

The background of the book cover is a photograph of a large, multi-story stone building, likely a university hall, with a prominent chimney and several windows. Bare trees are visible in the foreground and background, suggesting a late autumn or winter setting. The image is slightly faded to allow the text to stand out.

Python for Scientists

Introduction to Python for Scientific Computing

Zeroth Edition

Sejin Kim

Evaluation Copy

Python for Scientists

Introduction to Python for Scientific Computing

Zeroth Edition

Sejin Kim

Kenyon College, Gambier, Ohio



Kenyon College
Department of Mathematics & Statistics - Scientific Computing
Gambier, Ohio

Credits and acknowledgements in this textbook appear on the appropriate page within the text.

Many of the designations by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was made aware of a trademark claim, the designations have been printed in initial caps or all caps.

This book was typeset using L^AT_EX software.

Preface

This book is meant to be used in a first course in programming and computer science using the Python programming language. It takes the Python-as-a-tool approach by using practical examples and introducing real-world libraries that increase Python's capabilities to scientists. It assumes no prior programming experience and no mathematics experience beyond high school algebra.

Flexibility in Topic Ordering

This book was written to allow instructors latitude in reordering the material. To illustrate the versatility in material ordering, here is an alternative material flow.

Reordering: Variables First

In order to understand how variables work and how they are stored, a student needs to understand the different datatypes. The basic material covered in Chapter 3 covers the primitive datatypes in Python, whereas variables are covered in Chapter 4.

Accessibility to Students

It is not enough for a book to simply present the right topics in the right order. It is not even enough for it to be clear and correct when read by an instructor or some other experienced programmer, like a tutor. It's important that the material is presented in a way that is accessible to beginning students. However, subsequent versions of this textbook may improve readability. The authors and publishers encourage students and instructors alike to submit suggestions and recommendations for improving the readability of the content for future editions.

Edition Compatibility

All efforts will be made to make different editions compatible. Future editions will be written with two or three sections. For example, a version of the first edition may be written as 1.3.1. However, the book and subsequent editions are

written in a way that a student could read any version of the first edition. That is, versions 1.0.0, 1.1.0, and 1.1.1 should all be interoperable by material. Page numbers may differ slightly. We would encourage instructors to reference material based on chapter number, rather than page number, especially because page numbers do not line up between the instructor's edition and the student edition.

Printing

When printing this book, we *highly recommend* printing in color and at 100% scaling to ensure readability of code. Code snips in this book have been color-coded using color coding, a common feature found in many IDEs, including Visual Studio Code and Atom. At the very least, the colors will allow students to read and distinguish different elements of code.

Standards and Platforms

This edition is fully compatible with base Python 3 (tested up to Python 3.6). Additional packages include Pandas v.1, BeautifulSoup4 v4.9, and TensorFlow v.2. This book is intended to be operating system-agnostic. Where we provide operating-system-specific guidance, we will provide it for Windows 8.1 or later, macOS 10.12 or later, *and* Linux (based on Debian 8 or later and RHEL 7 or later).

Support Material

There is support material available to all users of this book and additional material available only to qualified instructors. All material is provided free of charge and under the MIT License.

Materials Available to All Users

- Source code from the book
- Student-led collaborative classroom activities (typically in Process Oriented Guided Inquiry Learning, or POGIL form)

Materials Available to Instructors Only

- Instructor's edition of the text, which includes instructor-specific notes and exercise answers
- POGIL answer documents

Integrated Development Environment (IDE) Recommendations

This text is written in such a way that instructors can use any integrated development environment that supports Python 3. However, the authors suggest the following IDEs, pending availability.

- **repl.it®** is an online and collaborative IDE that runs entirely in the user's web browser.
- **Spyder** is part of the Anaconda package for scientific Python development. It runs on an individual computer's hardware. For students who are more familiar with RStudio, it is possible to change the default layout to mimic RStudio with the editor in the top left, console in the bottom left, environment in top right, and help in the bottom right. You can make this change by going to View › Window layouts › Rstudio layout.
- **Microsoft® Visual Studio** is a much more advanced IDE, and being so feature rich, it may be *overpowered* for teaching students how to program. We would recommend using Spyder instead.
- **The R Project for Statistical Computing and RStudio** are primarily focused with statistics in R, but R and RStudio have a built-in and very comprehensive Python interpreter called Reticulate (a play on words with reticulated pythons).

Contributing

I believe that the greatest strength in an open-source textbook is the very fact that anyone can edit and improve it for the greater good. We encourage students and instructors alike to submit their suggested changes and recommendations for future editions of this book. This book will be published in an open repository on GitHub, and we invite changes in the following ways.

- Open an issue and describe what you think should be changed
- Edit the content yourself, then make a pull/merge request
- Contact the authors directly with what you think should be changed

License

The material in this book is licensed under the MIT license. A copy of the license is provided below.

Copyright © 2022 Sejin Kim

Permission is hereby granted, free of charge, to any person

obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Acknowledgements

Many people have assisted me by providing their suggestions, discussions, and other help in preparing this textbook. Much of the work for the zeroth edition of this book was written while I was working with the wonderful people at the Kenyon College Department of Mathematics and Statistics and in the Integrated Program in Humane Studies. I'd also like to extend a special thanks to the individuals who contributed by proofreading and contributing to this text: Dr. James Skon (Professor of Mathematics, Kenyon College), Dr. Nuh Aydin (Professor of Mathematics, Kenyon College), Dr. Bradley Hartlaub (Professor of Mathematics, Kenyon College), Dr. Jon Chun (Professor in the Integrated Program in Humane Studies, Kenyon College), Dr. Katherine Elkins (Professor in the Integrated Program in Humane Studies, Kenyon College), Ashleigh Zarley (LBIS, Kenyon College), Carter Schoenfeld (LBIS, Kenyon College), Josh Katz (Kenyon College), Dev Akre (Kenyon College), and George Novotny (Kenyon College).

Table of Contents

1	Introduction	1
1.1	Reading This Book	2
2	Development Basics	5
2.1	Introduction to Python	5
2.1.1	A Brief History of Python	5
2.1.2	Characteristics of Python Scripts	6
2.1.3	Layout of a Simple Python Script	7
2.2	The Tools	10
2.3	Hardware and Software	13
2.3.1	Hardware	13
2.3.2	Software	15
2.4	How to Program	17
2.4.1	A New Type of Problem	17
2.4.2	The Power of Pseudocode	17
3	Basic Datatypes	19
3.1	Strong and Weak Typing	20
3.2	Booleans	22
3.3	Numbers	25
3.3.1	Integers	25
3.3.2	Floats	26
3.4	Strings	28
4	General Python Programming	31
4.1	Basic Operations	31
4.1.1	Printing to the Console	31
4.2	Variables	37
4.2.1	Declaration and Initialization	37
4.2.2	Referring to Variables	42
4.2.3	Handling User Input	43
4.3	Whitespace	46
4.3.1	Why is Whitespace Important?	46
4.3.2	Common Whitespace Patterns	47

4.4	Comments	51
4.4.1	Line Comments	51
4.4.2	Block Comments	52
4.5	Errors	55
4.5.1	Logical Versus Syntactical Errors	55
4.5.2	Debugging	58
4.5.3	Handling Errors On the Fly	62
4.6	Typecasting	66
4.7	F-strings	70
4.8	Statements and Expressions Review	72
4.9	Arithmetic Operations	74
5	Complex Datatypes	81
5.1	Lists	82
5.1.1	Creating Lists	82
5.1.2	Accessing Data Inside of Lists	85
5.1.3	Appending to Lists	87
5.2	Dictionaries	91
5.2.1	Lists Versus Dictionaries	91
5.2.2	Creating Dictionaries	91
5.2.3	Accessing Data Inside of Dictionaries	93
5.2.4	Appending to Dictionaries	93
5.3	Tuples	96
5.4	Sets	98
6	Conditional Logic and Loops	99
6.1	Comparison Operators and Boolean Expressions	100
6.2	If, Else, and Else If Statements	104
6.3	For Loops	110
6.4	While Loops	116
6.5	Scope	121
7	Python Data Structures	125
7.1	Functions	126
7.1.1	Function Definition	126
7.1.2	Function Calls	130
7.2	Classes	134
7.2.1	Class Definitions	134
7.2.2	Class Calls	135
7.2.3	Class Methods	135
7.3	Functions Versus Methods	136
7.4	Structures (Optional)	137

8 File Handling	139
8.1 Reading Files	140
8.2 Writing Files	143
8.3 Different Kinds of Files	145
9 Jupyter Notebooks	147
9.1 What Are Jupyter Notebooks?	148
9.2 Basic Markdown Syntax	149
9.3 The Interactive Python Shell	152
10 Data Analysis with Pandas	153
10.1 Pandas are Not Bears	154
10.2 New Datatypes	156
10.3 Series	157
10.4 Dataframes	163
10.4.1 Getting Data	164
Making a Dataframe Naively	164
Reading a CSV	165
Reading a Pickle	167
10.4.2 Adding Data	168
10.4.3 Removing Data	168
10.4.4 Modifying Data	168
10.4.5 Transforming Data	168
10.5 Cleaning Up Data	169
10.6 Calculating Summary Statistics	170
10.7 Basic Hypothesis Tests with statsmodels	171
10.7.1 Z-Test for Proportions	171
10.7.2 T-Test for Differences in Means	171
10.7.3 Kruskal-Wallis Rank Test for Differences in Medians	171
10.8 Building Basic Models	172
10.8.1 Simple Linear Regression	173
10.8.2 Logistic Regression	173
10.9 Basic Graphs with Matplotlib	173
11 Requests, Web Scraping, and BS4	175
11.1 Webpage Structure	176
11.1.1 HTML	176
11.1.2 CSS and JavaScript	178
11.1.3 The requests Library	179
11.1.4 Parsing with Beautiful Soup	180
11.1.5 APIs	181
11.1.6 Reading a JSON response	181
11.1.7 Parsing JSON	182
Index	201

Evaluation Copy

Chapter 1

Introduction

Why should we study development, software or technology? Software is becoming a larger and larger part of our daily lives. Different machines use different forms of software. From ATMs to parking meters, compute servers to emergency radios, software runs our lives increasingly. In fact, if all of the software stopped working, planes would just drop out of the skies, stocks would crash instantly, and global communications would cease to exist.

Python is only one of many different programming languages out there, and it has its own special place in the development world. Other languages include C++, Haskell, Ruby, R, Java, JavaScript, and even Ada. Once you pick up Python, though, you'll find that other programming languages don't look so different.

You might have noticed that compared to many other code textbooks, this book is not nearly as long. We've tried to trim out much of the "fat" that is good in small amounts but becomes overburdening in larger quantities. Instead, we prefer to give you realistic code and problems that show you why you're doing what you're doing. Rather than pad the page count with information that you'll never use, we've instead chosen to give you the important stuff.

What is this text not? It's not a comprehensive view of the software development landscape, or the hardware or firmware world. We won't discuss in detail what a CPU does or how it works, or how a temperature sensor on a motherboard sends input. These topics (software development, physical interface control, computer architecture, among others) are sufficiently advanced topics that can take their own semester to learn. Well, we don't have all the time in the world, so we're going to go over some of the basics of Python instead and delve into some cool stuff you can do with the language.

1.1 Reading This Book

Most of this book is written in plain English, not in code. Standard textbook material will appear like this, in this font, with this formatting.

Sometimes, though, we will need to give you some code. Python code appears like this.

```
1 # A for loop
2 for i in range(8):
3     print("Iteration" + i)
```

Don't worry if you don't understand this code. What you should notice are the colors. Keywords have been written in blue and orange, operators in orange, strings in red, and comments in green. All of this helps you understand the code that was written to you, as a human. Your IDE, or integrated development environment, will also color the text, though the colors might be different. Even if the colors are different, they should be consistent throughout the document, and as you get used to programming, you will better understand what the colors mean.

Pay attention to the line numbers! Line numbers are marked on the left side of the script, and they indicate single lines as they are stored in the file, not as they are rendered. This is why a single logical line might span multiple physical lines, in the case where we've run out of space on the page. We'll cover this (it's called wrapping) in chapter 4.3, but for now, just be aware that you should read the line number on the left side of this textbook. Sometimes, we'll also give you some output. Output has no color formatting, and it just appears in a regular monospace font.

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
```

The purpose of this is to differentiate it from the regular English text and also from the Python code. This is what Python might output.

We've also provided you with some extra notes, warnings, and fun facts. These show up in colored boxes (one more reason to print in color or just view the textbook on your computer as a digital edition).

Note

Notes are provided in blue boxes. These can help you understand what you're doing a little bit better, but aren't, strictly speaking, necessary.

Warning

Warnings are provided in orange boxes. These can help you avoid massive headaches, and we'll try to warn you of common pitfalls here.

Fun Fact

Fun facts are provided in green boxes. These aren't very helpful, but they are fun! (At least we think so...)

Evaluation Copy

Chapter 2

Development Basics

2.1 Introduction to Python

2.1.1 A Brief History of Python

Most people assume that the creator of Python was perhaps referencing his favorite reptile when creating a name for the language, but Python's creator, Guido van Rossum, actually named the language after his favorite comedy group, Monty Python. In fact, take a look at the official Python release documentation and you'll find "something completely different," often a chunk of a Monty Python script or a notable quote. While most languages use *foo* and *bar* for their metasynctatic variables, Python's documentation uses *spam* and *eggs* instead.¹ Because of this, when you're reading documentation, it's useful to recognize what metasynctatic variables are being used and how. This will come with experience, we promise!

Python was originally derived from the ABC programming languages, but it had much larger feature support and easier syntax, which made it more accessible to more programmers. Python has undergone two major revisions since its original conception. Python 1 was the original Python, written in the 80s by van Rossum. Later, it was picked up by the programming community and embraced for its great utility as a rapid prototyping language and easy-to-learn syntax with relative power, and Python 2 was born in 2000. Python 2 introduced many great features, like Unicode support (yes, you can use emojis in your Python code, since they're valid Unicode characters!) and better garbage collection, which improved memory performance in long-running scripts. Finally, Python 3 was released in 2008, but much to the dismay of existing Python programmers, its syntax was not backwards compatible with Python 2, meaning that all of the Python 2 scripts had to be rewritten for the new Python 3 syntax. With Python 2's end-of-life in early 2020, Python 3 became the default for any-

¹"Metasynctatic variables" is just a fancy term for placeholders. For example, in a Python for loop, documentation will use `for spam in eggs` instead of `for foo in bar`.

one learning to program in Python, including you! While Python had its own influences, it also played roles in the development of new languages, including JavaScript, Ruby, and Swift.

2.1.2 Characteristics of Python Scripts

Python scripts have several characteristics about them that make them identifiable in the sea of source code. The first (and most eye-catching) when looking at files is the filename extension. Python scripts always end in `.py`. So, a script might be named `myScript.py` in order to be recognized as a valid Python script by the interpreter. However, `.py` is not the only extension used in Python. `.pyi` files contain Python information, such as PEP 484 hints (such as those for your classes). You typically won't see `.pyi` files except at the highest levels of Python programming, since most programmers opt to put their code hints inline with the code. I (sort of) lied when I said that Python was an interpreted language and that it wasn't possible to compile or choose to compile Python code. In reality, you can, but no one actually does, since it's much easier to share the original source code. However, they do have some advantages. `.pyc` files are compiled Python scripts, and they are created when a module or another Python script is imported into your Python script. The interpreter will "compile" the bytecode of the imported module so that the translation from source code to object code of this static code (you almost never edit another module, unlike your own code) only has to be done once. You can run compiled Python, but it's quite difficult and not of much use. If you see these files, just ignore them.

Python embraces 19 key principles for development in its language, which was written by Tim Peters in 1999. We'll go through most of these principles in passing throughout this text, but if you'd like to read the entire thing (it's not actually that long), you can type the following into the interactive Python interpreter or into a Python script and execute it.

```
1 import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules
.
Although practicality beats purity.
```

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

We won't cover some of these for the rest of this book, so here's the explanation to those right now.

- **There should be one— and preferably only one —obvious way to do it.** Programming is one of those disciplines where there isn't just one way to solve a problem. However, there's often a "good" way to solve a problem and a "bad" way to solve a problem. If it feels plainly wrong, think about what you're doing.
- **Although that way may not be obvious at first unless you're Dutch.** Python's creator Guido van Rossum was Dutch, and Peters is *probably* making a tongue-in-cheek jab at Larry Wall (who isn't Dutch) for creating Perl, where there are often many "good" ways to do the same task.

Some of these these tenets of Python programming extend to other languages, while others do not.

2.1.1.3 Layout of a Simple Python Script

The general form of a simple Python script is given below. As far as the Python interpreter is concerned, the *line breaks* and *spacing* between individual keywords doesn't really matter. However, we'll get to why you should care in Chapter 4.3. This is a part of programming known as *whitespace*.

Each line contains either nothing (a blank line) or a *statement*. A statement is just an instruction that is followed by the computer. If you have programmed in a different language, you might be used to ending lines with semicolons. However, Python does not have any special character to end its lines with. Because Python is an interpreted language, it is executed line-by-line. Comparatively speaking, compiled languages execute as one long string of

statements separated by semicolons. If you try to end a line with a semicolon as you would in Java, C++ or C#, you will end up with a syntax error.

```
1  import module1
2  import module2
3
4  def function1(arg1, arg2):
5      Variable_Declarations
6
7      Statement_1
8      Statement_2
9      ...
10     Statement_Last
11
12     return something
13
14 def function2(arg1):
15     Variable_Declarations
16
17     Statement_1
18     ...
19     return somethingElse
20
21 Main_Statement_1
22 Main_Statement_2
```

The first two lines are called **import statements**, and they tell the interpreter about certain extra modules that are used in your program. This is incredibly useful, since a lot of code already exists for a lot of complicated stuff, like math functions, data science, graphics, or web scraping, so you don't have to write these from scratch. We'll see how to use modules when we begin to work with files and in data science.

You should follow your instructor's directions on where to place your **import** statements. Most instructors have you place all of your **import** statements at the very beginning of your program. This means that you don't have to question what modules you have imported without having to search for all of them, useful in extremely long scripts. It also means that you can use anything in any of the modules without having to figure out where in the script you imported the module, since modules can only be used *after* they have been imported.

Next, we have function definitions. Don't worry exactly about what a function. We'll cover the subtleties of functions in Chapter 7.1. We mark the beginning of the function using the colon and what is inside of the function using indentation.

After our functions, we have the main part of the program. This is where we can refer to functions that are in modules that we have imported or those which we created ourselves. This is also where we'll control the basic flow of

the program.

As your programs become more and more sophisticated, they will also deviate more and more from this basic structure. Advanced programs can span across different files and folders. However, we need to start somewhere, and for our purposes, dealing with single files is perfectly adequate.

2.2 The Tools

Like a builder might use a hammer, a screwdriver, and a wrench, a programmer has some tools at their disposal. We use these tools to debug our code and turn it into instructions that our computer can understand.

Aside from the computer itself, all of our tools are in the digital world, but they're very powerful. Let's take a look at some of these tools.

The first and foremost is called an IDE, or integrated development environment. That's just a fancy way of saying a place where you can write and run code that you write. `repl.it`, Google®Colaboratory, Apple®Xcode and Microsoft®Visual Studio are all IDEs. IDEs can be broken down into two fundamental components: a text editor and a compiler.

A text editor is pretty simple. It's a place where you can edit plain text. If you're on a Mac, you can use Apple®TextEdit, and if you're on PC, you can use Notepad. Don't confuse these with software packages like Microsoft Word, Google Sheets, or Apple Pages. These are actually word processors, and they store data in binary files. You can also download different text editors, like BBEdit®for Mac, Atom for Mac, PC and Linux, or Notepad++ for PC and Linux, and they all have their own strengths and weaknesses. There's also text editors that can run in the terminal, like vim, emacs, and nano.

A compiler is a little bit more complicated. We code in Python, but computers can't natively understand Python. Python would be what developers call "source code", and it's almost always human readable, but not machine readable.² When we want to run a bit of our source code, we need to translate it into something that our computers can understand. This happens in two stages. The first stage converts our source code into assembly code, instructions that the computer can then break down. Assembly code is only readable to the trained eye, and is difficult to think about intuitively. The second stage breaks code down to machine code, or the 1's and 0's that the computer can understand, but that isn't human-readable.

All of that seems a little bit abstract still, so let's look at how that code looks in real life, if we were to open the files as a plain text file.

First, let's examine a sample Python script:

```
1 i = 0
2 while True:
3     print(i)
4     i += 1
```

With a little bit of sleuthing and some Python-to-English conversion, we can interpret this script as a program that will count up from 0 to infinity (or until the computer runs out of memory) and print the result.

²There are some source code languages that aren't human readable, like BrainF, but these are very much exceptions. These languages typically exist only as novelties.

Now, let's look a look at what the assembly code for this will be. It's sloppy, but the trained eye might be able to break this down.

