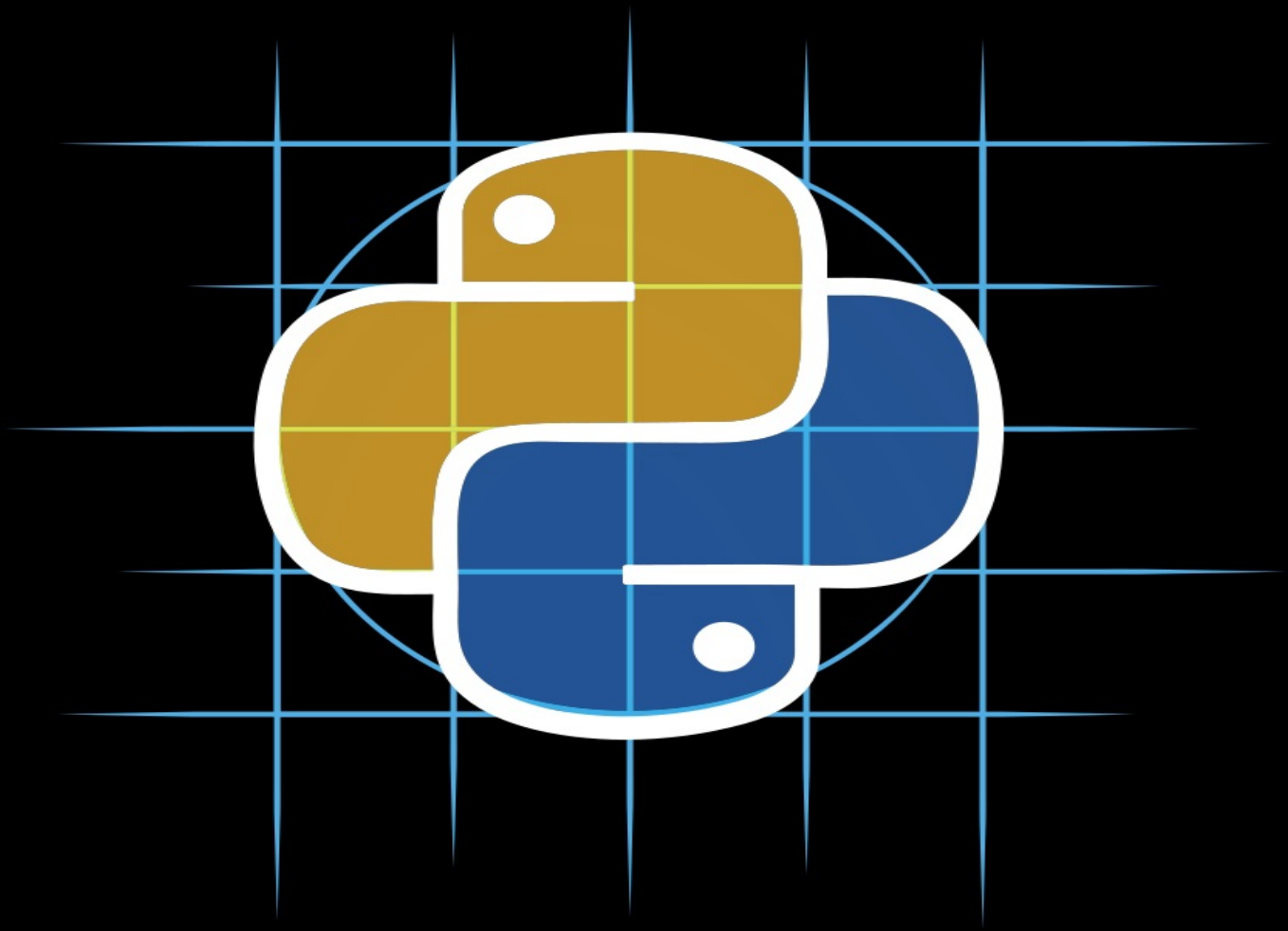
Treading on Python Series



ILLUSTRATED GUIDE TO PYTHON 3

A complete walkthrough of beginning Python with unique illustrations showing how Python really works

Matt Harrison

Treading on Python Series

Illustrated Guide to Python 3

A Complete Walkthrough of Beginning Python with Unique Illustrations Showing how Python Really Works

Matt Harrison

Technical Editors: Roger A. Davidson, Andrew McLaughlin

Version: Oct 2, 2017

Introduction

Are you ready to jumpstart your Python programming career? This book will arm you with years of knowledge and experience that are condensed into an easy to follow format. Rather than taking months reading blogs, websites, and searching mailing lists and groups, this book will allow a programmer to quickly become knowledgeable and comfortable with Python.

Programming is fun and Python makes it delightful. Basic Python is not only easy, but approachable for all ages. I have taught elementary students, teenagers, “industry” professionals and “golden years” folks the Python programming language. If you are willing to read and type, you are about to begin an exciting path. Where it ultimately takes you depends on how hard you are willing to work.

There are different levels of Python. Basic Python syntax is small and easy to learn. Once you master basic Python, doors will open to you. You should be able to read a lot of Python and understand it. From there, you can learn more advanced topics, specific toolkits, start contributing to open source projects written in Python, or use that knowledge to learn other programming languages.

A recommended approach for learning the language is to read a chapter and then sit down at a computer and type out some of the examples found in the chapter. Python makes it easy to get started, eliminating much of the hassle found in running programs in other languages. The temptation will

likely be to merely read the book. By jumping in and actually typing the examples you will learn a lot more than by just reading.

Why Python?

PYTHON IS BOOMING! IT IS THE TOP LANGUAGE BEING TAUGHT IN UNIVERSITIES. Python developers are among the highest paid. With the boom in data science, Python is quickly becoming one of the most desired skills for analytics. Operations are also adopting Python to manage their backend systems. They are discovering what web developers using Python have known for a long time; namely, that Python will make you productive.

Python has crossed the tipping point. No longer are only small, agile startups relying on it. Looking to take advantage of its power and efficiency, enterprises have also converged on Python. Over the past year, I've taught Python to hundreds of seasoned developers with years of experience at large companies, who are moving to Python.

Python enables increased productivity. I came to Python as a Perl programmer. At work, I was assigned to a team with a co-worker who was well versed in Tcl. Neither of us wanted to jump ship though both of us were interested in learning Python. In 3 days our prototype was completed, much faster than we expected, and we both promptly forgot our previous “goto” language. What appealed to me about Python was that it fit my brain. I firmly believe if you have some experience in programming, you can learn the basics of Python in a few days.

Python is easy to learn. For beginning programmers, Python is a great stepping stone. Learning to write simple programs is pretty easy, yet Python also scales up to complex “enterprise” systems. Python also scales with age

—I have personally seen people from 7-80+ learn basic programming skills using Python.

Which Version of Python?

This book will focus on Python 3. Python 2 has served us well over the years. The Python Software Foundation, which manages releases of the language, has stated that Python 2 is coming to an end. As such, after 2020 they will no longer support the language.

Python 3, has been out for a bit now and is somewhat backward incompatible with the 2 series. For green field development, you should move forward with Python 3. If you have legacy systems on Python 2, do not fret. In fact, most of the content in this book is perfectly applicable to Python 2. If you want to focus on Python 2, check out the prior version of this book.

Python installation

Python 3 is not installed on most platforms. Some Linux distributions ship with it, but Windows and Mac users will need to install it.

For Windows folks, go to the download area of the Python website [1] and find a link that says “Python 3.6 Windows Installer”. This will link to a .msi file that will install Python on your Windows machine. Download the file, open it by double-clicking it, and follow the instructions to finish the installation.

NOTE

On the Windows installer there is an option labeled "Add Python to PATH". Please make sure it is checked. That way, when you run `python` from the command prompt, it will know where to find the Python executable. Otherwise, you can go to System properties (click WIN+Pause, or run `environ` from the start menu), Advanced system settings, and click on the Environment Variables button. There you can update the `PATH` variable by adding the following:

```
C:\Program Files\Python 3.6;C:\Program Files\Python 3.6\Scripts
```

If you have UAC (User Account Control) enabled on Windows, then path is:

```
C:\Users\<username>\AppData\Local\Programs\Python\Python36
```

Likewise, Mac users should download the Mac installer from the Python website.

NOTE

Another option for installing Python is to use the Anaconda [2] distribution. This runs on Windows, Mac, and Linux, and also provides many pre-built binaries for doing scientific calculations. Traditionally, these libraries have been annoying to install as they wrap libraries written in C and Fortran, that require some setup for compiling.

Mac users might also want to look into the Homebrew version [3]. If you are already familiar with Homebrew, it is a simple `brew install python3` away.

Which editor?

In addition to installing Python, you will need a text editor. An editor is a tool for writing code. A skilled craftsman will invest the time to learn to use their tool appropriately and it will pay dividends. Learning to use the features of an editor can make churning out code easier. Many modern editors today have some semblance of support for Python.

If you are just beginning with Python and have not had much experience with real text editors, most Python installations include IDLE, which has decent Python editing features. The IDLE development environment also runs on Windows, Mac and Linux.

A feature to look for in editors is integration with the Python REPL [4]. Later you will see an example with IDLE. Hopefully your editor of choice will have similar features.

Popular editors with decent Python support include Emacs, Vim, Atom, Visual Studio Code, and Sublime Text. If you are interested in more fancy editors that have support for refactoring tools and intelligent completion, PyCharm and Wing IDE are also popular.

Summary

Python 3 is the current version of Python. Unless you are working on legacy code, you should favor using this version. You can find the latest version on the Python website.

Most modern editors contain some support for Python. There are various levels of features that editors and IDEs provide. If you are getting started programming, give the IDLE editor a try. It is a great place to start out.

Exercises

1. Install Python 3 on your computer. Make sure you can start Python.

2. If you are used to a particular editor, do some investigation into its support for Python. For example, does it:

    - Do syntax highlighting of Python code?
    - Run Python code in a REPL for you?
    - Provide a debugger for stepping through your Python code?

1 - https://www.python.org/download

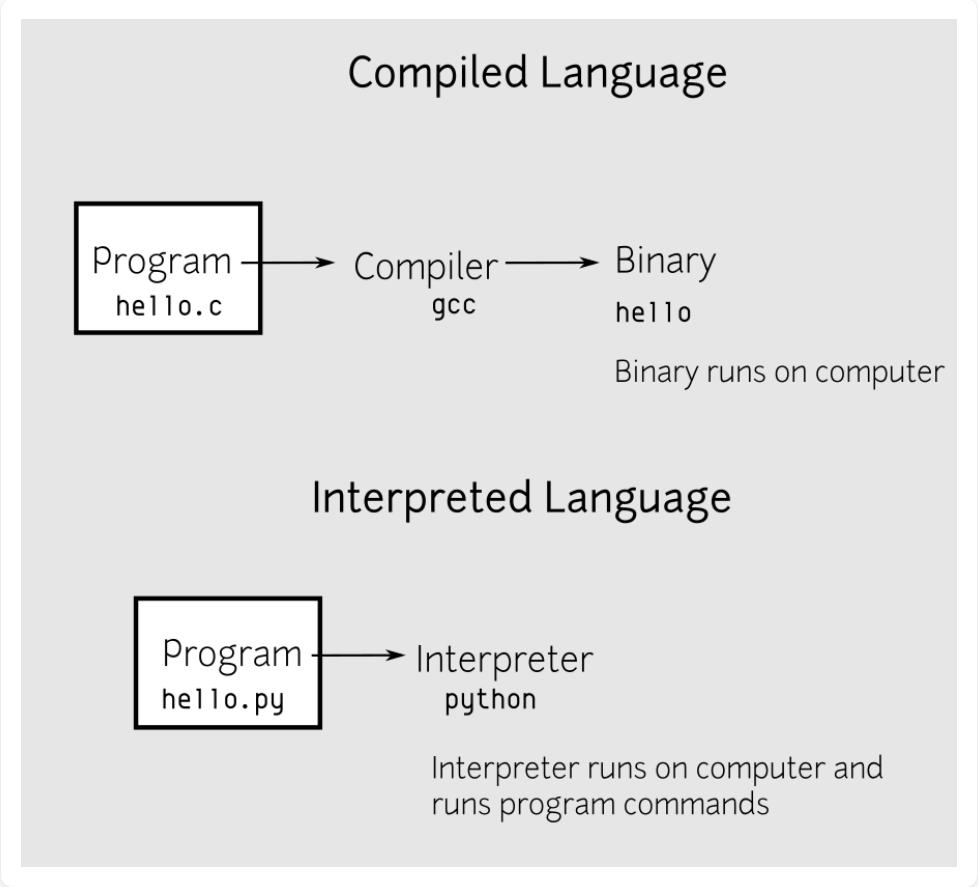
2 - https://www.anaconda.com/download/

3 - https://brew.sh/

4 - REPL stands for Read, Evaluate, Print, and Loop. You will soon see an example using it.

The Interpreter

PYTHON IS COMMONLY CLASSIFIED AS AN *INTERPRETED* LANGUAGE. ANOTHER TERM used to describe an interpreted language is *scripting* language. To run a computer program on the CPU, the program must be in a format that the CPU understands, namely *machine code*. Interpreted languages do not *compile* directly to machine code, instead, there is a layer above, an *interpreter* that performs this function.



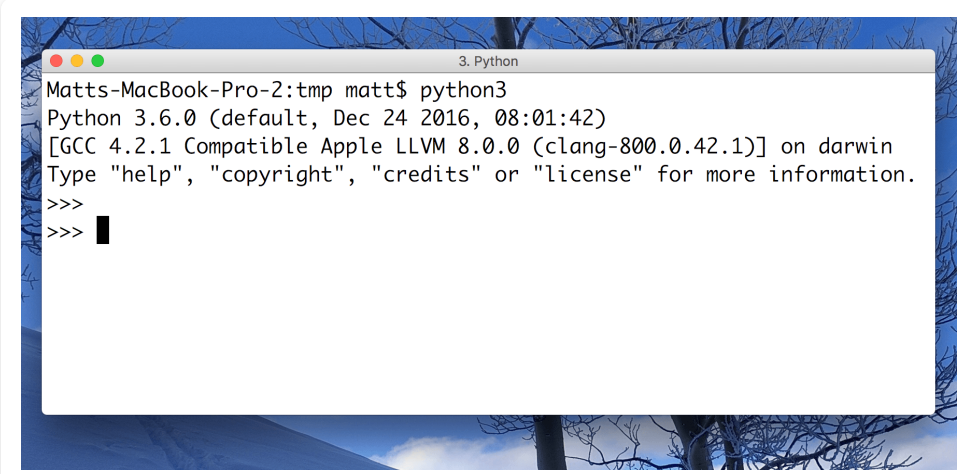
Difference between a compiled language and an interpreted language. A compiler runs to create an executable. An interpreter is an executable that loads code and runs it on top of itself.

There are pros and cons to this approach. As you can imagine, on the fly translating can be time consuming. Interpreted code like Python programs tend to run on the order of 10–100 times slower than C programs. On the flip side, writing code in Python optimizes for developer time. It is not uncommon for a Python program to be 2–10 times shorter than its C equivalent. Also, a compilation step can be time consuming and actually a distraction during development and debugging.

Many developers and companies are willing to accept this trade-off. Smaller programs (read fewer lines of code) take less time to write and are easier to debug. Programmers can be expensive—if you can throw hardware at a problem, it can be cheaper than hiring more programmers. Debugging 10 lines of code is easier than debugging 100 lines of code. Studies have shown that the number of bugs found in code is proportional to the numbers of lines of code. Hence, if a language permits you to write fewer lines of code to achieve a given task, you will likely have fewer bugs. Sometimes program execution speed is not that important and Python is sufficiently fast for many applications. In addition, there are efforts to improve the speed of Python interpreters such as PyPy [5].

REPL

Python has an *interactive interpreter* or *REPL* (Read Evaluate Print Loop). This is a loop that waits until there is input to read in, then evaluates it (interprets it), and prints out the result. When you run the `python3` executable by itself, you launch the interactive interpreter in Python. Other environments, such as IDLE, also embed an interactive interpreter.



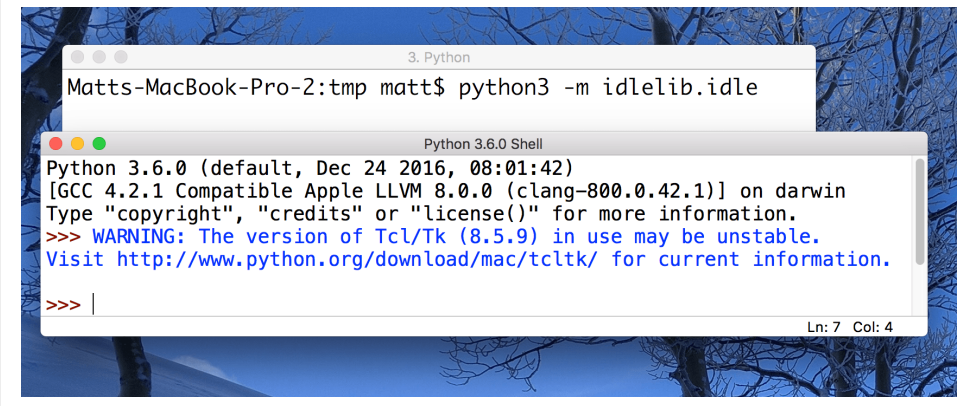
To launch the REPL, type `python3` in the prompt and it will start a Python session

NOTE

This book generally starts Python 3 with the `python3` executable. On Windows, the executable is named `python`. If you are on Windows, replace `python3` with `python`. On Unix systems you shouldn't have to change anything.

When you start the interpreter, it will print out the version of Python, some information about the build, and some hints to type. Finally, the interpreter will give you a prompt, >>>.

Another option is to start IDLE, the editor included with Python, by typing `python3 -m idlelib.idle`.



To launch the REPL in IDLE, click on the IDLE icon or type `python3 -m idlelib.idle`

NOTE

Some Linux distributions do not ship with all of the libraries from the Python standard library. This is annoying, but justified by the idea that a server doesn't need libraries to create client side applications. As such, Ubuntu and Arch (among others) don't provide the gui libraries necessary for IDLE on the default installation.

If you see an error like:

```
$ python3 -m idlelib.idle
** IDLE can't import Tkinter.
Your Python may not be configured for Tk. **
```

it is an indication you are missing the tkinter library.

On Ubuntu, you need to run:

```
$ sudo apt-get install tk-dev
```

On Arch, you need to run:

```
$ sudo pacman -S tk
```

A REPL example

Below is an example to show why the Read, Evaluate, Print Loop was given this name. If you typed `python3` from the command line [6], or launched IDLE, you will see >>>.

Type `2 + 2` like shown below and hit enter:

```
$ python3
>>> 2 + 2
4
>>>
```

In the above example, `python3` was typed, which opened the interpreter. The first >>> could be thought of as the *read* portion. Python is waiting for input. `2 + 2` is typed in, read, and *evaluated*. The result of that expression —4—is *printed*. The second >>> illustrates the *loop*, because the interpreter is waiting for more input.

The REPL, by default, prints the result of an expression to standard out (unless the result is `None`, which will be explained in a later chapter). This behavior is inconsistent with normal Python programs, where the `print` function must be explicitly invoked. But it saves a few keystrokes when in the REPL.

NOTE

The >>> prompt is only used on the first line of each input. If the statement typed into the REPL takes more than one line, the `...` prompt follows:

```
>>> sum([1, 2, 3, 4, 5,
...   6, 7])
```

These prompts are defined in the `sys` module:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
```

A later chapter will explain what modules are. For now, know that there are variables that define what the prompts look like.

The REPL ends up being quite handy. You can use the interactive interpreter to write small functions, to test out code samples, or even to

function as a calculator. Perhaps more interesting is to go the other way. Run your Python code in the REPL. Your code will run, but you will have access to the REPL to inspect the state of your code. (You will see how to do this in IDLE soon).

The `>>>` is a *prompt*. That is where you type your program. Type `print("hello world")` after the `>>>` and hit the enter key. Make sure there are not any spaces or tabs before the word `print`. You should see this:

```
>>> print("hello world")
hello world
```

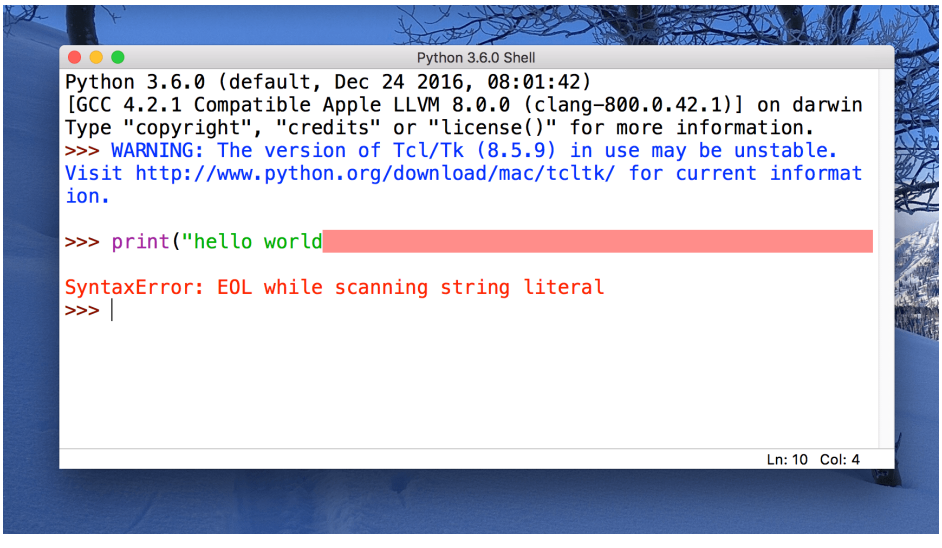
If you see this, congratulations, you are writing Python. Consider yourself inducted into the world of programming. You have just run a program—"hello world". Hello world is the canonical program that most people write when encountering a new language. To exit the REPL from the terminal, type `quit()`. Unix users may also type Ctl-D.

NOTE

Programming requires precision. If you were not careful in typing exactly `print("hello world")` you might have seen something like this:

```
>>> print("hello world
  File "<stdin>", line 1
    print("hello world
                      ^
SyntaxError: EOL while scanning string literal
```

Computers are logical, and if your logic does not make sense, the computer can warn you, perform irrationally (or at least what appears so to you), or stop working. Do not take it personally, but remember that languages have rules, and any code you write has to follow those rules. In the previous example, the rules that states if you want to print text on the screen you need to start and end the text with quotes was violated. A missing quote on the end of the line consequently confused Python.



IDLE will try to highlight where the error occurred. The highlight following `world` is supposed to indicate where the syntax error occurred. It is normally a salmon color.

Summary

As Python is an interpreted language, it provides a REPL. This allows you to explore features of Python interactively. You don't need to write code, compile it, and run it. You can launch a REPL and start trying out code.

For users who have used compiled languages, this might seem somewhat novel. Give it a try though, it can make development easy and quick. Also, don't be afraid to try out code from the REPL. I find that new Python users tend to be timid to use the REPL. Don't fear the REPL!

There are other REPLs for Python. One popular one is Jupyter, which presents a web-based REPL [7]. Start using the REPL, then you can jump into other more advanced REPLs.

Exercises

1. Open the Python 3 REPL and run "hello world". Review the chapter to see what this one line program looks like.
2. Open the REPL IDLE and run "hello world".

5 - https://www.pypy.org

6 - From the Windows run menu, type `cmd` (or Win-R) to get a prompt. On Macs, you can quickly launch a terminal by typing command-space, the typing `Terminal` and return. If you have installed Python 3, you should be able to launch it now on either platform by typing `python3`.

7 - https://jupyter.org/

Running Programs

WHILE USING THE INTERACTIVE INTERPRETER CAN BE USEFUL DURING development, you (and others) will want to deploy your program and *run* it outside of the REPL. In Python, this is easy as well. To run a Python program named `hello.py`, open a terminal, go to the directory containing that program and type:

```
$ python3 hello.py
```

NOTE

When running a command from the command line, this book will precede the command with a `$`. This will distinguish it from interpreter contents (>>> or `...`) and file contents (nothing preceding the content).

NOTE

The previous command, `python3 hello.py`, will probably fail unless you have a file named `hello.py`.

In the previous chapter you used the REPL to run “hello world”, how does one run the same program standalone? Create a file named `hello.py` using your favorite text editor.

In your `hello.py` file type:

```
print("hello world")
```

Save the file, go to its directory, and *execute* the file (here *execute* and *run* have the same meaning, i.e. type `python3` before the file name, and let the Python interpreter evaluate the code for you.)

> **NOTE**
>
> Typing `python3` standalone launches the interpreter. Typing `python3 some_file.py` executes that file.

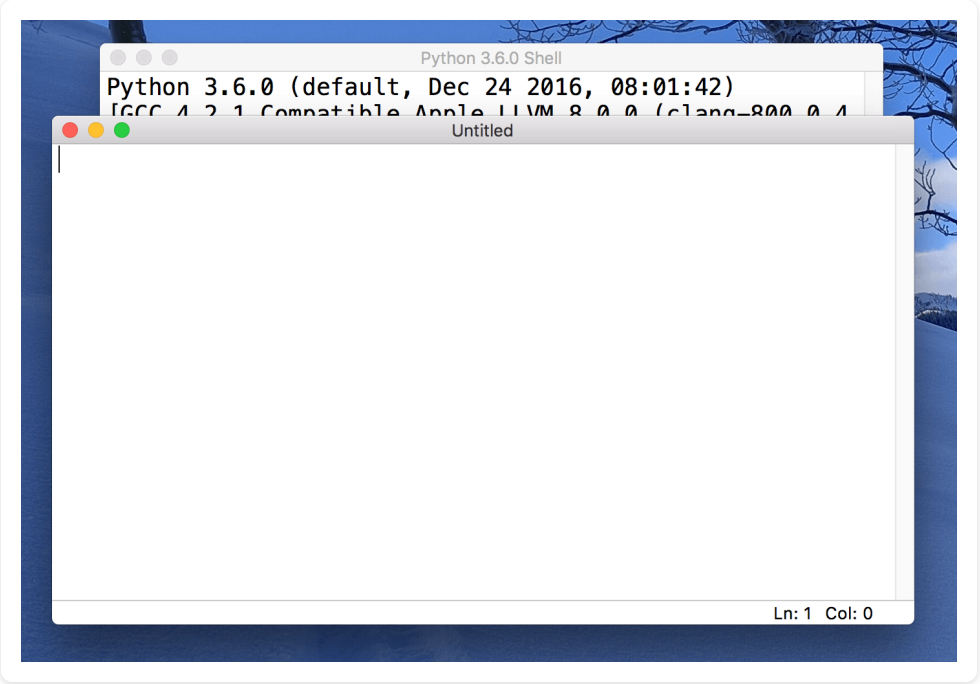
If you were successful at running `hello.py`, it would print out `hello world`.

Running from IDLE

You can also edit and run programs from IDLE. Start by launching IDLE, either by clicking on the application icon on your computer or by typing:

```
$ python3 -m idlelib.idle
```

When IDLE launches, you see the *Shell* window. This is a Python REPL. You can type code and Python will immediately evaluate it. To create a program you need to create a file with Python code in it. To do that, click on the “File” menu, and select “New File”. You should see a new window pop up. This window is not a Shell window, it is an *editor* window. You will notice that it is empty and there is no Python prompt in it. In this window, you will type Python code.

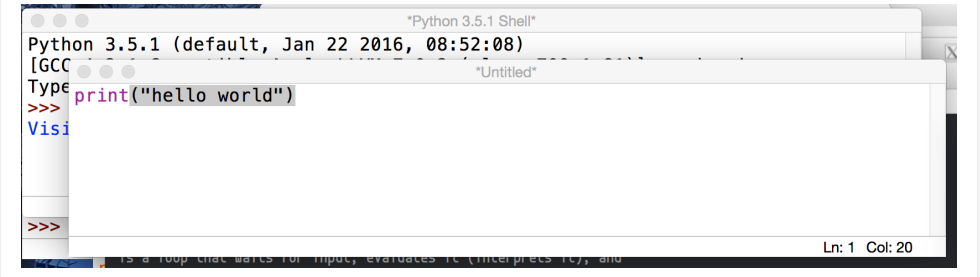


IDLE has been launched and a new editor window (*Untitled* because it has not been saved) has been opened

Type your code into the window. Since you are doing the hello world example, type:

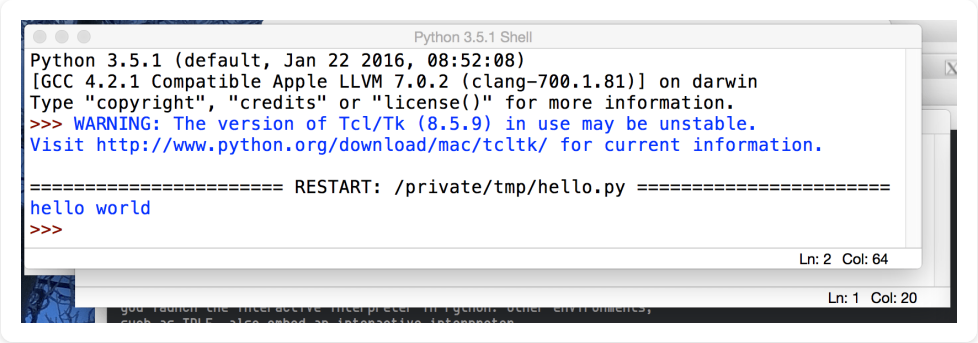
```
print("hello world")
```

You'll notice that IDLE uses a different color for `print` and `"hello world"`. This is called *syntax highlighting* and is meant to aid in understanding your code. Now, you need to run the code. The easiest way to do this is to hit the F5 key on your keyboard. Alternatively, you can click on “Run”, and select “Run Module”. IDLE will ask you to save your file. Save it as `hello.py`. IDLE will bring the shell window to focus, print a line stating that it has restarted the shell, and print out `hello world`.



Code typed into the editor window.

This might seem trivial, but the shell window now has the state of your code. In this case, you only printed to the screen, so there is not much state. In future examples, you will see how you can use the integration between the editor and shell to provide an environment where you can quickly try out code, see the results, and inspect the output of a program.



Result of running Hello World from IDLE

If you need help navigating from the command prompt check out the appendix.

Unixy embellishments

On Unix platforms (Linux and OS X among others), files such as `hello.py` are often referred to as *scripts*. A script is a program, but the term is often used to distinguish native code from interpreted code. In this case, scripts are interpreted code, whereas the output from the compilation step of a language that compiles to machine code (such as C) is *native code*.

NOTE

It is not uncommon to hear about shell scripts, Perl scripts, Python scripts, etc. What is the difference between a Python script and a Python program? Nothing, really it is only semantics. A Python script usually refers to a Python program run from the command line, whereas a Python program is any program written in Python (which run the gamut of small 1-liners to fancy GUI applications, to "enterprise" class services).

Unix environments provide a handy way to make your script executable on it is own. By putting a *hash bang* (`#!`) on the first line of the file, followed by the path to the interpreter, and by changing the *executable bit* on the file, you can create a file that can run itself.

To have the script execute with the Python interpreter found in the environment, update your `hello.py` file to:

```
#!/usr/bin/env python3
print("hello world")
```

NOTE

This new first line tells the shell that executes the file to run the rest of the file with the `#!/usr/bin/env python3` executable. (Shell scripts usually start with `#!/bin/bash` or `#!/bin/sh`.) Save `hello.py` with the new initial line.

Tip

`#!/usr/bin/env` is a handy way to indicate that the first `python3` executable found on your `PATH` environment variable should be used. Because the `python3` executable is located in different places on different platforms, this solution turns out to be cross-platform. Note that Windows ignores this line. Unless you are absolutely certain that you want to run a specific Python version, you should probably use `#!/usr/bin/env`.

Using hardcoded hashbangs such as:

- `#!/bin/python3`
- `#!/usr/bin/python3.3`

might work fine on your machine, but could lead to problems when you try to share your code and others do not have `python3` where you specified it. If you require a specific version of Python, it is common to specify that in the `README` file.

Now you need to make the file executable. Open a terminal, `cd` to the directory containing `hello.py` and make the file executable by typing:

```
$ chmod +x hello.py
```

This sets the *executable bit* on the file. The Unix environment has different permissions (set by flipping a corresponding bit) for reading, writing, and executing a file. If the executable bit is set, the Unix environment will look at the first line and execute it accordingly, when the file is run.

TIP

If you are interested in knowing what the `chmod` command does, use the `man` (manual) command to find out by typing:

```
$ man chmod
```

Now you can execute the file by typing its name in the terminal and hitting enter. Type:

```
$ ./hello.py
```

And your program (or script) should run. Note the `./` included before the name of the program. Normally when you type a command into the terminal, the environment looks for an executable in the `PATH` (an environment variable that defines directories where executables reside). Unless `.` (or the parent directory of `hello.py`) is in your `PATH` variable you need to include `./` before the name (or the full path to the executable). Otherwise, you will get a message like this:

```
$ hello.py
bash: hello.py command not found
```

Yes, all that work just to avoid typing `python3 hello.py`. Why? The main reason is that you want your program to be named `hello` (without the trailing `.py`). And perhaps you want the program on your `PATH` so you can run it at any time. By making a file executable, and adding a hashbang, you can create a file that looks like an ordinary executable. The file will not require a `.py` extension, nor will it need to be explicitly executed with the `python3` command.

Summary

Running Python programs is easy. There is no lengthy compilation step. You need to point Python to the program you would like to run. Many editors also have the ability to run Python code. It is worthwhile to investigate how to do it with your editor. With IDLE, it is simple: you hit F5.

Exercises

1. Create a file `hello.py` with the code from this chapter in it.
2. Run `hello.py` from a terminal.
3. Run `hello.py` from within IDLE.
4. If you have another editor that you prefer, run `hello.py` from it.
5. If you are on a Unix platform, create a file called `hello`. Add the hello world code to it, and make the appropriate adjustments such that you can run the code by typing:

```
./hello
```

Writing and Reading Data

Programs will typically have input and output. This chapter will show how to print values to the screen and allow the end user to type in a value. In Python, both of these are really straightforward.

Simple output

The easiest way to provide the user with output is to use the `print` function, which writes to *standard out*. Standard out is where the computer writes its output. When you are in a terminal, standard out is printed on the terminal

```
>>> print('Hello there')
Hello there
```

If you want to print out multiple items, you can provide them separated by commas. Python will insert a space between them. You can put strings and numbers in a `print` function:

```
>>> print('I am', 10, 'years old')
I am 10 years old
```

A later chapter will look at strings in detail. It will discuss how to format them to get output to look a certain way.

Getting user input

The built-in `input` function will read text from a terminal. This function accepts text which it prints out as a prompt to the screen and waits until the user types something on *standard in* and hits enter. Standard in is where the

computer reads its input. In a terminal, standard input can be read from what you type in:

```
>>> name = input('Enter your name:')
```

If you typed the above into the interpreter (the spaces around the = are not required but convention suggests that you type them to make your code more readable), it might look like your computer is frozen. In reality, Python is waiting for you to type in some input and hit enter. After you type something in and press enter, the variable `name` will hold the value you typed. Type the name `Matt` and press the enter key. If you print `name` it will print the value you just typed:

```
>>> print(name)
Matt
```

NOTE

The value entered into the terminal when `input` is called is always a *string*. If you tried to perform math operations on it, it might not give you the answer you want:

```
>>> value = input('Enter a number:')
3
>>> other = input('Enter another:')
4
```

If you try to add `value` and `other` right now, you concatenate them (or join them together) because they are strings:

```
>>> type(value)
<class 'str'>
>>> value + other
'34'
```

If you want to add these strings as if they were numbers you need to change them from a string into a number type. To convert a string to another type like an integer (a whole number) or float (a decimal number), you will need to use the `int` and `float` functions respectively.

If you want to numerically add `value` and `other`, you have to convert them to numbers using `int`:

```
>>> int(value) + int(other)
7
```

A future chapter will talk more about strings and number types.

Summary

Python gives you two functions that make it really easy to print data out to the screen and read input from the user. These functions are `print` and

`input`. Remember that when you call the `input` function, you will always get a string back.

Exercises

1. Create some Python code that will prompt you to enter your name. Print out `Hello` and then the name.
2. Create a program that will ask a user how old they are. Print out some text telling them how old they will be next year.

Variables

Now that you know about running programs via the interpreter (or the REPL) and the command line, it is time to start learning about programming. *Variables* are the basic building blocks of computer programs.

Variables are important in Python, because in the Python world, everything is an *object*. (This is not quite true, keywords are not objects). Variables allow you to attach names to these objects so you can refer to them in future code.

Mutation and state

Two important programming concepts are *state* and *mutation*. *State* deals with a digital representation of a model. For example, if you want to model a light bulb, you may want to store its current status—is it on or off? Other possibly interesting states you could store include the type of bulb (CFL or incandescent), wattage, size, dimmable, etc.

Mutation deals with changing the state to a new or different state. For the light bulb example, it could be useful to have a power switch that toggles the state and changes it from off to on.

How is this related to variables? Remember that in Python everything is an object. Objects have state, and might be mutated. To keep track of these objects, you use variables.

Once you have objects that hold state and are mutable, then you have opened a world of possibilities. You can model almost anything you want if

you can determine what state it needs, and what actions or mutations need to apply to it.

Python variables are like tags

Variables are the building blocks of keeping track of state. You might think of a variable as a label or tag. Important information is tagged with a variable name. To continue the light bulb example, suppose that you want to remember the state of your light bulb. Having that data is only useful if you have access to it. If you want to access it and keep track of its state you need to have a *variable* to tag that data. Here the state of the bulb is stored in a variable named `status`:

```
>>> status = "off"
```

This requires a little more examination because there is a bit going on there. Starting from the right, there is the word `"off"` surrounded by quotes. This is a *string literal*, or a built-in datatype that Python has special syntax for. The quotes tell Python that this object is a *string*. So Python will create a string object. A string stores textual data—in this case, the letters `off`.

This object has a few properties of interest. First, it has an *id*. You can think of the id as where Python stores this object in memory. It also has a *type*, in this case, a string. Finally, it has a *value*, here the value is `'off'`, because it is a string.

The = sign is the *assignment operator* in many programming languages. Do not be afraid of these technical terms, they are more benign than they appear. The assignment operator connects or binds together a variable name and its object. It indicates that the name on the left of it is a variable that will hold the object on the right. In this case, the variable name is `status`.

When Python creates a variable, it tells the object to increase its *reference count*. When objects have variables or other objects pointing to them, they

have a positive reference count. When variables go away (an example is when you exit a function, variables in that function will go away), the reference count goes down. When this count goes down to zero, the Python interpreter will assume that no one cares about the object anymore and *garbage collects* it. This means it removes it from its memory, so your program doesn't get out of control and use all the memory of your computer.

NOTE

If you want to inspect the reference count of an object, you can call `sys.getrefcount` on it:

```
>>> import sys
>>> names = []
>>> sys.getrefcount(names)
2
```

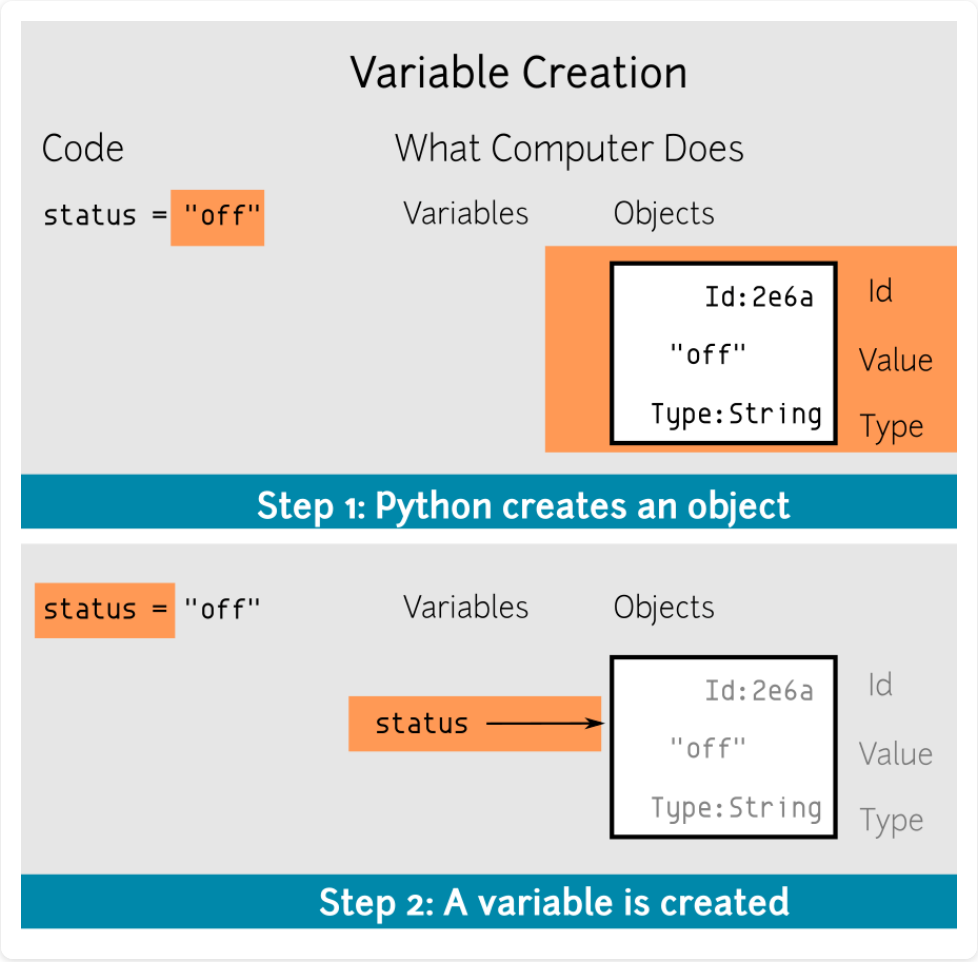
Do note that as this count may seem high, the documentation for this function states:

```
Return the reference count of object.  The count returned
is generally one higher than you might expect, because it
includes the (temporary) reference as an argument to
getrefcount().
```

Even though Python gives you this ability, typically you don't worry about the reference count and let Python handle cleaning up objects for us.

This is a feature of Python, and typically Python does this for you automatically, without any prompting from a user. In other languages, you need to manually tell the program to allocate and deallocate memory.

To drive the point home, reading the code again from the left this time. `status` is a variable that is assigned to the object that Python created for us. This object has a type, string, and holds the value of `"off"`.



Two steps of an assignment to a literal. First, Python creates an object. The object has a value, `"off"`, a type, string, and an id (the location of the object in memory). After the object is created, Python looks for any variable named `status`. If it exists, Python updates what object the variable is pointing to, otherwise, Python creates the variable and points it to the object.

Cattle tags

My grandfather had a cattle ranch, so I will pull out a non-nerdy analogy. If you have a cattle ranch of any size, it makes sense to have a good foreman who will keep track of your cattle (your investment).

One way to keep track of cattle is to use cattle tags. These small tags attached to the ear of a cow can be used to identify and track individual

cows.

To pull this back to programming, rather than running a cattle ranch, you are managing aspects of your program. A program can hold many distinct pieces of information that you want to remember or keep track of. This information is the *state*. For example, if you needed to track data pertinent to humans you might want to track age, address, and name.

Just like how ranchers tag their cattle to keep track of them, programmers create variables to keep track of data. Look at the example again:

```
>>> status = "off"
```

This tells Python to create a *string* with the contents of `off`. Create a variable named `status`, and attach it to that string. Later on, when you need to know what the status is, you can ask your program to print it out like so:

```
>>> print(status)
off
```

It is entirely possible to create *state* and lose it to the ether if you neglect to put it in a variable. It is somewhat useless to create objects that you would not use, but again it is possible. Suppose you want to keep track of the bulb's wattage. If you write:

```
>>> "120 watt"
```

It tells Python to create a string object with the content of `120 watt`. This is problematic because you forgot to assign it to a variable. Now, you have no way of using this object. Python will only let you access data that is stored in variables, so it is impossible for you to use this item now. Objects are accessed by using their variable names. If this was information that you needed in your program, a better solution would be the following:

```
>>> wattage = "120 watt"
```

Later on, in your program you can access `wattage`, you can print it out, and you can even assign another variable to it, or assign `wattage` to another new value (say if your incandescent bulb broke and you replaced it with an LED bulb):

```
>>> incandescent = wattage
>>> wattage = "25 watt"
>>> print(incandescent, wattage)
120 watt 25 watt
```

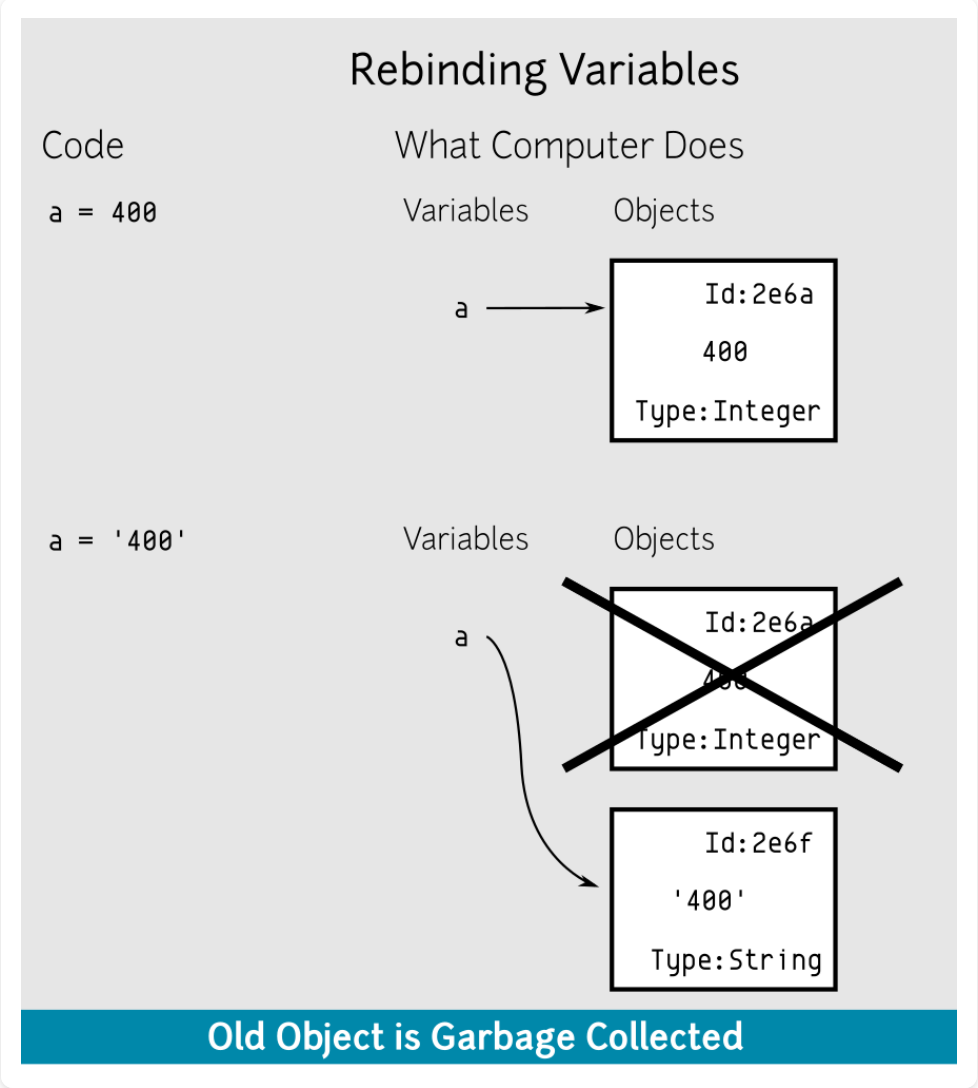
Managing state is a core aspect of programming. Variables are one mechanism to manage it.

Rebinding variables

Much like cow tags, variables tend to stay with an object for a while, but they are transferable. Python lets you easily change the variable:

```
>>> num = 400
>>> num = '400'  # now num is a string
```

In the above example `num` was originally pointing to an integer but then was told to point to a string.



This illustrates rebinding variables. Variables can be rebound to any type. Python makes no effort to prevent this or complain. When an object no longer has any variable pointing to it, it is cleaned up by Python, or garbage collected.

NOTE

The variable does not care about the type. In Python, the type is attached to the object.

There is no limit to how often you can change a variable. But you should be careful not to change a variable if you still need access to the old data. Once you remove all variables from an object, you are essentially telling

Python to destroy (*garbage collect* is the proper geeky term) the object when it has the chance, to free up any internal memory it occupies.

TIP

This is a case where Python allows you to do something, but you probably don't want to do it in real life. Just because you can rebind a variable to a different type, doesn't mean you should. Changing the type of a variable is confusing to you when you read your code later. It is also confusing to others who are using your code. Don't use the same variable to point to different types.

This is a point of confusion for those who are new to Python. They sometimes reuse the same variable throughout their code because they mistakenly believe that it will save memory. As you have seen this is not the case. The variable itself is very lightweight. The object is what uses memory. Reusing a variable is not going to change how memory on the object is handled, but it will be confusing to those who have to read the code later.

Naming variables

Python is somewhat particular about naming variables. It has conventions that most Python programmers follow. Some of these are enforced, some are not. One that is enforced by the interpreter is *a variable should not have the name of a keyword*. The word `break` is a keyword and hence cannot be used as a variable. You will get a `SyntaxError` if you try to use it as a variable. Even though this code looks perfectly legal, Python will complain:

```
>>> break = 'foo'
  File "<stdin>", line 1
    break = 'foo'
```

```
          ^
SyntaxError: invalid syntax
```

If you find yourself with a `SyntaxError` that looks like normal Python code, check that the variable name is not a keyword.

NOTE

Keywords are reserved for use in Python language constructs, so it confuses Python if you try to make them variables.

The module `keyword` has a `kwlist` attribute, which is a list containing all the current keywords for Python:

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert',
'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'finally', 'for', 'from', 'global',
'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try',
'while', 'with', 'yield']
```

Another method for examining keywords in the REPL is to run `help()`. This puts you in a help utility in the REPL, from which you can type commands (that aren't Python). Then type `keywords` and hit enter. You can type any of the keywords and it will give you some documentation and related help topics. To exit the help utility hit enter by itself.

Additional naming considerations

In addition to the aforementioned rule about not naming variables after keywords, there are a few best practices encouraged by the Python community. The rules are simple—variables should:

- be lowercase

- use an underscore to separate words
- not start with numbers
- not override a *built-in* function

Here are examples of variable names, both good and bad:

```
>>> good = 4
>>> bAd = 5  # bad - capital letters
>>> a_longer_variable = 6

# this style is frowned upon
>>> badLongerVariable = 7

# bad - starts with a number
>>> 3rd_bad_variable = 8
  File "<stdin>", line 1
    3rd_bad_variable = 8
                     ^
SyntaxError: invalid syntax

# bad - keyword
>>> for = 4
  File "<stdin>", line 1
    for = 4
        ^
SyntaxError: invalid syntax

# bad -  built-in function
>>> compile = 5
```

Tip

Rules and conventions for naming in Python come from a document named "PEP 8 – Style Guide for Python Code" [8]. PEP stands for Python Enhancement Proposal, which is a community process for documenting a feature, enhancement, or best practice for Python. PEP documents are found on the Python website.

NOTE

Although Python will not allow keywords as variable names, it will allow you to use a *built-in* name as a variable. Built-ins are functions, classes, or variables that Python automatically preloads for you, so you get easy access to them. Unlike keywords, Python will let you use a built-in as a variable name without so much as a peep. However, you should refrain from doing this, it is a bad practice.

Using a built-in name as a variable name *shadows* the built-in. The new variable name prevents you from getting access to the original built-in. Doing so essentially takes the built-in variable and co-opts it for your use. As a result, access to the original built-in may only be obtained through the `__builtins__` module. But it is much better not to shadow it in the first place.

Here is a list of Python's *built-ins* that you should avoid using as variables:

```
>>> dir(__builtins__)

['ArithmeticError', 'AssertionError',
'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError',
'BufferError', 'BytesWarning', 'ChildProcessError',
'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False',
'FileExistsError', 'FileNotFoundError',
'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError', 'IsADirectoryError',
'KeyError', 'KeyboardInterrupt', 'LookupError',
'MemoryError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError', 'RecursionError',
```

```
    'ReferenceError', 'ResourceWarning',
    'RuntimeError', 'RuntimeWarning',
    'StopAsyncIteration', 'StopIteration',
    'SyntaxError', 'SyntaxWarning', 'SystemError',
    'SystemExit', 'TabError', 'TimeoutError', 'True',
    'TypeError', 'UnboundLocalError',
    'UnicodeDecodeError', 'UnicodeEncodeError',
    'UnicodeError', 'UnicodeTranslateError',
    'UnicodeWarning', 'UserWarning', 'ValueError',
    'Warning', 'ZeroDivisionError', '_',
    '__build_class__', '__debug__', '__doc__',
    '__import__', '__loader__', '__name__',
    '__package__', '__spec__', 'abs', 'all', 'any',
    'ascii', 'bin', 'bool', 'bytearray', 'bytes',
    'callable', 'chr', 'classmethod', 'compile',
    'complex', 'copyright', 'credits', 'delattr',
    'dict', 'dir', 'divmod', 'enumerate', 'eval',
    'exec', 'exit', 'filter', 'float', 'format',
    'frozenset', 'getattr', 'globals', 'hasattr',
    'hash', 'help', 'hex', 'id', 'input', 'int',
    'isinstance', 'issubclass', 'iter', 'len',
    'license', 'list', 'locals', 'map', 'max',
    'memoryview', 'min', 'next', 'object', 'oct',
    'open', 'ord', 'pow', 'print', 'property', 'quit',
    'range', 'repr', 'reversed', 'round', 'set',
    'setattr', 'slice', 'sorted', 'staticmethod',
    'str', 'sum', 'super', 'tuple', 'type', 'vars',
    'zip']
```

Tip

Here are a few built-ins that would be tempting variable names otherwise: `dict`, `id`, `list`, `min`, `max`, `open`, `range`, `str`, `sum`, and `type`.

Summary

In Python, everything is an object. Objects hold state, which is also called the value. To keep track of objects you use variables. Python variables are like cattle tags, they are attached to the object and have a name. But the object has the important data, the value and type of data.

This chapter also discussed rebinding of variables. Python allows you to do this, but you should be careful not to change the type of the variable, as that can be confusing to readers of the said code. Finally, the chapter discussed naming conventions for Python variables.

Exercises

1. Create a variable, `pi`, that points to an approximation for the value of π. Create a variable, `r`, for the radius of a circle that has a value of 10. Calculate the area of the circle (π times the radius squared).
   You can do multiplication with `*` and you can square numbers using `**`. For example, `3**2` is `9`.
2. Create a variable, `b`, that points to the base of a rectangle with a value of `10`. Create a variable, `h`, that points to the height of a rectangle with a value of `2`. Calculate the perimeter. Change the base to `6` and calculate the perimeter again.

[8] - https://www.python.org/dev/peps/pep-0008/

More about Objects

This chapter will dive into objects a little bit more. You will cover three important properties of objects:

- identity
- type
- value

Identity

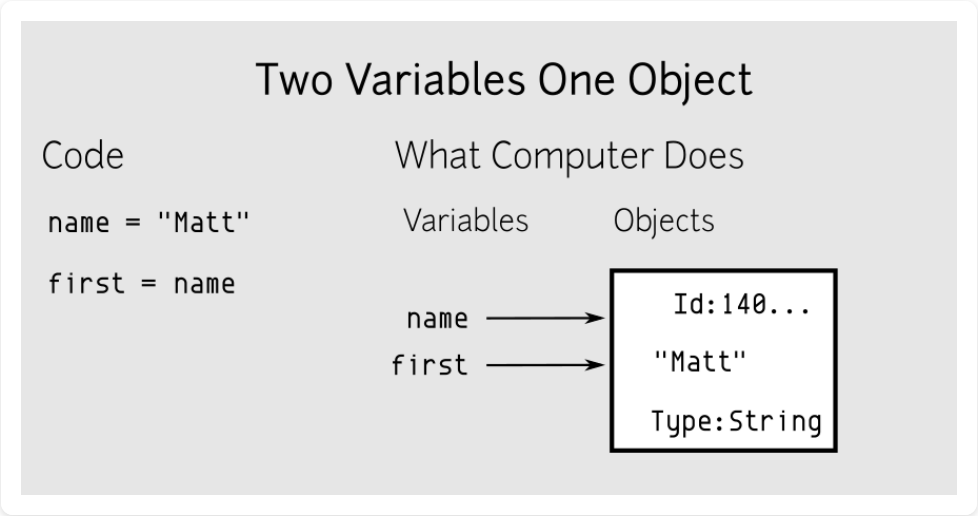
Identity at its lowest level refers to an object's location in the computer's memory. Python has a built-in function called `id` that tells you the identity of an object:

```
>>> name = "Matt"
>>> id(name)
140310794682416
```

When you type this, the identity of the string `"Matt"` will appear as `140310794682416` (which refers to a location in the RAM of your computer). This will generally vary from computer to computer and for each time you start the shell, but the `id` of an object is consistent across the lifetime of a program.

It is possible for a single cow to have two tags on its ears and it is also possible for two variables to refer to the same object. If you want another variable—`first`—to also refer to the same object referred to by `name`, you could do the following:

```
>>> first = name
```



This illustrates what happens when you bind a variable to an existing variable. They both point to the same object. Note that this does not copy the variable! Also note that the object has a value, `"Matt"`, a type, and an id.

This tells Python to give the `first` variable the same id as `name`. Running `id` on either of the two variables will return the same id:

```
>>> id(first)
140310794682416
>>> id(name)
140310794682416
```

What is identity used for? Actually not much. When you program in Python, you typically are not concerned with low-level details such as where the object is living in the RAM of the computer. But identity is used to illustrate when objects are created and if they are mutable. It is also used indirectly for doing *identity checks* with `is`.

The `is` operator checks for identity equality and validates whether or not two variables point to the same object:

```
>>> first is name
True
```

If you print either `first` or `name` at the REPL, it will print the same value because they are both pointing to the exact same object:

```
>>> print(first)
Matt
>>> print(name)
Matt
```

If I had a cattle tag, I could take it off of one cow and attach it to another. Just like a cattle tag, you can take a variable and point it to a new object. I can make `name` point to a new object. You will see that the identity of `name` has changed. But `first` is still the same:

```
>>> name = 'Fred'
>>> id(name)
140310794682434
>>> id(first)
140310794682416
```

Type

Another property of an object is its *type*. Common types are *strings*, *integers*, *floats*, and *booleans*. There are many other kinds of types, and you can create your own as well. The type of an object refers to the class of an object. A class defines the *state* of data an object holds, and the *methods* or actions that it can perform. Python allows you to easily view the type of an object with the built-in function, `type`:

```
>>> type(name)
<class 'str'>
```

The `type` function tells you that the variable `name` points to a string (`str`).

The table below shows the types for various objects in Python.

Object	Type
String	`str`
Integer	`int`
Floating point	`float`
List	`list`
Dictionary	`dict`
Tuple	`tuple`
function	`function`
User-defined class (subclass object)	`type`
Instance of class (subclass of class)	`class`
Built-in function	`builtin_function _or_method`
`type`	`type`

Due to *duck-typing*, the `type` function is not used too frequently. Rather than check if an object is of a certain type that provides an operation, normally you try and do that operation.

Sometimes though you have data and need to convert it to another type. This is common when reading data from standard in. Typically it would come in as a string, and you might want to change it into a number. Python provides built-in classes, `str`, `int`, `float`, `list`, `dict`, and `tuple` that convert (or coerce) to the appropriate type if needed:

```
>>> str(0)
'0'

>>> tuple([1,2])
(1, 2)

>>> list('abc')
['a', 'b', 'c']
```

NOTE

Duck typing comes from a saying used in the 18th century to refer to a mechanical duck. Much like the Turing Test, the saying went:

> *If it looks like a duck, walks like a duck and quacks like a duck, then it's a duck*

But I prefer the scene in *Monty Python and the Holy Grail*, where a lady is determined to be a witch because she weighs as much as a duck. (If this is confusing go watch the movie, I find it enjoyable). The idea is that because she had characteristics (her weight) that were the same as a duck's, she could be considered a witch.

Python takes a similar approach. If you want to loop over an object, you put it in a `for` loop. You don't check first to see if it is a list or a subclass of a list, you just loop over it. If you want to use a plus operation, you don't check to see if the object is a number, or string (or another type that supports addition). If the operation fails, that's ok, it is an indication that you are not providing the correct type.

If you are familiar with object-oriented programming, duck typing eases the requirement for subclassing. Rather than inheriting multiple classes to take advantage of behaviors they provide, you need to implement the protocols (usually by defining a method or two). For example, to create a class that adds, you need to implement a `.__add__` method. Any class can define that method and respond to the plus operation.

Mutability

Another interesting property of an object is its *mutability*. Many objects are *mutable* while others are *immutable*. Mutable objects can change their value in place, in other words, you can alter their state, but their identity stays the same. Objects that are immutable do not allow you to change their value. Instead, you can change their variable reference to a new object, but this will change the identity of the variable as well.

In Python, dictionaries and lists are mutable types. Strings, tuples, integers, and floats are immutable types. Here is an example demonstrating that the identity of a variable holding an integer will change if you change the value. First, you will assign an integer to the variable age and inspect the id:

```
>>> age = 1000
>>> id(age)
140310794682416
```

Notice that if you change the value of the integer, it will have a different id:

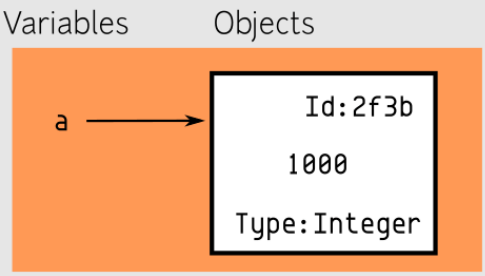
```
>>> age = age + 1
>>> id(age)
140310793921824
```

Immutable Integers

Code

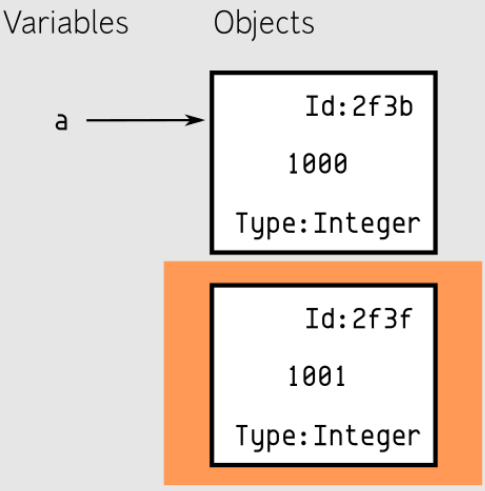
```
a = 1000
```

What Computer Does



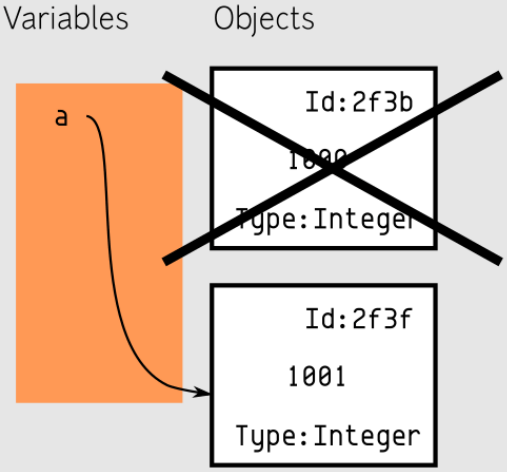
Step 1: Python creates an integer

```
a = a + 1
```



Step 2: Python adds 1 to a and creates an integer

```
a = a + 1
```



Step 3: Python rebinds a, garbage collects old object

This illustrates that when you try to change an integer, you will necessarily create a new integer. Integers are immutable, and you can't change their value.

Here is an example of changing a list. You will start with an empty list, and examine the `id`. Note that even after you add an item to the list, the identity of the list is unchanged, hence it is mutable. First, you will create a list and look at the id:

```
>>> names = []
>>> id(name)
140310794682432
```

Now, add a string into the list. There are a few things to note. The return value of the `.append` method didn't show anything (ie, it is not returning a new list). But if you inspect the `names` variable, you will see that the new name is in there. Also, the id of the list is still the same. You have mutated the list:

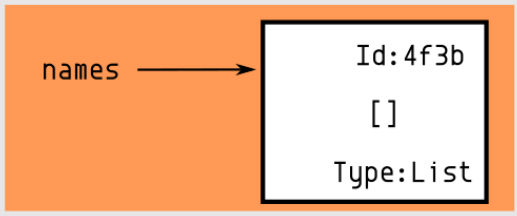
```
>>> names.append("Fred")
>>> names
['Fred']
>>> id(name)
140310794682432
```

Mutable Lists

Code

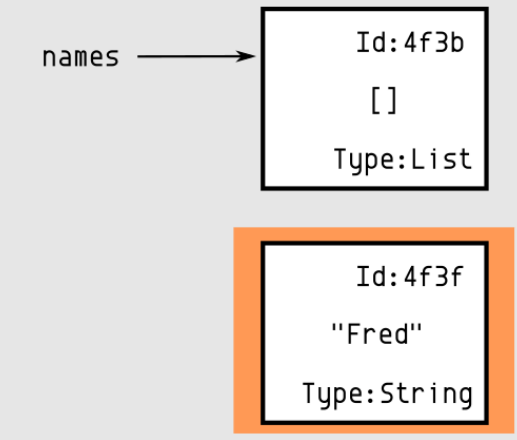
What Computer Does

```
names = []
```



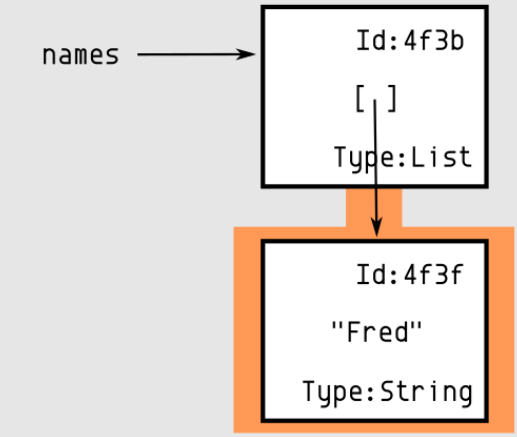
Step 1: Python creates an empty list

```
names.append("Fred")
```



Step 2: Python creates a string object

```
names.append("Fred")
```



Step 3: Python inserts string into existing list

This illustrates that when you append an object into a list, you change the value of the list. The list is mutated. You can add and remove items from it, but the id of the list stays the same.

Mutable objects should not be used for keys in dictionaries and can present problems when used as default parameters for functions.

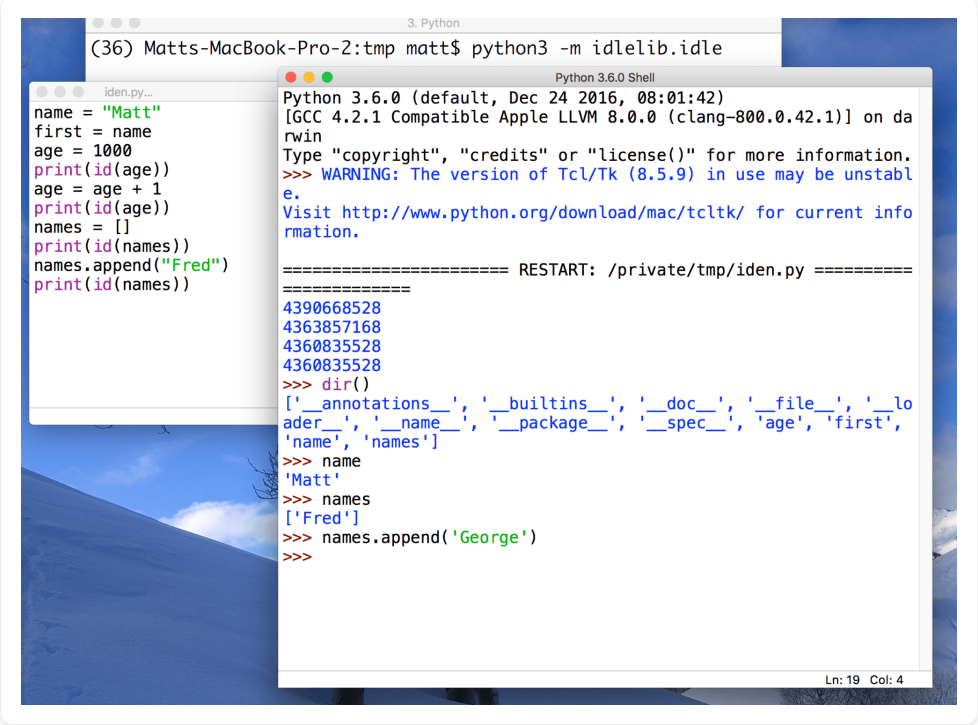
Using IDLE

At this point, it would be good to try this out in IDLE (or your favorite editor that has REPL integration). Because IDLE comes with a REPL, you could type in the previous code and inspect it from there. But you can also write code, run it, and then inspect it from the REPL. To try it, open a new file, and type the following code into it:

```
name = "Matt"
first = name
age = 1000
print(id(age))
age = age + 1
print(id(age))
names = []
print(id(names))
names.append("Fred")
print(id(names))
```

Save this as a file, call it `iden.py`. Then run the file. In IDLE, you need to hit F5 to do this. In the REPL, you should see four numbers printed out. The first two should be different, illustrating that an integer is immutable. The last two numbers are the same. They are the same because even though the list, `names`, was mutated, the id is still the same. This by itself is nothing particularly novel.

The interesting part now is that if you type `dir()` in the REPL, it will show you the variables. You will see that the global variables from `iden.py` are now available.



This illustrates running code and then inspecting it from the REPL in IDLE. If you do not use IDLE, figure out how to do this from your editor.

From the REPL in IDLE you have access to all the global variables. You can inspect `name` or `names`. You can even call functions or methods like `names.append("George")`.

The ability to inspect what just ran gives you the chance to quickly inspect the code, and try things out. It is not uncommon for experienced Python developers to write code in the REPL, paste it into a file, re-run the file, write more code in the REPL, and continue writing code in this manner.

Summary

In Python, everything is an object. Objects have three properties:

- A Type - Indicates what the class is for the object.
- A Value - The data that the object holds. When you test if an object is equal to another object (with ==), you are checking against the value.

- An Id - A unique id for the object. In the Python version found at www.python.org, this is essentially the location in memory of the object, which will be a unique value. When you check whether two objects have the same identity (with `is`), you are checking whether the id is the same.

You also examined how mutable objects, such as lists, can change their value, while immutable objects, like numbers or strings, cannot be changed.

Exercises

1. Create a variable that points to a floating point number. Examine the id, type, and value of that number. Update the variable by adding `20` to it. Re-examine the id, type, and value. Did the id change? Did the value change?
2. Create a variable pointing to an empty list. Examine the id, type, and value of the list. Append the number `300` to the list. Re-examine the id, type, and value. Did the id change? Did the value change?

Numbers

THIS CHAPTER WILL DISCUSS MANIPULATING NUMBERS WITH PYTHON. *INTEGERS* (whole numbers like -1, 5, or 2000) and *Floating Points* (the computer's approximation of real numbers like .333, 0.5 or -1000.234) are available in Python and provide easy numerical manipulation. Out of the box, Python provides support for addition, subtraction, multiplication, division, power, modulo, and more!

Unlike other languages, in Python, everything is an object, including numbers. Integers are of class `int`:

```
>>> type(1)
<class 'int'>
```

Floating point numbers are of class `float`:

```
>>> type(2.0)
<class 'float'>
```

Everything has a level of precision. It's up to the user to determine if it is sufficient for their calculations. Python's floats are represented internally using a binary representation (as per the IEEE 754 standard for floating point numbers). Floats have a certain amount of precision and rounding errors are possible. In fact, one should expect rounding errors. (If you need more precision, the `decimal` module provides a more precise albeit slower implementation).

As a quick example of precision, examine what happens when you perform this apparently simple subtraction operation:

```
>>> print(1.01 - .99)
0.020000000000000018
```

Tip

If you are interested in understanding more about floats and how computers represent them, Wikipedia [9] has more information on the subject.

Addition

The Python REPL can be used as a simple calculator. If you want to add two integers, type in the expression:

```
>>> 2 + 6
8
```

Note

The math example above did not bind the result to a variable. For a simple calculation printing out the result to the terminal may be sufficient. If you need the result after the fact, the Python interpreter stores the last result in a variable named _:

```
>>> 2 + 6
8
>>> result = _
>>> result
8
```

Note that adding two integers together results in an integer.

Likewise, you can also add two floats together:

```
>>> .4+.01
0.41000000000000003
```

This example illustrates once again that care is needed when using floating point numbers, as you can lose precision (the real result would be `0.41`).

What happens when you add an integer and a float?

```
>>> 6 + .2
6.2
```

Python decided that because you are adding an integer and a float, you need floating point arithmetic. In this case, Python converts or *coerces*, `6` to a float behind the scenes, before adding it to `.2`. Python has given you the answer back as a float.

NOTE

If you have an operation involving two numerics, coercion generally does the right thing. For operations involving an integer and a float, the integer is coerced to a float.

NOTE

Coercion between strings and numerics does not occur with most mathematical operations. Two exceptions are the string formatting operator, and the multiplication operator.

When you use `%` with a string on the left side (the *left operand*) and any object (including numbers) on the right side (the *right operand*), Python performs the formatting operator:

```
>>> print('num: %s' % 2)
num: 2
```

If the left operand is a string and you use the multiplication operator, `*`, Python performs repetition:

```
>>> 'Python!' * 2
'Python!Python!'

>>> '4' * 2
'44'
```

NOTE

Explicit conversion can be done with the `int` and `float` built-in classes. (Although these look like functions they are really classes):

```
>>> int(2.3)
2

>>> float(3)
3.0
```

Subtraction

Subtraction is similar to addition. Subtraction of two integers or two floats returns an integer or a float respectively. For mixed numeric types, the operands are coerced before performing subtraction:

```
>>> 2 - 6
-4

>>> .25 - 0.2
0.04999999999999999

>>> 6 - .2
5.8
```

Multiplication

In many programming languages, the * (asterisk) is used for multiplication. You can probably guess what is going to happen when you multiply two integers:

```
>>> 6 * 2
12
```

If you have been following carefully, you will also know what happens when you multiply two floats:

```
>>> .25 * 12.0
3.0
```

And if you mix the types of the product you end up with a float as a result:

```
>>> 4 * .3
1.2
```

Note that the float result in these examples appears correct, though you should be careful, due to floating point precision issues, not to assume that you would always be so lucky.

Division

In Python (like many languages), the / (slash) symbol is used for division:

```
>>> 12 / 4
3.0
```

Python 3 also addressed what many considered to be a wart in prior versions of Python. The result of dividing two integers is a float:

```
>>> 3 / 4
0.75
```

Previously, Python performed integer division. If you want that behavior, you can use the // operator. What integer does Python use? The *floor* of the real result—take the answer and round down:

```
>>> 3 // 4
0
```

Modulo

The *modulo* operator (%) calculates the modulus. This is the same as the remainder of a division operation when the operators are positive. This is useful for determining whether a number is odd or even (or whether you have iterated over 1000 items):

```
# remainder of 4 divided by 3
>>> 4 % 3
1

>>> 3 % 2  # odd if 1 is result
1

>>> 4 % 2  # even if 0 is result
0
```

Tip

Be careful with the modulo operator and negative numbers. Modulo can behave differently, depending on which operand is negative. It makes sense that if you are counting down, the modulo should cycle at some interval:

```
>>> 3 % 3
0
>>> 2 % 3
2
>>> 1 % 3
1
>>> 0 % 3
0
```

What should `-1 % 3` be? Since you are counting down it should cycle over to `2` again:

```
>>> -1 % 3
2
```

But when you switch the sign of the denominator, the behavior becomes weird:

```
>>> -1 % -3
-1
```

Python guarantees that the sign of the modulo result is the same as the denominator (or zero). To further confuse you:

```
>>> 1 % -3
-2
```

The takeaway here is that you probably do not want to do modulo with negative numbers on the denominator unless you are sure that is what you need.

Power

Python also gives you the *power* operator by using ** (double asterisks). If you wanted to square 4 (4 is the base, 2 is the exponent), the following code will do it:

```
>>> 4 ** 2
16
```

Exponential growth tends to let numbers get large pretty quickly. Consider raising 10 to the 100th power:

```
>>> 10 ** 100
10000000000000000000000000000000000000000
00000000000000000000000000000000000000000
0000000000000000000000
```

Programs need to use a certain amount of memory to store integers. Because integers are usually smaller numbers, Python optimizes for them, to not waste memory. Under the covers, it can coerce system integers to long integers to store larger numbers. Python 3 does this automatically for you.

An analogy might be a scale. If you are always weighing small amounts, you might want a small scale. If you deal in sand, you will probably want to put the sand in a bag to make it easier to handle. You will have a bunch of small bags that you will use. But if you occasionally need to weigh larger items that do not fit on the small scale, you need to pull out a bigger scale, and a bigger bag. It would be a waste to use the bigger bag and scale for many of the smaller items.

Similarly, Python tries to optimize storage space for integers towards smaller sizes. When Python does not have enough memory (a small bag) to fit larger integers in, it *coerces* the integer into a *long integer*. This is actually desirable because, in some environments, you run into an *overflow*

error here, where the program dies (or Pac-Man refuses to go over level 255—since it stored the level counter in an 8-bit number).

NOTE

Python includes the `operator` module which provides functions for common mathematical operations. When using more advanced features of Python such as `lambda` functions or *list comprehensions*, this module comes in handy:

```
>>> import operator
>>> operator.add(2, 4)  # same as 2 + 4
6
```

Order of operations

When you are performing math, you do not apply all the operations from left to right. You do the multiplication and division before the addition and subtraction. Computers work the same way. If you want to perform addition (or subtraction) first, use parentheses to indicate the order of operations:

```
>>> 4 + 2 * 3
10
>>> (4 + 2) * 3
18
```

As illustrated in the example, anything in parentheses is evaluated first.

Other operations

The help section from the REPL is pretty useful. There is a topic called `NUMBERMETHODS` that explains how all of the number operations work.

Summary

Python has built-in support for the basic mathematical operations. Addition, subtraction, multiplication, and division are all included. In addition, the

power and modulus operations are available. If you need to control which order the operations occur, wrap parentheses around the operation that you want to happen first.

If you need a simple calculator, rather than opening a calculator application, give Python a try. It should be more than capable for most tasks.

Exercises

1. You slept for 6.2, 7, 8, 5, 6.5, 7.1, and 8.5 hours this week. Calculate the average number of hours slept.
2. Is 297 divisible by 3?
3. What is 2 raised to the tenth power?
4. Wikipedia defines leap years as:

> *Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400. For example, the years 1700, 1800, and 1900 are not leap years, but the years 1600 and 2000 are.*
> —*https://en.wikipedia.org/wiki/Leap_year*

Write Python code to determine if 1800, 1900, 1903, 2000, and 2002 are leap years.

9 - https://en.wikipedia.org/wiki/Floating_point

Strings

Strings are immutable objects that hold character data. A string could hold a single character, a word, a line of words, a paragraph, multiple paragraphs, or even zero characters.

Python denotes strings by wrapping them with `'` (single quotes), `"` (double quotes), `"""` (triple doubles) or `'''` (triple singles). Here are some examples:

```
>>> character = 'a'
>>> name = 'Matt'
>>> with_quote = "I ain't gonna"
>>> longer = """This string has
... multiple lines
... in it"""
>>> latin = '''Lorum ipsum
... dolor'''
>>> escaped = 'I ain\'t gonna'
>>> zero_chars = ''
>>> unicode_snake = "I love \N{SNAKE}"
```

Notice that the strings always start and end with the same style of quote. As illustrated in the `with_quote` example you can put single quotes inside of a double-quoted string—and vice versa. Furthermore, if you need to include the same type of quote within your string, you can *escape* the quote by preceding it with a `\` (backslash). When you print out an escaped character the backslash is ignored.

NOTE

Attentive readers may wonder how to include a backslash in a string. To include a backslash in a normal string, you must escape the backslash with ... you guessed it, another backslash:

```
>>> backslash = '\\'
>>> print(backslash)
\
```

NOTE

Here are the common ways to escape characters in Python:

ESCAPE SEQUENCE	OUTPUT
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\b`	ASCII Backspace
`\n`	Newline
`\t`	Tab
`\u12af`	Unicode 16 bit
`\U12af89bc`	Unicode 32 bit
`\N{SNAKE}`	Unicode character
`\o84`	Octal character
`\xFF`	Hex character

Tip

If you do not want to use an escape sequence, you can use a *raw* string by preceding the string with an `r`. Raw strings are normally used two places. They are used in *regular expressions*, where the backslash is also used as an escape character. You can use regular expressions to match characters (such as phone numbers, names, etc) from text. The `re` module in the Python standard library provides support for regular expressions. Raw strings are also used in Windows paths where the backslash is a delimiter.

Raw strings interpret the character content literally (ie. there is no escaping). The following illustrates the difference between raw and normal strings:

```
>>> slash_t = r'\tText \\'
>>> print(slash_t)
\tText \\

>>> normal = '\tText \\'
>>> print(normal)
        Text \
```

Python also has a triple quoting mechanism for defining strings. Triple quotes are useful for creating strings containing paragraphs or multiple lines. Triple-quoted strings are also commonly used in *docstrings*. Docstrings will be discussed in the chapter on functions. Below is an example of a multi-line triple-quoted string:

```
>>> paragraph = """Lorem ipsum dolor
... sit amet, consectetur adipisicing
... elit, sed do eiusmod tempor incididunt
... ut labore et dolore magna aliqua. Ut
... enimad minim veniam, quis nostrud
... exercitation ullamco laboris nisi ut
... aliquip ex ea commodo consequat. Duis
```

```
... aute irure dolor in reprehenderit in
... voluptate velit esse cillum dolore eu
... fugiat nulla pariatur. Excepteur sint
... occaecat cupidatat non proident, sunt
... in culpa qui officia deserunt mollit
... anim id est laborum."""
```

A nice benefit of using triple-quoted strings is that you can embed single and double quotes inside it without escaping them:

```
>>> """This string has double " and single
... quotes ' inside of it"""
'This string has double " and single\nquotes \' inside of it'
```

Unless they butt up to the end of the string, then you will need to escape the final quote:

```
>>> """He said, "Hello""""
  File "<stdin>", line 1
    """He said, "Hello""""
                          ^
SyntaxError: EOL while scanning string literal

>>> """He said, "Hello\""""
'He said, "Hello"'
```

Formatting Strings

Storing strings in variables is nice, but being able to compose strings of other strings and manipulate them is also necessary. One way to achieve this is to use string formatting.

In Python 3, the preferred way to format strings is to use the `.format` method of strings. Below, you tell Python to replace `{}` (a placeholder) with the contents of `name` or the string `Matt`:

```
>>> name = 'Matt'
>>> print('Hello {}'.format(name))
Hello Matt
```

Another useful property of formatting is that you can also format non-string objects, such as numbers:

```
>>> print('I:{} R:{} S:{}'.format(1, 2.5, 'foo'))
I:1 R:2.5 S:foo
```

Format string syntax

Format strings have a special syntax for *replacement fields*. If an object is passed into the format string, attributes can be looked up using `.attribute_name` syntax. There is also support for pulling index-able items out by using `[index]` as well. The Python documentation refers to these as *field names*. The field names can be empty, a name of a keyword argument, a number of a positional argument, or index of a list or dictionary (in square brackets):

```
>>> 'Name: {}'.format('Paul')
'Name: Paul'

>>> 'Name: {name}'.format(name='John')
'Name: John'

>>> 'Name: {[name]}'.format({'name':'George'})
'Name: George'
```

The curly braces ({ and }) can also contain an integer inside of them. The integer refers to the zero based position of the argument passed into `.format`. Below is an example of using the numbers of positional arguments in the placeholders. The first argument to `.format`, `'Paul'`, is at position 0, the second, `'George'`, is position 1, and `'John'` is at 2:

```
>>> 'Last: {2} First: {0}'.format('Paul', 'George',
...     'John')
'Last: John First: Paul'
```

There is a whole language for formatting strings. If you insert a colon following the field name, you can provide further formatting information. The format is below. Anything in square brackets is optional:

```
:[[fill]align][sign][#][0][width][grouping_option][.precision][type]
```

The following tables lists the fields and their meaning.

