more efficient and easier to read.

We explored inheritance and how you can group related functionality into inherited classes. We explored the idea of multiple inheritance, which is the ability to get functionality from multiple base classes. Finally, we looked at how to internally document your code so that the code and documentation would never get separated and become inconsistent.

Questions

1. What are the four pillars of OOP?
2. How does Python handle method overloading?
3. How do we call base class methods in Python?
4. What are docstrings and what purpose do they serve?

Chapter 6
Advanced Manipulations

Introduction

In our previous chapters, we've looked at the basic building blocks of Python, most of which can be found in virtually any object oriented language. Now it is time to look at some of the things that make Python unique, or at least more unique. It is probably surprising to you, if you've worked in other languages like Java or C# that many of the *new* constructs in those languages came from Python. The notion of generators, lambdas, functions as first class members were all in Python long before the other *modern* programming languages had them. It often seems as if the standards committees for some of these languages look to Python for next steps. Maybe that's the way it should be.

Structure

- List comprehensions
- Building a string from a list
- Searching a list
- Using a set of function names to create an extensible state machine
- Set comprehensions
- Dictionary comprehensions

- Generators
- Filtering vs removing
- Slicing (for example, reversing a string in one line)
- Modifying values in a list (copying vs creating new lists)
- Lambda expressions
- The splat operator and unpacking

Objectives

By the end of this chapter, you should understand how to work with lists, sets, and dictionaries in order to write complex Python programs. You will learn the difference between iterators and enumerators, and how to use lambda functions to customize your own functions.

List comprehensions

In this chapter, we'll look at a number of things, but primarily we'll be focusing on the various comprehensions in Python. A comprehension seems confusing until you get it, then you wonder how you lived without it. While there is a lot to learn in comprehensions, the basic gist of it is simple. Comprehensions are constructs that allow you to create a sequence from a given sequence and a set of rules.

Comprehensions are made up of a few basic pieces:

- An input sequence.
- An output sequence.
- An optional predicate that modifies individual components of the input list to produce the output list.
- An output expression that provides members of the output list.

Before we discuss this in depth, let's look at a valid use case for a comprehension, and then the pieces that get us there:

Let's imagine that you want to create a list consisting of five elements, all integers. You could write a simple function, like this:

```
def create_a_list(n):
 ret_list = []
 for i in range(0, n):
ret_list.append(i)
return ret_list
```

This function simply creates a list of integers, given an input range. As a comprehension, however, we can reduce the number of lines and the complexity of the code by writing this:

```
number_of_values = 5

[n for n in range(0,number_of_values)])
```

Suppose we then print out the results of both the function, calling it with five elements, and the comprehension, also using five elements. The result is the same:

```
[0, 1, 2, 3, 4]

[0, 1, 2, 3, 4]
```

So, what is going on in this comprehension? The basic form looks like this for lists: a_list = [element for element in range(range)]

Where:

- **`a_list`** is the returned list that is generated by the comprehension.
- **`element`** is an index into some sort of predicate that produces a sequence. In this case, a for loop.
- **`range`** is simply the way in which we create the elements for our output list.

Admittedly, this is a simple example and doesn't seem terribly useful. Let's *spice it up* just a little bit. Suppose that we want to generate the squares of a range of values. We could write:

```
for I in range(0,5):
    print(I*I)
```

This would produce a list of squares. If we want to add them to a list, we'd have to change our loop a little:

```
squares = []

for I in range(0,5):
    squares.append(I*I)
```

Using a list comprehension, however, we can do this much more easily:

```
squares = [n*n for n in range(0,5)]

print(squares)
```

`[0, 1, 4, 9, 16]`

Now, this is all well and good, but really it just replaces a loop. We can do so much more with list comprehensions. Let's look at a very simple extension, squaring only the odd values in a range:

```
list_of_numbers = [0,1,2,3,4,5,6,7,8,9,10]

list_of_odds = [o*o for o in list_of_numbers if o % 2 != 0]
```

```
print("odd numbers: ")

print(list_of_odds)
```

```
odd numbers: [1, 9, 25, 49, 81]
```

As you can see, we can add conditionals to our processing, still keeping it to a single processing line. More importantly, that single line can easily be optimized by the interpreter. The inside of the brackets `[]` is called the **generator expression** because without the brackets, it is really a Python generator. The brackets mean that Python will take whatever is in the generator and generate the results as a list. As we'll see in a little bit, we aren't restricted to lists, we can create dictionaries and sets this way too.

Suppose we wanted to create a list from a string. We've looked at the opposite of this, which is using the `join()` function of a string to turn a list into a string. Now, let us go the opposite direction:

```
string = "Lets Make A List Out Of A Sentence"

list = [c for c in string]

print(list)
```

```
['L', 'e', 't', 's', '', 'M', 'a', 'k', 'e', '', 'A', '', 'L', 'i', 's',
't', '', 'O', 'u', 't', '', 'O', 'f', '', 'A', '', 'S', 'e', 'n', 't',
'e', 'n', 'c', 'e']
```

Of course, we can do more than this. Imagine, for example, that you not only want to turn a list into a string, but make it a list of only capital letters, converting anything in the string that is lower case to upper case. Again, this is trivial with a list comprehension:

```
upper_list = [c.upper() for c in string]

print(upper_list)

print(string)
```

```
['L', 'E', 'T', 'S', '', 'M', 'A', 'K', 'E', '', 'A', '', 'L', 'I', 'S',
'T', '', 'O', 'U', 'T', '', 'O', 'F', '', 'A', '', 'S', 'E', 'N', 'T',
'E', 'N', 'C', 'E']
```

Once again, we see that we have the pieces in our list comprehension. The output expression is c.upper(). This is used to create each element in the output list. The variable for our list is the for c in string part. This defines each iteration of the loop to produce the output list. We don't have a predicate in this example, but we did in the odd numbers one above, which was of o %2 == 0. This predicate is applied to each element of the list as it is iterated over, and, in this case, is used to screen out values.

What sorts of things can we do with the predicate? Let's imagine that you have are reading in code from a file. You have a list of each of the lines of the file, and want to get rid of the comments. In Python, of course, a comment line begins with

a pound sign (#). We'll just screen out every line that starts with a comment using a list comprehension:

```
list_of_strings = [
   "# This is a comment",
  "This is a line of code",
 "This is another line of code",
  "# Another comment",
"# A Third Comment",
 "Nothing but code"
]

code_lines = [line for line in list_of_strings if line[0] != '#']
print(code_lines)
```

```
['This is a line of code', 'This is another line of code', 'Nothing but code']
```

As you can see, our output list contains only those lines that don't begin with a comment. It is also worth noting that the list comprehension does not impact the original list. If we looked at `list_of_strings`, we would see that it contained all of the original strings, including the comment lines. This is important, as we often wish to manipulate a list to produce a new list, but need to keep the original around.

Speaking of lists of strings, suppose that we wanted to take an input list of strings and reverse them? We can do this easily with a list comprehension:

```
list_of_strings2 = [
    "This is a test",
    "This is another test",
    "Test 3"
]

reversed_lines = [line[::-1] for line in list_of_strings2]
print(reversed_lines)
```

```
['tset a si sihT', 'tset rehtona si sihT', '3 tseT']
```

Of course, lists aren't restricted to a single dimension, and neither are list comprehensions. In linear algebra, for example, we have the concept of an **identity matrix**. This is a two-dimensional matrix that has zeroes in all elements except where the row is equal to the column. An identity matrix looks like this physically:

```
[
[1, 0, 0],
[0,1,0],
[0,0,1]
]
```

We can create one of these with a very simple two-dimensional list comprehension:

```
# Identity matrix
id_matrix = [ [ 1 if item_idx == row_idx else 0 for item_idx in range(0, 3) ] for row_idx in range(0, 3) ]
print(id_matrix)
```

```
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

Likewise, we can create an empty matrix via the comprehension mechanism, and it is even simpler:

```
# Empty matrix
empty_matrix = [ [0 for i in range(0, 3)] for j in range(0, 3)]
print(empty_matrix)
```

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Finally, there's no restriction on the number of list elements you can create with a list comprehension. Suppose that we want to create a list some mathematical computation of the various indices of the lists. We can do that easily:

```
print([x+y+z for x in range(1,5) for y in range(1,5) for z in range(2,6)])
```

```
[4, 5, 6, 7, 5, 6, 7, 8, 6, 7, 8, 9, 7, 8, 9, 10, 5, 6, 7, 8, 6, 7, 8, 9, 7, 8, 9, 10, 8, 9, 10, 11, 6, 7, 8, 9, 7, 8, 9, 10, 8, 9, 10, 11, 9, 10, 11, 12, 7, 8, 9, 10, 8, 9, 10, 11, 9, 10, 11, 12, 10, 11, 12, 13]
```

Notice how the output in this case is a single list with the total of three ranges of values in it. This produces a long list.

What if we wanted to do different things based on the input values in the list? List comprehensions do support the `if...else` construct as well. First, let's look at a simple example:

```
l = [1,2,3,4,5,6,7,8,9,10]
print([x if x % 2 == 0 else x+1 for x in l])
```

```
[2, 2, 4, 4, 6, 6, 8, 8, 10, 10]
```

Next, consider the **`if...elif...else`** case. You can't actually does this in a list comprehension, at least not directly. The comprehension syntax uses the ternary

form of the if statement. Basically, this means if this, then that, **else** something **else**. But the something **else** part can be another **if...else** statement as well. Consider this example, where we want to print odd or even based on the numbers, except for **0**:

```
print(['zero' if v == 0 else 'odd' if v % 2 == 0 else 'even' for v in
range(0,5)])
```

```
['zero', 'even', 'odd', 'even', 'odd']
```

Here, we first test to see if the value is zero, and if so, we print that fact. Otherwise, we then examine each element and determine if it is odd or even and print the result. You have to think inside out to write these kinds of statements. Work your way out from the initial statement, where the value is **0**. That takes in the entire first part of the statement **('zero' if v == 0)**. Next, we look at the case where the value is odd **('odd' if v % 2 == 0)**. If neither of those pieces are true, the final **else** is called. This is the equivalent of the following single lines:

```
if v == 0:
x = 'zero'
elif x % 2 == 0:
x = 'odd'
else:
x = 'even'
```

Dictionary comprehensions

The notion of comprehensions isn't restricted to lists in Pythons. Most of the higher constructs permit it. The dictionary, for example, can be created via the comprehension mechanism. In general, a dictionary comprehension allows you to modify an existing iterable in some fashion to create a dictionary of keys and values. The iterable can be a list, for example:

```
list_of_words = ['the','cat','is','on','the','roof']
dict_from_list = {k:v for k,v in enumerate(list_of_words) }
print(dict_from_list)
```

```
{0: 'the', 1: 'cat', 2: 'is', 3: 'on', 4: 'the', 5: 'roof'}
```

So, what's going on here? First of all, the syntax for a dictionary comprehension is as follows:

```
{k:v <expression generating key and value}
```

In this case, we assign the key to the index of the list via the enumerate function, and use the list value as the value for the key. This way, we have a unique identifier

for the key as an index. What if we did it the other way? Notice that we have two values of the in our list of words. Dictionaries must be unique. So, what happens if we create the dictionary with the value as the key?

```
dict_from_list_2 = {k:v for v,k in enumerate(list_of_words) }

print(dict_from_list_2)
```

You might guess that this would produce an error since you are adding two keys with the same value. However, Python is nicer (or perhaps more confusing) than that. Instead, the output is:

```
{'the': 4, 'on': 3, 'is': 2, 'roof': 5, 'cat': 1}
```

Notice two things here. First, we get the second value of 'the' rather than the first. You can tell this because the index is four, rather than zero. Secondly, notice that the keys are not sorted in the order we would expect. A dictionary does not guarantee sort order of the keys; things are stored in the order of the hash of that key. Finally, notice that we have only one **the**, rather than two. Python is kind and doesn't generate an error during the comprehension execution, it just replaces the duplicate value with the latest key-value pair.

To look at what is actually happening; realize that this is exactly the same process as we saw in the list comprehension. The key and value are generated by the generator expression, which in this case is the **for** loop over the set of tuples generated by the **enumerate** function. Basically, the **for** loop is creating a new list of tuples:

```
('the', 0)

('cat', 1)

And so forth and so on.
```

Each of these tuples is then flattened into two values, the key and value. In the first example of dictionary comprehension, the two values are assigned to the k and v variables, so that they knew key becomes the index and the new value becomes the word from the list. These are then assigned back to the new k and v variables for the dictionary comprehension, which are used to create an entry in our new dictionary.

You can do quite a bit with dictionary comprehensions. For the most part, any **for** loop can be expanded out to become a dictionary. You can also use the **if** statement within the **for** loop. This is slightly different syntax than the list comprehension:

```
dict_of_odds = {k:k for k in range(0,6) if k % 2 != 0 }

print(dict_of_odds)

{1: 1, 3: 3, 5: 5}
```

It is important to realize that the names of the variables all have to match up. That is, we use k as both the key and the value (which is fine, you can use anything you like as the key and value variables). The **k** has to come from somewhere. If, for example, we wrote this:

```
dict_of_odds = {k:k for i in range(0,6) if i % 2 != 0 }
print(dict_of_odds)
```

Then the Python interpreter would generate an error when we tried to run this code:

```
  File "dictcomp.py", line 27, in <dictcomp>
    dict_of_odds = {k:k for i in range(0,6) if i % 2 != 0 }
NameError: name 'k' is not defined
```

This is because we use the `i` variable in the loop, but used the k variable in the comprehension.

You can nest **`if`** statements as well:

```
dict_of_odds = {k:k for k in range(0,6) if k % 2 != 0 if k != 3}
print(dict_of_odds)
```

Again, this is different than the `for` loop and the list comprehension. In dictionary comprehensions, the **`if`** statements are just *stacked* one after another.

Here's one more example of using a list and a conditional to create a dictionary. In this case, we are going to imagine that we have a class that contains methods we want to **`map`** to a dictionary of commands. This particular example will have a class with methods that convert a given input value into a specified type. Here's the class:

```
class Comparer:

   def __init__(self):
     self.value = "123"

   def return_as_string(self):
     return self.value

   def return_as_int(self):
     return int(self.value)

   def return_as_float(self):
     return float(self.value)
```

Nothing fancy here, just a class with three methods that we care about (we'll never call the **`__init__`** method directly of course). How are we going to add the various methods to a dictionary? We could do something like this:

```
d = {}
c = Comparer()
d['string'] =c.return_as_string
```

And then repeat this for each method. As you may remember, Python treats methods as a first class citizens, so we can assign them to things, like dictionary entries.You may remember that we looked at a function defined in Python called **dir**. This function returns a list of all of the methods and attributes of a given class. We could use the combination of dictionary comprehension along with the **dir()** function and accomplish our task much more easily:

```
print(dir(Comparer()))
dict = {v[10:]:v for v in dir(Comparer()) if v.startswith('return_as')}
print(dict)
```

The output of this little snippet of code is:

```
{'float': 'return_as_float', 'int': 'return_as_int', 'string': 'return_as_
string'}
```

This is exactly what we wanted.

Nested dictionary comprehensions

You can nest dictionaries, by making a dictionary the value of a given key in another dictionary. For example, we might do this:

```
dict_nested = {
'production': {
'url': 'http://prod.com',
'user': 'matt',
'password': 'pwd'
 },
'qa': {
'url': 'http://qa.com',
'user': 'qamatt',
'password': 'pwd'
 },
'staging': {
'url': 'http://staging.com',
'user': 'fred',
'password': 'pwd1'
  }
}
```

In this case, we have a dictionary called **`dict_nested`**. This dictionary is made up of three keys, which represent different environments for testing. We have the **`production`** environment, the **`qa`** environment, and the **`staging`** environment. Each of these has three keys, the **`url`** of the environment, the **`user`** and **`password`** to use when connecting to that environment.

We can use the dictionary comprehension method to extract out only the dictionary we want:

```
qa = {k:v for k,v in dict_nested.items() if k == 'qa'}
print(qa)
```

```
{'qa': {'user': 'qamatt', 'password': 'pwd', 'url': 'http://qa.com'}}
```

Further, we can extract out only the piece of the inner dictionary that we want by using something like this dictionary comprehension:

```
data = {
   outer_k: {inner_k: inner_v}
  for outer_k, outer_v in dict_nested.items()
 for inner_k, inner_v in outer_v.items()
   if inner_k == 'url'
}
print(data)
```

Note that the whitespace that has been inserted is simply for readability, there is no need to put things on a separate line if you don't want to.

Applying functions

We've looked at going through dictionaries using comprehensions that return pieces of the dictionary as data in order to produce a new dictionary. We can, however, apply user defined functions to the pieces if we want to.

```
def func(x):
   return x * x


dict_func = { k: func(k) for k in range(0,5)}
print(dict_func)
```

In this example, we are taking a list of integer values in the range of zero to four and using our user-defined function to square them. The key for the generated dictionary is the number in the **`range`** (for example,**`0`** to **`4`**) while the value of each key should be the square of that number. Running this little snippet results in:

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

This is exactly what we expected. As you can see, dictionary comprehensions are a very powerful tool in Python. Of course, any dictionary comprehension can be replaced by a loop or set of loops in your code. If your comprehension gets too complicated, as the url example above, you might consider breaking it out into simple code, or even a function in your code.

Restrictions on dictionary comprehensions

- **You can't modify an existing one, only create a new one.**
  This might not be obvious, but you can't modify an existing dictionary. You can only make a new dictionary, and put values into it that are somehow derived from the values in the original one. You can add additional keys or values to the dictionary in your code, but cannot change the dictionary that you start with. Of course, you can always assign the result to the original variable that holds the dictionary, but you aren't modifying it you are resetting it. This is an important distinction in Python.
- **You can't check existing keys in the new dictionary during creation.**
  You might wonder if it is possible to check the new dictionary values during the generation of the dictionary. The answer is no, because it doesn't really exist until the time that your generator code is executed. You can check the existing dictionary; of course, since it is already in place by the time your comprehension is called, but the one that is being created cannot be modified or checked at run-time.

Set comprehensions

At this point, it can't be at all surprising to know that Python also allows for set comprehensions. As expected, you can't modify an existing set with a comprehension, only create a new one. Likewise, of course, the comprehension must result in a valid set. A set cannot contain multiple entries of the same value. Like the dictionary, Python is polite about this. If you try to add values to the set that are already there, it will replace the old one with the new one.

Set comprehensions using the **`{}`** syntax only exist in Python 3. Before that, you'll have to use the **`set()`** function to create and work with sets.

You might guess, therefore, that one of the best uses of a set is to eliminate duplicates. In fact, this is one of the most basic forms of the set comprehension. Given a list, we can duplicate it as a list with a simple list comprehension like this:

```
list_copy = [x for x in original_list]
```

It shouldn't surprise you, therefore, that if we change the list comprehension to a set comprehension, we get the same result, but as a set:

```
my_list_with_dupes = [1,2,1,2,3,4,1,2,3,4,5,6,7,1,2,3]

my_set_without_dupes = {x for x in my_list_with_dupes}

print(my_set_without_dupes)
```

```
{1, 2, 3, 4, 5, 6, 7}
```

Notice that the set notation **{}** is used just the way that list notation **[]** is used for list comprehensions. Of course, the only difference between a set and a dictionary, which use the same **{}** notation, is that dictionaries are unique keys with non-unique values assigned to them.

Naturally, you can modify the entries in the set while you are building it. For example, suppose that we take the output from the set we just created, and form a new set that is made up of the squared values of each entry:

```
squared_set = {x*x for x in my_set_without_dupes}

print(squared_set)
```

```
{1, 4, 36, 9, 16, 49, 25}
```

As with list and dictionary comprehensions, the original set **(my_set_without_dupes)** is not modified by the comprehension. For all comprehensions, the only way to modify the existing construct is to assign it to the result of the comprehension, since it creates a new object.

It is important to note that the default comparisons are used for whatever type you are placing into your new set. For example, consider this list of strings converted to a string:

```
my_string_list = ['This', 'is', 'a', 'test', 'of', 'A', 'Capital', 'TEST']

print({x for x in my_string_list})
```

The output from this snippet is what you would expect:

```
{'This', 'Capital', 'of', 'is', 'TEST', 'A', 'a', 'test'}
```

Remember, though, that a set must be made up of unique entries. So, if we were to take the string list above and add only the lower case version of each string like this:

```
print({x.lower() for x in my_string_list})
```

In this case, the output is quite different from the previous example. Each string is converted to its lower case equivalent and then added to the output set:

```
{'is', 'of', 'this', 'test', 'a', 'capital'}
```

It is often useful to convert a string to a base form when adding to sets so that you don't have issues with the same words occurring over and over, differentiated only by case. If we are counting the distinct words in a sentence, for example, we don't care if you enter **This**, **this**, and **THIS**, they are all the same word and should only be counted once.

The standard example for set manipulation is the *Sieve of Erathotenes*, which finds all prime numbers within a given range. The algorithm here is simple; you begin with the prime number **2**, and generate a set of all multiples of that prime. Then you work your way up to the value you care about adding the multiples of each number to the collection. Once you have gone through all of the values, you then simply step through list and see whether the number you care about is in there. You don't add the starting point (**2**,**3**,**4**, and more) unless that value is generated by a multiple. So, for example, if we were looking for all primes between 1 and 9, we would generate these values:

```
4, 6, 8
6, 9
8, 12, 16
10, 15, 20
12
14
16
```

We would then walk through the list looking for values that aren't there which would result in the values **2**, **3**, **5**, and **7**. To do this in Python, we follow the exact same process, but we use the set comprehension to simplify the exercise:

```
def erathostenes(maximum):
    # First, generate a list of non-primes up to the maximum we care about
    non_primes = {j for i in range(2, maximum) for j in range(i * 2, maximum, i)}

    # Now, generate a set of values in the maximum range, so long as each value
     # is not found in the non-prime list.
     return {i for i in range(2, maximum) if i not in non_primes}
```

If you've ever tried writing this algorithm in a more modern language like Java or C++, you end up writing recursive loops to speed up things, or have loops of loops. Python makes this so much easier. If you run this on our input value, **9**, we will see:

```
{2, 3, 5, 7}
```

As a side note, this also illustrates that you can use all comprehensions within functions, and pass in variables rather than hard-coded names, to process data. Set comprehensions are pretty much exactly like list or dictionary comprehensions, aside from slightly different syntax. You can use the **if** statement within a comprehension:

```
set_of_odds = {x for x in range(0,10) if x % 2 != 0}
print(set_of_odds)
```

```
{1, 3, 5, 7, 9}
```

As a final example for set comprehensions, let's combine most of what we learned in this chapter with comprehensions to do something semi-useful. Imagine that you want to take some input text and produce a list of *important words* within the sentence to attempt to comprehend the purpose of the sentence. An *important* word is defined as one that isn't in a screening list of trivial words like **a**, **and**, **the** and the like. In most languages, this would be a fairly complex bit of code, but in Python, it is easy and readable:

```
unimportant_words = ['the', 'and', 'i', 'or', 'this', 'of', 'to', 'if']

sentence = "This is a test of the emergency broadcast system which tests
to see if the world has broken down"

word_list = sentence.split('')

important_word_list = {w.lower() for w in word_list if w not in unimportant_
words}

print(important_word_list)
```

The output from this snippet is:

```
{'which', 'world', 'test', 'tests', 'is', 'a', 'system', 'emergency',
'see', 'broken', 'down', 'broadcast', 'this', 'has'}
```

Let's take a look at what is going on in this example. First, we define our **unimportant** words in a simple list of strings. Then we split up our input sentence into words, using the split function of the string class. Now comes the comprehension part, as we iterate over the words in the list, looking for anything that isn't in the unimportant list. Those words are converted to lower case and added to the set. Notice the use of the not in construct to determine if a word is **not in** the list of words to screen out. If we used the in construct, we would only be adding the unimportant words to the list.

Comprehensions are an important part of Python, and something that it is really worth getting to know, if only to read other people's code. You don't have to use them, Python will accept for loops and if statements and function that replace them, of course, but they are something that most professionals use, so it is worth learning how they work and using them. While we are on the subject, though, let's talk a little bit about generators which are very much like comprehensions, but are also a valuable tool in and of themselves.

Generators

The basic syntax of a generator expression is: (some expression). Unlike comprehensions, however, generators do not return a collection such as a list or dictionary, but rather return a generator object. A generator object is an instance of a class that is very similar to an interator, without the need to implement the **__next__**

and __iter__ methods. For this reason, the complete piece is usually referred to as a **generator expression**, to contrast it with the notion of a comprehension. Let's look at a very simple generator, which returns the odd numbers between **0**and **10**.

```
odd_generator = (x for x in range(0,10) if x % 2 != 0 )
```

Note the difference between this and the list comprehension that we looked at earlier in the chapter. When written this way:

```
odd_comprehension = [x for x in range(0,10) if x % 2 != 0 ]

print(odd_comprehension)
```

Here, we are creating a comprehension that returns a list of odd numbers between the values of **0**and **10**, not inclusive. The output from this is:

```
[1, 3, 5, 7, 9]
```

When we create a generator object, such as the **odd_generator** above, and print it out, we get something very different:

```
<generator object <genexpr> at 0x7fd5b80299a8>
```

If you run your own copy of the code, you'll see a different memory address, of course, at the end of the print line, but you'll see the same beginning, **<generator object <genexpr>** at. What does this mean? It means that we have created a new object, rather than some syntactical short-hand for a loop. We invoke the generator object using the **next()** function:

```
print(next(odd_generator))
```

```
1
```

This doesn't seem terribly exciting, all we did was get the first odd number after 0. But it becomes a bit more exciting when we call it again:

```
print(next(odd_generator))
```

```
3
```

In fact, we can call this multiple times. Suppose, for example, we just keep calling **next()** on our generator object:

```
while(True):

   print(next(odd_generator))
```

We have talked about infinite loops and how they will never exit, so you would assume that this would just continually print out odd numbers until the end of the world. However, something very different happens when you run the script:

```
Traceback (most recent call last):

<generator object <genexpr> at 0x7fc9280199a8>

  File "generator.py", line 5, in <module>
```

```
    print(next(odd_generator))
StopIteration
1
3
5
7
9
```

This seems odd, doesn't it? Why didn't the **`while(True)`** loop continue to execute, and what is that **`Traceback`** thing at the top of the output? The answer lies in the generator object. When we created it, we established a range for it to generate odd numbers for, in this case the value of **`10`**. When you call the **`next()`** function, each time it retrieves the next value in the condition we established when we wrote the generator code. In this case, we will first get the value zero, then one, then two, and so forth and so on, until we hit ten. Each of these values is checked to see if it is odd, and if so, that value is then returned. Once the value is returned, the generator object then pauses, waiting for the next **`next()`** function call. When the loop in the generator is exhausted (in other words, when the value hits **`10`**), the generator knows that it is done. Calling the **`next()`** function one more time will raise an exception, in this case the **`StopIteration`** exception.

Now, try calling the function in a slightly different kind of loop:

```
for v in odd_generator:
   print(v)
```

Now, you would think this would raise the same exception and print it out on the command line, but you would be wrong. In fact, the output is:

```
1
3
5
7
9
```

Once all of the values have been printed, the **`for`** loop terminates and no error is displayed. This is due to the fact that the**`for`** loop *knows* about the *StopIteration* exception, and uses it to end the loop. The code hasn't changed, the error hasn't changed, but the behavior has changed. This is some behind the scenes magic for Python.

Generators do not have to be simple expressions. You can actually create a full generator function. The behind the scenes magic of returning a single item at a time

is performed by the **yield** statement. Let's look at how you might use such a function in your own code. In our first example, we'll just handle a specific number of entries:

```
def an_integer_generator(max):
   for i in range(0, max):
     if i % 2 != 0:
      yield i
```

Obviously, this is simply a function that returns the odd numbers between zero and the maximum range provided by the caller. The only difference is the use of the **yield** statement. The **yield** statement, which takes an argument, returns that value to the caller and waits for the next invocation by the caller. We can call it directly:

```
gen = an_integer_generator(10)
print(next(gen))
```

```
1
```

Not surprisingly, calling it with a single **next()** function invocation produces the first odd number, which is one. If we call it again, we'll get the next in the series (**3**,**5**,**7**, and more).

Alternatively, we can call it in a loop:

```
gen = an_integer_generator(10)
for g in gen:
   print(g)
```

```
1
3
5
7
9
```

Equally unsurprisingly, we get the list of odd numbers between **0**and **10**. You might wonder what happens if we combine the two uses of the generator object:

```
gen = an_integer_generator(10)
print("First odd: {0}".format(next(gen)))
for g in gen:
   print("Odd: {0}".format(g))
```

The output from this little snippet might surprise you a little bit. It looks like this:

```
First odd: 1
```

```
Odd: 3
Odd: 5
Odd: 7
Odd: 9
```

The important thing to notice here is that the generator object maintains its own state. When we call it with the **`next()`** function, we are moving the internal state pointer from the first element to the second element. From that point on, calling **`next()`** or using the gen object in a loop will move forward from the first position onward. The generator function is a form of iterator. For example, suppose that we wanted to write a class that implements the same functionality, but as a true iterator. We might do something like this:

```
class Odds:
   def __init__(self, max = 0):
     self.max = max
def __iter__(self):
self.n = 0
   return self
   def __next__(self):
     if self.n >= self.max:
      raise StopIteration
while True:
 if self.n % 2 != 0:
 r = self.n
            self.n = self.n + 1
            return r
else:
if self.n >= self.max:
raise StopIteration
else:
self.n = self.n + 1

o = Odds(10)
for odd in o:
   print(odd)
```

```
1
3
5
7
9
```

That's a pretty unwieldy class! But this is what is necessary to implement a true iterator in Python, we have to worry about the initialization **(__iter__)** method, as well as the class initialization **(__init__)** and the next method **(__next__)**.

All of this to replace our little four line function! Hopefully, you can see the power of generator functions and generator expressions and will use them in your own code. Please note that like all other things Pythonic, it is very easy to abuse the generator expressions and functions, so use them only when they really make sense.

That sums up, excuse the pun, the comprehensions and generators in Python. You can do a lot with a little code, which really is what Python is all about. In addition, if you follow the rules properly, using the yield statement or the **__iter__ and __next__** method overloading, the interpreter will do most of the behind the scenes magic for you to make your code easy for the next developer to use. Always remember, the next developer to use your code just might be you!

Building a string from a list

One of the most common programming problems is to assemble strings. We might have a list of numbers or words or letters that we want to output. For example, imagine a non-homogeneous list like this:

```
list_of_values = [1.0, 'this is a test', 2, 'c', 'hello world']
```

We could output these to the user like this:

```
for v in list_of_values:
   print(v)
```

If we do this, the user will see the following:

```
1.0
this is a test
2
c
hello world
```

That's not really what we want. We want to be able to output them as a single string, for example, to a log file. It turns out that Python 3 has an argument to the **print** statement that will allow you to do something like this:

```
for v in list_of_values:
   print(v, end = '')
```

```
1.0 this is a test 2 c hello world
```

This doesn't really solve our problem though, since the **`print()`** statement is only good for writing to the console output. What we want is a way to convert the list of entries into a single string that can then be written to a file using Python's file input/output functions which we will look at soon. We've previously looked at the string **`join()`** function, which looks like it might work:

```
s = ''.join(list_of_values)
print(s)
```

Sadly, trying to do this produces an error:

```
  File "stringoutput.py", line 3, in <module>
    s = ''.join(list_of_values)
TypeError: sequence item 0: expected str instance, float found
```

The **`join()`** method of the string class expects an iterable that contains strings. But the **`join()`** method is still the best choice for producing an output string from a string iterable, so how can we convert our list of non-strings into strings. The answer, as you might have guessed from the question being in the comprehensions chapter, is to use a list comprehension:

```
s = ''.join([str(l) for l in list_of_values])
print(s)
```

```
1.0 this is a test 2 c hello world
```

Why do we prefer the join() method over rolling our own function to accomplish the same thing? There are a few reasons. One is that being Pythonic means following the **Don't Repeat Yourself (DRY)** principle. Some people also call it **don't reinvent the wheel**, but **DRTW** doesn't have the same ring to it. The functionality already exists within the language; there is no reason to reproduce that functionality in your own code. Other Python programmers will be used to seeing the **`join()`** method, as well as the list comprehension, and should be able to immediately grasp your usage without having to dig into your code to see how it works. In addition, of course, the Python implementation has been rigorously tested by thousands of programmers, whereas yours might have been tried by yourself and a few of your co-workers. Does it right, use the existing functionality.

Another reason for doing it this way is that we can take advantage of the hours of design that went into the join method. For example, the **`join()`** method has a wonderful property. The string that is used to join with is reproduced for each element in the iterable. So, we could do this:

```
s = '|'.join([str(l) for l in list_of_values])
print(s)
```

This produces the following output, which is absolutely perfect for writing to a character delimited log:

```
1.0|this is a test|2|c|hello world
```

Speaking of logging things, one of the problems you may run into when doing so is that log files expect *special characters*like quotation marks and slashes to be *escaped*. This means that rather than writing a string like that's to the file, you would output **that**\'sso that the log parser can split the words properly. This is very common in programming, and you've likely seen in previously when working with outputs to files in other language. Python makes it very easy to get around this in our scenario above, and all we need to do is write the small function to do the string manipulation while still doing all of the rest of the work. We can implement a screening function like this:

```
def convert_string(s):
   ret = ""
  for c in s:
    if c in '\\\'"':
     ret = ret + '\\' + c
else:
      ret = ret + c
return ret
```

In our example function above, you can simply add any other characters that you need to be escaped into the string that is checked with the **if c in** portion, and your code will continue to properly escape output strings. To use this in our join function, we just replace the call to **str** with:

```
s = '|'.join([convert_string(str(l)) for l in list_of_values])
```

Now, if we have a list of elements in which there are strings that contain characters that need to be escaped before output, it will be done for us automatically:

```
list_of_values = [1.0, 'that\'s a test of the \"emergency broadcast
system\"', 2, 'c', 'hello world']

s = '|'.join([convert_string(str(l)) for l in list_of_values])
print(s)
```

```
1.0|that\'s a test of the \"emergency broadcast system\"|2|c|hello world
```

Notice that we still have to cast the object in the list to a string to make sure we get the right outputs. This string is then passed through our **convert_string**

function, which does the special character processing and the **`result`** is output to the log, or console, or whatever. Python makes it easy to do the jobs that professional programmers are accustomed to having to do as parts of their job.

Searching a string

Without a doubt, the most common programming problem is searching a given string for one or more occurrences of a value. Python, of course, makes this reasonably easy. For example, we can tell whether something is found within another string:

```
string_to_search = "This is a test of the emergency broadcast system. This
is not an error"

# Case 1: Just find out if the string has "error" in it
print("String contains error: {0}".format('error' in string_to_search))

# Case 2: Case sensitive search
print("String contains error: {0}".format('Error' in string_to_search))

# Case 3: Doing conversion to do case insensitive search
print("String contains error: {0}".format('error' in string_to_search.
lower()))
```

These three cases illustrate the various ways one might search for a substring within a string. The first two search for a given string with a given case of letters within the input string. The third example shows how to do the same search, but make it case-insensitive. We've gone through most of this before, when discussing the string class in *Chapter Two*. It is therefore not at all surprising to you that the output of these three calls is:

```
String contains error: True
String contains error: False
String contains error: True
```

Clearly, the second example is **`False`** because the string does not contain a character run that matches the Error case:

```
# Case 4: What if there are multiple occurrences?
string_to_search = "The error is that the error is no error"
print("String contains error: {0}".format('error' in string_to_search))
print("Position: {0}".format(string_to_search.find('error')))
```

```
# Case 4: What if there are multiple occurrences?
string_to_search = "The error is that the error is no error"
print("String contains error: {0}".format('error' in string_to_search))

print("Position: {0}".format(string_to_search.find('error')))
```

It isn't unusual for a string to contain multiple copies of the same substring. If we only want to verify that one is in there somewhere, we can use the **find()** method of the string class. If it returns anything other than minus one, we know it is there:

```
Position: 4
```

Sometimes, though, we want to find more than the first occurrence of the substring within the larger string. There are lots of ways to do this, from simply calling the find method over and over and passing in a new string formed by the substring of the original incremented past the position of the substring, to using self-built string searching algorithms. What, though, if we just used the Python generator expression concept to do this in a vastly easier way?