```
1      i: DB 0
2      .loop:
3          MOV i, eax
4          ADD eax, 1
5          PSH eax
6          CALL printf
7          POP eax
8          JMP .loop
```

This makes perfect sense to the computer, even if you don't understand it. (On a side note, take a computer architecture class to understand what this code is!)

Fun Fact

The game *Prince of Persia* (1989) was written entirely in assembly code, a near-superhuman feat given the amount of detail in the game!

One of the most important things to understand about computers is, they're not magic. It's not a mysterious black box, and as much as we'd like to think that they operate using magic blue smoke, that's not really the case. You can do as little or as much as you want with them, but they can do a lot for you. Have fun on your programming journeys! It will be difficult, but rewarding!

Exercise Questions

These exercise questions cover chapters 2.1 and 2.2.

Exercise 1

1. What placeholder words do Python's documentation use?
2. Does Python support emojis?
3. What version of Python is this book written for (and subsequently what you will be coding in)?
4. Are Python 2 and 3 interchangeable?

Exercise 2

1. What are some file extensions that you might find in a Python script?
2. What file extension will a regular Python script have?

3. What line of code can you run to get the Zen of Python in your Python shell?

Exercise 3

1. What is the difference between a statement and spacing?
2. Does Python execute code line-by-line or are Python statements executed separated by semicolons?
3. If your course has a style guide, where should you put your import statements according to that style guide?

Exercise 4

1. What kind of code do we, as programmers, typically write: source code, assembly code, or binary code?
2. What kind of code can a computer understand natively: source code, assembly code, or binary code?
3. What kind of code is used to create binary code: source code or assembly code?

Exercise 5

1. What is the difference between a text editor, a compiler, and an IDE?
2. Look at your course's syllabus. Are you using a text editor, a compiler, an IDE, or some combination of these? If it is a combination, what is the combination?

Exercise 6

(No wrong answers!)

1. Why are you choosing to learn how to program?
2. How familiar are you with computers in general (regardless of programming experience)?
3. Do you know any other programming languages? If so, which ones?

2.3 Hardware and Software

2.3.1 Hardware

If you've become captured by the world of computer hardware, it's easy to get dazzled by the latest and greatest processors from companies like Intel, Advanced Micro Devices (AMD), Nvidia, Qualcomm, or Apple. These processors are designed on the cutting edge of silicon manufacturing, and to mere mortals, they seem impossible to understand. But, the heart of every computer (aside from quantum computers - that's a whole other beast), the same basic mechanics are hard at work.

So, what does a computer actually do? Well, here's your 50,000-foot overview. There are entire courses dedicated to the inner workings of computer, such as computer architecture or system design courses, but this should give you a very basic understanding of how a processor works. We'll only cover the basic central processing unit (CPU) and a little portion of the random access memory (RAM) in this text. But what are those things? The CPU is responsible for, well, processing. It takes instructions from programs that you or someone else has written and it simply executes the instructions in that program. Likewise, the function of RAM is very simple: it's to store data that the CPU is working on.

At its very core, the CPU carries out something called the fetch-execute (sometimes also referred to as the fetch-decode-execute) cycle. To run this cycle, we need four basic components in our processor: a clock, a counter, an instruction register, and an accumulator. We'll also need direct access to our memory, that RAM from above.

This fetch-execute cycle is run every time the clock tells it to. When you see processors referred to by their clock speed, this is that number, measured in Hertz. Modern processors run at billions of times per second, or in gigahertz, but for now, we're going to slow it way down to one clock tick per sentence (or so). With each tick of this clock, the processor is going to do one of three things: fetch new instructions from the memory, decode the instructions that it just fetched, or execute the instructions that it just decoded. The instructions and the results from any evaluations are stored in memory. Each chunk of data, whether it's an instruction or some data chunk, is allotted its own **memory space** within the random access memory. This memory space, in theory, is not allowed to be touched by any other process.³

Memory is where the instructions and data that the processor is working on (or is about to work on) is stored. There are different types of memory in a computer. **Primary memory** holds data that the processor is working on. It is volatile, which means that if the computer loses power, the data in

³We say "in theory" here, since there is such an attack known as a buffer overflow attack, where a process can overwrite memory that it shouldn't have access to with malicious code. There are also other attacks that can provide unauthorized access to a memory space that shouldn't be exposed.

the memory is gone. DRAM, or dynamic random access memory, is a type of primary memory. **Secondary memory** holds data that can be loaded straight into primary memory for the processor to use. It is non-volatile, which means that the data persists across sessions. If the computer loses power, the data on the memory medium is not lost. Secondary memory is traditionally thought of as "hot storage," meaning that the data on the storage medium can be accessed immediately when it is requested. Secondary memory includes hard disk drives and solid-state drives. **Tertiary memory** is storage that is stored outside of the main computer. Tertiary memory includes CDs, DVDs, external hard drives, USB flash drives, or external storage arrays. Like secondary memory, it is non-volatile. However, unlike secondary memory, if you need access to it, you need to explicitly connect it to the computer that is requesting data on it. In essence, you can't access the data on a USB flash drive without plugging it in first. Tertiary memory is traditionally thought of as "cold storage." If hot storage is analogous to a food dish under a heat lamp ready to eat, cold storage's analogy is the food in the freezer that needs to be prepared before it can be served.

As humans, we use **human interface devices**, or **HIDs**. HIDs include input and output devices. An **input device** is simply any device that allows a user to communicate information to the computer. These include simple electronic switches, like buttons, all the way up to complex input devices, like keyboards, mice, or webcams. Not all input devices are HIDs, though. Environmental sensors, like humidity sensors or thermistors (a resistor that predictably changes resistance via thermal energy), are also input devices, but these are not HIDs.

There are also output devices. These are most commonly thought of as monitors, but monitors aren't the only type of output device. Seven-figure displays, lights, or relays (electronically controlled switches) can all be output devices.

Input and output devices work in conjunction with each other. For example, you might have a simple computer that turns on a light when the temperature exceeds 40°C. A thermistor is the input device, and the light is the output device. On your computer, your keyboard and mouse (or trackpad) are the input devices, and the monitor is the output device.

On modern desktop computers, such as your Mac or PC, the communication between the hardware and the software is controlled by the **firmware**. Your computer has something called the **basic input/output system**, or BIOS for short. The BIOS is a set of instructions in firmware which control the input and output operations of your computer. When you press the power button on your computer, it's your BIOS's job to instruct the computer's hardware on what exactly to do. In the last decade or so, there's been a move to more extensive BIOS interfaces using the Unified Extensible Firmware Interface, or UEFI. You can use the UEFI to edit the BIOS firmware settings.

Exercise Questions

These exercise questions cover chapter 2.3.1.

Exercise 7

1. What does CPU stand for?
2. What does RAM stand for?
3. What is the CPU responsible for doing?
4. What is the RAM responsible for doing?

Exercise 8

1. What is the difference between volatile and non-volatile memory?
2. Think about a USB flash drive that you can store files on. Is this volatile or non-volatile memory? Why?

Exercise 9

1. How is RAM different from ordinary non-volatile, secondary, or tertiary storage?
2. What is a memory space?
3. How is a memory space different from RAM?
4. Can a program be run from ordinary non-volatile, secondary, or tertiary storage mediums? Why or why not?

2.3.2 Software

As you might expect, you'll be writing *software* in an introductory programming course. More advanced programming courses might have you cover *firmware*, but for your purposes, you are only concerned with software. Programmers write software using *programming languages*. However, the term "programming language" is pretty broad; different languages operate at different *levels*, with *high-level languages* being the most popular in recent years. High-level languages are the most akin to human languages, and they are designed to be easy for humans to write and understand. Comparatively speaking, *low-level languages* are much closer to what a computer can understand. Python, C, C++, C#, Java, Swift, and Objective-C are all high-level languages. Assembly language (that really tough stuff to read from chapter 2.1) or machine code (the 1's and 0's themselves that a computer processor can directly interpret) is considered a low-level language.

However, computers can only read binary: the 1's and 0's that make up a software program. So, how can we convert the command `print('Hello, World!')` into a bunch of 1's and 0's that our processor can execute? We have to convert the code into something that the computer can read. There are two main

schools for converting code into machine code: *interpretation* and *compilation*. Both code interpreters and code compilers operate do the same thing (convert high-level language code into machine code), but how and when they make that conversion differs greatly. Since both types of conversion begin with code and end with a different kind of code, we need some new terms to refer to what kind of code is what. **Source code** is the high-level language code that you write in Python, C, C++, C#, Swift, or any other common programming language. After the conversion is made, the computer will produce **object code**, which is directly executable by the computer. You can open a source code file in a text editor, but you cannot natively execute it. Conversely, you can execute an object code file, but you cannot natively open it in a text editor. The decision to make the code an interpreted language or a compiled language happens when the language itself is being designed; you cannot choose which you'd like.

Interpreted languages are translated into object code while the actual code is being executed. As the program is being run, the computer is interpreting the code, line-by-line. This means that you don't have to wait for the code to be translated into your executable binary file, but it also incurs a performance penalty on the code's execution, since each line needs to be interpreted as it is needed. Interpretation, while perhaps insignificant on a small project, could add up to seconds or even minutes on a large project.

Conversely, **compiled languages** are translated into object code before any code is actually executed. Before your program can even run, it needs to be converted into a binary executable (**object code**) that can then be run. When you're ready to run your code, your computer will compile the code, then run the compiled code, not the source code. It means that it takes longer to get the compiled code (as the entire source code file or files are compiled), but that once the code is compiled, it's much faster than an interpreted language, since the computer's not trying to compile the instruction and execute the instruction, too, only execute the instruction.

Python is an interpreted language. For your purposes, the execution time between an interpreted and a compiled language do not matter nearly as much as writing good code in the first place. Both will seem nearly instantaneous to mere mortal humans like you or me.

2.4 How to Program

2.4.1 A New Type of Problem

The challenge of programming computers tends to be a mental block. Programming computers comes very easily to some people and is quite difficult for others. No matter who you are, programming teaches you how to solve problems a *different* way. This new way isn't necessarily a *better* way to solve problems, though. The method that you learn on how to solve programming challenges will probably be different from someone else's, so it's tough to say exactly what new things you'll learn on exactly how you solve problems, but you will learn: about programming, about how you think, about problem solving.

There seems to be a misconception that programmers know everything about the languages that they program in, but that's just not true. Programmers tend to know the most common syntax for the tasks that they do with the programming language that they use, but it's just unrealistic to assume that every programmer knows every single module. You'll probably end up having to do some research in the Python documentation, as well as any of the modules that you choose to use. Even experienced programmers have to consult documentation, so there's certainly no shame in doing so yourself. The documentation has been made available by the language designers to help you, and you should use it.

Besides, there's also Internet forums like StackOverflow. A word of warning: these forums are notoriously harsh, and the experienced users seem to believe that they're right and everyone else's solution is dead wrong. We wouldn't recommend trying to ask a question on StackOverflow unless you need a good dose of shame in your life, but you can certainly try to find help from answers to questions that others have already asked. Whatever you do, don't just copy and paste solutions that you find from the Internet. Not only will you not learn the basic syntax of how to program, but it may constitute an academic integrity infraction. Plus, stealing code just isn't cool.

2.4.2 The Power of Pseudocode

When most newbie programmers start a project, they choose to jump right into the code and start the actual act of programming straight away. This is not a good idea for many reasons, only one of which is that it lets you write very, very bad code.

Instead, you should be ready to write *pseudocode*. Pseudocode, as its name suggests, is code-like writing that doesn't follow all of the same syntax as real Python. After you write your pseudocode, you can "translate" it to Python or whatever programming language that you are using.

Pseudocode can take several different forms: fake code, napkin drawings, flowcharts, diagramming, and so many other forms. There's no specific syntax that you have to follow when you write your pseudocode. You could write the

same Python script in many different ways in pseudocode. For example, here's a Python script. Don't worry, you don't have to understand what this code does now.

```
1 name = None
2 name = str(input("What is your name?"))
3 print("Your name is", name)
```

This might not all make sense to you now, but here's a way to write this as pseudocode.

```
Empty variable called name
Fill name with user input as a string
Output "Your name is" and the name var
```

Here's another way.

```
variable name as Nonetype
output "What is your name"
name < user input
output concatenated str "Your name is" + name
```

This is only one type of pseudocode. Remember that pseudocode can also be flowcharts and diagrams. When your projects get more complicated, you should be comfortable with writing your algorithms out on paper. It will help you understand the nuances of your algorithms, and you can then convert that into code much easier.

Chapter 3

Basic Datatypes

Everything in programming circulates around data. Whether we want to store, print, evaluate, manipulate, or do something else to a chunk of data, we need to be able to express that data in some way.

Python provides us with several ways to describe data. If you have experience with other programming languages, this will be very familiar. Before we even get to storing or manipulating data, we need to know how Python will treat our data.

In this chapter, we'll go over the basic ***datatypes*** that exist in Python 3. Many of these datatypes exist in other languages, too. However, we won't see a lot of code here, as we're only going over the datatypes themselves.

A datatype, as its name would suggest, is the type of data that is being referred to. Let's say you wanted people to fill in a form. Some of the fields on that form would be alphabetical letters, like a name or email address. Other fields would be numbers, like age, height, or weight. Yet others would be binary options, like whether someone has taken a course before, where the only options are *yes* and *no*. Letters, numbers, and binary options are all datatypes, and in programming they have formal names.

3.1 Strong and Weak Typing

Variable typing (we'll get to what a variable is later) is an important part of programming. Some languages are strongly typed, while others are weakly typed. **Strongly typed** languages require their author to specify what kind of data is being stored or manipulated, and the type of data that is being stored in a specific memory space cannot be changed. The reason for this has to do with the size allotted for the memory space. More detailed data requires more space to store. If the author wants to change the datatype down the line, the language typically (but not always) requires the author to explicitly **typecast** the data. Typecasting is the process of changing the datatype that some chunk of data is described as. We'll examine how the allotment for each of these data types changes when we look at the difference between integers and floats.

Conversely, **weakly typed** languages do not require their author to specify what kind of data is being stored. Most languages will attempt to store the data in the smallest datatype possible and can typecast on the fly, meaning that the author can simply imply that the data should be of a different type and the language will adapt to that new language.

Python is a weakly typed language. As you'll see in later sections, it's possible to explicitly typecast some data (especially when you're trying to output that data), but you can also typecast on the fly when addressing data in defined memory locations.

Weakly typed languages, despite their name, sound like a pipe dream! They allow the programmer to just program without having to worry about how that data is being stored! Woah, slow down there, bucko! Not so fast: weakly typed languages have some serious implications for both the programmers who use them and for the program's users. Experienced programmers really like it when things are explicit. Remember, we control everything that the computer does, but sometimes, what the computer does is unexpected, and a shocking number of issues stem from mis-cast data if the programmer didn't know what kind of data they were dealing with. Plus, while it seems like a hassle to explicitly define your datatypes, any student who's taken an introductory programming course can tell you that knowing your datatypes becomes like knowing the back of your hand - it becomes second nature! So, it's worth committing the different datatypes to memory, as you'll use them a lot!

Exercise Questions

These exercise questions cover chapter 3.1.

Exercise 10

1. What are some of the advantages of a strongly typed language?
2. What are some of the disadvantages of a strongly typed language?
3. What are some of the advantages of a weakly typed language?

4. What are some of the disadvantages of a weakly typed language?

Exercise 11

1. What is the difference between strongly and weakly typed languages?
2. Can you typecast data with weakly typed languages?
3. What kind of language is Python: strongly or weakly typed?

3.2 Booleans

The most primitive data in every single programming language is the *Boolean* datatype. A Boolean represents either a true or a false. These values are stored in memory as a 1 or a 0. As such, it only requires one bit to store an entire Boolean.¹ This makes Booleans extremely fast to work with. It doesn't take as many processor cycles to work with a Boolean variable as it does to work with more complex datatypes.

In Python, the state of a Boolean is case-sensitive. When you write out a Boolean value in Python, the first letter is always capitalized. So, you would write Boolean values like this:

```
1 while (True):  
2     ...
```

```
1 variable = False
```

If you try to write a Boolean value with all lowercase letters, Python cannot evaluate your program, and it will crash.

Python requires the first letter of all Boolean values to be capitalized, but this is not exclusive to Python, nor are other languages syntactically bound like this. For example, Java, C++, and C# use all lowercase letters, `true` or `false`, while PHP uses all uppercase letters, `TRUE` or `FALSE`.² You don't need to get bogged down with how each language deals with Boolean states, only the language that you're actively working with. However, if you do read code in other languages, you shouldn't assume that it's wrong just because a Boolean value isn't capitalized. That might be the correct syntax for that language, and you should always check that language's documentation.

¹Technically speaking, it takes more than just one bit to store a Boolean value, since you also need to track the name of what thing that Boolean is telling the state of. However, the actual Boolean value itself only takes one bit to store.

²PHP is a very funky language, and while the official documentation states you should use all-caps, it's not technically required. There are many, many faults and issues with PHP, but it's not a topic for this book.

Dates are Bools?

"Why...why...why...if activeSpray...if not actively spraying and the duration isn't equal to zero, spray duration is equal to the duration input..."

"Is it because spray start time is a Boolean instead of a..."

claps "Lesson learned here, if you want to store time, don't use a Boolean!"

- William Osman (October 2018)^a

^a<https://youtu.be/JwU4VwZLGL4?t=437>

There are several ways that the term *Boolean* gets used in programming. In general, it can be used in three different ways: as a statement to set a variable value:

```
1 active = False
```

In a comparison expression (in the below expression, `laps < 10` would evaluate to `True` or `False`):

```
1 while (laps < 10):
2     ...
```

Or as a control mechanism:

```
1 while (True):
2     ...
```

You don't need to understand the distinction between these three uses, especially since we haven't covered control logic or variables yet, but you should be aware that you might see the term *Boolean* used in several different ways.

Programmers sometimes refer to Boolean values as "bools." If you hear the term "bool," it's referring to a Boolean value.

Exercise Questions

These exercise questions cover chapter 3.2.

Exercise 12

1. What states can a Boolean hold?
2. How does Python mandate that you write out these states in code?
3. At a very basic level, how is a `True` value stored in memory? Only consider the value itself; ignore the memory address.

Exercise 13

1. Do some research: How are Booleans written out in Matlab code?
2. Do some research: How are Booleans written out in JavaScript code?
3. Do some research: How are Booleans written out in C++ code?
4. Do some research: How are Booleans written out in Swift code?

3.3 Numbers

Booleans alone are very difficult to work with, especially if you want to store anything other than true or false. Let's be honest, as great and as simple as Booleans are, they're not the right tool for a lot of jobs. Next, we're going to cover two ways to represent numbers in Python.

3.3.1 Integers

Think back to your algebra class. Remember when you had to classify counting numbers from integers and integers from decimals? No? That's okay, let's jog your memory.

An integer is simply a whole number, including zero and negative numbers. 5 is an integer, and so are 13, 0, and -51. When writing integers in Python, always remember to include the number and the number only. No symbols (other than the negative symbol -) are accepted in an integer value. Trying to put another character in will result in a syntax error.

Integers in Python have a limited size. Since they need to be stored in a defined and limited memory space, they are assigned 32 bits. This equates to a range of -2147483648 to 2147483647. This is really, really big, but sometimes, it isn't big enough, so Python has another secret type called the `long` integer. Long integers have an infinite capacity. Since Python is a weakly typed language, it can detect if your integer is too large for a standard 32-bit integer space and automatically upgrade the datatype to a long integer.³ Neat!

Note that some languages have the ability to convert a numerical value into an unsigned value, meaning that the range stays the same, but instead of the range being from $-x$ to x , the range is from x to $2x$. Python 3, due to its unlimited long integer size, does not support unsigned numerical values.

What's that L doing?

Sometimes, you might see an integer printed with a L at the end of it in the environment viewer of your IDE, like 3018L. L stands for long, but functionally, it's the same as an integer in Python 3. 3018L really just means the integer in long form 3018.

Sometimes, programmers refer to integers as "ints." If you hear the term "int," it's referring to an integer.

³Python's method of dealing with values too large for the datatype can handle is only one way of handling this. Other languages may require you to explicitly typecast to a different variable type altogether, such as a double, that has the capacity for the data that the programmer is trying to store.

Different integers?

Depending on the version of the Python interpreter and what is returning the type, calling for the type of integer variable might give you "int32" or "int64" instead of just "int." Python will automatically typecast from int32 to int64 if your value is too large, so for your purposes, you only need to know that both int32 and int64 represent integer values.

Exercise Questions

These exercise questions cover chapter 3.3.1.

Exercise 14

1. What states can an integer hold?
2. Give three examples of integers.
3. How can you refer to an integer in speech, when talking with other programmers?

Exercise 15

1. Does Python support unsigned integers? If it does, why does it support unsigned integers? If not, why doesn't it need to support unsigned integers?
2. Do some research: What is a programming language that supports unsigned integers?

Exercise 16

1. What is the difference between a regular integer and a long integer?
2. Do you need to worry about whether a value is an integer or a long integer in Python? Why or why not?

3.3.2 Floats

But what if we need to use decimal points? This would be handy for many reasons, such as storing a money value, a precise mass, or many other things. An integer doesn't support decimal points, so instead, we can use a *float* or a *floating-point value*. In strongly-typed languages, floats are typically stored in 64-bits instead of 8- or 16-bits, meaning that if used to store whole numbers, they support much larger ranges, while still being able to hold decimal values.

In this spirit, Python 3 uses the term *float* to mean any number that isn't an integer, that would ordinarily be "too large" for an integer.

Like integers, floats can be negative. So, 3.5 is a float, but so is 78.38323526 and -635.5465. If a number has a decimal point, Python will automatically assume that it should store the value as a float.

At this point, it begs the question, why should we use integers at all? Well, floats take dramatically more memory to hold. For a more concrete example, check out this demonstration.⁴ The top line represents a big loop with an integer, and the bottom line represents the same loop, but with the execution time penalty of a floating point value. The faster line is taking less time, computationally, to evaluate the next position of compared to the slower line.

Exercise Questions

These exercise questions cover chapters 3.3.1 and 3.3.2.

Exercise 17

1. What states can a float hold?
2. Give three examples of floats.
3. How is a float different from an integer?

Exercise 18

1. Why shouldn't we use floats for everything?
2. Provide three examples where we should use floats over integers.
3. Provide three examples where we should use integers over floats.

⁴<https://processing.org/examples/integersfloats.html>

3.4 Strings

What if we want to represent anything other than a true/false value or a number, like text? Well, we have one last datatype at our disposal: the string. Why do we call it a "string?" Imagine that you had alphabet soup and you picked out the "alphabet" part. If you got a piece of twine and put your alphabet on the twine in a particular order, you are representing some information as a string of letters. In computer science terms, we refer to individual letters as *chars* or *characters* and to specific combinations of chars as *strings*.

In Python, strings are represented using quotation marks or single quotes. Python doesn't care whether you use either, as long as you're consistent. In Python, the following lines are syntactically valid strings:

```
1 "Aunt Jacky went up the hill."  
2 'Aunt Jacky went up the hill.'
```

However, the following lines is not syntactically correct, since the opening and closing quotes don't exactly match for each string:

```
1 "Aunt Jacky went up the hill.'  
2 'Aunt Jacky went up the hill."
```

You can always typecast an integer or float into a string, but you can't always typecast a string into an integer or a float. We'll get into exactly how typecasting works in Python in chapter 4.6. For now, let's just cover the basics of what can and cannot be typecast.

Let's say that you calculated the sum of two integers using Python and you now want to display this result to your user. You need to be able to display this newly created integer, but the `print()` function only accepts strings. Instead, we need to typecast our integer into a string. We can't do any arithmetic (covered in chapter 4.8) on a string, but we can print it.

Likewise, it's possible that we'll need to accept some user input. However, all of our standard mechanisms for accepting user input in the console only accepts strings, not only numbers. But what if we want our user to enter the price of something? Instead, we can accept our input as a string, then typecast it to a float. Again, we'll see exactly how to do this in chapter 4.6.

You may see the term *string literal*. This means a string that is literally spelled out, such as the ones that are described above. This is in contrast to dynamically created strings, such as those that are generated directly by Python code. If you write single or double quotes, then you are creating a string literal.

It's also worth mentioning that some word processors create something called "smart quotes," which angle the quotation mark based on where they fall in relation to the words around them (for clean opening and closing quotes). Smart quotes are NOT compatible with Python code, so if you paste in code from Word documents or Google Docs documents, be careful that your quotation marks are not smart quotes!

Exercise Questions

These exercise questions cover chapters 3.4.

Exercise 19

1. What states can a string hold?
2. Give three examples of strings, syntactically marked up for Python (include any symbols that denote a string).
3. Where does the term "string" come from?

Exercise 20

1. Can a string hold data that could also be stored by a float? Justify your answer.
2. Can a float hold data that could also be stored by a string? Justify your answer.
3. Why wouldn't we just store everything as a string in Python? Explain your answer.

Evaluation Copy

Chapter 4

General Python Programming

4.1 Basic Operations

4.1.1 Printing to the Console

A console application is one that operates entirely in the terminal. By itself, it has no graphical user interface, or GUI. GUIs can be built on-top of console applications, and the console can be hidden, but even in this case, in the background, the console is hard at work logging the events of your program. Try it! If you're on a Mac, open the Console app. It looks complicated, but at its base, it's very simple. It's just logging messages that are being sent to it by the system. Windows and Linux don't traditionally have anything similar, since both operate on a process model (individual applications are responsible for tracking their console messages), rather than a sandbox model (the host tracks console messages from all different applications).

The most basic function of Python is being able to print to the console. For the most part, when we're working with Python, we're going to be working with console applications, so it's vitally important that we know how to track the progress of our programs by using the `print()` function.

In most Python IDEs, there's a dedicated console for your program. This is where all of the input and output happens in a console program, and for all intents and purposes, it's your only way to interact with your code.¹ So, how do we print to the console? It's really simple, actually! The `print()` function allows us to print whatever is inside of the parentheses. When you write something inside of the parentheses, this is called *passing in an argument*. We'll get to what this exactly means when we talk about functions in chapter

¹Okay, technically, you could also interact with console applications using physical interfaces, like sensors and motors, but we won't cover these here.

7.1.

`print()` only requires one argument: what you actually want to print. It looks like this:

```
1 print("Hello, World!")  
  
1 Hello, World!
```

In the above example, observe how because we're printing a string, we need to use quotes. However, as noted in chapter 3.4, we could also write the code as follows, with single quotes:

```
1 print('Hello, World!')  
  
1 Hello, World!
```

The print function can accept multiple arguments, or things inside of the parentheses. We pass in multiple arguments by writing them inside of the parentheses, and we distinguish between these arguments using commas. The most common reason to pass in multiple arguments is to concatenate two or more things. **Concatenation** is the act of linking two or more things together in a chain. Consider the following example:

```
1 print('Hello, ', 'World!')  
  