FIELD	MEANING
fill	Character used to fill in `align` (default is space)
align	Alight output < (left align), > (right align), ^ (center align), or = (put padding after sign)
sign	For numbers + (show sign on both positive and negative numbers, - (default, only on negative), or *space* (leading space for positive, sign on negative)
#	Prefix integers. `0b` (binary), `0o` (octal), or `0x` (hex)
0	Enable zero padding
width	Minimum field width
grouping_option	Number separator , (use comma for thousands separator), _ (Use underscore for thousands separator)
.precision	For floats (digits after period (floats), for non-numerics (max length)
type	Number type or `s` (string format default) see Integer and Float charts

The tables below lists the various options you have for formatting integer and floating point numbers.

INTEGER TYPES	MEANING
`b`	binary
`c`	character - convert to Unicode character
`d`	decimal (default)
`n`	decimal with locale-specific separators
`o`	octal
`x`	hex (lower-case)
`X`	hex (upper-case)

FLOAT TYPES	MEANING
e/E	Exponent. Lower/upper-case e
f	Fixed point
g/G	General. Fixed with exponent for large, and small numbers (g default)
n	g with locale-specific separators
%	Percentage (multiplies by 100)

Some `format` examples

Here are a few examples of using `.format`. To format a string in the center of 12 characters surrounded by `*`, use the code below. `*` is the *fill* character, `^` is the *align* field, and `12` is the *width* field:

```
>>> "Name: {:*^12}".format("Ringo")
'Name: ***Ringo****'
```

Next, you format a percentage using a width of 10, one decimal place, and the sign before the width padding. `=` is the *align* field, `+` ensures that there is always a sign (negative or positive), `10.1` are the *width* and *precision* fields, and `%` is the *float type*, which converts the number to a percentage:

```
>>> "Percent: {:=+10.1%}".format(-44/100)
'Percent: -    44.0%'
```

Below are a binary and a hex conversion. The *integer type* field is set to `b` and `x` respectively:

```
>>> "Binary: {:b}".format(12)
'Binary: 1100'

>>> "Hex: {:x}".format(12)
'Hex: c'
```

NOTE

The `.format` method on a string provides an alternative for the `%` operator which is similar to C's `printf`. The `%` operator is still available and some users prefer it as it requires less typing for simple statements and because it is similar to C. `%s`, `%d`, and `%x` are replaced by their string, integer, and hex values respectively. Here are some examples:

```
>>> "Num: %d Hex: %x" % (12, 13)
'Num: 12 Hex: d'

>>> "%s %s" % ('hello', 'world')
'hello world'
```

TIP

A great resource for formatting is in the built-in help documentation, available in the REPL. Type:

```
>>> help()
```

Which puts you in the help mode, and gives you a `help>` prompt. Then type:

```
help> FORMATTING
```

You can scroll through here and find many examples. A bare return from the `help>` prompt will return you to the normal prompt.

Another resource is found at https://pyformat.info/. This website contains many formatting examples with both `.format` and the older `%` operator.

There is also an entry in `help` for strings, located under `STRINGS`.

F-Strings

Python 3.6 introduced a new type of string, called *f-string*. If you precede a string with an `f`, it will allow you to include code inside of the placeholders. Here is a basic example:

```
>>> name = 'matt'
>>> f'My name is {name}'
'My name is matt'
```

Python will look in the placeholder and evaluate the code there. Note that the placeholder can contain function calls, method calls, or any other arbitrary code:

```
>>> f'My name is {name.capitalize()}'
'My name is Matt'
```

You can also provide format strings following a colon:

```
>>> f'Square root of two: {2**.5:5.3f}'
'Square root of two: 1.414'
```

Summary

In this chapter, strings were introduced. Strings can be defined with various delimiters. Unlike other languages, which may distinguish between a string defined with `"` and one defined with `'`, Python makes no distinction. Triple-quoted strings, however, may span multiple lines.

We also looked at the `.format` method and gave examples of formatting strings. Finally, the chapter introduced a new feature in Python 3.6, f-strings.

Exercises

1. Create a variable, `name`, pointing to your name. Create another variable, `age`, holding an integer value for your age. Print out a string

formatted with both values. If your name was Fred and age was 23 it would print:

```
Fred is 23
```

2. Create a variable, `paragraph`, that has the following content:

   "Python is a great language!", said Fred. "I don't ever remember having this much fun before."

3. Go to https://unicode.org and find the symbol omega in the Greek character code chart. Create a string that holds the omega character, using both the Unicode code point (`\u` form) and Unicode name (`\N` form). The code point is the hex number in the chart, the name is the bolded capital name following the code point. For example, the theta character has the code point of `03f4` and a name of `GREEK CAPITAL THETA SYMBOL`.
4. Make a variable, `item`, that points to a string, `"car"`. Make a variable, `cost`, that points to 13499.99. Print out a line that has `item` in a left-justified area of 10 characters, and `cost` in a right-justified area of 10 characters with 2 decimal places and commas in the thousands place. It should look like this (without the quotes):

```
'car        13,499.99'
```

dir, help, and pdb

You have only touched the surface of strings, but you need to take a break to discuss two important functions and one library that come with Python. The first function is `dir`, which illustrates how powerful and useful the REPL is. The `dir` function returns the attributes of an object. If you had a Python interpreter open and wanted to know what the attributes of a string are, you can do the following:

```
>>> dir('Matt')

['__add__', '__class__', '__contains__',
'__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize',
'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

`dir` lists all the attributes of the object passed into it. Since you passed in the string `'Matt'` to `dir`, the function displays the attributes of a string. This handy feature of Python illustrates its “batteries included” philosophy.

Python gives you an easy mechanism to discover the attributes of any object. Other languages might require special websites, documentation or IDEs to access similar functionality. But in Python, because you have the REPL, you can get at this information quickly and easily.

The attribute list is in alphabetical order, and you can normally ignore the first couple of attributes starting with `__`. Later on, you will see attributes such as `capitalize` (which is a *method* that capitalizes a string), `format` (which as was illustrated previously, allows for formatting of strings), or `lower` (which is a *method* used to ensure the string is lowercase). These attributes happen to be *methods*, which are functions that are attached to objects. To call, or *invoke* a function, you place a period after the object, then the method name, then parentheses.

The three methods are invoked below:

```
>>> print('matt'.capitalize())
Matt

>>> print('Hi {}'.format('there'))
Hi there

>>> print('YIKES'.lower())
yikes
```

Dunder methods

You might be wondering what all the attributes starting with `__` are. People call them *special methods*, *magic methods*, or *dunder* methods since they start (and end) with double underscores (Double UNDERscores). “Dunder add” is one way to say `__add__`, “the add magic method” is another. Special methods determine what happens under the covers when operations are performed on an object. For example, when you use the `+` or `%` operator on a string, the `.__add__` or `.__mod__` method is invoked respectively.

Beginner Pythonistas can usually ignore dunder methods. When you start implementing your own classes and want them to react to operations such as + or %, you can define them.

Tip

In the `help()` documentation is an entry, `SPECIALMETHODS`, that describes these methods.

Another place where these are described is on the Python website. Go to Documentation, Language Reference, Data Model. It is tucked away in there.

`help`

`help` is another built-in function that is useful in combination with the REPL. The book previously mentioned invoking `help()`, without any arguments, to bring up help documentation.

The `help` function also provides documentation for a method, module, class, or function if you pass them in as an argument. For example, if you are curious what the attribute `upper` on a string does, the following gives you the documentation:

```
>>> help('some string'.upper)
Help on built-in function upper:

upper(...) method of builtins.str instance
    S.upper() -> str

    Return a copy of S converted to uppercase.
```

The `help` function, combined with the REPL, allows you to read up on documentation without having to go to a browser, or even have internet access. If you were stranded on a desert island, you should be able to learn

Python, provided you had a computer with Python installed and a power source.

pdb

Python also includes a debugger to step through code. It is found in a module named pdb. This library is modeled after the gdb library for C. To drop into the debugger at any point in a Python program, insert the code:

```
import pdb; pdb.set_trace()
```

These are two statements here, but I typically type them in a single line separated by a semicolon—that way I can easily remove them with a single keystroke from my editor when I am done debugging. This is also about the only place I use a semicolon in Python code (two statements in a single line).

When this line is executed, it will present a (pdb) prompt, which is similar to the REPL. Code can be evaluated at this prompt and you can inspect objects and variables as well. Also, breakpoints can be set for further inspection.

Below is a table listing useful pdb commands:

COMMAND	PURPOSE
h, help	List the commands available
n, next	Execute the next line
c, cont, continue	Continue execution until a breakpoint is hit
w, where, bt	Print a stack trace showing where execution is
u, up	Pop up a level in the stack
d, down	Push down a level in the stack
l, list	List source code around current line

NOTE

In *Programming Pearls*, Jon Bentley states:

> *When I have to debug a little algorithm deep inside a big program, I sometimes use debugging tools... though, print statements are usually faster to implement and more effective than sophisticated debuggers.*

I've heard Guido van Rossum, the creator of Python, voice the same opinion: he prefers *print debugging*. Print debugging is easy, simply insert `print` functions to provide clarity as to what is going on. This is often sufficient to figure out a problem. Make sure to remove these debug statements or change them to `logging` statements before releasing the code. If more exploration is required, you can always use the `pdb` module.

Summary

Python provides many tools to make your life easier. If you learn to use the REPL, you can take advantage of them. The `dir` function will help you see what the attributes of an object are. Then, you can use the `help` function to inspect those attributes for documentation.

This chapter also introduced the `pdb` module. This module allows you to step through code, which can be useful for debugging.

Exercises

1. Open a REPL, and create a variable, `name`, with your name in it. List the attributes of the string. Print out the help documentation for the `.find` and `.title` methods.

2. Open a REPL, and create a variable, `age`, with your age in it. List the attributes of the integer. Print out the help documentation for the `.numerator` method.

Strings and Methods

In the previous chapter you learned about the built-in `dir` function and saw some methods you can call on string objects. Because strings are immutable, these methods do not mutate the string, but rather return a new string or a new result. Strings allow you create a new version that is capitalized, return a formatted string, or create a lowercase string, as well as many other actions. You do this by calling *methods*.

Methods are functions that are called on an instance of a type. What does this mean? The string type allows you to *call* a method (another term for call is *invoke*) by placing a `.` (period) and the method name directly after the variable name holding the data (or the data itself), followed by parentheses with arguments inside of them.

Note

In this book, I place a period in front of methods. This is meant to remind you that you need to have an object before the method. I will mention the `.capitalize` method, rather than saying `capitalize`. The invocation looks like this on the `text` object:

```
text.capitalize()
```

This is in contrast to a function, like `help`, which you invoke by itself (there is no object or period before it):

```
help()
```

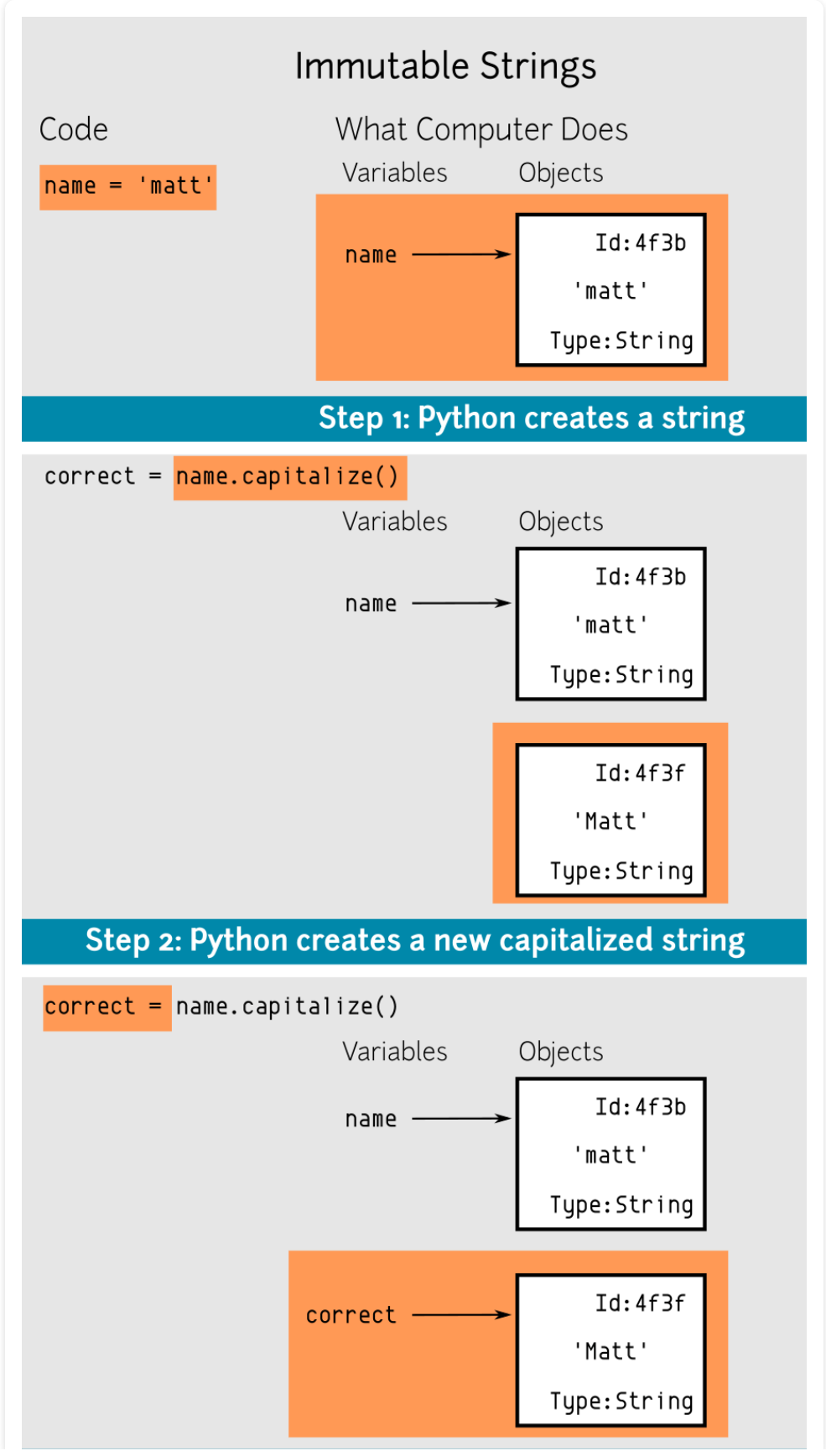

Immutable Strings
Code
What Computer Does
name = 'matt'
Variables
Objects
name
Id:4f3b
'matt'
Type:String
Step 1: Python creates a string
correct = name.capitalize()
Variables
Objects
name
Id:4f3b
'matt'
Type:String
Id:4f3f
'Matt'
Type:String
Step 2: Python creates a new capitalized string
correct = name.capitalize()
Variables
Objects
name
Id:4f3b
'matt'
Type:String
correct
Id:4f3f
'Matt'
Type:String

Illustration of calling a method on a string. The method does not change the string because it is immutable. Rather, the method returns a new string.

Here is an example of calling the `.capitalize` method on a variable pointing to a string and a string literal. Note that this does not change the object on which it is called. Because a string is immutable, the result of the method is a new object with the capitalized value:

```
>>> name = 'matt'

# invoked on variable
>>> correct = name.capitalize()
>>> print(correct)
Matt
```

Note that `name` does not change:

```
>>> print(name)
matt
```

Note, the `.capitalize` method does not have to be called on a variable. You can invoke the method directly on a string literal:

```
>>> print('fred'.capitalize())
Fred
```

In Python, methods and functions are *first-class objects*. As was previously mentioned, everything is an object. If the parentheses are left off, Python will not throw an error, it will only show a reference to a method, which is an object:

```
>>> print('fred'.capitalize)
<built-in method capitalize of str object at
0x7ff648617508>
```

Having first-class objects enables more advanced features like *closures* and *decorators* (these are discussed in my intermediate Python book).

NOTE

Do integers and floats have methods? Yes, again, everything in Python is and object and objects have methods. This is easy to verify by invoking `dir` on an integer (or a variable holding an integer):

```
>>> dir(4)
['__abs__', '__add__', '__and__',
 '__class__',
 ...
 '__subclasshook__', '__truediv__',
 '__trunc__', '__xor__', 'conjugate',
 'denominator', 'imag', 'numerator',
 'real']
```

Invoking a method on a number is somewhat tricky due to the use of the `.` to denote calling a method. Because `.` is common in floats, it would confuse Python if `.` were also used to call methods on numbers.

For example, the `.conjugate` method returns the complex conjugate of an integer. But if you try to invoke it on an integer, you will get an error:

```
>>> 5.conjugate()
Traceback (most recent call last):
  ...
  5.conjugate()
           ^
SyntaxError: invalid syntax
```

One solution to this is to wrap the number with parentheses:

```
>>> (5).conjugate()
5
```

Another option would be to assign a variable to `5` and invoke the method on the variable:

```
>>> five = 5
>>> five.conjugate()
5
```

However, in practice, it is fairly rare to call methods on numbers.

Common string methods

Here are a few string methods that are commonly used or found in the wild. Feel free to explore others using `dir` and `help` (or the online documentation).

`endswith`

If you have a variable holding a filename, you might want to check the extension. This is easy with `.endswith`:

```
>>> xl = 'Oct2000.xls'
>>> xl.endswith('.xls')
True
>>> xl.endswith('.xlsx')
False
```

NOTE

Notice that you had to pass in a *parameter* (or *argument*), `'xls'`, into the method. Methods have a *signature*, which is a funky way of saying that they need to be called with the correct number (and type) of parameters. For the `.endswith` method, it makes sense that if you want to know if a string ends with another string you have to tell Python which ending you want to check for. This is done by passing the end string to the method.

Tip

Again, it is usually easy to find out this sort of information via `help`. The documentation should tell you what parameters are required as well as any optional parameters. Here is the help for `endswith`:

```
>>> help(xl.endswith)
Help on built-in function endswith:

endswith(...)
    S.endswith(suffix[, start[, end]]) -> bool

    Return True if S ends with the specified
    suffix, False otherwise. With optional
    start, test S beginning at that position.
    With optional end, stop comparing S at
    that position. suffix can also be a
    tuple of strings to try.
```

Notice the line:

```
S.endswith(suffix[, start[, end]]) -> bool
```

The `S` represents the string (or *instance*) you are invoking the method on, in this case, the `xl` variable. `.endswith` is the method name. Between the parentheses, `(` and `)`, are the parameters. `suffix` is a *required* parameter, the `.endswith` method will complain if you do not provide it:

```
>>> xl.endswith()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: endswith() takes at least 1 argument
(0 given)
```

The parameters between the square brackets `[` and `]` are *optional* parameters. In this case, `start` and `end` allow you to only check a portion of the string. If you wanted to check if the characters starting at 0 and ending at 3 ends with `Oct` you could do the following:

```
>>> xl.endswith('Oct', 0, 3)
True
```

Strings also have a `.startswith` method, so the correct way to check if a string starts with `'Oct'` is:

```
>>> xl.startswith('Oct')
True
```

find

The `.find` method allows you to find substrings inside other strings. It returns the *index* (offset starting at 0) of the matched substring. If no substring is found it returns `-1`:

```
>>> word = 'grateful'

# 0 is g, 1 is r, 2 is a
>>> word.find('ate')
2
>>> word.find('great')
-1
```

format

`format` allows for easy creation of new strings by combining existing variables. The chapter on strings discussed this method:

```
>>> print('name: {}, age: {}'.\
...       format('Matt', 10))
name: Matt, age: 10
```

NOTE

In the above example, the `print` function is spread across two lines. By placing a `\` following a `.` you indicate to Python that you want to continue on the next line. If you have opened a left parenthesis, `(`, you can also place the arguments on multiple lines without a `\`:

```
>>> print("word".
...       find('ord'))
1
>>> print("word".find(
...       'ord'))
1
```

To help make the code more readable, indent the continued lines. Note that indenting with four spaces serves to indicate to anyone reading the code that the second line is a continuation of a previous statement:

```
>>> print("word".\
...     find('ord'))
1
>>> print("word".find(
...     'ord'))
1
```

Why spread code that could reside in a single line across multiple lines? Where this comes into play in most code is dealing with code standards that expect lines to be less than 80 characters in length. If a method takes multiple arguments, it may be hard to follow the 80 character limit. (Note that Python itself does not care about line length, but readers of your code might). It is not uncommon to see a separate line for each argument to a method:

```
>>> print('{} {} {} {} {}'.format(
...     'hello',
...     'to',
...     'you',
```

```
...     'and',
...     'you'
... ))
hello to you and you
```

join

Oftentimes you have a list (lists are discussed later in the book) of items and need to insert something between them. The `.join` method creates a new string from a sequence by inserting a string between every member of the list:

```
>>> ', '.join(['1','2','3'])
'1, 2, 3'
```

TIP

For most Python interpreters, using `.join` is faster than repeated concatenation using the `+` operator. The above idiom is common.

lower

The `.lower` method returns a copy of the string converted to lowercase. The is often useful for validating if the input matches a string. For example, some programs capitalize file extensions, while others do not. If you wanted to know if a file name had `TXT` or `txt` as an extension, you could do the following:

```
>>> fname = 'readme.txt'
>>> fname.endswith('txt') or fname.endswith('TXT')
True
```

A more Pythonic version would read:

```
>>> fname.lower().endswith('txt')
True
```

`startswith`

The `.startswith` method is analogous to `.endswith` except that it checks that a string starts with another string:

```
>>> 'Book'.startswith('B')
True
>>> 'Book'.startswith('b')
False
```

`strip`

The `.strip` method returns a new string that removes preceding and trailing *whitespace* (spaces, tabs, newlines). This may come in handy if you have to normalize data or parse input from a user (or the web):

```
>>> '   hello  there  '.strip()
'hello  there'
```

Note that three spaces at the front of the string were removed as were the two at the end. But the two spaces between the words were left intact. If you are interested in removing only the leading whitespace or rightmost whitespace, the methods `lstrip` and `rstrip` respectively will perform those duties.

`upper`

The `.upper` method is analogous to `.lower`. It returns a copy of the string with all of the letters capitalized:

```
>>> 'yell'.upper()
'YELL'
```

Other methods

There are other string methods, but they are used less often. Feel free to explore them by reading the documentation and trying them out. The appendix provides a list of them.

NOTE

The `STRINGMETHODS` entry in the `help` section from the REPL contains documentation for all of the string methods as well as some examples.

Summary

This chapter talked about methods. Methods are always called by putting an object and a period before the method name. You also looked at some of the more common methods of strings. One thing to remember is that a string is immutable. If you want to change a string's value, you need to create a new string.

Exercises

1. Create a string, `school`, with the name of your elementary school. Examine the methods that are available on that string. Use the `help` function to view their documentation.
2. Create a string, `country`, with the value `'usa'`. Create a new string, `correct_country`, that has the value in uppercase, by using a string method.
3. Create a string, `filename`, that has the value `'hello.py'`. Check and see if the filename ends with `'.java'`. Find the index location of `'py'`. See if it starts with `'world'`.
4. Open a REPL. Enter the help documentation and scan through the `STRINGMETHODS` entry.

Comments, Booleans, and None

This chapter will introduce comments, booleans, and None. Comments have the potential to make your code more readable. The boolean and None types are very common throughout Python code.

Comments

Comments are not a type per se because they are ignored by Python. Comments serve as reminders to the programmer. There are various takes on comments, their purpose, and their utility. There is a continuum from those who are against any and all comments, to those who comment almost every line of code, as well as those who are in between. If you are contributing to a project, try to be consistent with their commenting scheme. A basic rule of thumb is that a comment should explain the *why* rather than the *how* (code alone should be sufficient for the how).

To create a comment in Python, start a line with a #. Anything that follows the hash is ignored:

```
>>> # This line is ignored by Python
```

You can also comment at the end of a line:

```
>>> num = 3.14  # PI
```

Tip

A rogue use of comments is to temporarily disable code during editing. If your editor supports this, it is sometimes easier to comment out code rather than remove it completely. But the best practice is to remove commented-out code before sharing the code with others.

Other languages support multi-line comments, but Python does not. The only way to comment multiple lines is to start every line with #.

Tip

You may be tempted to comment out multiple lines of code by making those lines a triple-quoted string. This is ugly and confusing. Try not to do this.

Booleans

Booleans represent the values for true and false. You have already seen them in previous code examples, such as the result of `.startswith`:

```
>>> 'bar'.startswith('b')
True
```

You can also assign those values to variables:

```
>>> a = True
>>> b = False
```

NOTE

The actual name of the boolean class in Python is `bool`, not boolean:

```
>>> type(True)
<class 'bool'>
```

It can be useful to convert other types to booleans. In Python, the `bool` class can do that. Converting from one type to another is called *casting*. However, this is usually unnecessary due to the implicit casting Python performs when conditionals are evaluated. The conditional statement will do this casting for you.

In Python parlance, it is common to hear of objects behaving as "truthy" or "falsey"—that means that non-boolean types can implicitly behave as though they were booleans. If you are unsure what the behavior might be, pass in the type to the `bool` class for an explicit conversion (or cast).

For strings, an empty string is "falsey", while non-empty values coerce to `True`:

```
>>> bool('')
False
>>> bool('0')  # The string containing 0
True
```

Since a non-empty string behaves as truthy, you can test whether the string has content. In the code below `name` has been set, but imagine that it came from user input:

```
>>> name = 'Paul'
>>> if name:
...     print("The name is {}".format(name))
... else:
...     print("Name is missing")
The name is Paul
```

You don't have to test if the name has a length. So don't do this:

```
>>> if len(name) > 0:
...     print("The name is {}".format(name))
```

Also, you don't need to do this:

```
>>> if bool(name):
...     print("The name is {}".format(name))
```

because Python will evaluate the contents of the `if` statement, and coerce to a boolean for you. Because a string is `True` when it has content you only need:

```
>>> if name:
...     print("The name is {}".format(name))
```

Note

The built-in types, `int`, `float`, `str`, and `bool`, are classes. Even though their capitalization (lowercase) makes it look as if they were functions, they are classes. Invoking `help(str)` will confirm this:

```
>>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
```

This is one of those slight inconsistencies with Python. User-defined classes typically follow PEP8, which suggests camel cased naming of classes.

For numbers, zero coerces to `False` while other numbers have “truthy” behavior:

```
>>> bool(0)
False
>>> bool(4)
True
```

While explicit casting via the `bool` function is available, it is usually overkill, because variables are implicitly coerced to booleans when used in conditional statements. For example, container types, such as *lists* and *dictionaries*, when empty, behave as "falsey". On the flipside, when they are populated they act as "truthy".

Tip

Be careful when parsing content that you want to turn into booleans. Strings that are non-empty evaluate to `True`. One example of a string that might bite you is the string `'False'` which evaluates to `True`:

```
>>> bool('False')
True
```

Here is a table of truthy and falsey values:

Truthy	Falsey
`True`	`False`
Most objects	`None`
`1`	`0`
`3.2`	`0.0`
`[1, 2]`	`[]` (empty list)
`{'a': 1, 'b': 2}`	`{}` (empty dict)
`'string'`	`""` (empty string)
`'False'`	
`'0'`	

Tip

Do not test boolean values to check if they are equal to `True`. Do not explicitly cast expressions to boolean results. If you have a variable, done, containing a boolean, this is sufficient:

```
if done:
    # do something
```

While this is overkill:

```
if done == True:
    # do something
```

As is this:

```
if bool(done):
    # do something
```

Similarly, if you have a list and need to distinguish between an empty and non-empty list, this is sufficient:

```
members = []
if members:
    # do something if members
    # have values
else:
    # member is empty
```

Likewise, this test is superfluous. It is not necessary to determine the truthiness of a list by its length:

```
if len(members) > 0:
    # do something if members
    # have values
else:
    # member is empty
```

NOTE

If you wish to define the implicit truthiness for user-defined objects, the `.__bool__` method specifies this behavior. It can return `True`, or `False`. If this magic method is not defined, the `.__len__` method is checked for a non-zero value. If neither method is defined, an object defaults to `True`:

```
>>> class Nope:
...     def __bool__(self):
...         return False

>>> n = Nope()
>>> bool(n)
False
```

If this is confusing, feel free to come back to this example after reading about classes.

None

`None` is an instance of `NoneType`. Other languages have similar constructs such as *nil*, *NULL* or *undefined*. Variables can be assigned to `None` to indicate that they are waiting to hold a real value. `None` coerces to `False` in a boolean context:

```
>>> bool(None)
False
```

NOTE

A Python function defaults to returning `None` if no return statement is specified:

```
>>> def hello():
...     print("hi")

>>> result = hello()
hi
>>> print(result)
None
```

NOTE

`None` is a *singleton* (Python only has one copy of `None` in the interpreter). The `id` for this value will always be the same:

```
>>> a = None
>>> id(a)
140575303591440
>>> b = None
>>> id(b)
140575303591440
```

As any variable containing `None` is the same object as any other variable containing `None`. You typically use `is` to check for *identity* with these variables rather than using == to check for *equality*:

```
>>> a is b
True
>>> a is not b
False
```

`is` is faster than == and connotes to the programmer that identity is being compared rather than the value.

You can put the `is` expression in an `if` statement:

```
>>> if a is None:
...     print("A is not set!")
A is not set!
```

Since `None` evaluates to `False` in a boolean context you could also do the following:

```
>>> if not a:
...     print("A is not set!")
A is not set!
```

But, you should be careful as other values also evaluate to `False`, such as `0`, `[]`, or `''` (empty string). Checking against `None` is explicit.

Summary

In this chapter, you learned about comments in Python. Comments are started with a hash, and any content following the hash until the end of the line is ignored. There are no multi-line comments.

The chapter also discussed `True`, `False`, and boolean coercion. Most values are `True` in a boolean context (when used in an `if` statement). The `False` values are zero, `None`, and empty sequences.

Finally, the `None` object was mentioned. It is a singleton that is used to indicate that you have a variable that may be assigned a value in the future. It is also the result of a function that does not explicitly return a value.

Exercises

1. Create a variable, `age`, set to your age. Create another variable, `old`, that uses a condition to test whether you are older than 18. The value of `old` should be `True` or `False`.
2. Create a variable, `name`, set to your name. Create another variable, `second_half`, that tests whether the name would be classified in the second half of the alphabet? What do you need to compare it to?
3. Create a list, `names`, with the names of people in a class. Write code to print `'The class is empty!'` or `'Class has enrollments.'`, based on whether there are values in `names`. (See the tip in this chapter for details).
4. Create a variable, `car`, set to `None`. Write code to print `'Taxi for you!'`, or `'You have a car!'`, based on whether or not `car` is set (`None` is not the name of a car).

Conditionals and Whitespace

In this chapter, you will learn more about making comparisons in Python. Most code needs to make decisions about which path to execute, so you will look at how this is done.

In addition to the boolean values, `True` and `False`, in Python, you can also use expressions to get boolean values. If you have two numbers, you might want to compare them to check if they are greater than or less than each other. The operators, > and <, do this respectively:

```
>>> 5 > 9
False
```

Here is a table of comparison operations to create boolean values:

Operator	Meaning
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`==`	Equal to
`!=`	Not equal to
`is`	Identical object
`is not`	Not identical object

These operations work on most types. If you create a custom class that defines the appropriate magic methods, your class can use them as well:

```
>>> name = 'Matt'
>>> name == 'Matt'
True
>>> name != 'Fred'
True
>>> 1 > 3
False
```

NOTE

The “rich comparison” magic methods, `__gt__`, `__lt__`, `__ge__`, `__le__`, `__eq__`, and `__ne__` correspond to `>`, `<`, `>=`, `<=`, `==`, and `!=` respectively. Defining all of these can be somewhat tedious and repetitive. For classes where these comparisons are commonly used, the `functools.total_ordering` class decorator gives you all of the comparison functionality as long as you define `__eq__` and `__le__`. The decorator will automatically derive the remainder of the comparison methods. Otherwise, all six methods should be implemented:

```
>>> import functools
>>> @functools.total_ordering
... class Abs(object):
...     def __init__(self, num):
...         self.num = abs(num)
...     def __eq__(self, other):
...         return self.num == abs(other)
...     def __lt__(self, other):
...         return self.num < abs(other)

>>> five = Abs(-5)
>>> four = Abs(-4)
>>> five > four  # not using less than!
True
```

Decorators are considered an intermediate subject and are not covered in this beginning book.

TIP

The `is` and `is not` statements are for comparing *identity*. When testing for identity—if two objects are the same actual object with the same `id` (not just the same value)—use `is` or `is not`. Since `None` is a singleton and only has one identity, `is` and `is not` are used with `None`:

```
>>> if name is None:
...     # initialize name
```

Combining conditionals

Conditional expressions are combined using *boolean logic*. This logic consists of the `and`, `or`, and `not` operators.

BOOLEAN OPERATOR	MEANING
`x and y`	Both `x` and `y` must evaluate to `True` for true result
`x or y`	If `x` or `y` is `True`, result is true
`not x`	Negate the value of `x` (`True` becomes `False` and vice versa)

Below is a simple example for setting a grade based on a `score` using `and` to test whether the score is between two numbers:

```
>>> score = 91
>>> if score > 90 and score <= 100:
...     grade = 'A'
```

NOTE

Python allows you to do the above example using a *range comparison* like this:

```
>>> if 90 < score <=100:
...     grade = 'A'
```

Either style works, but range comparisons are not common in other languages.

Here is an example for checking if a given name is a member of a band:

```
>>> name = 'Paul'
>>> beatle = False
>>> if name == 'George' or \
...     name == 'Ringo' or \
...     name == 'John' or \
...     name == 'Paul':
...     beatle = True
... else:
...     beatle = False
```

NOTE

In the above example the `\` at the end of `"'George' or \"` indicates that the statement will be continued on the next line.

Like most programming languages, Python allows you to wrap conditional statements in parentheses. Because they are not required in Python, most developers leave them out unless they are needed for operator precedence. But another subtlety of using parentheses is that they serve as a hint to the interpreter when a statement is still open and will be continued on the next line, hence the `\` is not needed in that case:

```
>>> name = 'Paul'
>>> beatle = False
>>> if (name == 'George' or
...     name == 'Ringo' or
...     name == 'John' or
...     name == 'Paul'):
...     beatle = True
... else:
...     beatle = False
```

TIP

An idiomatic Python version of checking membership is listed below. To check if a value is found across a variety of values, you can throw the values in a set and use the `in` operator:

```
>>> beatles = {'George', 'Ringo', 'John', 'Paul'}
>>> beatle = name in beatles
```

A later chapter will discuss sets further.

Here is an example of using the `not` keyword in a conditional statement:

```
>>> last_name = 'unknown'
>>> if name == 'Paul' and not beatle:
...     last_name = 'Revere'
```

`if` statements

Booleans (`True` and `False`) are often used in *conditional* statements. Conditional statements are instructions that say "if this statement is true, perform a block of code, otherwise execute some other code." Branching statements are used frequently in Python. Sometimes, the "if statement" will check values that contain booleans, other times it will check *expressions* that evaluate to booleans. Another common check is for implicit coercion to "truthy" or "falsey" values:

```
>>> debug = True
>>> if debug:  # checking a boolean
...     print("Debugging")
Debugging
```

`else` statements

An `else` statement can be used in combination with an `if` statement. The body of the `else` statement will execute only if the `if` statement evaluates to `False`. Here is an example of combining an `else` statement with an `if` statement. The school below appears to have grade inflation:

```
>>> score = 87
>>> if score >= 90:
...     grade = 'A'
... else:
...     grade = 'B'
```

Note that the *expression*, `score >= 90`, is evaluated by Python and turns into a `False`. Because the "if statement" was false, the statements under the `else` block are executed, and the grade variable is set to `'B'`.

More choices

The previous example does not work for most schools. You can add more intermediate steps if needed using the `elif` keyword. `elif` is an abbreviation for “else if”. Below is a complete grading scheme:

```
>>> score = 87
>>> if score >= 90:
...     grade = 'A'
... elif score >= 80:
...     grade = 'B'
... elif score >= 70:
...     grade = 'C'
... elif score >= 60:
...     grade = 'D'
... else:
...     grade = 'F'
```

The `if`, `elif`, and `else` statements above each have their own block. Python will start from the top trying to find a statement that evaluates to `True`, when it does, it runs the block and then continues executing at the code following all of the `elif` and `else` blocks. If none of the `if` or `elif` statements are `True`, it runs the block for the `else` statement.

> **NOTE**
>
> The `if` statement can have zero or more `elif` statements. The `else` statement is optional. If it exists, there can only be one.

Whitespace

A peculiarity you may have noticed is the colon (`:`) following the boolean expression in the `if` statement. The lines immediately after the if statement were indented by four spaces. The indented lines are the *block* of code that is executed when the `if` expression evaluates to `True`.

In other languages, an `if` statement might look like this:

```
if (score >= 90) {
    grade = 'A';
}
```

In many of these languages, the curly braces, `{` and `}`, denote the boundaries of the `if` block. Any code between these braces is executed when the score is greater than or equal to 90.

Python, unlike those other languages, uses two things to denote blocks:

- a colon (`:`)
- indentation

If you have programmed in other languages, it is easy to learn the whitespace rules in Python. All you have to do is replace the left curly bracket ({) with a colon (`:`) and indent consistently until the end of the block.

Tip

What is consistent indentation? Normally either tabs or spaces are used to indent code. The Python interpreter only cares about consistency on a per-file basis. It is possible to have a project with multiple files that each use different indentation schemes, but this would be silly.

In Python, using four spaces is the preferred way to indent code. This is described in PEP 8. If you mix tabs and spaces in the same file, you will eventually run into problems.

Although spaces are the preferred mechanism, if you are working on code that already uses tabs, it is better to be consistent. In that case, continue using tabs with the code.

The `python3` executable will complain with a `TabError` when you mix tabs and spaces.

Summary

This chapter discussed the `if` statement. This statement can be used to create arbitrarily complex conditions when combining expressions with `and`, `or`, and `not`.

Blocks, indentation, and whitespace were also discussed. Sometimes when people encounter Python, the required whitespace rules may seem like a nuisance. I've run across such people in my trainings. When asked if and why they indent code in other languages, they reply “Of course, it makes code more readable”. In Python, you emphasize readability, and enforcing whitespace tends to aid in that.

Exercises

1. Write an if statement to determine whether a variable holding a year is a leap year. (See the *Numbers* chapter exercises for the rules for leap years).
2. Write an if statement to determine whether a variable holding an integer is odd.
3. Write an if statement. Look at the indented block and check if your editor indented with tabs or spaces. If it is indented with tabs, configure your editor to indent with spaces. Some editors show tabs differently, if yours does not distinguish between tabs and spaces, an easy way to check if the spacing is a tab is to cursor over it. If the cursor jumps four or eight characters, then it inserted a tab character.

Containers: Lists, Tuples, and Sets

MANY OF THE TYPES DISCUSSED SO FAR HAVE BEEN *SCALARS*, WHICH HOLD A SINGLE value. Integers, floats, and booleans are all scalar values.

Containers hold multiple objects (scalar types or even other containers). This chapter will discuss some of these types—lists, tuples, and sets.

Lists

Lists, as the name implies, are used to hold a list of objects. In Python, a list may hold any type of item and may mix item type. Though in practice, you only store a single item type in a list. Another way to think of a list is that they give an order to a sequence of items. They are a *mutable type*, meaning you can add, remove, and alter the contents of them. There are two ways to create empty lists, one is to invoke the `list` class, and the other is to use the square bracket literal syntax—`[` and `]`:

```
>>> names = list()
>>> other_names = []
```

If you want to have prepopulated lists, you can provide the values in between the square brackets, using the *literal syntax*:

```
>>> other_names = ['Fred', 'Charles']
```

NOTE

The `list` class can also create prepopulated lists, but it is somewhat redundant because you have to pass a list into it:

```
>>> other_names = list(['Fred', 'Charles'])
```

Typically, this class is used to coerce other sequence types into a list. For example, a string is a sequence of characters. If you pass a string into `list`, you get back a list of the individual characters:

```
>>> list('Matt')
['M', 'a', 't', 't']
```

Lists, like other types, have methods that you can call on them (use `dir([])` to see a complete list of them). For example, to add items to the end of a list, use the `.append` method:

```
>>> names.append('Matt')
>>> names.append('Fred')
>>> print(names)
['Matt', 'Fred']
```

Remember that lists are mutable. Python does not return a new list when you append to a list. Notice that the call to `.append` did not return a list (the REPL didn't print anything out). Rather it returns `None` and updates the list in place. (In Python, the default return value is `None` for a function or a method. There is no way to have a method that doesn't return anything).

Sequence indices

A list is one of the *sequence* types in Python. Sequences hold ordered collections of objects. To work effectively with sequences, it is important to understand what an *index* is. Every item in a list has an associated index, which describes its location in the list. For example, the ingredients in

potato chips are potatoes, oil, and salt, and they are normally listed in that order. Potatoes are first in the list, oil is second, and salt is third.

In many programming languages, the first item in a sequence is at index `0`, the second item is at index `1`, the third at index `2`, and so on. Counting beginning with zero is called *zero-based indexing*.

You can access an item in a list using the bracket notation and the index of said item:

```
>>> names[0]
'Matt'

>>> names[1]
'Fred'
```

List insertion

To insert an item at a certain *index*, use the `.insert` method. Calling `.insert` will shift any items following that index to the right:

```
>>> names.insert(0, 'George')
>>> print(names)
['George', 'Matt', 'Fred']
```

The syntax for replacement at an index is the bracket notation:

```
>>> names[1] = 'Henry'
>>> print(names)
['George', 'Henry', 'Fred']
```

To append items to the end of a list use the `.append` method:

```
>>> names.append('Paul')
>>> print(names)
['George', 'Henry', 'Fred', 'Paul']
```

NOTE

CPython's underlying implementation of a list is actually an array of pointers. This provides quick random access to indices. Also, appending and removing at the end of a list is quick (O(1)), while inserting and removing from the middle of a list is slower (O(n)). If you find yourself inserting and popping from the front of a list, `collections.deque` is a better data structure to use.

List deletion

To remove an item, use the `.remove` method:

```
>>> names.remove('Paul')
>>> print(names)
['George', 'Henry', 'Fred']
```

You can also delete by index using the bracket notation:

```
>>> del names[1]
>>> print(names)
['George', 'Fred']
```

Sorting lists

A common operation on lists is *sorting*. The `.sort` method orders the values in the list. This method sorts the list *in place*. It does not return a new, sorted copy of the list, rather it updates the list with the items reordered:

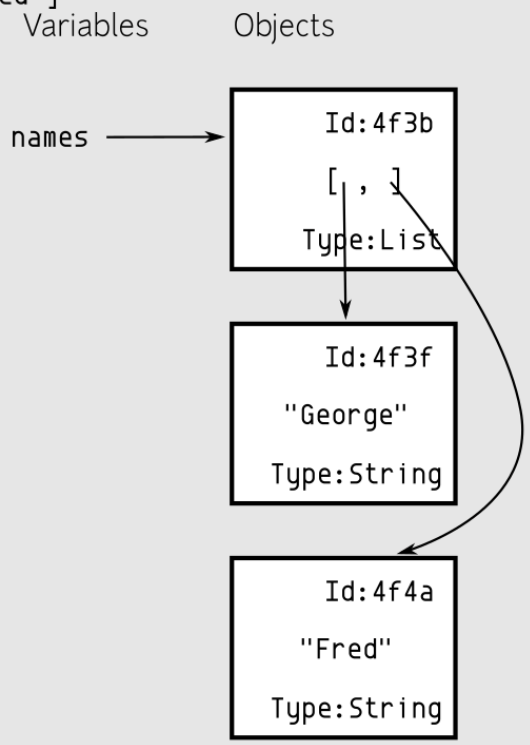
```
>>> names.sort()
>>> print(names)
['Fred', 'George']
```

Sorting Lists

Code

What Computer Does

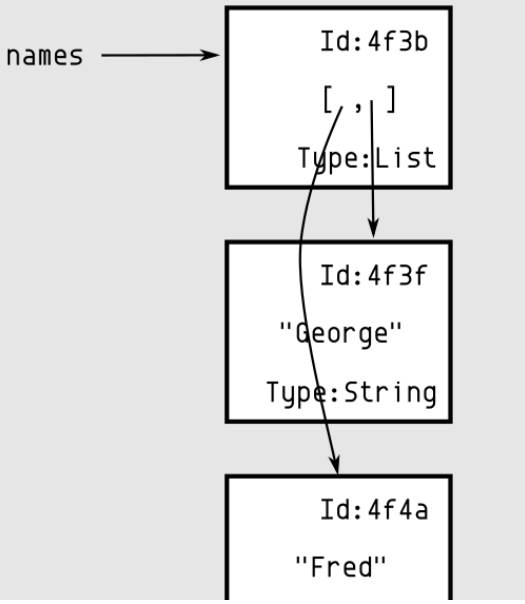
```
names = ['George','Fred']
```

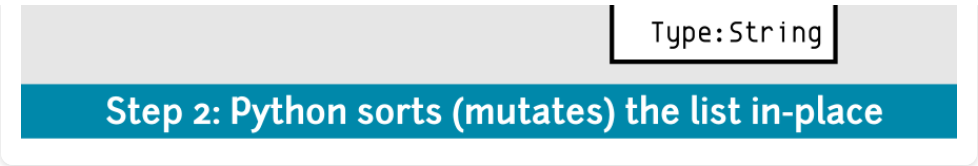


Step 1: Python creates a list

```
names.sort()
```

Doesn't return a list!





This illustrates sorting a list with the `.sort` method. Note that the list is sorted in-place. The result of calling the `.sort` method is the list is mutated, and the method returns `None`.

If the previous order of the list was important, you can make a copy of it before sorting. A more general option for sorting sequences is the `sorted` function. The `sorted` function works with any sequence. It returns a new list that is ordered:

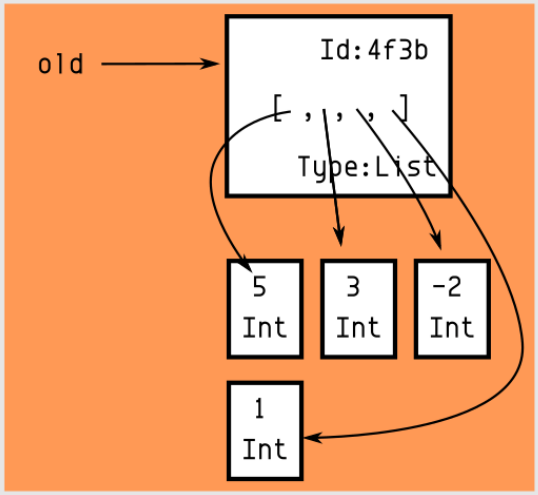
```
>>> old = [5, 3, -2, 1]
>>> nums_sorted = sorted(old)
>>> print(nums_sorted)
[-2, 1, 3, 5]
>>> print(old)
[5, 3, -2, 1]
```

Sorting Lists with Sorted

Code

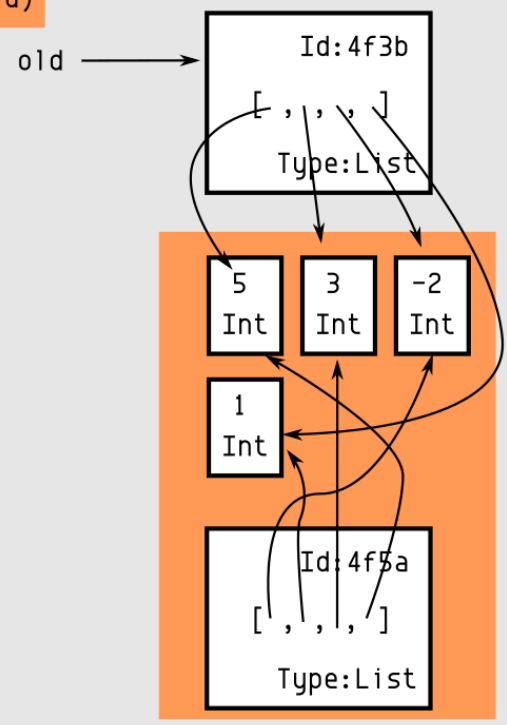
What Computer Does

```
old = [5, 3, -2, 1]
nums_sorted = sorted(old)
```



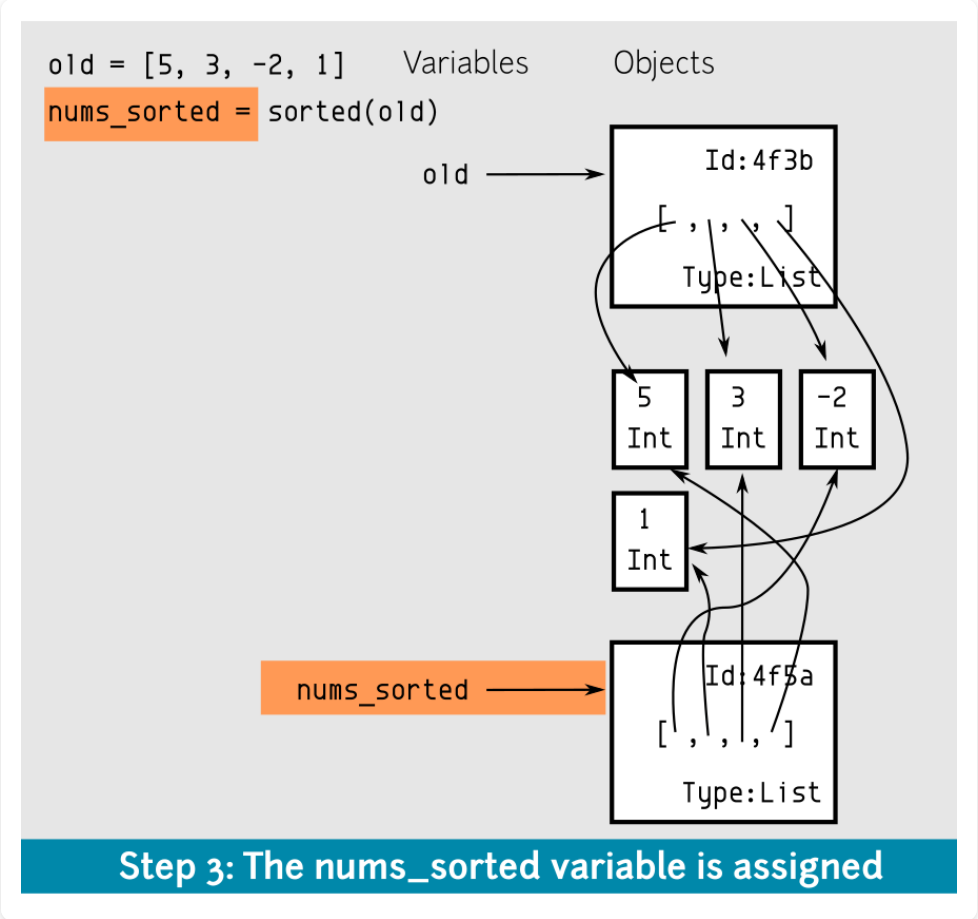
Step 1: Python creates a list

```
old = [5, 3, -2, 1]
nums_sorted = sorted(old)
```



Step 2: The sorted function creates a new list

This illustrates sorting a list by using the `sorted` function. Note that the list is not modified, rather a new list is created. Also, note that Python reused the items of the list. It did not create new items.



Final steps showing that the variable assignment is used to create a variable pointing to the new list. Note that the `sorted` function works with any sequence, not only lists.

Be careful about what you sort. Python wants you to be explicit. In Python 3, when you try to sort a list that contains heterogeneous types, you might get an error:

```
>>> things = [2, 'abc', 'Zebra', '1']
>>> things.sort()
Traceback (most recent call last):
  ...
TypeError: unorderable types: str() < int()
```

Both the `.sort` method and `sorted` function allow arbitrary control of sorting by passing in a function for the `key` parameter. The `key` parameter

can be any callable (function, class, method) that takes a single item and returns something that can be compared.

In this example, by passing in `str` as the `key` parameter, every item in the list is sorted as if it were a string:

```
>>> things.sort(key=str)
>>> print(things)
['1', 2, 'Zebra', 'abc']
```

Useful list hints

As usual, there are other methods found on lists. Do not hesitate to open the Python interpreter and type in a few examples. Remember that `dir` and `help` are your friends.

Tip

`range` is a built-in function that constructs integer sequences. The following will create the numbers zero through four:

```
>>> nums = range(5)
>>> nums
range(5)
```

Python 3 tends to be lazy. Note that `range` does not materialize the list, but rather gives you an iterable that will return those numbers when iterated over. By passing the result into `list` you can see the numbers it would generate:

```
>>> list(nums)
[0, 1, 2, 3, 4]
```

Notice that `range` does not include `5` in its sequence. Many Python functions dealing with final indices mean "up to but not including". (Slices are another example of this you will see later).

If you need to start at a non-zero number, `range` will accept two parameters. When there are two parameters, the first is the starting number (including itself), and the second is the "up to but not including" number:

```
# numbers from 2 to 5
>>> nums2 = range(2, 6)
>>> nums2
range(2, 6)
```

`range` also has an optional third parameter—*stride*. A stride of one (which is the default) means the next number in the sequence that `range` returns should be one more than the previous. A stride of `2` would return every other number. Below is an example that returns only even numbers below eleven:

```
>>> even = range(0, 11, 2)
>>> even
range(0, 11, 2)

>>> list(even)
[0, 2, 4, 6, 8, 10]
```

NOTE

The “up to but not including” construct is more formally known as the *half-open interval* convention. It is commonly used when defining sequences of natural numbers. This construct has a few nice properties:

- The difference between the end and start is the length when dealing with a sequence of consecutive numbers
- Two subsequences can be spliced together cleanly without overlap

Python adopts this half-open interval idiom widely. Here is an example:

```
>>> a = range(0, 5)
>>> b = range(5, 10)
>>> both = list(a) + list(b)
>>> len(both)   # 10 - 0
10

>>> both
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If you are dealing with numeric sequences you might want to follow suit, especially when defining APIs.

Famous computer scientist, Edsger Dijkstra, posited on the reasons for using zero-based indexing, and why it is the correct choice. [10] He ends by saying:

> *Many programming languages have been designed without due attention to this detail.*

Luckily, Python is not one of those languages.

Tuples

Tuples (commonly pronounced as either “two”-ples or “tuh”-ples) are *immutable* sequences. You should think of them as ordered records. Once you create them, you cannot change them. To create a tuple using the *literal syntax*, use parentheses around the members and commas in between. There is also a `tuple` class that you can use to construct a new tuple from an existing sequence:

```
>>> row = ('George', 'Guitar')
>>> row
('George', 'Guitar')

>>> row2 = ('Paul', 'Bass')
>>> row2
('Paul', 'Bass')
```

You can create tuples with zero or one items in them. Though in practice, because tuples hold record type data, this isn't super common. There are two ways to create an empty tuple, using either the `tuple` function or the literal syntax:

```
>>> empty = tuple()
>>> empty
()

>>> empty = ()
>>> empty
()
```

Here are three ways to create a tuple with one item in it:

```
>>> one = tuple([1])
>>> one
(1,)

>>> one = (1,)
>>> one
(1,)

>>> one = 1,
```

```
>>> one
(1,)
```

NOTE

Parentheses are used for denoting the calling of functions or methods in Python. They are also used to specify operator precedence. In addition, they may be used in tuple creation. This overloading of parentheses can lead to confusion. Here is the simple rule, if there is one item in the parentheses, then Python treats the parentheses as normal parentheses (for operator precedence), such as those that you might use when writing `(2 + 3) * 8`. If there are multiple items separated by commas, then Python treats them as a tuple:

```
>>> d = (3)
>>> type(d)
<class 'int'>
```

In the above example, `d` might look like a tuple with parentheses but Python claims it is an integer. For tuples with only one item, you need to put a comma `(,)` following the item—or use the `tuple` class with a single item list:

```
>>> e = (3,)
>>> type(e)
<class 'tuple'>
```

Here are three ways to create a tuple with more than one item. Typically, you would use the last one to be Pythonic. Because it has parentheses it is easier to see that it is a tuple:

```
>>> p = tuple(['Steph', 'Curry', 'Guard'])
>>> p
('Steph', 'Curry', 'Guard')

>>> p = 'Steph', 'Curry', 'Guard'
```

```
>>> p
('Steph', 'Curry', 'Guard')

>>> p = ('Steph', 'Curry', 'Guard')
>>> p
('Steph', 'Curry', 'Guard')
```

Because tuples are immutable you cannot append to them:

```
>>> p.append('Golden State')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute
'append'
```

NOTE

Why the distinction between tuples and lists? Why not use lists since they appear to be a super-set of tuples?

The main difference between the objects is mutability. Because tuples are immutable, they are able to serve as keys in dictionaries. Tuples are often used to represent a record of data such as the row of a database query, which may contain heterogeneous types of objects. Perhaps a tuple would contain a name, address, and age:

```
>>> person = ('Matt', '123 North 456 East', 24)
```

Tuples are used for returning multiple items from a function. Tuples also serve as a hint to the developer that this type is not meant to be modified.

Tuples also use less memory than lists. If you have sequences that you are not mutating, consider using tuples to conserve memory.

Sets

Another container type found in Python is a *set*. A set is an unordered collection that cannot contain duplicates. Like a tuple, it can be instantiated

with a list or anything you can iterate over. Unlike a list or a tuple, a set does not care about order. Sets are particularly useful for two things, removing duplicates and checking membership. Because the lookup mechanism is based on the optimized hash function found in dictionaries, a lookup operation takes very little time, even on large sets.

> **NOTE**
>
> Because sets must be able to compute a hash value for each item in the set, sets can only contain items that are *hashable*. A hash is a semi-unique number for a given object. If an object is hashable, it will always generate the same hash number.
>
> In Python, mutable items are not hashable. This means that you cannot hash a list or dictionary. To hash your own user-created classes, you will need to implement `__hash__` and `__eq__`.

Sets can be specified by passing in a sequence into the `set` class (another coercion class that appears as a function):

```
>>> digits = [0, 1, 1, 2, 3, 4, 5, 6,
...     7, 8, 9]
>>> digit_set = set(digits)   # remove extra 1
```

They can also be created with a literal syntax using `{` and `}`:

```
>>> digit_set = {0, 1, 1, 2, 3, 4, 5, 6,
...     7, 8, 9}
```

As was mentioned, a set is great for removing duplicates. When a set is created from a sequence, any duplicates are removed. The extra `1` was removed from `digit_set`:

```
>>> digit_set
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

To check for membership, you use the `in` operation:

```
>>> 9 in digit_set
True

>>> 42 in digit_set
False
```

NOTE

There is a contains protocol in Python. If a class (set or list, or user-defined class) implements the `__contains__` method (or the iteration protocol), you can use `in` with to check for membership. Due to how sets are implemented, membership tests against them can be much quicker than tests against a list.

Below, a set called `odd` is created. This set will aid in the following examples:

```
>>> odd = {1, 3, 5, 7, 9}
```

Sets provide *set operations*, such as union (`|`), intersection (`&`), difference (`-`), and xor (`^`).

The *difference* (`-`) operator removes items in one set from another:

```
>>> odd = {1, 3, 5, 7, 9}

# difference
>>> even = digit_set - odd
>>> even
{0, 8, 2, 4, 6}
```

Notice that the order of the result is somewhat arbitrary at a casual glance. If the order is important, a set is not the data type you should use.

The *intersection* (`&`) operation (you can think of it as the area where two roads intersect) returns the items found in both sets:

```
>>> prime = set([2, 3, 5, 7])

# those in both
>>> prime_even = prime & even
>>> prime_even
{2}
```

The *union* (`|`) operation returns a set composed of all the items from both sets, with duplicates removed:

```
>>> numbers = odd | even
>>> print(numbers)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Xor (`^`) is an operation that returns a set of items that only are found in one set or the other, but not both:

```
>>> first_five = set([0, 1, 2, 3, 4])
>>> two_to_six = set([2, 3, 4, 5, 6])
>>> in_one = first_five ^ two_to_six
>>> print(in_one)
{0, 1, 5, 6}
```

TIP

Why use a set instead of a list? Remember that sets are optimized for membership and removing duplicates. If you find yourself performing unions or differences among lists, look into using a set instead.

Sets are also much quicker for testing membership. The `in` operator runs faster for sets than lists. However, this speed comes at a cost. Sets do not keep the elements in any particular order, whereas lists and tuples do. If the order is important for you, use a data structure that remembers the order.

Summary

This chapter discussed a few of the built-in container types. You saw lists, which are ordered sequences that are mutable. Remember that a list method does not return a new list, but typically will change the list in place. Lists will support inserting any item, but you typically put items of the same type in a list.

You also saw tuples. Tuples are also ordered like lists, unlike lists, however, tuples do not support mutation. In practice, you use them to represent a record of data, like a row retrieved from a database. When they are representing records, they might hold different types of objects.

Finally, sets were introduced. Sets are mutable, but they are unordered. They are used to remove duplicates and check membership. Because of the hashing mechanism they use, these set operations are fast and efficient. But it requires that the items in the set are hashable.

Exercises

1. Create a list. Append the names of your colleagues and friends to it. Has the `id` of the list changed? Sort the list. What is the first item in the list? What is the second item in the list?
2. Create a tuple with your first name, last name, and age. Create a list, `people`, and append your tuple to it. Make more tuples with the corresponding information from your friends and append them to the list. Sort the list. When you learn about functions, you can use the `key` parameter to sort by any field in the tuple, first name, last name, or age.
3. Create a list of the names of the first names your friends. Create a list with the top ten common names. Use set operations to see if any of your friends have common names.

4. Go to Project Gutenberg [11] and find a page of text from Shakespeare. Paste it into a triple-quoted string. Create another string with a paragraph of text from Ralph Waldo Emerson. Use the string's `.split` method to get a list of words from each. Using set operations find the common words and words unique to both authors.
5. Tuples and lists are similar but have different behavior. Use set operations to find the attributes of a list object that are not in a tuple object.

10 - https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html

11 - https://www.gutenberg.org/

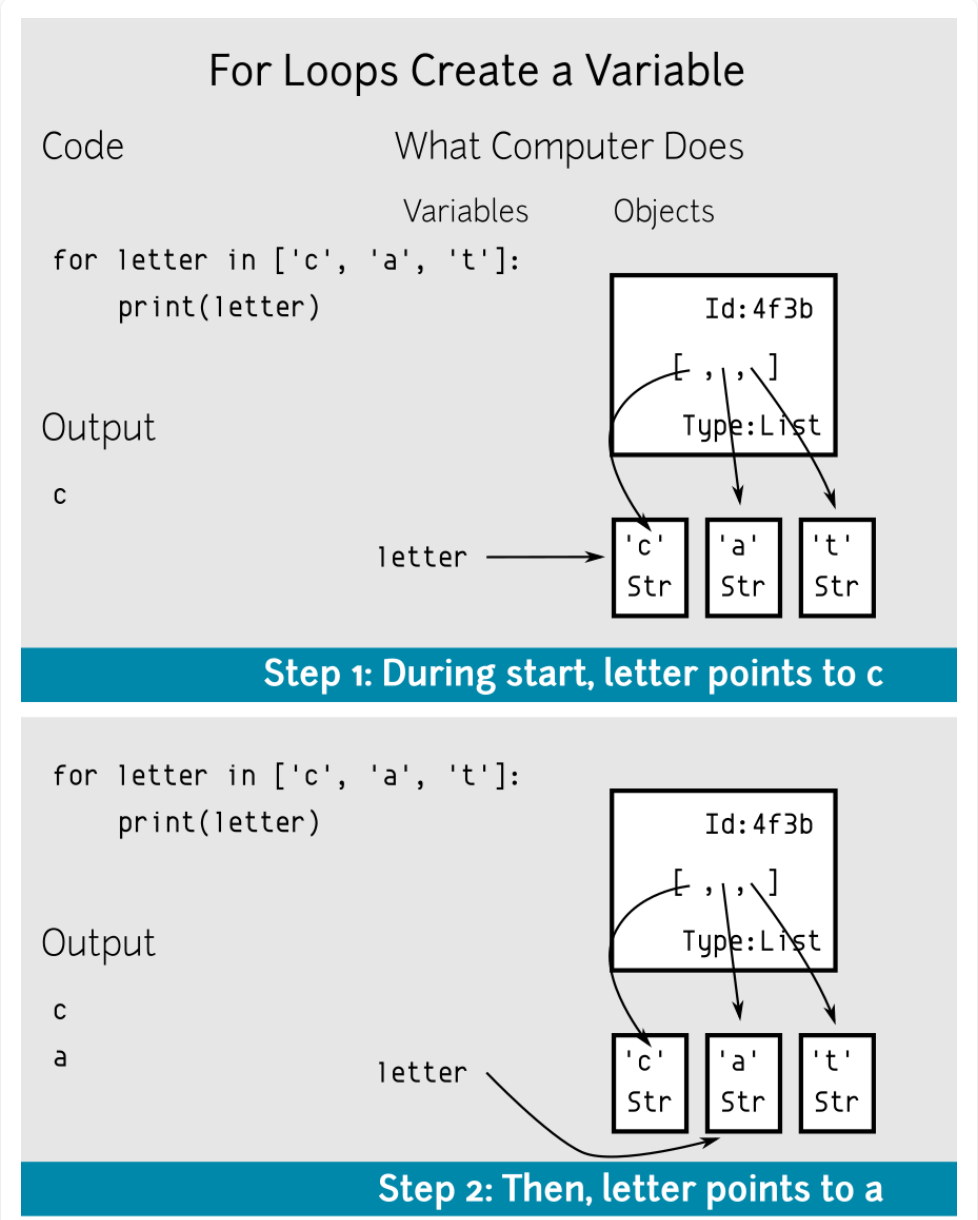
Iteration

A COMMON IDIOM WHEN DEALING WITH SEQUENCES IS TO LOOP OVER THE CONTENTS of the sequence. You might want to filter out one of the items, apply a function to it, or print it out. The `for` loop is one way to do this. Here is an example of printing out the strings in a list:

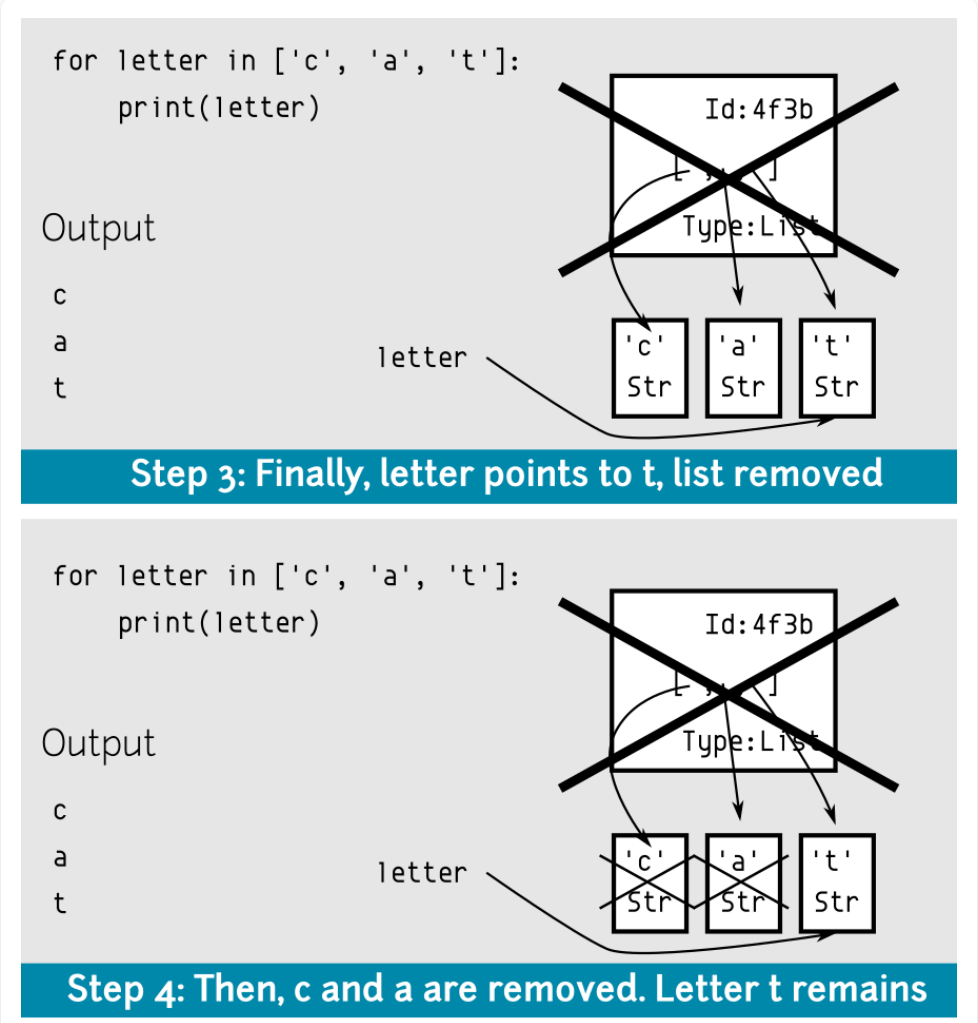
```
>>> for letter in ['c', 'a', 't']:
...     print(letter)
c
a
t

>>> print(letter)
t
```

NOTE

Notice that a `for` loop construct contains a colon (`:`) followed by the indented code. (The indented code is the *block* of the for loop).



This illustrates how a for loop creates a variable. A new variable, `letter`, is created. At first, it points to `'c'`, which is printed. Then the loop continues and the variable points to `'a'`.



The `for` loop continues with the final loop. After `'t'` is printed, the loop exits. At this point, the list literal is garbage collected. Because the only thing pointing to `'c'` and `'a'` was the list, those are also garbage collected. However, the `letter` variable is still pointing to `'t'`. Python doesn't clean up this variable, it hangs around after the `for` loop is done.

During a `for` loop, Python makes a new variable, `letter`, that holds the item of iteration. Note that the value of `letter` is not the index position, but rather the string. This variable is not cleaned up by Python after the `for` loop is done.

Looping with an index

In languages like C, when you loop over a sequence, you do not loop over the items in the sequence, rather you loop over the indices. Using those

indices you can pull out the items at those index values. Here is one way to do that in Python using the built-in functions `range` and `len`:

```
>>> animals = ["cat", "dog", "bird"]
>>> for index in range(len(animals)):
...     print(index, animals[index])
0 cat
1 dog
2 bird
```

The above code is a *code smell*. It indicates that you are not using Python as you should. Usually, the reason for iterating over a sequence is to get access to the items in the sequence, not the indices. But occasionally you will also need the index position of the item. Python provides the built-in `enumerate` function that makes the combination of `range` and `len` unnecessary. The `enumerate` function returns a tuple of (index, item) for every item in the sequence:

```
>>> animals = ["cat", "dog", "bird"]
>>> for index, value in enumerate(animals):
...     print(index, value)
0 cat
1 dog
2 bird
```

Because the tuple returns a pair of index and value, you can use *tuple unpacking* to create two variables, `index` and `value` directly in the for loop. You need to put a comma between the variable names. As long as the tuple is the same length as the number of variables you include in the `for` loop, Python will happily create them for us.

Breaking out of a loop

You may need to stop processing a loop early, without going over every item in the loop. The `break` keyword will jump out of the nearest loop you are in. Below is a program that adds numbers until it comes to the first

negative one. It uses the `break` statement to stop processing when it comes across a negative number:

```
>>> numbers = [3, 5, 9, -1, 3, 1]
>>> result = 0
>>> for item in numbers:
...     if item < 0:
...         break
...     result += item
>>> print(result)
17
```

NOTE

The line:

```
result += item
```

uses what is called *augmented assignment*. It is similar to writing:

```
result = result + item
```

Augmented assignment is slightly quicker as the lookup for the `result` variable only occurs once. Another added benefit is that it is easier to type.

NOTE

The `if` block inside the `for` block is indented eight spaces. Blocks can be nested, and each level needs to be indented consistently.

Skipping over items in a loop

Another common looping idiom is to skip over items. If the body of the `for` loop takes a while to execute, but you only need to execute it for certain items in the sequence, the `continue` keyword comes in handy. The

`continue` statement tells Python to disregard processing of the current item in the for loop and “continue” from the top of the for block with the next value in the loop.

Here is an example of summing all positive numbers:

```
>>> numbers = [3, 5, 9, -1, 3, 1]
>>> result = 0
>>> for item in numbers:
...     if item < 0:
...         continue
...     result = result + item
>>> print(result)
21
```

The `in` statement can be used for membership

You have used the `in` statement in a `for` loop. In Python, the `in` statement can also be used to check for membership. If you want to know if a list contains an item, you can use the `in` statement to check that:

```
>>> animals = ["cat", "dog", "bird"]
>>> 'bird' in animals
True
```

If you want the index location, use the `.index` method:

```
>>> animals.index('bird')
2
```

Removing items from lists during iteration

It was mentioned previously that lists are mutable. Because they are mutable you can add or remove items from them. Also, lists are sequences, so you can loop over them. Do not mutate the list at the same time that you are looping over it.

For example, if you wanted to filter a list of names so it only contained `'John'` or `'Paul'`, this would be the wrong way to do it:

```
>>> names = ['John', 'Paul', 'George',
...     'Ringo']
>>> for name in names:
...     if name not in ['John', 'Paul']:
...          names.remove(name)

>>> print(names)
['John', 'Paul', 'Ringo']
```

What happened? Python assumes that lists will not be modified while they are being iterated over. When the loop got to `'George'`, it removed the name from the list. Internally Python tracks the index location of the for loop. At that point there are only three items in the list, `'John'`, `'Paul'`, and `'Ringo'`. But the for loop internally thinks it is on index location 3 (the fourth item), and there is no fourth item so the loop stops, leaving `'Ringo'` in.

There are two alternatives to the above contruct of removing items from a list during iteration. The first is to collect the items to be removed during a pass through the list.

In a subsequent loop, iterate over only the items that need to be deleted (`names_to_remove`), and remove them from the original list (`names`):

```
>>> names = ['John', 'Paul', 'George',
...  'Ringo']
>>> names_to_remove = []
>>> for name in names:
...     if name not in ['John', 'Paul']:
...          names_to_remove.append(name)

>>> for name in names_to_remove:
...     names.remove(name)

>>> print(names)
['John', 'Paul']
```

Another option is to iterate over a copy of the list. This can be done relatively painlessly using the `[:]` slice copy construct that is covered in the chapter on slicing:

```
>>> names = ['John', 'Paul', 'George',
...  'Ringo']
>>> for name in names[:]:  # copy of names
...     if name not in ['John', 'Paul']:
...          names.remove(name)

>>> print(names)
['John', 'Paul']
```

`else` clauses

A `for` loop can also have an `else` clause. Any code in an `else` block will execute if the `for` loop did not hit a `break` statement. Below is sample code that checks if numbers in a loop are positive:

```
>>> positive = False
>>> for num in items:
...     if num < 0:
...         break
... else:
...     positive = True
```

Note that `continue` statements do not have any effect on whether an `else` block is executed.

The `else` statement is somewhat oddly named. For a `for` loop, it indicates that the whole sequence was processed. A typical use of an `else` statement in a `for` loop is to indicate that an item was not found.

`while` loops

Python will let you loop over a block of code while a condition holds. This is called a *while loop* and you use the `while` statement to create a it. A while loop is followed by an expression that evaluates to `True` or `False`. Then, a colon follows it. Remember what follows a colon (`:`) in Python? Yes, an indented block of code. This block of code will continue to repeat as long as the expression evaluates to `True`. This allows you to easily create an infinite loop.

You usually try to avoid infinite loops because they cause your program to "hang", forever caught in the processing of a loop with no way out. One exception to this is a server, that loops forever, continuing to process requests. Another exception seen in more advanced Python is an infinite generator. A generator behaves like a lazy list and only creates values when you loop over it. If you are familiar with stream processing, you could think of it as a stream. (I don't cover generators in this book, but do in my more advanced book.)

Typically, if you have an object that supports iteration, you use a `for` loop to iterate over the items. You use while loops when you don't have easy access to an iterable object.

A common example is counting down:

```
>>> n = 3
>>> while n > 0:
...     print(n)
...     n = n - 1
3
2
1
```

You can also use the `break` statement to exit a `while` loop:

```
>>> n = 3
>>> while True:
...     print(n)
...     n = n - 1
...     if n == 0:
...         break
```

Summary

This chapter discussed using the `for` loop to iterate over a sequence. You saw that you can loop over lists. You can also loop over strings, tuples, dictionaries, and other data structures. In fact, you can define your own

classes that respond to the `for` statement, by implementing the `.__iter__` method.

A `for` loop creates a variable during iteration. This variable is not garbage collected after the for loop, it sticks around. If your `for` loop is inside of a function, the variable will be garbage collected when the function exits.

The `enumerate` function was introduced. This function returns a sequence of tuples of index, item pairs for the sequence passed into it. If you need to get the index location as well as the item of iteration, use `enumerate`.

Finally, you saw how to break out of loops, continue to the next item of iteration, and use `else` statements. These constructs enable you to adeptly configure your looping logic.

Exercises

1. Create a list with the names of friends and colleagues. Calculate the average length of the names.
2. Create a list with the names of friends and colleagues. Search for the name `John` using a for loop. Print `not found` if you didn't find it. (Hint: use `else`).
3. Create a list of tuples of first name, last name, and age for your friends and colleagues. If you don't know the age, put in `None`. Calculate the average age, skipping over any `None` values. Print out each name, followed by `old` or `young` if they are above or below the average age.

Dictionaries

Dictionaries are a highly optimized built-in type in Python.

You can compare a Python dictionary to an English dictionary. An English dictionary has words and definitions. The purpose of a dictionary is to allow fast lookup of the word in order to find the definition. You can quickly lookup any word by doing a binary search (open up the dictionary to the midpoint, and determine which half the word is in, and repeat).

A Python dictionary also has words and definitions, but you call them *keys* and *values* respectively. The purpose of a dictionary is to provide fast lookup of the keys. You can quickly look for a key and pull out the value associated with it. Like an English dictionary, where it would take a long time to determine the word from a definition (if you didn't know the word beforehand), looking up the value is slow.

In Python 3.6, there is a feature that is new for dictionaries. The keys are now sorted by insertion order. If you are writing Python code that needs to work in prior versions, you will need to remember that prior to 3.6, keys had an arbitrary order (that allowed Python to do quick lookups but wasn't particularly useful to end users).

Dictionary assignment

Dictionaries provide a link from a *key* to a *value*. (Other languages call them *hashes*, *hash maps*, *maps*, or *associative arrays*). Suppose you wanted to store information about an individual. You saw how you could use a tuple to represent a record. A dictionary is another mechanism. Because

dictionaries are built into Python, you can use a literal syntax to create one. This one has first and last names:

```
>>> info = {'first': 'Pete', 'last': 'Best'}
```

NOTE

An alternate way to create a dictionary is with the built-in `dict` class. If you pass a list of tuple pairs into the class, it will return a dictionary:

```
>>> info = dict([('first', 'Pete'),
...              ('last', 'Best')])
```

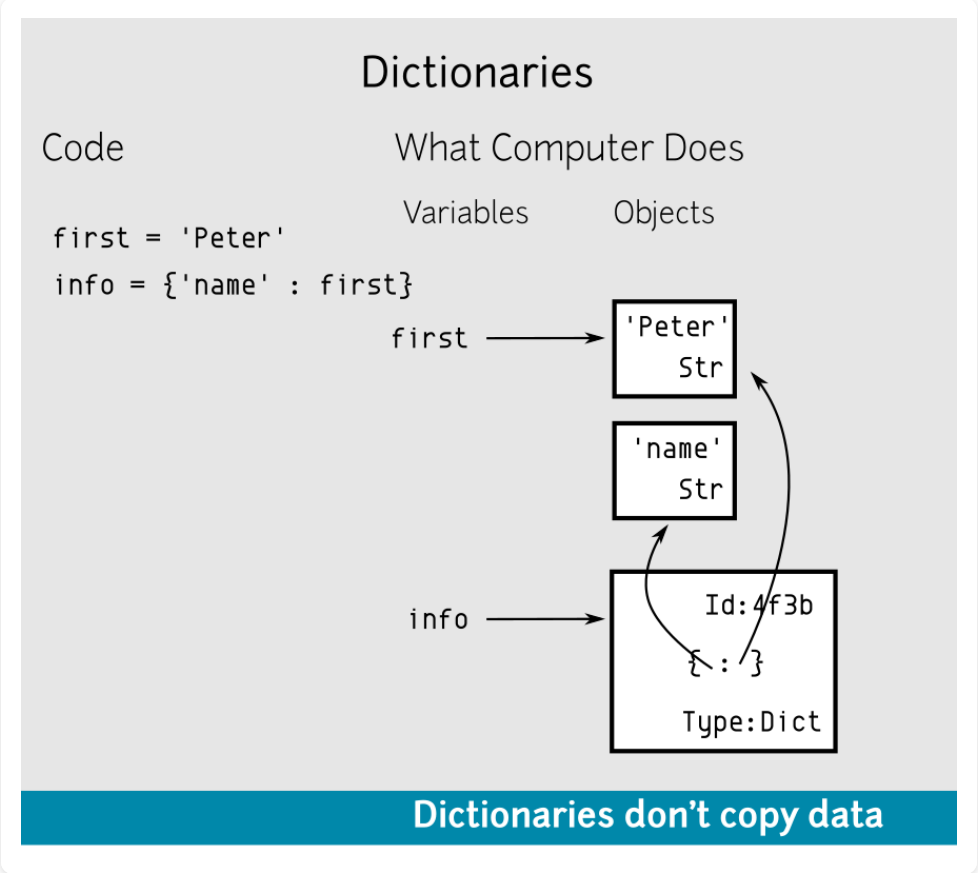
You can also use named parameters when you call `dict`:

```
>>> info = dict(first='Pete', last='Best')
```

If you use named parameters, they must be valid Python variable names, and they will be converted to strings.

You can use index operations to insert values into the dictionary:

```
>>> info['age'] = 20
>>> info['occupation'] = 'Drummer'
```



This illustrates the creation of a dictionary. In this case, you use an existing variable for the value. Note that the dictionary does not copy the variable, but it points to it (increasing the reference count).

In the above example, the keys are `'first'`, `'last'`, `'age'`, and `'occupation'`. For example, `'age'` is the *key* that maps to the integer `20`, the *value*. You can quickly look up the value for `'age'` by performing a lookup with an index operation:

```
>>> info['age']
20
```

On the flip side, finding what the key is that has the value `20` is a slow operation. It would be like giving someone a definition from an English Dictionary and asking what word has that definition.

The above example illustrates the literal syntax for creating an initially populated dictionary. It also shows how the square brackets (index

operations) are used to insert items into a dictionary and pull them out. An index operation associates a key with a value when used in combination with the *assignment operator* (=). When the index operation has no assignment, it looks up the value for a given key.

Retrieving values from a dictionary

As you have seen, the square bracket literal syntax can pull a value out of a dictionary when you use the brackets without assignment:

```
>>> info['age']
20
```

Be careful though, if you try to access a key that does not exist in the dictionary, Python will throw an exception:

```
>>> info['Band']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Band'
```

Because Python likes to be explicit and fail fast, raising an exception is the desired behavior. You know that the `'Band'` key is not in the dictionary. It would be bad if Python decided to return an arbitrary value for a key that was not in the dictionary. Doing so could allow errors to propagate further in your program and mask a logic error.

The `in` operator

Python provides an *operator*, `in`, that allows you to quickly check if a key is in a dictionary:

```
>>> 'Band' in info
False
>>> 'first' in info
True
```

As you have seen, `in` also works with sequences. You can use the `in` statement with a list, set, or string to check for membership:

```
>>> 5 in [1, 3, 6]
False

>>> 't' in {'a', 'e', 'i'}
False

>>> 'P' in 'Paul'
True
```

NOTE

Python 3 removed the `.has_key` method, which provided the same functionality as the `in` statement, but is specific to a dictionary. Hooray for steps forward towards consistency!

The `in` statement will work with most containers. In addition, you can define your own classes that respond to this statement. Your object needs to define the `.__contains__` or be iterable.

Dictionary shortcuts

The `.get` method of a dictionary will retrieve a value for a key. `.get` also accepts an optional parameter to provide a default value if the key is not found. If you wanted the genre to default to `'Rock'`, you could do the following:

```
>>> genre = info.get('Genre', 'Rock')
>>> genre
'Rock'
```

Tip

The `.get` method of dictionaries is one way to get around the `KeyError` thrown when trying to use the bracket notation to pull out a key not found in the dictionary.

It is fine to use this method because you are being explicit about what will happen when the key is missing. You should prefer to fail fast when you aren't being specific about the failing case.

`setdefault`

A useful, but somewhat confusingly named, method of dictionaries is the `.setdefault` method. The method has the same signature as `.get` and initially behaves like it, returning a default value if the key does not exist. In addition to that, it also sets the value of the key to the default value if the key is not found. Because `.setdefault` returns a value, if you initialize it to a mutable type, such as a dict or list, you can mutate the result in place.