```
def find_value( string, value ):
   pos = 0
 while pos != -1:
    p = string[pos:].find(value)
   if p == -1:
   raise StopIteration
 ret = pos + p
  pos = ret + 1
 yield ret

gen = find_value(string_to_search, "error")
```

With a generator expression function, we can just call the **next()** method on the generator object and retrieve the pieces that match our substring. One approach would be to do this:

```
p = next(gen)
print(string_to_search[p:])
p = next(gen)
print(string_to_search[p:])
p = next(gen)
print(string_to_search[p:])
```

Running code this way, we will see the following output (which is just showing the string starting at the returned position of the substring):

```
error is that the error is no error

error is no error

error
```

Once we reach the final entry in the string, the **`StopIteration`** exception is raised, so we can just use a loop to do the same thing:

```
print("In loop version:")
gen = find_value(string_to_search, "error")
for p in gen:
   print(string_to_search[p:])
```

```
In loop version:
error is that the error is no error

error is no error

error
```

As you can see, the generator concept is quite powerful in Python and can be used well beyond its expected applications.

Searching a collection

Beyond sorting, searching is the number one use case for a collection in any language. There are multiple ways to search a collection, depending on a number of external factors. Before deciding upon a strategy for searching, you first need to determine three things about your particular use case.

First, you need to know how often you are going to search for something. The strategies used for searching regularly are different than those for searches done only once in a blue moon. Whereas in the first case you want to structure your data specifically for searching, the second case often has you considering that almost as an afterthought.

Secondly, you need to know how fast your search needs to be. For example, if you are replying to a search request at human readable speed, it probably isn't important that your search be virtually instantaneous. On the other hand, if your search needs to do lookups for real-time processing, the speed of the search becomes a fundamental consideration in how you implement it.

Third, and finally, the size of your collection data set is important. Searches that work extremely quickly and efficiently for a small to medium data set will fail spectacularly in a Big Data environment.

In general we'll consider three searching mechanisms for Python. First, we'll look at a simple brute force approach, and a slightly more elegant way of doing the same thing. Let's imagine, for example, that we have a simple list of items. It might have a few hundred items at worst, and is not guaranteed in any way to be in sorted order. Also, the data in the collection is not guaranteed to be unique, which means that a search could return any of a set of values.

First of all, let's create a function that will generate us some random data:

```
from random import randint

# A function to generate a list of random numbers in a given range
def generate_random_list(list_size, max_value):
   ret_list = []
   for i in range(0, list_size):
     r = randint(0, max_value)
  ret_list.append(r)
return ret_list
```

This function accepts two parameters, the size of the list to generate and the maximum value to create within the list. The return from the function is a list of values. Once we have the list of values, we can explore how to search it.

We are going to use three different search types. First, we'll brute force search the list, running through it until we find a value. Next, we'll do the same thing, but use a list comprehension method to search for our value. Finally, we'll explore the concept of turning our list of values into a faster construct for searching, in this case a set. To measure the speed of the algorithm, we will use a Python class called **`timeit`**, which allows us to specify a function to run, and a number of iterations to run it, and returns the number of seconds it takes to run the function that many times. Because of the vagaries of computer processing, measuring a single call of one method vs another is often incorrect. If something is going on in the background, your answers will vary radically. By calling the function a sufficient number of times, that vagueness is smoothed out. We will run each approach a thousand times to get a solid number.

First, the brute force approach:

```
l = generate_random_list(20, 100)

value_to_search = l[-1]

# First, just try a brute force search
def brute_force_search():
```

```
   for value in l:
     if value_to_search == value:
   return True
return False

t = timeit.timeit(brute_force_search, number=1000)
print(t)
```

Next, we will look at the list comprehension approach.

```
# Next, see if a list comprehension is any different
def comprehension_search():
   r = [x  for x in l if x == value_to_search]
  return len(r) > 0

t = timeit.timeit(comprehension_search, number=1000)
print(t)
```

Finally, the set based approach:

```
# Finally, do a search based on a set
search_set = set()

def build_set():
   for value in l:
     search_set.add(value)

def find_in_set():
  return value_to_search in search_set

t = timeit.timeit(find_in_set, build_set, number=1000)
print(t)
```

For our final approach, you may notice that we created a secondary function to call to build the set. This is passed to the **`timeit`** function to be used as a **`setup`** function for our system, so that it is included in the timing.

The results of the three functions are as follows:

```
Searching for value 91
```

```
List to search: [7, 34, 5, 67, 52, 50, 70, 1, 43, 2, 13, 16, 92, 8, 85,
91, 49, 37, 58, 91]
0.0008070309999999997
0.0012734850000000013
0.0001256420000000022
```

As you can see, the time to search a set is, not surprisingly, the fastest way to search for a value. This, however, does require that you set up the set in the first place. The time to build a set is:

```
t = timeit.timeit(build_set, 1)
print(t)
3.167000000001835e-06
```

Hopefully, you can see that the overhead of creating the set is not especially high, particularly for smaller lists. For longer lists, the time to iterate through the entire list is likely to be even longer. There is, of course, work involved in creating the list, memory used to maintain the construct and so forth. If your data is changing, you will need to rebuild the set each time you want to search it.

Using a set of functions to create an extensible state machine

A *state machine* is a construct in computer science whereby an application transitions from one 'state' to another based on a set of criteria. For example, you might think of a word processor as a state machine. The initial state is with no file open. A command is issued to open a file, and the state transitions to opened, where the file is loaded from disk and displayed for the user. Once it is loaded, it might transition into the edit state, whereby the user can modify the file. This could go to the saved state if the user elects to save his or her changes or the closed state if the user decides not to edit the file or to throw away their changes.

State machines are often implemented using function pointers in many languages. In Python, we can use the fact that a function is a first-class element to store our states as a dictionary of functions to handle each of the states. Let's take a look at how we can do this in Python, and how such a construct works under the covers. First, here's the code for our application. You can store it in a file called **`state_machine.py`** (or whatever you want to call it) or load it from the files for this book:

```
def initial_state(command):
    print("Initial state")
    if command == 'start':
```

```
        return 'started'
    return 'error'

def start_state(command):
 print("Started state")
if command == 'stop':
  return 'stopped'
if command == 'open':
return 'opened'
 if command == 'close':
   return "closed"
 return 'error'

def open_state(command):
   print("Opened state")
   if command == 'stop':
     return 'stopped'
  if command == 'close':
    return 'closed'
 if command == 'quit':
   return 'initial'
 return 'error'

def close_state(command):
  print("Closed state")
 if command == 'quit':
    return 'initial'
 return 'error'

def error_state(command):
  print("Command {0} resulted in error, resetting".format(command))
 return initial"
```

```
def build_dictionary():
 dict = {}
 dict'initial'] = initial_state
 dict['started'] = start_state
 dict['opened'] = open_state
 dict['closed'] = close_state
 dict['error'] = error_state
 return dict

done = False
command_dictionary = build_dictionary()
current_state = 'initial'
while not done:
   print("Currently in the {0} state".format(current_state))
   command = input("Enter a command for this state: ")
   state_handler = command_dictionary[current_state]
   current_state = state_handler(command)
```

Let's first understand what is happening here. Each of the ***_state** functions simply handles a given command while in the state that the function is named for. If a command can't be processed for a given state, an error is return and the 'error state' is set. We build our set of command handlers in the **build_dictionary** function, which could easily be extended by adding new states and handlers. Our main loop, at the bottom of the function doesn't do very exiting things. It prompts the user for a command (using the input function, which we'll discuss at length in a forthcoming chapter) and then tries to `handle` that command using the current state handler. The state handler returns a new state, hopefully, and that is then used for future processing of user commands.

Here's what it looks like when it is running:

```
Currently in the initial state
Enter a command for this stateopen
Initial state
Currently in the error state
Enter a command for this statereset
Command reset resulted in error, resetting
Currently in the initial state
```

```
Enter a command for this statestart
Initial state
Currently in the started state
Enter a command for this stateopen
Started state
Currently in the opened state
```

You will notice that the state handler never exits the look, since the done variable is never set to true. This is probably the expected behavior in the real world. We may want to add a new command called **`exit`** or **`quit`** to terminate the loop for our purposes. This is left as an exercise to the reader, but it really shouldn't be very difficult.

Filtering vs removing

We often talk about filtering a collection as if we were removing items from the list, but this isn't the case. Most filtering returns a copy of the list, rather than modifying the original list. Python provides the **`filter()`** function to do exactly this:

```
from random import randint

list1 = [randint(0, 100) for x in range(0,10)]
print(list1)

def func_to_filter_with(x):
   if x > 10 and x < 50:
     return True
 return False

filtered_list = list(filter(func_to_filter_with, list1))
print(list1)
print(filtered_list)
```

For a randomly generated list, we get something like this as output:

```
[51, 28, 25, 4, 55, 62, 18, 72, 95, 78]
[28, 25, 18]
```

Notice that our filter function screens out everything that is not greater than ten and less than fifty. However, the returned 'filtered' list is a copy of the original, as we can

see in the two output lines. The new list does not contain the values we requested to be filtered, but it didn't modify our original list either. What if we do want to modify the original list? This is a little tricky. Python does provide the **`remove()`** method for lists, as well as the **`del`** statement that will remove an item from a list. For example:

```
list2 = [1,2,3,4,5,6]

print(list2)

list2.remove(3)

print(list2)
```

```
[1, 2, 3, 4, 5, 6]

[1, 2, 4, 5, 6]
```

It is important to know, however, that you cannot use remove in a loop for an input list. Let's take a look at the problem:

```
print("Remove example")

list1 = [1,51,8,52,9,53,2,57,3,4,55,56,57,8]


print(list1)

for x in list1:

   if x <= 50 or x >= 60:

     list1.remove(x)
```

Given the input list and the criteria for removal, you would expect to see an output of:

```
[51,52,53,55,56,57]
```

However, if you run the above code, you'll see the output is:

```
[51, 52, 53, 4, 56, 57, 8]
```

Why is this? This is a nasty little problem that has to do with side-effects of loop iteration and the **`remove`** operator. The for loop auto-increments to the next element in the list when it is running, and the **`remove`** operator modifies the actual list to be shorter. As a result, we skip over elements when we remove them. You should never remove or insert into a list while you are iterating over it. Instead, do it as a list comprehension and assign the result to the original list when you are finished:

```
list1 = [x for x in list1 if x > 50 if x < 60]

print(list1)
```

```
[51, 52, 53, 57, 55, 56, 57]
```

In the list comprehension, we first generate a new list from the original list that contains only the values between **`50`** and **`60`**. Then we assign this list to the original

variable. Note that the **list1** variable now points to a completely new block of memory!

Slicing

We've looked a little at slicing over the first few chapters of the book, now it is time to take a bit more of an in-depth look. Slicing, you may remember, is the ability to take slices of an array or string. All languages support some form of slicing for arrays, most of them allow you to get a single element. For example, given a **string="Hello world"**, virtually all languages treat the expression **string[0]** as 'H'. In Python, however, you can go a lot further than that with slices. Let's take a look at a few examples of slicing in Python to understand what can be done with them.

Possibly the most classic example of slicing in Python is to reverse a string in a single line of code. Here's how you do it, and why it works.

```
string_1 = "this is a test"

rev_string_1 = string_1[::-1]

print(string_1)

print(rev_string_1)
```

```
this is a test

tset a si siht
```

The essence of the slice operator for lists is **[start:stop:step]** or the 'three s' approach. The start is the position where you want to begin the slice. Omitting this parameter will default it to the beginning of the list, or zero. The stop parameter is the position where you want to end the slice. Omitting that parameter results in the end of the string (also called the **length of the string**). The final step parameter tells Python how many entries in the list to skip between steps. A negative number indicates that it should go backward in its iteration. So, using two defaults and a minus one results in the string going from its end to its beginning, one character at a time, resulting in the reversed string.

There's nothing magical about the string in this expression, you can do the exact same thing with a list of integer values or floats, as well:

```
array_of_integers = [1,2,3,4,5,6,7,8,9,10]

rev_array_of_integers = array_of_integers[::-1]

print(array_of_integers)

print(rev_array_of_integers)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

It might not be intuitively obvious, but the slice operator can be used to insert or remove elements as well. For example, suppose that we have a string that we wish to remove a piece of:

```
bad_string = "This is a string with a *bad* word in it"
idx = bad_string.find('*bad*')
good_string = bad_string[0:idx] + bad_string[idx+len('*bad*')+1:]
print(good_string)
```

```
This is a string with a word in it
```

For that matter, suppose that we want to replace a word in a string:

```
better_string=bad_string[0:idx]+"good"+bad_string[idx+len('*bad*')+1:]
rint(better_string)
```

```
This is a string with a good word in it
```

You can do the same for lists of integers, or floats, or complex values, or even objects. We can use the **step** operator in a positive fashion as well. Suppose we have a sorted list of integers and want all of the positive ones. We've looked at how to do this with a list comprehension, but you can do it with slicing as well:

```
all_numbers = [1,2,3,4,5,6,7,8]
even_numbers = all_numbers[1::2]
print(even_numbers)
```

```
[2, 4, 6, 8]
```

Naturally, any iterable can be sliced. For example, here's a non-homogenous list:

```
tuple_1 = ('a', 1, 2.3, "This is a test")
print(tuple_1[0:3])
```

```
('a', 1, 2.3)
```

If the slicing syntax bothers you, which it really shouldn't, Python provides a slice class as well, that can be used anywhere a slice syntax can be used:

```
# Use the slice object
array_of_stuff = [1,2,3,4,5,6,7,8,9]
s = slice(2,5)
print(array_of_stuff[s])
```

```
[3, 4, 5]
```

Finally, you can reverse just a small piece of an array, if you want to, rather than the whole thing using slicing syntax:

```
print(array_of_stuff[5:2:-1])
```

```
[6, 5, 4]
```

You might wonder why we use the reverse order of elements in the last example. If we wrote:

```
print(array_of_stuff[2:5:-1])
```

```
The result would be
[]
```

When the first two parameters are omitted, Python is *smart* enough to figure out you want to go from either the beginning to the end, or the end to the beginning, depending on the sign of the third parameter. When you do not omit the parameters, Python naturally assumes you know what you are doing and tries to go from the start to the end stepping by the step value. In the case of going from **2** to **5** by **-1**, you get an empty set. You can also use a negative **step** operator that is not equal to one and everything works as expected:

```
print(array_of_stuff[5:2:-2])
```

```
[6, 4]
```

Lambda expressions

The lambda expression is one of the most poorly understood, yet simple, concepts in any programming language. A lambda expression is just a one-line function, subject to a couple of simple rules that has no name. In most languages, such as C++, it is referred to as an *anonymous* function. The basic syntax of a lambda expression is:

```
lambda <arguments>: <code>
```

Normally, one assigns the **lambda** to a variable, which is then used exactly like a function. For example, here's a lambda that squares a number:

```
square = lambda x: x * x


two_squared = square(2)
print(two_squared)
```

There are two basic rules for a lambda expression:

- They may not contain anything but a single statement, and cannot have side-effects.
- The result of the expression is the return value from the lambda.

In Python 2, there was a third rule that no longer applies, which is that you cannot use a statement that returns no value, such as **print**, in a **lambda**. You can do that in Python 3, if you really want to:

```
log = lambda x: print("logging value for {0}".format(x))
log(12)
```

As a matter of style, this is a very poor use for a lambda, but it will work in Python 3. You can use lambdas for all sorts of things that you probably shouldn't use them for. For example, consider this ability to generate functions using lambdas:

```
def make_shifter(a_value_to_shift_by):
   return lambda x: x << a_value_to_shift_by

mult_2 = make_shifter(2)
val = mult_2(2)
print(val)
```

In this case, we are creating a new function in the **make_shifter** function via the **lambda** statement. This function, which will now be called **mult_2**, shifts left the value that is passed in by the value that was given to the **make_shifter** function. If this doesn't make sense, and honestly, it doesn't to a lot of beginning Python users, it is a good indication that you are abusing the language, rather than using it.

There are excellent reasons to use lambdas, such as in the **sorted** function. This function accepts a parameter called **key** which will permit you to change the way in which data is sorted. For example, let's create a list of a max of positive and negative values:

```
my_list = range(-3,3)
print(my_list)
my_list = sorted(my_list, key=lambda x: x*x)
print(my_list)
```

If we just sort this list normally, we'll see that you get the numbers in the same order, **-3**, **-2**, and more, to **+3**. However, by specifying a sort key which squares the value, we get back the list in sorted order without regard to the sign:

```
[0, -1, 1, -2, 2, -3]
```

We could, of course, create a function to do the same thing and pass the name of the function to the sorted routine, but this is easier and more efficient. You don't have to go find the definition of the function, or worry about someone modifying it for some other cause. It is right there in the call to sort and does exactly what you expect. The **(x*x)** expression eliminates the sign of the value, since a negative number squared is positive. It does not change the actual value, only the key used to sort the value. You can write your own functions that accept things like this, and wouldn't even know if someone passed you a lambda or a function.

The 'splat' operator and unpacking

Pity the poor asterisk. Not only is it used in English to indicate a footnote or other demarcation, but it is also used in **math** to denote multiplication. Were this not

enough, we also have its use in Ruby, and Perl, as a **`list`** operator. Python doesn't actually call the asterisk the **`splat`** operator, but most people have taken to referring to it when used to flatten an iterable.

```
Consider, for example, this code:

l1 = [1,2,3,4]

print(l1)

In this example, our output is:

[1, 2, 3, 4]
```

What if you didn't want the list output in list format? What if all you wanted was the list of values to be written to the output console? You could write them using a loop and one of the output functions, but Python prefers an easier way:

```
print(*l1)

1 2 3 4
```

The **`splat`** operator is even more useful when applied to dictionaries, if you are trying to print out just the keys:

```
d = {

'x': 1,

'y': 2.0,

'z': "Hello world"

}

print(d)

print(*d)

{'x': 1, 'y': 2.0, 'z': 'Hello world'}

x y z
```

You could accomplish the same thing in other ways, but the splat operator makes it easier. One more variant of the splat operator might be called the *double splat* operator. This one is very special and has limited applications, but when you need it, you really need it. The double splat **`(**)`** operator, when applied to a dictionary, turns a dictionary set of values into a key-value parameter list.

If we had a function like this, and called it as such:

```
def func(x,y,z):

    return x + y + z

print(func(1,2,3))
```

You'd see the result as 6. We could also write the call to the function as:

```
print(func(x=1,y=2,z=3))
```

In this case, we would get exactly the same result; we are simply passing in *named parameters* which allow us to change the order of them if we want. But what if we had all of these values bound up in a dictionary?

```
d = {
'x': 1,
'y': 2,
'z': 3
}
```

You can't call this function with the dictionary, the parameters don't match and you get an error. But you can use the double-splat operator to convert the dictionary into the named parameter set:

```
print(func(**d))
```

This prints out the same value. This really shows off the power of Python, since this is the methodology used by the interpreter to unpack values into a function. This functionality has been exposed for the programmer to use as well.

Conclusion

That finishes up our tour of the more advanced topics in Python manipulations. Hopefully, by this point, you have the skills you need to start putting together real applications in Python. From this point on, we will primarily be examining the tools that are offered with Python, beginning with file manipulations in the next chapter.

Questions

1. How do you convert a list to a string using comprehensions?
2. What is a set comprehension and how do you write one?
3. What is the difference between an enumerator and a generator?
4. How can you use a slice to reverse a string?
5. What is a lambda expression and how do you write one?

Chapter 7
File Input and Output

Introduction

While learning the types and manipulators of a language is core to understanding how to write code in that language, the most important parts are usually the pieces that make up the core libraries and functions of the language. For Python, this is as true as every other language. You may be able to slice strings and write lambda functions and even define cool new classes, but until you can do things like reading and writing for various file types, you aren't going to be producing any professional code.

Structure

- The open statement, using with
- JSON parsing
- Reading in text vs reading in lines
- Output formatting
- Pickling

Objectives

By the end of this chapter, you should be able to read and write files using Python. You will learn about text file manipulations along with JSON and binary formats.

You should be able to read input files by the character or line, and output files in either unstructured or formatted ways. Finally, you will learn something about pickling which is writing out complex data structures in a JSON format using a standard Python library.

Files

The ability to read and write files is fundamental to any programming language. For some languages, like basic, it was built into the core statements of the code itself. For others, like Fortran, C, C++,and C# it was added to the language via libraries written in low level languages. Input and output with files are core to corporate programming, be it logging information, reading input configurations, or producing reports. Python does not have *classes* per se to do input and output but it does have a nice set of functions that do the job. Let's take a look at them in this chapter. Along the way, we'll take a look at Python's generic interface to the operating system and a little bit of serialization.

Working with files

The process of working with files is the same in all languages. First, you open the file in a specified *mode*. This mode indicates whether you are going to read or write to the file. Some languages only support read and write mode. Python, as we will see shortly, like C, C++, C#, and Java support a wider spread of options.

Table 7.1 shows a list of the options you can use with the `open()` statement and what they entail to your application:

Symbol	Meaning
`r`	Read only mode. Writing to the file is prohibited.
`w`	Write mode. Writing to the file is permitted.
`x`	Exclusive mode. If anyone else has the file open for reading, this mode will fail.
`a`	Append mode. All writes will go to the end of the file if there is anything in it. If it doesn't exist, it will be created.
`t`	Text mode. Indicates that you are writing to the file as a text file. This is the default if no type is indicated in the open statement.
`b`	Binary mode. It allows you to write binary characters to the file. Not portable across operating systems.
`+`	This is the equivalent of read+write. You can read or write to the file.

The full form of the open statement is as follows, although the full version is rarely used:

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)
```

The file argument indicates the name of the file with which you wish to work.

- Mode is the argument from the table above.
- `Buffering` is used for indicating when a file should be written to disk. For binary files, this is in *chunks* of a size defaulted to for the operating system. For text files, buffering occurs at the line level. When a buffer is filled, the data is flushed to the disk file from memory.
- `Encoding` is the type of encoding to be used for the text in the file. The default is operating and environment dependent, but is normally UTF-8.
- The `errors` argument is used to determine how encoding errors should be handled. Some characters cannot be encoded in all encoding schemes and generate an error. If you are concerned about this, set this parameter to strict. If you are not worried about it because you are only writing text files, set it to none.
- The `newline` argument is used to control how the end of line character is interpreted. You can set it to `\r, \n, \r\n`, or None. This character is used to determine the end of line for reading and what to translate an end of line character into for writing. Once again, if you aren't working in multiple operating systems or your files do not need to be portable, you needn't worry about this one.
- The `closed` parameter is used to indicate whether or not the underlying file descriptor handle for the operating system is closed when the file is closed. Unless you are using a proprietary or special file system, you won't use this one ever.
- The `opener` parameter is an optional custom file opening function. This is an uncommon parameter used only when reading from a non-standard source.

Let's look at a few examples of using the open statement with various scenarios. First, let's consider an existing text file that exists on your local system.

```
f=open('test.txt', 'r')
```

In this case, we are opening the file in read only mode. Any attempt to write to the file will fail. If the file does not exist, it will fail. For example, let's imagine that we are trying to open a file that isn't there:

```
f = open('filedoesnotexist.txt', 'r')
Traceback (most recent call last):
  File "fileio.py", line 6, in <module>
    f = open('filedoesnotexist.txt', 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'filedoesnotexist.txt'
```

As you can see, the file did not exist (as we might have guessed from the name), so an error was generated. As we'll see in a little while, we can use the **operating**

system (OS) module to find out ahead of time whether or not a file exists under the name we are trying to open it as.

If, on the other hand, we use this statement:

```
f = open('filedoesnotexist.txt', 'a')
```

Now, if the file does not exist, it will be created as an empty file. The difference between the `a` and the `w` forms of the open is that if you use the w form, it will truncate the file if it does not exist, and create it if it does. The a form will create it if it does not exist, but not truncate it if it does. If, for example, we had a file called `test.txt` that contained a line like `this is the first line` and we ran this code:

```
f = open('test.txt', 'a')
f.write('this is a test')
f.close() # File still exists and contains two lines

f = open('test.txt', 'w') # File exists and contains no lines
```

The comments indicate what happens here. We can open this file for append and add a line to it, and then close it. Once we do this, it will contain two lines of text. After the second open, in write mode, it will contain no lines of text.

The return from the open statement is a `file` object. As you see in the example above, the file object can be written to using the write statement and closed using the close statement. By default, the Python interpreter will close a file object when it goes out of scope and is garbage collected. It is bad practice to rely on this, just as it is in C++ or C#, where the file classes often clean up after themselves. If you try to re-open a file when it is still open and the mode has changed, you may get strange errors that are difficult to reproduce.

Note that Python does not deal with files the way it does with the console. With the console, when you use the `print()` statement to output something, it automatically appends a new line to the end of the text. The write statement in the file class does not do the same. For example, suppose we log some data to a file:

```
f = open('test.txt', 'w')
import time
for i in range(0,5):
   f.write('{0}: This is test number {1}'.format(time.time(), i))
f.close()
```

You might expect this to write out lines containing the time and the text, one per line. This is not the case. Instead, you get:

```
cat test.txt
```

```
1564663930.1802938: This is test number 01564663930.180316: This is test
number 11564663930.180319: This is test number21564663930.180321: This is
test number 31564663930.180322: This is test number 4
```

It probably isn't clear from the printed text but this is one long line in the text file. If you wanted to have the information printed one line at a time, you need to insert a newline character:

```
f = open('test.txt', 'w')
import time
for i in range(0,5):
   f.write('{0}: This is test number {1}\n'.format(time.time(), i))
f.close()
```

This code produces the output you were expecting in the first place:

```
1564664163.536551: This is test number 0
1564664163.5365708: This is test number 1
1564664163.536574: This is test number 2
1564664163.536575: This is test number 3
1564664163.536577: This is test number 4
```

By the way, you might notice that in Python, unlike most programming languages, you can do direct formatted output to a file. This is because the file `write` method is accepting (in this case) a string, and the string formatting is modifying the string to be in the format that the developer wishes it to be.

Using the with statement

As you may have noticed, there is a pattern to using the open and write statements. You generally open a file, write something to the file, and then close the file. This pattern is so prevalent in the programming world that Python has created a simple way to accomplish it with a minimal amount of coding. This is the `with` statement. It has other purposes, but for now, let's just look at how you can use the `with` statement to avoid some of the coding overhead that is normally necessary and to avoid errors in your applications:

```
from random import randint
warning_levels = ['info', 'warning', 'error', 'critical error']
with open('test.txt', 'a') as f:
   which = randint(0, len(warning_levels))
  f.write('{0}: This is a {1} at{2}\n' \
.format(warning_levels[which],  warning_levels[which], time.time()))
```

This little code block will open a `log` file for appending to, write out a statement, and then close the file. Don't see the `close` statement? That's because it isn't written, but it is there. The `with` statement sets the scope of the variable, in this case, f, to be only within the indentation of the `with`. Once the `with` statement terminates, on the line following the `write`, the object that is created in the with statement is destroyed, thereby closing it.

You could easily encapsulate this little piece of code into a log class and use it within your own code to quickly open the log file, log information, and close the log file.

One final note on opening a file in Python, the path to the file is operating system dependent. That is, you could write the following in a Linux or Mac environment:

```
open('/usr/local/logs/log.txt', 'a')
```

```
whereas the same file in Windows environments might be:
```

```
open('c:/Windows/Logs/log.txt', 'a')
```

Also remember that the backslash character, which is often used in the Windows world as a directory separator, means something special in Python strings. You can always use the forward slash or use the escaped version (\ \) in your directory names.

Reading fixed length data from files in Python

In the programming world, most files consist of data in one of three formats. First, we have fixed length text based files. This is usually the output of some sort of older code or perhaps a text standard like **FIX** (used in financial environments). Python does a lovely job of reading fixed length files, as we can see in this simple example.

Suppose we have a file that looks like this:

```
001This is a test    ABCDE12345
002This is not a testBCDEF12345
003This is another teCDEFG12345
004This is a test 2  DEFGH12345
005This is thelast   ABCDE12345
```

The 'documentation' for this file format is as follows.

Each line consists of the following entries:

- The `line_code` which is a three character value indicates the sequence of the line in the file.

- The description which is an eighteen character free-form text field describes the data.
- The alpha code which is a five character code represents the hashed value of the input field.
- The final code is a check field to indicate whether or not the line is accepted by the system. A default value of' `12345` indicates the line is proper.

To read this file, we can do one of two things. We can read each line out of the file and then parse it into the pieces we want or we can read the individual fields one at a time. Let's experiment by reading in each field within our code one at a time. To read the file, we'd do something like this:

```
# Reading in fixed length strings from a file
with open('fixed_length.txt') as fixed:
   for i in range(0,4):
  line_code = fixed.read(3)
description = fixed.read(18)
alpha_code = fixed.read(5)
last_code = fixed.read(5)
spacing = fixed.read(1)


      print(line_code)
      print(description)
      print(alpha_code)
      print(last_code)
```

In this case, we are only reading in four lines from the file, to test our code and also because we do not currently know how to read until the end of the file. If we run this snippet of code, we'll see the following results printed to the console:

```
001
This is a test
ABCDE
12345
002
This is not a test
BCDEF
12345
```

```
003
This is another te
CDEFG
12345
004
This is a test 2
DEFGH
12345
```

As you can see, we have properly read in the file. Now, how did we accomplish this? The `read()` method of the file class reads in a specific number of characters from the file. When dealing with ASCII characters (single byte characters) this will be the same as the number of bytes in text. For a binary file, this would be the absolute number of bytes. For encoded files, this will be the number of bytes in a single character. Each piece is read from the file.This moves the file `pointer` indicating where to read from along with that number of positions. Each piece is read in and stored in the proper string field.

What if we wanted to keep reading in lines until we reached the end of the file? Python provides an easy way to tell if you've got what you expected when you do a read. It returns a sentinel value, `None`, if the read didn't return the proper number of bytes. So, when we read our first chunk indicating the number line number, we can check if we got the proper three characters. Modify our code loop at its top like this to fix the issue:

```
with open('fixed_length.txt') as fixed:
   done = False
  while not done:
line_code = fixed.read(3)
if len(line_code) != 3:
done = True
continue
```

Now, if you run the snippet, you will find that it terminates as soon as the last line is completed.

We can write this in a more Pythonic way by using tuples to store our data dictionary and using a generator to read in the file, in a way that could easily be reused for virtually any type of fixed length file:

```
# Generator to read file
def read_block(file_obj, size):
```

```
while True:
  data = file_obj.read(size)
  if not data:
   break
  yield data

# Our data dictionary
fields = [
   ('line_code', 3),
  ('description', 18),
 ('alpha_code', 5),
('last_code', 5),
('spacer', 1)
]

# Re-usable block to read a fixed length file

with open('fixed_length.txt') as fixed:
   done = False
  while not done:
    for field in fields:
     block = next(read_block(fixed, field[1]))
  if len(block) != field[1]:
  done = True
break
 print("Read block for {0} = {1}".format(field[0], block))
```

Obviously, if we wanted to make this truly reusable, we'd want to store the data rather than print it, but that small piece is left as an exercise for you.

Reading a text file by lines in Python

Sometimes, we just have a text file that contains lines of text we want to read sequentially. Python provides a few ways to read such a file into your application. Let's look at two of them and the differences between them. First, let's create a simple text file in your favorite editor and call it `text_lines.txt`. Put this text into the file:

```
This is line 1
This is line 2
    This is line 3 which is indented
This is line 5
# This is a comment

This is after a blank line
```

Note that the line following the `comment` line is blank. In the same directory (so that we don't have to deal with paths) create a Python file and place the following code:

```
# Read a text file by line
with open('text_lines.txt') as lines:
   done = False
     while not done:
      aline = lines.readline()
   if not len(aline):
   done = True
else:
aline = aline[0:len(aline) - 1]
print('['+aline+']')
```

You might be wondering about a few parts of this code, so let's examine it. First of all, of course, we open the file to read it and loop through the lines in the file. Each line is read using the `readline()` function of the file object. This function reads from the file until an end of line character (\n or \r or a combination of the two) is found in the file. At this point, it stops and returns the text that was read to the caller.

We check the length of the input line and if it is `0`, we stop the loop. Since the function includes the trailing end of line character, even a blank line will contain something. We then strip off the newline character using slicing and then print it out between square brackets, so that you can see what was read even if the string is now empty. In our case, we'll see this:

```
[This is line 1]
[This is line 2]
[    This is line 3 which is indented]
[This is line 5]
[# This is a comment]
```

```
[]
[This is after a blank line]
```

Notice that our `blank line` in the file translates into an empty string once the newline is removed, but is not considered an `end of file` marker.

As of Python 3, there is an easier way to write the above loop that works just as well and is much easier to read:

```
with open('text_lines.txt') as lines:
   for line in lines:
     line = line[0:len(line)-1]
   print(line)
```

This won't work before Python 3.4, as a warning, but it does work with later versions. We are essentiallytreating the file object as a generator of lines of text. It is a neat shortcut when you don't want to deal with the whole looping and checking.

If you aren't using Python 3, or simply don't like the generator syntax, you can use the `readlines()` method of the file object:

```
with open('text_lines.txt') as lines:
   for line in lines.readlines():
     line = line[0:len(line)-1]
   print('['+line+']')
```

It does exactly the same thing and produces the same result.

A readlines real-world example

If you have worked in the Linux/Unix world for any length of time, you've probably encountered the `tail` program. The tail program prints out the last few lines of a file, so that you can see what is going on with it. In general, it is used for viewing log files. Now, the tail program has a lot of fancy options and is capable of real-time displays, but do you ever wonder how it works at its core? Let's write a very simple `tail` program that simply shows a fixed number of lines for a file. Create a new Python file and give it the name `tailfile.py` and put the code:

```
number_of_lines = 5
file_name = 'tailfile.py'


# Open the file and read in all of the lines
with open(file_name, 'r') as f:
```

```
   lines = f.readlines()
  # Now, get the ones they want to see
 for i in lines[len(lines)-number_of_lines:len(lines)-1]:
  # Strip off trailing newline
 print(i[0:len(i)-1])
print('Done')
```

If you run this program, you'll see that it outputs the last three lines of the program itself. Later on, we'll learn how to `pass` command line arguments to the program, so you can print out any file and any number of lines. Here's the output you should see if you've done everything right:

```
    # Now, get the ones they want to see
    for i in lines[len(lines)-number_of_lines:len(lines)-1]:
       # Strip off trailing newline
       print(i[0:len(i)-1])
Done
```

So far, we have learned how to read in text files and process them in our own way. What happens if the files aren't text based but are binary instead?

Python and binary files

Sometimes, you want to work with a file that isn't in textual format. Text files are lovely and easy to read and parse, but they are a bit slow and very easy to decipher for a hacker. Sometimes, you want your data stored in a more efficient, more unreadable format. Unfortunately, that unreadability comes with a price; it is much harder to deal with binary files than it is with textual ones. With a text based file, you just read it in, figure out what you want out of it, parse it and then do whatever you like. With a binary file, it generally means you have to know what the format of the file is before you start. In the corporate programming world, it isn't unusual to have proprietary formats. We store user data in binary format because it is safer. We store images of our state in binary files so that we can load them directly into a state system without having to do a lot of conversion and checking.

Python supports binary files, as it does text files, through the underlying file object. This isn't surprising, given that the file class was written originally in C and it uses the same functionality you'd find in the standard C or C++ libraries. Working with them isn't quite as straightforward as text, though.

To write a binary file, you first have to open the file in binary mode. As we saw in the table earlier in this chapter, that means using the b argument. However, b just

tells Python to make the file open in binary mode. It does not tell it what you want to do. Thus, we have to specify either the w or r modifiers to tell Python that we want to write or read in the file. For example, let's say we want to write a binary file out and call it `mybinary.b`. The .b extension doesn't indicate anything to the operating system; it is solely for our usage. We would do this:

```
with open('mybinary.b', 'wb') as binary:
```

Of course, once we've opened the file for write mode, we need to write to it. Writing to a binary file is a little different than writing to a text file. For one thing, you need to know what data is stored there, or you need to write some kind of signal value that tells the reading program what the format of the data is. Let's assume, for this example that we know what we are doing and we are going to write out something that we can later read in. Here's the code that writes out our data, which is actually just a string. We're going to write the length of the string first, so that the reading program can then determine how big of a string to read in from the binary file:

```
# Write a binary file

with open('mybinary.b', 'wb') as binary:

    text = 'Hello world!'

    l = len(text)

    print("Length of text: {0}".format(l))

    byte_array = l.to_bytes(4, byteorder='big', signed=True)

    binary.write(byte_array)

    binary.write(text.encode('utf-8'))  # from text to binary
```

You can't really look at the binary file. Typing it in the terminal or command prompt will appear to show the string because the length byte isn't a visible character. Yet, it is there, and if you have some sort of a hexdump program, you can see it. Let's try to understand what is going on in this little exercise.

First of all, we are going to compute the length of the string using the `len()` operator as we have done so many times before. But now, we have to write that string out. This is done via the `to_bytes()` function that is a part of the `int` class. We specify three arguments to the `to_bytes()` function. First, we tell it the size of the value we are writing. Integers come in various sizes; we'll select four since that's the default `int` size on most systems. The next argument is the byte order. For most systems, you don't care about this argument since it is used only for portability. This argument tells the operating system whether you are writing a number `bigendian or smallendian`, which means whether the most significant byte of the number is the first or second part. If you don't understand this, don't worry about it, it is only needed when you move between different operating systems or different versions (32 bit vs 64 bit, for example) of the same operating system. For now, just use `big` and you'll be fine.

The final argument indicates whether the data we are writing is signed or not. This makes a difference in how the highest bit (the sign bit) of the data is interpreted. For a signed integer, if the sign bit is on, the number is negative. For an unsigned integer, it is just the highest value for that size of integer. In Python, we always use signed integers.

Now, we write out the length of the string, using the file write method as we would in a non-binary case. Then we write out the string itself, encoding it into whatever character scheme we would like to use. In this case, we chose UTF-8, which is standard text allowing for extended ASCII characters

Once the file is written, it would be nice to know how to get it back. Let's look at the code that reads in a string from a written file. First, take a look at the code and then we'll discuss it:

```
with open('mybinary.b', 'rb') as b2:
    data = b2.read(4)
    l = int.from_bytes(data, byteorder='big', signed=True)
    print("Length of string: {0}".format(l))
    data = b2.read(l)
    text = data.decode('utf-8') # from binary to text
    print(text)
```

As you can see, there are a few pieces to this. Obviously, we open the file in binary mode and indicate that we want to read from it, rather than write to it. Next, we use the `read()` function of the file class to read in a *chunk* of data. For Python binary files, everything is simply a stream of bytes. It has no idea what you want to do with it, or what it represents. This is a hangover from the underlying C code that implemented Python, to begin with. So, knowing that the file contains an integer of four bytes, we read in a chunk of four bytes from the file and then convert that into the integer value using the `from_bytes` method of the `int` class. This is printed out so we can verify that the input length of the string is the same as the length we wrote out earlier.

Once we have the length, we read in that many bytes to another *chunk* buffer. Then we convert it into a Python string by calling the decode method of the object that converts it from its binary representation to a textual representation that Python can work with and understands.

In general, when you are writing binary files, you should do it in three parts for each element you are writing. First, you write a standard `tag` that indicates to the reader what sort of data you are writing. Next, you write the length of that element so that the reader can then read in the properly sized chunk. Finally, you read the element itself and convert it into whatever the underlying data structure (`integer, float,`

`string`, and more) might be. In this way, you make your files portable and usable by others who may not have access to your original source code.

JSON parsing

JSON has become one of the most used standards in files in the software industry of late. **JSON** stands for **JavaScript Object Notation**. It is the serialization format used by JavaScript for transferring information that is stored in objects from one program to another. This may be via a file, via a REST interface, or just between two methods.

In general, a JSON entity looks like this:

```
{
"name": "value",
"dictionary_name": {
"value1": "value"
    }
}
```

It can contain single elements (such as name), dictionaries, or arrays. You can have dictionaries of dictionaries, arrays of dictionaries, arrays of entities, and arrays of arrays. Python 3 was built to work directly with JSON files and types and it translates them directly into Python dictionaries. For example, you can define a variable in JSON within a Python program:

```
json_variable = {
   "name": "value",
  "type": "a json variable",
 "dict": {
"value1": "a value",
 "value2": "another value"
 }
}

print(json_variable['name'])
print(json_variable['dict']['value1'])
```

This snippet will output the expected values:

```
value
a value
```

As you can see, the `json` variable itself looks like a valid JSON file. In fact, we could take that code after the equals sign, place it in a JSON file, and validate it with any JSON validator on the web. So, let's say that we have the above JSON in a file. How do we read it and parse it into a Python variable?

Obviously, the first step is to open the file. JSON is just text, so we won't have to bother with binary file operations here. Then, we read the text into a variable by simply doing a read on the entire contents of the file. Then we parse the JSON into a usable state. JSON isn't particularly difficult to parse but there is never a good reason to reinvent the wheel when someone has already done the job for you. In our case, the Python standard libraries contain JSON functionality, so we won't even need to install any new packages.

In this case, the package name is, not surprisingly, `json`. The method that we want to call within the `json` package is called `loads` for load from string. We could load directly from the file but we are trying to illustrate how `json` is the same as any other textual format. If you want to load it from the file, just call `json.load('file-name.json')` where `file-name.json` is the name of the JSON file, including the path that you want to read.

In our example, we took the JSON and stored it in a file called `json_example.json`. It is typical to name `json` files with a `.json` extension:

```
{
   "name": "value",
  "type": "a json variable",
 "dict": {
"value1": "a value",
"value2": "another value"
}
}
```

Note that the JSON text in the file looks almost exactly like our code, but without the variable name associate with it.

Then we write the code to process it:

```
import json


with open('json_example.json') as json_file:
   json_data = json_file.read()
```

```
parsed_data = json.loads(json_data)
print(parsed_data['name'])
print(parsed_data['dict']['value1'])
This little snippet of code prints out what you would expect:
value
a value
```

As mentioned, you can do the reading in a single line using the `load` statement of the `json` package as follows:

```
with open('json_example.json') as json_file:
   json_data = json.load(json_file)
print(json_data['name'])
print(json_data['dict']['value1'])
```

The difference is minimal, and you can choose which way works best for you. The main reason to use loads, rather than load, is that you can work with strings input from the user, or the one stored within your own application. The `json.load` requires that the entire file be in standard JSON format, so if your file contains other information, you should load the JSON string first and parse it with loads, rather than using load directly.

JSON writing

If you can read JSON from a file, you will want to be able to write JSON to a file. You may have noticed that a JSON file is an exact representation of a Python dictionary. As a result, it is trivial to write out a dictionary to JSON:

```
data = {
   'value1': 1,
  'value2': 2,
'a_dict_of_values' : {
'd1': 'hello',
'd2': 'world'
},
'value3': 1.234
}
with open('data.json', 'w', encoding='utf-8') as f:
   json.dump(data, f, ensure_ascii=False, indent=4)
```

The `dump()` function is used to write out `json` to a file assuming that the data is in dictionary format. Notice the `ensure_ascii` flag that allows us to write data to the file even if it is not in pure ASCII format (for example, extended characters in the string).

The `indent` parameter to the write makes the output look prettier. If we examine the output file produced by this snippet of code, we will see:

```
{
    "value3": 1.234,
    "value2": 2,
    "a_dict_of_values": {
"d1": "hello",
"d2": "world"
    },
    "value1": 1
}
```

Omitting the indent parameter produces the following JSON file:

```
{"value3": 1.234, "value1": 1, "a_dict_of_values": {"d1": "hello", "d2":
"world"}, "value2": 2}
```

This file is a perfectly legitimate JSON and can be read by your programs or any other programs expecting a JSON file input, but as you can see it is much harder to read. Finally, you can use the dumps() method of the json package to produce a string instead of writing to a file. This is useful for storing in databases or log files.

Serializing complex objects in JSON

If you have played with the code above a little, you must have discovered that it works just fine. You can write out a dictionary of values to a JSON file and read it back with ease. In the academic world or for toy projects, that's usually all you need to do. Sadly, the professional programmer is usually stuck working with non-trivial programs and problems. One of these is the fact that we like to do object oriented programming, and as such, end up with many classes and objects. These classes need to be serialized as well, and the methods we've looked at so far just don't work. For example, consider the following simple `Person` class:

```
import datetime


class Person:
```

```
   def __init__(self, first, last, birthdate):
  self.first_name = first
self.last_name = last
self.birthdate = birthdate
```

Obviously, our real class would be more complex but this is the only part that matters when we are discussing serialization of the class. If you try to dump an instance of this object using the `json dumps()` method, you will find that it doesn't work:

```
p = Person('matt', 'telles', datetime.datetime(1991, 1, 6))
print(json.dumps(p))
Traceback (most recent call last):
a value
  File "json_test.py", line 20, in <module>
    print(json.dumps(p))
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/
json/__init__.py", line 231, in dumps
    return _default_encoder.encode(obj)
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/
json/encoder.py", line 199, in encode
    chunks = self.iterencode(o, _one_shot=True)
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/
json/encoder.py", line 257, in iterencode
    return _iterencode(o, 0)
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/
json/encoder.py", line 179, in default
    raise TypeError(f'Object of type {o.__class__.__name__} '
TypeError: Object of type Person is not JSON serializable
```

You might be asking why this is not serializable. The reason is that the basic JSON functionality in Python is somewhat limited and only works with basic types. In a while, we'll look at a solution to this with an external package but for now, let's focus on solving the problem using the basic Python libraries and packages.

Fortunately for us, Python thought this through as it does with most problems. There is a parameter that can be passed to the `dumps` (and dump) function that acts as an 'encoder' for the specific data type you are trying to convert into JSON format. The parameter is called the `default` parameter and it is used like this:

```
print(json.dumps(p, default=PersonEncoder))
```

So, all we need to do is to create a `PersonEncoder` function and it will be called by the `dumps()` method to write the pieces out properly. As a note, you can also implement a full blown class and override its default method (which is what this argument is doing). Let's take a look at the class and see what it does with its input and the output it produces:

```
def PersonEncoder(p):
   if isinstance(p, Person):
  dict = {
 "first_name": p.first_name,
"last_name": p.last_name,
"birthdate": p.birthdate.strftime("%d %b %y")
}
return dict
else
  type_name = p.__class__.__name__
raise TypeError("Unexpected type {0}".format(type_name))
```

When we run the snippet of code now, we get:

```
{"first_name": "matt", "last_name": "telles", "birthdate": "06 Jan 91"}
```

This output is basically what we were trying to accomplish. We could modify this output by adding the indent parameter to make it prettier or we could replace the `dumps()` method call with a call to `dumps()` and write it out to a file, but the basics are here. Now, how does it work? First of all, we have defined a function that accepts a single parameter. For defensive purposes, we make sure the object that the developer has passed to us is, in fact, a `Person` instance. We could check if each one of the attributes existed in the object so that it worked with anything that was *Person-like*, but that's generally frowned upon. Act on one thing and make sure that it works is the Pythonic mantra.

Once we are sure we are working with a `Person` object, the next decision is what to do to it to convert the output properly. JSON, as we have seen, is basically a Python dictionary (or perhaps vice versa) so it makes sense to return a dictionary, but the result returned can be any serializable object. Thus, a tuple, array, simple value, dictionary or set can all be returned from this code.

To understand what is going on here, realize that the `json dumps()` method instantiates an object of type `JSONEncoder`. This class *understands* all the basic Python types: `int`, numbers (`float` and `decimal`), strings, booleans, arrays, and

objects. For objects, the only one directly understood is the dictionary. For each element passed to the `dumps()` method, the method does two things. First, it checks if there is an overloaded default method defined in the method call, as we have used. If there is, it calls this for each object. The second possible step is to see if someone has implemented a full overload of the `JSONEncoder` class and passed it into `dumps()` via the `cls=<class>` option. In that case, serialization is given to that class to perform. If neither of these is true, the basic `JSONEncoder` class is called and the result is written to the output (string or file).

If you wanted to overload the entire class, you could do that as well easily:

```
class PersonEncoderClass(json.JSONEncoder):

 def default(self, obj):
    return PersonEncoder(obj)

print(json.dumps(p, cls=PersonEncoderClass))
```

This uses the function we defined earlier, in your own code; you'd probably place that code within the specific class implementation.

Reading in text vs reading in lines

We talked briefly about the `readline` and `readlines` methods of the File class, but it is important to understand that each of these methods interprets a *line* very specifically. In Python 2, it was important as to which of these methods you selected. The `readline()` method would read a single line and return it. The `readlines()` method would read the entire file and then parse it into single lines based on the newline character. For Python3, the difference is really semantics and intent. If you call `readlines()` on a very large file, you will read the entire file into memory, which is not only memory intensive, but also very slow. In this case, and in cases where you only need to read a few lines, you should use the single `readline()` method instead.

Writing out lines

Clearly, if one can read in multiple lines from a text file, then one can write out multiple lines to a file, right? Of course, and Python provides two ways to do so, which are ever so slightly different. First, there is the `writeline()` method of the file class, that writes out a single line to a file. Alternatively, there is the `writelines()` method that writes out a block of lines to a file in a single call. Both of these methods operate similarly but differ in one very important aspect.

If we want to write out a bunch of lines using the `writeline()` method and make sure that we can have each line end up on its own line, we do this:

```
lines = ['When in the course of human events',
         'It becomes necessary for one people',
         'to dissolve the political bands which have connected them with
another',
          'and to assume among the powers of the earth, the separate and
equal station',
         'to which the Laws of Nature and of Nature's God entitle them,'
]
with open('thomas_jefferson.txt', 'w') as out_lines:
for line in lines:
  out_lines.writelines(line+'\n')
```

This code will output the start of Thomas Jefferson's famous quote to a file with a newline appended to the end of each line.

There is also the `writelines()` method, that writes lines in a batch to a file but does not append the new line to the end of the line so that they all run together in the file. This can be preferable, depending on how you need the data structured, but it is certainly something to be considered. This is an area that often trips up new Python programmers. The `readline` and `readlines` methods probably ought to be strip off the newline from input lines while the `writeline` and `writelines` methods probably ought to append them, but neither does. You can create your own method to do this, if that's your preference, and that's how Python rolls. If you aren't happy with the way things are done by default, then inherit from the class you want to modify and add the functionality you like. You get what you want, the rest of the Python community sticks with what they are used to, or, if your solution is wonderful, adopts it instead.

Output formatting

As much as we would love for the world to get away from the printed document and move into a paperless society, it is probably never going to happen. As long as there are printed documents, of course, there will be requirements that the documents be *pretty*. In programming parlance, the term 'pretty' means *well formatted*. We've looked a lot at output in this chapter but we haven't really focused at all on formatting the output. Let's take a look at that now.

If you are from the C or C++ world, you are certainly familiar with the printf function. This function is the core of formatted printing in those languages. In fact, the function name itself comes from the print formatted concept. Java has the `String.format` function which works in almost the same way. For Python, it is a little more complicated since the data type is not known in all cases, but when we

do know what we are printing, we can format it the way we like. Let's imagine that we have a set of data representing students in a class. This data contains three fields, the name, the overall grade in the class, and the number of days the student missed class. The data itself looks like this:

```
data = [
 {
  'name': 'matt',
  'grade': 98.6,
 'days_missed': 4
},
{
  'name': 'teresa',
'grade' 99.9,
'days_missed': 0
},
{
  'name': 'sarah',
'grade': 92.0,
'days_missed': 12
},
{
  'name': 'rachel',
        'grade': 92.0,
        'days_missed': 4,
},
{
        'name': 'jenny',
        'grade': 89.0,
        'days_missed': 0
}
]
```

As you can see, we have a data structure which is an array of dictionaries, each of which contains the same data. We could have implemented this as an array of class objects but this is simpler and easier to see for this example. What we want is to output this data in columnar format, so that each of the elements lines up vertically

and properly spaced to fit a given number of spaces in the output report. Our first attempt might look something like this:

```
for d in data:
  printd['name'], d['grade'], d['days_missed'])
```

You might think that this would properly create columnar output and you would be right. However, when we look at the output, we see that it looks like this:

```
matt 98.6 4
teresa 99.9 0
sarah 92.0 12
rachel 92.0 4
jenny 89.0 0
```

This meets our criteria for output to be in columns but the columns are not aligned. The longer names push the grades to the right. We can fix this, but to do so, we need to learn a little bit about formatting in Python.

The `print` statement in Python with arguments can be thought of in the following generic method:

```
print('<formatting statement>' % (values))
```

Where `<formatting statement>` is a `printf` style set of formats and the (values) argument is a tuple containing the list of data elements to print. For example, for our report output above, we'd have something like this:

```
for d in data:
    print("<formatting statement" % (d['name'], d['grade'], d['days_
missed']))
```

The trickis figuring out what the formatting statement should look like. Each element of the formatting statement looks like this: `%fw.pc` where w is the width of the output column, `p` is the precision to be used for the element, `f` is a set of potential flags to use when outputting the column, and c is the character that represents the type of data we are writing.

From the Python documentation, the possible values of the flag part are shown in *Table 7.1:*

Flag	Meaning
#	Value uses *alternative* form.
0	Numeric values are zero padded.
-	The value is left justified.

(space)	The value should be printed with a space leading.
+	Numbers should be printed with a leading +.

Table 7.1: *Python formatting flags*

The width argument, as well as the precision argument, are numbers that are used to determine how wide the column should be and how many decimal places should be used. For example, `7.2` say to format the number to `7` characters wide, with `2` digits after the decimal place.

The c argument is a little more complex. It lists the type of data you wish to have output. Once again, from the Python documentation, the list of possible values and their meanings is shown in *Table 7.2:*

Conversion	Meaning
d'	Signed integer decimal
i'	Signed integer decimal
o'	Signed octal value
u'	Obsolete type – it is identical to d
x'	Signed hexadecimal (lowercase)
X'	Signed hexadecimal (uppercase)
e'	Floating point exponential format (lowercase)
E'	Floating point exponential format (uppercase)
f'	Floating point decimal format
F'	Floating point decimal format
g'	Floating point format. It uses lowercase exponential format if exponent is less than -4 or not less than precision, it uses decimal format otherwise.
G'	Floating point format. It uses uppercase exponential format if exponent is less than -4 or not less than precision, it uses decimal format otherwise.
c'	Single character (accepts integer or single character string)
r'	String (converts any Python object using repr())
s'	String (converts any Python object using str())
a'	String (converts any Python object using ascii())
%'	No argument is converted, results in a % character in the result.

Table 7.2: *The conversion type list in Python formatting*

So, given all this information, how do we output the data we want in columnar format so that it looks'pretty'to the end user? The answer, anticlimactically, is pretty simple:

```
for d in data:
   print("%-10s %5.2f %3d" % (d['name'], d['grade'], d['days_missed']))
```

We output the name as `10` characters, left justified. The grade is output in a field of five characters, with two digits to the right of the decimal point and the number of days missed is output as a whole number in three characters. In the case of the numbers, they are all right justified so that decimal points and such line up properly. The output is:

```
matt       98.60   4
teresa     99.90   0
sarah      92.00  12
rachel     92.00   4
jenny      89.00   0
```

As you can see, everything works just as it should! Oh, finally, yes, this works perfectly for writing to a file as well:

```
with open('formatted_data.txt', 'w') as formatted_output:
   for d in data:
    formatted_output.write("%-10s %5.2f %3d\n" % (d['name'], d['grade'], \
       d['days_missed']))
```

Pickling

Earlier in this chapter, we discussed serializing Python objects. We looked at how you would *specialize* the JSON encoding of your own data structures. We had mentioned that there are easier ways to do this than to override the JSON encoder in Python. As our last topic in this chapter, let's look at that problem and how Python programmers have solved it. For Python, serialization has a number of issues. First of all, the JSON file format is easily read and isn't safe for storing sensitive information. In addition, because it is a textual format, JSON is large and often unnecessarily complex. For this reason, the Python programmers designed a system called **pickling** objects which stores the data in a well-known binary format that can be sent to other applications and *unpickled* into the original data. Pickle files are not *safe* as you can dump them in a hex program or other investigation, but they are compact, store the data in a logical fashion, and can be trusted to always be deconstructed back into their original formats.

Here's how you do pickling in Python:

```
import datetime
import pickle
```

```
class Person:

   def __init__(self, first, last, birthdate):
     self.first_name = first
   self.last_name = last
 self.birthdate = birthdate


p = Person('matt', 'telles', datetime.datetime(1991, 1, 6))
with open('person.pickle', 'wb') as pickle_file:
   pickle.dump(p, pickle_file)
```

You will note that the pickle file is in binary format. Pickle does all the work of converting each of the elements of the object, a `Person` in this case, into a binary format. It also keeps track of the type of data in the file. Pickling is awesome for things like storing configurations for systems, or game save data or any other use where you want to make sure you get back exactly what you wrote out.

Reading a pickle file is equally trivial:

```
with open('person.pickle', 'rb') as pickle_file:
   p2 = pickle.load(pickle_file)


print(p2.first_name, p2.last_name, p2.birthdate)
```

The `pickle` library does all the work of loading the file, parsing out each element, and putting it into an object of the proper type.

There are a few things you need to understand about *pickling*. First, it is not hacker-proof. Pickle files can be modified, sometimes in malicious ways. Thus, only use this for internal purposes. Secondly, pickling does not handle versioning. If we modify our class to add or remove an attribute, bad things will happen if we try to unpickle the file into an instance of the new class.

That wraps up all of the basic file operations in Python. At this point, you should be able to write programs of some reasonable size, if not in complexity. We will be working on complexity in the next chapter.

Conclusion

In this chapter, you should have learned about working with files in Python. Files are fairly simple collections of data that are essential to any professional program. They require you to understand the use of the `File` class in Python and allow you to reuse the same structure to write binary, unstructured, and structured text files. Hopefully, you learned about JSON and pickling, and how they can be used to store your application data.

Questions

1. What is the `with` statement and why would you use it in Python?
2. What is the difference between standard text files and JSON files?
3. How do you write to a binary file in Python?
4. What is pickling and why would you use it?

Chapter 8
Imports and Reuse

Introduction

One of the areas that separate amateur programmers from professionals is the area of reuse. Amateur programmers tend to reinvent the wheel with each new thing they learn. Their code tends to be of the *copypasta* variety, with the same code appearing over and over across modules. This is because they remember doing it that way once, and just copy the code over and modify it slightly for a new use. Professional programmers, of course, know better than to copy code. They will look at existing code to see if it does what needs to be done and reuse or refactor it to make it work more generically. This way, less code gets written, which means less code needs to be maintained and debugged.

Structure

- Importing modules
- Importing packages
- Dynamic imports
- Paths and setting
- Using the os module – directories and file information
- Listing functions in a module

- Listing Python files in a directory
- Reflection

Objectives

In this chapter, you will learn a lot more about Python packages and modules. You'll learn how to import them from the external environment, how to create your own, and how to dynamically import packages when you need them at runtime. We'll explore a few packages built into Python to do things such as getting a list of the modules available and the functions within them. Finally, we'll explore the reflection module in Python to get information about your own packages and functions.

Import and reuse of code

In Python, the basic unit of reuse is the package. A package is a set of Python files in a directory structure. The name of the directory structure defines the name of the package, and any subdirectories in the structure are subpackages. There are three types of packages in Python:

- **Cache package:** First, we have the `system` package. These are packages we have already looked at, such as `os, sys,` and `json`. These packages ship with the Python install and are automatically available to you when you start up the system.
- **Third-party package:** These are packages that are installed into your system via the `pip` command. Examples of third-party packages might be `flask` or requests; we will look at both in a future chapter.
- **Local package:** These are packages that you define in your own code or projects, or copy over from other projects to use in your current project. Local packages are structured the same way, as we will shortly see, and are treated the same as if they were third-party of system packages. The beauty of Python is that it is flexible and extensible, but also that it treats all things the same way.

When you request a package, via an import or another method, Python uses a specific order of operations to find it. First, it searches the modules cache, which contains every package that has been loaded into the system during your session. You can look at the system cache by examining the `sys.modules` variable:

```
import sys

print(sys.modules)
```

For the version of Python being used in this book, this producesthe output shown in *Figure 8.1:*

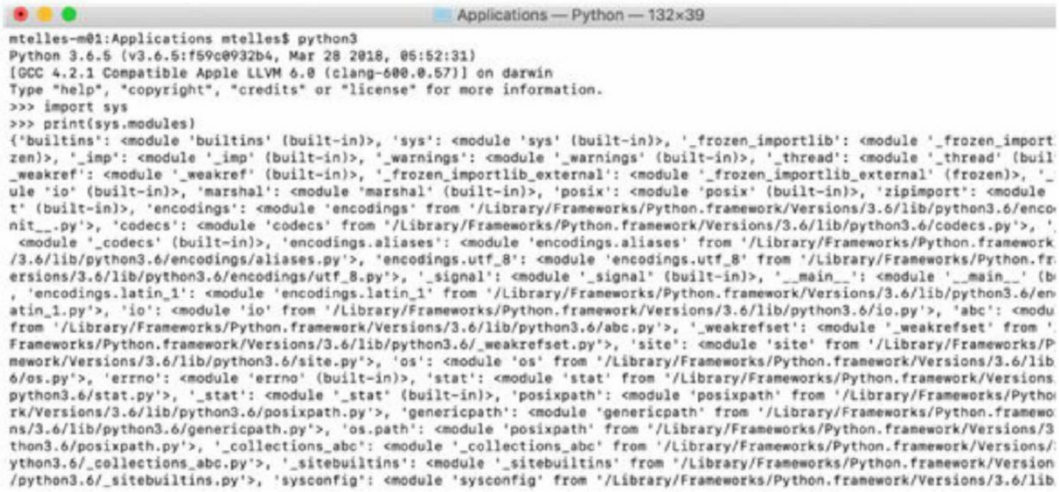

Figure 8.1: *The packages loaded in Python locally.*

If the package is not found in the cache, Python then searches the built-in modules and packages. These would be the packages like os, sys, and so forth.

Finally, if the package is not found in either of these two paths, the local file structure is examined and the package is either found, or a `NameError` is generated.

Why do we care about all this? We care about how Python loads packages because we use a lot of packages when writing Python code. It is the Pythonic approach to not reinvent the wheel, to not do work that others have done already, probably better than us and certainly better tested. Python considers reuse to be so important that it has an entire website (PyPi) devoted to finding and storing packages that are useful for Python developers.

It is important to understand the difference between a *module* and a *package* in Python. In most cases, the terms are used almost interchangeably, and for good reason. A module is one or more Python files that can be imported for use in another Python file. A package is a special form of a module that contains metadata about itself via the `__init__.py` file that must be present. We'll get into more details about this in a while, but for now, understand that all packages are modules but not all modules are packages.

Importing

In Python, the import statement brings code into your module so that you can work with it. If you don't import a module or package, trying to use it will result in an error from the interpreter. Importing is closely related to scope in Python. Scope determines which name Python associates with a given text string. For example, in classes, we have:

```
class Foo:
   my_name = "Foo"
   def __init__(self):
      self.x = 10
   def process_something(self):
      a_local_variable = self.x * 10
f = Foo()
```

In this tiny bit of code, we have four different levels of scope. The `f` variable, of type `Foo`, is defined from the line on which it appears all the way to the bottom of the program file. However, if we try to use `f` above its definition (or assignment, which in Python is the same thing), we'd get an error.

Likewise, the `Foo` class is a scope unto itself. It is used by `f`, so it has to be defined before `f` is instantiated as a `Foo` object. This is similar to other languages like C++ or Java, where you must define or include a class before you can use it.

Within `Foo`, we have the `my_name` class variable. This variable is defined for the entire scope of the `Foo` class, and can be used outside of it by prefacing it with the scope name:

```
n = Foo.my_name
```

Within the `__init__` method we see an object instance variable (or attribute) called x, which is assigned the value of x. Since the `__init__` method is called before any other method in the class, you can safely use `self.x` in any method of the class, as shown in the `process_something` method.

There is a local variable called `a_local_variable` in the `process_something` method. Aside from the rather ridiculous name, it is important to realize that `a_local_variable` is only valid within the `process_something` method. You can't use it from outside the class, you can't use it in another method, and you can't use it before it is assigned a value in the `process_something` method.

Scoping also applies to `import` statements. You cannot use something from a package before you import it.

The `import` statement itself has two variants:

```
import <package-or-module>
```

and

```
from <package or module> import <some symbol>
```

Imports are done via one of the two methods:

- The first version imports an entire package and anything in the package but requires that you scope the names of whatever you imported so that Python can figure out what you want.

- The second version imports a very specific symbol from the package specified and then scopes it to be within the current file.

An example is probably useful here. Looking back at our printing of the system modules, you will notice that we imported the entire sys package and then accessed the `modules` variable from it using `sys.modules`. On the other hand, we could have written this snippet like this:

```
from sys import modules

print(modules)
```

Notice that we are only importing the `modules` element from the `sys` package. If we tried to use something else from sys:

```
print(sys.builtin_module_names)
```

It generates the following error from the interpreter:

```
Traceback (most recent call last):
 File "html/package_test.py", line 6, in <module>
  print(sys.builtin_module_names)
NameError: name 'sys' is not defined
```

This seems a bit strange, given that we imported `sys` above, doesn't it? But in fact, we did not import the package. We imported a single element of a package, the element being `modules`. To fix this, we can either import the entire `sys` module:

```
import sys

print(sys.builtin_module_names)
```

Or, we can once again import the piece we want and remove the scope:

```
from sys import builtin_module_names

print(builtin_module_names)
```

From a Pythonic point of view, the second approach is usually preferred, as it reduces the overhead. However, if you are going to be using a lot of the module, it probably makes sense to just import the whole thing, as we did in the past with the json package.

One more word about Python imports and conventions— it is a commonly accepted practice to import packages and/or modules in the following order:

1. Standard library imports (`sys`, `json`, and more).
2. Third party imports (`requests`, `flask`, and more)
3. Local program imports (local packages or modules)

Now that we understand this, let's do a bit of importing!

Importing modules

If you come from the OOP world, as most of us do, you are most likely accustomed to breaking up your code into pieces, with a single file containing each class, and files that might contain utility functions that are used throughout the system. Python helps you implementsuch structures easily. For example, let's say we have written some functions that we are extremelyproud of. We place these functions in a file called `imported_functions.py` because we want the rest of the world to import our functions and use them!

The `imported_functions.py` file looks like this:

```
def double_me(x):
   return x*2


def triple_me(y):
   return y*3


def reverse_a_number(n):
   s = str(n)
  s = s[::-1]
 return int(s)
```

These functions are not exactly the things that dreams are made of, with the possible exception of the third one, which uses some cool Python coding to reverse a number, rather than a string. Anyway, we shall assume that these functions and file were written by programmer A. Now, a bit later, programmer **B** comes along and wants to use the function `reverse_a_number`, because it is so cool, in his own file in the same project. We'll call his file, `main_import.py`:

```
import imported_functions

print(imported_functions.reverse_a_number(456))
```

Of course, our intrepid programmer B could also write this in a more efficient way:

```
from imported_functions import reverse_a_number


print(reverse_a_number(456))
```

In the latter case, those two marvelous other functions in the file are ignored, but the code is a little smaller. In both cases, however, we see that things work just the way we expect them to:

When we are importing modules, this is all there is to it, with one exception. You can create subdirectories where you place code that is alike. For example, we might have a directory tree that looks like this:

```
<project>:
   main.py
   utils:
   utilities.py
```

Let's say that `utilities.py` contains a function, `reverse_a_string`:

```
def reverse_a_string(s):
   return s[::-1]
```

Returning to our main file, we now want to import this function into our code. You might think we could do something like this:

```
from utilities import reverse_a_string

print(reverse_a_string("Hello world"))
```

This fails with an error, saying that you can't find the `utilities` module:

```
Traceback (most recent call last):
File "main_import.py", line 5, in <module>
    from utilities import reverse_a_string
ModuleNotFoundError: No module named 'utilities'
Instead, you must do it this way:
from util.utilities import reverse_a_string

print(reverse_a_string("Hello world"))
```

The *dot notation* for finding things is probably known to anyone who has ever used Linux or Unix. For Windows, just think of them as slashes and all will be fine. If you wanted to import all the functions in the file, you could use the standard `import` statement:

```
import utils.utilities


print(utils.utilities.reverse_a_string("Hello world"))
```

Of course, you can have as many nesting of directories as you like, and to use them, just import the directory list with dots (.). The `import x.y.z.a.b.c.functions` would `import` a module called functions.py in the subdirectory tree x/y/z/a/b/c.

Importing packages

When you want to use a new package, you need to import it into your code base. There is no way to *globally* import a package, so you have to import it where you need it. It is typical, and standard, to do so at the top of the file in which the file is needed. As mentioned, if you are only using a single class or function from the package, it is generally best to use the `from <package> import <what-you-need>` format. Note that if you start importing a single element from a package and discover later that you need to import more or even all the elements from that same package, you have two options. You can change the `import` statement to bring in everything from that package by using:

```
from <package> import *
```

Alternatively, you can change your import to import `<package>`. There are some advantages and disadvantages to the asterisk (*) approach that you should be aware of before doing it this way.

Going back to our `import` example, we could change the import in our `main_import.py` file to read:

```
from imported_functions import *


print(reverse_a_number(456))
```

In this case, we do not need to scope our function call to `reverse_a_number`, which we would need to do if we simply imported `imported_functions`. There's a downside to this i.e. you are pulling in a lot of other functions you may or may not need. That doesn't really hurt things that much, they just occupy some memory. That said, it does hurt you in namespace collision.

Suppose, you have two packages, `pkg1` and `pkg2`, that have a single file in them, `utilities.py` each. In `pkg1's` version of `utilities`, there is a function called `print_me` which prints out a specific sort of data. In `pkg2's` version of `utilities` there is another function called `print_me` which prints out a very different sort of data.

```
def print_me(s):
   print("Utilities print: {0}".format(s))
```

Listing 1: pkg1/utilities.py

```
def print_me(s):
 print("Utilities 2 print: {0}".format(s))
```

Listing 2: pkg2/utilities.py

Neither of these are problems unto themselves. But let's create a file for a new project and include both of these projects in it:

```
from pkg2.utilities import *
from pkg1.utilities import *


print_me("This is a test")
```

Without running the code, try to guess at whether the `print_me` function in the first `utilities` package or the second `utilities` package is called. Then take a look at this snippet, which is almost, but not quite, the same:

```
from pkg1.utilities import *
from pkg2.utilities import *


print_me("This is a test")
```

Does it surprise you to know that these two snippets produce different results? Remember, when we talked about functions, you learned that Python does not support overloading. If you create two functions of the same name, you get the latest one that is processed. The same fact applies if these two functions are imported from different packages.

You might think that you can fix this by doing something like this:

```
pkg1.utilities.print_me("This is a test")
```

This doesn't work, and the reason is that `pkg1` is being treated as a module, and not a package. We can fix this by making it into a true package. In the `pkg1` directory, create a file called `__init__.py`.

In this file, place a single line:

```
from . import utilities
```

This syntax is known as *relative importing* and only works within packages. You cannot use the syntax in your main program or within a file outside a package definition. Now, you can change your main program to read:

```
import pkg1


pkg1.utilities.print_me("This is a test")
```

If you do it this way, all works as expected. If you duplicate the same thing in the `pkg2` directory, you will find that you can call both of them directly.

```
import pkg1
import pkg2


pkg1.utilities.print_me("This is a test")
pkg2.utilities.print_me("This is another test")
```

The output from this snippet is, as expected:

```
Utilities print: This is a test
Utilities 2 print: This is another test
```

Of course, you are required to fully scope the call within your main program. If you do not do this and try to use the `from <package> import utilities` syntax, you will find that once again, you are stuck with the last in wins scenario.

Dynamic imports

So far, the import statement in Python probably looks very familiar. You can include another package, similar to how the `#include` statement in C++ or the import statement in Java work. However, there is a side effect for Python that might not be intuitively obvious from the beginning. Because Python is an interpreted language, unlike the statically compiled languages C++ and Java, you can do anything the interpreter does. That's because you are given access to all the functionality of the language itself.

For Python, dynamic importing is a three-step process. First, you must locate and load the module that contains the code that you want to import. Next, you must add that module to the system modules list so that the interpreter will find it when you want to use it. Finally, you must do something with the module that has now been added to the system. There's a fourth, optional, step that allows you to set up the path that is searched for the modules. Let's look at each piece separately.

First of all, we need to locate and load the module. For this example, we are going to assume that the file name is `class.py` and that the file is located in the current directory. This may or may not match your case, but it is easy enough to modify. To do this, follow these steps:

1. First, we load the module that contains our code:

```
import os
dir_path = os.path.dirname(os.path.realpath(__file__))


path = dir_path + "/class.py"
module_name = "Foo"
```

```
import importlib.util
import sys

# Load the module from the specified location
module_spec = importlib.util.spec_from_file_location(module_name,
path)
module = importlib.util.module_from_spec(module_spec)
```

2. Next, we add the module to the system `modules` list and load the module into the interpreter:

```
# Add the module to the system modules list
sys.modules[module_spec.name] = module

# Load the module into the interpreter
module_spec.loader.exec_module(module)
```

3. Presumably, we know the name of the class we are trying to create, so we can now do that:

```
# Now, create the class from the module
c = module.Foo()
print("Foo.var = {0}".format(c.var))
```

4. Finally, we add the path (if it is not already there) to the system path for Python to search for imports and resolution:

```
# Set up the path so other things get processed properly
sys.path.append(dir_path)
```

You might wonder what if I don't happen to know the name of the class I want to import. This can happen in the case where you are writing an extensible system. Users can extend it by writing their own modules that conform to some specification. So, you might know the name of the class, in a string, and the method that you want to call, from the specification, but not have the name in your code. Python permits that too. Take a look at this simple code:

```
# Create a class from a string instead
class_name = 'Foo'
class_ = getattr(module, class_name)
instance = class_()
print("Foo.var = {0}".format(instance.var))
```

Notice that we are passing the class name as a string `Foo` to the module to retrieve the class definition from the module using the `getattr` method. This method retrieves information from the object dictionary, as discussed in previous chapters.

Finally, what if you don't know what the information you want to get out of the dynamically created object might be? Suppose you are allowing the user to import data from their own objects. They give you a class name and a variable within that class and want you to load that data into your application. Not surprisingly, we can use the `getattr` method to do this as well!

```
instance = class_()

var = getattr(instance, 'var')

print("Foo.var = {0}".format(var))
```

In each one of these cases, Python does all the heavy lifting for you, and you get back exactly what you expect to from the dynamic class or object.

Working with the os module

You may have noticed the use of the os module in the example used in the previous section where we retrieved the current working directory. The os module is a direct wrapper around system information, but generalized in a way so that it works equally well on a Linux system, on Mac, and a Windows box. It is worth taking some time and space to investigate this very useful package in Python. We will skip the operating system dependent methods in the module since they will just confuse those of you working on a different operating system.

Working with the environment is done with the `getenv` and `putenv` methods. These two methods return environment variable settings. In Windows, these can be set for a given process or they can be set in the operating system via a GUI program. In either case, the usage is the same. To retrieve a given environment variable setting, you need to use the `getenv` method:

```
setting = os.getenv('name-of-variable')
```

If the variable requested does not exist, then in typical Pythonic dictionary fashion, it returns `None`. Similarly, there is a `putenv` method that will set the value of a given environment variable:

```
os.puteve('name-of-variable', value)
```

Python being Python, there is an `unsetenv` that will delete a given environment variable. It is important to note, however, that environment variables are only set for the current process (application run). Once your application terminates, the environment will revert to its previous state.

Why would you use environment variables? Suppose that you need to store some information *globally* to be available to all parts of your application. This might

include the current user, or perhaps some information that other modules will need while they are running, like the current directory name, or the role that the current user is fulfilling in the application. In your startup code, you might have something like this:

```
if os.getenv('role') == None:
    os.putenv('role', 'Manager')
```

And then check the `role` environment variable later on in the program. Sadly, this doesn't work in all cases and is not the preferred method for doing things. Getting an environment variable via the `getenv` method is always a good thing to do, and always works. However, the `putenv` method is not always supported directly in all operating systems. It will appear to work, but will not change the environment. Instead, you should use the environ variable within the `os` module. Let's look at a good example of how you might do this:

```
import os
if os.getenv('user_role') == None:
   os.environ['user_role'] = 'Manager'


# Now, somewhere else in the code


if os.getenv('user_role') == 'Manager':
   print("You are a manager")
```

Not surprisingly, if you run this code, you will get `You are a manager` printed to the console.

Another set of matching methods that is very useful are the directory functions, `getcwd` and `chdir`, which retrieve the current working directory and set it. Note that the format of the directories is operating system dependent. These two functions modify the environment directly, so you can set the currentworking directory via the chdir (change directory) method, and then retrieve it later via `getcwd` (get current working directory). If these methods look familiar to you, this is because they are direct models of the Linux/Unix terminal functions.

Before moving on to directories and file information we'll discuss one more useful function i.e. the `tmpfil()` method. The `tmpfil()` method creates a temporary file that you can use as a file object. Thus, you can write to it, read from it, and so forth. The difference between this and the usual file is that a temporary file will be deleted as soon as all the file descriptors pointing to it (for example, anything that has the file open) are gone.

Directory and file information

Python makes it easy to work with both directories and files. You can list directories, change the current working directory, and enumerate the files within a directory. To accomplish this, the os module provides a set of methods that help you out. Let's look at a very simple example of this that allows you to list the directories at a given level, and then allows the user to select one of those directories and prints out the files within that directory.

```
# Get all the directories at the current level and print them out
files = [f for f in os.listdir(os.getcwd()) if not os.path.isfile(f) and f[0] != '.']
for idx, file in enumerate(files):
   print(idx, ':', file)
# Get the directory selection from the user
selection = int(input('Select a directory by number: '))
print("Looking at files in {0}".format(files[selection]))

# Print out the files in that directory
full_dir = os.getcwd() + '/' + files[selection]
files_1 = [f for f in os.listdir(full_dir)]
for file in files_1:
   print(file)
```

Depending on where you run this snippet of code from, you'll see a different output each time, so there is no point telling you what you should see. Note that the `listdir()` method does not return things in any particular order, so you might see a different list each time. You could, of course, sort the results and always give them back in the same order. Note the purpose of the `isfile()` method of the `os` module to screen out things that don't appear to be files in the first look. This is so that we can only list the files in the directory. In our second loop, we don't use it so that we can print out everything in the directory for the user. Finally, note that we screen out anything that begins with a '.' since Linux and Unix treat these files specially and don't show them.

There is also a special method called `os.walk()` which will `walk` the directory tree from a given level and recursively return all files and directories within that tree.

Listing installed packages

If you have ever used pip to install packages, you will notice that it is capable (if the package contains the proper metadata) of installing all the requirements for a

package. For example, if you install flask authorization, it *knows* that you need to have flask installed. Further, it will check if you have the package flask installed on your system before trying to install it. Have you ever wondered how that is possible? The answer lies in the `pkgutil` module of Python.

The `pkgutil` module does a lot of the job that the `pip` command does. You can run, for example, `pip freeze` to find out all the packages and their versions that are installed in a given environment, or virtual environment. We can use the `pkgutil` package in our code to accomplish most of the same thing. For example, let's look at all the installed packages on our system. Note that if you are using a virtual environment for your project, it will pick up the system packages, as along with all virtual environment packages that are installed. The code to do this isn't complicated, but we can take a look at what it looks like:

```
import pkgutil


for mi in pkgutil.iter_modules():
  if mi.ispkg:
    print(mi.name)
```

The `pkgutil` class has a method called `iter_modules`, which returns all installed modules for the system at the point at which it is called. The return from the method call is a list of module information objects. One of the pieces of the module information object is whether or not the item is a package. We query that attribute and, if it is set to `True` we print out the name of the package. Your mileage will vary tremendously for this one, but when this was run against our example project, this is a snippet of what showed up:

```
cards
pkg1
pkg2
util
```

This is wonderful until you realize that any good-sized set of Python projects on your system is going to have potentially hundreds of packages loaded. What if we just want to know about our own packages for one project? Well, there's an answer in the `pkgutil` package for this too. Try this:

```
import os
pkgs = pkgutil.walk_packages([os.getcwd()])
for pkg in pkgs:
   if pkg.ispkg:
     print(pkg)
```

Running this within our little project directory will result in a display that looks something like this:

```
ModuleInfo(module_finder=FileFinder('.'), name='cards', ispkg=True)
ModuleInfo(module_finder=FileFinder('.'), name='pkg1', ispkg=True)
ModuleInfo(module_finder=FileFinder('.'), name='pkg2', ispkg=True)
ModuleInfo(module_finder=FileFinder('.'), name='util', ispkg=True)
```

Your display will contain the path to the modules in the parentheses following the `FileFinder()` part of the output. You can see that it shows us the various packages that we have defined in the book to this point. We could then look at each one if we wanted to, to get more information about them. How do we get information about specific files or classes? The answer lies in the Python reflection system, which we will talk about next.

Reflection

Suppose you have a class in Python. This class contains things like attributes (members), methods, and perhaps even a base class that it implements. Let's start with something simple, like this one that models a very simple person:

```
class Person:

   def __init__(self):
   self.name = ""
 self.age = -1

   def set_age(self, age):
     self.age = age
   return self

def get_age(self):
   return self.age

   def set_name(self, n):
     self.name = n
 return self

def get_name(self):
  return name
```

The above code contains nothing complex, just an example class that we can use to examine through our code. Suppose you want to write something that will display the data for your classes in a command line program. You could feed it a given Python file and it would spit out a definition of what it found in the file. You could either use that as documentation or you could use it to see what signatureshave changed in a given Python file over time. This can be very useful for testing purposes since knowing what has changed lets you start a chain of tests that you have to run. If method A of class B has changed, the only things you need to worry about testing are the things that rely on method A. This could cut way back on your testing time, which we know is a pretty big part of the release process in the real world.

In order to get the information about the class, we need to use all the things that we've learned so far in this chapter. We need to dynamically import the module (file), load the classes that are in the module, and finally, examine each one of the classes. The examination will consist of determining the methods that the class implements, as well as the arguments to those methods. It will also detect all the class attributes that are defined. Let's take a look at the code and its output, and then we'll work our way through the thing and understand it.

```
import inspect

def load_module(name):

    # Load the module file
    module = __import__(name)

    return module

def get_classes(module):
   class_names = []
  for name in dir(module):
    obj = getattr(module, name)
if inspect.isclass(obj):
  class_names.append(name)
return class_names

def describe_func(obj):
sig = inspect.signature(obj)
print("      Arguments: ")
```

```
for s in sig.parameters:
 print('         {0}'.format(s))

def describe_classes(file_name):
   module = file_name.replace('.py', '').replace('/','.')
  mod = load_module(module)
 entries = get_classes(mod)

# Go through them one at a time.
for e in entries:
   obj = getattr(mod, e)
 if inspect.isclass(obj):
  print("Class: {0}".format(e))
try:
 ins = obj()
   methods = inspect.getmembers(ins, predicate=inspect.ismethod)
  for mn, mv in methods:
   print("   Method: {0}".format(mn))
  describe_func(mv)
  print("   Attributes:")
   for name in ins.__dict__:
   print('   ' + name)
except Exception(e):
 print(e)

describe_classes('example_class.py')
```

If we run this program, using the example class we have defined above in Person, we get the following output on our console:

```
Class: Person
   Method: __init__
  Arguments:
Method: get_age
  Arguments:
```

```
Method: get_name
  Arguments:
Method: set_age
  Arguments:
 age
 Method: set_name
  Arguments:
      n
Attributes:
 name
  Age
```

This is all wonderful, of course, but you want to know how and why it works, which is normal for a programmer. We don't care so much about what something does as much as how it does it. The engineering mind is a lovely thing for a programmer. So let's take a look at it. Here are the steps to follow:

1. First, we need to load the module, which we do with the same code that we have looked at previously:

   ```
   module = file_name.replace('.py', '').replace('/','.')
   mod = load_module(module)
   ```

   Nothing new here, we just make sure that the module name doesn't contain the `.py` extension and that any slashes in the directory name are converted to dots so that they conform to the Python module naming conventions.

2. Next, we call the `get_classes` method, which extracts the names of each class within the module. The code looks like this and really gets to the heart of how inspection and reflection work within the Python environment:

   ```
   def get_classes(module):
      class_names = []
     for name in dir(module):
       obj = getattr(module, name)
   if inspect.isclass(obj):
     class_names.append(name)
   return class_names
   ```

   Note that we use the `dir()` method to grab all the symbols within the module. The `dir()` method is moderately faster than some of the other alternatives

and is supported in every version of Python that has been released. The line which calls `getattr` is obtaining an object from the name in the returned list. We can only interrogate objects, not strings. We call the inspect `isclass` method to determine if this is, in fact, a class, and if so we return it in a list of strings to the caller.

3. The next block of code goes through that list back in the main calling function:

```
ns = obj()
methods = inspect.getmembers(ins, predicate=inspect.ismethod)
for mn, mv in methods:
    print("   Method: {0}".format(mn))
    describe_func(mv)
    print("   Attributes:")
    for name in ins.__dict__:
        print('   ' + name)
```

   We extract the name, which is easy enough, and then retrieve each class attribute by interrogating its `__dict__` member after instantiating a copy of the class. This method of finding the members accomplishes two things. Instantiating the object not only creates memory for it, but it also calls the `__init__` method. This adds all the attributes to the object that we care about. We also get any class level variables that were defined when the object itself was instantiated.

4. Finally, we *describe* each method by calling our own `describe_func` function:

```
def describe_func(obj):
   sig = inspect.signature(obj)
  print("      Arguments: ")
 for s in sig.parameters:
  print('         {0}'.format(s))
```

The list of parameters sent to a function is called its *signature* and we retrieve it using the signature method of the inspect class with the object we created earlier. The signature method returns a `SignatureInfo` object that contains, among other things, the list of parameters, which we print out.

Using Reflection

Now that you know what reflection is, and how it works, wouldn't it be nice to understand how it can be used in real-world programs? Clearly, it is lovely to load a

class and look at its attributes, but what if you wanted to call some of those methods and retrieve some of those attributes in your work applications? Since this is what the interpreter is doing under the covers, it should be clear that this is not only possible but quite realistic as well. Here's how you go about retrieving an attribute from a dynamic class in Python.

First, let's modify our `describe_classes` method so that it returns a list of the classes and instances of those classes to the calling program:

```
def describe_classes(file_name):
   module = file_name.replace('.py', '').replace('/','.')
  mod = load_module(module)
 entries = get_classes(mod)


obj_dict = {}


# Go through them one at a time.
for e in entries:
  obj = getattr(mod, e)
if inspect.isclass(obj):
 print("Class: {0}".format(e))
try:
 ins = obj()
 methods = inspect.getmembers(ins, predicate=inspect.ismethod)
  for mn, mv in methods:
    print("   Method: {0}".format(mn))
  describe_func(mv)
  print("   Attributes:")
   for name in ins.__dict__:
     print('   ' + name)


except Exception(e):
 print(e)
obj_dict[e] = ins
return obj_dict
```

At this point, when the `describe_classes` methods return, we have a dictionary that contains the names of the classes along with the instances of those classes that we can use. Let's imagine we want to retrieve the age of the person in the instance. As you may remember, we initialized the age to be –1 in the `__init__` method, so that's what we would expect to see. Let's add some code to our main program to retrieve it.

```
d = describe_classes('example_class.py')
# Get the Person object out of the dictionary
person = d['Person']


# Invoke the get_age method of the object
get_age = getattr(person, "get_age")
age = get_age()
print(age)
```

There are a number of interesting things here. First of all, of course, we get the object of the `Person` class that was instantiated in the `describe_classes` function out of the dictionary by its class name. Now, however, we want to retrieve a *function pointer* (if you are of the C or C++ bent) within that object for the `get_age` method. This is done, as has been done repeatedly, through the `getattr` method, which is the generic way of retrieving information about an object in Python. The return from this call is a function, which we can call. Notice that we don't have to pass the `self` part since it was built into the returned pointed by the `getattr` call. We invoke the method and print out the age. As expected, we see the following output:

```
-1
```

So, the function call works and all is well with the world. However, it isn't really useful to only invoke methods that take no parameters. What if we want to first set the age in the object and then retrieve it? This is more like a real-world scenario. We can add the following code to our little driver program to accomplish this task:

```
set_age = getattr(person, "set_age")
set_age(29)
age = get_age()
print(age)
The output now is:
-1
29
```

As you can see, both the set and get methods are properly invoked and the object is properly modified and interrogated in our code. This, again, is pretty simple. A single parameter is easy to pass. Even multiple parameters are easy to pass:

```
def a_function_with_parameters(a,b,c,d):
    x = a+b+c
   y = b+c+d
  z = c+d+a
  print(x,y,z)

a_function_with_parameters(1,2,3,4)

import sys
func_ptr = getattr(sys.modules[__name__], 'a_function_with_parameters')
func_ptr(1,2,3,4)
```

Now, here's an interesting question. Suppose you have the function defined above as `a_function_with_parameters`. Remember that Python allows us to call functions with parameters that are named, rather than in the order they are defined. So, we could call this method:

```
a_function_with_parameters(a=1, b=2, c=3, d=4)
```

You can see where this is going. Suppose we have a function (or method, the process is the same) that takes a set of named parameters and we want to invoke it by name instead of position. This happens quite often when you have a user-defined function and a list of parameters it can take, but you aren't quite sure what order the user defined them in. So long as the parameter name matches, we can make this work.

The first thing to understand is that a list of named parameters can be represented by, but not used as, a dictionary. We can think of the parameters above as:

```
dict_params = {
    'a': 1,
   'b': 2,
  'c': 3,
  'd': 4
}
```

If we try to just call the function with the dictionary of parameters, we will get errors:

```
a_function_with_parameters(dict_params)
```

```
Traceback (most recent call last):
  File "dump_class.py", line 88, in <module>
    func_ptr(dict_params)
TypeError: a_function_with_parameters() missing 3 required positional
arguments: 'b', 'c', and 'd'
```

This happens because the function is defined to take four parameters, not one. Can we turn the dictionary into a list of parameters to pass? Yes, we can. Remember, a while ago we discussed the splat and double-splat operators. We discussed how the double splat operator flattened out a dictionary into a list of elements that could be printed. Well, that's not the only use for the double-splat, here's another one:

```
func_ptr(**dict_params)
```

This will output the exact same thing as if we passed each one of the elements individually on the function call:

```
6 9 8
```

By now you should have a good handle on reflection and inspection in Python. By this point in the book and your professional experience with the language, you should feel comfortable writing basic Python code and understand enough of the more complicated stuff to be able to read it. As we move forward, we'll be looking more into what makes Python such a valuable language to professional developers. We'll also look at the third-party packages, the tools, and the tricks and tips that make it great.

Conclusion

In this chapter, we learned about how packages and modules work in Python. We explored the use of importing and loading of packages and the difference between loading an entire package and just a function from it. We looked at classes and modules in packages and how to use reflection to obtain information about those bits of code.

In our next chapter, we'll be looking at a variety of things that don't fit well under a single topic but are useful nonetheless.

Questions

1. What is the difference between a system package and a user-defined package?
2. How do you import just a function from a package?
3. How do we list the files in a directory in Python?
4. What is reflection and what is it good for?

Chapter 9
Miscellaneous

Introduction

There are lots of bits and pieces that don't fit into a specific niche for any language, Python is no exception. In this chapter, we'll look at the pieces that may or may not have been touched upon previously or are things that you would expect a professional programming language to contain. We'll examine things like decorators and variable arguments, metaclasses, and namespaces. Unlike most of the chapters, this one probably doesn't have an overriding theme to hold it together, feel free to leaf through the pieces you find interesting or need to know about at any given time.

Structure

- Decorators
- Character encoding
- Variable arguments
- Keyword arguments
- Properties
- Description strings
- Namespaces
- Context managers

- Metaclasses
- Dynamic classes and functions
- Shallow vs deep copying
- Exception handling

Objectives

By the end of this chapter, you should have learned about a slew of different pieces of Python, and you should be able to understand much more complex code than before. You will be able to work with different types of arguments to methods and functions, implement properties, and work with metaclasses and namespaces.

Decorators

The first section we'll talk about is the Python decorator. Decorators are a form of *metaprogramming*, which is extending existing programming through external devices. Basically, a decorator is a wrapper around a function that provides additional functionality to that function. It maymodify the behavior of the function, such as changing the return type, or it may simply add new abilitiesto the function that the original writer never considered.

Decorators are another form of function pointers. They accept a function as an argument, then either *wrap* the function and return a new function, or modify the inputs or outputs to the function. A decorator cannot physically change the code of a function unless it delves into the minutia of dynamic coding. This is one area we will not discuss in this book, as it is a form of hacking and generally is up to no good. Let's take a look at a very simple decorator.

Imagine that you have a function that prints out someone's name, which is passed to it as an argument. There's no particular reason for doing this, it is simply what the function does. Now, you realize that you want the function to greet the person by name. You could modify the function. Let's say it looks like this:

```
def print_name():
   print("matt")
```

We could change it to read:

```
def print_name():
   print("Hello matt")
```

Of course, we could have dozens of such functions, and we'd have to go in and modify each one. Yes, this is a contrived example; you would never do such a thing. You'd refactor all of them down to a single function and modify it. Enter the decorator function. Let's take a look at how it is used, and then we'll see how it works:

```
def hello(f):
   def wrapper():
     print("Hello ",end='' )
   f()
   return wrapper

@hello
def print_name():
   print("matt")

print_name()
```

The output from this snippet is:

```
Hello matt
```

Clearly, it does what we wanted it to do. But the real question is how is it accomplishing this task? Magic is a very poor reason for code to work.

First of all, we have our regular function, `print_name`. All it is doing is outputting a name to the console. Next, we have our decorator function, called `hello`. It accepts a single argument, which is the functionthat needs to be wrapped. Note that it has a function defined inside of it. Python permits you to create functions at any scope, so we can create them within functions, class methods, or anywhere else. They exist from a scoping point of view within that defined space. Our subfunction, called `wrapper` takes no arguments because it inherits the data from its parent, `hello()`. As a result, we print out a string `Hello` without a new line, and then call the function that was passed to us. So, the final output is `Hello` and the name in `print_name`. Finally, we return the wrapper function pointer from the hello decorator. This is what makes the *magic* happen.

Implementing the decorator is only half the battle, after implementing it we have to use it. Notice the use of the `@hello` above the `print_name` method. This syntax is Python shorthand to say that this function is wrapped by the decorator function called `hello`. When the `print_name` function is called, Python takes the following steps:

1. It locates the function in the cache.
2. It observes that the function is wrapped by a decorator (how this happens is internal).
3. It instantiates a wrapper object and passes the hello function to it.
4. It runs the function in the wrapper object, which creates the subfunction called `wrapper`.

5. It retrieves the wrapper function as the output of the `hello` object.
6. It invokes the wrapper function, which prints out the string.

Admittedly, from this viewpoint, it doesn't look very useful. For one thing, we are always printing out the same name and only prepending the same string to it. As we will see, however, decorators are much more powerful than this.

First of all, let's address the biggest issue in the above code. The `print_name` function shouldn't be printing a hard-coded name; it ought to be printing a variable passed into it. In the following snippet we change the function so that it does just that:

```
@hello
def print_name(name):
   print(name)
```

Now, if we run the snippet again, we would expect it to greet the name we pass in, right? Not quite, as you'll see in this example:

```
print_name('fred')
Traceback (most recent call last):
  File "decorator.py", line 72, in <module>
    print_name('fred')
TypeError: wrapper() takes 0 positional arguments but 1 was given
```

You might be wondering why the interpreter is complaining about the `wrapper` function, and why it is not printing out the name we requested. The answer lies in the function that Python is trying torun, which is `wrapper`. This is the function that gets returned by the decorator and the one that will end up calling the `print_name` function in the long run. So, we need to somehow pass through the name of the person we want to greet.

We could fix t1e problem by simply doing this:

```
def hello(f):
   def wrapper(name):
     print("Hello ",end='' )
f(name)
return wrapper

@hello
def print_name(name):
print(name)

print_name('fred')
```

This snippet results in the expected output with no errors. Sometimes, this is the right way to do things, to simply insert the proper parameters to the `wrapper` function and pass them along to the wrapped functions. The problem here is that decorators are meant to be more generic than that. We expect them to work for any sort of function that we wrap up and call with the right parameters, without us having to know much about the underlying implementation of the function that is decorated.

We will look at this idea in a few moments when we talk about variable arguments. For now, though, let's tackle a different use of the decorator. Suppose you wanted to validate that a given argument to a method is of a specific type. For example, if we want to square a value, it would be nice to know that the value we are squaring is a valid integer type. We could do this in the function, but with decorators, there is a much easier and generic approach.

Consider the code in the following snippet. The function `print_int` expects a single argument, an integer value, which it is going to print out to the console. This is obviously not a useful function, but it illustrates the core of what we are trying to do without cluttering it up with a lot of extraneous code that doesn't show the decorator function.