1 Hello, World!
```

This would print the exact same thing as the previous two examples in the terminal. The only difference is that they're separated by a comma. When concatenating two elements together, Python's default behavior is to add an extra space, which is why the above example would print the exact same result as the first two examples. In fact, if we tried to put an extra space inside of our strings that we were passing in, we'd end up with a double-space.

```
1 print('Hello, ', 'World!')  
  
1 Hello,  World!
```

It's subtle, but in the output, there's an extra space between the comma: `Hello, World!` versus `Hello, __World!`.

We can change this default behavior by adding in another argument: `sep=""`. The `sep` argument specifies how each element that is to be concatenated should be separated, and it should equate to a string. This string can be empty, or it can be full of stuff. Consider the following examples:

```
1 print('Hello, ', 'World!', sep=" ")
```

```
1 Hello,World!
```

```
1 print('Hello,', 'World!', sep="CATS RULE")
```

```
1 Hello,CATS RULEWorld!
```

There's one more common argument to pass into the `print()` function: `end=""`. The `end` argument specifies what the end of the line should look like, and it brings up another topic: *escape characters* or *escape sequences*.

Every language needs some way to represent non-printing characters, such as new lines, carriage returns, or tabs. Most languages use a standard based off of the backslash character, including Python.

Escape Sequence	Meaning
<code>\n</code>	New line
<code>\r</code>	carriage Return
<code>\t</code>	Horizontal Tab
<code>\v</code>	Vertical Tab
<code>\'</code>	Single Quote
<code>\"</code>	Double Quote
<code>\\</code>	Backslash Literal

Escape characters allow you to print special characters that otherwise couldn't be represented. While these aren't all of the escape characters available in Python, these are the most commonly used ones.

ANSI Escape Characters

These escape characters are shared between nearly every programming language, and they are standardized by the American National Standards Institute, or ANSI.

Escape characters aren't recognized everywhere

In most plaintext strings, escape characters are recognized. However, in a parsed setting, your escape characters might not be escaped, or the parser might even recognize the escape characters as escape characters and deliberately pass them through! For example, the `\n` escape sequence doesn't exist in HTML, so trying to use the `\n` escape sequence in HTML will print a literal `\n`. Always double-check your work when using escape characters.

The `\n` escape character will add a newline character to wherever it is placed in the string literal. It is commonly used in conjunction with the carriage return character, or `\r`, since not all operating systems respect either `\n` or `\r`, but putting both in requires all operating systems to do either one.

```
1 print("Hello,\n\rWorld!")
```

```
1 Hello,
2 World!
```

The `\t` and `\v` characters add a horizontal and a vertical tab, respectively. It is used to vertically or horizontally align text in the console, just as you would in a word processor document. If you're not familiar with how tabs work, you can think of the screen as a grid, where each of the cells is several characters wide. A horizontal tab character will move everything after the tab character to the next cell, and it will continue to stack new cells.

```
1 print("Hello,\tWorld!")
```

```
1 Hello,   World!
```

```
1 print("Hello,\vWorld!")
```

```
1 Hello,
2     World!
```

You can also escape quotes inside of quotes by using the backslash, or the escape character. Using the escape character will force Python to evaluate the string literally, rather than trying to parse anything that comes after the backslash.

```
1 print("Hello, \"World!\")
```

```
1 Hello, "World!"
```

But what if we need to print an actual backslash? Well, then we need to escape our escape character. We can use double backslashes `\\`, and here's what it means: the first backslash does what it always does and tells the interpreter that we need to prepare to evaluate an escape character; the second backslash tells the interpreter that we're actually trying to output a backslash.

```
1 print("\\")
```

```
1 \
```

Now, we can use our escape characters to customize the end behavior of a Python print function. Just like `sep`, we can pass in `end` as an argument.

```
1 print("Hello, World!", end="\n")
2 print("Hello, Python!", end="")
```

```
1 Hello, World!
2 Hello, Python!
```

The ANSI standard specifies that a stray backslash (just a `\` by itself) or an undefined escape sequence (say, `\z`, which isn't a valid escape sequence) have undefined behavior. This means that depending on which interpreter you are using (and even which version of the same interpreter you are using), your interpreter could do what its authors found convenient. This means that two different interpreters will do two different things when presented with an undefined or stray escape sequence. Instead, you should only use the escape sequences that are defined by ANSI, which are provided above.

Exercise Questions

These exercise questions cover chapter 4.1.

Exercise 21

1. What is the console?
2. What is the function name to output to the console?
3. What goes inside of the parentheses?
4. Name two arguments that you could pass to the console output function.

Exercise 22

1. Write some code to print the following string on one line, without specifying the end of the line: Hello, Python
2. Write some code to print "Hello", then "Python" by concatenation, but with no space in between.

3. Write some code to print "Python" and end the line with a tab, rather than a newline and carriage return.

Exercise 23

Are the following `print()` statements syntactically correct? Why or why not?

1. `print Hello, World!`
2. `print "Hello, World!"`
3. `print(Hello, World!)`
4. `print("Hello, World!"`
5. `print("Hello, "Dottie!"")`
6. `print('Hello, World!')`
7. `print('Hello, "Dottie!"')`
8. `print('Hello, World!')`

4.2 Variables

We've alluded to storing data a lot already, but how do we actually do this? Well, we use variables. Just like in algebra, variables in Python can be represented by lots of different things, and they can represent a lot of other things.

We'll use variables to store nearly everything in Python. They're the best way to store the state of something while you're working on it, track how many times you've run through a loop (we'll get there later), store user information, and so many other things. In fact, variables are foundational to every single programming language.

All data that is worked on in variables is stored in memory when you're running a Python script. When you declare a new variable, your variable is assigned a memory address in the computer's RAM. You can do anything you want with that memory space, including filling it with data or editing the data that's inside of it. For a review on what a memory space is, refer to chapter 2.2.

Since all of your variables are stored in memory, they can only persist while the program itself is running. After your program is terminated, the memory space is marked as free by the operating system, meaning that any other program is now free to overwrite that memory.

4.2.1 Declaration and Initialization

There are two steps to creating a new variable in Python: declaration and initialization. **Declaration** means that you are telling Python that you are going to create a new variable. **Initialization** is the process of putting some initial data into that variable that can then be manipulated later. Initializing a variable doesn't necessarily mean that you cannot change the variable's value later.² Declaration and initialization are typically done on the same line of code. By using a single equal sign =, we can initialize and declare a variable in one fell swoop. Python will do its best to figure out what kind of variable it is, depending on what kind of information you give it. Consider the following code.

```
1 cashValue = 3.39
2 accountHolder = "Stinky Pete"
3 isBroke = True
4 kitties = 390
```

This code is declaring four new variables: `cashValue`, `accountHolder`, `isBroke`, and `kitties`. Each of these variables is being initialized to a different type. `cashValue` is being initialized to the value 3.39, which would be stored as a float. Likewise, `accountHolder` is being initialized to the value "Stinky Pete",

²The exception to this is if you decide to use constant values, or `consts`. You might set a constant for the value of Pi or for the force of gravity. Constants cannot be changed after initialization.

which is a string. `isBroke` is being initialized to the Boolean value `True`. Note that `kitties` is being initialized to 390, and Python will try to store this value as an integer, since there isn't a decimal point. If we wanted to force Python to store `kitties` as a float, we could just add a `.0` to the end of the initialization value:

```
1 kitties = 390.0
```

When declaring or updating the value of a variable, the name of the variable always goes to the left side of the equal sign. You can think of a variable statement as putting the thing on the right side of the equal sign into the variable on the left side of the equal sign. So, the following code is syntactically correct:

```
1 accountHolder = "Stinky Pete"
```

But the following code is not syntactically correct.

```
1 "Stinky Pete" = accountHolder # WRONG
```

In the above examples, can you guess what the variables are describing? Probably. The variable names `cashValue`, `accountHolder`, and `isBroke` explain what information the variables hold pretty well. However, the variable `kitties` doesn't really make sense. Is this how many kitties I have? Is it my love or hatred of cats on a scale from 1 to 1000? We don't really know. In programming (not just in Python), it is critically important that you get into the good habit of declaring good variable names. Be descriptive, but concise. A better variable name might be `numKitties`.

There are several rules that you **MUST** follow when making new Python variables. Failing to follow these rules will result in syntax errors:

- A variable name must start with a letter or the underscore character `_`
- A variable name cannot start with a number
- A variable name can only contain alphanumeric characters and underscores (A-Z, a-z, 0-9, and `_`)
- Variable names are case-sensitive: `age`, `Age`, and `AGE` are three different variables
- Variable names cannot be reserved words in Python, such as `print`

Now is also a good time to go over naming conventions. There are several different naming conventions that you might see while programming. It doesn't terribly matter, but whichever convention you go with, you should stick with it throughout the entire project.

- `snake_case` is written with underscores in between each word. In Python, snake case is typically used for package or modules that must be longer than one word. Avoid using snake case when possible.

- flatcase is written with no spaces and no additional capitalization. In Python, flatcase is typically used for methods and functions. However, you may find it easier to use camelcase.
- lowercaseCamelCase is written with the first word in all lowercase and all following words capitalized. In Python, lowercase camel case is typically used for variable names.
- UppercaseCamelCase is written with all of the words capitalized. In Python, uppercase camel case is typically used for method names.
- ALLCAPS is written with all letters capitalized. In Python, all caps case is typically used for constants.
- train-case is written with dashes in between each word. Python doesn't typically use train-case.
- sArCaSmCaSe Is UsEd To InDiCaTe ExTrEmE sArCaSm.³

When using camel case, capitalize all letters of abbreviations (e.g. `HTTPSServer` instead of `HttpsServer`). Always avoid naming this with `0` (easily confused with `0`) or `I` (easily confused with `1` or `L`).

It is possible to declare a new variable without initializing it, but you must declare a variable before initializing it. If you don't know what type of variable will be, but you know that you'll need a variable, you can initialize to something called the ***Nonetype***. To do so, simply set the initialization to `None`. This is the practical equivalent to declaring without initializing.⁴ Consider the following code.

```
1 emptyVar = None
2 print(type(emptyVar))
```

```
1 <class 'NoneType'>
```

In general, you should always try to initialize your variables, even if it's just to `0` (if it's an integer), `0.0` (if it's a float), or `""` (if it's a string). If it's a Boolean variable, you can initialize it to `True` or `False`, depending on what makes sense.

³The pOwEr To UsE SaRcAsMcAsE cOmEs WiTh GrEaT rEsPoNsiBiLiTy. UsE iT wIsELY.

⁴Other languages have no `Nonetype`, since they support direct declaration without initialization. For example, in C++, you could run the line `'int age;'`, which would declare the `'age'` variable without initializing it with anything.

Initialize to type variables

Initializing your variables helps you determine what datatype they should be, since Python is a weakly-typed language. As you become more experienced with Python datatypes, you should be able to instantly determine a datatype just by looking at the variable's first initialization. Therefore, it's worth learning the datatypes by heart - it'll come in handy later!

We've talked about variables, but there's a specific type of variable in Python called a constant.⁵ The difference between a regular variable and a constant is that a constant *cannot* be changed after initialization. To mark a constant, you should name it in all caps. It might be useful to use constants to store data that you know will never change, such as a conversion factor or a mathematical constant, like π or e .

```
1 PI = 3.1415
2 E = 2.71
3 GRAVITY = 9.8
```

Changing Constants

Some languages, like Python, treat constants and variables in the same way, meaning that it is possible to change the value of a constant (although you shouldn't!) Other languages essentially lock away the constant once it is declared as a constant. In these languages, after the first constant initialization, the interpreter or compiler will throw an error or warning if you try to reinitialize a constant.

Exercise Questions

These exercise questions cover chapter 4.2.

Exercise 24

1. When you create a variable, where is it stored: primary, secondary, or tertiary memory?
2. What are the two steps for creating a variable?

⁵Yes, in math, variables and constants are different concepts, but in Python, a constant is a subset of the variable.

3. Can you do those two steps in one line, or must they be done in two lines?
4. What character should you use to initialize a variable?

Exercise 25

1. What is the difference between a datatype and a variable?
2. What datatype can a variable be?

Exercise 26

What is a good variable name for the following pieces of data? What type of variable would Python probably initialize it to? Provide an example of how Python would store that data. For example, if you came up with a variable named `numCookies` that was an integer, then an example of the data might be 5.

1. The name of your favorite sports team
2. What home stadium your team plays at (or home city)
3. The number of wins your team got
4. Whether your team qualified for the championship playoffs or not
5. The total number of matches or games your team played
6. The height of your favorite player on that team in meters
7. The average weight of the athletes on that team in kilograms
8. The number of players on the team
9. Whether the team is sponsored or not

Exercise 27

Are the following variables syntactically correct for Python? Why or why not?

1. `pointsReceived`
2. `PointsReceived`
3. `10DayPointsReceived`
4. `_pointsReceived`
5. `_PointsReceived`

Exercise 28

1. Consider the variables `pointsReceived` and `PointsReceived`. Are these variables interchangeable? Why or why not?
2. Consider the variables `_pointsReceived` and `pointsReceived`. Are these variables interchangeable? Why or why not?

4.2.2 Referring to Variables

Now that you've created variables, it'd also be really handy to get the value of those variables down the line. Being able to refer to variables means that you can get and change their value.

The way in which you refer to an existing variable will depend on the manner in which you need to refer to it. In most cases, it is possible to simply refer to a variable by name. For example, let's declare a new variable as a string, then print the contents of that variable.

```
1 stringToPrint = "Python is cool!"
2 print(stringToPrint)
```

```
1 Python is cool!
```

Observe how we are *not* using any quotation marks inside of our print statement. This is because we are not actually printing "stringToPrint," but rather, we are printing the string literal that is stored as a value of the variable `stringToPrint`.

Recall how we created a variable and filled with something. The process of creating the variable was called the declaration, and the "filling" was called the initialization. We can also reinitialize a variable by simply overwriting the contents of the memory space that the variable uses. We need not redeclare the variable, since it already exists. Attempting to redeclare an existing variable will result in a syntax error. Consider the following code.

```
1 string1 = "Python is cool!"
2 print(string1)
3 string1 = "Python is rad!"
4 print(string1)
```

```
1 Python is cool!
2 Python is rad!
```

In this code sample, we are printing the same variable `string1` twice. However, we also observe that the output changes. This is because we're reinitializing the variable in the third line using `string1 = "Python is rad!"`. Because of this, we will get a different result when we try to print the same variable.

The same can also be said for variables that have non-string contents. For example, consider the following code.

```
1 int1 = 5
2 print(str(int1))
3 int1 = 7
4 print(str(int1))
```

```
1 5
2 7
```

Since we're changing the contents of the variable, the second print statement is simply getting the updated variable value.⁶

Exercise Questions

These exercise questions cover chapter 4.2.2.

Exercise 29

1. Define a new variable `name` and initialize it to your name.
2. Now, print "Your name is: " and concatenate it with the `name` variable in the print statement.

Exercise 30

1. Using your variable naming rules, create a new variable that would describe the name of a building on your college campus. Initialize it to the name of a building on your college campus.
2. Create a new variable that describes the discipline of that building (math, history, psychology, English, etc.) and initialize it to that value.
3. Write a print statement that will print "___ is a ___ building." Fill the first blank with the variable for your building's name, and fill the second blank with the variable for the discipline of that building. Use string concatenation to print all of the variables from the same print statement.

4.2.3 Handling User Input

It's also possible that you'll want your user to input something on the keyboard so that you can evaluate it in your program. The ability to input text is a very useful. For now, we're only going to focus on taking in text. We'll cover how to turn that text into numbers, like integers and floats, in chapter 4.6 on typecasting.

On the console, the function to take in user input is quite aptly named `input()`. `input()` must always be preceded by a variable and the variable assignment operator; it cannot stand on its own like `print()` can. Consider the following example code.

```
1 print("What is your name?", end = " ")
2 name = input()
```

⁶For now, ignore the `str()` function. This is typecasting the integer into a string so that it can be printed by the `print()` function. We'll cover typecasting in chapter 4.6.

```
3 print("Your name is ", name, ".", sep="")
```

```
1 What is your name? Stinky Pete
2 Your name is Stinky Pete.
```

When you run this code, your program will prompt you for your name. You can then type directly into the console using your keyboard. When you press Enter, the contents that you typed are put into the variable `name` as a string. The last line then prints some string literals and your name.

When using `input()`, it doesn't matter whether you input only letters or only numbers. The `input()` function will always return the input as a string. You can, however, typecast a variable as some other datatype. See chapter 4.6 on typecasting.

The `input()` function can take one argument, a string to prompt the user. You can either pass in a variable which is of type string or just use a string literal. So, the following code is logically the same as the previous example.

```
1 name = input("What is your name?")
2 print("Your name is ", name, ".", sep="")
```

```
1 What is your name? Stinky Pete
2 Your name is Stinky Pete.
```

Exercise Questions

These exercise questions cover chapters 4.2.3.

Exercise 31

1. Is it possible to use the `input()` function standalone (that is, without an variable and assignment operator)? Why or why not?
2. What happens when you try to use the `input()` function standalone? If you receive an error, what kind of error is it?

Exercise 32

1. Create a new variable `age` and initialize it to the `Nonetype`. See chapter 4.2.1 for a review on the `Nonetype`.
2. Use the `input()` function to ask for the user's age and put the result into the `age`.
3. Print the datatype of `age`. What datatype is it? Why is this so? (Hint: to print the datatype of a variable, you can print the `type(variableName)`.)

Exercise 33

1. Create a new variable `color` and initialize it to be an empty string.
2. Write an `input()` statement that prompts the user for their favorite color and puts the result in the `color` variable.
3. Write a single print statement that prints the user's favorite color and compliments their color choice.
4. Write another `input` statement that prompts the user for their least favorite color and overwrites the contents of the `color` variable.
5. Write another single print statement that prints the user's least favorite color and tells them that you agree the color looks bad.

4.3 Whitespace

Whitespace is a crucial component of every programming language, and using whitespace correctly is very important. Different languages handle whitespace differently, but you should know how to correctly format your code, regardless of which language you're writing code in.

4.3.1 Why is Whitespace Important?

Donald Knuth once said, "Programs are meant to be read by humans and only incidentally for computers to execute." As programmers, it is in your best interest to make your code readable. More readable code means that you can understand it, find problems faster, and others can catch onto your programming style quickly and understand your code swiftly. Messy code is difficult to read. Even if your instructor does not require you to or does not penalize for incorrect whitespace, it is still an important skill.

Python requires that you use the right indentation when using code blocks, such as loops or conditional logic, as you'll see in chapter 6. Other languages use curly braces {} to denote groups of code that should run when certain conditions are met. Regardless, there are some rules that you should follow when writing code, especially around comparison operators and assignment operators.

Consider the following code, which uses poor or inconsistent whitespace.

```
1 if(a>0):
2     print(a)
3 else:
4     if(a <10):
5         print(a)
6     else:
7         print("a is negative or greater than 10")
```

This code is fairly sloppy. Sure it's readable, but keep in mind that this is only a short snippet of code. If our code was hundreds of lines long, it's easy to see how small indentation errors could add up, resulting in trouble finding where the beginnings and ends of functions are, where your loops are controlled, among other things.

Instead, we could fix our code as follows.

```
1 if (a > 0):
2     print(a)
3 else:
4     if (a < 10):
5         print(a)
6     else:
7         print("a is negative or greater than 10")
```

Observe how this code uses consistent whitespace patterns for all of its indentation. One indentation is, in this case, always two spaces. Indentation doesn't always need to be two spaces. It could just as easily be four or eight spaces, or a tab character.

Regardless of which whitespace rules you choose to follow and those you choose to ignore, think about how your decisions will affect the readability of the code.

Whitespace is also vertical. You can generally put empty lines at logical breaks in the code, such as between loops, functions, or if/else statements. Consider the following code.

```
1 def function1(arg1, arg2):
2     return arg1 + arg2
3 def function2(arg1, arg2):
4     return arg1 * arg2
5 def function3(arg1, arg2):
6     return arg1 // arg2
```

You don't need to understand exactly what this code does, but while it does have good horizontal whitespace, we could improve its vertical whitespace by adding extra empty lines. Compare the previous example to the following code.

```
1 def function1(arg1, arg2):
2     return arg1 + arg2
3
4 def function2(arg1, arg2):
5     return arg1 * arg2
6
7 def function3(arg1, arg2):
8     return arg1 // arg2
```

Breaking apart each of these logical blocks by adding extra empty lines doesn't affect the functionality of the software, but it does make it a lot easier to read. It's easier to see where the block starts and where it ends.

4.3.2 Common Whitespace Patterns

There are several whitespace patterns that are used in programming, especially in Python. In general, follow the following rules.

- The assignment operator `=` should always be surrounded by one space on either side. For example, use `cats = 10` instead of `cats=10`.
- The comparison operators (`==`, `!=`, `<`, `>`, `<=`, and `>=`) should always be surrounded by one space on either side. For example, use `cats >= 10` instead of `cats>=10`.

- Sometimes, functions (both calls and definitions) should have a space between the function name and the parentheses with arguments. For example, `func (arg)` instead of `func(arg)`. This book does not use this style of notation.
- Use the same indentation standard throughout your entire project. This is non-negotiable. Which standard you use is up to you, but stay consistent. You can use 2 spaces, 4 spaces, 8 spaces, tab, 2 tabs, or something else that conveys indentation, as long as you're consistent.
- Avoid splitting individual lines into two different lines if it would disrupt the flow of the software. Instead, turn on the soft wrap functionality in your text editor.

Different computers and different text editors have different standards for how they'll treat spaces between brackets and braces and how they write tabs in, so when you change computers, you should double-check what the tab settings are for that computer, as you might need to change them.

Soft wrap is a really handy viewing mode that you can use in your text editor, and we mentioned it above. But what is it? Soft wrap allows your editor to wrap lines that are too long for one line into the next line without affecting any functionality of your source code and without changing of the line numbers. When you write in a standard word processor like Microsoft Word or Google Docs, you've actually already used soft wrap. Notice how as you type, when you approach the end of the line, your word processor makes a new line for you without you having to explicitly hit the carriage return key. The same thing can be done in your text editor or IDE, and the option is typically found under the View options as Soft wrap or Toggle Soft Wrap. In advanced text editors, there are three options: soft wrap, hard wrap, and no wrap. Most text editors only use soft wrap and no wrap. The difference between a soft wrap and a hard wrap is that soft wrapping will add your extra line without breaking any words (it breaks on spaces) whereas a hard wrap will add your extra line wherever the end of the line is (it breaks on characters).

You can play around with which whitespace rules you'd like to use and which you will ignore. The important thing is that you try to stay organized and that you are consistent throughout your entire source code file or project.

Exercise Questions

These exercise questions cover chapters 4.3.1 and 4.3.2.

Exercise 34

1. What is whitespace?
2. What directions does whitespace apply: horizontal, vertical, or horizontal *and* vertical?

3. Why is whitespace so important?
4. Does your course instructor or grader penalize for improper whitespace? If so, what is the penalty?

Exercise 35

Open your text editor or IDE. Use it to answer the following questions. You may also need to open a source code file.

1. What does the soft-wrap option do? Where is it?
2. Where is the option to change the tab setting, if you have an option to do so? What is it currently set to: 2 spaces, 4 spaces, 8 spaces, 1 tab, 2 tabs, or something else?
3. Do all text editors or IDEs use the same default indentation or wrap settings?

Exercise 36

1. What should surround the assignment operator to achieve good whitespace?
2. What should surround a comparison operator for good whitespace?
3. Should you put an extra space in between two characters in a comparison operator (`!=` versus `! =`)?
4. Is it okay to change the indentation standard or whitespace pattern in the middle of a project? When is it okay to, if ever?
5. If you had a really long line that was wider than your text editor window, what should you do?

Exercise 37

1. Teamwork is an important part of programming. Say you're working on a team with several people. One of your team members doesn't put a space between functions and arguments, while another does. What can you do rectify this situation? Should you just ignore it and let people do what they want if it will make good code? Why or why not?
2. Sometimes, you'll need to use someone else's computer, including their text editor in the way that it was set up. What are some things you might want to check before starting to work on your project on that computer?

Exercise 38

Fix the following lines of code by correcting the whitespace errors. What did you fix, and why did you choose to make that fix? If there are no errors, say so.