`.setdefault` can be used to provide an accumulator or counter for a key. For example, if you wanted to count the number of people with same name, you could do the following:

```
>>> names = ['Ringo', 'Paul', 'John',
...          'Ringo']
>>> count = {}
>>> for name in names:
...     count.setdefault(name, 0)
...     count[name] += 1
```

Without the `.setdefault` method, a bit more code is required:

```
>>> names = ['Ringo', 'Paul', 'John',
...          'Ringo']
>>> count = {}
>>> for name in names:
...     if name not in count:
...         count[name] = 1
```

```
...     else:
...         count[name] += 1

>>> count['Ringo']
2
```

Tip

Performing these counting types of operations was so common that later the `collections.Counter` class was added to the Python standard library. This class can perform the above operations much more succinctly:

```
>>> import collections
>>> count = collections.Counter(['Ringo', 'Paul',
...     'John', 'Ringo'])
>>> count
Counter({'Ringo': 2, 'Paul': 1, 'John': 1})
>>> count['Ringo']
2
>>> count['Fred']
0
```

Here is a somewhat contrived example illustrating mutating the result of `.setdefault`. Assume you want to have a dictionary mapping a name to bands that they played in. If a person named `Paul` is in two bands, the result should map `Paul` to a list containing both of those bands:

```
>>> band1_names = ['John', 'George',
...  'Paul', 'Ringo']
>>> band2_names = ['Paul']
>>> names_to_bands = {}
>>> for name in band1_names:
...     names_to_bands.setdefault(name,
...          []).append('Beatles')
>>> for name in band2_names:
...     names_to_bands.setdefault(name,
...         []).append('Wings')
>>> print(names_to_bands['Paul'])
['Beatles', 'Wings']
```

To belabor the point, without `setdefault` this code would be a bit longer:

```
>>> band1_names = ['John', 'George',
... 'Paul', 'Ringo']
>>> band2_names = ['Paul']
>>> names_to_bands = {}
>>> for name in band1_names:
...     if name not in names_to_bands:
...         names_to_bands[name] = []
...     names_to_bands[name].\
...         append('Beatles')
>>> for name in band2_names:
...     if name not in names_to_bands:
...         names_to_bands[name] = []
...     names_to_bands[name].\
...          append('Wings')
>>> print(names_to_bands['Paul'])
['Beatles', 'Wings']
```

Tip

The `collections` module from the Python standard library includes a handy class—`defaultdict`. This class behaves like a dictionary but it also allows for setting the default value of a key to an arbitrary factory. If the default factory is not `None`, it is initialized and inserted as a value any time a key is missing.

The previous example re-written with `defaultdict` is the following:

```
>>> from collections import defaultdict
>>> names_to_bands = defaultdict(list)
>>> for name in band1_names:
...     names_to_bands[name].\
...         append('Beatles')
>>> for name in band2_names:
...     names_to_bands[name].\
...        append('Wings')
>>> print(names_to_bands['Paul'])
['Beatles', 'Wings']
```

Using `defaultdict` is slightly more readable than using `setdefault`.

Deleting keys

Another common operation on dictionaries is the removal of keys and their corresponding values. To remove an item from a dictionary, use the `del` statement:

```
# remove 'Ringo' from dictionary
>>> del names_to_bands['Ringo']
```

Tip

Python will prevent you from adding to or removing from a dictionary while you are looping over it. Python will throw a `RuntimeError`:

```
>>> data = {'name': 'Matt'}
>>> for key in data:
...     del data[key]
Traceback (most recent call last):
  ...
RuntimeError: dictionary changed size during iteration
```

Dictionary iteration

Dictionaries also support iteration using the `for` statement. By default, when you iterate over a dictionary, you get back the keys:

```
>>> data = {'Adam': 2, 'Zeek': 5, 'Fred': 3}
>>> for name in data:
...     print(name)
Adam
Zeek
Fred
```

Note

The dictionary has a method—`.keys`—that will also list out the keys of a dictionary. In Python 3, the `.keys` method returns a *view*. The view is a window into the current keys found in the dictionary. You can iterate over it, like a list. But, unlike a list, it is not a copy of the keys. If you later remove a key from the dictionary, the view will reflect that change. A list would not.

To iterate over the values of a dictionary, iterate over the `.values` method:

```
>>> for value in data.values():
...   print(value)
2
5
3
```

The result of `.values` is also a view. It reflects the current state of the values found in a dictionary.

To retrieve both key and value during iteration, use the `.items` method, which returns a view:

```
>>> for key, value in data.items():
...     print(key, value)
Adam 2
Zeek 5
Fred 3
```

If you materialize the view into a list, you will see that the list is a sequence of (key, value) tuples—the same thing that `dict` accepts to create a dictionary:

```
>>> list(data.items())
[('Adam', 2), ('Zeek', 5), ('Fred', 3)]
```

Tip

Remember that a dictionary is ordered based on key insertion order. If a different order is desired, you will need to sort the sequence of iteration.

The built-in function `sorted` will return a new sorted list, given a sequence:

```
>>> for name in sorted(data.keys()):
...     print(name)
Adam
Fred
Zeek
```

The `sorted` function has an optional argument, `reverse`, to flip the order of the output:

```
>>> for name in sorted(data.keys(),
...     reverse=True):
...     print(name)
Zeek
Fred
Adam
```

NOTE

It is possible to have keys of different types. The only requirement for a key is that it be *hashable*. For example, a list isn't hashable because you can mutate it, and Python can't generate a consistent hash value for it. If you used a list as a key and then mutated that key, should the dictionary return the value based on the old list, or the new one, or both? Python refuses to guess here and makes you use keys that don't change.

It would be possible to insert items into a dictionary using both integers and strings as keys:

```
>>> weird = {1: 'first',
...     '1': 'another first'}
```

Typically, you don't mix key types, because it is confusing to readers of the code and also makes sorting keys harder. Python 3 won't sort mixed type lists without a `key` function telling Python explicitly how to compare different types. This is one of those areas where Python gives you the ability to do something, but that doesn't mean you should. In the words of Python core developer Raymond Hettinger:

> *Many "Can I do x in Python" questions equate to "Can I stop a car on train tracks when no trains are coming?" Yes you can, No you shouldn't*
>
> *—@raymondh*

Summary

This chapter discussed the dictionary. This data structure is important because it is a building block in Python. Classes, namespaces, and modules

in Python are all implemented using a dictionary under the covers.

Dictionaries provide quick lookup or insertion for a key. You can also do a lookup by value, but it is slow. If you find yourself doing this operation often, it is a code smell that you should use a different data structure.

You saw how to mutate a dictionary. You can insert and remove keys from a dictionary. You can also check a dictionary for membership using the `in` statement.

You looked at some fancier constructs, using `.setdefault` to insert and return values in one operation. You saw that Python includes specialized dictionary classes, `Counter` and `defaultdict` in the `collections` module.

Because dictionaries are mutable, you can delete keys from them using the `del` statement. You can also iterate over the keys of a dictionary using a `for` loop.

Remember that Python 3.6 introduced ordering to the dictionary. The keys are ordered by insertion order, not alphabetic or numeric order.

Exercises

1. Create a dictionary, `info`, that holds your first name, last name, and age.
2. Create an empty dictionary, `phone`, that will hold details about your phone. Add the screen size, memory, OS, brand, etc. to the dictionary.
3. Write a paragraph in a triple-quoted string. Use the `.split` method to create a list of words. Create a dictionary to hold the count for every word in the paragraph.
4. Count how many times each word is used in a paragraph of text from Ralph Waldo Emerson.
5. Write code that will print out the anagrams (words that use the same letters) from a paragraph of text.

6. The *PageRank* algorithm was used to create the Google search engine. The algorithm gives a score to each page based on incoming links. It takes one input: a list of pages that link to other pages. Each page initially gets a score set to 1. Then multiple iterations of the algorithm are run, typically ten. For each iteration:

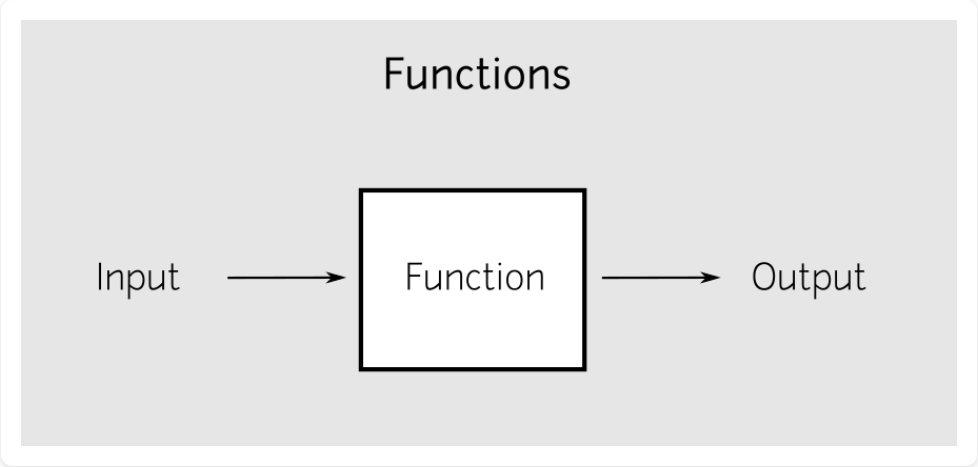
    - *A page transfers its score divided by the number of outgoing links to each page that it links to.*
    - *The score transferred is multiplied by a damping factor, typically set to .85.*

    Write some code to run 10 iterations of this algorithm on a list of tuples of source and destination links, ie:

```
links = [('a', 'b'), ('a', 'c'), ('b', 'c')]
```

Functions

We have come a long way without discussing functions, which are a basic building block of Python programs. Functions are discrete units of code, isolated into their own block. You have actually been using built-in functions along the way such as `dir`, `help`, (and the classes that behave like coercion functions—`float`, `int`, `dict`, `list`, and `bool`).



A function is like a box. It can take input and return output. It can be passed around and reused.

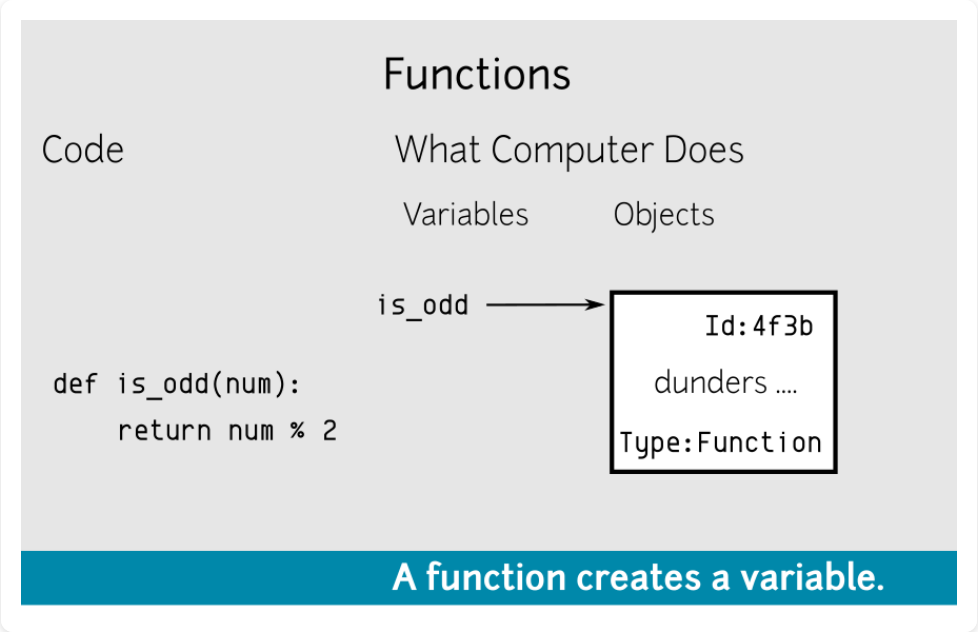
One way to think of a function is as a black box that you can send input to (though input is not required). The black box then performs a series of operations and returns output (it implicitly returns `None` if the function ends without `return` being called). An advantage of a function is that it enables code reuse. Once a function is defined, you can call it multiple times. If you have code that you repeatedly run in your program, rather than copying and pasting it, you can put it in a function once, then call that function multiple times. This gives you less code to reason about, making your programs

easier to understand. It is also easier to change code (or fix bugs) later as you only have to do it in one place.

Here is a simple example of a function. This function, named `add_2`, takes a number as input, adds `2` to that value, and returns the result:

```
>>> def add_2(num):
...     '''
...     return 2 more than num
...     '''
...     result = num + 2
...     return result
```

What are the different components of a function? This whole group of code is known as a *function definition*. First, you see the `def` statement, which is short for *define* (i.e. define a function). Following `def` is a required space (one space is fine) and then the function name—`add_2`. This is the name that is used to *invoke* the function. Synonyms for invoke include call, execute, or run the function. When Python creates a function, it will create a new variable, using the name of the function.



This illustrates the creation of a function. Note that Python creates a new function object, then points a variable to it using the name of the function. Use the `dir` function on the function name to see the attributes of the newly created function.

Following the function name is a left parenthesis followed by `num` and a right parenthesis. The names between the parentheses (there can be an arbitrary number of names, in this case, there is only one), are the input *parameters*. These are the objects that you pass into a function.

After the parentheses comes a colon (`:`). Whenever you see a colon in Python, you should immediately think the colon is followed by an indented block—similar to the body of the `for` loops shown previously. Everything that is indented is the *body* of the function. It is also called a *block of code*.

The body of a function is where the logic lives. You might recognize the first 3 lines of indented code as a triple-quoted string. This is not a comment, though it appears to have comment-like text. It is a string. Python allows you to place a string immediately after the `:`, this string is called a *docstring*. A docstring is a string used solely for documentation. It should describe the block of code following it. The docstring does not affect the logic in the function.

Tip

The `help` function has been emphasized through this book. It is important to note that the `help` function gets its content from the *docstring* of the object passed into it. If you call `help` on `add_2`, you should see the following (provided you actually typed out the `add_2` code above):

```
>>> help(add_2)
Help on function add_2 in module
__main__:

add_2()
    return 2 more than num
(END)
```

Providing docstrings can be useful to you as a reminder of what your code does. If they are accurate, the docstrings are invaluable to anyone trying to use your code.

Following the docstring (note that a docstring is optional), comes the logic of the function. The `result` is calculated. Finally, the function uses a `return` statement to indicate that it has output. A `return` statement is not necessary, and if it is missing a function will return `None` by default. Furthermore, you can use more than one return statement, and they do not even have to be at the end of the function. For example, a conditional block may contain a return statement inside of an `if` block and another in an `else` block.

To recap, the main parts of a function are:

- the `def` keyword
- a function name

- function parameters between parentheses
- a colon (`:`)
- indentation

    - *docstring*
    - *logic*
    - *return statement*

Creating functions is easy. They allow you to reuse code, which makes your code shorter and easier to understand. They are useful to remove global state by keeping short-lived variables inside of the function body. Use functions to improve your code.

Invoking functions

When you call or execute a function, you are *invoking* a function. In Python, you invoke functions by adding parentheses following the function name. This code invokes the newly defined function `add_2`:

```
>>> add_2(3)
5
```

To invoke a function, list its name followed by a left parenthesis, the input parameters, and a right parenthesis. The number of parameters should match the number of parameters in the function declaration. Notice that the REPL implicitly prints the result—the integer `5`. The result is what the `return` statement passes back.

You can pass in whatever object you want to the `add_2` function. However, if that object doesn't support addition of a number, you will get an exception. If you pass a string in, you will see a `TypeError`:

```
>>> add_2('hello')
Traceback (most recent call last):
```

```
  ...
TypeError: must be str, not int
```

Scope

Python looks for variables in various places. You call these places *scopes* (or *namespaces*). When looking for a variable (remember that functions are also variables in Python, as are classes, modules, and more), Python will look in the following locations, in this order:

- Local scope - Variables that are defined inside of functions.
- Global scope - Variables that are defined at the global level.
- Built-in scope - Variables that are predefined in Python.

In the following code is a function that will look up variables in each of the three scopes:

```
>>> x = 2  # Global
>>> def scope_demo():
...     y = 4  # Local to scope_demo
...     print("Local: {}".format(y))
...     print("Global: {}".format(x))
...     print("Built-in: {}".format(dir))

>>> scope_demo()
Local: 4
Global: 2
Built-in: <built-in function dir>
```

Note that after `scope_demo` is invoked, the local variable `y` is garbage collected and no longer available in the global scope:

```
>>> y
Traceback (most recent call last):
  ...
NameError: name 'y' is not defined
```

Variables defined inside of a function or method will be local. In general, you should try to avoid global variables. They make it harder to reason

about code. Global variables are common in books, blogs, and documentation because using them requires writing less code, and allows you to focus on concepts without them being wrapped in a function. Using functions and classes are some of the tools that help you remove global variables, make your code more modular, and easier to understand.

NOTE

Python will allow you to override variables in the global and built-in scope. At the global level, you could define your own variable called `dir`. At that point, the built-in function, `dir`, is *shadowed* by the global variable. You could also do this within a function, and create a local variable that shadows a global or built-in:

```
>>> def dir(x):
...    print("Dir called")

>>> dir('')
Dir called
```

You can use the `del` statement to delete variables in the local or global scope. In practice though, you don't see this, as it is better to avoid shadowing built-ins in the first place:

```
>>> del dir
>>> dir('')
['__add__', '__class__', '__contains__', ... ]
```

HINT

You can use the `locals` and `globals` functions to list these scopes. These return dictionaries with their current scope in them:

```
>>> def foo():
...     x = 1
...     print(locals())

>>> foo()
{'x': 1}
```

The `__builtins__` variable lists the built-in names. If you access its `__dict__` attribute, it will give you a dictionary similar to `globals` and `locals`.

Multiple parameters

Functions can have many parameters. Below is a function that takes two parameters and returns their sum:

```
>>> def add_two_nums(a, b):
...     return a + b
```

Note that Python is a dynamic language and you don't specify the types of the parameters. This function can add two integers:

```
>>> add_two_nums(4, 6)
10
```

It can also add floats:

```
>>> add_two_nums(4.0, 6.0)
10.0
```

Strings too:

```
>>> add_two_nums('4', '6')
'46'
```

Note that the strings use `+` to perform string *concatenation* (joining two strings together).

However, if you try to add a string and a number, Python will complain:

```
>>> add_two_nums('4', 6)
Traceback (most recent call last):
  ...
TypeError: Can't convert 'int' object to str implicitly
```

This is an instance where Python wants you to be more explicit about what operation is desired, and it isn't going to guess for you. If you wanted to add a string type to a number, you might want to convert them to numbers first (using `float` or `int`). Likewise, it might be useful to convert from numbers to strings if concatenation is the desired operation. Python does not implicitly choose which operation to perform. Rather, it throws an error which should force the programmer to resolve the ambiguity.

Default parameters

One cool feature of Python functions is *default parameters*. Like the name implies, default parameters allow you to specify the default values for function parameters. The default parameters are then optional, though you can override them if you need to.

The following function is similar to `add_two_nums`, but will default to adding `3` if the second number is not provided:

```
>>> def add_n(num, n=3):
...     """default to
...     adding 3"""
...     return num + n

>>> add_n(2)
5
>>> add_n(15, -5)
10
```

To create a default value for a parameter, follow the parameter name by an equal sign (=) and the chosen value.

NOTE

Default parameters must be declared after non-default parameters. Otherwise, Python will give you a `SyntaxError`:

```
>>> def add_n(num=3, n):
...     return num + n
Traceback (most recent call last):
  ...
SyntaxError: non-default argument follows
default argument
```

Because default parameters are optional, Python forces you to declare required parameters before optional ones. The above code wouldn't work with an invocation like `add_n(4)` because the required parameter is missing.

Tip

Do not use mutable types (lists, dictionaries) for default parameters unless you know what you are doing. Because of the way Python works, the default parameters are created only once—at function definition time, not at function execution time. If you use a mutable default value, you will end up re-using the same instance of the default parameter during each function invocation:

```
>>> def to_list(value, default=[]):
...     default.append(value)
...     return default

>>> to_list(4)
[4]
>>> to_list('hello')
[4, 'hello']
```

The fact that default parameters are instantiated at the time the function is created is considered a wart by many. This is because the behavior can be surprising. The workaround is to push the creation of the default values from function definition time (which only occurs once) to function runtime (which will create a new value every time the function is executed).

Change the mutable default parameters so that they default to `None`. Then, create an instance of the desired mutable type within the body of the function if the default is `None`:

```
>>> def to_list2(value, default=None):
...     if default is None:
...         default = []
...     default.append(value)
...     return default

>>> to_list2(4)
[4]
>>> to_list2('hello')
['hello']
```

The following code:

```
...     if default is None:
...         default = []
```

Can also be written as a single line using a *conditional expression*:

```
...     default = default if default is not None else []
```

Naming conventions for functions

Function naming conventions are similar to variable naming conventions (and are also found in the PEP 8 document). This is called *snake case*, which is claimed to be easier to read. Function names should:

- be lowercase
- have_an_underscore_between_words
- not start with numbers
- not override built-ins
- not be a keyword

Languages such as Java have adopted the camel case naming convention. In that style you might have a variable named `sectionList` or a method named `hasTimeOverlap`. In Python, you would typically name these `section_list` and `has_time_overlap` respectively. While Python code should follow the PEP 8 conventions, there is also an allowance in PEP 8 for consistency. If the code you are working on has a different style of naming convention, follow suit and be consistent with the existing code. Indeed, the `unittest` module in the standard library still has Java style conventions (because it was originally a port from the Java library junit).

Summary

Functions allow you to encapsulate change and side effects within their body. In this chapter, you learned that functions can take input and return output. There can be multiple input parameters, and you can also provide default values for them.

Remember that Python is based on objects, and when you create a function, you also create a variable with the function name that points to it.

Functions can also have a docstring which is a string written immediately after the declaration. These strings are used to present documentation when the function is passed into the `help` function.

Exercises

1. Write a function, `is_odd`, that takes an integer and returns `True` if the number is odd and `False` if the number is not odd.
2. Write a function, `is_prime`, that takes an integer and returns `True` if the number is prime and `False` if the number is not prime.
3. Write a binary search function. It should take a sorted sequence and the item it is looking for. It should return the index of the item if found. It should return `-1` if the item is not found.
4. Write a function that takes camel cased strings (i.e. *ThisIsCamelCased*), and converts them to snake case (i.e. *this_is_camel_cased*). Modify the function by adding an argument, `separator`, so it will also convert to kebab case (i.e. *this-is-camel-case*) as well.

Indexing and Slicing

Python provides two constructs to pull data out of sequence-like types (lists, tuples, and even strings). These are the *indexing* and *slicing* constructs. Indexing allows you to access single items out of a sequence, while slicing allows you to pull out a sub-sequence from a sequence.

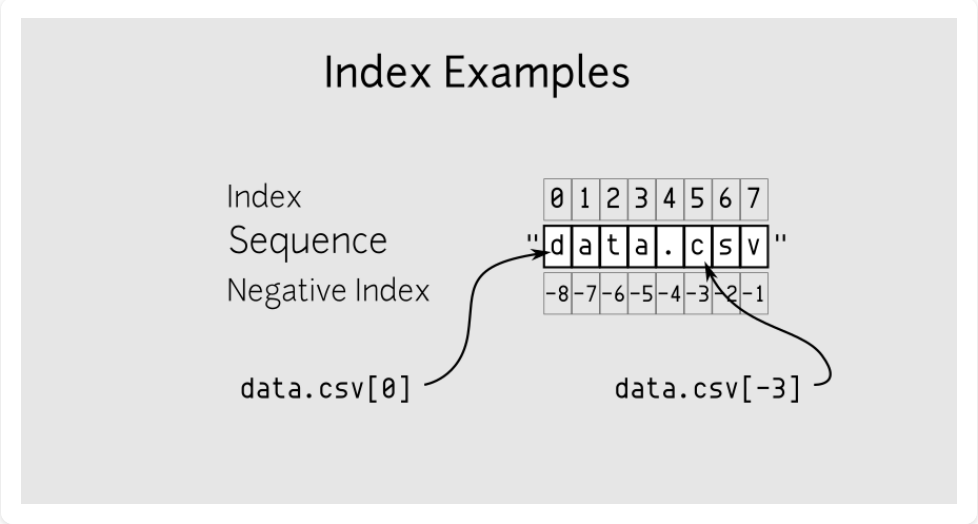
Indexing

You have seen indexing with lists. For example, if you have a list containing pets, you can pull out animals by index:

```
>>> my_pets = ["dog", "cat", "bird"]
>>> my_pets[0]
'dog'
```

Tip

Remember that in Python indices start at `0`. If you want to pull out the first item you reference it by `0`, not `1`. This is called *zero-based indexing*.



This illustrates positive and negative index positions.

Python has a cool feature where you can reference items using negative indices. `-1` refers to the last item, `-2` the second to last item, etc. This is commonly used to pull off the last item in a list:

```
>>> my_pets[-1]
'bird'
```

Guido van Rossum, the creator of Python, tweeted to explain how to understand negative index values:

> *[The] proper way to think of [negative indexing] is to reinterpret* `a[-X]` *as* `a[len(a)-X]`
> *—@gvanrossum*

You can also perform an index operation on a tuple or a string:

```
>>> ('Fred', 23, 'Senior')[1]
23

>>> 'Fred'[0]
'F'
```

Some types, such as sets, don't support index operations. If you want to define your own class that supports index operations, you should implement

the `.__getitem__` method.

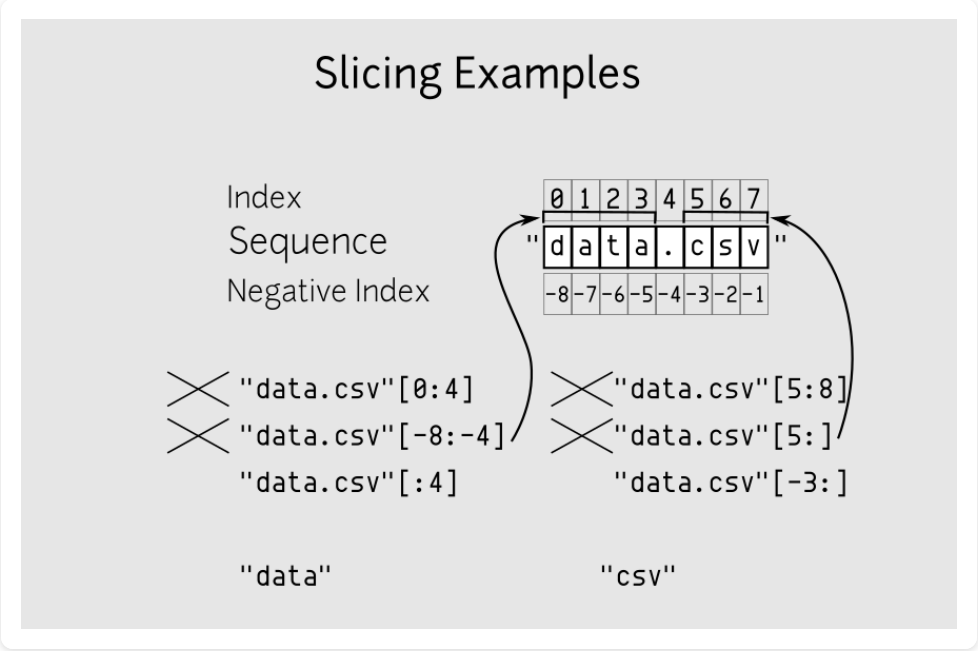
Slicing sub lists

In addition to accepting an integer to pull out a single item, a *slice* may be used to pull out a sub-sequence. A slice may contain the start index, an optional end index, and an optional *stride*, all separated by a colon.

Here is a slice that pulls out the first two items of a list:

```
>>> my_pets = ["dog", "cat", "bird"]  # a list
>>> print(my_pets[0:2])
['dog', 'cat']
```

Remember that Python uses the *half-open interval* convention. The list goes up to but does not include the end index. As mentioned previously, the `range` function also behaves similarly with its second parameter.



This illustrates slicing off the first four characters of a string. Three options are shown. The last option is the preferred way. You don't need the zero index since it is the default. Using negative indices is just silly. Slicing off the final three characters is shown as well. Again, the final example is the idiomatic way to do it. The first two assume the string has a length of eight, while the final code will work with any string that has at least three characters.

When you slice with a colon (`:`), the first index is optional. If the first index is missing, the slice defaults to starting from the first item of the list (the zeroth item):

```
>>> print(my_pets[:2])
['dog', 'cat']
```

You can also use negative indices when slicing. A negative index can be used in the start location or ending location. The index `-1` represents the last item. If you slice up to the last item, you will get everything but that item:

```
>>> my_pets[0:-1]
['dog', 'cat']
>>> my_pets[:-1]  # defaults to 0
['dog', 'cat']

>>> my_pets[0:-2]
['dog']
```

The final index is also optional. If the final index is missing, the slice defaults to the end of the list:

```
>>> my_pets[1:]
['cat', 'bird']
>>> my_pets[-2:]
['cat', 'bird']
```

Finally, you can default both the start and end indices. If both indices are missing, the slice returned will run from the start to the end (which will contain a copy of the list). This is a construct you can use to quickly copy lists in Python:

```
>>> print(my_pets[:])
['dog', 'cat', 'bird']
```

Striding slices

Slices also take a *stride* following the starting and ending indices. The default value for an unspecified stride is `1`. A stride of `1` means take every

item from a sequence between the indices. A stride of `2` would take every second item. A stride of `3` would take every third item:

```
>>> my_pets = ["dog", "cat", "bird"]
>>> dog_and_bird = my_pets[0:3:2]
>>> print(dog_and_bird)
['dog', 'bird']

>>> zero_three_six = [0, 1, 2, 3, 4, 5, 6][::3]
>>> print(zero_three_six)
[0, 3, 6]
```

NOTE

Again, the `range` function has a similar third parameter that specifies stride:

```
>>> list(range(0, 7, 3))
[0, 3, 6]
```

Strides can be negative. A stride of `-1` means that you move backward right to left. To use a negative stride, you should make the start slice greater than the end slice. The exception is if you leave out the start and end indices, a stride of `-1` will reverse the sequence:

```
>>> my_pets[0:2:-1]
[]
>>> my_pets[2:0:-1]
['bird', 'cat']

>>> print([1, 2, 3, 4][::-1])
[4, 3, 2, 1]
```

The next time you are in a job interview and they ask you to reverse a string, you can do it in a single line:

```
>>> 'emerih'[::-1]
'hireme'
```

Of course, they will probably want you to do it in C. Just tell them that you want to program in Python!

Summary

You use indexing operations to pull single values out of sequences. This makes it easy to get a character from a string, or an item from a list or tuple.

If you want subsequences, you can use the slicing construct. Slicing follows the half-open interval and gives you the sequence up to but not including the last index. If you provide an optional stride, you can skip elements in a slice.

Python has a nice feature that allows you to use negative values to index or slice relative to the end of the sequence. This lets you do your operations relative to the length of the sequence, and you don't have to worry about calculating the length of sequence and subtracting from that.

Exercises

1. Create a variable with your name stored as a string. Use indexing operations to pull out the first character. Pull out the last character. Will your code to pull out the last character work on a name of arbitrary length?
2. Create a variable, `filename`. Assuming that it has a three letter extension, and using slice operations, find the extension. For `README.txt`, the extension should be `txt`. Write code using slice operations that will give the name without the extension. Does your code work on filenames of arbitrary length?
3. Create a function, `is_palindrome`, to determine if a supplied word is the same if the letters are reversed.

File Input and Output

Reading and writing files is common in programming. Python makes these operations easy. You can read both text and binary files. You also have a choice of reading a file by the number of bytes, lines, or even the whole file at once. There are similar options available for writing files.

Opening files

The Python function, open, returns a file object. This function has multiple optional parameters:

```
open(filename, mode='r', buffering=-1, encoding=None,
     errors=None, newline=None, closefd=True, opener=None)
```

Hint

Windows paths use `\` as a separator, which can be problematic. Python strings also use `\` as an escape character. If you had a directory named "test" and wrote `"C:\test"`, Python would treat `\t` as a tab character.

The remedy for this is to use raw strings to represent Windows paths. Put an `r` in front of the string:

```
r"C:\test"
```

You typically are only concerned with the first two or three parameters. The first parameter, the filename, is required. The second parameter is the

mode, which determines if you are reading or writing a file and if the file is a text file or binary file. There are various modes.

File modes

Mode	Meaning
`'r'`	Read text file (default)
`'w'`	Write text file (overwrites if exists)
`'x'`	Write text file, throw `FileExistsError` if exists.
`'a'`	Append to text file (write to end)
`'rb'`	Read binary file
`'wb'`	Write binary (overwrite)
`'w+b'`	Open binary file for reading and writing
`'xb'`	Write binary file, throw `FileExistsError` if exists.
`'ab'`	Append to binary file (write to end)

For details on the rest of the options to the `open` function, pass it into `help`. This function has very detailed documentation.

If you are dealing with text files, typically you will use `'r'` as the mode to *read* a file, and `'w'` to *write* a file. Here you open the file `/tmp/a.txt` for writing. Python will create that file, or overwrite it if it already exists:

```
>>> a_file = open('/tmp/a.txt', 'w')
```

The file object, returned from the `open` call, has various methods to allow for reading and writing. This chapter discusses the most commonly used methods. To read the full documentation on a file object pass it to the `help` function. It will list all of the methods and describe what they do.

NOTE

`/tmp` is where Unix systems place temporary files. If you are on Windows, you can inspect the `Temp` variable by typing:

```
c:\> ECHO %Temp%
```

The value is usually:

```
C:\Users\<username>\AppData\Local\Temp
```

Reading text files

Python provides multiple ways to read data from files. If you open a file in text mode (the default), you read strings from the file or you write strings to it. If you open a file in binary mode (using a `'rb'` for reading or `'wb'` for writing), you will read and write *byte* strings.

If you want to read a single line from an existing text file, use the `.readline` method. If you don't specify a mode, Python will default to reading a text file (mode of `r`):

```
>>> passwd_file = open('/etc/passwd')
>>> passwd_file.readline()
'root:x:0:0:root:/root:/bin/bash'
```

Be careful, if you open a file for reading that does not exist, Python will throw an error:

```
>>> fin = open('bad_file')
Traceback (most recent call last):
  ...
IOError: [Errno 2] No such file or
directory: 'bad_file'
```

Tip

The `open` function returns a *file object* instance. This object has methods to read and write data.

Common variable names for file objects are `fin` (file input), `fout` (file output), `fp` (file pointer, used for either input or output) or names such as `passwd_file`. Names like `fin` and `fout` are useful because they indicate whether the file is used for reading or writing respectively.

As illustrated above, the `.readline` method will return one line of a file. You can call it repeatedly to retrieve every line or use the `.readlines` method to return a list containing all the lines.

To read all the contents of a file into one string use the `.read` method:

```
>>> passwd_file = open('/etc/passwd')
>>> print(passwd_file.read())
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
```

You should always close your files when you are done with them by calling `.close`:

```
>>> passwd_file.close()
```

Closing your files is easy. A bit later you will dig into why you should make sure you close them.

Reading binary files

To read a binary file, pass in `'rb'` (read binary) as the mode. When you read a binary file you will not get strings, but byte strings. Don't worry, as the interface for byte strings is very similar to strings. Here are the first

eight bytes of a PNG file. To read eight bytes, you will pass `8` to the `.read` method:

```
>>> bfin = open('img/dict.png', 'rb')
>>> bfin.read(8)
b'\x89PNG\r\n\x1a\n'
```

Notice the `b` in front of the string, specifying that this is a byte string. You can also use `.readline` on binary files and it will read until it gets to a `b'\n'`. Binary strings versus normal strings will be discussed in a later chapter.

Iteration with files

Iteration over sequences was discussed previously. In Python, it is easy to iterate over the lines in a file. When dealing with a text file, you can iterate over the `.readlines` method to get a line at a time:

```
>>> fin = open('/etc/passwd')
>>> for line in fin.readlines():
...     print(line)
```

However, because `.readlines` returns a list, Python will need to read the whole file to create that list, and that could be problematic. Say the file was a large log file, it could possibly consume all of your memory to read it. Python has a trick up its sleeve though. Python allows you to loop over the file instance itself to iterate over the lines of the file. When you iterate directly over the file, Python is lazy, and only reads the lines of text as needed:

```
>>> fin = open('/etc/passwd')
>>> for line in fin:
...     print(line)
```

How can you iterate directly over the file instance? Python has a dunder method, `.__iter__`, that defines what the behavior is for looping over an

instance. It just so happens that for the file class, the `.__iter__` method iterates over the lines in the file.

The `.readlines` method should be reserved for when you are sure that you can hold the file in memory and you need to access the lines multiple times. Otherwise, directly iterating over the file is preferred.

Writing files

To write to a file, you must first open the file in *write* mode. If `mode` is set to `'w'`, the file is opened for writing text data:

```
>>> fout = open('/tmp/names.txt', 'w')
>>> fout.write('George')
```

The above method will try to overwrite the file `/tmp/names.txt` if it exists, otherwise, the file will be created. If you don't have permission to access the file, a `Permission Error` will be thrown. If the path is bad, you will see a `FileNotFoundError`.

Two methods used to place data in a file are `.write` and `.writelines`. The `.write` method takes a string as a parameter and writes it to the file. The `.writelines` method takes a sequence containing string data and writes it to the file.

NOTE

To include newlines in your file, you need to explicitly pass them to the file methods. On Unix platforms, strings passed into `.write` should end with `\n`. Likewise, each of the strings in the sequence that is passed into to `.writelines` should end in `\n`. On Windows, the newline string is `\r\n`.

To program in a cross platform manner, the `linesep` string found in the `os` module defines the correct newline string for the platform:

```
>>> import os
>>> os.linesep  # Unix platform
'\n'
```

Tip

If you are trying this out on the interpreter right now, you may notice that the `/tmp/names.txt` file is empty even though you told Python to write `George` in it. What is going on?

File output is *buffered* by the operating system. In order to optimize writes to the storage media, the operating system will only write data after a certain threshold has been passed. On Linux systems, this is normally 4K bytes.

To force writing the data, you can call the `.flush` method, which *flushes* the pending data to the storage media.

A more heavy-handed mechanism, to ensure that data is written, is to call the `.close` method. This informs Python that you are done writing to the file:

```
>>> fout2 = open('/tmp/names2.txt',
...              'w')
>>> fout2.write('John\n')
>>> fout2.close()
```

Closing files

As mentioned previously, calling `.close` will write the file buffers out to the storage media. The best practice in Python is to always close files after you are done with them (whether for writing or reading).

Here are a few reasons why you should explicitly close your files:

- If you have a file in a global variable, it will never be closed while your program is executing.
- cPython will automatically close your files for you if they are garbage collected. Other Python implementations may not.

- Unless you call `.flush` you won't know when your data is written.
- Your operating system probably has a limit of open files per process.
- Some operating systems won't let you delete an open file.

Python usually takes care of garbage collection for you and you don't have to worry about cleaning up objects. Opening and closing files are exceptions. Python will automatically close the file for you when the file object goes out of scope. But this is a case where you shouldn't rely on garbage collection. Be explicit and clean up after yourself. Make sure you close your files!

Python 2.5 introduced the `with` statement. The `with` statement is used with *context managers* to enforce conditions that occur before and after a block is executed. The `open` function also serves as a context manager to ensure that a file is opened before the block is entered and that it is closed when the block is exited. Below is an example:

```
>>> with open('/tmp/names3.txt', 'w') as fout3:
...     fout3.write('Ringo\n')
```

This is the equivalent of:

```
>>> fout3 = open('/tmp/names3.txt', 'w')
>>> fout3.write('Ringo\n')
>>> fout3.close()
```

Notice that the `with` line ends with a colon. When a line ends with a colon in Python, you always indent the code that follows. The indented content following a colon is called a *block* or the body of the context manager. In the above example, the block consists of writing `Ringo` to a file. Then the block finishes. You can't really see it in the above example, but you know a `with` block ends when the code is dedented. At this point, the context manager kicks in and executes the exit logic. The exit logic for

the file context manager tells Python to automatically close the file for you when the block is finished.

> **TIP**
>
> Use the `with` construct for reading and writing files. It is a good practice to close files, and if you use the `with` statement when using files, you do not have to worry about closing them. The `with` statement automatically closes the file for you.

Designing around files

You have seen how to use functions to organize and compartmentalize complicated programs. One benefit to using functions is that you can re-use those functions throughout your code. Here is a tip for organizing functions that deal with files.

Assume that you want to write code that takes a filename and creates a sequence of the lines from the file with the line number inserted before every line. At first thought it might seem that you want the API (application programming interface) for your functions to accept the filename of the file that you want to modify, like this:

```
>>> def add_numbers(filename):
...     results = []
...     with open(filename) as fin:
...         for num, line in enumerate(fin):
...             results.append(
... '{0}-{1}'.format(num, line))
...     return results
```

This code will work okay. But what will happen when you are required to insert line numbers in front of lines from a source other than a file? If you want to test the code, now you have access to the file system. One solution

is to refactor the `add_numbers` function so that it will only open the file in a context manager, and then call another function, `add_nums_to_seq`. This new function contains logic that operates on a sequence rather than depending on a filename. Since a file behaves as a sequence of strings, you preserve the original functionality:

```
>>> def add_numbers(filename):
...     with open(filename) as fin:
...         return add_nums_to_seq(fin)

>>> def add_nums_to_seq(seq):
...     results = []
...     for num, line in enumerate(seq):
...         results.append(
... '{0}-{1}'.format(num, line))
...     return results
```

Now you have a more general function, `add_nums_to_seq`, that is easier to test and reuse, because instead of depending on a filename, it depends on a sequence. You can pass in a list of strings, or create a fake file to pass into it.

HINT

There are other types that also implement the file-like interface (`read` and `write`). Any time you find yourself coding with a filename, ask yourself if you may want to apply the logic to other sequence-like things. If so, use the previous example of refactoring the functions to obtain code that is much easier to reuse and test.

Summary

Python provides a single function to interact with both text and binary files, `open`. By specifying the mode, you can tell Python whether you want to read or write to the file. When you are dealing with text files, you read and

write strings. When you are dealing with binary files, you read and write byte strings.

Make sure you close your files. Using the `with` statement is the idiomatic way to do this. Finally, make sure your functions deal with sequences of data instead of filenames as this makes your code more generally useful.