```
def check(typ):
   def checker(f):
  def checked_func(arg):
   if isinstance(arg, typ):
            return f(arg)
     else:
     raise TypeError
return checked_func
return checker


@check(int)
def print_int(val):
   print(val)
```

Notice that in this case, we first accept a function to the `checker` function. Because our decorator itself must accept an argument, which is the type of the argument to check, we must then have a function inside of it which accepts the argument that we are looking to check. This function, the `checker` function, will return the `final` function to be called by the Python interpreter. Within the `checker` function, we will then write another function to check the argument that is passed to the function. This function, called `checked_func`, simply validates that the argument is of the

type that was requested by the developer who decorated the function in the first place. If it is, the original function (`print_int`, in this case) is called. If the argument does not match, we raise an exception (we'll talk about later in this chapter) and fail the program.

One of the most interesting uses of the decorator in Python is to emulate the C# decorator. In C#, for example, you can mark a given unit test method as a test by including the `@test` decorator above it:

```
@test
void do_some_test() {
}
```

Python also supports unit tests, as we'll see in the next chapter, with the `unittest` package. However, thatpackage requires that you begin each test name with the prefix `test_`. Wouldn't it be nice to be able tomark tests to be run without having to name them according to some arbitrary standard? Of course, it would. The decorator construct allows us to do this, although in a slightly different fashion. One of the coolest Python features is that all classes and objects are mutable. That means we can add or subtract attributes from those classes at run-time, something that is not conceivable in a compiled language.

Suppose we create a decorator that marks a given method or function, with a test attribute that indicates to the test runner that it should be run in a given test environment. Let's see what that decorator would look like, first, and then examine a very trivial test runner that will decide whether or not to run a given function based on the test decorator.

First, the decorator:

```
def test():
   def decorator(func):
     func.is_test = True
   return func
return decorator
```

As you can see, the decorator accepts a function, as all decorators do. This one, however, sets an attribute on the function. Since functions are just objects in Python, like everything else, they can have their own attributes. As a result, we can then test for that attribute using the Python `hasattr` function, which is a sibling of the `getattr` and `setattr` functions. If you are uncomfortable setting an attribute that doesn't already exist due to your previous exposure to compiled languages like C++ or Java, feel free to use the `setattr` function instead:

```
setattr(func, 'is_test', True)
```

This line does the same thing but looks like Pythonish and more compiled language-y.

Our next step is to figure out whether a given function is, or is not, a test based on the attribute we have set. To do this, we'll implement a simple function that tests the attribute presence and indicates to the caller whether or not it is there. This is better than directly reading the attribute from the function object. Also, it is more Pythonic since it allows for a different implementation to be used in setting the test attribute and testing for it later on. This is an excellent example of data hiding in OOP:

```
def is_test(test):
   if hasattr(test, "is_test"):
     return True
return False
```

Let's implement two functions. One will be a test and be marked as such with the test decorator. The other will be a *normal* function and will not be a test:

```
@test()
def test_all_the_things():
   print("This is a test")


def not_a_test_function():
   print("This is not a test function")
```

Using our `is_test` method, we can see whether the decorator works properly in the positive and negative sense. This is a pretty ideal test for a `test` decorator, isn't it?

```
print("test_all_the_things is a test = {0}".format(is_test(test_all_the_
things)) )
print("not_a_test_function is a test = {0}".format(is_test(not_a_test_
function)) )
```

The output is as expected:

```
test_all_the_things is a test = True
not_a_test_function is a test = False
```

If we were to combine the reflection with the directory functions of the os module (both looked at in the previous chapter), we could implement ourselves a pretty decent unit testing module. There would be only one more thing we would need: a way to time the tests so that we could report on them. Guess what? We can use a decorator to implement a timer!

```
import time
```

```
def timer(func):
   def wrapper():
      t1 = time.time()
    res = func()
  t2 = time.time()
 wrapper.elapsed_time = t2-t1
 return res
return wrapper


@timer
def a_long_function():
time.sleep(2)
return 3
```

Examining this code, we see that the timer decorator accepts a function, as usual. It also implements a wrapper which sets up a timer by checking the initial time and following it up with the final time after calling the function for which it wraps. Notice that it captures the result of the function and returns that result to the caller, just as if the decorator were never there in the first place. The idea behind decorators is to make their use as transparent as possible.

The function, when run, returns the value `3`. So, if we call this thing:

```
f = a_long_function
print(f())
print( f.elapsed_time )
```

```
3
2.00405216217041
```

we will see that it calls the function and prints out the result, then prints the time that it took to run the function. Notice that since we are just implementing a `sleep` in the function before returning the value, we will mostly see the two second sleep in our timing. This is done to make it apparent that a value was returned.

You might be wondering why we assign a value to the function object rather than just calling it directly. This is so that we can extract the `elapsed_time` attribute from the object. If we had chosen to return the elapsed time along with the return value of the function in the return from the `wrapper()` function, we could just print them all out at once. Since we can't do this:

```
print(a_long_function().elapsed_time)
```

because the result of the function is an integer, and the integer does not have an attribute `elapsed_time` associated with it, the result will be an error from the interpreter.

Decorators are very useful and powerful tools in Python, but it is essential to understand that they are not really necessary. You can accomplish the same thing with function wrappers, assigning functions asobjects to other functions. This is a bit of syntactic sugar that makes it easier and nicer to read but isn't something you can't live without.

Variable arguments

In the previous section, we looked at decorators that wrap a function that takes no arguments. The reason for this is we really hadn't looked at how to pass a variable list of arguments to a generic function like a decorator. Let's do that in this section so that we can complete our discussion. As you know, there are two kinds of parameters one can send to a method or function in Python. First, you can pass a list of arguments in order. Conventionally, these are called **arguments** to the method or function. Secondly, you can pass keyword arguments of the form key-value to the function or method. In the first case, the arguments must be passed in order. For example, if we have a method like this:

```
def compute_them(a,b,c):
    return a+b-c
```

and we call it with the following arguments:

```
compute_them(1,2,3)
```

the result of this function call is `0` since `1+2 is 3 – 3 = 0.` On the other hand, we can pass arguments by keyword:

```
compute_them(a=1, c=3, b=2)
```

This is the equivalent of calling the method with `1`,`2`, and `3` as arguments in order.

When we talk about passing arguments to a function or method in a variable list, we have the same two possibilities. Let's say that we want to write a function that accepts some variable number of arguments as integers and adds them to return the sum of the values. We could call it like this:

```
sum_it_up(1,2,3,4)
```

Or we could call it like this:

```
sum_it_up( 1,2,3)
```

The function `sum_it_up` needs to determine how many arguments are passed to it and add each one of them together. The question, of course, is how do we do this? If we passed in a list of integers, we could write something like this:

```
def sum_it_up(array_of_ints):
    total = 0;
    for I in array_of_ints:
        total = total + i
    return total
```

This wouldn't quite fit the way we want to call them but it is close. Remember, however, the splat operator. It takes a list of items and converts them into a flattened bunch of items. It turns out that in Python, not surprisingly, we can use the splat operator in the signature of the function to accomplish the same process. Python treats our variable length list of arguments into an array to send to the function or method, and then it sends it to the function. So, we can re-write the above function as:

```
def sum_it_up(*args):
   total = 0
  for i in args:
    total = total + i
return total
```

There is nothing special about the name `args`, you can use anything you want. What is important is that you specify it with the splat (*) operator in the signature. The other important thing to know is that you can have normal parameters to start the function signature and have a splat argument as the final argument. In this case, you will use the normal arguments from call, followed by all other arguments tucked into the variable part. For example, suppose we wanted to re-write the function so that you passed in the initial total:

```
def sum_it_up(total, *iargs):
   for i in iargs:
     total = total + i
return total
```

Calling the function this way:

```
print(sum_it_up(1,2,3,4,5,6))
```

Does it surprise you that whether or not we use the total argument, we get the same result? It really shouldn't, since we initialized total in this case to zero in the completely variable case and the first argument in the second. However, if we

changed the meaning of the first argument, making it a string to print out, it would definitely change the results.

Now, there is the keyword argument case. Keywords are not implemented with the splat operator but rather with the double splat (**) operator:

```
def sum_up_values(**kwargs)
```

We could now call it with something like `sum_up_values(value1=1, value2=2, ...)`. Note that we would have to modify the code of the function to look at the keyword argument list as well. The keyword arguments are expressed as tuples in the `kwargs.items()` return:

```
def sum_it_up(total, *iargs, **kwargs):
   for i in iargs:
  total = total + i
for ik in kwargs.items():
  total = total + ik[1]
return total
print(sum_it_up(1,2,3,4,5,6,value1=1, value2=2))
```

This function call prints out `24`, as you would expect. You might ask, what happens if I use the `args` and `kwargs` variables in the function signature, but do not pass any keyword arguments (or arguments)?

```
print(sum_it_up(1,2,3,4,5,6))
```

```
21
```

Likewise, we could call the function with no arguments and only keyword arguments:

```
print(sum_it_up(1,value1=1, value2=2))
```

```
4
```

In this case, the only important thing is that we must pass at least one argument, the total argument because it is neither variable nor default. Given all this, let's go back to our decorator that acts as a timer for function calls. We can now modify it to accept both variable and keyword arguments:

```
import time

def timer(func):
   def wrapper(*arg, **kw):
     t1 = time.time()
   res = func(*arg, **kw)
```

```
 t2 = time.time()
wrapper.elapsed_time = t2-t1
return res
return wrapper
```

So, now, we can use the `timer` decorator with functions that take arguments:

```
@timer

def a_long_function(value):
   time.sleep(2)
  return value

f = a_long_function
print(f(25))
print( f.elapsed_time )
```

The output for the above code snippet will be:

```
25
2.004503011703491
```

As you can see, we have combined decorators with variable argument lists and timing functions to create something really useful. This should be your approach when programming in Python professionally, use what is there, use what you have built, use what you have learned, and create something beautiful and new out of it.

Character encoding

One of the more frustrating things in modern programming is having files encoded in various character sets. In the *good old days*, a file was either ASCII or binary, with no in between. Now, with JSON files containing encoded texts or files written by foreign software that uses a different character set, we must be more aware of the encoding of characters in our files and user interfaces.

As a professional, you are aware that there are encodings beyond that of simple ASCII text. In Python, the default encoding is UTF-8. This is essentially extended ASCII, allowing you to gather characters beyond the 256 byte limit imposed by that standard. ASCII, as you may or may not remember, only defines a handful of characters beyond the upper and lower case letters; and the numeric and shifted numeric values. Oddly, the remainder of the set is generally unprintable characters, such as the newline and linefeed characters, the bell, tab, and others that were useful

back in the days of teletypes. We don't use these much anymore but they are all still supported.

UTF-8 is primarily English and other Roman based languages. UTF-16, which is quickly becoming the default for the more global Internet, is a double-byte character set, allowing it to display and process other languages such as the Asian and Russian Cyrillic languages. The Python string class properly handles both UTF-8 and UTF-16, and with the `encode()` and `decode()` functions of the class, it can handle virtually any character set defined on your computer. To programmers, this is mostly hidden from us and rightfully so. We don't honestly care, except to be aware of the character sets and that they may bepresent. If your particular application needs them, you'll find Python more than capable of dealing with encoding and decoding characters.

It is much more likely that you will need the ability to send characters in a different format. For example, when dealing with older software files, you may need to be able to write octal as well as binary. You might need to be able to write hexadecimal or binary, or convert from characters to integers and back again. For this, Python has a rich set of functionality that will allow you to do what you need to do.

Table 9.1 shows the various functions that are useful in Python and what they do:

Function	Purpose
`asci`	Converts an input into an ASCII representation
`Bin`	Converts an integerto a binary representation
`Bytes`	Converts a string in a given encoding to a list of bytes
`Chr`	Converts an integer into a character
`Hex`	Converts an integer to a hexadecimal value
`Int`	Converts a string to an integer
`Oct`	Converts an integer to an octal representation
`Ord`	Converts a character to its integer equivalent
`Str`	Converts a number or other object to a string

Table 9.1: *The Python encoding conversion functions.*

To give you an idea of how it all works here's a very simple snippet of code:

```
s="This is a test"
print(ascii(s))
print(bin(5))
print(bytes(s, 'utf-8'))
print(chr(95))
print(hex(32))
```

```
print(int('1234'))
print(oct(12))
print(ord(s[0]))
print(str(1234))
```

And here is the output of the said snippet:

```
'This is a test'
0b101
b'This is a test'

_
0x20
1234
0o14
84
1234
```

When you need the above functionality, use it, but most of the time you will find that Python just does what you need it to do in its default behavior.

Properties

Properties are simply attributes of classes that have a simplified structure. Developers do not like to call set and get methods when they can simply assign variables to values. The issue is that we need to make sure those values are valid. Properties are how we protect our data.

Throughout the book, we have discussed Python attributes for classes and have written code like this to deal with them:

```
class OldP:
   def __init__(self):
     self.x = 0

def get_x(self):
  return self.x

   def set_x(self, v):
     self.x = v
```

We do this to add data encapsulation to our classes. By using getter and setter methods, we can enforce the ability to screen out bad values or to change the units of our internal data before returning it to the calling application. This is considerably better than the bad old days when we would write something like:

```
class BadOldP:

def __init__():

  self.x = 0
```

Then the programmer would randomly modify the attribute in code, changing the internal attribute x to be whatever their heart's desires were at that particular moment. This is *bad* because if we change the internal usage of the 'x' variable or the representation to something else, we cause the code to break. Imagineif a new developer comes on board and says, *x is a dumb name, we should use meaningful variables names* and changed all of the x variables to `x_axis_value`. Well-meaning and deep down, we know theywould be right, but it breaks a lot of existing code.

Of course, creating setter and getter methods doesn't solve the problem. After all, if we have a setter that looks like this:

```
def set_x(v):

if v > 0 and v < 100:

        self.x = v
```

It would appear that this setter screens out all values in our class outside the range of `1` to `99`. This worksquite nicely, as long as all developers play by the rules we have established and use the setter and getter methods to retrieve the values. If one developer directly modifies the property in his code and sets the value to `200`, for example, things will break. There has to be a better way, and the developers of Python recognized this, thus with Python 3 theycreated properties.

This might seem to be out of line with the previous parts of this chapter, but it is not, properties are implemented via the decorator approach in Python. For example, we could have a class that implements a property called x and screen out bad values:

```
class P:


   def __init__(self,x):

     self.x = x


   @property

  def x(self):

  return self.__x
```

```
   @x.setter
  def x(self, v):
 if v > 0 and v < 1000:
self.__x = v
```

The property decorator consists of two parts, which is a little clunky but does work. Let's look at the steps necessary to convert this class to use properties:

1. First, we define the *property name* we are going to implement. The name of the property is the name of the method below it, in this case,x. The property decorator makes this into the `getter` for the property and defines the property within the class. The property decorator creates an internal attribute for the class, which is named `__<attribute-name>`. If you want to use the actual attribute, you need to use the __ version of it.
2. Once we have a property, we have a `setter` decorator that is applied to the property name and becomes the setter method that assigns values (or doesn't) to the attribute. The important part of this is that the getter and setter methods are hidden from the user of the class:

   ```
   p = P(5)
   p.x = 10
   print(p.x)
   p.x = 2000
   print(p.x)
   ```

   10

   10

3. One of the nicest things about the property decorator is that we can create read-only attributes if we want to, simply by omitting the setter decorator for a given attribute:

   ```
   class P1:
      def __init__(self):
        self.__x = 0


   @property
   def x(self):
     return self.__x
   ```

4. Now, if you try to set the attribute x in an instance of the class `P1`, you will get an error:

```
p = P1()
print(p.x)
p.x = 1000
print(p.x)
Traceback (most recent call last):
  File "properties.py", line 43, in <module>
    p.x = 1000
AttributeError: can't set attribute
```

Description strings

Python was designed for professional programmers. It was also designed for reuse, and nothing spells reuse better than documentation. For this reason, the Python language contains a built-in process for documenting a class, method or function, called the **docstring**. You can use the docstring in any class or method:

```
class P1:
"""
  The P1 class shows how to implement read-only attributes
   """
  def __init__(self):
    self.__x = 0

  @property
 def x(self):
   return self.__x


   def print_x(self):
 """
     This method prints out the value of x
  It has no arguments
"""
```

```
    print(self.__x)


    p = P1()
    print(p.__doc__)
    print(p.print_x.__doc__)
```

This code displays the following output, as shown in *Figure 9.1:*

```
The P1 class shows how to implement read-only attributes


    This method prints out the value of x
    It has no arguments
```

Figure 9.1: *Output from the docstrings program*

By providing `doc` strings we have built-in documentation that should always be up to date for the current code. For a language like Python, that has a lot of different versions, in which some things work differently, this is a godsend for the developer. You should always document your code, of course, and Python makes it easy. By the way, the `__doc__` attribute is a string and you can modify it. This may be the best example of things you can do but you shouldn't, in Python:

```
p.__doc__ = p.__doc__ + " and I hate it."
print(p.__doc__)
```

Namespaces

Namespaces in Python are defined as *dictionaries mapping groups of elements to names*, which is a fancy way of saying that they are ways to organize things. Namespaces are needed because of name collisions. For example, imagine that you have two packages, `A` and `B`, and both implement the `foo()` method. If `I` write this code:

```
from A import *
from B import *
foo()
```

Which `foo()` will be called, the version from the `A` package or the `B` package? In Python, the principle for importing is *last in wins*, so the `B` version of the `foo` function will be called. What if you wanted to call the A version? The simple answer is, you can't. If you import all names from a given namespace you overwrite the entries for all previous package names. For this reason, Python provides the limited import statement:

```
from A import func1, func2, func3
from B import foo
```

In this case, we can guarantee that `foo()` is only called from the `B` module. But wait, you scream, what if I need both of them? Let's consider that case. Imagine, we have two modules in our program, `test_module_1`:

```
def foo():
   print("This is test_module_1 foo")
and test_module_2:
def foo():
   print("This is test_module_2 foo")
```

Now, we have a third file in which we need, for whatever reason, to use both the `test_module_1` and `test_module_2` versions of the `foo()` function. If we try this:

```
from test_module_1 import foo
from test_module_2 import foo

foo()
```

the output is:

```
This is test_module_2 foo
```

This is in keeping with the last in wins idea. But what if we want to use both? We can actually do this, simply by taking advantage of the 'as' version of the import to give a symbol a new name:

```
from test_module_1 import foo as f1
from test_module_2 import foo as f2

f1()
f2()
```

```
This is test_module_1 foo
This is test_module_2 foo
```

As you can see, Python provides for most scenarios that you might envision. By using the import statement carefully and managing your namespaces carefully, you will find that you rarely run into problems.

Context managers

In Python, a context manager is any class that implements the `__enter__` and `__exit__` methods. It is called a **context manager** because it controls the context of

the data within it. You've seen context managers before in the **with** statement for opening files:

```
with open('file.txt', 'r') as file:
    do_something_with_the_file(file)
```

This is the equivalent of writing:

```
try:
   file = open('file.txt', r')
   do_something_with_the_file(file)
except:
finally:
    close(file)
```

The idea here is to make it as simple as possible to write code that might have to handle exceptions and to deal with standard exceptions in a standard way, cleaning up after yourself as you go. This is the Pythonic mantra; always leave the system in a good state.

You can write your own context managers, of course, and you should for any resource that needs to be cleaned up after you are done with it regardless of whether there is a problem or not. However, you can use context managers in another way. Suppose you want to make sure that a set of tags is always closed in an XML document. You can use the context manager methods for this:

```
class XmlTag:

   def __init__(self, tag):
     self.tag = tag

   def __enter__(self):
     print("<%s>" % self.tag)
   return self

   def __exit__(self, exc_type, exc_val, exc_tb):
     print("</%s>" % self.tag)
   return False

with XmlTag('head') as head:
   print("This is the body of the xml head")
```

The `__enter__` method of our slightly absurd class prints out the opening tag for the XML block we want to do. The `__exit__` method prints out the closing tag. In between, we can do whatever we want such as printing out the body. Note that the `__enter__` method should return the object that is being used.On the other hand, the `__exit__` method should return True if the interpreter should handle whatever exception occurs and False if it should not. If we return True from the `__exit__` method the exception will be bubbled up to the caller, otherwise, it will be handled and eaten in our method.

Metaclasses

In today's world, meta means *deep*, although that's barely where the word came from. In Python,meta means things that have a deep understanding of themselves and the Python environment. Metaclasses are those that help to build and initialize other classes. For example, when you create a new class and instantiate it, do you know that deep down; you are invoking methods of the object class? It is true. What is important to understand is that in Python, everything you encounter, whether it is a class instance, a variable, or a function, is an object. Underlying the object is the type, which means something in Python.

At the heart of the object class are metaclasses. The base metaclass is type. That isn't a typo, it is actually called **type**. The type metaclass is responsible for creating and initializing objects. It does this through two main methods, `__new__` and `__call__`. When you instantiate a new class, the base class `__call__` method is invoked. If you don't define a base class, object is automatically used and the typeversion of `__call__` is called. The `__call__` method calls two other methods within thebase class, the derived class or the metaclass. The first is `__new__` which does the work of constructing the type in memory. Once the `__new__` method has been called, the __init__ method is called, as we have seen over and over in the book.

We already know how to override `__init__` and add new attributes and such to the class. What we haven't looked at is overriding the `__new__` and `__call__` methods. We can override any of these methods in a metaclass, what we need to understand is what to override and when. Let's imagine, for a moment, that we want to add a new attribute to every single object created of our class that contains the date and time the object was created. We could add a new attribute using the following code:

```
import time


class Meta(type):


   def __new__(cls, name, bases, dict):
     dict['created_at'] = time.localtime()
```

```
   obj = super().__new__(cls, name, bases, dict)
 return obj


class MySubClass(metaclass=Meta):


   def __init__(self):
     self.name = "Hello"


for i in range(0,4):
   msc = MySubClass()
  t = msc.created_at
   s = time.strftime("%Y-%m-%d %H:%M:%S", t)
   print("The object was created at {}".format(s))
   time.sleep(2)
```

We place the `sleep` function call in there so that each object is created at slightly different times. The output from this snippet of code is somewhat surprising:

```
The object was created at 2019-08-12 13:20:07
The object was created at 2019-08-12 13:20:07
The object was created at 2019-08-12 13:20:07
The object was created at 2019-08-12 13:20:07
```

Why are all the created times the same? To understand this, we have to look at what is going on here. The `__new__` function accepts a class to instantiate. Anything that is added to the object at this point is a class attribute. If we want to create a new attribute for each instance of the class, we need to override the `__call__` method and be sure to do it after the initialization has taken place:

```
import time

class Meta(type):

   def __call__(self, *args, **kwargs):
     obj = object.__new__(self, args, kwargs)
   obj.created_at = time.localtime()
   obj.__init__()
 return obj
```

If we do it this way, you will see that the output looks like this:

```
The object was created at 2019-08-13 08:47:17
The object was created at 2019-08-13 08:47:19
The object was created at 2019-08-13 08:47:21
The object was created at 2019-08-13 08:47:23
```

This indicates that the created attribute was set for each object, rather than the class. It is worth mentioning that creating your own custom metaclasses is generally frowned upon by the Python community because they introduce unnecessary complexity and maintenance costs in your code. As with all other things Pythonic, it is up to you to decide if the rewards make the risks worth it.

Dynamic classes and functions

If you have worked in other languages, you know that defining a new class can be something of a chore. You have to write all the boilerplate code for each class, including constructors, destructors, assessors and the like. What if you could automate all of that? Well, certainly, some languages do contain utilities that can generate classes for such things as database tables, forms and the like. Of course, you have to generate them, compile them, and then link them into your application to use them. What if you could do that in real time in your application?

Picture this;your user wants to access a new database table. You call one function, and suddenly you have a class that wraps that database table. Sounds fantastic? No, it sounds like Python. After all, the Python interpreter reads lines of text and turns them into objects, so how hard can it be to use that same functionality in your own programs? Turns out it isn't very hard at all.

The core of this functionality lies in the `type()` class. The `type` class, by its nature, returns a new type. It is used by the interpreter to create new classes all the time, and you can use the same functionality in your code. The constructor for the `type` class looks like this:

```
type( class_name, base_class(es), attributes )
```

Where:

- `class_name` is the name of the class you want to create. This, obviously, must conform to Python's naming conventions.
- `base_class(es)` is a list of classes from which to inherit. Normally, one uses object here, but you could use your own classes if you wanted.
- `attributes` is a list of attributes to pass to the initialization of the class. This is as if you were setting them in the `__init__` method of the class.

Let's create a very simple function that will create a new class for us!

```
def create_class(class_name, base_class, **kwargs):
   return type(class_name, (base_class, ),dict(**kwargs))
```

In this case, we are simply making the base class into a list and pushing the arguments into a dictionary, which the type constructor requires. Does it work?

```
cls = create_class('MyClass', object, name = 'my_name', date = time.
localtime() )
obj = cls()
print(obj.name,obj.date)
```

```
my_name time.struct_time(tm_year=2019, tm_mon=8, tm_mday=13, tm_hour=8,
tm_min=47, tm_sec=25, tm_wday=1, tm_yday=225, tm_isdst=1)
```

So far, so good. But is this really our class or just some random object?

```
print(repr(obj))
```

```
<__main__.MyClass object at 0x7fe4c0158e80>
```

Look at that! We have dynamically created a class. You can see where this would be immensely useful for mapping database tables, interfacing to new resources, or just storing data to be serialized into a JSON file in your application.

Deep vs shallow copying

Before we wrap up this chapter, two remaining concepts need to be covered. In the books for beginners, these two might warrant entire chapters of their own, but you've seen all of this stuff before albeit in a different language with a different format. The first of the two subjects is the difference between deep copying and shallow copying.

A deep copymakes a complete copy of an object or data structure, traversing down the sub-elements of the object. A shallow copy makes a new object but keeps the same deep elements within it, as if it were just *pointing* at the original.

Let's look at a simple example to understand what is going on:

```
class AnObject:


   def __init__(self):
   self.list_of_integers = [1,2,3,4]
 self.dictionary_element = {
'a': 1,
'b': 2
}
```

```
    def print_me(self):
      print(self.list_of_integers)
    print(self.dictionary_element)

a = AnObject()
a.list_of_integers.append(5)
a.print_me()
b = a
b.list_of_integers.append(6)
b.print_me()
a.print_me()
```

The output from this little snippet is:

```
[1, 2, 3, 4, 5]
{'a': 1, 'b': 2}
[1, 2, 3, 4, 5, 6]
{'a': 1, 'b': 2}
[1, 2, 3, 4, 5, 6]
{'a': 1, 'b': 2}
```

As you can see, the copy here is shallow. Changes made to the `a` object are also reflected in the `b` object. The Python library contains `a` class that makes this slightly easier, it is called **copy**, which directly makes both shallow and deep copies. Let's look at how this works:

```
print("Shallow copy")
import copy
c = copy.copy(a)
c.list_of_integers.append(7)
c.print_me()
a.print_me()

print("Deep copy")
d = copy.deepcopy(c)
d.list_of_integers.append(8)
```

```
d.print_me()
c.print_me()
```

This snippet produces the output shown in *Figure 9.2:*

```
Shallow copy
[1, 2, 3, 4, 5, 6, 7]
{'a': 1, 'b': 2}
[1, 2, 3, 4, 5, 6, 7]
{'a': 1, 'b': 2}
Deep copy
[1, 2, 3, 4, 5, 6, 7, 8]
{'a': 1, 'b': 2}
[1, 2, 3, 4, 5, 6, 7]
{'a': 1, 'b': 2}
```

Figure 9.2: *Shallow vs deep copy output*

If you have worked in C++ or Java, and implemented copy constructors, this should almost be second nature to you, and you can see how valuable the copy module is. You could easily write your own copy constructor, either by hand or by using the copy module! This feature alone makes Python more valuable in many ways than those so-called *powerful* compiled languages, as it will remove a large chunk of bugs that would otherwise arise in your code.

Exception handling

Virtually every modern programming language has some form of exception handling. Exceptions areexactly what they sound like, an exceptional problem. As with C#, C++, or Java, you shouldn't use an exception for changing the flow of your application, you should only use it when something really bad is happening. Examples might include dividing by zero, running out of disk space or memory, finding a corrupt block of data in your database or the like.

The basic form of exception handling is as follows:

```
try:
<some code>
except <SomeException> as <Some Variable>
except:
<Generic error>
else:
```

```
<when no exception occurs>
finally:
<after all exception handling is processed or no error occurs>
```

Exception handling can be a bit of a surprise sometimes if you are trying to guess what went wrong. Consider, for example, the following code:

```
try:
   x = y/0
except ZeroDivisionError as zde:
   print("You divided by zero!")
  print(zde)
except:
   print("Some other exception")
  print(e)
else:
   print("You didn't create an exception")
finally:
   print("All done now!")
```

Before you try running this snippet, try to guess the output. You might expect to see `You divided by zero!` with the snippet terminating, but you would be wrong. In fact, this code prints out, `Some other exception` followed by `All done` now. The `finally` clause is always executed whether or not an exception occurs in the `try` block. But why do we get the other exception? This is why it is important not to generically `catch` exceptions. We can modify the code as follows:

```
try:
   x = y/0
except ZeroDivisionError as zde:
   print("You divided by zero!")
   print(zde)
except Exception as e:
   print("Some other exception")
  print(e)
else:
   print("You didn't create an exception")
```

```
finally:
   print("All done now!")
```

In this case, you'll see that the output is:

```
Some other exception
name 'y' is not defined
```

All done now!

Oh dear! We forgot to define the y variable before we used it, which actually generated an exception we could `catch`. Unless you honestly don't care why something fails, always check the exception you return.

You can raise your own exceptions with the `raise` statement:

```
try:
   raise Exception("This is an exception")

except Exception as e:
   print(e)
  Exception: This is an exception
```

Finally, you might wonder if you can handle an exception and then pass it on to the caller. For example, consider the case where you try to open a file and it fails. You might want to log this information, then let the caller decide if this is important or not. It may not be a fatal error, perhaps the caller just needs to know if the file exists. This would be a very poor choice for exception handling since we can test to see if the file exists before we try opening it, but it could happen. In this case, we can use the empty `raise` statement:

```
try:
   raise Exception("This is an exception")

except Exception as e:
   print(e)
   raise
```

```
This is an exception
Traceback (most recent call last):
  File "copy_example.py", line 49, in <module>
    raise Exception("This is an exception")
Exception: This is an exception
```

From this point, the exception is *bubbled up* as it is called in most programming languages. Handling exceptions is a basic part of writing solid code, so make sure that anything you call that can throw an exception is wrapped in an exception handling block. Unlike most languages, exceptions in Python are not overly costly, probably because as an interpreted language it is no slower than processing text, so it is a good idea to handle what you can.

Conclusion

This chapter has been something of a grab bag of concepts and constructs that professionals tend to use in coding. Hopefully, now you have a good understanding of the pieces that go into writing professional Python programs and can use them in your own applications. This chapter also finishes the basic language instruction for the book. From this point on, we'll be looking only at things that can help you write more professional programs with minimum fuss.

In the next chapter, we will look at the various third party packages available for Python that will help you to write effective code without reinventing the wheel.

Questions

1. What is a decorator and why would you want to use one?
2. What are the two ways to pass variable arguments in Python?
3. What is a property and how is it implemented in Python?
4. Why should you implement docstrings and how do you view them?

Chapter 10
Not Reinventing the Wheel

Introduction

As you have seen in the preceding chapters, Python is all about making it quick and easy to develop powerful and professional applications that are easy to read maintain. One of the reasons for this is the rich wealth of constructs designed not for the esoteric language lawyer but for the professional programmer who wants to get things done without having to write a ton of code. From list comprehensions to developing classes and instances with minimal code, Python is all about writing code quickly that isn't ugly or obtuse.

Structure

- Itertools
- Flask
- Numpy
- Logging
- Unit test
- Mocking
- Concurrency
- The emoji package

- The pprint package
- The requests package

Objectives

In this chapter, we are going to explore the majority of the *big* packages for Python. These are the packages that are used by virtually all Python programmers out there to make things consistent and to ensure that code works well.

Not reinventing the wheel

You may have guessed that the ability to write concise and powerful code has led to Python being used in a lot of different environments. Of course, the fact that Python is one of the best among the *write once, run anywhere* languages doesn't hurt. Java may make the claim, but Python walks the walk in the portability arena. Most Python packages and extensions are written in Python as well, so it makes it really easy to port them to new environments. By encapsulating the operating system functionality in its own package, Python also makes it trivial to work in different environments. All of this leads to a lot of people writing a lot of good Python code.

The Python community is all about being *open source* and sharing of ideas and code. Much of that code has been put together in specific packages that solve specific needs. In this chapter, we are going to explore some of the most commonly used Python third party packages, how you install them, how you work with them, and what needs they solve for you. By reusing the code that others have put blood, sweat, and tears into, you don't reinvent the wheel!

In this chapter, we are going to explore a variety of packages for Python. By the end of the chapter, you should have a basic understanding of the following areas:

- Itertools
- Flask
- Numpy
- Logging
- Unit testing
- Mocking
- Concurrency
- The emojis package
- Pprint
- Requests for http

Itertools

The first package we are going to look at is the `itertools` package. Unlike many of the packages for Python, itertools does not have a cute name. The `itertools` package works with iterators, extending them in new and useful ways. Itertools is a part of the Python 3 install; you don't need to do anything special to install it.

The `itertools` package contains a wealth of iterators that provide you with easy ways to do things that you would otherwise have to write a bunch of code to do. For example, the package provides the `count()` function, which simply produces an endless series of numbers from a given starting point, with an optional step index. You could do this in our own code by doing something like this:

```
idx = 10

while (True):

if idx > 15:

        break

    Idx = idx + 1
```

The itertools `count()` function makes this simpler:

```
for idx = count(10):

if idx > 15:

break
```

It is important to note that the `count()` method counts the iterations, not the ending value. We can also use the normal Python tools available for iterables, such as enumerate:

```
from itertools import count


for idx, val in enumerate(count(10)):

   if idx > 5:

     break

 print(val)
```

The `itertools` package contains functions you didn't know you needed, until you need them. Great examples of this are the `cycle()` and `repeat()` functions. The cycle function simply cycles through a list or other iterable worth of values, repeating them endlessly. The repeat function does the same thing for a single value, except that you can indicate how many times to repeat this. The `cycle` function can be very useful for testing a set of values sent through a given function:

```
from itertools import cycle
notes = ['do', 'ri', 'mi', 'fa', 'sol']
for idx, n in enumerate(cycle(notes)):
   print(n)
  if idx > 8:
    break
```

This snippet displays the output shown in *Figure 10.1:*

```
do
ri
mi
fa
sol
do
ri
mi
fa
sol
```

Figure 10.1: *Output from enumerate loop*

The repeat function, on the other hand, is reminiscent of the yes command line program in Unix, which simply sends `y` or `yes` to answer prompts in a script.