```
1. _____  
1  if (var != 3):  
   _____  
   _____  
2. _____  
1  print("x is the smallest")  
   _____  
3. _____  
1  def addFour (inputVal):  
   _____  
   _____  
4. _____  
1  def addFour (inputVal):  
2      # Do something  
3  def addFive (inputVal):  
4      # Do something  
   _____
```

4.4 Comments

Comments are a critical part of any program, even if they don't contribute to the actual functionality of your code. Just like how whitespace is so important, your comments are the key for you being able to understand your code. Talk to any experienced programmer, and they'll have stories of how they spent all night working on a chunk of code, then forgot what it meant or how it worked the very next day and had to rewrite the entire chunk of code.

While you don't need to comment every single print statement, commenting with enough detail that others can understand your code is a good habit to get into.

4.4.1 Line Comments

The most common type of comment is a line comment. A line comment can either take an entire line (no code is executed on that line) or it can take the latter half of a line.

The symbol for a line comment is a pound sign (hash sign or octothorpe) `#`. On a line, everything before the pound sign is evaluated and executed, and everything after the pound sign is ignored by the interpreter. Consider the following chunk of code.

```
1 print("Hello, Python!") # This is a comment, and it
   won't execute
2 # This line also won't execute anything
```

```
1 Hello, Python!
```

You can use line comments to describe a line of code as shown above. They're also handy if you're testing out different lines of code and you want to only evaluate certain lines. You can also stack pound signs. Consider the following chunk of code.

```
1 ##### Example #####
2 print("This line will execute.") # Known and working
3 #print("This line won't execute.") # Perhaps not
   working
4 # Another full line comment
```

```
1 This line will execute.
```

Instead of having to delete lines of code that might or might not work, you can just comment them out until you're absolutely sure that you don't need them. They won't evaluate or execute until the pound sign before them is removed.

4.4.2 Block Comments

If you need to comment out a large chunk of code, you can use a block comment. Block comments are handy at the beginning of your document, since they allow you to put a lot of non-code text in one place. You'll often see block comments used to describe what a file does, display a license agreement, or just to prevent a block of code from executing.

By using a block comment, you can avoid writing (and later removing) a lot of pound signs before each line that you don't want to execute. To use a block comment, simply put three quotation marks at the beginning of the section you want to comment out, then put three matching quotation marks at the end of the section you want to comment out. The quotation marks must be on their own line.

Consider the following chunk of code.

```
1 """
2 This is a block comment.
3 None of the code inside of these marks will execute.
4 This is great for writing long prose.
5 """
6 print("But this code will execute.")
```

```
1 But this code will execute.
```

Note that none of the commented lines have pound signs at the beginning of them. Since they're enclosed by the quotation marks, none of these lines will execute.

The following code is not syntactically correct. Remember that the quotation marks must be on their own line.

```
1 """This is a bad block comment.
2 This will result in an error."""
```

Block comments are a powerful tool, and they can be used to test parts of your code without having to write pound signs before every single line. For example, consider the following code.

```
1 print("This code will execute.")
2 """
3 print("But this code won't execute.")
4 print("That's good, because there's a lot of these
   lines.")
5 #print("Yay!")
6 """
7 print("So will this line of code.")
```

```
1 This code will execute.  
2 So will this line of code.
```

Also note that some IDEs and text editors might color your block comments differently from your line comments. This is nothing to be worried about.

Exercise Questions

These exercise questions cover chapters 4.4.1 and 4.4.2.

Exercise 39

1. What are some reasons to write comments?
2. How can you mark a line comment in Python?
3. How can you mark a block comment in Python?

Exercise 40

1. Are the marks for line comments the same as in Perl? If not, what is the symbol or symbol combination?
2. Are the marks for line comments the same as in C++ or C#? If not, what is the symbol or symbol combination?
3. Are the marks for block comments the same as in Swift? If not, what is the symbol or symbol combination?
4. Are the marks for block comments the same as in HTML? If not, what is the symbol or symbol combination?

Exercise 41

1. Where can a line comment be placed? Is it possible to end a line comment?
2. Where can a block comment be placed? How must the quotation marks be written out, or can they just be written anywhere to begin the comment?

Exercise 42

Using what you know about Python, write a comment for the following blocks of code. If you want to put multiple comments in, specify in between which line numbers you would put those comments (e.g., before line 1, between lines 3 and 4). If you're not sure what the code does, make your best guess (imagine that it's your code that you need to explain to someone else), but be descriptive.

```
1.   
1 result = input("What is your grade?")
```

```
2. 

---

1 print("The result is: ", end = "")  
2 print(str(result * 4))  


---

3.   
1 tax = input("What was your latest in taxes? ")  
2 deductible = input("What was your latest  
    insurance deductible rate? ")  
3 print("Here's some information we calculated: ")  
    

---


```

4.5 Errors

Errors are an inevitable part of programming, but it's a critical skill to understand how to read errors and what kind of errors you might get. Being able to understand the error messages that you get means that you'll be able to resolve them much faster.

4.5.1 Logical Versus Syntactical Errors

There are two major types of errors: *logical errors* and *syntactical errors*. You've seen the term "syntactical error" thrown around previously in this book, but what do they actually mean?

A *syntax error* is an error that results from incorrectly written code. Syntax errors can occur from things as simple as wrong indentation or mismatched quotation marks to memory leaks and segmentation faults. Conversely, a *logical error* is an error that results from valid code that doesn't do what was expected of it. Logical errors are much tougher to catch, since as the code is running, it would appear that everything's working correctly, even though there's an issue with something that you wrote.

Consider the following code. Can you spot the error? (Hint, it's in line 1.)

```
1 fahrenheit = int(input("Input a temperature in degrees
    Fahrenheit'))
2 celsius = (fahrenheit - 32) * (5 / 9)
3 print("That's ", celsius, " degrees celsius!", sep =
    "")
```

This is an example of a syntax error. If you tried to write this exact code in your IDE and run it, you'd get an error on line 1. The interpreter sees an opening quotation mark, but it can't find a matching closing quotation mark, so it throws an error instead. In this case, it's clear that there's an issue, since our interpreter throws the error:

```
SyntaxError: EOL while scanning string literal
```

Our IDE also gives us hints with the coloring of the text. In this text, strings are colored red, so the fact that we see red coloring on line 2 means that we never closed the string on line 1. There are several default error types, of which `SyntaxError` is one. Some of the other types of errors are as follows.

- `AssertionError`: raised when an assertion fails
- `AttributeError`: raised when an attribute reference or attribute assignment in a function fails; if the object doesn't support attribute references or assignments, Python will raise a `TypeError` instead
- `EOFError`: raised when the `input()` function reaches the end of a file or hits the end-of-file condition without reading any data

- **ImportError**: raised when the `import` statement has trouble trying to load a module or when you try to load a module that doesn't exist
- **ModuleNotFoundError**: raised when you try to load a module that doesn't exist
- **IndexError**: raised when a sequence subscript, like in a list or dictionary, is out of range; we will go over indices in sequences in Chapter 5
- **KeyError**: raised when a dictionary key is not found in a set of existing keys; we will go over keys in dictionaries in Chapter 5
- **KeyboardInterrupt**: not strictly an error, but raised when program execution is halted after the user hits the interrupt key (normally **Control-C** or **Delete**)
- **MemoryError**: raised when an operation runs out of memory but the situation might be able to be rescued by deleting some objects
- **NameError**: raised when a local or global name (variable) is not found
- **OSError**: raised when the host operating system (like Windows, MacOS, or Linux) encounters a system-related error in response to a Python request, including I/O failures like "file not found" or "disk full" errors
- **OverflowError**: raised when an arithmetic operation is too large to be represented
- **RecursionError**: raised when the maximum recursion depth is exceeded
- **RuntimeError**: the catch-all for when an error is encountered that doesn't fall into any other error category
- **SyntaxError**: raised when the Python parser encounters a syntax error
- **IndentationError**: raised when indentation is incorrect
- **TypeError**: raised when an operation or function is applied to an object of inappropriate type
- **ValueError**: raised when an operation or function receives an argument that has the right type but inappropriate value
- **ZeroDivisionError**: raised when trying use division or modulo with a divisor of zero

There are plenty of other errors that you could encounter, including ones that are not defined by Python itself.

Now, consider the following code. Can you spot the error? (Hint, it's in line 2.)

```
1 fahrenheit = int(input("Input a temperature in degrees  
    Fahrenheit"))  
2 celsius = (fahrenheit - 32) * (6 / 9)  
3 print("That's ", celsius, " degrees celsius!", sep = "  
    ")
```

In this case, the interpreter won't throw an error. Everything is *syntactically* correct, even though things aren't *logically* correct. This is an example of a **logical error**. The conversion factor from Fahrenheit to Celsius is not correct; the conversion factor should be 5/9, not 6/9. If we attempt to run this code, we would see that 50 degrees Fahrenheit is supposedly 12 degrees Celsius; it's supposed to be 10 degrees Celsius.

It's important to be diligent when writing code. If your code is longer, you should make sure to put in ample comments to let yourself know where things might go wrong, and don't be afraid to break out your calculator and double-check your answers.

There's also a note to be made about errors versus warnings. Errors are syntactical issues with the execution of your program. If you encounter an error when executing your code, this means that your code is syntactically incorrect, and the Python interpreter is unable to recover from your error. Encountering an error will halt the execution of your code and you'll end up with an error code. However, sometimes, we make mistakes without realizing that we've made mistakes. These mistakes don't really affect the functionality of the code, but they're also not best programming practice. In this case, Python will give you a warning. If an error is getting expelled from school, a warning is more of a slap on the wrist for misbehaving. However, you shouldn't disregard warnings as "not important," even though they aren't *technically* wrong. For one, if you have a warning, it means that something's wrong, and faulty code is never good. Just like error messages, you can interpret warning messages and fix the issues that are causing them. Never just ignore a warning message. You should at least acknowledge why that warning is occurring and deal with it in some way, even if that solution is to leave a comment indicating why the warning doesn't apply to your code.

Don't ignore warnings

We've said it before, and we'll say it again: never just ignore a warning. They could be giving you some very valuable information (including potential syntax or logical errors)!

Warning messages can occur for all sorts of reasons, including out-of-date packages, missing functions that aren't called, possible indentation errors, or many other things.

Exercise Questions

These exercise questions cover chapters 4.5.1.

Exercise 43

1. Name three different types of errors that you might receive in Python.
2. For each of the three errors that you encountered in part (1), write lines of code that will result in the error.
3. For each of the three errors that you created in part (2), give the error code that you received and what the error code tells you.

Exercise 44

Examine the following error message.

```
File "<stdin>", line 1
    input("What is your name '')
```

```
SyntaxError: EOL while scanning string literal
```

1. What type of error is this?
2. What, generally, does this error type indicate?
3. What is the error message?

4.5.2 Debugging

There are many, many types of errors that you might experience while programming, and being able to understand what those errors are is key to remedying them. Error messages contain useful information, and you shouldn't dismiss their importance.

Error codes will typically contain certain bits of information, including the file that the error is occurring in, the line number of the error, the line itself, where in the line the error is occurring, and some error code. Let's take a look at a few errors to figure out what errors look like.

Consider the following code.

```
1 # This is some code that won't run quite right
2 # Line 3 is a print statement
3 print('What\'s your name?')
```

Here's the error that this code threw.

```
File "main.py", line 3
    print('What\'s your name?')
```

```
SyntaxError: EOL while scanning string literal
```

In the first line of the error, the interpreter is offering some important information. Firstly, it's telling us that this error is occurring in the "main.py" file. This is especially important if you're working with multiple files for just one program, which can happen quite easily if you're working with classes. The interpreter is telling us that on line 3, there's an error. Next, the interpreter is copying the offending line so that we can get a glimpse of what might be causing the error. On the next line, the interpreter is trying to point out where in the line it thinks the error is coming from. Depending on the type of error, the interpreter might be close or right on the issue, and other times, it'll put the carot at the end of the line if it can't figure it out. Lastly, the interpreter will give you the error code.

These error codes can be quite difficult to decipher, but there's two parts to the error: the classification of the error and the actual error code. In this case, the error class is a `SyntaxError` and the error code is an `EOL while scanning string literal error`.

Let's look at another syntax error.

```
1 name = input('What is your name?\n')
2 print('Hi, ', userName, sep="")
```

Here's the output of this code.

```
What is your name?●
Stinky Pete
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    print('Hi, ', userName, sep="")
NameError: name 'userName' is not defined
```

Let's break down this output. The first line of the output is working as we expect it to. The `input()` is printing a prompt for our user, which is showing up. The second line of the output is what our user entered. However, on the third line, we see a `Traceback`, with the most recent call listed last. If you had a chain of errors, which typically occurs when you're using functions, classes, or modules, the root of these errors would show up so that you can trace back the issue. Next, we see the filename (main.py) and the line number that the error is occurring in. Next, the interpreter is giving us the offending line of code, but it doesn't have a clue where the error is, so it doesn't even try to point out where it thinks the error is. Lastly, it gave us the error class and the error code. The error class is `NameError`, which might occur if there's an unspecified variable or object that's being called. Sure enough, the error code indicates that `'userName'` is not defined. This code's author accidentally wrote the wrong variable name in the print statement. If we replace `userName` with `name`, the code runs properly.

In the process of programming, you'll probably run into errors that you haven't seen before, such as `TypeError`s, `IndexError`s, `ImportError`s, `KeyError`s,

or `ModuleNotFoundError`. Before you paste the error code into a search engine (really, we don't recommend this), you should take a close look at the error to figure out why it might be occurring. Chances are, it's a pretty simple fix. If you can't figure out why the error code is occurring, you can try to search keywords from the error, especially if the error is occurring because of some variable name.

You may also get something called a traceback, especially if your error has been caused by a chain of events (like a function call). Don't worry about understanding how chains of events are formed (this is called data structures, which we'll cover in Chapter 7). As its name suggests, a traceback provides breadcrumbs for you to find the origin of the error. The first breadcrumb will fall on the line of code that caused the execution error. The second breadcrumb will give you the line of code that caused the first breadcrumb's error. The third breadcrumb will give you the line of code that caused the second breadcrumb's error, and so on. Python *could* just give you the line of code that caused the exact error, but it often doesn't reveal deeper problems that will help you diagnose a problem. The error may occur anywhere between the first and last breadcrumb, but as you get better at reading a traceback error log, you'll also get better at reading exactly where an error occurred.⁷

⁷Read this footnote after you've read Chapter 7.1 on functions, since we'll give you some more information on tracebacks that rely on knowledge from that chapter. Tracebacks are commonly found when you have a function that called another function. For example, let's say you had a function called `double()` that was responsible for doubling an integer or a float. Then, later in the code, you called `double("spot")`. Obviously, "spot" is not an integer nor a float, and this will result in a syntax error. However, the act of calling `double("spot")` is not inherently syntactically invalid: the act of multiplying "spot" by two is invalid. If Python only gave us the line of code that failed, we'd only get a line saying that `double("spot")` failed, but not why it failed or why "spot" is an invalid argument. However, the traceback would give us the following information: the line `double("spot")` failed in the function `double()` because the `double()` function encountered a `TypeError`. Along with this information, the traceback will also give us line numbers for each line of code the error occurred and which file it's in.

How far to traceback?

When examining a traceback, you will almost *never* need to go beyond the code that you wrote. If your traceback goes beyond your code and into a module or library that you imported (like Pandas), your immediate suspicion should be something that you wrote. The only exception to this is if you know that there's something wrong with the library that the library developers need to fix - in this case, you'll provide the library developers with your full traceback so they can see what line of code caused the library to fail. Never give a traceback to a developer unless you are certain that the error was caused by an error in the library (out of respect for their time and effort) and they ask for the traceback to prove that whatever you ran is what caused the library to fail. Giving a traceback to a developer because your code might be wrong and blaming it on the developer isn't a good look. (It's actually pretty rude.)

Exercise Questions

These exercise questions cover chapters 4.5.1 and 4.5.2.

Exercise 45

1. Name three different types of errors that you might receive in Python.
2. For each of the three errors that you encountered in part (1), write lines of code that will result in the error.
3. For each of the three errors that you created in part (2), give the error code that you received and what the error code tells you.

Exercise 46

Examine the following error message.

```
File "<stdin>", line 1
    input("What is your name'')
    ^
SyntaxError: EOL while scanning string literal
```

1. What type of error is this?
2. What, generally, does this error type indicate?
3. What is the error message?

4.5.3 Handling Errors On the Fly

Sometimes, you'll need to deploy code where things might happen to your precious code. Someone might put in something that wasn't expected, and you need a way to handle those errors that might pop up. In fact, just as you tried to resolve errors in the previous section, you can also raise your own exceptions. Thankfully, Python has a simple way of handling these errors on the fly. This is colloquially known as try-catch.

There is a base block in Python called the `try` block. It must always be paired with the `except` block. Python will attempt to run all of the code inside of the `try` block. If it succeeds, then the code in the exception block is not run. If there is an issue in your `try` block, then Python will stop trying to execute the code in the `try` block and will instead run the code in the `except` block. For this reason, you should make doubly sure that the code in your exception block will always be syntactically correct, otherwise it will cause program termination.

In Python, we can indicate that something is inside of something else by using a colon and indentation. We can't just make anything into a parent block, but `try` and `except` are things that are.

Consider the following block of code. Assume that `x` has not been previously initialized to any value.

```
1 print(x)
```

Based on what we've already seen, this will result in a `NameError`. However, we can deal with this error by using a `try...except` set.

```
1 try:
2     print(x)
3 except:
4     print("Something went wrong.")
```

```
1 Something went wrong.
```

If we had something outside of the exception block, the program would continue to run. For example, consider the following code.

```
1 try:
2     print(x)
3 except:
4     print("Something went wrong.")
5 print("All done!")
```

```
1 Something went wrong.
2 All done!
```

Since the `print("All done!")` is outside of the exception block because of its indentation, it is run regardless of the results of the `try...except` blocks.

We can also throw different exceptions depending on the type of error that we get. For example, consider the following block of code.

```
1 try:
2     print(x)
3 except NameError:
4     print("X isn't defined!")
5 except:
6     print("Something else went wrong!")
```

X isn't defined!

We anticipated that we might get a `NameError`, so we set aside a special exception block just for `NameErrors`. We could do the same for different kinds of errors by stacking our exception blocks.

```
1 try:
2     print(x)
3 except TypeError:
4     print("X is of the wrong type!")
5 except NameError:
6     print("X isn't defined!")
7 except:
8     print("Something else went wrong!")
```

X isn't defined!

You can also throw your own exceptions. To throw your own exceptions, you can use `raise`. You can either define what kind of error to raise and what the error code is. You can also make a general exception by using the `Exception` error class.

Consider the following code.

```
1 x = "hello"
2 if not type(x) is int:
3     raise Exception("Only integers allowed")
```

File "main.py", line 3, in <module>
 raise Exception("Only integers allowed")

Exception: Only integers allowed

You don't have to understand exactly what this code does, but do pay attention to the exception. This code tests if the variable `x` is an integer. If it isn't, then it'll throw a generic exception. We can also throw a `TypeError`, since we could generally classify this as a datatype error. Consider the following code.

```
1 x = "hello"
2 if not type(x) is int:
3     raise TypeError ("Only integers allowed")
```

```
File "main.py", line 3, in <module>
    raise TypeError ("Only integers allowed")
```

```
TypeError: Only integers allowed
```

There is one final block that you can include in a try-except block in Python: **finally**. The **finally** block runs regardless of whether the code in the **try** block successfully executed or not. Most of the time, developers just skip the **finally** block, since the code that falls outside of the **except** block runs anyways. However, this can still be useful for explicitly running something.

This is massively useful if you plan on throwing a lot of errors and you'd like them to be remotely organized.

The careful and controlled use of exception handling is generally classified under graceful failure. Graceful failure is the ability for a system to fail without crashing or even self-recovering. The mode of recovery might include restarting, giving a warning, or bypassing that portion of the script.

Why is graceful failure important?

Why is graceful failure such an important tool in the developer's toolbox? Let's try something: if you're on Windows, open the Event Viewer, then go to Windows Logs, then to System Logs; if you're on macOS, open the Console; if you're on Linux, use the command `"tail -f /var/log/syslog"`.^a Then (on Windows and macOS), change the filters to show errors or faults. Now, just wait. Every entry that shows up is a place that your entire operating system could have just crashed if that error weren't handled gracefully. Thanks to the careful programming of the operating system designers, we don't have to worry about these errors crippling our ability to use our computers!

^aIf you're new to Linux, here's the breakdown of this command: the `"tail"` keyword gets the last ten lines of a file, and the `-f` flag enables the filestreaming mode so you can see when the file is written to by the system in real time. The `/var/log` directory holds most of the logs on most Linux systems, and the `syslog` file holds almost all system-related messages (except for authentication messages).

Scammers love the logs!

Tech support scammers love using the logs to convince people that their computer has serious issues. In reality, these errors have been handled gracefully by the operating system, but seeing "error" or "critical" on the screen looks scary! We now know that these *would* have been critical issues only if they were not gracefully handled. Now, you know just how innocuous these supposedly serious issues are!

Exercise Questions

These exercise questions cover chapter 4.5.3.

Exercise 47

1. What are the keywords used for a try-catch block in Python?

Exercise 48

1. If an exception is thrown, where can one get the exception message from? That is, let's say we wanted to handle the exception gracefully, but we still wanted to know what the error was before we continue. Where is the message located, and how can we view it? Provide either theory or a working example.

Exercise 49

1. What is an exception error class?
2. Provide five examples of possible exception error classes.
3. Is it possible to make your own exception error class?

4.6 Typecasting

Typecasting is an important part of dealing with data. Sure, we've talked about typecasting in previous chapters, but what actually is it, and how can we use it?

Recall the material that you learned in Chapter 3, on the basic datatypes in Python. We know how to use Booleans, integers, floats, and strings, but it'd also be really handy to be able to convert between different datatypes. This is the essence of typecasting. Some of the functionality in Python and some functions can only accept certain types of data, so typecasting allows us to have the exact type of data needed to execute these operations.

Let's look at one of the most common functions that you'll use in Python: the `input()` function. We know that `input()` can take one argument, but what does it spit out?

A little bit of sleuthing in the Python documentation reveals that `input()` always returns a string. Consider the following code. Assume that the variable `price` has not been declared or initialized already.

```
1 price = input("Input the total price of your groceries
               as a decimal:")
2 print(price)
```

What datatype would `price` be? No matter what kind of data we try to input in our program, `price` will always be a string, since that's the datatype that `input()` will always insert into the variable that it is being assigned to.

This would be great if we could always work with strings, but this just isn't the case. For example, arithmetic operations, which we'll cover in chapter 4.8, require us to use an integer or a float, not a string.

As you can probably guess, this is where typecasting comes in. We can try to typecast our input into the datatypes that we need so that we can perform our basic arithmetic operations. Consider the following code. Assume that the variable `price` has not been declared or initialized already. It's okay if you don't understand what's going on in the third line yet.