Exercises

1. Write a function to write a comma separated value (CSV) file. It should accept a filename and a list of tuples as parameters. The tuples should have a name, address, and age. The file should create a header row followed by a row for each tuple. If the following list of tuples was passed in:

   ```
   [('George', '4312 Abbey Road', 22),
    ('John', '54 Love Ave', 21)]
   ```

   it should write the following in the file:

   ```
   name,address,age
   George,4312 Abbey Road,22
   John,54 Love Ave,21
   ```

2. Write a function that reads a CSV file. It should return a list of dictionaries, using the first row as key names, and each subsequent row as values for those keys. For the data in the previous example it would return:

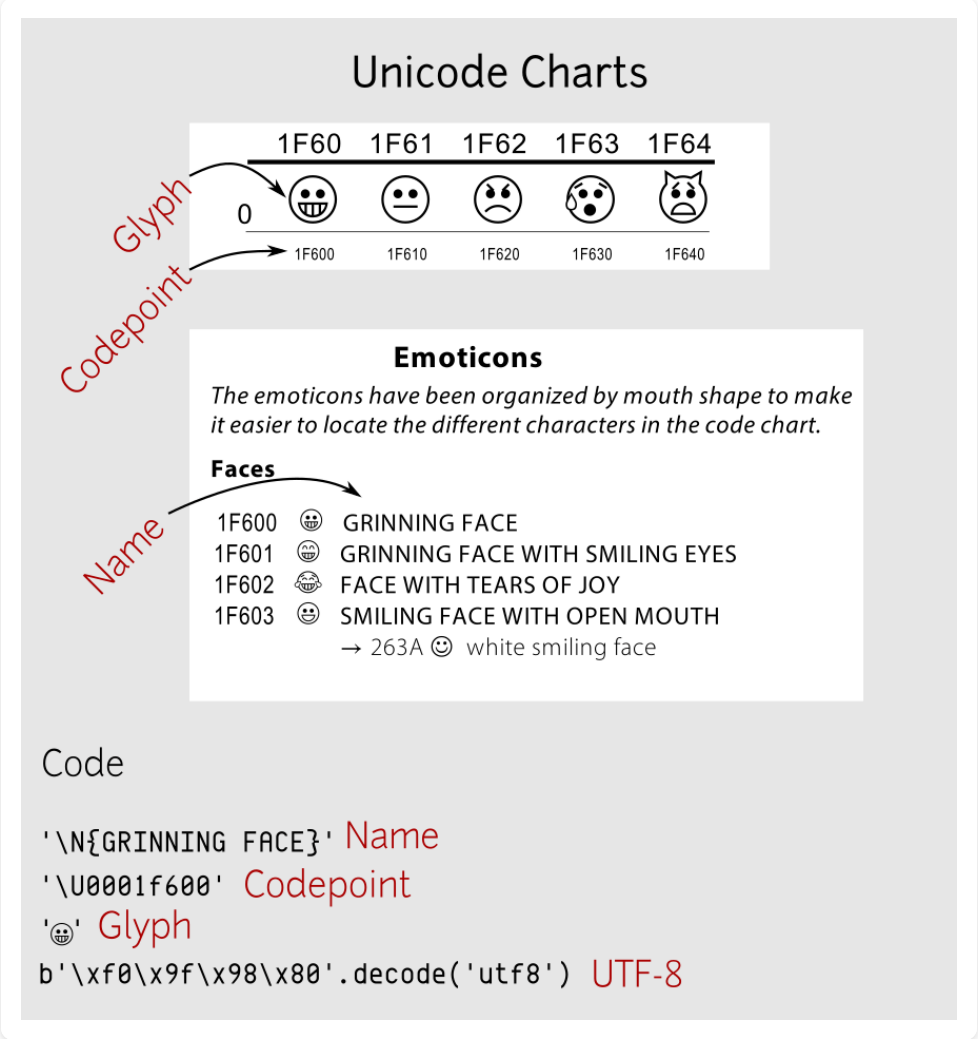
   ```
   [{'name': 'George', 'address': '4312 Abbey Road', 'age': 22},
    {'name': 'John', 'address': '54 Love Ave', 'age': 21}]
   ```

Unicode

We've seen strings all over the place, but we haven't really talked about one of the biggest changes that came in Python 3, Unicode strings! Python 2 had support for Unicode strings, but you needed to explicitly create them. This is no longer the case, everything is Unicode.

Background

What is Unicode? It is a standard for representing glyphs (the characters that create most written language, as well as symbols and emoji). The standard can be found on the Unicode website [12], and is frequently updated. The standard consists of various documents or charts that map *code points* (hexadecimal numbers such as `0048` or `1F600`) to glyphs (such as `H` or 😀), and names (*LATIN CAPITAL H* and *GRINNING FACE*). The code points and names are unique, though many glyphs may look very similar.



This illustrates how to read code charts found at unicode.org. The charts have tables listing glyphs by their hex code points. Following that table is another table with the codepoint, glyph, and name. You can use a glyph, the name, or the code point. If the code point has more than 4 digits, you need to use capital `U` and left pad with `0` until you have 8 digits. If it has 4 or less digits, you need to use a lowercase `u` and left pad with `0` until you have 4 digits. Also shown is an example decoding the UTF-8 byte string to the corresponding glyph.

Here is a condensed history. As computers became prevalent, different providers had different schemes to map binary data to string data. One *encoding*, ASCII, would use 7 bits of data to map to 128 symbols and control codes. That works fine in a Latin character-centric environment such as English. Having 128 different glyphs would provide enough space for lowercase characters, uppercase characters, digits, and common punctuation.

As support for non-English languages became more common, ASCII was not sufficient. Windows systems through Windows 98 supported an encoding called Windows-1252, that had support for various accented characters, and symbols such as the Euro sign.

All of these encoding schemes provided a one-to-one mapping of bytes to a character. In order to support Chinese, Korean, and Japanese scripts, many more than 128 symbols were needed. Using four bytes provided support for over 4 billion characters. But this encoding came at a cost. For the majority of people using only ASCII-centric characters, requiring those characters to be encoded in four times as much data seemed like a colossal waste of memory.

A compromise that provided both the ability to support all characters but not waste memory, was to stop encoding characters to a sequence of bits. Rather, the characters were abstracted. Each character instead mapped to a unique *code point* (that has a hex value and a unique name). Various encodings then mapped these code points to bit encodings. Unicode maps from character to a code point—it is not the encoding. For different contexts, an alternate encoding might provide better characteristics.

One encoding, UTF-32, uses four bytes to store the character information. This representation is easy for low-level programmers to use as indexing operations are trivial. But it uses four times the amount of memory as ASCII to encode a Latin sequence of letters.

The notion of variable width encodings also helped alleviate memory waste. UTF-8 is one such encoding. It uses between one and four bytes to represent a character. In addition, UTF-8 is backward compatible with ASCII. UTF-8 is the most common encoding on the web, and is a great choice for encoding since many applications and operating systems know how to deal with it.

Tip

You can find out what the preferred encoding is on your machine by running the following code. Here is mine on a 2015 Mac:

```
>>> import locale
>>> locale.getpreferredencoding(False)
'UTF-8'
```

Let's be clear. UTF-8 is an encoding of bytes for Unicode code points. To say that UTF-8 is Unicode is sloppy at best and connotes a lack of understanding of character encodings at worst. In fact, the name is derived from Unicode Transformation Format - 8-bit, ie. it is a format for Unicode.

Let's look at an example. The character named, *SUPERSCRIPT TWO*, is defined in the Unicode standard as code point *U+00b2*. It has a *glyph* (a written representation) of ². ASCII has no way to represent this character. Windows-1252 does, it is encoded as the hex byte *b2* (which happens to be the same as the code point, note that this is not guaranteed). In UTF-8 it is encoded as *c2 b2*. Likewise, UTF-16 encodes it as *ff fe b2 00*. There are multiple encodings for this Unicode code point.

Basic steps in Python

You can create Unicode strings in Python. In fact, you have been doing it all along. Remember in Python 3, strings are Unicode strings. But you probably want to create a string that has a non-ASCII character. Below is code to make a string that has x squared (x²) in it. If you can find the glyph you want to use, you can copy that character in into your code:

```
>>> result = 'x²'
```

This works in general. Though you might run into problems if your font does not support said glyph. Then you will get what is sometimes called

tofu because it looks either like an empty box or a diamond with a question mark. Another way to include non-ASCII characters is to use the hex Unicode code point following `\u`:

```
>>> result2 = 'x\u00b2'
```

Note that this string is the same as the previous string:

```
>>> result == result2
True
```

Finally, you can use the name of the code point inside of the curly braces in `\N{}`:

```
>>> result3 = 'x\N{SUPERSCRIPT TWO}'
```

All of these work. They all return the same Unicode string:

```
>>> print(result, result2, result3)
x² x² x²
```

The third option is a little verbose and requires a little more typing. But if you don't speak native Unicode or have font support, it is perhaps the most readable.

Tip

The Python help documentation has an entry for Unicode. Type `help()`, then `UNICODE`. It actually doesn't discuss Unicode that much, rather the different ways to create Python strings.

Encoding

Perhaps one of the keys to grokking Unicode in Python is to understand that you *encode a Unicode string to a byte string*. You never encode a byte string. You *decode a byte string to a Unicode string*. Likewise, you never

decode a Unicode string. Another way of looking at encoding and decoding is that encoding transforms a human-readable or meaningful representation into an abstract representation meant for storage (Unicode to bytes or letter to bytes), and decoding transforms that abstract representation back to a human readable or meaningful representation.

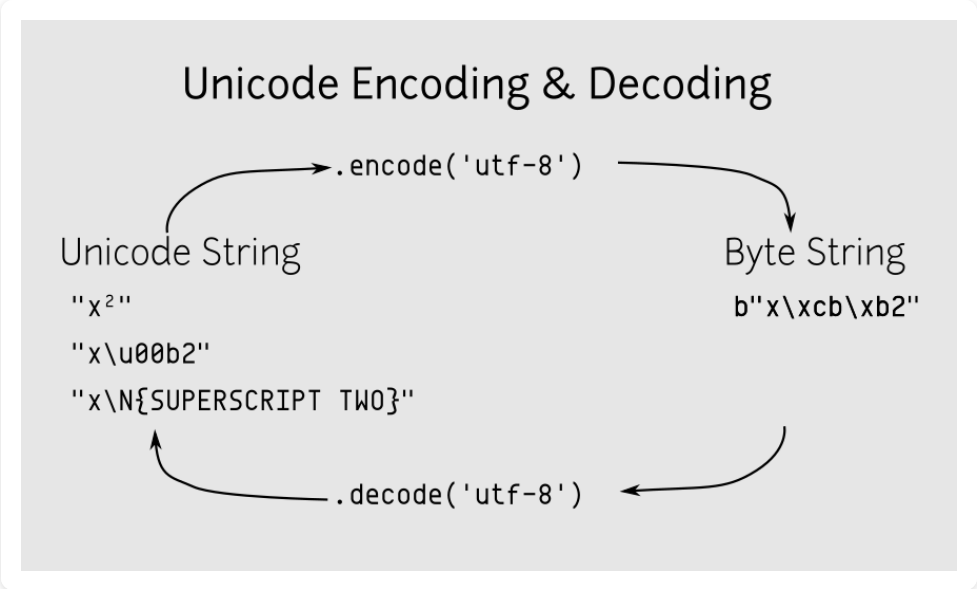


Image illustrates the *encoding* (in this case using UTF-8) of a Unicode string into its byte representation, and then the *decoding* of the same byte string back into Unicode (also using UTF-8). Note that you should be explicit when decoding as there are other encodings that if used, might produce erroneous data or mojibake (character transformation).

Given a Unicode string, you can call the `.encode` method to see the representation in various encodings. Let's start off with UTF-8:

```
>>> x_sq = 'x\u00b2'
>>> x_sq.encode('utf-8')
b'x\xc2\xb2'
```

If Python does not support an encoding, it will throw a `UnicodeEncodeError`. This means that the Unicode code points are not supported in that encoding. For example, ASCII, does not have support for the squared character:

```
>>> x_sq.encode('ascii')
Traceback (most recent call last):
```

```
  ...
UnicodeEncodeError: 'ascii' codec can't encode character
'\xb2' in position 1: ordinal not in range(128)
```

If you use Python long enough, you will probably encounter this error. It means that the specified encoding does not have the ability to represent all of the characters. If you are certain that you want to force Python to this encoding, there are a few different options you can provide to the `errors` parameter. To ignore characters Python can't encode, use `errors='ignore'`:

```
>>> x_sq.encode('ascii', errors='ignore')
b'x'
```

If you specify `errors='replace'`, Python will insert a question mark for the unsupported bytes:

```
>>> x_sq.encode('ascii', errors='replace')
b'x?'
```

NOTE

The `encodings` module has a mapping of aliases of encodings that Python supports. There are 99 encodings in Python 3.6. Most modern applications try to stay in UTF-8, but if you need support for other encodings, there is a good chance that Python has support for it.

This mapping is found in `encodings.aliases.aliases`. Alternatively, the documentation on the module at the Python website [13] has a table of the encodings.

Here are some other possible encodings for this string:

```
>>> x_sq.encode('cp1026')  # Turkish
b'\xa7\xea'
>>> x_sq.encode('cp949')   # Korean
b'x\xa9\xf7'
```

```
>>> x_sq.encode('shift_jis_2004')  # Japanese
b'x\x85K'
```

Though Python supports many encodings, their usage is becoming more rare. Typically they are used for legacy applications. Today, most applications use UTF-8.

Decoding

Decoding has a specific meaning in Python. It means to take a sequence of bytes and create a Unicode string from them. In Python, you never encode bytes, you decode them (in fact there is no `.encode` method on bytes). If you can remember that rule, it can save a bit of frustration when dealing with non-ASCII characters.

Let's assume that you have the UTF-8 byte sequence for `x²`:

```
>>> utf8_bytes = b'x\xc2\xb2'
```

When you are dealing with character data represented as bytes, you want to get it into a Unicode string as soon as possible. Typically, you will only deal with strings in your application and use bytes as a serialization mechanism (i.e. when persisting to files or sending over the network). If you received these bytes somehow and the framework or library that you were using did not convert it into a string for you, you could do the following:

```
>>> text = utf8_bytes.decode('utf-8')
>>> text
'x²'
```

Another thing to note is that in general, you cannot divine what a sequence of bytes was encoded as. You know that the sequence in the code listed is UTF-8 because you just created it, but if unidentified bytes were sent to you from another source then it could certainly be another encoding.

NOTE

You need to be told the encoding. Otherwise, you might try to decode with the wrong encoding and have bad data. When this happens, the miscoded string is called *mojibake*. This is a Japanese word that means character mutation, but mojibake sounds cooler.

Here are some examples of erroneous decoding:

```
>>> b'x\xc2\xb2'.decode('cp1026')  # Turkish
'ÌB¥'

>>> b'x\xc2\xb2'.decode('cp949')  # Korean
'x짼'

>>> b'x\xc2\xb2'.decode('shift_jis_2004')  # Japanese
'xツイ'
```

But some encodings don't support all byte sequences. If you decode with the wrong encoding, sometimes instead of getting bad data, you get an exception. For example, ASCII, does not support this byte sequence:

```
>>> b'x\xc2\xb2'.decode('ascii')
Traceback (most recent call last):
  ...
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc2
in position 1: ordinal not in range(128)
```

A `UnicodeDecodeError` means that you were trying to convert a byte string to a Unicode string and the encoding did not have support to create Unicode strings for all of the bytes. Typically, this means that you are using the wrong encoding. For best results, try to determine and use the correct encoding.

If you really don't know the encoding, you can pass the `errors='ignore'` parameter, but then you are losing data. This should only be used as a last resort:

```
>>> b'x\xc2\xb2'.decode('ascii', errors='ignore')
'x'
```

Unicode and files

When you read a text file, Python will give you Unicode strings. By default, Python will use the default encoding (`locale.getpreferredencoding(False)`). If you want to encode a text file in another encoding, you can pass that information with the `encoding` parameter to the `open` function.

Likewise, you can specify the encoding when you open a file for writing. Remember, you should treat encodings as a serialization format (used to transfer information over the internet or store the data in a file). Here are two examples of writing to a file. With the first file no encoding is defined, and it therefore defaults to UTF-8 (per the system's default encoding). In the second example, CP949 (Korean) is set as the encoding parameter:

```
>>> with open('/tmp/sq.utf8', 'w') as fout:
...     fout.write('x²')

>>> with open('/tmp/sq.cp949', 'w', encoding='cp949') as fout:
...     fout.write('x²')
```

From the UNIX terminal, look at the files. You see that they have different contents, because they are using different encodings:

```
$ hexdump /tmp/sq.utf8
0000000 78 c2 b2
0000003

$ hexdump /tmp/sq.cp949
0000000 78 a9 f7
0000003
```

You can read the data back:

```
>>> data = open('/tmp/sq.utf8').read()
>>> data
'x²'
```

Remember, that in Python, *explicit is better than implicit*. When dealing with encodings, you need to be specific. Python on a UTF-8 system will try and decode from UTF-8 when you read a file. This is fine if the file is encoded with UTF-8, but if it is not, you will either get mojibake, or an error:

```
>>> data = open('/tmp/sq.cp949').read()
Traceback (most recent call last):
  ...
UnicodeDecodeError: 'utf-8' codec can't decode byte
0xa9 in position 1: invalid start byte
```

If you are explicit and tell Python that this file was encoded with the CP949 encoding, you will get the correct data from this file:

```
>>> data = open('/tmp/sq.cp949', encoding='cp949').read()
>>> data
'x²'
```

If you are dealing with text files that contain non-ASCII characters, make sure you specify their encoding.

Summary

Unicode is a mapping of code points to glyphs. In Python, strings can hold Unicode glyphs. You can *encode* the Unicode string to a byte string using various encodings. You never decode Python strings.

When you read a text file, you can specify the encoding to make sure that you get a Unicode string containing the correct characters. When you write text files, you can use the `encoding` parameter to declare the encoding to use.

UTF-8 is the most popular encoding these days. Unless you have a reason to use another encoding, you should default to using UTF-8.

Exercises

1. Go to http://unicode.org and download a chart that has code points on it. Choose a non-ASCII character and write Python code to print the character by both the code point and name.
2. There are various Unicode characters that appear to be upside down versions of ASCII characters. Find a mapping of these characters (they should be a search away). Write a function that takes a string with ASCII characters, and returns the upside down version of that string.
3. Write the upside down version of your name to a file. What are some of the possible encodings that support your name? What are some encodings that don't?
4. Smart quotes (or curly quotes) are not supported with ASCII. Write a function that takes an ASCII string as input and returns a string where the double quotes are replaced with smart quotes. For example, the string *Python comes with "batteries included"*, should become *Python comes with “batteries included”* (if you look closely, you will see that with smart quotes, the left quotes curve differently than the right quotes).
5. Write a function that takes text with old school emojis in it (`:)`, `:P`, etc.). Using the emoji chart [14], add code to your function that replaces the text emojis, with Unicode versions found in the chart.

12 - https://unicode.org
13 - https://docs.python.org/3/library/codecs.html
14 - http://unicode.org/emoji/charts/full-emoji-list.html

Classes

Strings, dictionaries, files, and integers are all objects. Even functions are objects. In Python, almost everything is an object. There are some exceptions, keywords (such as `in`) are not objects. Also, variable names are not objects, but they do point to them. This chapter will delve deeper into what an object really is.

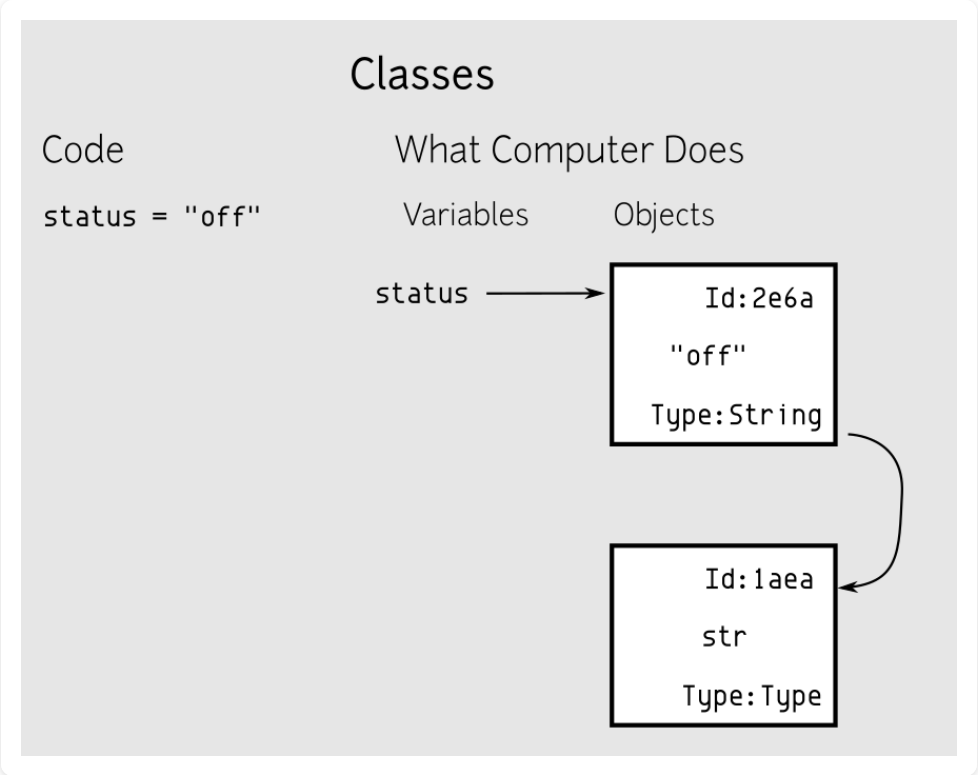
Object is a somewhat ambiguous term. One definition of "Object-Oriented Programming" is using structures to group together data (state) and methods (functions to alter state). Many object-oriented languages such as C++, Java, and Python use *classes* to define what state an object can hold and the methods to alter that state. Whereas classes are the definition of the state and methods, *instances* are occurrences of said classes. Generally when people say *object* they mean an instance of a class.

In Python, `str` is the name of the class used to store strings. The `str` class defines the methods of strings.

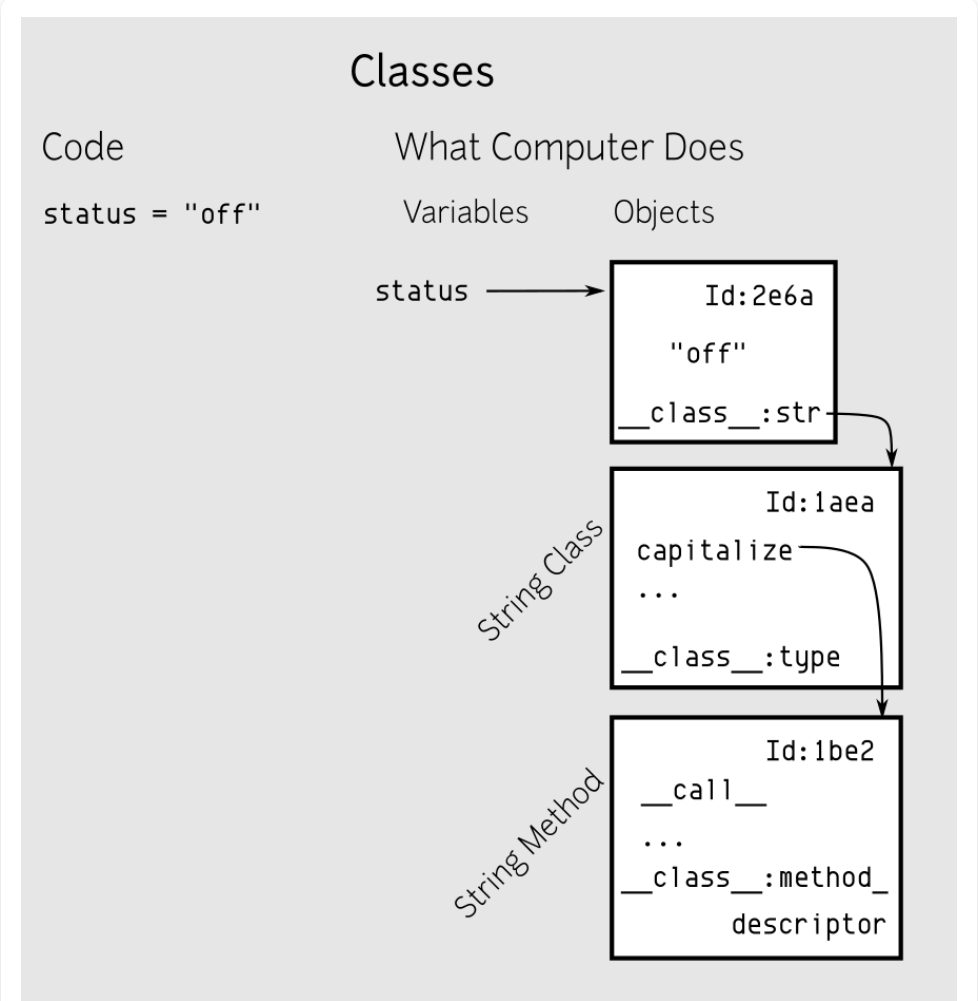
You can create an instance of the `str` class, `b`, by using Python's literal string syntax:

```
>>> b = "I'm a string"
>>> b
"I'm a string"
```

There are many terms that you will see thrown about to talk about `b`. You may hear, "`b` is a string", "`b` is an object", "`b` is an instance of a string". The latter is perhaps the most specific. But, `b` is not a string class.



This illustrates a string object. All objects have types, which are really just the class of the object. In this case the class is `str`.



This illustrates an updated version of a string object. Type has been changed to `__class__`, because when you inspect the object, the `__class__` attribute points to the class of the object. Note that this class has various methods, this image only shows `capitalize`, but there are many more. Methods are also objects, as shown in the image.

NOTE

The `str` class can also be used to create strings, but is normally used for casting. Strings are built into the language, so it is overkill to pass in a string literal into the `str` class.

You wouldn't say:

```
>>> c = str("I'm a string")
```

Because Python automatically creates a string when you put quotes around characters. The term *literal* means that this is a special syntax built into Python to create strings.

On the flipside, if you have a number that you want to convert to a string, you could call `str`:

```
>>> num = 42
>>> answer = str(num)
>>> answer
'42'
```

It is said that Python comes with "batteries included"—it has libraries and classes predefined for your use. These classes tend to be generic. You can define your own classes that deal specifically with your problem domain. You can create customized objects that contain state, and as well as logic to change that state.

Planning for a class

First of all, classes are not always needed in Python. You should give some thought to whether you need to define a class or whether a function (or group of functions) would be sufficient. Classes might be useful to provide a programmatic representation of something that is a physical object, or a

conceptual object. Something that is a description such as speed, temperature, average, or color are not good candidates for classes.

Once you have decided that you want to model something with a class, ask yourself the following questions:

- Does it have a name?
- What are the properties that is has?
- Are these properties constant between instances of the class? ie:
    - Which of these properties are common to the class?
    - Which of these properties are unique to each member?
- What actions does it do?

Here’s a concrete example. Assume that you work for a ski resort and want to model how people use the chairlifts. One way to do so would be to create a class that defines a chair on a chairlift. A chairlift, if you are not familiar with it, is a contraption that has many chairs. These chairs are like a bench that multiple skiers may sit on. Skiers queue up at the bottom of a hill to board the chairs, then unboard once the chair has lifted them to the top of the hill.

Obviously, you will need to abstract your model to some degree. You are not going to model every low-level property of a chair. For example, you don't care whether a chair is made of steel or aluminum or wood for calculating usage. That might matter for other people.

Does this thing you want to model have a name? Yes, chair. A few properties of the chair include chair number, capacity, whether it has a safety bar, and if it has padded seats. To dive in a little deeper, capacity can be broken down into maximum capacity and current occupants. Maximum

capacity should stay constant, whereas occupants can change at a point in time.

A chair is involved with a few actions. Some actions include adding people at the bottom of the hill and removing people when they reach the top of the hill. Another action might be the position of the safety bar. You are going handle loading and unloading but ignore bar position for your model.

Defining a class

Here is a Python class to represent a chair on a chairlift. Below is a simple class definition. The comments are numbered for discussion following the code:

```
>>> class Chair:                        # 1
...     ''' A Chair on a chairlift ''' # 2
...     max_occupants = 4              # 3
...
...     def __init__(self, id):        # 4
...         self.id = id               # 5
...         self.count = 0
...
...     def load(self, number):        # 6
...         self.count += number
...
...     def unload(self, number):      # 7
...         self.count -= number
```

The `class` statement in Python defines a class. You need to give it a name (1), followed by a colon. Remember that in Python, you indent following a colon (unless it is a slice). Note, that you indented consistently below the class definition. Also note that the name of the class was capitalized.

NOTE

Class names are actually *camel cased*. Because `Chair` is a single word, you might not have noticed. Unlike functions where words are joined together with underscores, in camel casing, you capitalize the first letter of each word and then shove them together. Normally class names are nouns. In Python, they cannot start with numbers. The following are examples of class names, both good and bad:

```
- Kitten       # good
- jaguar       # bad - starts with lowercase
- SnowLeopard  # good - camel case
- White_Tiger  # bad - has underscores
- 9Lives       # bad - starts with a number
```

See PEP 8 for further insight into class naming.

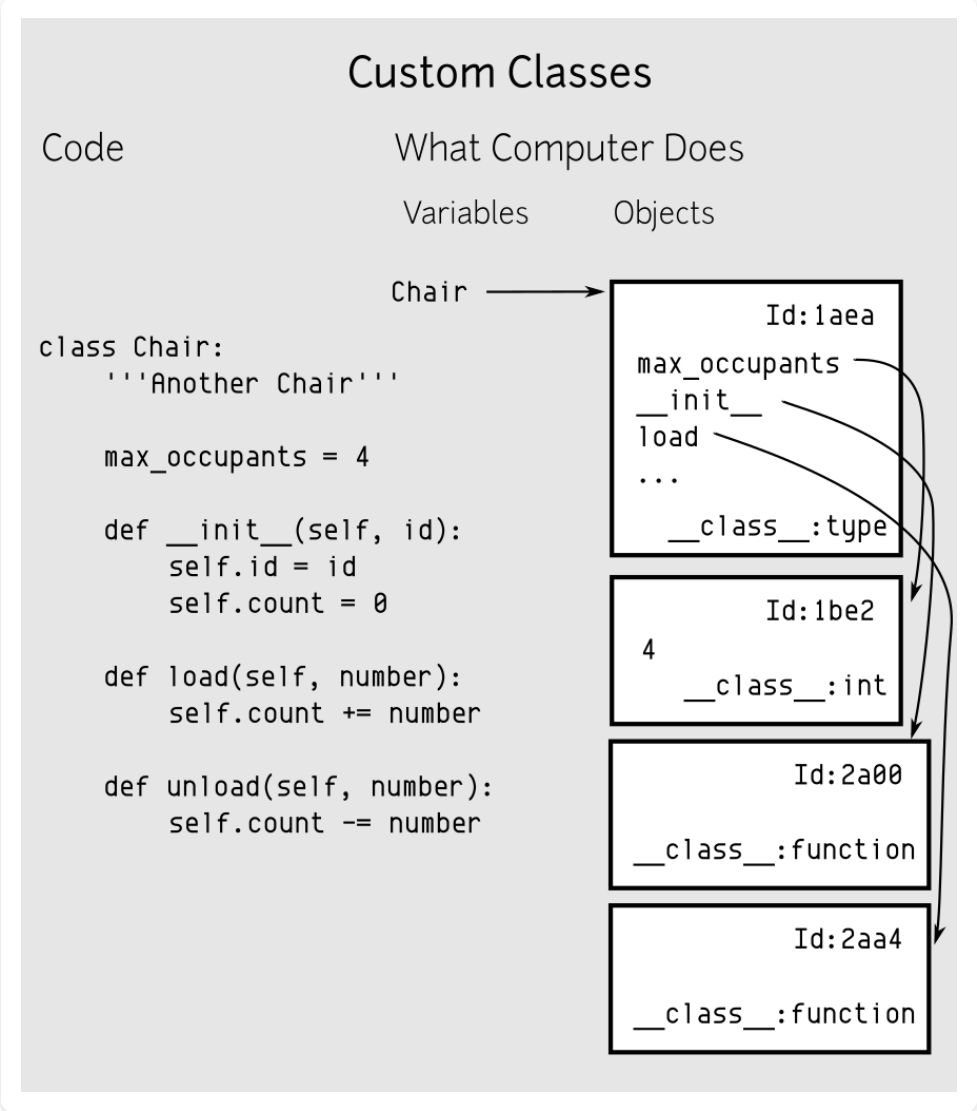
Note that many of the built-in types do not follow this rule: `str`, `int`, `float`, etc.

Immediately following the declaration of the class, you can insert a docstring (2). This is just a string. Note that if it is a triple-quoted string, it may span multiple lines. Docstrings are optional, but they are useful to readers of your code, and also appear when you use the `help` function to inspect your code in the REPL. Use them judiciously and it will pay great dividends.

Inside the indented body of the class, you can create *class attributes* (3). A class attribute is used to hold state that is shared among all instances of a class. In this example, any chair that you create will have a maximum of four occupants. (Skiers call these types of chairlifts quads). There are

advantages to creating class attributes. Because the class is setting this number, you don't have to repeat yourself and set the value each time you create a chair. On the flipside, you have hardcoded your chair to only support four seats on it. Later, you will see how to override the class attribute.

Next you see a `def` statement (4). It looks like you are defining a function inside of the class body. And you are, except when you define a function directly inside of a class body, you call it a *method*. Because this method has a special name, `__init__`, you call it a *constructor*. It has two parameters, `self` and `id`. Most methods have `self` as the first parameter. You can interpret `self` as being the instance of the class.



This illustrates what happens when you define a class. Python will create a new type for you. Any class attributes or methods will be stored as attributes of the new class. Instance attributes (`id` and `count`) are not found in the class as they will be defined on the instances.

A constructor is called when you create an instance of a class. If you consider a class to be a factory that provides a template or blueprint for instances, then the constructor is what initializes the state for the instances. The constructor takes an instance as input (the `self` parameter), and updates it inside the method. Python takes care of passing around the instance for us. This can be confusing and will be discussed more later.

Inside the body of the constructor (5) (it is indented because it follows a colon), you attach two attributes that will be unique to the instance, `id` and `count`. On most chairlifts, each chair on the chairlift has a unique number painted on the chair. The `id` represents this number. Also, a chair may hold a number of skiers—you store this in the `count` attribute and initialize it to zero. Note, that the constructor does not return anything, but rather it updates values that are unique to the instance.

> **NOTE**
>
> Remember that `id` is a built-in function in Python, but you can also use that as an attribute name in a class. You will still have access to the `id` function. Every time you want to access the `id` attribute you will do a lookup on the instance. If the instance was named `chair`, you would get the `id` by calling `chair.id`. So, this is not shadowing the built-in function and is ok.

You can tell that the constructor logic is finished when you see that the indentation level has stepped back out. You see another method defined (6), `load`. This method represents an action that an instance of the class can perform. In this case, a chair may load passengers onto it, and this method tells the instance what to do when that happens. Again, `self` (the instance), is the first parameter to the method. The second parameter, `number`, is the number of people that get on the chair. Remember that a chair on a chairlift can usually hold more than one person, and in this case, you said that your chair may hold up to four people. When skiers board the chair you would want to call the `load` method on the chair, and inside of the body of that method, update the `count` attribute on the instance.

Likewise, there is a corresponding method, `unload` (7), that should be called when skiers dismount at the top of the hill.

NOTE

Don't be intimidated by methods. You have already seen many methods, such as the `.capitalize` method defined on a string. Methods are functions that are attached to a class. Instead of calling the method by itself, you need to call it on an instance of the class:

```
>>> 'matt'.capitalize()
'Matt'
```

In summary, creating a class is easy. You define the attributes you want. The attributes that are constant are put inside the class definition. The attributes that are unique to an instance are put in the constructor. You can also define methods that contain actions that modify the instance of the class. After the class is defined, Python will create a variable with the name of the class that points to the class:

```
>>> Chair
<class '__main__.Chair'>
```

You can inspect the class with the `dir` function. Note that the class attributes are defined on the class, and you can access them directly on the class:

```
>>> dir(Chair)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'load', 'max_occupants', 'unload']
```

```
>>> Chair.max_occupants
4
```

The figure in this section showed how the attributes and methods of the class are stored. We can also inspect them from the REPL. Because everything is an object in Python, they will all have a `__class__` attribute:

```
>>> Chair.__class__
<class 'type'>
>>> Chair.max_occupants.__class__
<class 'int'>
>>> Chair.__init__.__class__
<class 'function'>
>>> Chair.load.__class__
<class 'function'>
>>> Chair.unload.__class__
<class 'function'>
```

The methods are also defined on the class, but the instance attributes are not. Because instance attributes are unique to the instance, they will be stored on the instance.

If you have docstrings defined on your class or its methods, you can inspect them with `help`:

```
>>> help(Chair)
  Help on class Chair in module __main__:

  class Chair(builtins.object)
   |  A Chair on a chairlift
   |
   |  Methods defined here:
   |
   |  __init__(self, id)
   |
   |  load(self, number)
   |
   |  unload(self, number)
   |
   |  ----------------------------------------------------
   |  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:
 |
 |  max_occupants = 4
```

Creating an instance of a class

Now that you have defined a class to represent a chair, you can create *instances* of chairs. One way to think of the `Chair` class is to compare it to a factory. The factory takes in bare objects and churns out chair objects.

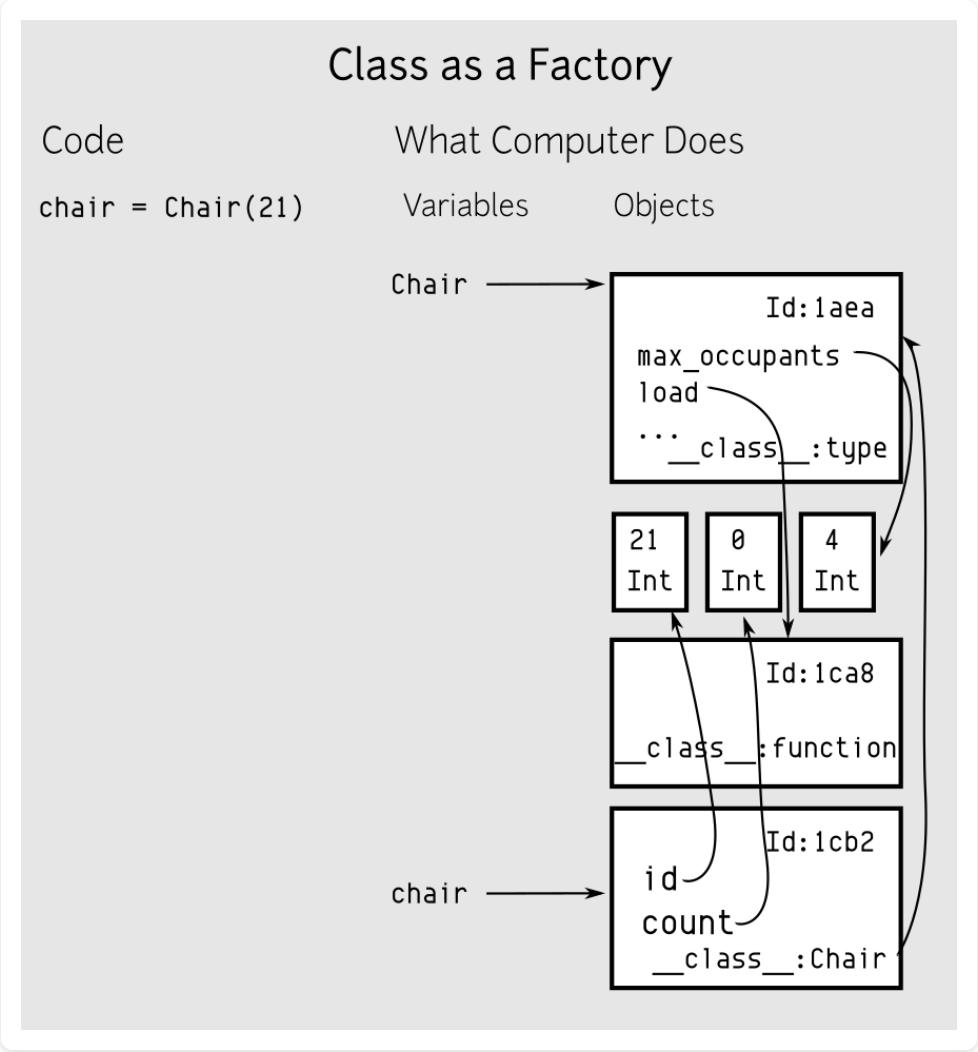
To be more specific, this is pretty much what the constructor method, `__init__` does. You'll notice that `self` is the first parameter, this is the bare object. Python assigns the `__class__` attribute (pointing the `Chair` class) to the object before passing it into the constructor.

Another analogy that might help understand classes is to consider how cartoons depict the delivery of babies. The unborn babies live in the clouds somewhere. At some point in time, a stork comes along, picks up a baby from the cloud and delivers it to a crib in the parent's house. When you call the constructor, Python goes out and picks a baby up from the cloud (gets an object for you). It delivers it to the house, making it a member of the family (it sets the `__class__` to `Chair` or whatever your class is). Once the baby is inside the house, you can mold her, cut her hair, etc. Remember that objects store state and mutate the state.

Below is code to create chair number 21 in Python. When you invoke the class (i.e. put parentheses after the class name), you tell Python that you want to call the constructor. Unlike some languages, you don't need to say `new` or add the variable type, you just add parentheses with the constructor parameters following the name of the class:

```
>>> chair = Chair(21)
```

Again, to be specific with terminology, the `chair` variable points to an object or instance. It does not point to a class. The object has a class, `Chair`. The instance has a few attributes, including `count` and `id`.



This illustrates constructing an object. When the constructor is called, the proverbial Python "stork" drops a baby object as `self` into the constructor. The baby object has the `__class__` attribute set, but the constructor is free to alter the instance, by adding attributes. This object will turn into `chair`.