```
from itertools import repeat
for ans in repeat('yes', 3):
   print(ans)
```

```
yes
yes
yes
```

Let's looks at one more example from `itertools` that should give you an idea of the power that is stored in this package. If you are used to writing SQL code, you've probably done the `GROUPBY` statement a fair amount. Given a list of data, the group by statement will group it together so that they are under a given key. There are a couple of ways we can use the `groupby` function in `itertools`. First, let's look at a very simple example of grouping duplicate values in a list:

```
list = [1,2,1,3,1,4,2,3,4]
from itertools import groupby
for key, group in groupby(sorted(list), lambda x:x):
```

```
   print("Group: " + str(key))
  for element in group:
    print('   ' + str(element))
```

The output from this little snippet is a grouped list of the values in the array, first sorted and then broken down by the repeating values:

```
Group: 1
   1
   1
   1
Group: 2
  2
   2
Group: 3
   3
 3
Group: 4
  4
   4
```

The `groupby` function is much more powerful than this, of course. We can use it on more complex structures, just as we would for a database query. Imagine we have a database of people with the city and state in which they reside. We want to break them down by state, so we can do some numerical analysis on their other data. Imagine that the data structure looks like this:

```
darray = [
    {
    'name': 'matt',
'city': 'new york',
    'state': 'NY'
    },
    {
        'name': 'fred',
        'city': 'albany',
        'state': 'NY'
```

```
    },
    {
        'name': 'irving',
        'city': 'atlanta',
        'state': 'GA'
    },
    {
        'name': 'tony',
        'city': 'duluth',
        'state': 'MN'
    }]
```

Now, we want to break them down, so we'll use the `groupby` function to break them down by states:

```
for key, group in groupby(darray, lambda x : x['state']):
   print("Group: " + str(key))
  for element in group:
  print('   ' + str(element))
```

```
Group: NY
   {'name': 'matt', 'city': 'new york', 'state': 'NY'}
  {'name': 'fred', 'city': 'albany', 'state': 'NY'}
Group: GA
 {'name': 'irving', 'city': 'atlanta', 'state': 'GA'}
Group: MN
 {'name': 'tony', 'city': 'duluth', 'state': 'MN'}
```

Alternatively, we could do some sort of computation on the groups. For example, suppose we just want to know how many people are there in each group:

```
for key, group in groupby(darray, lambda x : x['state']):
   print("Group: {0} Count: {1} ".format(key, len(list(group))))
```

```
Group: NY Count: 2
Group: GA Count: 1
Group: MN Count: 1
```

Note that we have to first convert the result of the group to a list in order to count it, as it is a *groupable* object that has no count mechanism built-in. We could use this for reporting, mass mailing, or whatever you want.

Itertools also contains some absolutely fantastic statistical functions, including combinations, permutations, and Cartesian products from iterables. If you need any sort of work generating sets of data, the `itertools` library should be a part of your conversation to start with.

Flask

In today's world, few things are as important in the software development world as microservices. These small HTTP services make it easy for websites and distributed applications to talk to each other and to handle very specific pieces of a bigger system. In addition, microservices allow easy updates to each piece of a system, without having monolithic systems that cannot be upgraded except by a single deployment. Features can be added to a single microservice and tested in place without exposing them to the rest of the world. In the Python world, the HTTP service handling is done primarily by a package called **Flask**.

The Flask package is used for a number of things, including writing simple HTTP servers and websites, but for this book, we are going to focus on the use of Flask as a microservice library. To understand microservices, you must understand the verbs used in HTTP processing. For a microservice, there are two bits of information that need to be dealt with. First, there is a *verb* that indicates what sort of manipulation you wish the service to accomplish for you. There are four verbs that are understood by all services as shown in *Table 10.1:*

Verb	Meaning
`GET`	Retrieve information from the service.
`POST`	Add new information to the service.
`PUT`	Update new information on the service.
`DELETE`	Remove information from the service.

Table 10.1: *The HTTP verbs*

The Flask package makes it remarkably easy to implement these verbs and to put together a microservice in a few dozen lines of code. You will spend much more time working on the data structures and storage of the information than on worrying about how to get it in and out of the service.

First, you need to install the Flask system. We will use pip to install Flask and its various components. Flask itself is a single package, but it has many additional components and extensions that are used. In our case, we are going to install Flask first:

```
pip install flask
```

This installs the basic package and the decorators that make it up. In addition, you will want to install `flask_httpauth`, which handles basic authentication. Flask itself contains the core of the code for the microservice architecture, but it is aided by utilities like `request`, `jsonify`, and `make_response`, which is used in the following example.

Create a new file to store your microservice. We'll call our one `flask_test.py`, but you can call it anything you like. At the top of your file place the import for the core `flask` package:

```
from flask import Flask
```

Now, add the following lines to your file:

```
app = Flask(__name__)

if __name__ == '__main__':
   app.run(debug=True)
```

Believe it or not, that's all you need to do to have a microservice. It doesn't do anything yet, but we'll get there. First, let's create some data for our microservice to serve up:

```
addresses = [{
   'name': 'matt telles',
  'address': '1313 Mockingbird Lane',
 'city': 'New York',
 'state': 'NY',
 'zip_code': '10012'
}]
```

Our system will, clearly, serve up addresses to any callers. We'll add methods to retrieve the address information from the service, addnew addresses, and update existing addresses. Naturally, to complete the **CRUD (Create, Read, Update, Delete)** operations, we'll add the ability to delete existing address information. Here are the steps involved:

1. Let's begin by adding a method to retrieve the current list of addresses. It will be nothing fancy, just a GET operation that retrieves the entire list. Pagination and such won't be considered in this example, although it is something you'd want in your own applications.

   ```
   from flask import jsonify

   @app.route('/api/v1/addresses', methods=['GET'])
   ```

```
def get_addresses():
   return jsonify(addresses)
@app.route
```

2. This is a part of the `Flask` package that is used to define a route. A route is the base URL (for testing purposes, this is always localhost, on port `5000`), followed by a path within the base URL. In this case, we are using the `/api/v1/addresses path`. The `/api/v1` is a pretty typical way of specifying that this is not a website address. It isn't a rule, per se, but it is a convention that is followed by most application services. Following that piece is the name of the service that we are using, in this case `addresses`, to indicate that this API handles address information. You can obviously call this anything you want, but it is best to be consistent.
3. Following the path, which is what you would enter into a web browser or other method of getting information from the web, you have the `methods=` section. Flask refers to the verbs as methods, so the method we are implementing here is `GET` which means that you will retrieve information. It is standard to use `GET` only for read purposes and not to allow updates.
4. Below the decorator is the actual function definition. This is standard Python; it just defines the actual function we will be implementing in our code to do the GET of information. Note that you can implement `Flask` methods as Python methods or functions, within classes or without them. It is completely up to you. Because a function is just an object in Python, there is no functional difference to Flask.
5. Within our method, we call the `jsonify` function, which is also imported from Flask. This method does JSON serialization of the data it is called with and sends it back to the user. Note that we simply return the data; `Flask` does the remaining task of wrapping it up in an HTTP message with proper headers and such.

The next question, obviously, is how do we test this program? We can simply run it using the python interpreter, and if you run it you should see something like this displayed in your IDE or command window:

```
* Serving Flask app "flask_test" (lazy loading)
* Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
    Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
```

```
* Debugger is active!
* Debugger PIN: 252-695-625
```

This is `Flask` telling us that it is running, that the service is accessible on port `5000` of localhost, and that it is running in debug mode. The debug mode is useful for detecting errors in the code, as it will print in the response to requests and the command line display of your application as well. Once you see this message, you can hit the web service in any way you like.

Our testing tool of choice is `curl`. It is available on virtually any platform, works from the command line, can be put in scripts, and is small and fast. You can use other tools, from Postman to writing your own, to even using a web browser, but for now, we'll focus on the curl commands needed to test the service.

The `curl` command to retrieve data uses the default `GET` verb and is quite simple:

```
curl -i http://localhost:5000/api/v1/addresses
```

We specify the address of the server using the -i option, and that's all that is needed. Curl knows how to do a get, and no other information is needed for this simple case. The response looks something like this:

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 150
Server: Werkzeug/0.15.5 Python/3.7.2
Date: Thu, 15 Aug 2019 13:53:31 GMT

[
{
"address": "1313 Mockingbird Lane",
 "city": "New York",
   "name": "matt telles",
   "state": "NY",
 "zip_code": "10012"
 }
]
```

As you can see, our service is working and serving up the information that we requested. You can call it as many times as you like, and it will serve up that single address over and over. This isn't very interesting, so maybe we should allow the user to add new addresses to the little *database* we are storing things in.

The verb, or method, used to add data to the system is `POST`. Therefore, the decorator has to handle the POST command for adding new data. It looks like this:

```
@app.route('/api/v1/addresses', methods=['POST'])
def save_address():
```

In `Flask`, information is passed on through the request object, which is a kind of global entity to your application. You get to it by importing the request component:

```
from flask import request
```

Let's look at the implementation of the post handler:

```
from flask import make_response
def extract_address(json_data):
   address = {
     'name': json_data['name'],
   'address': json_data['address'],
 'city': json_data['city'],
  'state': json_data['state'],
 'zip_code': json_data['zip_code']
}
return address
def save_address():
   if request.json is None:
     return make_response( jsonify({'error': 'No content'}), 400 )
   if not 'name' in request.json:
     return make_response( jsonify({'error': 'Missing key name'}), 400)
   addresses.append(extract_address(request.json))
  return jsonify({'status': 'ok', 'id': str(len(addresses))})
```

First of all, we have some new imports. The `make_response` component is just a wrapper that allows you to generate an HTTP compatible response from a Python dictionary and an error code. The request component is what we receive from the user. Note that `Flask` does not pass things through to the function, but rather makes that information available through the `Flask` package methods.

The request object contains a variety of information. The raw data is stored in the data attribute. If the data is sent as a JSON piece, the parsed JSON will be stored in the request json attribute. In our case, we expect the data to be in JSON format, so if it is not there (that is, `response.json` has nothing in it and is None), we will

just return an error. Errors consist of two parts, an HTTP error code and a message. Our message, in this case, is just `No content` indicating we found nothing to add. The error code is 400, which is just an HTTP status saying that the request was bad. Likewise, if the JSON content doesn't look right to us (in this case we are just verifying that the name piece is there), we will return an error as well.

We've added a `utility` function, `extract_address` to take the pieces of the address out of the JSON content in the request and return it as an address dictionary object so that we can reuse this functionality later on when we want to update things. Presuming that all the extraction goes well, the data is then added to our database of addresses and a response is sent back to the caller. The response consists of a message `ok` and the id of the new address, which is just the number of entries in the database.

Sending a `POST` message through curl is slightly more complicated. This book is not intended as a tutorial on the curl program, it has its own help messages and there is plenty of information available on the web, thus we'll just present the basic command:

```
curl -i -H 'Content-Type: application/json' -X POST -d '{ "name": "fred mcmurray", "address": "1212 Main Street", "city": "Albany", "state": "AL", "zip_code": "12345"}' http://localhost:5000/api/v1/addresses
```

The important aspects of this command are that we are using the `POST` command (`-xPOST`) and we are passing data (`-d` followed by the JSON command). One note here, if you are working in the Windows environment and using the standard `curl` program that was built for Windows in a Command Prompt, you may have issues sending things with double quotation marks, since Windows has a way of interpreting them that is different from Linux, Unix, or Mac. You can try triple quotes if this happens or choose the Windows PowerShell window, which does things in a more standard way.

Whatever the case, when you send the message to our updated service, you should see the following response:

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 35
Server: Werkzeug/0.15.5 Python/3.7.2
Date: Thu, 15 Aug 2019 14:23:26 GMT

{
  "id": "2",
"status": "ok"
}
```

Once we have the system running, the next step is to add some of the pieces that are needed to make it of professional quality.

Adding authentication

If you are just playing with web services, it is fine to leave them open to the world to get to. After all, if it is running on your local machine, not too many people across the world will have access to the service. When you move your service to a more global space, like in the cloud, however, it is important that you only allow those who should have access to modify your data. For this reason, `Flask` allows you to add authentication to your service quickly and easily. We are just going to use the most basic and not too complex authentication to illustrate how it could be done.

First, you need to add the following to your service module:

```
from flask_httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()
```

This imports the `Flask` basic authentication package from the system (you may need to import it if you have not already) and creates a new authentication module. The authentication module, auth, requires that you do a little bit of setup. Add the following function to your module:

```
users = {
   'matt': 'mattpassword',
   'george': 'georgepassword',
}
@auth.get_password
def get_password(username):
   for user in users:
     return users[username]
return None
```

Notice the use of the decorator `get_password` assigned to the auth object. This tells `Flask` to call this method to retrieve the password for the user and compare it to the one sent in the HTTP headers for authentication. If you do not send a user name and password in your request for authenticated methods, you'll see an error like this in your `curl` command:

```
HTTP/1.0 401 UNAUTHORIZED
Content-Type: text/html; charset=utf-8
Content-Length: 19
```

```
WWW-Authenticate: Basic realm="Authentication Required"
Server: Werkzeug/0.15.5 Python/3.7.2
Date: Thu, 15 Aug 2019 14:22:06 GMT
```

With `curl`, to send the authentication information, just use the –u flag. It looks like `–uuser:password` where the user and password are the information you want to send to Flask. This will be encoded and decoded on the backend side.

Not all Flask route handlers require authentication, nor is it on by default. To use authentication for our `POST` handler, let's tell Flask to require it:

```
@app.route('/api/v1/addresses', methods=['POST'])
@auth.login_required
def save_address():
   if request.json is None:
     return make_response( jsonify({'error': 'No content'}), 400 )
if not 'name' in request.json:
  return make_response( jsonify({'error': 'Missing key name'}), 400)
addresses.append(extract_address(request.json))
return jsonify({'status': 'ok', 'id': str(len(addresses))})
```

You can see that by adding the `auth.login_required` decorator to our method, it automatically tells Flask to require the user to provide login information. Once you have logged in, you'd receive a token back from the service that would be sent in all additional requests. Alternatively, you can just send the user name and password with each request, which is probably easier in our case.

After we add a new entry, we can then request the `GET` again to verify that the new data has been added to the database:

```
curl -i http://localhost:5000/api/v1/addresses
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 293
Server: Werkzeug/0.15.5 Python/3.7.2
Date: Thu, 15 Aug 2019 14:43:57 GMT

[
 {
 "address": "1313 Mockingbird Lane",
```

```
   "city": "New York",
   "name": "matt telles",
   "state": "NY",
  "zip_code": "10012"
},
{
   "address": "1212 Main Street",
   "city": "Albany",
  "name": "fred mcmurray",
  "state": "AL",
   "zip_code": "12345"
 }
]
```

Here we see that the new address was added and is retrieved by the `GET` call. Finally, let's round out our Flask exploration by adding the update and delete functions:

```
@app.route('/api/v1/addresses/<int:address_id>', methods=['PUT'])
def update_address(address_id):
   addresses[address_id-1] = extract_address(request.json)
  return make_response(jsonify({'status': 'ok'}), 200)

@app.route('/api/v1/addresses/<int:address_id>', methods=['DELETE'])
def delete_address(address_id):
   del addresses[address_id-1]
  return make_response(jsonify({'status': 'ok'}), 200)
```

Obviously, if you were writing your own service, you would be storing data in a legitimate database using much more stringent tests for the data coming in, checking for duplicates, bad data and the like. That's outside the scope of this exercise, which was simply to illustrate how quick and easy it is to implement a microservice using Flask. Hopefully, you've seen just that!

Numpy

When Python was first created, in the mid-1980s, the focus of the language was primarily on scientists. These folks needed to do complex analysis on data, primarily mathematical data and didn't want to take the time to learn compilers, IDE's and

languages that required you to devote your life to them. Python was something of a godsend to these scientists, who simply wanted to get their work done and publish their results. What Python really lacked at that time was a good way to handle some of the math that they were accustomed to using. At that time, the only solid math package was MATLAB, which was, and still is, a somewhat arcane system for manipulating mathematics, via matrices and general equations, using a proprietary language and system. For all of its faults, MATLAB was astonishingly powerful, allowing scientists to do an entire day's worth of calculations with a few lines of strange syntax.

The Pythonistas of the day were horrified by MATLAB but envious of its power. A man named *Jim Hugunin*, along with several other developers lending hands and code, created a package called Numeric, which did a lot of matrix work and linear algebra algorithms. This was built off work done by *Jim Fulton* and was published at MIT for use by Python programmers. A bit after Numeric came out, a competing package called **Numarray** was created, which had better performance for large arrays but poorer performance for smaller ones. Finally, in 2005, *Travis Oliphant* came along and merged the two competing code bases into a single package with the advantages of both called Numpy.

The Numpy package is primarily based around n-dimensional arrays, called **ndarrays**. These arrays can be created, manipulated, added, multiplied and the like. Numpy has a full linear algebra algorithm library and can quickly and easily work with arrays of virtually any size. Unlike standard Python lists (which are really arrays under the covers), Numpy arrays are typed, allowing developers to be careful with their data.

Installing Numpy

There are two methods for installing Numpy. Let's look at the possible steps:

- `pip install numpy`: With that said, if you are only going to use `numpy` for its matrix and linear algebra capabilities this is all you have to do. If, however, you want to use the system for scientific programming, as it was intended, you should install the full `scipy` package. This book is not about full scientific programming, so we won't be digging into most of the esoteric of such things, but you can install the full scientific programming pack in Python by using the following command.
- `pip install --user numpy scipy matplotlib ipython jupyter pandas sympy nose`: As you can see by the ordering, `numpy` is the basis for most of the scientific programming packages. The others, `scipy`, `pandas`, and `nose`, for example, are for specific mathematical manipulations. Others, such as `matplotlib`, allow you to do graphic plotting of your data. The scientific community relies heavily on `numpy` and its related packages, and if you are in that industry it behooves you to learn them.

Getting started: The basic array

The `numpy` array class is just a Python list of sorts. The difference is that it *understands* the dimensions of the array and can work with it directly using matrix math. To create a simple one-dimensional array in Numpy, which is the same as a Python list, we can write code like this:

```
import numpy

my_list = [1,2,3,4,5,6]
my_numpy_array = numpy.array(my_list)
print(my_numpy_array)
print(my_numpy_array.shape)
```

In this example, we are creating a simple Python list, called `my_list`. That list is then passed to the Numpy array method, which creates a new Numpy array instance from it. We then print out that array and look at its shape. The shape is a tuple that represents the dimensions of the array. For example, if we run this code, we will see the following output:

```
[1 2 3 4 5 6]
(6,)
```

This indicates that Numpy understands this to be a one-dimensional array with 6 elements in the first dimension. If we wanted to create a two-dimensional array, otherwise known as a simple `matrix`, we could do this:

```
row_1 = [1,2,3]
row_2 = [4,5,6]
row_3 = [7,8,9]

matrix = numpy.array([row_1, row_2, row_3])
print(matrix)
print(matrix.shape)
```

In this case, we have three rows with three columns of data in each. Note that we pass the individual lists as a single array to Numpy. The array method only accepts lists and knows how to *parse* them into rows and columns. The output from this little snippet is as follows:

```
[[1 2 3]
[4 5 6]
[7 8 9]]
(3, 3)
```

In this output, we see that Numpy knows this to have three rows of three columns. The shape of the array, therefore, is a tuple of two elements, the rows and the columns.

You can also create *standard* matrices using Numpy. For example, the zeroes method will create a matrix of a given size, setting all elements to `0`:

```
zero_matrix = numpy.zeros((3,3))
print(zero_matrix)
[[0. 0. 0.]
[0. 0. 0.]
[0. 0. 0.]]
```

Likewise, there are methods called ones, which create a matrix of a given shape filled with ones. Then we have full, which creates a matrix of a given shape filled with a specified value, and random, which creates a matrix with random numbers inserted.

Accessing Numpy Data

The Numpy array can be used much as if it were a two-dimensional list in Python, but with a few differences. For example, we can slice a Numpy array:

```
sub_array = matrix[0:3, 1]
print(sub_array.shape)
print(sub_array)
```

Given the matrix defined above, this code snippet will return the middle column of the data. Slicing is done via the row and column format, so we are telling Numpy to give us back data starting with row zero and ending three rows into the array. Likewise, it returns the middle column, since, like normal lists, Numpy arrays are zero-based indexed.

Numpy supports a range of values, using the `arange()` method:

```
five_columns = numpy.array(numpy.arange(0, 5))
print(five_columns)
```

This snippet produces a single dimension array with 5 elements, with the data stored in those elements the list of the range:

```
[0 1 2 3 4]
```

Data types

The Numpy library is typed, meaning that all the data stored in a given array must be heterogeneous, but also meaning that each element is stored in the specific type the user wishes. The default, of course, is integer, but you can define matrices in floating point math as well:

```
float_matrix = numpy.array([[1.0,2.0,3.0],[4.0,5.0, 6.5],[1.5, 2.5, 3.5]],
dtype=numpy.float)
print(float_matrix.shape)
print(float_matrix)
print(float_matrix.dtype)
(3, 3)
[[1.  2.  3. ]
[4.  5.  6.5]
[1.5 2.5 3.5]]
Float64
```

As you can see, Numpy *knows* what type the data is and can work with it in its native format. This is not to say you can't mix matrices of different types:

```
irow_1 = [1,2,3]
irow_2 = [4,5,6]
irow_3 = [7,8,9]

i_matrix = numpy.array([row_1, row_2, row_3], dtype=numpy.int)

frow_1 = [11.0,12.0,13.0]
frow_2 = [14.0,15.0,16.0]
frow_3 = [17.0,18.0,19.0]
f_matrix = numpy.array([frow_1, frow_2, frow_3], dtype=numpy.float)

print(i_matrix + f_matrix)
```

As you can see from the above, Numpy supports normal addition of matrices and can handle matrices of different types. You can store any type in a matrix, and it will be processed if it is possible. We could, for example, create two matrices of complex values:

```
cmp_1 = [(1+2j),(2+3j),(3+4j)]
cmp_2 = [(4+5j),(5+6j),(6+7j)]
cmplx_1 = numpy.array(cmp_1)
cmplx_2 = numpy.array(cmp_2)
print(cmplx_1+cmplx_2)
```

Needless to say, you can add floating point and integer matrices, or integer and complex matrices, or any combination thereof. It also supports matrix multiplication, both with scalar (non-matrix) values, as well as between two matrices:

```
print(matrix)
print(matrix*2)
[[1 2 3]
[4 5 6]
[7 8 9]]
[[ 2  4  6]
[ 8 10 12]
[14 16 18]]
print(i_matrix*f_matrix)
[[ 11.  24.  39.]
[ 56.  75.  96.]
[119. 144. 171.]]
```

That covers the basics of creating arrays, but once you have one what do you do with them?

Modifying arrays with Numpy

One of the most powerful aspects of the Numpy library is the ability to modify a matrix so that it can be used in other ways. For example, suppose we take the 3x3 matrix that we defined above and 'flatten' it to be a one-dimensional array:

```
print(matrix)
print(matrix.reshape((1,9)))
[[1 2 3]
[4 5 6]
[7 8 9]]
[[1 2 3 4 5 6 7 8 9]]
```

Data within an array can be modified in place if you like:

```
print(matrix)
matrix[1][1] = 99
print(matrix)
[[1 2 3]
[4 5 6]
[7 8 9]]
[[ 1  2  3]
```

```
[ 4 99  6]
[ 7  8  9]]
```

Numpy mathematical functions

Having a matrix is all well and good, but unless you have all the supporting functionality to work with that matrix, it is just a pretty thing rather than a useful one. Numpy has a lot of support for mathematical functions that can be applied directly to a matrix or multiple matrices:

Trigonometric and hyperbolic functions are all supported, such as sin, cos, tan, sinh, `cosh`, and `tanh`. There are also functions to convert degrees to radians and back. If you need to know the functions, the `numpy` documentation page at docs.scipy.org will show you all of these and many more. Unless you are into trigonometric work, it's likely you will never look at these.

More interesting to general programmers are the rounding functions, which permit you to round data to a specified number of decimal places, or to truncate data to the next lowest or highest values. To get an idea of what they look like, let's look at a very simple example. To round data to a fixed number of decimal places, we use the round method of `numpy`:

```
frow_1 = [11.234567,12.12345,13.454354]
frow_2 = [14.234234,15.765468,16.8974598]
frow_3 = [17.123123,18.1123123,19.83645]
f_matrix = numpy.array([frow_1, frow_2, frow_3], dtype=numpy.float)
print(numpy.round(f_matrix, 2))
[[11.23 12.12 13.45]
[14.23 15.77 16.9 ]
[17.12 18.11 19.84]]
```

The floor method, similarly, returns a matrix that consists of all the data points in the matrix rounded down to the next lowest integer value but stored as the floating point value that was initially stored there. If you use the `floor()` method on an integer matrix, it will be converted into a floating point matrix, but the values obviously will not be changed:

```
print(numpy.floor(f_matrix))
print(numpy.floor(matrix))
[[11. 12. 13.]
[14. 15. 16.]
[17. 18. 19.]]
```

```
[[ 1.  2.  3.]
[ 4. 99.  6.]
[ 7.  8.  9.]]
```

Please note that the `floor()` method does not round the values, it truncates them to a lower value. Thus, our value of `15.765468` is simply truncated down to `15`, not rounded up to `16`.

The converse of the `floor()` method is the `ceil()` method, which returns the integer value which is greater than or equal to the value. The `floor()` method works, as does the `ceil()` method, on an element wise process, examining and modifying each element in the matrix across the rows and down the columns. The return, as always is a matrix:

```
frow_1 = [11.234567,12.12345,13.454354]
frow_2 = [14.234234,15.765468,16.8974598]
frow_3 = [17.123123,18.1123123,19.83645]
f_matrix = numpy.array([frow_1, frow_2, frow_3], dtype=numpy.float)
print(numpy.ceil(f_matrix))
[[12. 13. 14.]
[15. 16. 17.]
[18. 19. 20.]]
```

Unsurprisingly, floating point values with anything non-zero after the decimal point will be rounded up to the next integer value, regardless of whether the decimal portion is greater than one half or not. An important point about both the `ceil` and `floor` methods is that both do not modify the existing matrix. They return a new matrix which has the values copied and modified:

```
print(numpy.ceil(f_matrix))
print(f_matrix)
[[12. 13. 14.]
[15. 16. 17.]
[18. 19. 20.]]
[[11.234567  12.12345   13.454354 ]
[14.234234  15.765468  16.8974598]
[17.123123  18.1123123 19.83645  ]]
```

The Numpy package supports matrix comparison in a variety of ways. For example, we can take two matrices and create a third matrix which contains the greatest or least element in each position using the maximum and minimum methods:

```
frow_1 = [11.234567,12.12345,23.454354]
frow_2 = [14.234234,5.765468,16.8974598]
print(numpy.maximum(frow_1, frow_2))
```

```
[14.234234 12.12345  23.454354]
```

```
print(numpy.minimum(frow_1, frow_2))
```

```
[11.234567   5.765468  16.8974598]
```

All the matrix comparison and manipulation functions make certain assumptions about the data they are working with. To add two matrices, for example, they must each have the same number of rows and columns. Numpy doesn't make you provide two matrices in that formation, but it does require that they be convertible into that formation. This conversion process is called **broadcasting** in Numpy parlance. Broadcasting allows Numpy to reshape a given matrix into whatever it needs to be to make an operation work. For example, let's assume we have our matrix from above:

```
[[ 1  2  3]
[ 4 99  6]
[ 7  8  9]]
```

Now, let's say that you want to add to this an integer matrix of a single dimension. In true matrix math, you can't do this; you have to add matrices that have equal dimensions (or shapes, as Numpy calls them). However, Numpy will attempt to make your array into the proper shape to accomplish the task. For example, in our above code, let's say that we give it a new matrix to add that looks like this:

```
matrix_add = [1, 0, 1]
```

In order to add this matrix to our existing matrix, we would need to increase the number of rows in the array to three. In addition, we need to fill those rows with something. Numpy assumes that what you really wanted to do was *stack* that row three times as if it were:

```
[[1,0,1],
 [1,0,1],
 [1,0,1]]
```

So, when we write code like this:

```
matrix_add = [1, 0, 1]
print(matrix+matrix_add)
```

we are really adding a 3x3 matrix to an existing 3x3 matrix. The result is what you would have thought it would be, another 3x3 matrix with all the elements added in the proper places (row and columns):

```
[[ 2  2  4]
[ 5 99  7]
[ 8  8 10]]
```

If you are not happy with the way Numpy chooses to resize a matrix to do the work you want, you can always use the reshape method to put it into the format you are looking at.

Finally, there are Numpy methods for working with specific axes of the matrix. The `sum()` method, for example, can sum up all the values in a given row or column, depending on which axis you choose.