```
1 price = input("Input the total price of your groceries
               as a decimal:")
2 price = float(price)
3 tax = price * 0.08
4 print("You need an extra $", tax, " for tax, sep = "
      )
```

This program expands on our previous program by calculating our tax. It multiplies the float variable `total` by a given tax rate, 8%. However, I'd like to draw your attention to the second line:

```
1 price = float(price)
```

```
6     price = float(price)
7 except TypeError:
8     print("Not a decimal: did you add a dollar sign by
          accident?")
9 except:
10    print("Something went wrong.")
11 tax = price * 0.08
12 print("You need an extra \$", tax, " for tax, sep = ""
    )
```

We'll get to what the `while` is in chapter 6.3, but the gist of the code is, Python will declare a new variable `price` to be 0.0. As long as the price is 0.0, Python will assume that the user hasn't inputted a valid price and will keep prompting the user until they enter a valid price that can be typecast into a float. Then, it can carry on with the rest of the program.

Exercise Questions

These exercise questions cover chapter 4.6.

Exercise 50

1. What is typecasting?
2. What are the basic datatypes in Python that can be typecast?
3. Why might one want to typecast?

Exercise 51

1. What is the syntax to typecast an integer into a string?
2. What is the syntax to typecast a string into an integer?
3. What is the syntax to typecast an integer into a float?

Exercise 52

Write the applicable code to typecast the following variables into a string. If typecasting cannot be performed or is not applicable to the variable, explain why.

1. `v = 3`
2. `cats = "Tabby"`
3. `isBlue = True`
4. `r2 = 3.2209`

Exercise 53

Write the applicable code to typecast the following variables into an integer. If typecasting cannot be performed or is not applicable to the variable, explain why.

1. `v = 3`
2. `cats = 5.0`
3. `n = "three"`
4. `k = 9.003`

Exercise 54

1. Provide an example for why you would want to typecast a string into an integer.
2. Provide an example for why you would want to typecast an integer into a string.

4.7 F-strings

Beginning in Python 3.6, Python included something called a *f-string*.⁸ F-strings allow you to print two or more things, one of which as a variable, without explicitly concatenating. Essentially, f-strings create areas with placeholder objects without having to close your string quotes, typecast, or add a concatenation operator.

The fundamental parts of a f-string are the literal and the variable. The literal portion of the f-string is printed just as any other string is. As a review, this is a string literal printed.

```
1 print("The cat is orange.")
```

```
The cat is orange.
```

Traditionally, if we wanted to print the variable `color`, we would need to do this.

```
1 color = "orange"
2 print("The cat is " + color + ".", sep = "")
```

```
1 The cat is orange.
```

This should look familiar. We still have a string literal portion ("The cat is "), a variable ("orange"), and another string literal ("."). However, we could use a f-string to avoid having to close our quotes out at all. To do this, we can simply enclose our variable inside of curly braces `{}`. The curly braces tell Python that the contents of the curly brace are actually a variable, and we want Python to take that variable instead of the literal string.

The above example, as a f-string, looks like this.

```
1 color = "orange"
2 print(f'The cat is {color}.')
```

```
The cat is orange.
```

We're not actually concatenating here, so we don't need the `sep` argument to dictate what the separator value should be between concatenated objects. Instead, we can build our separation right into our f-string.

In the above f-string, take note of the following. The f-string is predicated with a `f`. This tells Python that we want to use a f-string, rather than a traditional string. Next, there is no space between the `f` and the opening quote

⁸The ability to use f-strings should be in standard Python after version 3.6, but some interpreters don't respect f-strings, instead printing literally or even erroring out because of a non-escaped special character (a curly brace). Your instructor should be able to tell you whether the interpreter that you are using supports f-strings to avoid immense frustration!

' . Putting a space between the `f` and the opening quote `'` is not syntactically correct. Finally, our variable `color` is only enclosed in curly braces inside of our f-string.

If we had a non-string value in a variable, like an integer or float, we cannot just print that value. As a review, we end up with a `TypeError`.

```
1 print("The value is " + 1)
```

```
TypeError: can only concatenate str (not "int") to str
```

To correct this, we must typecast our integer, then print the typecasted value.

```
1 print("The value is " + str(1))
```

```
The value is 1
```

In a f-string, we do not need to typecast a non-string variable. Typecasting is implicitly done at runtime to any non-string values, meaning that we can simply call the object.

```
1 value = 1
2 print(type(value))
3 print(f'The value is {value}')
```

```
<class 'int'>
The value is 1
```

Observe how we never typecasted `value`. The type of `value` is still an integer when we print it in our f-string. However, it is implicitly typecast to a string before being printed, meaning that we never need to explicitly typecast the value.

F-strings are a great way to increase the readability of your code. They decrease the amount of extra symbols, since we don't need to explicitly concatenate, typecast, or close our string to still call a variable inside of a string.

4.8 Statements and Expressions Review

So far, we've learnt of several different statements and expressions. From variable declaration and initialization to function calls, you've already learned how to use statements. Let's look at some other types of statements that you'll run into, especially as your programming skills grow.

Firstly, let's review the assignment operator `=`. The assignment operator is used to assign a value to a variable. For example, consider the following code.

```
1 cats = "tabby cat"
```

We can also use a function on the right side of an assignment operator, as long as that function produces something (return something). For example, consider the following code.

```
1 cats = input("Input your favorite cat: ")
```

However, if the function doesn't produce something, then Python will create a syntax error. The only way to prevent this from happening is to handle the exception; using exception handling, you can assign the `Nonetype` to the variable. For example, consider the following code.

```
1 cats = print("tabby cat")
```

This code will result in a syntax error, since the `print()` function can take arguments, but cannot return anything back.

We've also seen standalone statements, such as the `print()` function. Statements like these don't use any operators, such as `=`. If you try to use an assignment operator with a standalone statement, you'll end up with a syntax error, since this is not syntactically correct, according to Python.

Lastly, we've seen the block format, where certain code routines are run as part of another block of code. Most recently, we've seen this used in `try...except` blocks. To denote that a code routine should run as a part of another line of code, like `try` or `except`, we should use an indentation. The exact style of indentation isn't terribly important, so you could use two spaces, four spaces, a tab, or two tabs, but according to our whitespace rules, it's important to be consistent with whichever style we choose to use.

For example, this chunk of code will run a `try...except`. The underscores represent spaces.⁹

```
1 # DO NOT RUN
2 try:
3     __price = input("Input a price: ")
4     __price = float(price)
```

⁹This code is not technically correct: the underscore might cause some interpreters to misinterpret the purpose of the underscore. Avoid using more than one underscore unless you have a very good reason to do so.

```
5 except:
6     __print("That's not a valid price.")
```

This chunk of code will also run the same code, even though it uses more spaces. The important thing is that we're consistent with how many spaces we choose to use.

```
1 # DO NOT RUN
2 try:
3     ____price = input("Input a price: ")
4     ____price = float(price)
5 except:
6     ____print("That's not a valid price.")
```

Again, assume that the underscores represent spaces. This application of indentation will become more apparent and necessary in later chapters, such as when we move onto loops and control structures.

4.9 Arithmetic Operations

Now that we know how to get numbers into Python, it'd be really handy to learn what we can do with these numbers.

Python allows us to perform basic arithmetic operations, as the title of this chapter might suggest. Python provides seven primitive arithmetic operations:

Operation	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
//	Integer Division
%	Modulo

You've probably seen at least four of these operations in elementary school, but perhaps you haven't seen all of them, and that's okay. In Python, we need to write our code line-by-line, so it would be difficult to write things like exponents on a single line without confusing characters with other commonly used reserved characters. This is why we need to use these seemingly arbitrary symbols or symbol combinations.

Firstly, let's discuss where you might use these symbols. In programming (not just in Python), arithmetic operations are typically run next to an assignment operator =, so that their result can be assigned to a variable. Consider the following code.

```
1 seven = 2 + 5
```

Observe how the variable name is on the left side and the value is on the right side. Recall how the assignment operator must always be used in this manner, with the variable name on the left and the value to be assigned to that variable on the right.

This is especially important in programming, since we do things that mathematicians might become very confused by. Consider the following:

$$x = 2 + 4$$

This is a mathematically correct equation. We're taking some value on the right side and putting it into the variable x . Now, consider the following in a mathematical context:

$$x = x + 2$$

This doesn't make much sense, and in fact, it is wrong mathematically. There is no case where x and $x + 2$ can be the same. However, this is perfectly okay in Python. Consider the following in a programming context:

```
1 x = 5
2 x = x + 2
```

In this case, we're initializing the value of the variable `x` to be 5, then we're adding 2 to the value of `x`. Even though this is not *mathematically* correct, this is correct in *programming*. However, it's worth noting that the following is *not* syntactically correct:

```
1 x = 5
2 x + 2 = x # WRONG
```

As we saw in chapter 4.2, our variable needs to be the only thing on the left side. Again, this might be mathematically correct, but it's incorrect in Python.

```
1 2 + 4 = x # WRONG
```

is not the same as

$$2 + 4 = x.$$

Now, let's talk about the arithmetic symbols. First, the `+` symbol is for addition. As you saw above, you need to put one thing to the left and one thing to the right of the addition operator. However, you could also stack multiple addition operators in Python. For example, consider the following.

```
1 x = 2 + 3 + 4
2 print(x)
```

9

In this case, Python will evaluate your expression from left to right. So, in the above example, the interpreter would add `2 + 3`, then add 4 to the result.

Likewise, we can use `-` to subtract. You need to put one thing to the left and one thing to the right of the subtraction operator, just like in addition. If you second number is larger than your first number, the result will be stored as a negative number.

```
1 y = 5 - 7
2 print(y)
```

-2

The `*` operator multiplies the first number by the second number. Python will respect the polarity of numbers, just as you were taught in math class.

```
1 z = -3 * -5
2 print(z)
```

15

How can Python know the difference between a negative number and the subtraction operator? For one, the negation operator must come with something on the left and the right, whereas a negative number can only have something on the right (nothing on the left). Secondly, negative numbers must be written with no space between the negation symbol and the number itself as shown. Even with our whitespace rules, this is considered common practice.

The last of our common operators is the division operator, or /. The division operator can take two elements, one before and one after. If the number that results from the division is not an integer, the result will be stored as a floating point number.

```
1 a = 15 / 5
2 print(a)
3 b = 16 / 5
4 print(b)
```

```
3
3.2
```

That covers the last of the four common operators. However, there are some operators that are very useful for programmers. One of these operators is the *exponentiation operator*. This operator is written as two asterisks with no space in between in Python: `**`.¹⁰ Putting a space in between will confuse your Python interpreter. The number that comes before the operator will be treated as the base and the number that comes after will be treated as the exponent. So, take a look at the following code.

```
1 c = 2 ** 4
2 print(c)
```

```
16
```

This is the same as the following, mathematically.

$$c = 2^4$$

^ Versus **

Python is one of the few languages that use `**` to indicate exponentiation. Most languages use the carot `^`. So, instead of `2**4`, we would write `2^4` instead.

¹⁰Not all languages use two asterisks. Other languages use a carot or require a separate function for exponentiation.

Another operator that's very useful is the *integer division* operator. Similar to the exponentiation operator, the integer division operator is written as two forward-slashes with no space in between in Python: `//`. Integer division is similar to regular division, but the result is only the whole number portion of the result; everything after the decimal point, including the decimal point, is dropped.

```
1 b = 16 / 2
2 print(b)
3 d = 16 // 2
4 print(d)
```

```
3.2
3
```

As its name suggests, integer division returns an integer, never a float, unlike regular division.

Quite the inverse of integer division, the last operator is the *modulo*. The modulo operator is represented with a percent sign `%`. While integer division returns the whole number portion of the division as an integer, the module returns the remainder portion of the division as an integer, without the leading decimal point.

```
1 b = 16 / 2
2 print(b)
3 d = 16 // 2
4 print(d)
5 e = 16 % 2
6 print(e)
```

```
3.2
3
2
```

It's also possible to combine different operations in one line. Python respects the order of operations, and it groups its operations in a fairly standard way, with the exception of the integer division and modulo, which aren't a part of the standard order of operations.

As a review, PEMDAS stands for Parentheses, Exponents, Multiplication, Division, Addition, and Subtraction. In Python, all parentheses are evaluated first. If there are multiple operations in a parentheses, it will evaluate them according to the order of operations, but if all of the operations fall in the same category, it will evaluate left to right. Python allows the nesting of parentheses, just like in regular math.

Order of Operations

Python respects the order of operations. Not all programming languages do, though!

Consider the following line.

```
1 print(1 + 2 * (3 + (2+2)))
```

```
15
```

In the above example, Python will evaluate the innermost parentheses first $2+2 = 4$, then the outer parentheses $3+4 = 7$. If there are multiple parentheses at the same depth, then Python evaluates by the most inner depth, from left to right.

```
1 print(1 + 2 * ((1 + 2) + (2+2)))
```

```
15
```

The above code will evaluate the $1 + 2 = 3$ first, then $2 + 2 = 4$, then use those results to evaluate $3 + 4 = 7$. Next, Python will evaluate all exponents. All exponents are evaluated from left to right.

Next, Python will evaluate multiplication, division, integer division, and modulo, from left to right. It doesn't matter which operation comes first within this class of operations, all are evaluated from left to right.

```
1 print(2 * 2 / 8 * 4)
```

```
2.0
```

In this example, Python will evaluate the first $2 * 2 = 4$, since it's the first evaluation of the expression. Next, Python will evaluate $4/8 = 0.5$, then $0.5*4 = 2.0$. Since the division result in the second step results in a float, the entire result is printed as a float.

Lastly, Python will evaluate any addition or subtraction, from left to right. Like multiplication, division, integer division, and modulo, it doesn't matter whether an addition or subtraction operator comes first. Both are evaluated at the same priority level.

```
1 print(2 + 4 - 3 + 8)
```

```
11
```

Since all of the operations in the above example fall in the same class (addition and subtraction), the entire expression is evaluated from left to right: first $2+4$, then $6-3$, then $3+8$.

If the code were as follows, then it would be evaluated differently.

```
1 print(2 + 4 * 3 + 8)
```

```
22
```

Since there's a higher priority operation (multiplication), this will be evaluated first: $4 * 3 = 12$. Then, since the remaining operations are of the same priority, they are evaluated from left to right: $2 + 12 = 14$, then $14 + 8 = 22$.

If you're ever unsure of which operation will be evaluated first, you should use a pair of parentheses to dictate which operation is run first. As mentioned above, nested parentheses are ordered by depth, with the deepest being evaluated first.

Exercise Questions

These exercise questions cover chapter 4.8.

Exercise 55

1. How many primitive arithmetic operations exist in Python?
2. Write out all of the arithmetic operations that Python offers.

Exercise 56

Consider the following code snips. Are they syntactically correct? Why or why not?

1. `x = x + 9`
2. `x = x + x + 6`
3. `x + 9 = x`

Exercise 57

1. Say you wanted to write a negative number. Do you have to do anything special to make Python understand that you want a negative number instead of a subtraction arithmetic operation?
2. Does Python respect the order of operations?
3. Write a line of Python code to prove that Python respects or does not respect the order of operations, including what the answer *should* be and the output that Python *actually* gives you.

Exercise 58

Write the applicable code to evaluate the following lines in Python. Imagine that you are typing directly into the Python interpreter as a calculator. That means there's no need to put the result into a variable.

1. Two plus two
2. Four plus three plus nine
3. Eight subtracted from ten
4. Negative seven times positive four
5. Twenty divided by four
6. Twenty divided by three, with a decimal remainder
7. Twenty divided by three, with no remainder
8. The remainder portion of eight divided by three, with no leading decimal point
9. Eight squared
10. Eight cubed
11. Eight to the eighth power
12. Eight plus three, three minus two, and four times seven, all multiplied together

Chapter 5

Complex Datatypes

It may seem strange that we aren't covering complex datatypes right next to simple datatypes, but it's for a reason: knowing how to use simple datatypes and basic Python functionality is key to understanding and utilizing these complex datatypes.

Python has four complex datatypes that are available for you to use out-of-the-box: lists, dictionaries, tuples, and sets. We will go over all four types, but we will spend most of our focus on lists and dictionaries.

It's also possible for us to make our own complex datatypes, and we'll do that when we cover classes in Chapter 7.

5.1 Lists

Let's imagine for a moment: you want to process the statistics for each rider on your favorite pro cycling teams. Sure, you could create variables for name, functional threshold power (FTP), peak power, and watts per kilogram for every rider. What might this look like?¹

```
1 ath1name = "Peter Sagan"
2 ath1ftp  = 470
3 ath1peak = 1230
4 ath1wpk  = 6.7
5 ath2name = "Caleb Ewan"
6 ath2ftp  = 471
7 ath2peak = 1903
8 ath2wpk  = 7.0
```

Here we run into an inherent problem with individual variables: it gets really messy, really fast. It'd be really easy to spend all day making individual variables for every single rider. Well, there's a better way: lists. Lists are one of the *complex datatypes* in Python, and they allow us to store multiple pieces of individual data in a single data structure.

You can think of a list as a sequence of individual pieces of data. Instead of storing our data as individual variables, we can store all of the names in one list, all of the FTP data in a different list, all of the peak power figures in a third list, and all of the watts per kilogram measurements in a fourth list. Let's look at how to create these lists and get the data out of them.

5.1.1 Creating Lists

Creating lists isn't a particularly difficult task, and the same general structure applies as when declaring and initializing a variable. Let's use the same data as above, but expand on it a little bit. For these examples, we'll use five athletes. If we were to write each of these as individual variables, we'd have 20 variables, but instead, we can store all of the data we need in just four variable.

Lists are stored in variables, just like individual values. We even use the same assignment operator `=`. The only thing that's different is the form of the data that we're putting into our variable. In Python, we create lists using square brackets `[]` to denote the start and end of the list and commas `,` to delimit each of the elements in the list. Consider the following code.²

```
1 names = ["Peter Sagan",
2         "Caleb Ewan",
3         "Mathieu Van Der Poel",
4         "Chris Froome",
```

¹Readers should note that this data is fabricated. Do not use these figures in your code.

²Again, all of the data has been fabricated.

```
5     "Mark Cavendish"]
6 ftps = [470, 471, 460, 480, 465]
7 peaks = [1230, 1903, 1653, 1403, 1109]
8 wpks = [6.7, 7.0, 6.7, 6.3, 6.2]
```

As you can see, we're adding elements to our lists just as we would assign individual values to a variable. We use the same syntax to indicate a string (double quotes) or float (decimal points).

Common pitfall: comma placement

When writing out a list of strings, it is surprisingly easy to accidentally put your commas inside of your strings instead of outside, as the former is common in English writing. If you do put a comma inside of a string, there is nothing to delimit the two strings, so Python won't be able to store the string and your program will likely crash.

If your code isn't working, you might want to double-check that you've placed your commas correctly. The following code has just one misplaced comma that will prevent the code from running properly.

```
1 names = ["Peter Sagan,"
2         "Caleb Ewan",
3         "Mathieu Van Der Poel",
4         "Chris Froome",
5         "Mark Cavendish"]
```

It's also possible to mix the datatypes that go in your lists. For example, we could also structure our data by creating one variable for each athlete, as shown here.

```
1 saganData = ["Peter Sagan", 470, 1230, 6.7]
2 ewanData = ["Caleb Ewan", 471, 1903, 7.0]
3 vanDerPoelData = ["Mathieu Van Der Poel", 460, 1653, 6
4                   .7]
4 froomeData = ["Chris Froome", 480, 1403, 6.3]
5 cavendishData = ["Mark Cavendish", 465, 1109, 6.2]
```

When you're writing your own programs, there isn't necessarily a "right" and a "wrong" way to represent your data, but you should consider what kind of problem you're trying to solve and construct your variables around that. There are certainly "worse" and "better" ways to represent data. If you were trying to create a program that calculated average stats for all of the athletes on a team, then the first model might work better. It's a lot easier to grab all of

the elements of just one array, rather than having to iterate through each of the variables for each athlete. However, if you were trying to create a program that showed all of the data for a certain athlete, then the second model would be easier to work with. It would be a lot easier than grabbing one bit of data from a bunch of variables.

Exercise Questions

These exercise questions cover chapter 5.1.

Exercise 59

1. In your own words, describe what a list is.
2. What advantages do lists offer over simpler datatypes?
3. What drawbacks to lists have compared to their simpler counterparts?

Exercise 60

1. Provide an example of a list with all of the same datatype from earlier in this chapter.
2. For the list that you chose, what datatypes are in the list?
3. Can you put multiple datatypes into one list? If so, provide an example of this from earlier in this chapter. If not, provide the error message that Python generates.

Exercise 61

1. What characters are used to enclose a list in Python?
2. What separates each of the elements in a Python list?

Exercise 62

1. Write a Python list that has the names of five of your professors or instructors and put the list into a variable called `instructors`.
2. Write a Python list that has five breeds of dogs and put the list into a variable called `dogs`.
3. Write a Python list that has the names of four cities and put the list into a variable called `cities`.
4. Write a Python list that has the number of stories of three residence halls on your campus and put the list into a variable called `resHallHeights`. Follow the line with a comment with the name of the residence halls, but do not store the names in the actual list.

5.1.2 Accessing Data Inside of Lists

Now that we've created our list, it's probable that we'd want to access some data inside of that list.

In the past, when we stored individual elements to a variable, we could just call that variable to get its contents, such as shown here.

```
1 ath4name = "Chris Froome"  
2 print(ath4name)
```

```
Chris Froome
```

There's nothing special about the code that we just saw. But, what if we tried to print a list?

```
1 froomeData = ["Chris Froome", 480, 1403, 6.3]  
2 print(froomeData)
```

```
['Chris Froome', 480, 1403, 6.3]
```

Python very helpfully prints everything in the list. It can't read our mind and figure out that we only want the name, or the first element of the vector.³⁴ Instead, we need to instruct Python to only give us a certain *element* in the list. An element is one individual chunk of data in a list.

Why the quotes?

Observe in the above list how Chris Froome is printed surrounded by single quotes. This is because we are mixing our datatypes in a list, and Python wants to make sure that we know that this is a string. If you see quotes surrounding a list element, you should automatically consider it to be a string element. There is no such differentiation, however, between integers and floats.

³The term "list" is a term very specific to Python. In other languages, the more commonly used term is "vector." If you see the term "vector," understand it to mean what a Python list is.

⁴There is a difference between arrays and vectors, namely that arrays are assigned a specific length in memory when they are declared, while vectors can be expanded and shrunken at will. Since Python does not differentiate between arrays and vectors, we will not cover their differences in depth in this book, but you'll find more detail in a book on a language with memory management, such as C++.

Typecasting a list versus list element

Recall how we typecast variables in chapter 4.6. We can also typecast elements of a list. Simply refer to the variable that the list is in, along with the element index. Remember to not typecast the entire list if you’re only trying to typecast an element in a list.

We can tell Python to give us the n th element in a list, where n is the index of the element in the list, starting at 0. You can think of the n th element in a list as you would think of an element in a sequence in calculus. The n th element of the Fibonacci sequence F could be denoted as F_n , where n is the element of the sequence. In this case, instead of sequences, we’re looking at lists, and instead of numbers, any type of data can be represented. Let’s look at our Chris Froome list in a table, along with the index⁵ numbers.

Index	0	1	2	3
Data	"Chris Froome"	480	1403	6.3

Indices start at zero

Remember, in Python, indices start at 0, not at 1!

If we wanted to get the name, then we could ask Python to give us the 0th element of the array. In Python, we do this using square brackets, just not in the same way that we used to initialize the array. Instead, we can write the name of the variable that has the data in it, followed by square brackets with the index number of the element of the array. Don’t add an extra space in between the variable name and the square brackets with the index number.

```
1 froomeData = ["Chris Froome", 480, 1403, 6.3]
2 print(froomeData[0])

Chris Froome
```

Since we’re printing a specific element of the array, it prints without the square brackets, commas, or any quotation marks around strings.

⁵In this book, we refer to the plural of "index" as "indices." However, you may also see "indexes" written in documentation or in other textbooks.

It's very easy for beginning programmers to forget that arrays start at zero in almost every programming language.⁶ If you're having issues getting a specific element of a list, you should double-check that your index number is correct.

Just like how we've accessed an element in a list, we can also change that item's value. Changing the value of an element of a list is just like changing an individual variable. However, instead of referring to the variable as a whole, you should only refer to the specific index of data that is to be overwritten. For example, say I got Chris Froome's FTP incorrect: it should be changed to 483. I know that the index of the FTP in the `froomeData` list is at location 1, so I can get the old value easily.

```
1 print(froomeData[1])
```

```
480
```

We can also edit that value as if we were editing a regular variable.