You can access an instance attribute from the instance, `chair`:

```
>>> chair.count
0

>>> chair.id
21
```

Python has a hierarchy for looking up attributes. First, Python will look for the attribute on the instance. If that fails, Python will try to find the attribute on the class. Because instances know about their class, Python will look there next. If that fails Python will raise an `AttributeError`, an apt error for a missing attribute. The `max_occupants` attribute is actually stored on the class, but you can access it from the instance:

```
>>> chair.max_occupants
4
```

Under the covers Python is doing this for you:

```
>>> chair.__class__.max_occupants
4
```

Attribute lookup is different that variable lookup. Recall that Python looks for variables first in the local scope, then global scope, then the built-in scope, and finally raises a `NameError` if it was unsuccessful. Attribute lookup occurs first on the instance, then the class, then the parent classes, and will raise a `AttributeError` if the attribute was not found.

Calling a method on a class

When you have an instance of a class, you can call methods on it. Methods —like functions—are invoked by adding parentheses with any arguments inside of them. Here you will call the `.load` method to add 3 skiers to the chair:

```
>>> chair.load(3)
```

Here's a review of the syntax of method invocation. First, you list the instance, `chair`, which is followed by a period. A period means search for an attribute in Python (unless it is following a number literal). When you see a period following an instance, remember that Python is going to search for what follows the period.

First, Python will look on the instance for `load`. This attribute was not found on the instance, recall that only `count` and `id` were set on the instance inside of the constructor. But the instance has a link to its class. So if the search fails on the instance, Python will look for those attributes on the class. The `.load` method is defined on the `Chair` class, so Python returns that. The parentheses mean invoke or call the method. You are passing `3` in as a parameter to the method.

Recall that the declaration of `load` looked like this:

```
...     def load(self, number):        # 6
...         self.count += number
```

In the declaration, there were two parameters, `self` and `number`. But in the invocation, you only passed a single parameter, `3`. Why the mismatch? The `self` parameter represents the instance, which is `chair` in your case. Python will call the `.load` method by inserting `chair` as the `self` parameter and `3` as the `number` parameter. In effect, Python handles passing around the `self` parameter for you automatically.

NOTE

When you call:

```
chair.load(3)
```

What happens under the covers is similar to this:

```
Chair.load(chair, 3)
```

You can try this out to validate that it works, but you wouldn't do this in real life because it is harder to read, and also requires more typing.

Examining an instance

If you have an instance and want to know what its attributes are, you have a few options. You can look up the documentation (if it exists). You can read the code where the class was defined. Or you can use `dir` to examine it for you:

```
>>> dir(chair)
['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'count', 'id', 'load',
'max_occupants', 'unload']
```

Recall that the `dir` function lists the attributes of an object. If you look at the documentation for `dir`, you see that the definition previously provided for `dir` is not quite correct. The documentation reads:

> *return an alphabetized list of names comprising (some of) the attributes of the given object, and of attributes reachable from it.*
> —`help(dir)` *(emphasis added)*

This function shows the attributes that are reachable from an object. The actual state of the instance is stored in the `__dict__` attribute, a dictionary mapping attribute names to values:

```
>>> chair.__dict__
{'count': 3, 'id': 21}
```

So, the instance really only stores `count` and `id`, the other attributes are available through the class. Where is the class stored? In the `__class__` attribute:

```
>>> chair.__class__
<class '__main__.Chair'>
```

It is important that an instance know what its class is because the class stores the methods and class attributes.

Private and protected

Some languages have the notion of private attributes or methods. These are methods that are meant to be implementation details and end users can't call them. In fact, the language may prevent access to them.

Python does not make an effort to prevent users from doing much of anything. Rather, it takes the attitude that you are an adult and you should take responsibility for your actions. If you want to access something, you can do it. But you should be willing to accept the consequences.

Python programmers realize that it can be convenient to store state and methods that are implementation details. To signify to end users that they should not access these members, you prefix their name with an underscore. Here is a class that has a helper method, `._check`, that is not meant to be called publicly:

```
>>> class CorrectChair:
...     ''' A Chair on a chairlift '''
...     max_occupants = 4
...
...     def __init__(self, id):
...         self.id = id
...         self.count = 0
...
...     def load(self, number):
...         new_val = self._check(self.count + number)
...         self.count = new_val
...
...     def unload(self, number):
...         new_val = self._check(self.count - number)
...         self.count = new_val
...
...     def _check(self, number):
...         if number < 0 or number > self.max_occupants:
...             raise ValueError('Invalid count:{}'.format(
...                              number))
...         return number
```

The `._check` method is considered private, only the instance should access it inside the class. In the class, the `.load` and `.unload` methods call the private method. If wanted, you could call it from outside the class. But you shouldn't, as anything with an underscore should be considered an implementation detail that might not exist in future versions of the class.

A simple program modeling flow

Let's use the class above to model flow of skiers up a hill. You will make some basic assumptions, such as every chair has an equal probability of 0 to `max_occupants` riding on it. It will turn on the chairlift, load it, and run forever. Four times a second it prints out the current statistics:

```
import random
import time

class CorrectChair:
    ''' A Chair on a chairlift '''
    max_occupants = 4

    def __init__(self, id):
        self.id = id
        self.count = 0

    def load(self, number):
        new_val = self._check(self.count + number)
        self.count = new_val

    def unload(self, number):
        new_val = self._check(self.count - number)
        self.count = new_val

    def _check(self, number):
        if number < 0 or number > self.max_occupants:
            raise ValueError('Invalid count:{}'.format(
                             number))
        return number

NUM_CHAIRS = 100

chairs = []
for num in range(1, NUM_CHAIRS + 1):
```

```
    chairs.append(CorrectChair(num))

def avg(chairs):
    total = 0
    for c in chairs:
        total += c.count
    return total/len(chairs)

in_use = []
transported = 0
while True:
    # loading
    loading = chairs.pop(0)
    in_use.append(loading)
    in_use[-1].load(random.randint(0, CorrectChair.max_occupants))

    # unloading
    if len(in_use) > NUM_CHAIRS / 2:
        unloading = in_use.pop(0)
        transported += unloading.count
        unloading.unload(unloading.count)
        chairs.append(unloading)
    print('Loading Chair {} Count:{} Avg:{:.2} Total:{}'.format
          (loading.id, loading.count, avg(in_use), transported))
    time.sleep(.25)
```

This program will run forever printing out how many people are riding on the chairlift. It outputs to the terminal, but the `print` function could be replaced with code that writes a CSV file as well.

By changing two numbers, the global `NUM_CHAIRS` and the class attribute `CorrectChair.max_occupants`, you could change how the model behaves to represent larger or smaller chairlifts. The call to `random.randint` could be replaced with a function that more precisely represents the usage distribution.

Summary

We dove into classes a bit in this chapter. The terminology around classes was discussed. You can say object or instance; they are interchangeable.

Each object has a class. The class is like a factory that determines how its instance objects behave.

An object is created by the constructor method. This method is named `__init__`. You may define other methods for classes as well.

Give some thought to creating a class. What are the attributes? If the attributes remain consistent across the objects, define them on the class. If they are unique to the object, set them in the constructor.

Exercises

1. Imagine you are designing a banking application. What would a customer look like? What attributes would she have? What methods would she have?
2. Imagine you are creating a Super Mario game. You need to define a class to represent Mario. What would it look like? If you aren't familiar with Super Mario, use your own favorite video or board game to model a player.
3. Create a class that could represent a tweet. If you aren't familiar with Twitter, Wikipedia describes it as:

   > *[...] an online news and social networking service where users post and interact with messages, "tweets", restricted to 140 characters.*
   > —*https://en.wikipedia.org/wiki/Twitter*

4. Create a class that could represent a household appliance (toaster, washer, refrigerator, etc.).

Subclassing a Class

BESIDES GROUPING STATE AND ACTION IN A COHERENT PLACE, CLASSES ALSO enable re-use. If you already have a class but want one that behaves slightly differently, one way to re-use the existing class is to *subclass* it. The class that you subclass from is called the *superclass*. (Another common term for superclass is *parent class*).

Suppose that you wanted to create a chair that can hold six people. To create a class representing a six-person chair, `Chair6`, which is a more specialized version of a `Chair`, you can create a subclass. Subclasses allow you to *inherit* methods of parent classes, and *override* methods you want to change.

Here is the class `Chair6`, which is a subclass of `CorrectChair`:

```
>>> class Chair6(CorrectChair):
...     max_occupants = 6
```

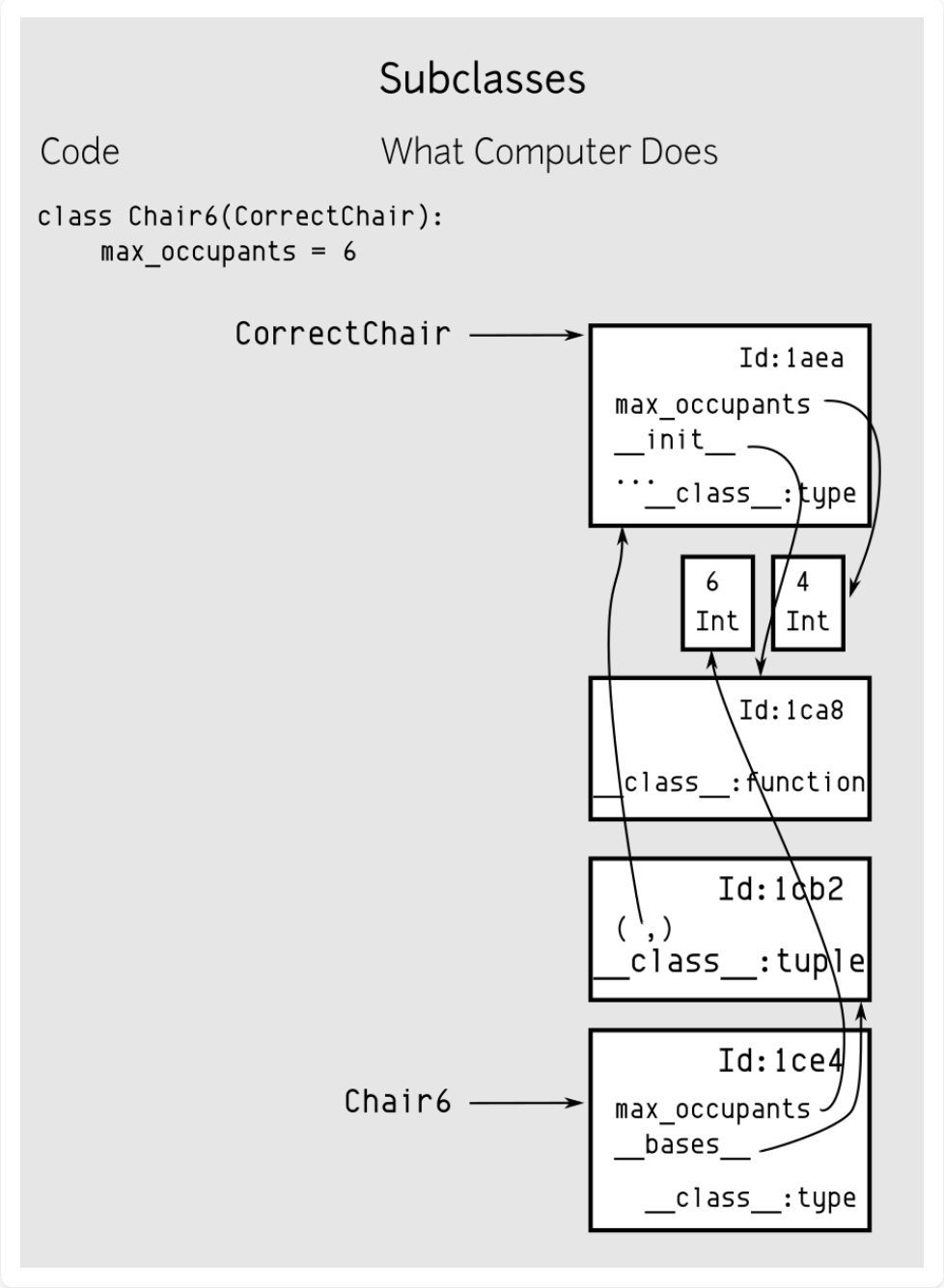


Illustration of the `__bases__` attribute in a subclass. This connection between a subclass and its parent classes allows you to look up attributes in a well-defined manner. If the instance of a subclass has an attribute defined, it uses that attribute. If not, after searching the instance, the class (`__class__`) of the instance is searched. Failing that, parent classes (`__bases__`) of the class are searched.

Note that you put the parent class, `CorrectChair`, in parentheses following the class name. Notice that `Chair6` doesn't define a constructor

inside of the body, yet you can still create instances of the class:

```
>>> sixer = Chair6(76)
```

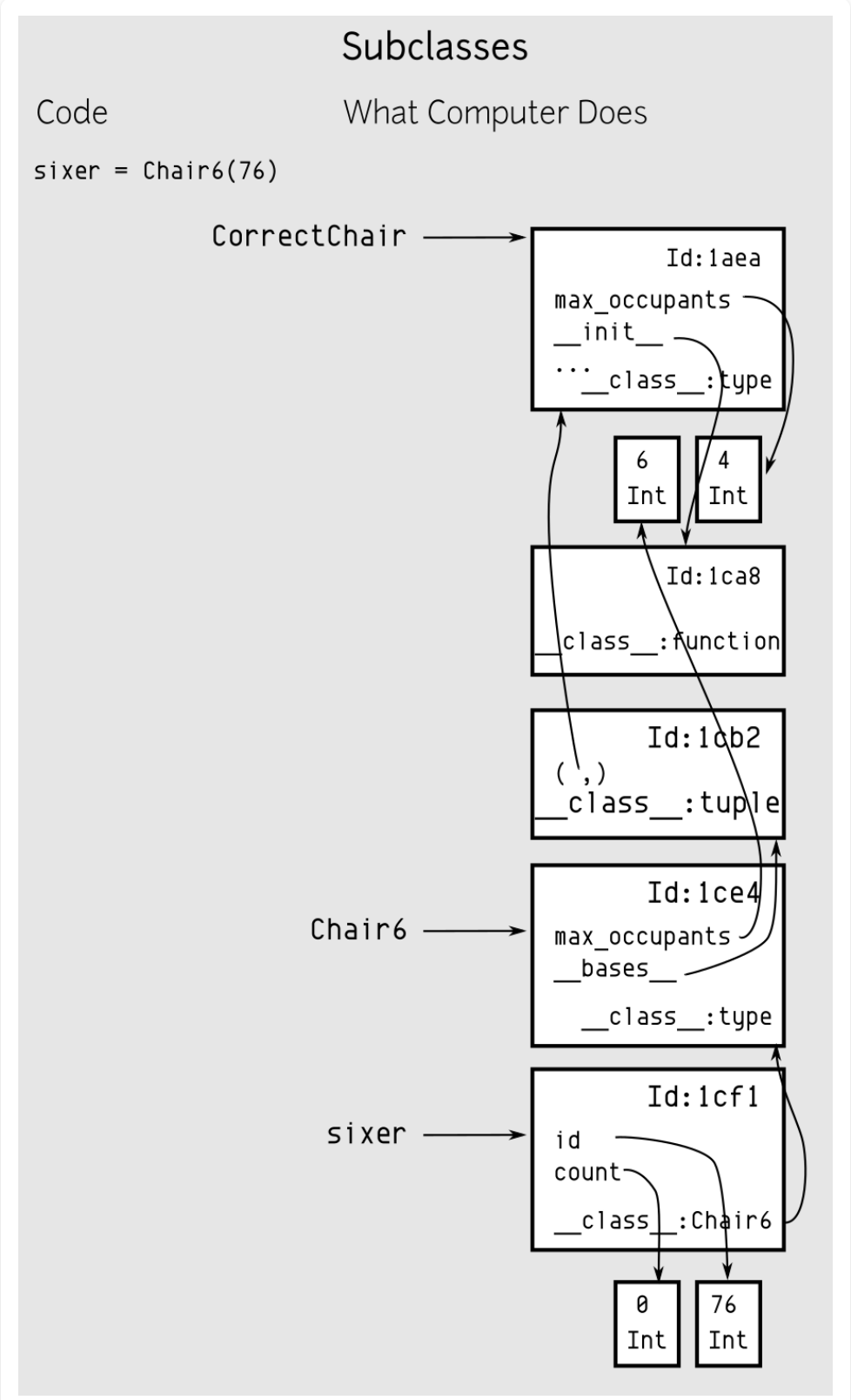


Illustration depicting the instantiation of a subclass. Note that the instance points to its class and that the class points to any parent classes using the __bases__ attribute.

How does Python create an object, when the class doesn't define the constructor? Here is what happens: when Python looks for the `.__init__` method, it will search for it on `Chair6` first. Since the `Chair6` class only has a `max_occupants` attribute, Python will not find the `.__init__` method there. But, because `Chair6` is a subclass of `CorrectChair`, it has a `__bases__` attribute that lists the base classes in a tuple:

```
>>> Chair6.__bases__
(__main__.CorrectChair,)
```

Python will then search the base classes for the constructor. It will find the constructor in `CorrectChair` and use it to create the new class.

The same lookup happens when you call `.load` on an instance. The instance doesn't have an attribute matching the method name, so Python looks at the class of the instance. `Chair6` does not have the `.load` method either, so Python looks for the attribute on the base class, `CorrectChair`. Here the `.load` method is called with a count that is too large, leading to a `ValueError`:

```
>>> sixer.load(7)
Traceback (most recent call last):
  File "/tmp/chair.py", line 30, in <module>
    sixer.load(7)
  File "/tmp/chair.py", line 13, in load
    new_val = self._check(self.count + number)
  File "/tmp/chair.py", line 23, in _check
    number))
ValueError: Invalid count:7
```

Python finds the method on the base class, but the call to the `._check` method raises a `ValueError`.

Counting stalls

A semi-common occurrence is that a skier fails to mount the chair correctly. In that case, a chairlift operator will slow down or stop the chairlift to assist

the skier. You can use Python to create a new class that will be able to account for the number of times a stall happens.

Assume that you want to call a function every time `.load` is called, and this function will return a boolean indicating whether a stall occurred. The function takes as parameters the number of skiers and the chair object.

Below is a class that accepts an `is_stalled` function in the constructor. It will call this function every time `.load` is invoked to determine if the chairlift has stalled:

```
>>> class StallChair(CorrectChair):
...     def __init__(self, id, is_stalled):
...         super().__init__(id)
...         self.is_stalled = is_stalled
...         self.stalls = 0
...
...     def load(self, number):
...         if self.is_stalled(number, self):
...             self.stalls += 1
...         super().load(number)
```

To create an instance of this class, you must provide an `is_stalled` function. Here is a simple function that stalls ten percent of the time:

```
>>> import random
>>> def ten_percent(number, chair):
...     """Return True 10% of time"""
...     return random.random() < .1
```

Now you can create an instance using the `ten_percent` function as your `is_stalled` parameter:

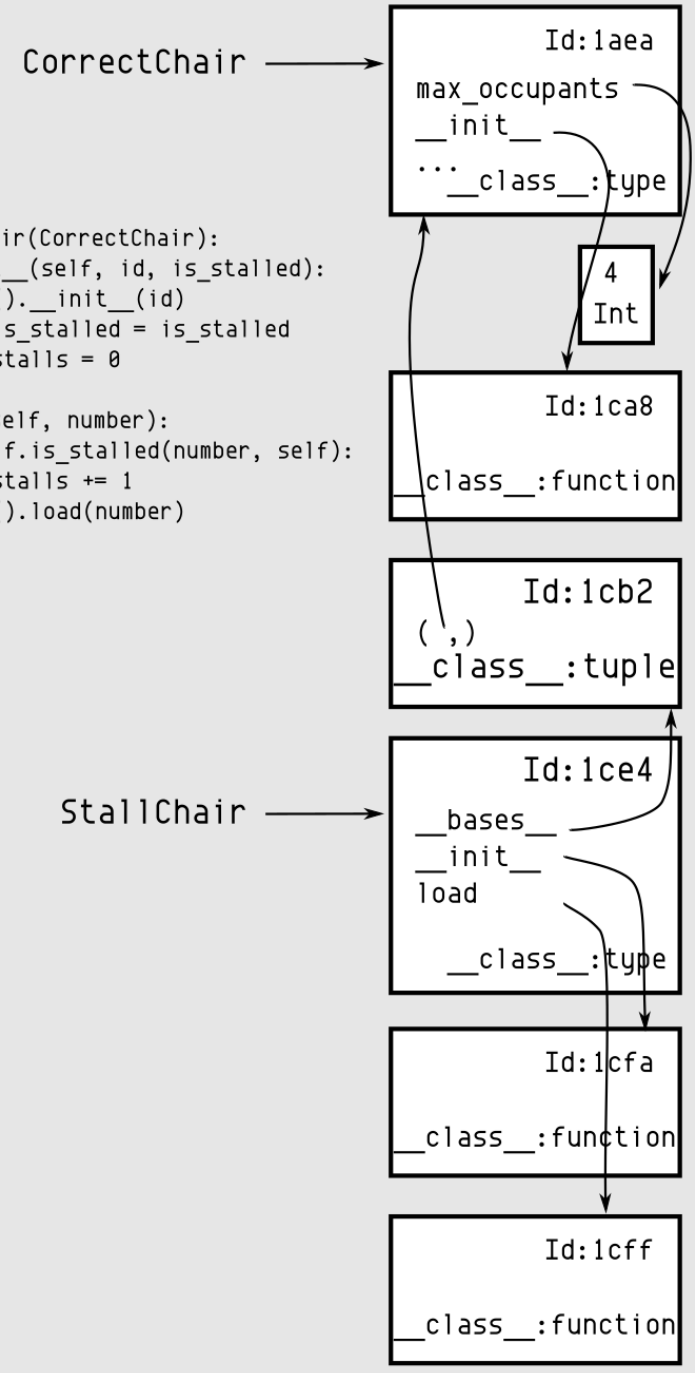
```
>>> stall42 = StallChair(42, ten_percent)
```

Subclasses

Code

```
class StallChair(CorrectChair):
    def __init__(self, id, is_stalled):
        super().__init__(id)
        self.is_stalled = is_stalled
        self.stalls = 0

    def load(self, number):
        if self.is_stalled(number, self):
        self.stalls += 1
        super().load(number)
```

What Computer Does



A figure showing the code for creating a subclass with custom methods. Note that you use `super()` to call the method on the parent class. This illustrates what objects are created when you create a class that is a subclass.

`super`

Remember that `StallChair` defines its own `.__init__` method, which is called when the instance is created. Note that the first line in the constructor is:

```
super().__init__(id)
```

When `super` is called inside of a method, it gives you access to the correct parent class. This line in the constructor allows you to invoke the `CorrectChair` constructor. Rather than repeat the logic of setting the `id` and `count` attribute, you can reuse the logic from the parent class. Because `StallChair` has additional attributes it wants to set on the instance, you can do that following the call to the parent constructor.

Note that the `.load` method also has a call to `super`:

```
def load(self, number):
    if self.is_stalled(number, self):
        self.stalls += 1
    super().load(number)
```

In the `.load` method, you call your `is_stalled` function to determine whether the chair was stalled. Then you dispatch back to the original `.load` functionality found in `CorrectChair` by using `super`.

Having general code appear in one place (the base class) eliminates bugs and repeating code.

NOTE

There are two cases where `super` really comes in handy. One is for resolving method resolution order (*MRO*) in classes that have multiple parents. `super` will guarantee that this order is consistent. The other is when you change the base class, `super` is intelligent about determining who the new base is. This aids in code maintainability.

Summary

This chapter discussed subclasses, which are new specialized classes that reuse code from their base class (also referred to as superclass or parent class). For any method that you don't implement in a subclass, Python will reuse the parent class' functionality. You can choose to implement a method to override completely, or you can call out to `super`. When you call `super`, it gives you access to the parent class so you can reuse any functionality found there.

Exercises

1. Create a class to represent a cat. What can a cat do? What are the properties of a cat? Create a subclass of a cat for a tiger. How does it behave differently?
2. In the previous chapter, you created a class to represent Mario from the Super Mario Brothers video game. In later editions of the game, there were different characters that you could play. They all had the same basic functionality [15] but differed in skill. Create a base class for the character, then implement four subclasses. One for Mario, Luigi, Toad, and Princess Toadstool.

Name	Mario	Luigi	Toad	Princess Toadstool
Speed	4	3	5	2
Jump	4	5	2	3
Power	4	3	5	2
Special Skill				Float jump

15 - https://www.mariowiki.com/Super_Mario_Bros._2#Playable_characters

Exceptions

A computer may be told to perform an action that it cannot do. Reading files that do not exist or dividing by zero are two examples. Python allows you to deal with such *exceptions* when they occur. In these cases, Python *throws an exception* or *raises an exception*.

Normally when an exception occurs, Python will halt and print out a *stack trace* explaining where the problem occurred. This stack trace is useful in that it tells you the line and file of the error:

```
>>> 3/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

The above states that in line 1 of the file `<stdin>` (this is the name of the interpreter "file") there was a divide by zero error. When you execute a program with an exception, the stack trace will indicate the file name and line number of where the problem occurred. This example from the interpreter isn't particularly helpful as there is only a single line of code. But in bigger programs you could have deeply nested stack traces due to functions calling other functions and methods:

If you had the following in a file and a tried to run it, you would see a traceback:

```
def err():
    1/0

def start():
     return middle()
```

```
def middle():
    return more()

def more():
    err()

start()
```

Here is the stacktrace:

```
Traceback (most recent call last):
  File "/tmp/err.py", line 13, in <module>
    start()
  File "/tmp/err.py", line 5, in start
    return middle()
  File "/tmp/err.py", line 8, in middle
    return more()
  File "/tmp/err.py", line 11, in more
    err()
  File "/tmp/err.py", line 2, in err
    1/0
ZeroDivisionError: division by zero
```

Tracebacks are easiest to read when you start from the bottom, find the error, and see where it occurred. As you move up the traceback, you are in effect looking up the call chain. This can help you pinpoint what is going on in your program.

Look before you leap

Suppose you have a program that performs division. Depending on how it is coded, it may be possible that it tries to divide by zero at some point. There are two styles for dealing with exceptions that programmers commonly use. The first is *look before you leap* (LBYL). The idea is to check for exceptional cases before performing an action. In this case, the program would examine the denominator value and determine if it is zero or not. If it is not zero, the program could perform the division, otherwise, it could skip it.

In Python, look before you leap can be implemented with `if` statements:

```
>>> numerator = 10
>>> divisor = 0
>>> if divisor != 0:
...     result = numerator / divisor
... else:
...     result = None
```

NOTE

Look before you leap is not always a guarantee of success. If you check that a file exists before you open it (looking before leaping), that does not mean that the file will still be around later. In multi-threaded environments, this is known as a *race condition*.

NOTE

`None` is used to represent the undefined state. This is a common idiom throughout Pythondom. Be careful though, try not to invoke methods on a variable that is assigned to `None`, as that will raise an exception.

Easier to ask for forgiveness

Another option for dealing with exceptions is known as *easier to ask for forgiveness than permission* (EAFP). The idea here is to always perform the operation inside of a *try* block. If the operation fails, the exception will be *caught* by the *exception* block.

The `try...except` construct provides a mechanism for Python to catch exceptional cases:

```
>>> numerator = 10
>>> divisor = 0
>>> try:
...     result = numerator / divisor
```

```
... except ZeroDivisionError as e:
...     result = None
```

Notice that the `try` construct creates a block following the `try` statement (because there is a colon and indentation). Inside of the try block are the statements that might throw exceptions. If the statements actually throw an exception, Python looks for an except block that *catches* that exception (or a parent class of it).

In the code above, the except block states that it will catch an exception that is an instance (or subclass) of the `ZeroDivisionError` class. When the stated exception is thrown in the `try` block, the `except` block is executed and `result` is set to `None`.

Note that line:

```
except ZeroDivisionError as e:
```

has `as e:` on the end. This part is optional. If it is included then `e` (or whatever you choose as a valid variable name) will point to an instance of a `ZeroDivisionError` exception. You can inspect the exception as they often have more details. The `e` variable points to the *active exception*. If you leave "`as e`" off the end of the `except` statement, you will still have an active exception, but you won't be able to access the instance of it.

TIP

Try to limit the scope of the `try` block. Instead of including all the code in a function inside a `try` block, include only the line that will possibly throw the error.

Because the look before you leap style of handling is not guaranteed to prevent errors, in general, most Python developers favor the easier to ask

for forgiveness style of exception handling. Here are some rules of thumb for exception handling:

- Gracefully handle errors you know how to handle and can reasonably expect.
- Do not silence exceptions that you cannot handle or do not reasonably expect.
- Use a global exception handler to gracefully handle unexpected errors.

Tip

If you were making a server type application that needs to run without stopping, here is one way to do it. The functions `process_input` and `log_error` don't exist but serve as placeholders:

```
while 1:
    try:
        result = process_input()
    except Exception as e:
        log_error(e)
```

Multiple exception cases

If there are multiple exceptions that your code needs to be aware of, you can chain a list of `except` statements together:

```
try:
    some_function()
except ZeroDivisionError as e:
    # handle specific
except Exception as e:
    # handle others
```

In this case, when `some_function` throws an exception, the interpreter checks first if it is a `ZeroDivisionError` (or a subclass of it). If that is not

the case, it checks if the exception is a subclass of `Exception`. Once an `except` block is entered, Python no longer checks the subsequent blocks.

If an exception is not handled by the chain, code somewhere up the stack must deal with the exception. If the exception is unhandled, Python will stop running and will print the stack trace.

An example of dealing with multiple exceptions is found in the standard library. The `argparse` module from the standard library provides an easy way to parse command line options. It allows you to specify a type for certain options, such as integers or files (the options all come in as strings). In the `._get_value` method there are examples of multiple `except` clauses. Based on the type of the exception that occurs a different error message is provided:

```
def _get_value(self, action, arg_string):

    type_func = self._registry_get('type', action.type, action.type)
    if not callable(type_func):
        msg = _('%r is not callable')
        raise ArgumentError(action, msg % type_func)

    # convert the value to the appropriate type
    try:
        result = type_func(arg_string)

    # ArgumentTypeErrors indicate errors
    except ArgumentTypeError:
        name = getattr(action.type, '__name__', repr(action.type))
        msg = str(_sys.exc_info()[1])
        raise ArgumentError(action, msg)

    # TypeErrors or ValueErrors also indicate errors
    except (TypeError, ValueError):
        name = getattr(action.type, '__name__', repr(action.type))
        args = {'type': name, 'value': arg_string}
        msg = _('invalid %(type)s value: %(value)r')
        raise ArgumentError(action, msg % args)

    # return the converted value
    return result
```

NOTE

This code example also shows that a single `except` statement can catch more than one exception type if you provide a tuple of exception classes:

```
except (TypeError, ValueError):
```

NOTE

This example also shows an older style of string formatting using the `%` operator. The lines:

```
msg = _('invalid %(type)s value: %(value)r')
raise ArgumentError(action, msg % args)
```

would be written in a more modern style as:

```
msg = _('invalid {type!s} value: {value!r}')
raise ArgumentError(action, msg.format(**args))
```

`finally` clause

Another clause for error handling is the `finally` clause. This statement is used to place code that will always execute, regardless of whether an exception happens or not. If the `try` block succeeds, then the `finally` block will be executed.

The `finally` always executes. If the exception is handled, the `finally` block will execute after the handling. If the exception is not handled, the `finally` block will execute and then the exception will be re-raised:

```
try:
    some_function()
except Exception as e:
    # handle errors
```

```
finally:
    # cleanup
```

Usually, the purpose of the `finally` clause is to cleanup external resources, such as files, network connections, or databases. These resources should be freed regardless of whether an operation was successful or not.

An example from the `timeit` module found in the standard library might aid in seeing the utility of the `finally` statement. The `timeit` module allows developers to run a benchmark on their code. One of the things that it does while running the benchmark is tell the Python garbage collector to disable itself during the run. But, you want to ensure that garbage collection is working after the benchmark is done regardless of whether the run worked or errored out.

Here is the method, `timeit` that dispatches to run the benchmark. It checks if the garbage collector was enabled, then turns off the garbage collector, runs the timing code, and finally, re-enables garbage collection if it was previously enabled:

```
def timeit(self, number=default_number):
    """Time 'number' executions of the main statement.

    To be precise, this executes the setup statement once, and
    then returns the time it takes to execute the main statement
    a number of times, as a float measured in seconds.  The
    argument is the number of times through the loop, defaulting
    to one million.  The main statement, the setup statement and
    the timer function to be used are passed to the constructor.
    """
    it = itertools.repeat(None, number)
    gcold = gc.isenabled()
    gc.disable()
    try:
        timing = self.inner(it, self.timer)
    finally:
        if gcold:
            gc.enable()
    return timing
```

It is possible that the call to `self.inner` will throw an exception, but because the standard library uses `finally`, garbage collection will always be turned back on regardless (if the `gcold` boolean is true).

> **NOTE**
>
> This book doesn't address context managers, but to prepare you for your future as a Python expert, here is a hint. The `try`/`finally` combination is a code smell in Python. Seasoned Python programmers will use a *context manager* in these cases. Put this on your list of things to study after you have mastered basic Python.

`else` clause

The optional `else` clause in a `try` statement is executed when no exception is raised. It must follow any `except` statements and executes before the `finally` block. Here is a simple example:

```
>>> try:
...     print('hi')
... except Exception as e:
...     print('Error')
... else:
...     print('Success')
... finally:
...     print('at last')
hi
Success
at last
```

Here is an example from the `heapq` module found in the in the standard library. According to the comments, there is a shortcut if you want the get the smallest values and you ask for more values than the size of the heap. However, if you try to get the size of the heap and get an error, the code calls `pass`. This ignores the error and continues on with the slower way of

getting the small items. If there was no error, you can follow the `else` statement and take the fast path if `n` is greater than the size of the heap:

```
def nsmallest(n, iterable, key=None):
    # Code removed here ....

    # When n>=size, it's faster to use sorted()
    try:
        size = len(iterable)
    except (TypeError, AttributeError) as e:
        pass
    else:
        if n >= size:
            return sorted(iterable, key=key)[:n]

    # Code removed here .... Try slower way
```

Raising exceptions

In addition to catching exceptions, Python also allows you to *raise* exceptions (or throw them). Remember that the Zen of Python wants you to be explicit and refuse the temptation to guess. If invalid input is passed into your function and you know that you will not be able to handle it, you may raise an exception. Exceptions are subclasses of the `BaseException` class, and are raised using the `raise` statement:

```
raise BaseException('Program failed')
```

Normally you will not raise the generic `BaseException` class but will raise subclasses that are predefined, or defined by you.

Another common way of using the `raise` statement is to use it all by itself. Recall that when you are inside of an `except` statement, you have what is called an active exception. If that is the case, you can use a *bare* `raise` statement. A bare `raise` statement allows you to deal with the exception, but re-raise the original exception. If you try to use:

```
except (TypeError, AttributeError) as e:
    log('Hit an exception')
```

```
    raise e
```

You will succeed in raising the original exception, but the stack trace will state that the original exception now occurred in the line with `raise e` rather than where the exception first occurred. You have two options to deal with this. The first is a *bare ``raise``* statement. The second is using exception chaining, described later.

Here is an example from the `configparser` module in the standard library. This module handles reading and creating INI files. An INI file is usually used for configuration and was popular before JSON and YAML were around. The `.read_dict` method will try to read the configuration from a dictionary. If the instance is in strict mode, it will raise an error if you try to add the same section more than once. If you are not in strict mode, the method allows duplicate keys; the last one wins. Here is a portion of the method showing the bare `raise` method:

```
def read_dict(self, dictionary, source='<dict>'):
    elements_added = set()
    for section, keys in dictionary.items():
        section = str(section)
        try:
            self.add_section(section)
        except (DuplicateSectionError, ValueError):
            if self._strict and section in elements_added:
                raise
        elements_added.add(section)
        # code removed from here ....
```

When a duplicate is added in strict mode, the stack trace will show the error in the `.add_section` method, because that is where it occurred.

Wrapping exceptions

Python 3 has introduced another feature similar to a bare `raise` in “PEP 3134 – Exception Chaining and Embedded Tracebacks” When handling an

exception, another exception may occur in the handling code. In this case, it is useful to know about both exceptions.

Here is some basic code. The function, `divide_work`, might have problems dividing by zero. You can catch that error and log that it occurred. Suppose that your `log` function is calling out to a cloud-based service that is down (you'll simulate that by having `log` raise an exception):

```
>>> def log(msg):
...     raise SystemError("Logging not up")

>>> def divide_work(x, y):
...     try:
...         return x/y
...     except ZeroDivisionError as ex:
...         log("System is down")
```

When you call `divide_work` with `5` and `0` as input, Python will show two errors, the `ZeroDivisionError`, and the `SystemError`. It will show `SystemError` last because it happened last:

```
>>> divide_work(5, 0)
Traceback (most recent call last):
  File "begpy.py", line 3, in divide_work
    return x/y
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "begpy.py", line 1, in <module>
    divide_work(5, 0)
  File "begpy.py", line 5, in divide_work
    log("System is down")
  File "begpy.py", line 2, in log
    raise SystemError("Logging not up")
SystemError: Logging not up
```

Suppose your cloud logging service is now working (the `log` function no longer throws an error). If you want to change the type of the

`ZeroDivisionError` in `divide_work` to `ArithmeticError`, you can use a syntax described in PEP 3134. You can use the `raise ... from` syntax:

```
>>> def log(msg):
...     print(msg)

>>> def divide_work(x, y):
...     try:
...         return x/y
...     except ZeroDivisionError as ex:
...         log("System is down")
...         raise ArithmeticError() from ex
```

You will see two exceptions now: the original `ZeroDivisionError` and the `ArithmeticError` which is no longer shadowed by `ZeroDivisionError`:

```
>>> divide_work(3, 0)
Traceback (most recent call last):
  File "begpy.py", line 3, in divide_work
    return x/y
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "begpy.py", line 1, in <module>
    divide_work(3, 0)
  File "begpy.py", line 6, in divide_work
    raise ArithmeticError() from ex
ArithmeticError
```

If you want to suppress the original exception, the `ZeroDivisionError`, you can use the following code. This is described in "PEP 0409 – Suppressing exception context":

```
>>> def divide_work(x, y):
...     try:
...         return x/y
...     except ZeroDivisionError as ex:
...         log("System is down")
...         raise ArithmeticError() from None
```

Now you only see the outermost error, `ArithmeticError`:

```
>>> divide_work(3, 0)
Traceback (most recent call last):
  File "begpy.py", line 1, in <module>
    divide_work(3, 0)
  File "begpy.py", line 6, in divide_work
    raise ArithmeticError() from None
ArithmeticError
```

Defining your own exceptions

Python has many built-in exceptions defined in the `exceptions` module. If your error corresponds well with the existing exceptions, you can re-use them. The following lists the class hierarchy for the built-in exceptions:

```
BaseException
     SystemExit
     KeyboardInterrupt
     GeneratorExit
     Exception
          StopIteration
          ArithmeticError
               FloatingPointError
               OverflowError
               ZeroDivisionError
          AssertionError
          AttributeError
          BufferError
          EnvironmentError
               IOError
               OSError
          EOFError
          ImportError
          LookupError
               IndexError
               KeyError
          MemoryError
          NameError
               UnboundLocalError
          ReferenceError
          RuntimeError
               NotImplementedError
          SyntaxError
               IndentationError
```

```
          TabError
     SystemError
     TypeError
     ValueError
          UnicodeError
               UnicodeDecodeError
               UnicodeEncodeError
               UnicodeTranslateError
     Warning
          DeprecationWarning
          PendingDeprecationWarning
          RuntimeWarning
          SyntaxWarning
          UserWarning
          FutureWarning
          ImportWarning
          UnicodeWarning
          BytesWarning
```

When defining your own exception, you should subclass from `Exception` or below. The reason for this is that other subclasses of `BaseException` are not necessarily “exceptions”. For example, if you caught `KeyboardInterrupt`, you wouldn’t be able to stop the process with control-C. If you caught `GeneratorExit`, generators would stop working.

Here is an exception for defining that a program is missing information:

```
>>> class DataError(Exception):
...     def __init__(self, missing):
...         self.missing = missing
```

Using your custom exception is easy:

```
>>> if 'important_data' not in config:
...    raise DataError('important_data missing')
```

Summary

This chapter introduced strategies for dealing with exceptions. In Look Before You Leap, you make sure the environment will not throw an error before trying something. In Easier to Ask for Forgiveness than Permission,

you wrap any code that you know might throw an error with a `try`/`catch` block. You should favor the latter style of programming in Python.

The various mechanisms for catching errors, raising them, and re-raising them were discussed. Finally, the chapter showed how you could subclass an existing exception to create your own.

Exercises

1. Write a program that serves as a basic calculator. It asks for two numbers, then it asks for an operator. Gracefully deal with input that doesn't cleanly convert to numbers. Deal with division by zero errors.
2. Write a program that inserts line numbers in front the lines of a file. Accept a filename being passed in on the command line. Import the `sys` module and read the filename from the `sys.argv` list. Gracefully deal with a bogus file being passed in.

Importing Libraries

The previous chapters have covered the basic constructs for Python. In this chapter, you'll learn about importing code. Many languages have the concept of *libraries* or reusable chunks of code. Python comes with a whole swath of libraries, commonly referred to as “batteries included”. You need to know how to use the batteries that are found in these libraries.

To use a library, you have to load the code from that library into your *namespace*. The namespace holds the functions, classes, and variables you have access to. If you want to calculate the sine of an angle you will need to define a function that does that or load a preexisting function. The built-in `math` library has a `sin` function that calculates the sine of an angle expressed in radians. It also has a variable that defines a value for pi:

```
>>> from math import sin, pi
>>> sin(pi/2)
1.0
```

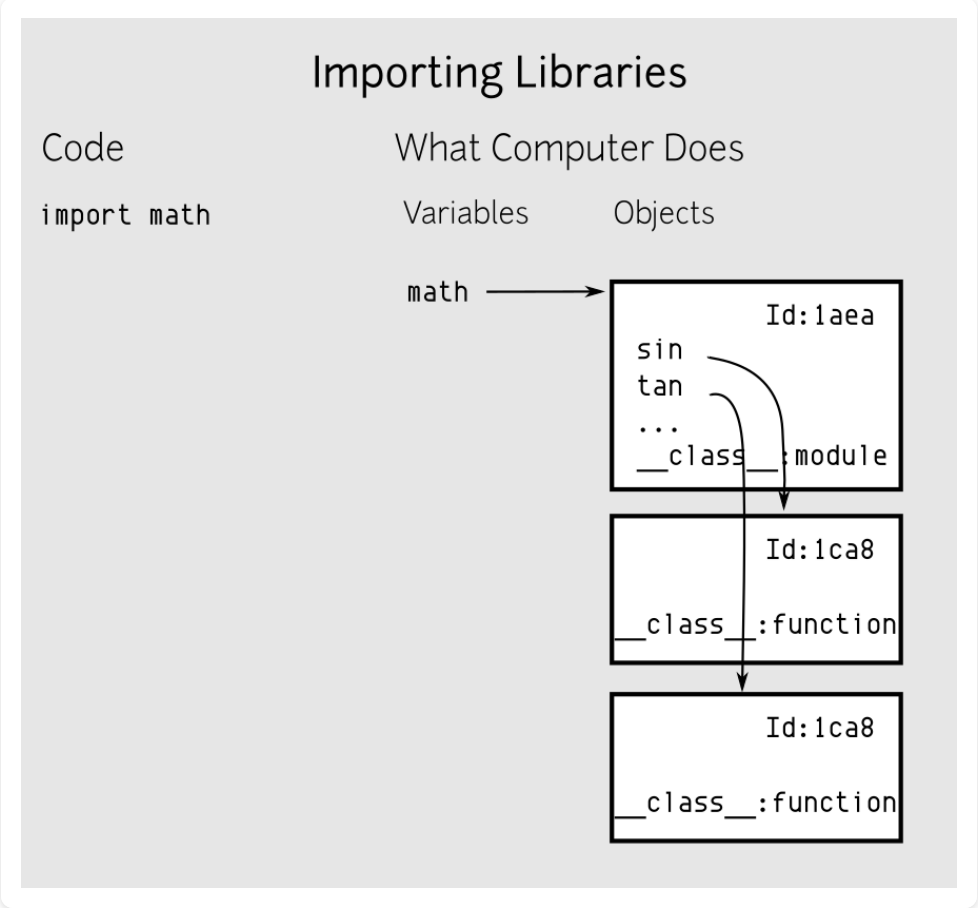
The above code loads the `math` module. But it doesn't put `math` in your namespace. Rather it creates a variable that points to the `sin` function from the `math` module. It also creates a variable that points to the `pi` variable found in the `math` module. If you inspect your current namespace using the `dir` function you can confirm this:

```
>>> 'sin' in dir()
True
```

Multiple ways to import

In the previous example, you imported a single function from a library. It is also possible to load the library into your namespace and reference all of its classes, functions, and variables. To import the `math` module into your namespace type:

```
>>> import math
```



This illustrates importing a module. Note that this code creates a new variable, `math`, that points to a module. The module has various attributes that you can access using a period.

In the above, you imported the `math` library. This created a new variable, `math`, that points to the module. The module has various attributes. You can use the `dir` function to list the attributes:

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
```

```
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Most of these attributes are functions. If you want to call the `tan` function, you can't invoke it, because `tan` isn't in your namespace, only `math` is. But, you can do a lookup on the `math` variable using the period (`.`) operator. The period operator will look up attributes on an object. Because everything in Python is an object, you can use the period to lookup the `tan` attribute on the `math` object:

```
>>> math.tan(0)
0.0
```

If you want to read the associated documentation for the `tan` function, you can enlist the `help` function for aid:

```
>>> help(math.tan)
Help on built-in function tan in module math:

tan(...)
    tan(x)

    Return the tangent of x (measured in radians).
```

Tip

When would you import a function using `from` versus the `import` statement? If you are only using a couple attributes from a library, you might want to use the `from` style import. It is possible to specify multiple comma-delimited attributes in the `from` construct:

```
>>> from math import sin, cos, tan
>>> cos(0)
1.0
```

However if you need to access most of the library, it requires less typing to import the library using the `import` statement. It is also a hint to anyone reading your code (including you), as to where the function (or class or variable) came from.

NOTE

If you need to import multiple attributes from a library, you might need to span multiple lines. If you continue importing functions on the next line, you will run into an error:

```
>>> from math import sin,
...   cos
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: trailing comma not allowed
without surrounding parentheses
```

You will need to use a backslash, to indicate that the line continues on the next line:

```
>>> from math import sin,\
...   cos
```

Otherwise, you can specify multiple imports with parentheses. Open parentheses (or braces, or brackets) indicate that the statement continues on the next line:

```
>>> from math import (sin,
...   cos)
```

This latter form is more idiomatic Python.

Conflicting import names

If you were working on a program that performs trigonometric operations, you might already have a function named `sin`. What if you also want to use the `sin` function from the `math` library? One option is to `import math`, then `math.sin` would reference the library and `sin` would reference your function.

Python has another option too. You can redefine the name of what you want to import using the `as` keyword:

```
>>> from math import sin as other_sin
>>> other_sin(0)
0.0
```

Now, `other_sin` is a reference to the `sin` found in `math` and you may continue using your `sin` without having to refactor your code.

The `as` keyword construct also works on `import` statements. If you had a variable (or a function) that conflicted with the `math` name in your namespace, the following would be one way to get around it:

```
>>> import math as other_math
>>> other_math.sin(0)
0.0
```

Tip

The `as` keyword can also be used to eliminate typing. If your favorite library has overly long and verbose names you can easily shorten them in your code. Users of the Numpy [16] library have adopted the standard of reducing keystrokes by using a two-letter acronym:

```
>>> import numpy as np
```

The Pandas [17] library has adopted a similar standard:

```
>>> import pandas as pd
```

Star imports

Python also allows you to clobber your namespace with what are known as *star imports*:

```
>>> from math import *
>>> asin(0)
```

```
0.0
```

Notice that the above code calls the arc sine function, which was not defined. The line where `asin` is invoked is the first reference to `asin` in the code. What happened? When you say `from math import *`, that is a star import, and it tells Python to throw everything from the `math` library (class definitions, functions, and variables) into the local namespace. While this might appear handy at first glance, it is quite dangerous.

Star imports make debugging harder because it is not explicit where code comes from. Even worse are star imports from multiple libraries. Subsequent library imports might override something defined in an earlier library. As such, star imports are discouraged and frowned upon.

TIP

Do not use star imports!

The possible exceptions to this rule are when you are writing your own testing code, or messing around in the REPL. Library authors do this as a shortcut to importing everything from the library that they want to test. And this often ends up in the documentation. Do not be tempted because you see it in other's code, to use star imports in your code.

Remember the Zen of Python:

> *Explicit is better than implicit*

Nested libraries

Some Python packages have a nested namespace. For example, the XML library that comes with Python has support for `minidom` and `etree`. Both

libraries live under the `xml` parent package:

```
>>> from xml.dom.minidom import \
...      parseString
>>> dom = parseString(
...     '<xml><foo/></xml>')

>>> from xml.etree.ElementTree import \
...      XML
>>> elem = XML('<xml><foo/></xml>')
```

Notice that the `from` construct allows importing only the functions and classes needed. Using the `import` construct (without `from`) would require more typing (but also allow access to everything from the package):

```
>>> import xml.dom.minidom
>>> dom = xml.dom.minidom.parseString(
...     '<xml><foo/></xml>')

>>> import xml.etree.ElementTree
>>> elem = xml.etree.ElementTree.XML(
...      '<xml><foo/></xml>')
```

Import organization

According to PEP 8, import statements should be located at the top of the file following the module docstring. There should be one import per line and imports should be grouped by:

- Standard library imports
- 3rd party imports
- Local package imports

An example module might have the following at the start:

```
#!/usr/bin/env python3
"""
This module converts records into JSON
and shoves them into a database
"""
import json             # standard libs
```

```
import sys

import psycopg2         # 3rd party lib

import recordconverter  # local library
...
```

TIP

It is useful to organize the grouped imports alphabetically.

TIP

It can be useful to postpone some imports to:

- Avoid *circular imports*. A circular import is where modules mutually import one another. If you are not able (or willing) to refactor to remove the circular import, it is possible to place the import statement within the function or method containing the code that invokes it.
- Avoid importing modules that are not available on some systems.
- Avoid importing large modules that you may not use.

Summary

This chapter discussed importing libraries in Python. Python has a large standard library [18], and you will need to import those libraries to use them. Throughout this book, it has emphasized that everything is an object, and you often create variables that point to those objects. When you import a module, you create a variable that points to a module object. Anything

within the module's namespace can be accessed using the lookup operation (`.`).

You can also selectively import parts out of the module's namespace, using the `from` statement. If you want to rename what you are importing, you can use an `as` statement to change what the variable name will be.

Exercises

1. Find a package in the Python standard library for dealing with JSON. Import the library module and inspect the attributes of the module. Use the `help` function to learn more about how to use the module. Serialize a dictionary mapping `'name'` to your name and `'age'` to your age, to a JSON string. Deserialize the JSON back into Python.
2. Find a package in the Python standard library for listing directory contents. Using that package, write a function that accepts a directory name. The function should get all the files in that directory and print out a report of the count of files by extension type.

16 - numpy.scipy.org
17 - pandas.pydata.org
18 - https://docs.python.org/3/library/index.html

Libraries: Packages and Modules

The previous chapter discussed how to import libraries. This chapter will dive a little deeper into what constitutes a library. There are two requirements for importing a library:

1. The library must be a *module* or a *package*
2. The library must exist in the `PYTHONPATH` environment variable or `sys.path` Python variable.

Modules

Modules are Python files that end in `.py`, and have a name that is importable. PEP 8 states that module filenames should be short and in lowercase. Underscores may be used for readability.

Packages

A *package* in Python is a directory that contains a file named `__init__.py`. The file named `__init__.py` can have any implementation it pleases or it can be empty. In addition, the directory may contain an arbitrary number of modules and sub packages.

When writing code, should you prefer a module or a package? I usually start simple and use a module. When I need to break coherent parts out into their own modules, then I refactor into modules in a package.

Here is an example from the directory layout of the popular SQLAlchemy [19] project (an Object Relational Mapper for databases).

```
sqlalchemy/
  __init__.py
  engine/
    __init__.py
    base.py
  schema.py
```

PEP 8 states that directory names for packages should be short and lowercase. Underscores should not be used.

Importing packages

To import a package, use the `import` statement with the package name (the directory name):

```
>>> import sqlalchemy
```

This will import the `sqlalchemy/__init__.py` file into the current namespace if the package is found in `PYTHONPATH` or `sys.path`.

If you wanted to use the `Column` and `ForeignKey` classes found in the `schema.py` module, either of the code snippets below would work. The first puts `sqlalchemy.schema` in your namespace, while the latter only puts `schema` in your namespace:

```
>>> import sqlalchemy.schema
>>> col = sqlalchemy.schema.Column()
>>> fk = sqlalchemy.schema.ForeignKey()
```

or:

```
>>> from sqlalchemy import schema
>>> col = schema.Column()
>>> fk = schema.ForeignKey()
```

Alternatively, to access only the `Column` class, import that class in one of the following two ways:

```
>>> import sqlalchemy.schema.Column
>>> col = sqlalchemy.schema.Column()
```

or:

```
>>> from sqlalchemy.schema import Column
>>> col = Column()
```

PYTHONPATH

PYTHONPATH is an environment variable listing non-standard directories that Python looks for modules or packages in. This variable is usually empty. It is not necessary to change PYTHONPATH unless you are developing code and want to use libraries that have not been installed.

> **TIP**
>
> Leave PYTHONPATH empty unless you have a good reason to change it. This section illustrates what can happen if you change it. This can be confusing to others trying to debug your code who forget that PYTHONPATH has been changed.

If you had some code in /home/test/a/plot.py, but were working out of /home/test/b/, using PYTHONPATH allows access to that code. Otherwise, if plot.py was not *installed* using system or Python tools, trying to import it would raise an ImportError:

```
>>> import plot
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named plot
```

If you start Python by setting the PYTHONPATH, it indicates to Python where to look for libraries:

```
$ PYTHONPATH=/home/test/a python3
Python 3.6.0 (default, Dec 24 2016, 08:01:42)
>>> import plot
```

```
>>> plot.histogram()
...
```

> **Tip**
>
> Python packages can be installed via package managers, Windows executables or Python specific tools such as pip [20].

sys.path

The `sys` module has an attribute, `path`, that lists the directories that Python searches for libraries. If you inspect `sys.path`, you will see all the locations that are scanned:

```
>>> import sys
>>> sys.path

['',
'/usr/lib/python35.zip',
'/usr/lib/python3.6',
'/usr/lib/python3.6/plat-darwin',
'/usr/lib/python3.6/lib-dynload',
'/usr/local/lib/python3.6/site-packages']
```

Tip

If you see errors like:

```
ImportError: No module named plot
```

Look at the `sys.path` variable to see if it has the directory holding `foo.py` (if it is a module). If `plot` is a package, then the `plot/` directory should be located in one of the paths in `sys.path`:

```
>>> import plot
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named plot
>>> sys.path.append('/home/test/a')
>>> import plot
>>> plot.histogram()
```

Alternatively, you can set `PYTHONPATH` to point to that directory from the command used to invoke Python.

Again, typically you don't manually set `sys.path` or `PYTHONPATH`, normally you install libraries, and the installer puts them in the correct location.

Tip

If you want to know the location of the library on the filesystem, you can inspect the `__file__` attribute:

```
>>> import json
>>> json.__file__
'/usr/lib/python3.6/json/__init__.py'
```

This only works with libraries implemented in Python. The `sys` module is not implemented in Python, so this fails:

```
>>> import sys
>>> sys.__file__
Traceback (most recent call last):
  ...
AttributeError: module 'sys' has no attribute '__file__'
```

Summary

This chapter discussed modules and packages. A module is a Python file. A package is a directory that has a file named `__init__.py` in it. A package may also contain other modules and packages.

There is a list of paths that determines where Python looks to import libraries. This list is stored in `sys.path`. You can inspect it to see where Python looks. You can also update its value via the `PYTHONPATH` environment variable, or mutate the list directly. But typically, you don't set this variable to install packages, you use something like pip to install packages.

Exercises

1. Create a module, `begin.py`, that has a function named `prime` in it. The `prime` function should take a number and return a boolean indicating

whether the number is a prime number (divisible only by 1 and itself). Go to another directory, launch Python, and run:

```
from begin import prime
```

It should fail. Update the `sys.path` variable, so that you can import the function from the module. Then, set `PYTHONPATH` to get it to load.

2. Create a package, `utils`. In the `__init__.py` file, place the `prime` code from the previous exercise. Go to a different directory in the terminal, launch Python, and run:

```
from utils import prime
```

It should fail. Update the `sys.path` variable, so that you can import the function from the package. Then, set `PYTHONPATH` to get it to load.

19 - https://www.sqlalchemy.org/

20 - https://pip.pypa.io/

A Complete Example

This chapter covers how to layout code within a script. It includes the source for a simplified implementation of the Unix command `cat`. The `cat` command reads filenames from the command line and prints their contents to the screen. There are various options to do things like add line numbering. This script will illustrate how the code is laid out in a typical Python file.

`cat.py`

Below are the contents of the Python implementation of the Unix command `cat`. It only includes an option for adding line numbers (`--number`), but none of the other `cat` options. You can put this in a file named `cat.py`:

```
#!/usr/bin/env python3

r"""A simple implementation of the unix ``cat``
command. It only implements the ``--number``
option. It is useful for illustrating file
layout and best practices in Python.

This is a triple quoted docstring for the whole
module (this file). If you import this module
somewhere else and run ``help(cat)``, you will
see this.

This docstring also contains a ``doctest`` which
serves as an example of programmatically using
the code. It also functions as a doctest. The
``doctest`` module can execute this docstring
and validate it by checking any output.

>>> import io
>>> fin = io.StringIO(\
```

```
...     'hello\nworld\n')
>>> fout = io.StringIO()
>>> cat = Catter([fin],
...     show_numbers=True)
>>> cat.run(fout)
>>> print(fout.getvalue())
     1  hello
     2  world

"""

import argparse
import logging
import sys


__version__ = '0.0.1'


logging.basicConfig(
    level=logging.DEBUG)


class Catter(object):
    """
    A class to concatenate files to
    standard out

    This is a class docstring,
    ``help(cat.Catter)`` would show
    this.
    """

    def __init__(self, files,
                 show_numbers=False):
        self.files = files
        self.show_numbers = show_numbers

    def run(self, fout):
        # use 6 spaces for numbers and right align
        fmt = '{0:>6}  {1}'
        for fin in self.files:
            logging.debug('catting {0}'.format(fin))
            for count, line in enumerate(fin, 1):
                if self.show_numbers:
                    fout.write(fmt.format(
                        count, line))
                else:
```

```
                    fout.write(line)


def main(args):
    """
    Logic to run a cat with arguments
    """
    parser = argparse.ArgumentParser(
        description='Concatenate FILE(s), or '
        'standard input, to standard output')
    parser.add_argument('--version',
        action='version', version=__version__)
    parser.add_argument('-n', '--number',
        action='store_true',
        help='number all output lines')
    parser.add_argument('files', nargs='*',
        type=argparse.FileType('r'),
        default=[sys.stdin], metavar='FILE')
    parser.add_argument('--run-tests',
        action='store_true',
        help='run module tests')
    args = parser.parse_args(args)

    if args.run_tests:
        import doctest
        doctest.testmod()
    else:
        cat = Catter(args.files, args.number)
        cat.run(sys.stdout)
        logging.debug('done catting')

if __name__ == '__main__':
    main(sys.argv[1:])
```

If you don't feel like typing in this code, you can grab a copy here [21].

What does this code do?

This code will echo the contents of a file (optionally with line numbers) to a terminal on Windows and Unix systems:

```
$ python3 cat.py -n README.md
     1  # IllustratedPy3
     2
     3  If you have questions or concerns, click on Issues above.
```

If you run this code with `-h` it will print out the help documentation for the command line arguments:

```
$ python3 cat.py -h
usage: cat.py [-h] [--version] [-n] [--run-tests] [FILE [FILE ...]]

Concatenate FILE(s), or standard input, to standard output

positional arguments:
  FILE

optional arguments:
  -h, --help    show this help message and exit
  --version     show program's version number and exit
  -n, --number  number all output lines
  --run-tests   run module tests
```

The command line parsing functionality is implemented in the `main` function and provided by the `argparse` module found in the standard library. The `argparse` module handles command line argument parsing.

If you want to know what the `argparse` module does, you can look up the documentation on the web or you can use the `help` function. The basic idea of the module is that you create an instance of the `ArgumentParser` class and call `.add_argument` for each command line option. You provide the command line switches, tell it what kind of action to do (the default stores the value following the switch), and provide help documentation. After you have added arguments, you call the `.parse_args` method on the command line arguments (we get that from `sys.argv`). The result of `.parse_args` is an object that has attributes attached to it based on the option names. In this case, there will be a `.files` attribute and a `.number` attribute.

NOTE

You can also use the REPL to inspect this module and any documentation that it ships with. Remember the `help` function? You can pass a module to it and it will print out the module level docstring.

If you wanted to look at the source code for this module, you can do that too. Remember the `dir` function? It lists attributes of an object. If you inspect the `argparse` module, you see that it has a `__file__` attribute. This points to the location on your computer where the file is:

```
>>> import argparse
>>> argparse.__file__
'/usr/local/Cellar/python3/3.6.0/Frameworks/
Python.framework/Versions/3.6/lib/python3.6/argparse.py'
```

Because it is written in Python (some modules are written in C), you can inspect the source. At this point, you should be able to read the module and understand what it is trying to do.

After parsing the arguments, you create an instance of the `Catter` class that is defined in the code, and call the `.run` method on it.

This example might seem a little overwhelming, but you have covered all of the syntax that it illustrates over the course of the book. The rest of this chapter will discuss the layout and other aspects of this code.

Common layout

Here are the common components found in a Python module and the order in which they are found:

- `#!/usr/bin/env python3` (shebang) (used if module also serves as a script.)

- module docstring
- imports
- metadata/globals
- logging
- implementation
- `if __name__ == '__main__'`: (used if module also serves as a script.)
- argparse

> **Note**
>
> The above list is a recommendation. Most of those items can be in an arbitrary order. And not every file will have all these items. For instance not every file needs to be executable as a shell script.
>
> You are free to organize files how you please, but you do so at your own peril. Users of your code will likely complain (or submit patches). As a reader of code, you will appreciate code that follows the recommendation, since it will be quickly discoverable.

Shebang

The first line in a file (that is also used as a script) is the *shebang* line (`#!/usr/bin/env python3`). On Unix operating systems, this line is parsed to determine how to execute the script. Thus, this line is only included in files that are meant to be executable as scripts.

It should say `python3`, as `python` refers to Python version 2 on most systems.

NOTE

The Windows platform ignores the shebang line. So this is safe to include. Indeed you will find it in libraries that are also popular on Windows.

NOTE

Rather than hardcoding a specific path to a Python executable, `/usr/bin/env` selects the first `python3` executable found on the user's `PATH`. Tools such as `venv` [22] will modify your `PATH` to use a custom `python3` executable and will work with this convention.

TIP

On Unix systems, if the directory containing the file is present in the user's `PATH` environment variable, and the file is executable, then the file name alone is sufficient for execution from a shell.

Type:

```
$ chmod +x <path/to/file.py>
```

to make it executable.

Docstring

A module may have a module-level docstring at the top of the file. It should follow the shebang line but precede any other Python code. A docstring serves as an overview of the module and should contain a basic summary of the code. Also, it may contain examples of using the module.

Tip

Python includes a library, `doctest` that can verify examples from an interactive interpreter. Using docstrings that contain REPL code snippets can serve both as documentation and simple sanity tests for your library.

`cat.py` includes doctest code at the end of its docstring. When `cat.py` runs with `--run-tests`, the `doctest` library will check any docstrings and validate the code found in them. This was included for illustration purposes only. Normally a non-developer end user would not see options for running tests in a script, though you could include doctests in the docstrings. In this case, the `--run-tests` option is included as an example of using the `doctest` module.

Imports

Imports are usually included at the top of Python modules. The import lines are normally grouped by location of the library. First, list any libraries found in the Python standard library. Next list third-party libraries. Finally, list the libraries that are local to the current code. Such organization allows end users of your code to quickly see imports, requirements, and where the code is coming from.

Metadata and globals

If you have legitimate module-level global variables, define them after the imports. This makes it easy to scan a module and quickly determine what the globals are.

Global variables are defined at the module-level and are accessible throughout that module. Because Python allows any variable to be modified, global variables are potential sources of bugs. In addition, it is

easier to understand code when variables are defined and modified only within the function scope. Then you can be sure of what data you have and who is changing it. If you have multiple places where a global variable is being modified (especially if it is in a different module) you are setting yourself up for a long debugging session.

One legitimate use for globals is to emulate *constants* found in other programming languages. A constant variable is a variable whose value does not change. Python doesn't support variables that don't change, but you can use a convention to indicate to the user that they should treat a variable as read-only. PEP 8 states that global constants should have names that are the same as variables except they should be capitalized. For example, if you wanted to use the golden ratio you could define it like this:

```
>>> GOLDEN_RATIO = 1.618
```

If this code was defined in a module, the capitalization serves as a hint that you should not rebind this variable.

NOTE

By defining constants as globals, and using well thought-out variable names, you can avoid a problem found in programming—*magic numbers*. A magic number is a number sitting in code or a formula that is not stored in a variable. That in itself is bad enough, especially when someone else starts reading your code.

Another problem with magic numbers is that the same value tends to propagate through the code over time. This isn't a problem until you want to change that value. Do you do a search and replace? What if the magic number actually represents two different values, i.e. the number of sides of a triangle and number of dimensions? In that case, a global search and replace will introduce bugs.

The solution to both these problems (context and repetition) is to put the value in a named variable. Having them in a variable gives context and naming around the number. It also allows you to easily change the value in one place.

In addition to global variables, there are also *metadata* variables found at that level. Metadata variables hold information about the module, such as author and version. Normally metadata variables are specified using "dunder" variables such as `__author__`.

For example, PEP 396 recommends that the module version should be specified in a string, `__version__`, at the global module level.

> **Note**
>
> It is a good idea to define a version for your library if you intend on releasing it to the wild. PEP 396 suggests best practices for how to declare version strings.

Other common metadata variables include author, license, date, and contact. If these were specified in code they might look like this:

```
__author__ = 'Matt Harrison'
__date__ = 'Jan 1, 2017'
__contact__ = 'matt_harrison <at> someplace.com'
__version__ = '0.1.1'
```

Logging

One more variable that is often declared at the global level is the logger for a module. The Python standard library includes the `logging` library that allows you to report different levels of information in well-defined formats.

Multiple classes or functions in the same module will likely need to log information. It is common to perform logger initialization once at the global level and then reuse the logger handle that you get back throughout the module.

Other globals

You should not use a global variable when local a variable will suffice. The common globals found in Python code are metadata, constants, and logging.

It is not uncommon to see globals scattered about in sample code. Do not fall to the temptation to copy this code. Place it into a function or a class. This will pay dividends in the future when you are refactoring or debugging your code.

Implementation

Following any global and logging setup comes the actual meat of the code—the implementation. Functions and classes will be a substantial portion of this code. The `Catter` class would be considered the core logic of the module.

Testing

Normally, bonafide test code is separated from the implementation code. Python allows a small exception to this. Python docstrings can be defined at module, function, class, and method levels. Within docstrings, you can place Python REPL snippets illustrating how to use the function, class, or module. These snippets, if well crafted and thought-out, can be effective in documenting common usage of the module.

Another nice feature of `doctest` is validation of documentation. If your snippets once worked, but now they fail, either your code has changed or your snippets are wrong. You can easily find this out before end users start complaining to you.

TIP

`doctest` code can be in a stand-alone text file. To execute arbitrary files using doctest, use the `testfile` function:

```
import doctest
doctest.testfile('module_docs.txt')
```

NOTE

In addition to `doctest`, the Python standard library includes the `unittest` module that implements the common xUnit style methodology—setup, assert and teardown. There are pro's and con's to both `doctest` and `unittest` styles of testing. `doctest` tends to be more difficult to debug, while `unittest` contains boilerplate code that is regarded as too Java-esque. It is possible to combine both to achieve well documented and well tested code.

`if __name__ == '__main__':`

If your file is meant to be run as a script, you will find this snippet at the bottom of your script:

```
if __name__ == '__main__':
    sys.exit(main(sys.argv[1:]) or 0)
```

To understand the statement, you should understand the `__name__` variable.

`__name__`

Python defines the module level variable `__name__` for any module you *import*, or any file you execute. Normally `__name__`'s value is the name of the module:

```
>>> import sys
>>> sys.__name__
'sys'
>>> import xml.sax
>>> xml.sax.__name__
'xml.sax'
```

There is an exception to this rule. When a module is *executed* (i.e. `python3 some_module.py`), then the value of `__name__` is the string

"__main__".

In effect, the value of __name__ indicates whether a file is being loaded as a library, or run as a script.

NOTE

It is easy to illustrate __name__. Create a file, some_module.py, with the following contents:

```
print("The __name__ is: {0}".format(__name__))
```

Now run a REPL and *import* this module:

```
>>> import some_module
The __name__ is: some_module
```

Now *execute* the module:

```
$ python3 some_module.py
 The __name__ is: __main__
```

It is a common idiom throughout Pythondom to place a check similar to the following at the bottom of a module that could also serve as a script. This check will determine whether the file is being executed or imported:

```
if __name__ == '__main__':
    # execute
    sys.exit(main(sys.argv[1:]) or 0)
```

This simple statement will run the main function when the file is executed. Conversely, if the file is used as a module, main will not be run automatically. It calls sys.exit with the return value of main (or 0 if main does not return an exit code) to behave as a good citizen in the Unix world.

The main function takes the command line options as parameters. sys.argv holds the command line options. It also contains python3 at the

front, so you need to slice `sys.argv` to ignore that, before you pass the options into `main`.

> **Tip**
>
> Some people place the execution logic (the code inside the `main` function) directly under the `if __name__ == '__main__':` test. Reasons to keep the logic inside a function include:
>
> - The `main` function can be called by others
> - The `main` function can be tested easily with different arguments
> - Reduce the amount of code executing at the global level

Summary

In this chapter, you dissected Python code in a script. The chapter discussed best practices and common coding conventions.

By laying out your code as described in this chapter, you will be following best practices for Python code. This layout will also aid others needing to read your code.

Exercises

1. Copy the `cat.py` code. Get it working on your computer. This isn't just busy work. Much of the time when you are programming, you aren't creating something from scratch, but rather are reusing code that others have written.
2. Write a script, `convert.py`, that will convert a file from one encoding to another. Accept the following command line options:
    - An input filename

- An input encoding (default to utf-8)
- An output encoding
- An option for handling errors (ignore/raise)

21 - https://github.com/mattharrison/IllustratedPy3/

22 - https://docs.python.org/3/library/venv.html

Onto Bigger and Better

At this point you should have a pretty good understanding of how Python programs work. You should feel comfortable using the Python REPL and exploring classes using `dir` and `help`.

What comes next? That is up to you. You should now have the prerequisites to use Python to create websites, GUI programs, or numerical applications.

One huge benefit of Python is the various communities associated with the different areas of programming. There are local user groups, newsgroups, mailing lists and social networks for many aspects of Python. Most of these groups are welcoming to new programmers and willing to share their knowledge. Do not be afraid to try something new, Python makes it easy and most likely there are others who have similar interests.

File Navigation

If you are not familiar with file navigation from a terminal, here is a basic introduction. First, you need to open a terminal. A terminal is one of those windows you see on the movies where hackers type in a lot of text. You don't have to use one to program, but being able to navigate and run commands from the terminal is a useful skill to have.

Mac and Unix

On Mac, type command-space to bring up Spotlight, then type `terminal` to launch the terminal application included on Macs.

On Linux systems, launching a terminal depends on your desktop environment. For example, on Ubuntu systems you can hit the keyboard shortcut ctr-alt-T. A minimal terminal found in most systems is called `xterm`.

There are a few commands to know:

- `cd` - *Change directory* changes to a different directory. Typing:

  ```
  $ cd ~/Documents
  ```

  will change to the `Documents` directory found in your home folder (`~` is a shortcut for home, which is `/Users/<username>` on Mac and `/home/<username>` on Linux.
- `pwd` - *Print working directory* will list the current directory that you are in.
- `ls` - *List directory* will list the contents of the current directory.

If you had a Python script located at `~/work/intro-to-py/hello.py`, you could type the following to run it:

```
$ cd ~/work/intro-to-py
$ python3 hello.py
```

Windows

On Windows, type Win-R to bring up the "Run Window" interface, then type `cmd` to launch the command prompt.

There are a few commands to know:

- `cd` - *Change directory* changes to a different directory. Typing:

  ```
  c:> cd C:\Users
  ```

  will change to the `C:\Users` directory.
- `echo %CD%` - Will list the current directory that you are in.
- `dir` - List *directory* will list the contents of the current directory.

If you had a Python script located at `C:\Users\matt\intro-to-py\hello.py`, you could type the following to run it:

```
C:> cd C:\Users\matt\intro-to-py
C:\Users\matt\intro-to-py> python hello.py
```

Useful Links

Here are some useful Python links:

- https://python.org/ - Python home page
- https://github.com/mattharrison/Tiny-Python-3.6-Notebook - Python 3.6 reference
- http://docutils.sourceforge.net/ - reStructuredText - lightweight markup language for Python documentation
- https://pyformat.info - Useful string formatting reference
- https://pypi.python.org/pypi - Python Package Index - 3rd party packages
- https://www.python.org/dev/peps/pep-0008/ - PEP 8 - Coding conventions
- https://www.anaconda.com/download/ - Anaconda - Alternate Python installer with many 3rd party packages included
- https://www.djangoproject.com/ - Django - Popular web framework
- http://scikit-learn.org/ - Machine learning with Python
- https://www.tensorflow.org/ - Deep learning with Python
- https://www.reddit.com/r/Python/ - News for Python

About the Author

Matt Harrison has been using Python since 2000. He runs MetaSnake, a Python and Data Science consultancy and corporate training shop. In the past, he has worked across the domains of search, build management and testing, business intelligence, and storage.

He has presented and taught tutorials at conferences such as Strata, SciPy, SCALE, PyCON, and OSCON as well as local user conferences. The structure and content of this book is based on first-hand experience teaching Python to many individuals.

He blogs at `hairysun.com` and occasionally tweets useful Python related information at `@__mharrison__`.

About the Technical Editors

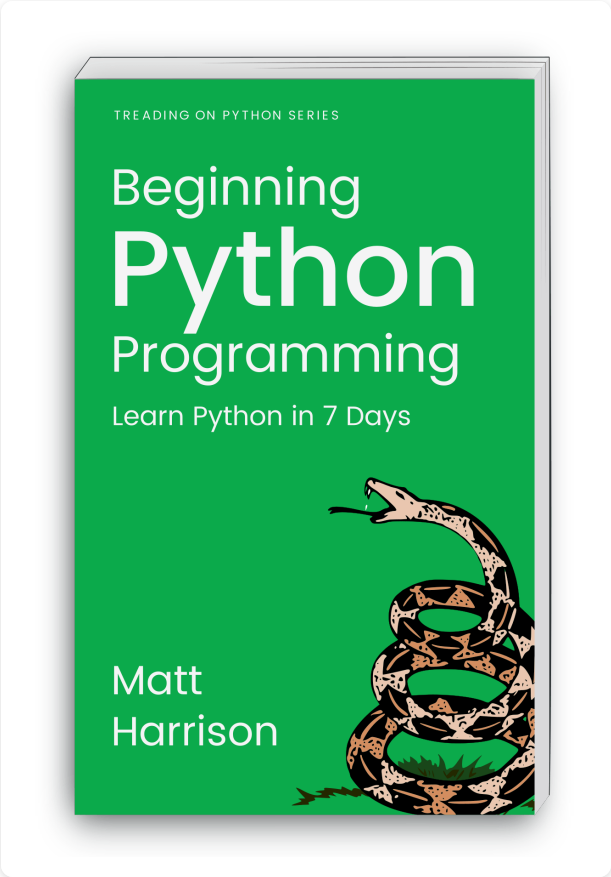
Roger A. Davidson is currently the Dean of Mathematics at American River College in Sacramento, CA. His doctorate is in aerospace engineering, but

he also holds degrees in computer science, electrical engineering and systems engineering in addition to a recent graduate certificate in data science (where his current love of Python began). During his career thus far Roger has worked for NASA, Fortune 50 companies, startups, and community colleges. He is passionate about education, science (not just the data kind), wild blackberry cobbler and guiding diverse teams to solve big problems.

Andrew McLaughlin is a programmer and designer, a sysadmin by day, and family man by night. With a love for detail, he's been building things for the web since 1998. A graduate with honors from George Fox University, Andrew has a degree in Management and Information Systems. In his free time he enjoys hiking with his wife and two kids, and occasionally working in his woodshop. He has all ten fingers. Follow him on twitter: `@amclaughlin`

Also Available

Beginning Python Programming



Treading on Python: Beginning Python Programming [23] by Matt Harrison is the complete book to teach you Python fast. Designed to up your Python game by covering the basics:

- Interpreter Usage
- Types
- Sequences

- Dictionaries
- Functions
- Indexing and Slicing
- File Input and Output
- Classes
- Exceptions
- Importing
- Libraries
- Testing
- And more …

Reviews

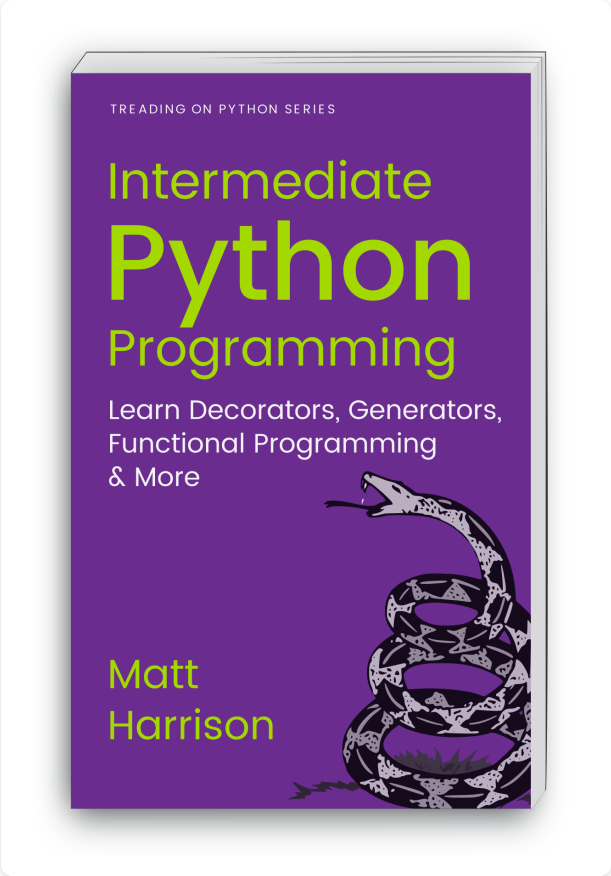
Matt Harrison gets it, admits there are undeniable flaws and schisms in Python, and guides you through it in short and to the point examples. I bought both Kindle and paperback editions to always have at the ready for continuing to learn to code in Python.
—S. Oakland

This book was a great intro to Python fundamentals, and was very easy to read. I especially liked all the tips and suggestions scattered throughout to help the reader program Pythonically :)
—W. Dennis

You don't need 1600 pages to learn Python
Last time I was using Python when Lutz's book Learning Python had only 300 pages. For whatever reasons, I have to return to Python right now. I have discovered that the same book today has 1600 pages.
Fortunately, I discovered Harrison's books. I purchased all of them, both Kindle and paper. Just few days later I was on track.

Harrison's presentation is just right. Short and clear. There is no 50 pages about the Zen and philosophy of using if-then-else construct. Just facts.
—A. Customer

Treading on Python: Vol 2: Intermediate Python



Treading on Python: Vol 2: Intermediate Python [24] by Matt Harrison is the complete book on intermediate Python. Designed to up your Python game by covering:

- Functional Programming
- Lambda Expressions
- List Comprehensions
- Generator Comprehensions

- Iterators
- Generators
- Closures
- Decorators
- And more …

Reviews

Complete! All you must know about Python Decorators: theory, practice, standard decorators.
All written in a clear and direct way and very affordable price.
Nice to read in Kindle.
—F. De Arruda (Brazil)

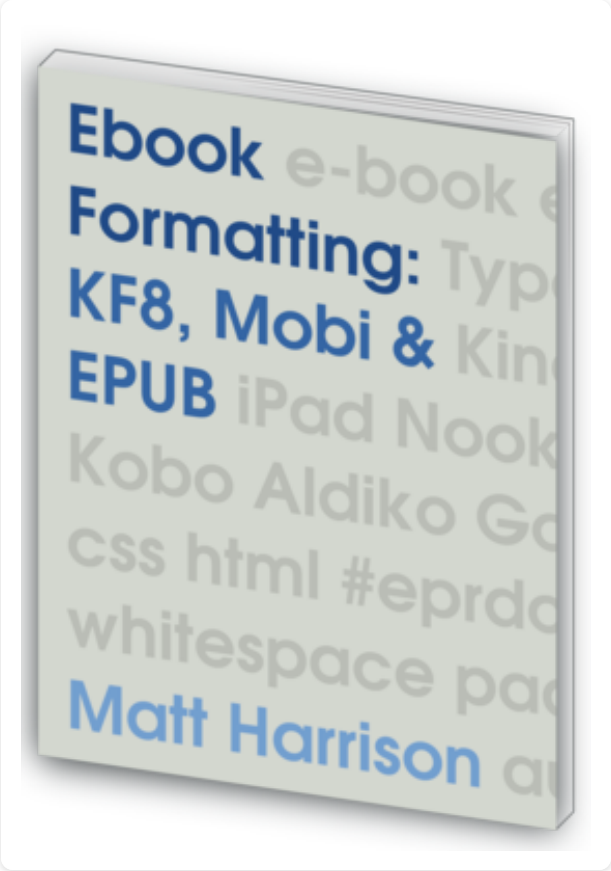
This is a very well written piece that delivers. No fluff and right to the point, Matt describes how functions and methods are constructed, then describes the value that decorators offer.
…
Highly recommended, even if you already know decorators, as this is a very good example of how to explain this syntax illusion to others in a way they can grasp.
—J Babbington

Decorators explained the way they SHOULD be explained …
There is an old saying to the effect that "Every stick has two ends, one by which it may be picked up, and one by which it may not." I believe that most explanations of decorators fail because they pick up the stick by the wrong end.
What I like about Matt Harrison's e-book "Guide to: Learning Python Decorators" is that it is structured in the way that I think an

introduction to decorators should be structured. It picks up the stick by the proper end…
Which is just as it should be.
—S. Ferg

This book will clear up your confusions about functions even before you start to read about decoration at all. In addition to getting straight about scope, you'll finally get clarity about the difference between arguments and parameters, positional parameters, named parameters, etc. The author concedes that this introductory material is something that some readers will find "pedantic," but reports that many people find it helpful. He's being too modest. The distinctions he draws are essential to moving your programming skills beyond doing a pretty good imitation to real fluency.
—R. Careago

Ebook Formatting: KF8, Mobi & EPUB



Ebook Formatting: KF8, Mobi & EPUB by Matt Harrison is the complete book on formatting for all Kindle and EPUB devices. A good deal of the eBooks found in online stores today have problematic formatting. *Ebook Formatting: KF8, Mobi & EPUB* is meant to be an aid in resolving those issues. Customers hate poorly formatted books. Read through the reviews on Amazon and you'll find many examples. This doesn't just happen to self-published books. Books coming out of big Publishing Houses often suffer similar issues. In the rush to put out a book, one of the most important aspects affecting readability is ignored.

This books covers all aspects of ebook formatting on the most popular devices (Kindle, iPad, Android Tablets, Nook and Kobo):

- Covers

- Title Pages
- Images
- Centering stuff
- Box model support
- Text Styles
- Headings
- Page breaks
- Tables
- Blockquotes
- First sentence styling
- Non-ASCII characters
- Footnotes
- Sidebars
- Media Queries
- Embedded Fonts
- Javascript
- and more …

23 - http://hairysun.com/books/tread/

24 - http://hairysun.com/books/treadvol2/

One more thing

THANK YOU FOR BUYING AND READING THIS BOOK.

If you have found this book helpful, I have a big favor to ask. As a self-published author, I don't have a big Publishing House with lots of marketing power pushing my book. I also try to price my books so that they are much more affordable.

If you enjoyed this book, I hope that you would take a moment to leave an honest review on Amazon. A short comment on how the book helped you and what your learned makes a huge difference. A quick review is useful to others who might be interested in the book.

Thanks again!

Write a review here.

Table of Contents