```
print(matrix)
print(matrix.sum(axis=0))
print(matrix.sum(axis=1))
[[ 1  2  3]
[ 4 99  6]
[ 7  8  9]]

[ 12 109  18]

[  6 109  24]
```

You can see from the above code that the axis zero represents the columns, so the `sum()` function called for axis zero adds all the values down the column and places it in that column of the result matrix. It works its way across the rows, whereas using axis one will add all the values across a row and down the columns to produce a result.

There is a great deal more to working with Numpy than what we've discussed in this basic introduction. If you are looking for any sort of mathematical work, from matrices to linear algebra, please don't roll your own. The Numpy library and more all-encompassing SciPy distribution can do it all, and it is not only well designed but also well tested by programmers using it for years.

Logging

If you have spent any time in the professional software development world, you know about logging. From the early days of using some variety of the print statement to debug code or just output information about the application that is running, to full blown web servers and services that log all their configuration and error information to a central data store for customer support, logging is central to the development effort. It should be no surprise then, that Python provides a complete logging module for your use in professional application development.

The logging module is built into Python and can be accessed by importing logging. There are two sorts of logging available, default and custom. The default logger is the one you see when you get an error, such as an exception, in the running of your code. You might see something like this when running a unit test, for example:

```
INFO: Traceback (most recent call last):
    File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/
python2.7/unittest/suite.py", line 70, in __call__
    return self.run(*args, **kwds)
    File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/
python2.7/unittest/suite.py", line 108, in run
```

The first line, with the Traceback information, shows the level of the log statement. In this case, it is `INFO`. You can use debug, info, warning, error, and critical as logging levels in the Python logger. What these levels really *mean* is up to you, they are simply output by the logger. The reasons for the levels are that the logger can be set to only output statements of a certain level. For example, you might turn off debug output when you are running your application in a production environment, or even turn off info and warning statements, focusing only on the more serious errors.

For the default logger, all logging is done to the console. You can capture this in various ways on variousoperating systems by redirecting the standard output to a file, a device, or whatever you wish. To use the logging system, you'd do this:

```
import logging


logging.info("About to start the program")
logging.debug("The value of the configuration item is found in config.json")
logging.error("This is a spurious error that will likely drive people
nuts")
```

Now, you might think that this would output all three lines, but in fact, the output varies by your system. On my system, which is configured to output errors, the output is:

```
ERROR:root:This is a spurious error that will likely drive people nuts
```

This shows me the error line, who was running (root, in this case), and the error description. If we wanted to see all the outputs, we could change the configuration of the default logger:

```
import logging


logging.basicConfig(level=logging.DEBUG)
logging.info("About to start the program")
```

```
logging.debug("The value of the configuration item is found in config.json")
logging.error("This is a spurious error that will likely drive people nuts")
```

This will output all three lines to the console because it is set to output anything greater than the `DEBUG` level of logging.

Sometimes, particularly if you are developing custom Python packages, you will want to create your own logs, so that the information is displayed in the log in the way you need it to debug people's issues with your package. In this case, you can use a custom log.

One of the biggest advantages of using a custom logger for your package is that you can send the log output to a specified file just for your logger:

```
logger = logging.getLogger("MyCustomLogger")
fh = logging.FileHandler('mycustomlogger.log')
fh.setLevel(logging.DEBUG)
logger.addHandler(fh)
logger.error("This is an error")
```

For example, with the above code, if we add a standard logging statement:

```
logging.error("This is an error that doesn't go to the file")
```

you would find that the file `mycustomlogger.log` contains the line `This is an error` but does not contain the line `This is an error that doesn't go to the file`.

You can add exception handling to your log output quite simply in the logger module. You simply add the `exc_info = true` flag to the log statement. If an exception occurs, it will be printed out without you having to track it down and show it:

```
try:
   this_isnt_going_to_work = 1 / 0
except Exception as e:
   logging.error("Exception occurred", exc_info=True)
ERROR:root:Exception occurred
Traceback (most recent call last):
   File "/Users/mtelles/PycharmProjects/html/logging_examples.py", line
17, in <module>
    this_isnt_going_to_work = 1 / 0
ZeroDivisionError: division by zero
```

Finally, you can change the way the data is output by overriding the `Formatter` for the log:

```
import logging

logging.basicConfig(format='%(asctime)s-%(levelname)s-%(message)s',
level=logging.ERROR)
logging.error("This is another error")
```

This formatter prints out the date and time of the error, the level of the error, and the message that was passed to it. In this case, we've also restricted the error level which is suppressed to be anything below error. This will print out the following:

```
2019-08-20 09:03:03,305-ERROR-This is another error
```

Configuration needs to be done before you do any logging or it won't apply. So, if we do some logs and then configure it, we won't see the date, time,or level.

Unit test

One of the biggest differences between the entry level developer and the professional programmer is in writing tests for one's code. A unit test is a form of **white box testing**. This means that you can look into the code and figure out what should and shouldn't be going on. For example, consider the following code:

```
def div_it(v1, v2):
   return v1/v2
```

Clearly, this silly function takes two arguments and divides the first by the second. The number of issues here is significant, but let's take a look at a naïve approach to testing this function using the Python unit test package.

First of all, if you have used test packages of other languages, Python is a bit unique in that naming is a big part of the process. Test files should be named beginning with `test_` for the test runner to pick them up when running from the command line. In addition, you must name all your tests prefixed with `test_` as well. The standard `unittest` package does not have a way to mark a specific class method as a test otherwise. Classes are used in Python for managing tests; they must derive from the `unittest.TestCase` class in order to be run as a test.

Let's look at a very simple example of some tests for the above function, using the Python `unittest` package:

```
import unittest

class TestTheFunction(unittest.TestCase):

   def test_with_one(self):
```

```
    assert div_it(1,1) == 1

def test_with_zero(self):
  assert(div_it(0,0)) == 0

if __name__ == '__main__':
   unittest.main()
```

In our test file, we first import the `unittest` package into our project. This is necessary to get all the classes and methods of the package. Next, we define our test class. In this case, the class is called `TestTheFunction`, indicating that we are testing the function `div_it` in our code. The `div_it` function is also included in this file so that we don't have to import it; it has been omitted from the listing simply to save some space on the printed page.

Our class contains two test methods, `test_with_one` and `test_with_zero`. No, they aren't brilliant bits of testing code, and you'd likely be fired if you wrote this sort of test in your own production code, but they illustrate what is needed to implement a test.

Notice that we have a check if this is the main file, and if so, to call the `unittest.main` function at the bottom. This is needed because the `unittest` module has to scan the file for test classes (those derived from `unittest.TestClass`) and test methods within those classes. If you run this file from the command line:

```
python3 test_funcs.py
```

You will see the following output, indicating that the tests have run:

```
======================================================================
ERROR: test_with_zero (__main__.TestTheFunction)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_funcs.py", line 15, in test_with_zero
    assert(div_it(0,0)) == 0
  File "test_funcs.py", line 6, in div_it
    return v1/v2
ZeroDivisionError: division by zero


----------------------------------------------------------------------
Ran 2 tests in 0.000s
```

Wait. What is this? An error? How can that possibly be? All sarcasm aside, it is clear that our little function has some issues. Division by zero is the most obvious one. But our test caught the problem, so we can fix it. Let's modify the function:

```
def div_it(v1, v2):
   if v2 == 0:
     return 0
return v1/v2
```

Now, when we re-run the tests:

```
Python3 test_funcs.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

We see that everything passes. It is a best practice to first write a test, and then write the code to make the test pass. Continue to do that until you have exhausted all the requirements for your feature.

You may have noticed the assert statements in the test methods. You can use any standard assert statement in the code, as well as a few special ones for testing as shown in *Table 10.2:*

Function name	Meaning
`assertEqual(a,b)`	Validates that a equals b
`assertNotEquals(a,b)`	Validates that a is not equal to b
`assertTrue(a)`	Validates that a is truthy
`asserFalse(a)`	Validates that a is not truthy
`assertIs(a,b)`	Validates that a is ba
`assertIn(a, list)`	Validate that a is in the iterable list
`assertIsInstance(a, t)`	Validate that a is of type t

Table 10.2: *The unittest package assert*

Now that we have understood the basics of the `unittest` package, it is time to use the package to do some testing.

Setup and teardown

In many cases, your tests will require you to do some initialization before running. For example, you might have to connect to a database or start up some process in the background. Doing so at the start of each and every test produces a lot of copy/paste of code. For this reason, the Python `unittest` package provides the `setUp` and `tearDown` methods, which can be overridden within your test class. If you do so, the `setUp` will be called before each test method, and the `tearDown` method will be called after each `test` method.

Similarly, the `unittest` package has methods that are called once before any test is run, and after all tests have been run:

```
class Test(unittest.TestCase):


@classmethod

def setUpClass(cls):

# Initialize things for all tests

@classmethod

def tearDownClass(cls):

    # Do things after all tests have run
```

Note that the `setupClass` and `tearDownClass` must be implemented as `classmethod` methods since they are statically called.

There are lots of other rather cool things in the `unittest` package if you plan to use it in your code and it is strongly recommended that you do. Also, you should take a look at the full definition of the module.

If you don't like putting the `unittest.main` call at the bottom of each of your test files, you can run them using the `unittest` module using the following syntax:

```
python3 -m unittest <filenames>
```

This invokes the `unittest` package, calling its main function and passes along the list of filenames that you have specified. The `unittest` package will scan each file for test classes and within them for test methods that need to be invoked. It will produce a list of all failed tests as well.

Mocking

When you hear the word mock, two things might come to mind. If you are not a software developer, the odds are good you'll think of someone taunting you or making fun of you. You might even thing of fakes, like *mock turtle*. For software developers, however, mock means fake, or shim, or any of the other terms that have

been used in the software world for replacements for existing code. A mock, in Python, is a way to insert your own code in place of existing code. That code maybe yours, it may be in a third-party package you use, or it might even be in the wrapper for the operating system you are running upon.

The Python `mock` library is made considerably easier by the way in which Python works. Because everything is an object and every object is just a dictionary of attributes and functionality, mocking an object just means replacing the bits of functionality you want. For example, if you wanted to mock an object that is retrieved by a given function in order to observe the behavior of the function when those settings are used, it is very trivial to do so. Let's look at a couple of examples of mocking.

First of all, you need to include the mock library. Mock is not a standard part of the Python distribution, so you will use pip to install it. As with all things pip, you can either install it in the system as a whole or in your personal project. It is generally better to install things locally:

```
pip install --user mock
```

Note the use of a double dash before the user flag.

The next thing we need is something to mock. Mocking a simple function, or even classis very simple. What becomes a little more complicated is when the thing you want to mock isn't directly within your control. For example, consider the following code snippet:

```
def is_this_a_leap_day():
   m = datetime.datetime.today().month
  d = datetime.datetime.today().day

   if (m == 2) and (d == 29):
     return True

   return False

def do_something_with_leap_days():
   if is_this_a_leap_day():
     print("Doing something with a leap day")
else:
  print("Not doing anything with leap days")
```

The code that we are interested in testing here is the `do_something_with_leap_days` function. This function is easy to test most of the time, since it is *not a leap day*,

and we could easily verify that the path for not a leap day works properly. We could individually test the other path, but that might obscure something that happens when it is a leap day. We could mock the `is_this_a_leap_day` function, but that might hide bugs within that code. It is better to actually mock the system `datetime` call and let the function work naturally given a new date and time. In general, it is best to mock things as far down the calling chain as possible, since that will permit you to observe changes all the way through the system, rather than just at the point where you are concerned about.

So, how do we go about mocking the system call to `datetime`? Again, the first thing we need to do is to import the `mock` package so that we have access to all of its functionality. The mock package contains a class called **Mock**, not surprisingly, which does all of the work of mocking items:

```
import mock

import datetime
```

Next, we need to set up the portion of the code we want to mock. We can do this in a variety of ways, but you usually want to mock the simplest and most discrete item you need to modify:

```
def get_mock_today():

   print("Mock called")

  m = mock.Mock()

 m.month = 2

 m.day = 29

 return m


datetime.datetime = mock.Mock()

datetime.datetime.today.return_value = get_mock_today()
```

Let's take a look at what is going on here. First of all, we created a simple function just so we can illustrate what happens and print out some information. You don't have to do it this way; you could actually just override the return. The function simply sets the portions of the `mock` object that we want to return. Mock objects are like dynamic objects in C# or similar things in Java. They can have any attribute you want to be assigned to them and will return themselves in the format of a standard Python object.

Once the function is defined, we need to assign it to something. We do this by creating another mock object and assigning it to the system `datetime` class instance. From here, we need to tell the mock to return the values we want to the calling application. This is done via the mock `return_value` attribute. When the mock object is invoked, the `return_value` attribute is returned to the caller.

If we call the function before we set up the mock, presuming that it is not a leap day, we will see the following output:

```
Not doing anything with leap days
```

Now, if we install our mock object and call the function again, we see a very different output:

```
Mock called
Doing something with a leap day
```

As you can see, we have modified the behavior of the system without modifying our code at all. This is incredibly useful for testing and debugging systems and should be considered an essential part of your programming arsenal in Python.

There is another aspect of mocking that is worth discussing. Sometimes, it isn't that you want to change the behavior of a function or method in your application. You may want to simply verify that a given piece of code was called properly. In Java, we often do this sort of testing to verify that a dynamically loaded module was called when the system responds to a given message. Otherwise, it is really hard to tell if something was done correctly or not. The Python mocking library also has a way to do this. Consider the following code:

```
def a_function_that_has_to_be_called(a, b, c):
    return a+b+c


def a_weird_function_to_test(b):
    if b == False:
      return a_function_that_has_to_be_called(1,2,3)
```

A contrived example, yes, but it illustrates a point. We want to be sure that our function `a_function_that_has_to_be_called` is called when the system is running. Note that we don't want to change the behavior of the function; we just want to verify its call. Here's how you do it using the `mock` package in Python:

```
myMock = mock.Mock(spec=a_function_that_has_to_be_called)
a_function_that_has_to_be_called = myMock
myMock.assert_called_with(a=1, b=2, c=3)
```

The `myMock` object is not passed to anything nor is it inserted into the call chain. Instead, we are setting it up to `model` a given function using the `spec=` parameter. This parameter tells the mock to `wrap` that object. When we reassign our local function to be the function, the outer function will invoke our mock instead. We can then verify that our mock was called. Note that we are not returning anything from our mock, so the return from the outer function is just our `mock` object. If you want to make sure everything works properly, you need to return a proper value in the mock

return_value. If you run this code, you will see no output since we aren't printing anything and the mock assert doesn't fail. Try running it this way:

```
myMock.assert_called_with(a=3, b=2, c=1)
Traceback (most recent call last):
   File "/Users/mtelles/PycharmProjects/html/mock_example.py", line 43,
in <module>
    myMock.assert_called_with(a=1, b=2, c=3)
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/
site-packages/mock/mock.py", line 932, in assert_called_with
    raise AssertionError(error_message)
AssertionError: expected call not found.
Expected: mock(a=1, b=2, c=3)
Actual: not called.
```

There are many uses for mocks, but the important aspects are how you should not use them. You should only mock things outside your control. Never mock something that you can easily modify. Do not mock the pieces of the code that you want to test, only those that you are sure of.

Concurrency

In the software world, concurrency and threading are a generally big deal. While Python is not generally used at low levels that require such things, there are many cases where you will want to know how to use the concurrency packages in your own applications. It is important to note that, especially in Python, CPU bound applications will not benefit well from threading, whereas I/O bound applications will benefit greatly. Your mileage will always vary; do not assume that a badly written program will suddenly become usable due to speed gains from threading.

Let's look at a set of very simple functions:

```
def print_names(names):
   for name in names:
     print("Hello {0}".format(name))
   time.sleep(1)

def reverse_name(names):
   for name in names:
  print("Goodbye {0}".format(name[::-1]))
time.sleep(1)
```

Obviously, these are contrived functions intended to illustrate the point and not to be used in some sort of production system. The functions themselves aren't important, merely that they do a fair amount of output and do very little computation.

We can run these functions and collect some metrics about them:

```
names = ['Anita','Barry','Christian','Daniel','Edward','Fred','George']


t1 = time.time()
print_names(names)
reverse_name(names)
t2 = time.time()
print("Time: {0}".format(t2-t1))
```

Of course, in this example, the two functions are being called sequentially, so the time to execute them will be the combined time to run each one. If we look at the output of the running of this code, you'll see that this is clearly true, and that the order is very well defined and obvious:

```
Hello Anita
Hello Barry
Hello Christian
Hello Daniel
Hello Edward
Hello Fred
Hello George
Goodbye atinA
Goodbye yrraB
Goodbye naitsirhC
Goodbye leinaD
Goodbye drawdE
Goodbye derF
Goodbye egroeG
Time: 14.048261880874634
```

Now, suppose we run them in threads. In Python, threading is done with the threading package using the Thread object. Using it is quite simple; you just supply the function and the arguments to the function to the thread:

```
from threading import Thread


t1 = time.time()
thread_1 = Thread(target=print_names, args=(names,))
thread_2 = Thread(target=reverse_name, args=(names,))
thread_1.start()
thread_2.start()
thread_1.join()
thread_2.join()
t2 = time.time()
print("Time: {0}".format(t2-t1))
```

If you run this code, you will see the following output:

```
Hello Anita
Goodbye atinA
Hello Barry
Goodbye yrraB
Hello Christian
Goodbye naitsirhC
Goodbye leinaD
Hello Daniel
Goodbye drawdE
Hello Edward
Goodbye derF
Hello Fred
Goodbye egroeG
Hello George

Time: 7.0181310176849365
```

Two things should be apparent from the output. First of all, the time is much less than the first case, about half as much time. This indicates that the two functions are being run more or less in parallel, something you want when you are doing threading. Secondly, and more importantly, you notice that the order of operations is quite different. In some cases, you will see that the outputs from the first and second

functions just alternate. In some cases, that is not the outcome. Threads that need to synchronize on things, especially data, must use a lot of effort to make sure they stay in synch.

A few notes on Python threading. First of all, you will notice that there are three steps for using threads in Python.

1. Create a Thread object with the function you wish to run in a thread, passing along any arguments that you wish to pass to the function. Note that the arguments parameter is a tuple, so if you only have a single argument, you must pass it with a trailing comma.
2. Call the start method of the thread object, which will start the thread. If this is a background thread, and you don't really care when it finishes, this is all you need to do. This is bad practice, however, as it can leave your program hanging when you try to exit.
3. Call the thread join method to wait for it to finish.

That's all there is to it! It should be noted that the Thread class in Python is not really multi-threaded, but instead uses a manager to switch between threads. Python supports all the usual thread classes, such as semaphores, locks, and timers. There is also a barrier class, used to keep multiple threads in sync. All of this is well beyond the scope of this book, as there could be entire books on multi-threading in any language along with all the pitfalls and advantages involved.

The emoji package

Sometimes, you add features that are critical to your application. Sometimes, you add a feature that is just pure fun so that the user can get some enjoyment out of what is otherwise a dreary day. The `emoji` package is definitely of the latter variety. Emojis started as text-based emoticons, a way of showing an expression such as a smile or wry grin, with a few characters like a semi-colon and a parenthesis. The prototypical emoticon is :) looking like a sideways smile.

As software expanded and operating systems became more and more graphical, and with the rise of mobile phone software, emoticons grew into emojis, which are graphical in nature. The emoji comes from Japan, as one might guess from the name. They came from Japanese phone manufacturers, who were trying to keep up with their users who loved graphics. Looking at the emoticon, which was originally a set of characters, and later an expanded ASCII character set, they chose to use embeddable test that would be rendered in a standard way.

Python supports emojis via the emoji package, which is a third-party package available on PyPI. You install it in the standard way:

```
pip install emoji
```

There isn't a lot to say about using emojis, they work pretty much the way you would expect. To render an emoji in your code, you use the `emoji.emojize` method, with a textual code embedded in the string passed to it. The actual code varies depending on the emoji you want to display. Here's an example of how you would output a couple of emojis on the console line:

```
import emoji


print("This is a test :thumbs_up:")
print(emoji.emojize("This is a test :thumbs_up:"))
print(emoji.emojize("That was really funny!:grinning_face_with_big_
eyes:"))
```

Note that the first line outputs a simple string with no emojis, while the remaining two illustrate two of the possible codes (thumbs up and grinning face with big eyes). How they render on your specific system varies, but the idea is always the same. If you want to see all the emoji shortcuts, you can use this code:

```
for k, v in emoji.UNICODE_EMOJI.items():
   print(v)
```

As you might expect, the emoji package is used quite heavily in social media applications!

The pprint package

The `pprint` package is one of the `utility` packages that you don't know you need until you need it. The `pprint` package allows you to print things in an organized fashion. If you are retrieving json data from a REST service, for example, it is painful to look at the data when it is just spewed out onto the console. Consider, for example, the following:

```
dict = {
   "name": {
  "first": "matt",
"middle": "a",
 "last": "telles"
},
"address": {
  "city": "new york",
"state": "ny",
```

```
"zip_code": "10012",
"street": "1313 Mockingbird Lane"
},
"age": "29"
}


print(dict)
```

Using the standard print function results in the amazingly unhelpful output of:

```
{'name': {'first': 'matt', 'middle': 'a', 'last': 'telles'}, 'address':
{'city': 'new york', 'state': 'ny', 'zip_code': '10012', 'street': '1313
Mockingbird Lane'}, 'age': '29'}
```

What is a part of what here? Very hard to track down which element is a component of which parent using this output. Using the `pprint` package, however, makes it extremely legible:

```
import pprint


pprint.pprint(dict)
{'address': {'city': 'new york',
             'state': 'ny',
             'street': '1313 Mockingbird Lane',
             'zip_code': '10012'},
'age': '29',
'name': {'first': 'matt', 'last': 'telles', 'middle': 'a'}}
```

Here, we can see exactly what we are trying to do, with the `address`, `age`, and `name` components lay out and their children inside them indented. You can do the same with JSON strings or any other sort of structured data.

The requests package

Saving the best for last, we have the requests package. The requests package in Python allows you to make REST calls to any service that *talks* REST out there in the wild. It works with virtually any service, whether or not authentication is required, with or without cookies, using any sort of encoding or processing.

REST calls break down into four 'verbs'. The verbs represent the kind of functionality you wish the service to implement. They are `GET`, `PUT`, `POST`, and `DELETE`. These

four verbs roughly correspond to the four CRUD operations, `GET` is the same as `READ`, `POST` is the same as `CREATE`, `PUT` is the same as `UPDATE`, and `DELETE` is the same as its **CRUD** acronym piece.

To use the requests library, you need to install it into your Python environment:

```
pip install requests
```

You then import the package, as usual, with:

```
import requests
```

To see how it works, let's grab the source code to the Google search home page:

```
import requests
from pprint import pprint


req = requests.get('http://www.google.com')
print(req.content)
```

The output from this will be the same thing you would see if you did a view source of the page in a web browser, which looks something like this:

```
b'<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"
lang="en"><head><meta content="Search the world\'s information, including
webpages
```

Obviously, we've omitted a big chunk of it, but you can see that this is the head of the page. The `get()` method retrieves whatever is at the URL specified and returns it.

We can retrieve data using parameters as well. For example, the well-known jsontest.com site is used to test REST libraries. We can use this to test requests by making a request for the current time and date:

```
# Test getting the time and date from jsontest
req = requests.get('http://date.jsontest.com')
pprint(req.json())
```

This snippet will output the current date and time in JSON format:

```
{'date': '08-26-2019',
'milliseconds_since_epoch': 1566831702564,
'time': '03:01:42 PM'}
```

Obviously, your output will be different based on the time you make the request, but you see the format of the returned JSON.

You can do POSTs in requests as well, quite easily. The aforementioned jsontest.com site supports a service called validate, which will accept an input JSON string and

tell you whether or not it is valid. Let's try it with the string we got back from the date request above, to see if they send us a valid date, shall we?

```
# Test validatting json

json_data = req.json()

req = requests.post("http://validate.jsontest.com?json="+str(json_data))

print(req.content)
```

```
'{\n   "size": 3,\n   "parse_time_nanoseconds": 112803,\n   "object_or_
array": "object",\n   "validate": true,\n   "empty": false\n}
```

Isn't it nice to know that they return valid JSON?

You might wonder how you send data to a service that expects its information to be packaged into the body of the request. It is almost as easy as pie in requests:

```
def generalized_post(url, json_data):

   headers = {'content-type': 'application/json'}

  response1 = requests.post(url, json=json_data, headers=headers)

 return response1
```

This simple function wraps up all of the request post requirements into a single callable unit. You pass it the JSON data you want to post and the URL to post to, and it sets up the headers and sends the data. Note the use of the `json` parameter in the `POST` command. This puts the data into the body of the request, rather than as part of the URL. The headers parameter sets certain header values that you might need to set. In this case, we are telling the service that we are sending data in JSON format. You might also include things here like the user name, password, or a token used to identify the caller to the system.

The headers parameter is set upon return as well, so that you can check for a CSRF token, for example, when you do a login to a system. You can easily examine the returned headers in the `response` object. Finally, if you need to set cookies, you use the cookies parameter:

```
my_cookies = { "cookie_1": "some-cookie-value" }

response = requests.post(some_url, cookies=my_cookies)
```

The nicest part about the requests library is that it is amazingly simple to use for simple cases and makes it very easy to implement difficult cases as well. If you are using REST services in your environment at work, you should seriously consider using the requests library to interact with and to test those services.

That concludes our somewhat whirlwind tour of some of the most popular Python packages out there for use in your applications. Remember, do not reinvent the

wheel! First, consult PyPI to see if a package already exists to suit your needs, then install it and use it rather than writing your own.

Conclusion

In this chapter, we've gone through many of the major packages for the Python programming languages. These packages have extensive usage and have been well defined and well maintained for years. You should try very hard to see if a package exists before you decide to write your own. If you do so, not only will you have to maintain your own code, but you will likely make the same mistakes that have been made in the past!

In our final chapter, we'll explore a set of tricks and tips that will help you write more professional Python.

Questions

1. Why should we use existing packages instead of writing our own?
2. What is the itertools package and why is it used?
3. Why use Flask instead of writing our own web server?
4. How can we display emojis in our code easily?
5. What package is used for more elegant displays of output?

Chapter 11
General Tips and Tricks

Introduction

Sometimes the most important things you learn from a book are the little tips and tricks that the experienced developers know that you haven't yet encountered. In this chapter, we'll look at those things that will help you in your professional career. From determining which version of Python you are using to determining the size of your objects in memory for optimization, this chapter will help you in your work.

Structure

Here are the things you will learn in this chapter:

- Implementing a switch statement using dictionaries
- Remove all duplicates from a list
- Determine the size of your objects in memory
- Find the most frequent item in a list
- Creating an `enum` in a class
- Detect Python version.
- Using the _ (underscore) operator
- Discovering where a module is imported from

- Swapping two values without an intermediate temporary
- Using the `classmethod` decorator to create static methods
- Using the `**kwargs` to pass a named list of parameters
- Type hints
- Finding the day of the week using the calendar module
- Working with regular expressions

Objectives

By the end of this chapter, you should have picked up at least a few tips and tricks that you will be using in your day to day programming in Python. You'll understand better the `classmethod` decorator, understand how to use type hints to tell other developers the best way to use your code, and translate several concepts from your previous languages, such as `enum` and `switch` statements, to Python.

Implementing a switch statement with dictionaries

Switch statements exist in multiple other languages, and provide a way to avoid detailed `if...elsestatements`. We've briefly discussed the issue of Python not having the equivalent of a switch statement. In C#, or C++, or Java, you can write code like this:

```
switch (some_value) {

case something1:

case something2:

}
```

Python has no such construct, which leads to some ugly code in your own applications that can look like:

```
if some_value == something1:

    # Do something

elseif some_value == something2:

    # Do something else
```

This may not appear too bad, but consider the case of an application that has potentially dozens or more possible values to check. The `if...elseif...else` statements quickly get out of hand and often lead to errors because problems are introduced during maintenance coding. Someone might add an and, or an or to a `if` statement and cause the whole house of cards to collapse.

We looked at one alternative, which is to use a dictionary of function pointers to solve the problem. For example, consider the case of writing a text based game where the end user types in a command and the application responds to it. We might have code that looks like this:

```
done = False

def handle_north():
   print("Moving north")

def handle_south():
   print("Moving south")

def handle_east():
   print("Moving east")

def handle_west():
   print("Moving west")

def handle_quit():
   global done
   done = True

command_handlers = {
   'n': handle_north,
  'north': handle_north,
 's': handle_south,
   'south': handle_south,
  'e': handle_east,
   'east': handle_east,
  'w': handle_west,
   'west': handle_west,
  'quit': handle_quit,
   'q': handle_quit
}

def command_processor(cmd):
   cmd_func = command_handlers.get(cmd, None)
  if cmd_func == None:
    print("Invalid command: {0}".format(cmd))
```

```
    else:
      cmd_func()

while not done:
    cmd = input("Enter a command: ")
  command_processor(cmd)
```

Now, there is nothing wrong with this code. It works, it is reasonably Pythonic, and it is fairly easy to read. We could make this a little nicer by wrapping the code up into a class, so that we could reuse different command handlers for different programs, or even change the overall architecture of the command handler without anyone outside the class seeing the difference. This is an important aspect of Python OOP. Allow the user to get the job done without having to know the internals of the system. Let's see what a class based approach might look like:

```
class CommandProcessor:

    def __init__(self):
      self.done = False

def handle_north(self):
  print("Moving north")

    def handle_south(self):
      print("Moving south")

    def handle_east(self):
      print("Moving east")

    def handle_west(self):
      print("Moving west")

    def handle_quit(self):
      self.done = True

    def process(self):
      while not self.done:
    cmd = input("Enter a command: ")
# Convert this into a method
attr_name = 'handle_' + cmd
method_handler = getattr(self, attr_name, None)
if method_handler  == None:
```

```
   print("Invalid command: {0}".format(cmd))
else:
    method_handler()

# Main program goes here
c = CommandProcessor()
c.process()
```

You can probably see that this is essentially the same code, just wrapped up in a class. Rather than using a dictionary, we take advantage of the fact that we can get method names as attributes within the class structure. We could have done this with functions, but then we run the risk of picking up a completely unrelated function in our system.

Of course, this being Python, there is always another alternative. In this case, let's borrow from our OOP programming language siblings and not worry about any sort of Pythonic magic to find our handlers. We'll use base classes with a single class to do all the handling:

```
class CommandHandlerManager:
    command_handlers = {}
    def __init__(self):
        pass
    def register(self, command_type, class_handler):
       if command_type not in self.command_handlers:
            self.command_handlers[command_type] = class_handler
        return self
    def fetch_handler(self, command_type):
        if command_type in self.command_handlers:
            return self.command_handlers[command_type]
        return None

class BaseCommandHandler(object):
    def __init__(self):
        pass
    def handle_command(self):
        pass

class NorthCommandHandler(BaseCommandHandler):
    def __init__(self):
```

```
        super().__init__()
        CommandHandlerManager().register('north', self)
        CommandHandlerManager().register('n', self)
    def handle_command(self):
        print("Moving North")
        return False

class QuitCommandHandler(BaseCommandHandler):
    def __init__(self):
        super().__init__()
        CommandHandlerManager().register('quit', self)
        CommandHandlerManager().register('q', self)

    def handle_command(self):
        print("Quitting")
        return True

class Game:
    def __init__(self):
        NorthCommandHandler()
        QuitCommandHandler()
    def process(self):
        done = False
        while not done:
            cmd = input("Enter a command: ")
            hndlr = CommandHandlerManager().fetch_handler(cmd)
            if hndlr == None:
                print("Invalid command")
            else:
                done = hndlr.handle_command()

g = Game()
g.process()
```

What is different in this case? We aren't using any magic to retrieve our handlers, rather they register themselves with a single manager class. There is a slight bit of trickery here in that we use a static class variable to manage the handlers, this is so that we cannot worry about keeping a singleton object around as we would have to do in other languages.

The big advantage to this system is that we can extend it easily without any code modifications. If we wanted to handle another command, we'd simply create a new class that handled whatever we wanted and register it with the command handler. The remainder of the program would continue to work exactly the way it did. This is a nice step up from the class wrapper, because we would have to add new methods to the class in the second example to add new commands. Here, we can just add a brand new class that knows nothing about the others. In fact, we could use dynamic programming to load new classes from external files if we really wanted to.

Remove duplicates from a list

It is inevitable, if you have a list of items that you are going to end up with duplicates. Whether the list is of emails for a campaign, or account identifiers for processing, there is always some danger in processing the same item twice. It may mean annoying a customer, which is bad, or in double billing someone, which is really a disaster. For this reason, de-duplicating a list is one of the most common things you can do in programming. Naturally, Python makes it easy to do this, and provides you with a vast set of ways to do it. Let's look at a few options.

For simplicity, we will start with a list of integer values that contains some duplicates:

```
# Given a list with duplicates
list_with_duplicates = [1,2,3,12,1,2,3,4,5,6,1,2,3,7,8,9]
```

The first approach might be a simple brute force approach:

```
list_without_duplicates = []
for pd in list_with_duplicates:
   if pd not in list_without_duplicates:
     list_without_duplicates.append(pd)

print(list_without_duplicates)
```

```
[1, 2, 3, 12, 4, 5, 6, 7, 8, 9]
```

As you can see, this method works. It has some performance issues for large lists, since you are essentially creating a full copy of the list. Surely we can do better than that? We can, in fact, do somewhat better than that. Let's try a different method, this time using a dictionary to hold our intermediary values:

```
# Convert the list to a dictionary
dictionary_without_duplicates = dict(zip(list_with_duplicates, list_
with_duplicates))
print(dictionary_without_duplicates)
```

```
{1: 1, 2: 2, 3: 3, 12: 12, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
```

Once again, this works, and has the advantage of taking less space than duplicating the entire list. Ofcourse, we still need to convert it back to a list when we are done, which might be somewhat painful, since we must extract the keys and add them to a list.

A third alternative is to use a set in place of a dictionary, since a set has the property that it does not contain duplicates by its nature. Knowing this, we probably should have started with a set rather than alist, but the presumption here is that the list came from something that needed the duplicates at the time.

```
# Convert to a set

list_without_duplicates = list(set(list_with_duplicates))

print(list_without_duplicates)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 12]
```

Again, this works and has the advantage of less memory and a more compact format. There is an issue here, however, that might not be apparent. Looking at the first output compared with the rest of them, you'll see a glaring difference. The dictionary and set both fail to maintain the order of the elements. If you don't care about the order, that's no big deal, but in many cases you do, such as when you are processing a list of events. Let's see what we can do about that:

```
interlist = {}

list_without_duplicates = [interlist.setdefault(x, x) for x in list_with_
duplicates if x not in interlist]

print(list_without_duplicates)
```

```
[1, 2, 3, 12, 4, 5, 6, 7, 8, 9]
```

Once again, this is pretty fast, since it uses a list comprehension to rebuild our list into a dictionary which is then converted back into a list. There's a fair amount of overhead, and this isn't really that different from the dictionary approach, but it does maintain order.

Finally, we can use a collection that was built to be optimized for exactly this purpose. With Python 3.5 and above, the `OrderedDict` class was added, which is a dictionary that maintains its order, as its name would suggest. We can use this to do the same job:

```
from collections import OrderedDict

list_without_duplicates= list(OrderedDict.fromkeys(list_with_
duplicates))

print(list_without_duplicates)
```

```
[1, 2, 3, 12, 4, 5, 6, 7, 8, 9]
```

This approach is fast, it is relatively compact, and it maintains order. So, you can select what works best for your situation, and doesn't require you to reinvent the wheel.