```
1 froomeData[1] = 483
2 print(froomeData[1])
```

```
483
```

It might also be helpful to know how many elements are in the list. Python can provide this via the `len()` function, which lets us know the length of the list. `len()` takes one argument, the name of a list, and it returns an integer with the number of elements in the list. Let's say that we forgot how many athletes were in the `names` list. We could use the following to print the length of the list.

```
1 print(len(names))
```

```
5
```

The return value of the `len()` function is just a regular integer.

5.1.3 Appending to Lists

Down the line, it's possible that you'd want to append to a list. Appending an element means that you are adding that new element to the end of an existing list. In order to append to a list, your list must already exist.

Let's go back to the original example of lists. It is provided below, for your convenience.

```
1 names = ["Peter Sagan",
2         "Caleb Ewan",
```

⁶Arrays start at one in Matlab for some ungodly reason, but it is very much the exception.

```

3     "Mathieu Van Der Poel",
4     "Chris Froome",
5     "Mark Cavendish"]
6 ftps = [470, 471, 460, 480, 465]
7 peaks = [1230, 1903, 1653, 1403, 1109]
8 wpk = [6.7, 7.0, 6.7, 6.3, 6.2]

```

⁷ Using this code, imagine that we wanted to add another athlete, Tadej Pogačar. We could reinitialize all of the lists with the new data, but that's an awful lot of work for something so trivial. Thankfully, Python has a much easier way: the `append()` method. `append()` only takes one argument: the element that is to be appended to the end of the existing list. Instead of passing in the list to append to as an argument, we will instead use a *method* on the list itself. A method is a way of manipulating a data structure, like a list.

The syntax to add an element to a list is as follows.

```

1 names.append("Tadej Pogacar")
2 ftps.append(466)
3 peaks.append(1369)
4 wpk.append(6.5)

```

Since we are using a method on a data structure (the structure is a list), we will put the name of the method after the name of the data structure itself, separated by a period. We can then pass any arguments that we need to pass into to the method inside of the parentheses. Do not confuse methods with functions. They are similar, but they aren't exactly the same. We'll get to those differences in chapter 7.3.

What if we want to add multiple pieces of data to an existing list? Let's say that we had two lists, one for male and one for female athletes, but we needed to combine these lists into one. We can use the `extend()` method. `extend()` takes one argument, another list.⁸ Consider the following code, which creates two lists of names, one for males and one for females, then combines them into one list.

```

1 maleNames = ["Peter Sagan",
2             "Caleb Ewan",
3             "Mathieu Van Der Poel",
4             "Chris Froome",
5             "Mark Cavendish"]
6 femaleNames = ["Hannah Barnes",
7               "Ella Harris",
8               "Jessica Pratt",

```

⁷For reference, FTP (functional threshold power), peak power, and WPK (watts per kilogram) are real units for measuring cycling performance, even if the actual measurements are made up. They represent the power that a cyclist can put out.

⁸You can also pass in a tuple or a set, but these are topics that we won't cover in depth in this book.

```
9      "Ellen van Dijk",  
10     "Katie Hall"]  
11 allNames = maleNames  
12 allNames.extend(femaleNames)
```

We begin by creating our two lists as normal. Then, we can put our list of male names into a new list called `allNames`. We can then extend the `allNames` list using the list of female names. After this, `allNames` has all of the athlete names.

Append Versus Extend

If you attempt to use the `append` method to append a list to another list, you'll end up creating a nested list, not adding the components of the list to the other list. If you attempt to use the `extend` method to add a single item to a list, you'll end up with an error, since the `extend` method expects a list as its argument.

Just as we have appended to a list, we can also remove items from an existing array. We can do this either by name or by index number. Let's say that we wanted to remove Mark Cavendish from the `allNames` list. We know that his index number is 4 (remember, indices start at 0) and that his entry is "Mark Cavendish", and we can use either to remove the element.

If we wanted to remove by name, we can use the `remove()` method. We can pass in the exact data that we want to remove as whatever datatype the data is stored as at that list location. That is, if the data is an integer, we can pass an integer into the `remove()` method. However, we know that the entry is a string, so we'll pass in a string literal.

```
1 allNames.remove("Mark Cavendish")
```

We can also remove by index by using the `pop()` method. The `pop()` method can only take an integer, the index number of the element.

```
1 allNames.pop(4)
```

If you do not specify an argument for `pop()`, Python will remove the last element of the array.

Finally, we can clear all of the elements of a list. Clearing the elements only clears the contents; it does not remove the list itself. The list only has no content. We do this using the `clear()` method. `clear()` takes no arguments.

```
1 maleNames.clear()
```

Exercise Questions

Exercise 63

1. Where do indices start in Python lists?
2. Lists contain multiple objects. What is the formal name for these individual objects?
3. If you wanted to access the n th element of a list, what index number would you give Python in terms of n ?

Exercise 64

Consider the `instructors` list that you made in chapter 5.1.1.

1. How long is the list? Give the Python code that gave you how long the list is.
2. What is the third element of that list? Give the Python code that gave you the third element. Remember, indices start at 0 in Python, so make sure you're getting the third element, not the fourth element!
3. Add two new professors or instructors to the `instructors` list. What code did you use? Also give the new length of the list, along with the code that you used to get the length of the list.
4. Print the `instructors` list. Then, remove the third professor or instructor from the `instructors` list by the value of the element (rather than the index). What code did you use?
5. Without printing the list again, remove the element with index number 4 (not the fourth) from the `instructors` list using the index number (rather than the element value). What code did you use?

Exercise 65

Consider the `cities` list that you made in chapter 5.1.1.

1. Change the third element (careful about the what the index number is!) to "Berlin" (it's okay if you already have Berlin in your list, we'll just add it again). Provide your code.
2. Run the following in the Python interpreter: `cities[2]`. What is the output?
3. Clear all of the elements from the `cities` list, then print the list. What is the code that you used to clear the elements from the list? What is the output?
4. Now, what datatype is `cities`: list, list of strings, or Nonetype?

5.2 Dictionaries

Lists are just one way of storing multiple pieces of data. There are also dictionaries, and they operate in a fundamentally different ways compared to lists.

5.2.1 Lists Versus Dictionaries

While lists and dictionaries might appear similar on the surface, each has their own positives and negatives. The underlying concept of each data structure is different, for one. When we took a look at lists and how to get certain elements from lists, we saw that we could draw a table with our index and the data at that index location. Comparatively speaking, dictionaries use a key instead of an index. While lists store data as a sequence of data, lists are stored in key-data pairs. The keys in dictionaries are like sub-variable names, and they describe a specific piece of data within the larger dictionary.

Another way to think of dictionaries in Python is to compare them to real dictionaries with words and definitions. In a word dictionary, the words are sorted in some logical manner (alphabetically) and the accompanying data (the definition) is stored alongside the words. In Python, the words are the keys and the definitions are the data.

5.2.2 Creating Dictionaries

Just like creating a list is like declaring a variable, creating a dictionary is like creating a list.

Let's take a look at how we could represent the data in the second model of lists in chapter 5.1. The data is provided here again.

```
1 saganData = ["Peter Sagan", 470, 1230, 6.7]
2 ewanData = ["Caleb Ewan", 471, 1903, 7.0]
3 vanDerPoelData = ["Mathieu Van Der Poel", 460, 1653, 6
4                   .7]
4 froomeData = ["Chris Froome", 480, 1403, 6.3]
5 cavendishData = ["Mark Cavendish", 465, 1109, 6.2]
```

At first glance, the name makes sense, but what are the numbers next to the names? We might be able to remember, but it'd be a lot easier if everything was labeled. This is the perfect opportunity to use a dictionary. We could represent the above data as shown in this code.

```
1 saganData = {
2     "name": "Peter Sagan",
3     "ftp": 470,
4     "peak": 1230,
5     "wpk": 6.7
6 }
```

```
7 ewanData = {
8     "name": "Caleb Ewan",
9     "ftp": 471,
10    "peak": 1903,
11    "wpk": 7.0
12 }
13 vanDerPoelData = {
14     "name": "Mathieu Van Der Poel",
15     "ftp": 460,
16     "peak": 1653,
17     "wpk": 6.7
18 }
19 froomeData = {
20     "name": "Chris Froome",
21     "ftp": 480,
22     "peak": 1403,
23     "wpk": 6.3
24 }
25 cavendishData = {
26     "name": "Mark Cavendish",
27     "ftp": 465,
28     "peak": 1109,
29     "wpk": 6.2
30 }
```

To create a dictionary (or five), surround the contents of each dictionary with curly braces `{}`. Specify the key first, in quotes, as it is a string literal. Delimit the key and the data using a colon `:`, then enter the data that should occupy that key. Separate individual lines with commas `,`. It seems complicated, but with enough time, it's actually pretty simple.

As you can see, you can use the same key names in multiple different variables. However, you cannot use the same key name within one dictionary. For example, the following code is not syntactically correct, even though the values are different.

```
1 saganData = {
2     "name": "Peter Sagan",
3     "ftp": 470,
4     "peak": 1230,
5     "wpk": 6.7
6     "wpk": 6.9
7 }
```

`SyntaxError: invalid syntax`

However, we could use the following dictionary, which is syntactically correct.

```
1 saganData = {
2     "name": "Peter Sagan",
3     "ftp": 470,
4     "peak": 1230,
5     "wpk1": 6.7
6     "wpk2": 6.9
7 }
```

5.2.3 Accessing Data Inside of Dictionaries

Accessing data inside of dictionaries is remarkably similar to how we access data inside of lists. We even use the same characters!

When accessing data inside of a dictionary, we still use the square brackets `[]` that we used to access data from lists, but what goes inside of those brackets is different. In lists, we put the index number of the element of the array. However, we don't have an index number in a dictionary. Instead, we can use the element's key to get its value. Since we're using a string literal, we need to put the key's name inside of quotes. Consider the following code.

```
1 froomeData = {
2     "name": "Chris Froome",
3     "ftp": 480,
4     "peak": 1403,
5     "wpk": 6.3
6 }
7 print(froomeData["ftp"])
```

```
480
```

If you've forgotten what you named your keys, you can also pull up a list of keys for any given dictionary by using the `keys()` method, as shown below.

```
1 print(froomeData.keys())
```

```
dict_keys(['name', 'ftp', 'peak', 'wpk'])
```

Python will return the name of all of the keys as a list.

5.2.4 Appending to Dictionaries

Appending to dictionaries is an inevitable part of programming as you become more familiar with the tools of the language. You'll never be able to store all the data in your variable initialization, so we'll need to know how to append.

Unlike lists, we don't have a simple method to append a new key to a dictionary, like `append()`. Instead, we can append items by simply defining a

new key and data pair for the dictionary. Let's say that we wanted to add an updated FTP value for Chris Froome in a key called `updatedFTP`.

```
1 froomeData = {
2     "name": "Chris Froome",
3     "ftp": 480,
4     "peak": 1403,
5     "wpk": 6.3
6 }
7 froomeData["updatedFTP"] = 485
8 print(froomeData.keys())
```

```
dict_keys(['name', 'ftp', 'peak', 'wpk', 'updatedFTP'])
```

As we can see, there is a new key named `updatedFTP`. We can then use this key as we would any other.

However, dictionaries are not method-less. In order to modify existing data, we can use the `update()` method. `update()` takes one argument, a dictionary with new data. Let's say that we wanted to change Chris Froome's FTP, since it was wrong; instead of 480, it should be 483. We can use `update()` to pass in a dictionary specifying the key that should be changed and the value that it should be changed to.

```
1 froomeData = {
2     "name": "Chris Froome",
3     "ftp": 480,
4     "peak": 1403,
5     "wpk": 6.3
6 }
7 froomeData.update({"ftp": 483})
```

The brackets and braces might look confusing, but break it down and it becomes quite simple. The outer parentheses `()` are for the `update()` method, and they specify arguments for that method. The curly braces `{}` are for the dictionary, and they specify that the data being passed in is a dictionary and what the contents of that dictionary are.

Of note is that if you attempt to run the `update()` method on a dictionary that doesn't already have that key, Python will create the key and populate it with the data that you specify. You can then use this key as if you had created it by using the first method of appending to a dictionary.

Exercise Questions

These exercise questions cover chapter 5.2

Exercise 66

1. What is a dictionary?
2. What are the differences between Python lists and dictionaries?
3. Lists store data in `[?]-[?]` pairs, whereas dictionaries store data in `[?]-[?]` pairs. (Fill in the blanks)
4. Suppose you wanted to store the names of all of the NHL teams. Would you use a list or a dictionary? Why?

Exercise 67

1. Consider a potential dictionary named `falcons` that would hold team data from the Atlanta Falcons NFL team. What are some key names that this dictionary might have?
2. Consider a potential dictionary that would hold information on one of the dorms on your campus. What might you name the dictionary? What are some key names that this dictionary might have?

Exercise 68

1. Create a new dictionary inside of a variable named `eagles`. Add four keys: `code`, `city`, `wins`, and `winrate`. Initialize all of the keys to a `Nonetype`.
2. In the `eagles` dictionary, change the `code` value to a string `PHI`.
3. In the `eagles` dictionary, change the `city` value to a string `Philadelphia`.
4. In the `eagles` dictionary, change the `wins` value to an integer `4`.
5. The Eagles played 16 games, of which they won four, tied one, and lost eleven. Use Python to calculate the win percentage. Win percentage is calculated using the following formula: $(2 \times \text{wins} + \text{ties}) / (2 \times \text{total games played}) \times 100$. Put the value in the `eagles` dictionary in the `winrate` key as a float. Try to do this in a single line.

5.3 Tuples

Tuples are most often used to pass data between different data structures, which we'll cover in Chapter 7. You may have heard of tuples in precalculus, calculus, linear algebra, or some other mathematics course, where the definition of a tuple is a sequence of n elements, where n is some non-negative integer. Tuples, by definition, are ordered (given that they are a sequence), and they must be finite.

For example, $(2, 3, 4, 5, 6)$ is a valid mathematical tuple. It has a $n = 5$, which is greater than zero and is finite. The size of this tuple is immutable - this is a 5-tuple, and it will always be a 5-tuple.

Tuple Notation

If you haven't worked with tuples before, n -tuple notation is used to give the size of the tuple. A 5-tuple means that the tuple has 5 elements in it.

Tuples are very similar in Python. Like a mathematical tuple, a tuple must have a size greater than 0 and must be finite. Unlike mathematical tuples, tuples can have different datatypes in them. Like in a list, you can mix booleans, integers, floats, and strings in a single tuple. You can also refer to tuple elements by their index number using square brackets, just like in lists. However, you cannot make a tuple larger or smaller after you create its size. If you create a 3-tuple, your tuple size is limited to 3. Tuples are also immutable. Once you create a tuple, you cannot change its value.

Consider the following tuple.

```
1 leafygreens = ("Romaine Lettuce", "Iceberg Lettuce", "
  Arugula")
```

We could refer to the second element (iceberg lettuce) by its index number 1.

Indices start at zero

Indices start at 0 in Python!

```
1 print(leafygreens[1])
```

```
Iceberg Lettuce
```

However, we cannot redefine the second element to be Butterhead Lettuce.

```
1 # NOT CORRECT
2 leafygreens[1] = "Butterhead Lettuce"
```

If I want to change an element in the tuple, I need to reinitialize the entire tuple variable.

```
1 leafygreens = ("Romaine Lettuce", "Butterhead Lettuce", "Arugula")
2 print(leafygreens[1])
```

```
1 Butterhead Lettuce
```

If we want to store data, it is almost always more practical to use a list or a dictionary. However, there are still some cases where we would want to use a tuple: when we are interacting with functions. When we want to call a function and pass specified arguments to that function, we should use a tuple, since a tuple is of a fixed length and is immutable. In this use case, we will almost never see the tuple placed into a variable, but rather given as a tuple literal. Because of their limited use case, you probably won't want to use a tuple unless you see it in library documentation for some module that you're using. We wanted to introduce the concept of tuples to you regardless, since they do exist.

5.4 Sets

Sets are the simplest of the complex datatypes, so if you understand lists, dictionaries, and tuples, you already understand a set.

In mathematics, sets are simply a collection of elements. There are no orders, sequences, or indices. Sets also cannot contain duplicate entries. Any duplicate entries will be ignored, since in a set, all that matters is whether an element is present in the set or not. This means that the set 2, 3, 4 and 2, 3, 4, 3 are the same, since the values 2, 3, and 4 are in both sets. In mathematics, we write sets inside of curly braces: 2, 3, 4, 5, 6.

Chapter 6

Conditional Logic and Loops

Being able to use conditional logic and applying that logic to control the flow of your programs is an important part of programs. If there's something that computers are really good at doing, it's doing things procedurally, and loops and conditional logic are the first step to this procedural approach to computing.

6.1 Comparison Operators and Boolean Expressions

In order to understand how to use conditional logic, we need to first understand how to test conditions. The very basics of condition testing is understanding how to use comparison operators. You’ve used comparison operators before in math. You know, the $=$, $<$, $>$, \leq , \geq and \neq symbols?

The symbols in computer science are fairly standard, but they’re different than mathematical symbols. For one, the equal sign $=$ is already reserved for the assignment operator, and for two, we don’t even have half of these keys on our keyboards! Instead, programmers use programming comparison operators. The mathematical equivalents and descriptions are provided below.

Operator	Math Equivalent	Description
<code>==</code>	$=$	Equal to
<code>!=</code>	\neq	Not equal to
<code><</code>	$<$	Less than
<code>></code>	$>$	Greater than
<code><=</code>	\leq	Less than or equal to
<code>>=</code>	\geq	Greater than or equal to

We can use these symbols to test Boolean expressions in our conditional logic statements.

Variables Versus Expressions

Do not confuse Boolean variables with Boolean expressions. They are not the same thing. A Boolean variable holds a value of type Boolean. A Boolean expression tests a condition and evaluates to true or false.

Using our comparison operators, we can test conditions in Python. Before we get to the actual `if` statements, it’s important that we understand how to use these comparison operators and how they evaluate.

In Python, we typically use comparison operators to compare either two variables, a variable and a literal, a function result and a literal, or a function result and a variable. Let’s start with the most basic (but slightly contrived) example: comparing two literals. Consider the following Boolean expression.

```
1 4 > 3
```

This expression can either evaluate to **True** or **False**. To manually evaluate how the Boolean expression will evaluate, we can ask the question: is this statement

correct? Only consider whether the full statement is correct, not a part of the statement.

In the above example, we can ask the question: is 4 greater than 3? Yes, it is! Therefore, the expression will evaluate to **True**.

Here's another example.

```
1 5 != 6
```

We know that the `!=` is the not-equal-to comparison operator, so our question should be: is 5 not equal to 6? Yes, so the expression will evaluate to **True**. Here's one that might be confusing.

```
1 5 != 5
```

Again, we're using the not-equal-to comparison operator, so the question should be: is 5 not equal to 5? No, so the expression will evaluate to **False**.

If you want to test multiple things at once, you can also do that in Python. There are two main ways to test multiple things in programming: "and" and "or". Using "and" requires that all of the Boolean expressions evaluate to **True** in order for the entire compound Boolean expression to evaluate to **True**. The "or" only requires that one of the Boolean expressions evaluates to **True** in order for the entire compound Boolean expression to evaluate to **True**.

In Python, we can denote the "and" by writing `and`. Similarly, we can denote the "or" by writing `or`. When writing a compound Boolean expression, you can stack multiple single Boolean expressions on top of each other, as shown here.¹

```
1 5 != 6 and 6 != 7
```

The statement shown would evaluate to **True**, since both of the single Boolean expressions would evaluate to **True**: yes, 5 is not equal to 6, and 6 is not equal to 7. You can also use more than two Boolean expressions.

```
1 5 != 6 and 7 != 7 and 8 == 8
```

In this case, the compound Boolean expression would evaluate to **False**, since the second condition `7 != 7` evaluates to **False**, so the entire expression evaluates to **False**.

It is also possible to combine `and` and `or`. Consider the following expression.

```
1 5 != 6 and 7 != 7 or 5 != 6 and 2 != 3
```

If we evaluate the first half of the compound Boolean expression, we can see that this would evaluate to **False**. However, the second half of the Boolean expression would evaluate to **True**, since both of the conditions are **True**. The

¹Python is relatively special in that it uses the keywords "and" and "or". Most other languages use the symbols `&&` (two ampersands) to mean "and" and `||` (two vertical pipes) to mean "or."

two sides of the expression are separated by a `or`, so only one side needs to evaluate to `True` for the entire thing to evaluate to `True`. Essentially, when you use `and`, only one condition needs to be false for the whole thing to evaluate to `False`. When you use `or`, only one condition needs to be true for the whole thing to evaluate to `True`.

Note that some languages have an additional operator called the strict equality operator, represented by three equal signs `===`. In a strict equality, not only must the values that are being compared match, but so must the datatype.

In JavaScript, another programming language that has the strict equality operator, you can see how the strict equality can change the result of the Boolean expression.²

```
1 0 == '0' // TRUE
2 0 === '0' // FALSE
3 0 == '' // TRUE
4 0 === '0' // FALSE
```

Because the datatypes don't match exactly (the first item is an integer, the second is a string), JavaScript is evaluating a strict equality to be false, even though loose equalities might evaluate to true (in the first line, JavaScript is actually typecasting the string into an integer in order to test it).

While this seems useful, Python does not support the strict equality. The closest thing to this in Python is a combination of Boolean expression tests:

```
1 a == b and type(a) == type(b)
```

Exercise Questions

These exercise questions cover chapter 6.1.

Exercise 69

1. What is a comparison operator?
2. What can a Boolean expression evaluate to?
3. List the six comparison operators in Python and what they indicate.

Exercise 70

What do the following Boolean expressions equate to in Python?

1. `4 > 3`
2. `4 >= 4`
3. `4 > 4`
4. `"Python" == "Is Awesome"`
5. `4 < 3`

²In Javascript, double forward-slashes indicate a comment, not integer division.

6. `"4" == 4`
7. `int("4") == 4`
8. `4 != 4`
9. `4 != 5`

Exercise 71

Write the following Boolean expressions with the correct Python syntax.

1. 4 is equal to 8
2. 4 is not equal to 9
3. The variable `k` is less than 3 or greater than or equal to 5
4. The variable `m` is greater than 3 and less than 20
5. The variable `p` is greater than 3 or less than 0 but is not exactly 1

6.2 If, Else, and Else If Statements

It's one thing to just run lines of code sequentially, but one of the biggest strengths of computers is being able to make decisions based on empirical data. Using conditional logic and Boolean expressions, computers can run different pieces of code depending on whether some condition is met or not. This is the basis of the `if`, `else`, and `elif` (else if) tools in Python.

The most primitive type of conditional logic is the humble `if` statement. The `if` statement allows you to test a condition and execute a chunk of code based on the outcome of that test. Being comfortable with how `if` statements work will make learning how to use loops much easier, as the control mechanism in loops is essentially another `if` statement.

Recall how we used indentation in chapter 4.5.3. In that chapter, we used colons and indentation to indicate which portions of code were a part of which block. The same structure is used for all of the conditional logic and loop blocks, too. Let's take a look at some code.

```
1 a = 5
2 b = 7
3 if (a < b):
4     print ("a is less than b")
```

```
1 a is less than b
```

Let's break down this chunk of code. In the first two lines, we're defining and initializing two integer variables, `a` and `b`. Then, we're creating an `if` statement. We can read this `if` statement as meaning "if `a` is less than `b`, then execute this chunk of code." In an `if` statement, Python will evaluate the Boolean expression inside of the parentheses. If the Boolean expression evaluates to `True`, then the code inside of the `if` statement is run. If the Boolean expression evaluates to `False`, then the code inside of the `if` statement will not run at all. Instead, Python will skip the contents of the `if` statement and resume running the code that is no longer indented.

We can see how this works here.

```
1 c = 7
2 d = 6
3 if (c < d):
4     print ("c is less than d")
5 print ("This is outside of the if statement")
```

```
1 This is outside of the if statement
```

How can we tell that the code outside of the `if` statement will run? Notice how the second print statement (`print("This is...")`) is not indented at the same level as the first print statement (`print("a is less...")`).

If the Boolean expression evaluates to `True`, then the code inside of the `if` statement will run, then Python will resume any code outside of the `if` statement. Take a look at the following code.

```
1 a = 5
2 b = 7
3 if (a < b):
4     print ("a is less than b")
5 print ("This is outside of the if statement")
```

```
1 a is less than b
2 This is outside of the if statement
```

The `if` statement can only dictate whether the code inside of the statement can run. It cannot affect anything outside of the `if` statement.

We can also tell Python to test a condition, then run a certain chunk of code if the condition evaluates to `True` or another chunk of code if the condition evaluates to `False`. In Python, we do this using the `if` and `else` statements. As their names suggest, `if` will test the condition, and like we've seen, we must provide it with a Boolean expression to test. The `else` statement will handle the code that's run if the `if` statement evaluates to `False`. Unlike the `if` statement, the `else` statement cannot take anything else. Let's take a look at some code.

```
1 a = 5
2 b = 7
3 if (a < b):
4     print ("a is less than b")
5 else:
6     print ("b is less than a")
```

```
1 a is less than b
```

```
1 c = 7
2 d = 6
3 if (c < d):
4     print ("c is less than d")
5 else:
6     print ("d is less than c")
```

```
1 d is less than c
```

In the first example, the Boolean expression in the `if` statement evaluates to `True`, so the code inside of the `if` statement is executed. The code inside of the `else` statement is ignored. Similarly, in the second example, the Boolean

expression in the `if` statement evaluates to `False`, so the code inside of the `if` statement is ignored and the code inside of the `else` statement is executed.

But, what if we want to test more than just one condition with only two possible execution cases? Python has one additional tool for this exact case: the `elif` statement. `elif` is short for "else if," and it can go in between the `if` statement and the `else` statement. Like the `if` statement, you need to provide a Boolean expression to be evaluated as part of the `elif`. Take a look at the following chunk of code.

```
1 x = 3
2 y = 2
3 z = 5
4 if (z < x and z < y):
5     print ("z is the smallest")
6 elif (y < x and y < z):
7     print ("y is the smallest")
8 else:
9     print ("x is the smallest")
```

```
1 y is the smallest
```

This code looks complicated, but let's break it down. First, we're declaring three variables, `x`, `y`, and `z`. Each of these variables is being initialized to a unique integer value. Next, we're testing if `z` is the smallest of the three variables by comparing it to both `x` and `y`. If it is the smallest, then we can print that it's the smallest and exit the group of `if/else` statements. If it's not, then we can test a second condition: if `y` is the smallest. If it is, then we can print that it's the smallest and exit the group of `if/else` statements. Otherwise, `x` must be the smallest variable.

If we wanted to test more than just two, we can add multiple `elif` statements. In fact, we can use as many `elif` statements as we need to test all of the conditions that we need. Take a look at the following code, which works similarly to the previous example, but compares four numbers.

```
1 x = 3
2 y = 2
3 a = 1
4 z = 5
5 if (z < x and z < y and z < a):
6     print ("z is the smallest")
7 elif (y < x and y < a and y < z):
8     print ("y is the smallest")
9 elif (a < x and a < y and a < z):
10    print ("a is the smallest")
11 else:
12    print ("z is the smallest")
```

```
1 a is the smallest
```

In this example, we're using two `elif` statements inside of our `if/else` statement group. Because these `elif` statements are written in between the `if` and `else` statements, they will be evaluated within this `if/else` statement group. In fact, it is syntactically illegal to have an `elif` statement outside of the `if/else` statement group. The following code is not syntactically valid.

```
1 # WRONG
2 x = 3
3 y = 2
4 z = 5
5 if (z < x and z < y):
6     print ("z is the smallest")
7 else:
8     print ("x is the smallest")
9 elif (y < x and y < z):
10    print ("y is the smallest")
```

```
1 SyntaxError: invalid syntax
```

Remember that within your `if/else` statement group, the `else` statement must be the last thing that's introduced into the group. You can think of it as the "catch all" in the group. You can have as many `elif` statements as you need, but your `else` statement must be the last thing.

We can also test whether an element is present inside of a complex datatype as covered in Chapter 5. We can directly test whether an element is in a list, tuple, or set.

Math Minded?

Mathematically, we can represent this using the set membership symbol \in . If we were to ask whether i is in the set 2, 3, 4, we could represent this as $i \in 2, 3, 4$ mathematically or "i in 2, 3, 4" in Python.

Exercise Questions

These exercise questions cover chapter 6.2.

Exercise 72

1. What are the three new keywords discussed in this section?

2. How does Python tell what is inside of a block?
3. What does Python use to determine whether a block should be run?
4. Where must an `elif` block be placed?

Exercise 73

1. Create an `if` statement that executes the code inside of it if the `numDogs` variable is greater than or equal to 5.
2. Create an `if` statement that executes the code inside of it if the `cats` variable is exactly equivalent to the string `"orange tabby"`.
3. Create an `if` statement that executes the code inside of it if the `isDone` variable is exactly equivalent to `False`.

Exercise 74

For this exercise, consider the `if` statement that tests whether the `numDogs` variable is greater than or equal to 5.

1. Inside of the initial `if` statement, create a print statement that outputs the following string: `We have more than five dogs!`
2. Create an `elif` statement that tests whether the `numDogs` variable is less than or equal to 3. Inside of this `elif` statement, create a print statement that outputs the following string: `We have less than three dogs..`
3. Create a final `else` statement that outputs the following string and explain why this would execute in context of the other `if` and `elif` statements: `We have exactly four dogs..`

Exercise 75

For this exercise, consider the `if` statement that tests whether the `cats` variable is exactly equivalent to the string `"orange tabby"`.

1. Inside of the initial `if` statement, create a print statement that outputs the following string: `The tabby cat is orange!`
2. Create an `elif` statement that tests whether the `cats` variable is exactly equivalent to the string `"grey tabby"` and if so, outputs the following string: `The tabby cat is grey!`
3. Create an `elif` statement that tests whether the `cats` variable is exactly equivalent to the string `"brown tabby"` and if so, outputs the following string: `The tabby cat is brown!`
4. Create a final `else` statement that outputs the following string and explains why this would execute in context of the other `if` and `elif` statements: `"The tabby cat is something else"`.

Exercise 76

Write a short script that asks the user to input a number from 1 to 100. Test whether that number is between 1 and 100, then tell the user whether the number is greater than 50, or less than or equal to 50.

Exercise 77

Cast your mind back to when we covered arithmetic. Write the Boolean expressions to evaluate the following. Write some code to prove your Boolean expressions, and for each, test the following values for *i*, *j*, and *k*: 39, 43, 44, 50.

1. Whether the variable *i* is divisible by 5.
2. Whether the variable *j* is a multiple of 4.
3. Whether the variable *k* is a divisor of 3.

6.3 For Loops

Another important component of programming procedurally is the **for** loop. A **for** loop allows you execute a chunk of code a defined number of times.

A **for** loop will execute exactly as many times as you specify it should run. **for** loops operate through iteration, typically through numbers, though they can also iterate through complex data structures, like lists or dictionaries. Whether you pass in an integer literal or a variable, the principle between a **for** loop is the same. When writing a **for** loop, you need to provide three things: a starting point, an ending point, and the amount to iterate by for each run of the loop.

The simplest way to get started with **for** loops is with integer literal values. In Python, we need to introduce another function: the **range()** function. **range()** can take one, two, or three arguments, and each will provide additional functionality; it returns a list of numbers.

Let's take a look at what **range()** can produce. If we pass it only one argument, Python will assume that the list should start at 0 and be as long as you specified in that argument. That is, the list will span in intervals of 1 from $[0, n - 1]$. Remember that counting starts at zero in Python!

```
1 range(5)
```

This range function would produce the list $[0, 1, 2, 3, 4]$. If we pass in two arguments, then we can specify the beginning and the end of the list. Counting will begin exactly at the first number and end at one less than the second number that you specify, just like if you had only specified one argument, except if the first argument was 0. In fact, when you only pass one argument, Python assumes that the first argument *is* 0! Take a look at the following code.

```
1 range(2, 6)
```

This range function would produce the list $[2, 3, 4, 5]$. Notice how there are as many elements in the array as $b - a$, where b is the second argument and a is the first element.

If you do not provide a third argument, the second argument must be larger than the first argument, since the iteration direction goes up. But, what if we wanted to produce a list that counted down? Well, we could provide a third argument, the iterator. The iterator will tell Python how large the steps should be and in which direction the steps should go while generating the list. The third argument can be a positive or negative integer.

```
1 range(2, 6, 2)
```

This range function would produce the list $[2, 4]$. Notice how the range of the list is the exact same as in the previous range example with two arguments. However, since we've specified that the third argument as 2, Python will only use every other element from the previous array. If we had specified the third

element as 3, Python would only use every third element from the previous array.

If your third argument is positive, then your second element must be greater than your first element. If your third argument is negative, then your second element must be less than your first element. Consider the following code and the lists that they would produce.

```
1 range(6, 2, -1)
```

This would produce the list [6, 5, 4, 3].

```
1 range(6, 2, -2)
```

This would produce the list [6, 4].

Now that we can use the `range()` function, we can apply it to our `for` loops. `for` loops always contain two keywords (`for` and `in` and two additional components (an iterator variable and something to iterate through). The general form of a `for` loop is as follows:

```
1 for variable in list:
2     # code to be run
```

Let's break down this `for` loop. The first thing in this statement is the keyword: `for` lets Python know that we want to use a `for` loop. The next thing is a variable declaration. We need this variable to keep track of how many times we've run the loop. For this, just name the variable name that you want to use. Python will declare a new variable named whatever you specified, so it must follow all of our variable naming rules. The type of the variable depends on the last element inside of your loop definition, which we'll get to later. Next is another keyword that lets us know that the variable that we just declared will be iterating inside of something else. The last thing is the list that we'll be iterating through. You can also use the `variable` inside of the loop. Take a look at the following loop:

```
1 for i in range(5):
2     print(i)
```

```
1 0
2 1
3 2
4 3
5 4
```

In this loop, we can see that we're using `i` as the variable that we're using to iterate through this loop. We can then generate a list using our `range()`, and according to the `range()` function documentation, we know that the list that we're using is [0, 1, 2, 3, 4]. Since we're using the `range()` function, the variable type of `i` (our iterator variable) will be an `int`. Lastly, inside of the `for` loop, we are printing the iterator variable with each iteration of the loop.

This is the most common way to run a loop a specified number of times. This is really useful if you know exactly how many times you'll need to run a loop. For example, say you're writing a program that calculates the average of four numerical grades from 0 to 100. While you could write four `input()` statements that put the values that the user inputs into a specific variable, we could also use a `for` loop. This would allow us to put the value that the user inputs into a list of our naming. We could do this as shown in the following sample code.

```
1 grades = [0, 0, 0, 0] # declare a new variable of type
   list
2 for i in range(4):
3     grades[i] = int(
4         input("Input grade number " + str(i + 1) + " of
               4: ")
5     )
6 average = (grades[0] + grades[1] + grades[2] + grades
           [3]) / 4
7 print("Average is ", str(average))
```

Again, we can break this code down line-by-line. In line 1 (`grades = [0...]`), we're declaring a new variable as a list with four integers in it. Next, we're creating a `for` loop that will iterate through another list of integers `[0, 1, 2, 3]`, which has been created by our `range()` function. The current loop iteration number is stored in the variable `i`, which is an integer, as decided by the list from the `range()` function. Reminder: the iteration variable (like `i`) will take on whatever datatype the iteration list is. Inside of our `for` loop, we're using the `input()` function to take in four scores. We know that the `input()` function will always return a string, so we're also typecasting the value of the user input into an integer before storing it in the `grades` list. You'll also notice in the `input()` function that we're asking the user for score number `i + 1`, instead of `i`. Again, remember that indices in Python start at 0, not 1. If we asked for `i`, we'd be asking the user for score 0, 1, 2, and 3, instead of the more reasonable request of 1, 2, 3, and 4. Finally, we're using some arithmetic to calculate the average of the scores that our user gave us, storing it in the new variable `average`, and printing `average`. Because our division operation would almost certainly cause us to end up with a decimal, `average` will probably be a float. If the average just happens to be a whole number, then Python will use the integer datatype.

So far, we've only used the `range()` function to produce a list of integers that would be iterated over in our `for` loop. However, notice how we've always referred to the result of the `range()` function as a *list*. So, can we also just pass in a list as our iterator for a `for` loop? Yes, we can! Python allows us to iterate through a list, whether we use a list literal or a variable that contains a list. We also mentioned above that the iterator variable would take on the datatype of the iteration list, and here's where we can put that to use.

Let's try to create two **for** loops, one using the **range()** function and one with a list literal filled with integers.

```
1 for i in range(4):  
2     print (i)
```

```
0  
1  
2  
3
```

```
1 for i in [0, 1, 2, 3]:  
2     print (i)
```

```
0  
1  
2  
3
```

If it wasn't clear what **range()** was doing before, it should be clear now: **range()** is creating a list, and it's a shortcut to writing out each element that we want to print. This is really useful if you need to iterate a bunch of times (100, 1000, or even more!).

However, we're not limited to passing in integers in our lists. We can also pass in other datatypes, including floats and strings.

```
1 for i in ["Ronaldo", "Messi", "Neymar"]:  
2     print (i)
```

```
1 Ronaldo  
2 Messi  
3 Neymar
```

```
1 for i in [1.34, 2.71, 3.14]:  
2     print (i)
```

```
1 1.34  
2 2.71  
3 3.14
```

In the first example, we're iterating through three strings, and in the second example, we're iterating through three floats. Both of these are being passed in as list literals, but there's nothing preventing us from passing in a variable with a list in it.

```
1 cities = ["New York", "Atlanta", "Columbus"]
2 for i in cities:
3     print (i)
```

```
1 New York
2 Atlanta
3 Columbus
```

When iterating through an explicitly defined list (whether it's a list literal or a variable with a list), Python will always loop through the list according to the index number in ascending order. It won't sort by alphabetical order or string length, and you'll need to do some extra computing if you want your Python script to do such.

Exercise Questions

These exercise questions cover chapter 6.3.

Exercise 78

Consider the following for loop declarations. Are they syntactically correct? Why or why not?

1. `for i in range(5):`
2. `for i in range(1, 5):`
3. `for i in range(1, 6, 2):`
4. `for i in 5:`
5. `for (i in range(5)):`
6. `for (i in 5):`
7. `for i in [1, 3, 4, 6]`
8. `for i in [1, 2, 3, 4]`
9. `for i in ["1", "2", "3", "4"]`
10. `for i in ["apple", "strawberry", "banana"]`
11. `for i in fruits where fruits is a variable with the list of strings: ["apple", "strawberry", "banana"]`

Exercise 79

1. Using exactly two lines of code, write a loop that prints the following lines:
pear
papaya
pomelo
grape

2. Using exactly two lines of code, print the numbers from 1 to 100.
3. Using exactly two lines of code and the variable `a` as the iterator in your loop, print `a` asterisks, with each set of `a` asterisks on their own line, with the shortest line being just one asterisk and the longest line being ten asterisks. It should look like this:

```
*
**
***
****
*****
*****
*****
*****
*****
*****
*****
```

6.4 While Loops

The last major loop structure is the **while** loop. Compared to a **for** loop, a **while** loop is much more simple: it tests a condition and executes the code inside of that statement if the test is true and skips the code if not. **for** loops are really great at doing something a specified number of times, but **while** loops are really good at doing something either something a specified number of times or an indeterminate number of times. Let's look at both of these.

The easiest way to look at a **while** loop is as a different kind of iterative loop. However, while a **for** loop iterates using a list, **while** loops iterate using a condition. For example, we could set a variable **run** to start at 0, and each time the loop runs, the value of **run** is increased by 1. Then, we could set the condition of our **while** loop to be while **run** is less than 100. Let's take a look at this code in practice.

```
1 run = 0
2 while (run < 100):
3     print(run)
4     run = run + 1
```

```
1 0
2 1
3 2
4 ...
5 97
6 98
7 99
```

As you can see, this code runs exactly 100 times, starting at 0 and ending at 99. Once **run** is equal to 100, the condition is no longer met, so the code inside of the **while** loop is no longer run. If we wanted to print a different range of numbers, we can just change the **run** variable initialization and the condition of our **while** loop. If we wanted to print from 1 to 100 instead of 0 to 99, we could run the following code instead.

```
1 run = 1
2 while (run < 101):
3     print(run)
4     run = run + 1
```

```
1 1
2 2
3 3
4 ...
5 98
```

```
6 99
7 100
```

Similarly, if we wanted to change how much `run` increased with each iteration of our `while` loop, we could change the last line in the `while` loop.

```
1 run = 0
2 while (run < 101):
3     print(run)
4     run = run + 2
```

```
1 0
2 2
3 4
4 ...
5 96
6 98
7 100
```

In each iteration of our `while` loop, we're increasing the value of `run` by 2 instead of 1, which ends up giving us all of the even numbers.

As you can see, to change the value of `run`, we've been writing that the value of the variable should be the value of the variable itself plus some other number: `run = run + n`, where `n` is the number to increase by. Since increasing (or decreasing) the value of a variable by 1 is such a common practice across so many different applications, Python actually has a built-in mechanism for this exact purpose.

```
1 run = 0
2 while (run < 100):
3     print(run)
4     run += 1
```

In Python, we can replace the regular assignment operator `=` with a different combination of symbols to do arithmetic: `+=` or `-=`. The first will add the value of whatever is on the right side of the to the variable on the left, while the latter will subtract. The most typical combination is `variable += 1` to add 1 to the variable (equivalent to `variable = variable + 1`), but you don't have to use 1. `variable += 2` (equivalent to `variable = variable + 2`) or `variable -= 1` (equivalent to `variable = variable - 1`) are both syntactically correct.³

However, it's more likely that you'll be using a `while` loop for something other than iteration; otherwise, why would we have `for` loops? As its name

³Python's method of adding and subtracting one from a variable is just one way of doing it. Other languages, like C++, C#, Java, and JavaScript use double plus signs `++` or double negative signs `--` instead (`variable++` or `variable--`). This is equivalent to `variable += 1` or `variable -= 1` in Python.

suggests, **while** loops are great at testing conditions. It'd be a more realistic use case to test whether a specific variable is a certain value to determine if a certain chunk of code should be run again. We can use this property of **while** loops to take in an indeterminate number of variables. For example, consider the example from the **for** loop section where we took in four scores and computed the average. What if we didn't know how many scores the user needed to input? We could use a **while** loop to just keep taking in new scores until the user inputs something specific. Take a look at the following chunk of code.

```
1 grades = []
2 newGrade = 0
3 while (newGrade != -1):
4     newGrade = int(input("Input another grade or -1 to
5         stop: "))
6     if (newGrade != -1):
7         grades.append(newGrade)
8 average = sum(grades) / len(grades)
9 print("The average is", str(average))
```

Let's break this code down line-by-line. In the first two lines, we're creating two new variables: **grades**, which is an empty list, and **newGrade**, which is an integer. Next, we're creating a **while** loop. We know that a grade can never be negative, so we can let our user know to use -1 to stop asking for new scores. Next, we are having our user input the new grade and are putting that value into the variable **newGrade**. Then, we're testing whether **newGrade** is our special -1 value to figure out whether we should add the **newGrade** to the **grades** list, as we don't want to add the -1 value itself. This code will continue asking the user to input another grade until they enter a -1 value. Finally, it will calculate the sum of the grades list divided by the length of the grades list, then print the result.

It's also worth mentioning that it's possible to create an infinite loop in Python. Infinite loops can be dangerous, since they can take up all of the processor cycles and make halting execution difficult. The difficulty of this is has to do with the specific Python interpreter that you're using, but it's a good idea to just not chance things. For example, the following example creates an infinite loop.

```
1 while True:
2     print ("Hello, World!")
```

```
1 Hello, World!
2 Hello, World!
3 Hello, World!
4 ...
```

This code has nothing to ever turn the condition false, so this loop will run forever (or more specifically, until the program runs out of memory). Avoid writing infinite loops unless you have a really, really good reason to!

Infinite Loop Detection

Some IDEs have the ability to detect an unintentional infinite loop which is uncontrollable, and they'll insert special interrupts in order to help you halt the script. However, many IDEs do not have this functionality, so don't rely on your IDE to help you halt an infinite loop.

Copy and Pasting

"I was interviewing for a bunch of positions, and I was terrible at coding back then. Like, absolutely terrible. And they wanted me to write a function where, if there were a hundred lockers, write a function to go through each one and check something. So I wrote a function, but it was just for locker 1. And they asked me how I would repeat this a hundred times. Now, the right answer is to just write a loop, a for loop for a hundred times. And I said, 'just copy and paste it! Just copy and paste it a hundred times!'...I didn't get the job."

- Jeremy Wang (aka Disguised Toast, June 2020)^a

^a<https://youtube.com/watch?v=oiNPgJmtzVI&t=84>

Exercise Questions

These exercise questions cover chapter 6.4.

Exercise 80

1. What is the difference between a for loop and a while loop?
2. If a condition is never met in a while loop, what will happen?

Exercise 81

1. Write an example of an infinite while loop.
2. Write an example of an infinite for loop. Consider what needs to happen for an infinite loop to occur and replicate this in a for loop.

Exercise 82

Create a small script that asks the user for a sequence of numbers over and over again, until they put a negative number (like -1). Store all of the results in a list with the variable name `numbers`. The end list should look something like `numbers = [33, 48, 9, 6, 83]`.

6.5 Scope

One important concept that's difficult for many introductory programming students to grasp is scope. Scope allows us to keep our variables restricted to a certain block, and being able to effectively use scope to write clean code is an essential skill in higher levels of programming. Misusing scope or not knowing how it works is the source of a lot of frustration, and at worst, it can cause some pretty serious security issues.

Consider the following code.

```
1 # MARKER A
2 for i in range(5):
3     # MARKER B
4     print(i)
5 # MARKER C
```

We can tell what this code will do, but direct your attention to the markers in each of the comments. Where can we access the variable `i`? We know that we can't use it at marker A, since the variable hasn't even been declared yet. We know that we can use it inside of the `for` loop, though; we've seen this done plenty before. But what about marker C? In fact, we *cannot* use `i` here, since it is outside of the *scope* of where the variable was first declared.

Scope dictates where we can use a variable. In Python, a variable can be used at the level it was declared at and anywhere deeper, but never outside, unless it has been declared to be a *global variable*. This means that if you were to use a nested `for` loop inside of an `if` statement, you could use the iterator variable only inside of the `for` loop, not just anywhere in the `if` statement. However, if we declared a variable inside of the `if` statement, it can be used anywhere inside of the `if` statement, including inside of the nested `for` loop. Consider the following code.

```
1 a = 1
2 if a > 5:
3     b = 7
4     while (c < 10):
5         c = 3
6         c += 1
```

In the above code, we can use `a` anywhere inside of the program. We can use `b` anywhere inside of the `if` statement, but not outside of that specific `if` statement. We can use `c` only within the `while` loop.

Furthermore, if two statements are at the same level, they cannot share variables.

```
1 while (a < 10):
2     print(a)
3     a += 1
```

```
4 while (b < 15):  
5     print(b)  
6     b += 3
```

In the above example, `a` can only be used within the first `while` loop, and `b` can only be used within the second `while` loop. Attempting to use a variable outside of scope may result in a `NameError`, as the variable doesn't technically exist outside of its own scope.

Reinitialization Warning

Variables are *not* destroyed once execution moves beyond the loop, conditional, function, or class, even if they are not accessible. If you want to re-run the block from scratch, you *must* reinitialize them to what their starting values should be.

Another topic to discuss is the global variable: what if we want to be able to use a variable anywhere within our code? We can use a global variable instead of a local variable. Global variables can be accessed from within any class, including those in different files. You can simply make a global variable by declaring it in the outermost scope level, as shown here.