Determine the size of your objects in memory

Sometimes, programming is not about the inputs from the user, or the outputs to disk or even the processing of data. Sometimes, programming is about optimization and worrying about memory and disk space usage. It isn't a great idea to optimize too much when you are working with Python, as the language itself just isn't made to be overly memory efficient. To get an idea of why that is, let's look at the size of various simple objects in Python.

To determine the size of an object, you use the `sys.getsizeof()` method, after importing the `sys` package. Let's look at a few to get a feel for it:

```
import sys

print("An Integer:")
i = int(100)
print(sys.getsizeof(i))

print("Empty string:")
empty_string = ""
print(sys.getsizeof(empty_string))

print("A normal string")
s = "Hello world"
print(sys.getsizeof(s))

print("Increasing string:")
for i in range(0,5):
   empty_string = empty_string + chr(i)
  print(sys.getsizeof(empty_string))
```

The output from this code snippet is shown in *Figure 11.1:*

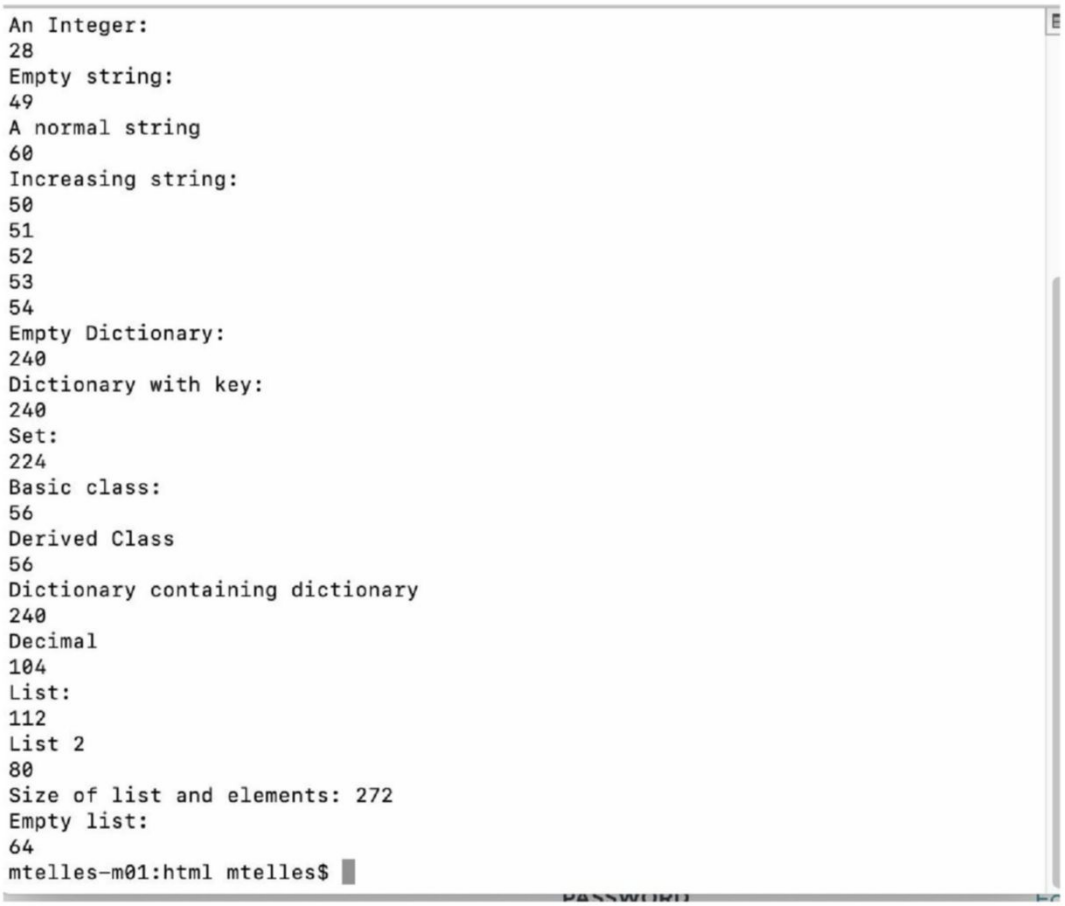

Figure 11.1: *Printing the size of objects*

There are some interesting things to take away from this. First of all, the size of an integer is 28 bytes, which is considerably higher than any other language you might work in. In C++, for example, even along integer is only about 8 bytes. Why the massive overhead? Remember, that everything in Python is an object. Thus, every object has all of the stuff that goes along with object. It has a name, and a class, and a dictionary of attributes. That adds up.

Next up is the string class, which you'll see is even bigger. An empty string is 49 bytes. That's rather large, and should make you think twice about creating multiple strings to concatenate them together later. Admittedly, `50` bytes here or there is just not that important in today's multi-gigabyte environments, but they do add up eventually. Notice also that each character added to the base empty string only adds one byte (assuming a single-byte character set) to the overall total, so there is really no harm to long strings.

Next up are dictionaries, which are one of the more fundamental types in Python. Let's look at the overhead of a dictionary that is empty, and one with keys:

```
print("Empty Dictionary:")
dict = {}
print(sys.getsizeof(dict))
print("Dictionary with key:")
dict['key1'] = 'value1'
print(sys.getsizeof(dict))
```

```
Empty Dictionary:
240
Dictionary with key:
240
```

Now that's interesting, isn't it? The size of a dictionary with a single key is the same as an empty dictionary! How can that possibly be? The answer lies in how dictionaries are implemented. Each dictionary has a hash table built into it to store keys and their values. When a dictionary is initially created, that hash table is allocated for use so that each additional key does not add significant time to allocate more memory. Dictionaries are allocated in blocks, so until you exceed a single block, the size will not change.

Sets are quite similar, having a base size that is expanded as you add to them:

```
print("Set:")
set1 = set()
print(sys.getsizeof(set1))
```

```
Set:
224
```

Since everything is an object, you might be wondering what the overhead of creating your own classes, and derived classes might be. Let's look:

```
print("Basic class:")
f = Foo()
print(sys.getsizeof(f))


class DerivedFoo(Foo):
```

```
    def __init__(self):
        super().__init__()

print("Derived Class")
df = DerivedFoo()
print(sys.getsizeof(df))
```

```
Basic class:
56
Derived Class
56
```

Likewise, you might think that a dictionary that contains another dictionary would be much larger than a single dictionary, right?

```
print("Dictionary containing dictionary")
dict_with_dictionary = {}
dict_with_dictionary['test'] = {
    'a': 1,
    'b': 2
}
print(sys.getsizeof(dict_with_dictionary))
```

```
Dictionary containing dictionary
240
```

That doesn't make sense, does it? In fact, it doesn't make sense at all. But let's keep going and maybe we can figure out what might be going on here.

```
print("Decimal")
import decimal

d = decimal.Decimal()
print(sys.getsizeof(d))
```

```
Decimal
104
```

So decimals are larger than integers, since decimals contain more information, this makes sense:

```
print("List:")
l = [1,2,3,4,5,6]
print(sys.getsizeof(l))


print("List 2")
l2 = [decimal.Decimal(100), decimal.Decimal(200)]
print(sys.getsizeof(l2))
```

```
List:
112
List 2
80
```

Wait, we know that a Decimal object is fairly large, so why is a list of them smaller than a list of integers? The answer lies in how `getsizeof` works. For example:

```
list_1 = []
print("Empty list:")
print(sys.getsizeof(list_1))
```

```
Empty list:
64
```

So, the overhead of a list object is 64 bytes. Our list with six elements in it has `112` byte, which is `48` bytes more than the empty list. The list with two `Decimal` elements in it is `80`, which is `16` bytes more than the empty list. That would mean the size of a single `Decimal` is `8` bytes? That makes no sense, since we looked at Decimal and they were well over `100` bytes. So why are we getting these silly answers? The `getsizeof` method returns a shallow evaluation of the object passed to it. It does not traverse the data structure to see how big each elements in it is, it treats each element in it (the length of the structure) as containing only pointers. Each pointer in Python is eight bytes, so you can see how the math works.

What if we wanted to find the actual size taken up by a list? We could do something like this:

```
size = sys.getsizeof([])
for li in l2:
   size = size + sys.getsizeof(li)
print("Size of list and elements: {0}".format(size))
```

```
Size of list and elements: 272
```

So, the size of our list of `Decimal` values is `288` bytes. Since the base list size is `64` bytes, and each `Decimal` object is `104` bytes, that means that the total stored in the list is `64+104+104`, which is `272`, which is the correct value. This method won't work if the objects stored in the list are containers themselves, since again, you will only get the size of the container plus the size of a pointer for each element, but it works well to illustrate the point.

Find the most frequent item in a list

We've all been there, having to find that item that occurs most frequently in a data structure. Maybe it is the most purchased item in your shopping site. Perhaps it is the web page that gets hit the most often. If you are a tester, it could easily be the test that has the most failures over the last year. Whatever it is, you want an easy way to find the data you need, and Python is here to help you.

The problem is, your data isn't always something simple. Take a look at these two simple lists:

```
list_1 = [1,2,3,2,3,2]
list_2 = ['a', 'b', 'a', 'b', 'c']
```

Obviously, we can't do simple math on the individual items, since the second list contains characters. It could contain the words of a book, and you want to find the most commonly used word in the work. Maybe it is a list of UPC values for commonly purchased items. Whatever it is, about all we canguarantee is that the data is probably comparable, in that we can compare one of the items to another. Yet, we need to find the most common element in the iterable.

Your first choice might be a simple brute force approach. It would likely look something like this:

```
def most_common_brute_force(l):

   # Find the counts of all elements
  dict_of_counts = {}
 for i in l:
   if i in dict_of_counts.keys():
    dict_of_counts[i] = dict_of_counts[i] + 1
else:
 dict_of_counts[i] = 1

   max_count = -1
   max_value = -1
```

```
  for k, v in dict_of_counts.items():
     if v > max_count:
       max_count = v
        max_value = k

 return max_value

print(most_common_brute_force(list_1))
print(most_common_brute_force(list_2))
```

```
2
A
```

As you can see, the brute force method works, but it is a bit ugly and not terribly efficient. We reduce the full list to a dictionary, which as we've seen is considerably bigger in size but contains fewer elements. We then iterate over the dictionary elements, meaning that we go through the entire list two times. Surely there is something a bit more Pythonic?

In fact, we can use one of the packages we looked at last chapter, the itertools package, to do a lot of the work for us:

```
import itertools

def most_common_itertools(l):
   # First, sort the list
  sl = sorted(l)

 # Next, get all the groups
groups = itertools.groupby(sl)

   # Find the most common
  max_count = -1
 max_value = -1
for k,v in groups:
  count = sum(1 for _ in v)
if count > max_count:
 max_count = count
max_value = k
   return max_value
```

```
print(most_common_itertools(list_1))
print(most_common_itertools(list_2))
```

This also works, and produces the same output. It is slightly more efficient and a bit more Pythonic. Also, please note the line which reads:

```
count = sum(1 for _ in v)
```

This is a really nice and efficient way to retrieve the count of items in a generator, presuming that the generator actually terminates (for example, a generator that generates Fibonacci series values would not). It is quite Pythonic and easy to read.

Of course, all of these have the same problems, we are iterating over the items too many times. The `groupby` method, while easy to use, isn't always the best choice for things like this, since it was designed to be a more general solution to grouping data. So how should we best do this? It turns out that the designers of Python considered this problem when they were designing Python 3, and added a new set of collections to the language standard packages. One of these packages includes a class called Counter which does exactly this:

```
from collections import Counter

def most_common_counter(lst):
   data = Counter(lst)
  return data.most_common(1)[0][0]

print(most_common_counter(list_1))
print(most_common_counter(list_2))
```

This is a great way of finding the most common element, and should be used so long as you have access to the `Counter` class. If you don't, there's another way to do it that is almost as efficient, if a little harder to read:

```
def most_common_using_comprehension_and_max(l):
   return max(((item, l.count(item)) for item in set(l)), key=lambda val:
val[1])[0]
```

Whichever method you choose, please don't brute force the solution unless there is no other choice. That's just reinventing the wheel.

Creating an enum in a class

Once upon a time, we used either strings or numeric values to represent values. We might have had something like this in C:

```
if (x == 0) { // State: Off
```

```
 Set_state(1); // State: Booting
}
```

This was ugly, it was confusing, and if someone changed the order of the state in your system everything would break. The designers of C (and C++, and Java, and C#) decided that it made more sense to have a different sort of value, called an **enumeration**, not to be confused with enumerating the values in an iterable. An `enum` replaced numeric constants with human-readable names:

```
enum {
    OFF = 0,
    BOOTING,
    RUNNING,
    SHUTDOWN
} State;
```

In the early editions of Python, there was no such thing as an `enum`, so we ended up with the same ugly kinds of code that we had in the early days of C. With Python 3 (and later backported into Python 2) we have the new and shiny `Enum` class. It isn't quite the same as an enumeration in the other languages, but you can use it in a very similar fashion. Here's how you do it:

```
from enum import Enum

class MyClass:

   State = Enum('State','Off Booting Running ShutDown')

   def __init__(self):
     self.my_state = MyClass.State.Off

def set_state(self, state):
  if state in MyClass.State:
   self.my_state = state

   def get_state(self):
     return self.my_state

mc = MyClass()
print(mc.get_state())
mc.set_state(MyClass.State.Running)
print(mc.get_state())
print(mc.get_state().name)
```

Note that `Enum` is a class that produces an enum-like structure. We can embed the enum in our own class or create one as a public class that is derived from `Enum` that can be used in multiple classes. Note that we assign the values of the State enum class as strings, but they appear to be constants. This is Python *magic* but it makes the code much easier to read. Also notice that we can check to see if a given value is in an enumeration, since trying to set an enumerated value that isn't there will cause a `KeyError` exception. Finally, you can get the human-readable name to print out by referencing the `name` portion of the class instance.

Detect Python version

Python is generally agnostic of versions. In most upgrades, the changes are either; extensions, adding new functionality or functions, or they are brand new, with no ties to the past. There are, however, some things that require you to actually know about the version of the system you are using. For example, if you are using a language feature that only exists in a later version, you don't want to call it in an earlier version. While syntax issues, like the parentheses around a `print()` function call generally will not work at all, some things will work differently or produce strange results when called in a previous version. In order to make your code safe for all versions, you need to be able to detect what version of the interpreter you are using and adapt to them. In this tip, we'll explore just how to do that.

For this example, we will take advantage of the fact that the integer division changed between Python 2 and Python 3. In the earlier version, integer division returned the truncated result. For example, dividing one into two gave you a result of zero, since the result is a fraction less than one. Dividing three by two gave you one, since the actual result was one and a half. In Python 3, however, the result was exactly what you would expect, a half in the first case and one and a half in the second case. In order to get the same result in both cases, you needed to cast your integers to floating point numbers. How do you know, though, to do this? You use the `sys.version` information:

```
import sys

print(sys.version_info)
i_1 = 3
i_2 = 2
result = 0
if sys.version_info.major != 3:
   result = i_1 / i_2
else:
   result = i_1 / i_2

print(result)
```

If you run the above code in version 3, you'd see 1.5 as the output. If you ran the above code in Python 2. you'd see `1.0`. This isn't really a safe change especially if you are checking the result. So, to be safe, we'd modify this code to be:

```
if sys.version_info.major != 3:
   result = float(i_1) / float(i_2)
else:
   result = i_1 / i_2
```

As a side note, if the minor version, such as the 7 in Python 3.7 or the 4 in Python 3.4 matters to you, because of a package you are using, you can check the `sys.version_info.minor` setting to know which one you are using.

Using the _ (underscore) operator

The underscore (`_`) operator in Python is used when you need a placeholder for something, but don't really care about what it gets filled in with. There are many examples of why you might want to use the underscore, such as not cluttering up your code with variables that are never used, or avoiding warnings from some interpreters and compilers about unused variables. One of the most common reasons to use it, and one you should become accustomed to seeing is in a loop that processes multiple return values. Imagine you had code that did something like this:

```
def function_that_returns_multiple_values(x):

   return x*2, x*3, x+1

for i in range(0,5):
   square, cube, added_one = function_that_returns_multiple_values(i)
   print(square, cube)
```

Obviously, this code calls some function that returns three values for whatever reason. We, however, only care about two of them. So, we have this variable called `added_one` hanging out there for no purpose.

We can re-write this snippet this way, to avoid that added variable:

```
def function_that_returns_multiple_values(x):

   return x*2, x*3, x+1

for i in range(0,5):
   square, cube, _ = function_that_returns_multiple_values(i)
  print(square, cube)
```

The code still works, and still does what it used to, but no longer has that messy variable name that never gets used again. You might ask, if you are only using a single piece of the returned variable list, can you use the underscore multiple times? The answer is absolutely.

```
for i in range(0,5):
   square, _, _ = function_that_returns_multiple_values(i)
```

This is seen in a lot of existing Python code out there, so knowing what it is will help you in a code interview or in a review of someone else's application source.

Discovering where a module is imported from

Although it is rarely needed when actually writing code, it can be very useful to be able to track down where a given function or module is being imported from when you are debugging applications. As this information often isn't needed until the program is written, it can be important to be able to document and list where each piece of your code comes from. This helps you to find out if you are importing the wrong version of something, or if the module you have imported has changed over time. Fortunately, since Python needs that information as well, it is very easy to get at it for your own purposes.

For example, consider the idea of importing a piece of the requests package that we looked at in the last chapter. We can see where it is being used by writing code like this:

```
import requests

print( __name__ )

import sys
d = sys.modules['requests']
print(d.__file__)
```

This will tell us the name of the current module (`__name__`) and then will look up the requests module in the system modules list. This tells us exactly which one is used, since it is the one that the Python interpreter will use. From there, we can print out the actual file that contains the package. Note that since we are looking at the package as whole, the name will be the `__init__` file that is set up for the package as we've done in the past.

What if you wanted to print out the same information for your own functions? Support that you know that a function was imported from somewhere, but you

aren't sure which place? This can happen when you have name collision, and want to be sure you have the right version. Let's create a simple utility function that does this for you:

```
def print_information(a_func):
   print(a_func.__name__)
  name = a_func.__module__
 if name == '__main__': # Need to get the actual module name
f = sys.modules[a_func.__module__].__file__
print(f)


print_information(print_information)
```

As you can see, we have to check to see if the function is being called from the main function in Python, since if it is, the module will always be called `__main__`. If so, we find the module for the function itself and print that out.

Put these two together, and you will see something like the following output:

```
__main__
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/requests/__init__.py
print_information
where_module_imported.py
```

The last lines may vary slightly depending on what you called your module and what version of Python you have installed.

Swapping two values without an intermediate temporary

It seems like every book on programming has to contain a way to swap two values without using an intermediary temporary variable. This seems silly, since it isn't really the sort of thing you would ever do in production code, but someone out there is going to ask for it in an interview for Python, and every other language.

You might think you could do something like this:

```
i = 1
j = 2

print(i,j)
```

```
i = i + j
j = i - j
i = i - j
print(i,j)
```

This certainly works, as long as the two variables happen to be integers, or floating point, values. However, if you assign `Hello world` to the first variable and `Goodbye cruel world` to the second, you'll see a very different output:

```
Traceback (most recent call last):
  File "swap.py", line 7, in <module>
    j = i - j
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

We can't subtract strings, so this doesn't work. How can we implement a way to swap two values regardless of the types of the values? Well, when you think about it, tuples will do this for us. We can swap the elements in a tuple easily enough, and even wrap it up in a simple function that will work for any data type:

```
def swap(a, b):
   return b,a


i = "Hello world"
j = "Goodbye cruel world"
i, j = swap(i, j)
print(i)
print(j)
i = 1
j = 2
i, j = swap(i,j)
print(i)
print(j)
```

The output from this snippet is:

```
Goodbye cruel world
Hello world
2
1
```

Show that off at your next Python interview, and you'll find that not only will it work and get you a good review, but also it will show that you understand much more about Python than the average programmer!

Using the classmethod decorator to create static methods

One of the biggest problems that any professional developer has when moving from one language to another is to cope with missing features, or features that work remarkably differently from their old language. If you are accustomed to C++ and use the static keyword when working with class methods so that you can use them as `utility` methods, you may bemoan the fact that Python appears to have no such keyword. It is true that Python has no `static` keyword, but that doesn't mean you can't create `static` class methods in Python. In fact, depending on how you wish to do it, there are two ways to do it, both using decorators.

Let's look at how you use the decorators, and then we'll take a moment to explain the differences between the two:

```
class Name:

   def __init__(self, first, middle, last):
     self._name = first + ' ' + middle + ' ' + last

   def name(self):
     return self._name

   @classmethod
  def from_first_and_last(self, first, last):
   self._name = first + ' ' + last
 return self._name

   @staticmethod
  def from_first_and_middle(first, middle):
    return first + ' ' + middle + '.'
```

We can call the different types of methods like this:

```
n1 = Name('matt', 'a', 'smith')
print('Standard: ' + n1.name())
print('Classmethod with Class: ' +Name.from_first_and_last('matt', 'smith'))
print('Class with object: ' +n1.from_first_and_last('matt', 'othersmith'))
```

```
print('Object: ' +n1.name())
print('Static with class: ' +Name.from_first_and_middle('matt', 't'))
print('Static with object: ' +n1.from_first_and_middle('matt', 't'))
print('Object: ' +n1.name())
```

The `@classmethod` decorator makes the method a class-level method. That is, it is called with the class (or object, depending on how it is invoked) as the first argument. When called with a class as the first argument, such as `Name.from_first_and_last`, it acts as if it were working on the class itself, so that setting attributes in the class level. These attributes can be overridden at the instance level, so the class method that sets the `_name` attribute is actually setting the class variable, which won't be the one returned by instance. To understand this, look at the following code:

```
n2 = Name('a', 'b', 'c')
n2.from_first_and_last('ralph', 'jones')
n3 = Name('a', 'b', 'c')
print(n2.name())
print(n3.name())
```

You would expect the `n3` variable to contain the name `a b c` and it does, since that is the normal way that it is set. However, you would think that the `n2` variable would print out `ralph jones`, but it does not. If you look at the `n2` variable in the debugger (or using `print`) you will see that it does contain `a b c` as its `_name` attribute.

On the other hand, if you do this:

```
print(Name._name)
```

The output here is:

```
ralph jones
```

Static methods, on the other hand, do not take an instance or a class argument. A static method is best used as a pure utility method, since you can't change instance variables with them. Class methods, on the other hand, are best used for the class as a whole.

Using the **kwargs to pass a named list of parameters

In previous chapters, we've looked at the keyword argument method of accepting arguments into a function, so that a caller can provide arguments in any order they would like to. For example:

```
import decimal
```

```
def function_with_kwargs(**kwargs):
   for k, v in kwargs.items():
     print("Key {0} = Value {1}".format(k,v))

function_with_kwargs(arg_1 = 10, arg_2 = 'Hello', arg_3 = decimal.
Decimal(1))
```

In this case, the user can provide the three arguments in any order they like so long as the names match up to what the function is expecting. In this case, we don't even care about that, we just print out whatever we find.

Suppose, however, you had a function that looked like this:

```
def function_without_kwargs(arg1, arg2, arg3):

   print("Arg1 = {0}".format(arg1))
  print("Arg2 = {0}".format(arg2))
 print("Arg3 = {0}".format(arg3))

function_without_kwargs(1,2,3)
```

Obviously, calling the function with the proper arguments in the proper order works just fine. Consider for a moment, however, a case where the programmer using your function only knows the names of the arguments, and doesn't know, or care, about the order of them in the function prototype. Is there some way we can help that poor programmer to call things properly anyway? In fact, Python does allow for such a thing:

```
def build_dict(arg1, arg2, arg3):
   return {
     'arg1': arg1,
   'arg2': arg2,
 'arg3': arg3
}

function_without_kwargs(**build_dict(1,2,3))
```

In this case, the `build_dict` function is provided just to show that we are probably doing this from something that knows nothing about your function in the first place. It simply produces a dictionary that is contains the keys which represents the arguments and values that go with them to be passed to the function.

There is one caution with this approach, however. If we had a dictionary with all of our possible arguments and values and passed it to the function, we would get any error:

```
def build_dict(arg1, arg2, arg3):
    return {
        'arg1': arg1,
        'arg2': arg2,
        'arg3': arg3,
        'arg4': 45
    }

function_without_kwargs(**build_dict(1,2,3))
```

```
TypeError: function_without_kwargs() got an unexpected keyword argument
'arg4'
```

We could use what we have learned in introspection and reflection to figure out just how many arguments we need and build the dictionary properly. Try to implement such a thing, and you'll learn just how easy it is.

Type hints

One of the nicest things about compiled languages is that they have well defined types for functions or methods return values, as well as the individual parameters that are sent to those functions or methods. For example, in C++, if we create a function

```
int double_x(int x) {
    return x * 2;
}
```

And try to call that function with:

```
double_x("hello world")
```

We will get a compile time error indicating that the argument x is supposed to be an integer and that we can't do that. In Python, of course, we could easily do the exact same thing. That's okay, most of the time, and is the power of the language. However, in complex functions or methods, it isn't always clear what type the programmer that implemented the code intended something to be. What is often worse is that a returned value isn't defined as well, so we could be getting back just about anything. If you don't know, you normally fall back upon the documentation or, if there is nothing there, to finding the source code to the function. Barring all

that, the developer is forced to rely on the debugger, checking the actual type of the returned value to see what they got and what they can do with that.

The Python developers didn't want to lose the ability to be dynamic and allow the system to coerce things into the proper type when needed, but they also recognized that the documentation issue is a serious one. For this reason, Python 3.7 added *type hints*. Type hints are exactly what they sound like, hints as to the type of things. You can use type hints for parameters to a function or method, as well as for the return type for that method.

Let's consider a very simple function that builds a greeting string for the user. The code looks like this:

```
def say_hello_to_user(user: str) -> str:
   say_hello = "Hi there " + user
    return say_hello
```

The *type hints* here indicate that they '`user`' parameter is a string, by including the type of the variable after a colon and the variable name in the function definition. The -> construct indicates the return type of the function or method.

This does two things. For one thing, when you request help, using the built-in `help()` function in Python, you will get a useful message indicating what the function accepts and returns:

```
Help on function say_hello_to_user in module __main__:
say_hello_to_user(user: str) -> str
```

In addition, the `__annotations__` attribute of the function contains the information as well:

```
{'user': <class 'str'>, 'return': <class 'str'>}
```

What is important here is that the type hints do not restrict you from calling the function or method incorrectly. You can still call it with a non-string value and you will get an error. There are other interpreters/compilers such as `mypy` which will actually do the validation and check the inputs for you, but that's a separate thing. One last thing worth mentioning here, if you need to set a default value for your parameter, it needs to follow the type hint:

```
def say_hello_to_user(user: str = "guest") -> str:
   say_hello = "Hi there " + user
  return say_hello

print(say_hello_to_user('Matt'))
print(say_hello_to_user())
```

```
Hi there Matt
Hi there guest
```

Finding the day of the week using the calendar module

Here's a tip that you are certain to use. Sometimes, you need to know what day it is. Not because you've been working for a long time and have forgotten what day of the week it is, but because you have to display the current day of the week, Monday, Tuesday, and many more on a report or screen. You would think this sort of thing would be included in every date library and package ever created, but it sadly is not. We can use the calendar module from Python to get the information we want, however.

```
from datetime import datetime
import calendar


dt = datetime.today()
dn = dt.weekday()
print("Today is day number: {0}".format(dn))
print("Today is a {0}".format(calendar.day_name[dn]))
```

This works and produces the proper day of the week, as well as the number of the date. It does not, however, feel terribly Pythonesque. Let's spiff it up a little bit, make it into a reusable function that has rational defaults, and produces the same data:

```
def DayOfWeek(d=None):
   if d == None:
     d = datetime.today()
   dn = d.weekday()
  return (dn, calendar.day_name[dn])


print(DayOfWeek())
d2 = datetime.today() + timedelta(1)
print(DayOfWeek(d2))
```

As you can see, this version not only properly calculates the day of the week as a number and a `string`, but also returns them as a `tuple` that can be used for whatever purpose the user wanted. You can easily include this in a `utility` library and never worry about it again.

One note here, the `timedelta` class is used to add or subtract time periods from a given date object. The `timedelta` allows you to add days, months, years and all manner of time values. By default, all of the parameters to the method are zero, but you can override whatever pieces you want. The first argument is days, so we are adding one day to day, producing tomorrow as a date.

Working with regular expressions

If there is a single area of computer software development that absolute divides developers, it would be regular expressions. For some, they are a godsend. For others, they are the devil incarnate. Whether you are in the former camp or the latter, the reality is that sooner or later you are going to end up either using one yourself, or finding one in an application code base that you have to maintain. Python has an excellent regular expression class and library, and supports all standard expression types for matching and searching.

Regular expressions have three basic uses in your code. First, you can match specific patterns in a string, returning all of the substrings that have a match to your expression. Second, you can search for a specific string pattern, and find out where it exists within the string. Finally, you can search for and replace a string. Let's look at some very simple examples to help you understand how you can use **regular expressions (RE)** in your code.

If you have ever used Cucumber or a similar testing product, you know that test cases are written in pseudo-English. The engine uses a matching system to determine whether or not something should be executed within the code and match it to the code that needs to be run. Suppose that you wanted to implement such a thing in your own code, or simply wanted to execute user commands of a given format. Here's how you use RE to accomplish this:

```
import re

line = "Given that there is a user logged in"

matches = re.match( r'^Given', line, re.M | re.I )
if matches:
   print(matches.group(0))
else:
   print("No matches")
```

Notice that we use the re match function to check our input line. We are looking for all lines that begin with `Given` since that's the start of a test case in `Cucumber`. The carat `(^)` matches a string beginning on a line. So this will find us all matches that

begin with `Given`. The `re.M` flag indicates that we will allow multiple lines and that any line in the string starting with the requested pattern will match. The `re.I` flag indicates that we ignore case, so it will match `Given`, `given`, or even `gIvEn` and return the match. The return from the match function is a match object, which contains the groups of characters that fit our requested pattern. Match is usually used to find specific string patterns within blocks of text.

The next possibility is that you just want to know if a given pattern exists within a string. You don't care if there is one or more of them, simply that there is one or there isn't. For this, there is the search function. The search function is used this way:

```
# See if the match line contains an account
matches = re.search(r'account', line, re.M | re.I)
if matches:
   print("An account search")
else:
   print("Not an account search")
```

Once again, you can use any regular expression characters in the first argument, which is the pattern to match. Search doesn't modify the string, nor does it return any other matches in the string.

The real purpose to the regular expression module, of course, is to match things that don't simply begin with a string or end with a string, but contain some set of characters that we are interested in. For example, let's imagine that we have a code that is embedded in a string. We know that the code looks like a lower-case letter followed by an upper case letter and then a number. We would like to extract these codes from the string. We can easily do this with regular expressions:

```
def match_lower_upper_number(s):
   groups = re.match(r'[a-z][A-Z][0-9]', s)
 return groups


print(match_lower_upper_number('aA0'))
print(match_lower_upper_number('Ba0'))
print(match_lower_upper_number('aA-'))
```

The `[]` expression indicates a character set that we are looking to match. This can be a range of characters like `a-z`, indicating the lower case alphabet or `A-Z` which is the upper case alphabet. It can be the number set `0-9`. In addition, there are shortcuts for most of these elements. For example, matching a single digit is represented by `\D` which can replace `[0-9]`. For a complete list of the sets, please refer to the `RE`

module documentation. You can also match multiple characters using the * symbol and a single character of any sort using the period '.' symbol.

For our tip for this section consider the concept of validating a phone number. Phone numbers can be difficult, since they can be in a lot of formats. You can have `xxx-xxx-xxxx` or `(xxx) xxx-xxxx` or even `xxxxxxx` for your inputs. It is vastly easier to validate a phone number if you are simply looking at the digits that make it up. You have either seven or ten digits, it could be a phone number and you can apply the rules that apply. But how do we get rid of all the other stuff around it? The answer, of course, lies in the power of regular expressions. Let's write a function that just strips out anything that isn't a number, and returns the result to us as a string:

```
phone_no_1 = "303-555-1212"
phone_no_2 = "3035551212"
phone_no_3 = "(303) 555-1212"

def strip_all_but_numbers(s):
  ret = re.sub(r'\D', '', s)
 return ret

print(strip_all_but_numbers(phone_no_1))
print(strip_all_but_numbers(phone_no_2))
print(strip_all_but_numbers(phone_no_3))
```

The output from this, for all three strange phone number strings, is:

```
3035551212
3035551212
3035551212
```

As you can see, the power of regular expressions is vast, but you can use only the pieces you need or want.

That completes our tips and tricks for Python. Hopefully, you enjoyed the list, and found a few that you can use in your own coding. Remember, Python is all about not reinventing the wheel. If someone has already solved your problem, why not use that solution? Imitation, after all, is the sincerest form of flattery!

Good luck and happy Pythoning!

Conclusion

In this chapter, we learned a bunch of nifty tricks and tips that can be used in professional programs. You learned how to determine the largest pieces of your

code so that you can optimize it. You learned about the regular expression package in Python and how it can be used to make sure the values input from the user are valid. You also learned about how to get the day of the week, something that is guaranteed to be needed in any program that does reporting.

Questions

1. What are some methods for removing duplicates from a list?
2. What class contains the date and time information needed to find out the day of the week?
3. What is the underscore operator and why would you use it?
4. What are keyword arguments and when can you use them?