```
1 a = 0  
2 while (a < 10):  
3     ...
```

Let's say that you define a variable in the outermost scope level, as shown above. Then, inside of a function (covered in the next chapter), you want to use the exact same variable and memory space. You can use the `global` keyword to specify that Python should reference the exact same memory space that the variable label points to.

```
1 a = 0  
2 def someFunction():  
3     global a  
4     # do something  
5     ...
```

In the above function, the scope of `a` is accessible inside of the function, and its value is the exact same as outside of the function. This doesn't terribly matter for loops and conditional logic, but it does for classes and functions.

Exercise Questions

These exercise questions cover chapter 6.5.

Exercise 83

1. In your own words, define scope.
2. If a variable is "out of scope," what does that mean?
3. If you try to call a variable that is out of scope, what type of error might Python throw?

Exercise 84

1. What is global scope?
2. What are some advantages of global scope?
3. What are some disadvantages of global scope?

Exercise 85

For this exercise, consider the following code.

```
1 # MARKER A
2 i = input("Input a number")
3 # MARKER B
4 if (i > 5):
5     # MARKER C
6     print(i)
7 else:
8     # MARKER D
9     if (i == 5):
10        # MARKER E
11        k = True
12    else:
13        # MARKER F
14        k = False
```

1. At marker A, what variables are accessible, if any?
2. At marker B, what variables are accessible, if any?
3. At marker C, what variables are accessible, if any?
4. At marker D, what variables are accessible, if any?
5. At marker E, what variables are accessible, if any?
6. At marker F, what variables are accessible, if any?
7. Consider the variable `i`. Is `i` globally scoped? Why or why not?
8. Consider the variable `k`. Is `k` globally scoped? Why or why not?

Evaluation Copy

Chapter 7

Python Data Structures

Data structures are absolutely mandatory to using any programming language effectively. So far, all of our programs have been operating in a very linear matter: our program starts at the top, and it ends at the bottom. Now, we're going to examine some ways that Python can run that aren't linear. In the beginning, it will almost certainly feel useless. Why would we jump around when we can just write the code in the main body of the program? However, trust us, it'll come in handy down the road, and when applied correctly, both classes and functions can save a lot of time when writing complex programs.

7.1 Functions

Functions can seem somewhat useless. After all, if we wanted to run the same code, we could just copy and paste it! However, this isn't terribly efficient, and there's something to be said about code cleanliness that makes it easier to read. Functions can keep you from writing the same code over and over again. Think back to loops. Sure, we *could* just copy and paste a certain chunk of code to be run 100 times, but that's a lot of code for something that can be done with literally just one extra line of code. Functions can save you a lot of time when programming by allowing you to essentially reference another chunk of code with a single line of code.

The easiest way to think of functions in programming is as functions in algebra. When you first learned about mathematical functions, you were probably told that something goes in and something (probably different) comes out. For example, let's look at the function $y = 2x$. This is pretty simple: the input value x is multiplied by 2, and the output is y . If you put in the value 0, then you'll get 0 out, but if you put in anything else, it will be different. The input 4 will yield the output 8. The input 6 will yield the output 12. The input a will yield the output $2a$.

Let's consider a more complex function: $y = 2x + 3z - 9$. In this function, we have two coefficients (2 and 3, next to the x and z , respectively), a constant (9), and an output (y). If we give the values 5 for x and 42 for z , we get an output of 125 ($125 = 2 \times 5 + 3 \times 42 - 9$). We could also set a preemptive value for x or y , if the user doesn't tell us what x or z should be, like 0. So if we gave only the value 5 for x , then we can assume that z should be 0, so we would get an output of 1 ($1 = 2 \times 5 + 3 \times 0 - 9$).

Programming functions aren't that dissimilar in principle, but their inner functionality is a little bit more complicated than Algebra I functions. In programming, a function can, but doesn't always have to, take something in. Similarly, a programming function can, but doesn't always, have an output. Unlike math functions, functions in Python do need to have their own unique names.

7.1.1 Function Definition

Now that we know what functions are, how can we create them?

At their most simple, functions consist of four things: the function name, the arguments, the operations, and the return. Remember in chapter 4.1 how we talked about arguments? Now, we're actually going to use functions and those argument things! In programming, your *arguments* are the function's inputs. You can pass in as many arguments as you'll need to construct your function and do what you need. Some functions take in one argument, while others might take in up to 10 or 15 arguments. Going back to our math function, this would be having multiple variables opposite your output. For example, consider the quadratic equation, which takes in three variables.

$$x = \frac{-b \pm \sqrt{b^2 - 4 \times a \times c}}{2 \times a}$$

In Python, a , b , and c would be referred to as arguments. You should make a mental note that parentheses `()` are almost always associated with arguments of a function.

Similarly, the x in the quadratic equation above would be the output. In programming, the output of a function is called the **return value**. It's important that you don't confuse your *return* with the `print()` function. In the past, when we've referred to "output," it's typically meant that we were printing something to the console. However, now we're using "output" to mean something different. It's possible to return a value from a function without printing it.

In Python, we need to give each of our functions a name. Giving our functions a name allows us to refer to the functions later. In programming, we typically name our functions using alphabetical characters only, in camel-case. It is unusual to see a function named using symbols, such as dashes or underscores. Like variables, it's important that your functions are named something concise yet descriptive. Use your experience in naming variables to create good function names!

The contents of a function can manipulate any of the variables passed in as arguments. Inside of the function, any of the variables that you listed as arguments have already been declared and initialized, so you can just start using them!

Let's look at a function definition. In this function, we're going to calculate x from the quadratic formula, as shown above.

```

1 def quadraticFormula(pol, a, b, c):
2     negB = b * -1
3     top = 0
4     if (pol == "plus"):
5         top = negB + sqrt(b**2 - 4 * a * c)
6     elif (pol == "minus"):
7         top = negB - sqrt(b**2 - 4 * a * c)
8     bottom = 2 * a
9     x = top / bottom
10    return x

```

Notice how the first thing in this function is `def`. This lets Python know that we'd like to define a new function. The `def` is short for *definition*, and writing it lets Python know that the next thing is the function's name, followed by the arguments. Our function `quadraticFormula` takes four arguments: the polarity of the top plus/minus (`pol`), a (`a`), b (`b`), and c (`c`). Inside of the function, we're creating a new variable that can't be accessed outside of the function `negB`, which is the negative value of the argument `b`. Next, our function calculates the top and the bottom of the function. The top uses the polarity argument

to determine whether we should add or subtract the square root. Finally, we divide the top by the bottom and return that value. This function doesn't print anything on its own. Instead, its result needs to be printed by the function call, which we'll get to in the next section.

This is also the perfect opportunity to examine how we can use functions inside of other functions. Take a look at the lines that compute `top`. Inside, we see that most of the line is just simple arithmetic, but there's also another function call: `sqrt()`. `sqrt()` is a function in Python that calculates the square root, and we're able to use this function inside of our own function. In Python, we can actually use any function inside of our own functions, including other functions that we've made ourselves. It's even possible to call your own function in a technique called recursion, although this is a technique out of the scope of this book.

You can also see in the above code that we're returning `x`, which is a `float` or an `integer`. However, we don't need to just return a number; we can return any type of data that we want to, including strings, Boolean values, or compound datatypes, like lists and dictionaries. Let's look at a slightly simpler function that takes in a list as an argument and returns a simple Boolean variable. This function will test whether the list is longer or shorter than 10 elements. If it's longer than 10 elements, it will return `True`, otherwise it will return `False`.

```
1 def isLongerThan10(listToTest):
2     if (len(listToTest) > 10):
3         return True
4     else:
5         return False
```

In the previous example, we can see that our function name is `isLongerThan10` and that it takes one argument `listToTest`. Inside of the function, we see that we're using the variable `listToTest` that came as one of our arguments. We don't need to declare or initialize this variable, as it is an argument to the function that already has a value. Within our function, we're running an `if` statement to determine whether we should return `True` or `False`.

When we return a variable from a function, we need to put the return value into a variable in our function call. We'll see how to do this when we learn about function calls in the next section.

Functions that return something are called *returning functions*. It is also possible to create a function that doesn't return anything. Functions that don't return anything are called *void functions*. When writing a void function, you don't need to specify any return value. The function can still take arguments, but it doesn't have any output. For example, say you wanted to write a function that printed out an ASCII cow.¹

```
1 def __()
```

¹Reminder: Watch out for backslashes - you need to escape backslashes by adding another backslash to print just one. Why? Think about what `\n` does.

```

2             (oo)
3    /-----\ /
4    / |      ||
5    * ||----||
6      ^^      ^^

```

We could write each print statement out, line-by-line, every time you wanted to print out a cow. This would take some time, though, if we wanted to be able to print out a cow whenever we wanted to. Instead, we could write a void function that prints a cow, but that doesn't return anything. Again, remember that the `print()` function and the return value from a function are two different things.

```

1 def cow():
2     print("             (__)")
3     print("             (oo)")
4     print("    /-----\\ /")
5     print("    / |      ||")
6     print("    *  ||----||")
7     print("      ^^      ^^")

```

In the above example, there's not a single `return` statement like we've seen in the previous examples that return floats, integers, or Boolean values. Instead, calling this function will just run the code inside of the function, literally printing a cow to the console.

Void functions can also take arguments. Our previous example was named `cow()`, and it only printed a cow, but we could create another function named `animals()` that printed one of three animals, which would be passed in as an argument of type string.

```

1 def animals(animal):
2     if (animal == "cow"):
3         print("             (__)")
4         print("             (oo)")
5         print("    /-----\\ /")
6         print("    / |      ||")
7         print("    *  ||----||")
8         print("      ^^      ^^")
9     elif (animal == "pig"):
10        print("    n..n")
11        print("e__ (oo)")
12        print("(____)")
13        print("//  \\\\")
14    elif (animal == "fox"):
15        print("|\\_/_/|, ,----,~~('")
16        print("(.\\\".)~~~)('~)")
17        print(" \\o/\\ /---~\\\\\\ (~)")
18        print(" _// _// ~)")

```

```
19         else:
20             print ("Please use argument \"cow\", \"pig\"
                    or \"fox\".")
```

As you can see, the `animals()` function takes in one argument `animal`. Again, this function has no return type; it's a void function, even though it takes in an argument.

7.1.2 Function Calls

Calling functions is even easier than making them. In fact, you've already called plenty of functions as you've been writing code! Whenever you use `print()`, `input()`, or anything else with parentheses, you're actually calling functions that are already built into Python!

You can also call your own functions, once you've written them. Our first example of a void function printed a cow. We can call this function, just like we called the `input()` function without any arguments, since `cow()` takes no arguments.

```
1 cow()

1         (__)
2         (oo)
3     /-----\ / - moo
4   / |         ||
5  *  ||-----||
6     ^^         ^^
```

Above, we also created a new function called `animals()`, which takes in an argument `animal`, which can be either "cow", "pig", or "fox". To call this function, we can just write our function, along with the required argument. For example, for us to print a fox, we can call `animals("fox")`.

```
1 animals("fox")

1         | \_ / | , , ----- , ~ ~ '
2         ( . " . ) ~ ~         ) ( ~ )
3 ring-ding-ding-ding-ding - \ o / \ / --- ~ \ \ ~ )
4         _ //         _ // ~ )
```

In this function, we're not returning anything, so we don't need to put the contents of the function return value into a variable. But, what if we are returning something? In this case, we need to do something with the return value. We can either pass it into another function, such as `print()`, or we can put its result into a variable for later use.

Consider the following function, which finds the y-value, given a slope, a x-intercept, and a x-value.

```
1 def linear(m, x, b):
2     result = m * x + b
3     return result
```

We can then call this function, passing in the three arguments specified and either printing them (using them in another function) or storing their result in a variable, since the function does return something.

```
1 print(linear(7, 4, 4))
2 print(linear(3, 6, 8))
3 result1 = linear(3, 9, 2)
4 result2 = linear(6, 2, 2)
```

```
1 32
2 26
```

Observe how because the `linear()` function returns something, we must do something with that returned value. However, with the `cow()` or `animals()` function, we cannot do anything except call the function, since it does not return any value. In fact, trying to print or put the result of a void function is syntactically incorrect.

```
1 # INCORRECT!
2 print(cow())
3 # CORRECT
4 cow()
5 # INCORRECT
6 linear(7, 4, 4)
7 # CORRECT
8 print(linear(7, 4, 4))
```

Some IDEs also have the ability to gather special bits of information from your function definition that can be useful in larger projects. This is typically done inside of a block comment just after the function definition. You should consult with your IDE's developer to learn how your IDE can help you with your function declarations.

In Anaconda Spyder, the comment format is as follows. You can have Spyder automatically generate the format of the block comment by typing your opening quotes directly after the function definition.²

```
1 def calculateGrades(grade):
2     """
3     Calculates the grades of a student and returns the
        average.
```

²Anaconda Spyder refers to arguments as "parameters". Again, consult your IDE's developer to learn how it refers to arguments and return values.

```

4
5     Parameters
6     -----
7     grade : list of integers
8         A list of exam scores to be averaged.
9
10    Returns
11    -----
12    result : float
13        The average score of all scores in the list.
14
15    """
16    result = sum(grade) / len(grade)
17    return result

```

While this is a block comment and the Python interpreter won't use anything inside of it, it's easier for you to read. As a bonus, any time you click on a function call in your code and press Ctrl/Cmd + I, Anaconda will show you exactly what arguments you need to pass in its integrated help window! This feature and its specific functionality is specific to Spyder.³

Other IDEs might handle the block comment slightly differently. repl.it, a popular online IDE, doesn't prepopulate your block comment with anything, but it will show anything in your block comment if you hover over the function call, so this is a great place to put what this function does, what its arguments are, and what it returns. A simple template is provided for you here, if your IDE doesn't provide you with one.

```

1  """
2  Description: Write a description here for what your
3              function does.
4  Arguments:
5      arg1: (int) Description of the argument
6      arg2: (string) Description of the argument
7  Returns:
8      returnVar: (float) Description of what the function
9                  returns
10 """

```

If your IDE supports a specific template, it will auto-generate it once you create your block comment after your function definition. So, you can simply define a block comment as the first line of your function. If it is supported, it will be created for you. Otherwise, you can just paste in the template that works the best for you.

Like whitespace, it's a good idea to get into the habit of always writing a detailed block comment for each of your functions, including what the function

³As of Spyder v.4.1.4

does, what its arguments are, and what it returns. Regardless of what template you use, make sure that you're descriptive in your code.

7.2 Classes

⁴Learning how to effectively use classes will prepare to practice a discipline of programming known as *object-oriented programming* or *OOP*. Object-oriented programming falls under the idea of programming composition, and it hinges on the idea that objects can contain other objects. That means that you could put a box inside of another box. Not only could you put a smaller box inside of a bigger box, you could put multiple smaller boxes inside of the larger box. The key behind object-oriented programming is that it allows us to store information together, rather than parsimoniously.

Consider how the leap from individual variables to complex datatypes, like dictionaries, enhanced our ability to communicate clearly while writing less code. Further object orientation helps the code's reader understand not only that the information is together, but that they are related. The goal is to help us represent simple information unambiguously and simply.

Let's consider how we could store a three-part date with year, month, and day. We could try to store this as a string, but there are lots of different ways to represent a date in a string.

```
1 "1980-01-02"
2 "1980/1/2"
3 "1/2/80"
```

To alleviate this, we could store it in three separate variables.

```
1 year = 1970
2 month = 1
3 day = 2
```

However, this requires us to use two more variables than just storing the information in a single string. We could even try to use a dictionary, but we run into the same representation issue.

```
1 date1 = {"year": 1980, "month": 1, "day": 2}
2 date2 = {"year": 1980, "month": "January", "day": 2}
```

Strictly speaking, both of these are correct, but only one would be valid unless we managed to code in every single option. This is where classes come in.

7.2.1 Class Definitions

Defining a class is the first step of using classes, and it looks shockingly similar to working with functions. Let's take a look at a function definition first:

```
1 def yint (m, x, b):
```

⁴If you haven't grasped the concept and application of functions yet, stop reading here. Go understand functions first, otherwise classes won't make any sense. It's okay.

Now, let's look at a class definition that has the above function definition.

```
1 class graphing:  
2     def yint (m, x, b):
```

As you can see, the process of defining classes is remarkably similar to that of functions.

7.2.2 Class Calls

Being able to call your class methods is important to actually being able to use those functions.

7.2.3 Class Methods

We've used the term "method" already, but what exactly is it?

7.3 Functions Versus Methods

Now that we know what a class method is, how exactly is it different from a regular old function? Sure, on the surface, they look the exact same and they even use the exact same definition syntax!

7.4 Structures (Optional)

There's a concept in other programming languages known as structures, or structs. Structs are another way of structuring data compared to classes, but unfortunately, Python doesn't natively support structs. However, we can still go over what a struct is so that should you choose to pick up a new programming language down the line, you can understand how that code works.

Let's say that you wanted to store a date. There's several ways that we could store this date. We could make an entire class called `date`, but this seems kind of wasteful and somewhat backwards. It's a lot of code for something so simple. Likewise, we could store our year, month and date in individual variables, but this seems kind of messy and like it could cause readability issues. Instead, we can make a struct. A struct is like a combination of data that can be accessed as a single datatype, but without the overhead of a class.

In performance oriented languages, like C++, C#, Swift, or Java, structs are better than classes, since they require less memory to store and the runtime doesn't need to use all of the features of a class. In our previous example, we could make a `date` struct that had three elements: year, month, and day. We can then call the entire struct to get all of its data or get specific pieces of data out of this struct by using our period notation.

Python is not a performance oriented language. It has a lot of overhead, especially as an interpreter-driven language, so the performance advantages of a dedicated struct data structure don't really apply to it.

Evaluation Copy

Chapter 8

File Handling

In chapter 4.2, we went over variables, and it was noted that "all data that is worked on in variables is stored in memory when you're running a Python script." "Since all of your variables are stored in memory, they can only persist while the program itself is running. After your program is terminated, the memory spaces is marked as free by the operating system, meaning that any other program is now free to overwrite that memory." But what if we want to persist data between sessions? The simplest way to do this is by interacting with files that are stored at the secondary or tertiary levels. Python has the ability to both read and write to those files in several different ways, each of which has its own advantages and disadvantages.

8.1 Reading Files

The easiest way to interact with files is to open them in a read-only mode. Read-only files cannot be edited by your Python script, meaning that if something goes wrong in your script, nothing will happen to the file. For the most part, you'll be working with plain-text files. These are files that can be opened in simple text editors, like Atom, BBEdit, or Notepad++. Programmers like to work with plain-text files, since they're easy to open, edit, and close without any decoding. File formats like `.docx` (Microsoft Office Word 2013 or later) or `.odt` (OpenDocument Text format) are much more complicated, and it's difficult to edit them with plain-text editor unless you're really determined and you know what you're doing. For these purposes, we'll only be working plain-text files.

Plain-text file formats include `.txt` (text document), `.csv` (comma-separated values), `.tsv` (tab-separated values), `.xml` (eXtensible Markup Language), `.json` (JavaScript Object Notation), or even source code, like `.py` (Python script) or `.cpp` (C++ source code). These file formats don't require any special software to open, so they can just be opened with a standard text editor. This also means that they're really easy to manipulate in programming languages like Python. That's right, you could use Python to make more Python!

For now, we'll only focus on the most common type of plain-text file format to manipulate: simple text documents. Later, we'll also manipulate comma-separated value files, but we'll use another library to do this. The most basic way to interact with a text document is by reading it. Reading a document means that you are storing all of the contents of the document in memory, then using Python to manipulate the temporary version of the document in the memory. The original file is never altered. By consequence, this is also the safest way to manipulate files. There's never an opportunity for the file to be corrupted by the operations that are happening to it.

In Python, we can open a file using the `open()` function. `open()` is built right into Python, so you can just call it. It returns a file object, which is a compound datatype that contains the contents of the file, along with some extra bits of information that are useful, especially in higher levels of programming. By itself, a file isn't very useful, so we also need to use the `read()` method on the file object. `read()` returns a string with the contents of the entire file.

When we use the `open()` function to open the file, we also need to specify the *mode* by which we are opening the file. Since we are only looking to read the file, we can use the argument `"r"` for "read-only". This argument is passed in as the second argument of the `open()` function. Don't confuse the `"r"` for read-only with the `read()` method, which operates on the file object. Let's take a look at how we use the `open()` function and `read()` method.

```
1 file = open("fileToOpen.txt", "r")
2 fileContents = file.read()
```

In this code, we're opening the file named `fileToOpen.txt` in read mode, as indicated by the second argument of the `open()` function. We're storing the

file object that the `open()` function returned in the variable `file`. By itself, `file` isn't very useful, so we need to `read()` the contents of the `file` into `fileContents`. We can then do whatever we'd like to `fileContents`. No additional changes will or can be made to the original `fileToOpen.txt`.

However, when you run the `read()` function, you'll probably notice that it puts everything on one line. This is because as the string is returned from the `read()` function, unnecessary whitespace is discarded "as a courtesy" to the programmer. If you want to keep the whitespace, you'll need to use the `readline()` method instead of the `read()` method. `readline()` reads just one line from the file object. Consider the following code.

```
1 file = open("fileToOpen.txt", "r")
2 print(file.readline())
```

This code would actually only read the first line of the text file, which might be useful, but you probably want the entire text file. Instead, we can run `readline()` multiple times.

```
1 file = open("fileToOpen.txt", "r")
2 print(file.readline())
3 print(file.readline())
```

This code will read the first two lines of the text file. Again, this might be useful if you know that you only have two lines. However, it'd probably be easier to read the entire file line-by-line. We can do this by iterating through the file object using a `for` loop, where the iteration variable is any of your choosing and the list to iterate through is the file itself. Each element of the file is a line. Consider the following code.

```
1 file = open("fileToOpen.txt", "r")
2 for i in file:
3     print(i)
```

This will print each line in the file. We could also store the contents of the file in a list. Let's say that you had a list of words, with one word per line. You could read your list of words line-by-line and put each word into the next element of your list.

```
1 file = open("fileToOpen.txt", "r")
2 listOfWords = []
3 for i in file:
4     listOfWords.append(i)
```

When you've finished working with a file, it's important to close the file. Even if you open the file in read-mode, closing the file allows the memory that was used to point to the file to be freed, and it's especially important when editing and writing to files. Regardless of how you're working with a file, you should get into the habit of closing a file when you're done with. To indicate that you'd like to close the file, you can use the `close()` method on the file object.

```
1 file.close()
```

It's worth noting that Python will look for the file in the current working directory. If you're using a local IDE, like Anaconda Spyder, you can run the command `pwd` in the interactive Python shell, then put your file in that same directory in order for it to be found by your Python script. You can also navigate around your current working directory by using relative filepaths: `.` means your current directory and `..` means one directory up. Alternatively, you can specify an absolute directory. An absolute directory path is one where the entire filepath is written, from the root to the file itself. If `./fileToOpen.txt` is using the relative filepath, the absolute filepath might be `Users/Guest/Downloads/fileToOpen.txt`. You should be able to view the properties of a file in your operating system to view its absolute filepath.

If you're using an online IDE, like repl.it, you can just reference the file by name unless it's in a subfolder. When using online IDEs, your file is almost always stored in the same directory as your Python script. If your file is in a subfolder from your script, you can just specify the subfolder before naming the file, along with a forward slash. For example, if your file is in the `TestDocuments` directory, you can specify that your file should be read from `TestDocuments/fileToOpen.txt`.

8.2 Writing Files

What if you want to do more than read files, you also want to write to those files? Well, Python has the ability to do so, but first, we need to understand what the dangers of writing to files are.

When we opened files in read-only mode, we were offered the safeguard that if our code wasn't completely correct, we couldn't also muck up the file. The file was opened in such a way that Python had no way to change any of the contents of the file. However, when we open a file in read-write mode, we lose that safeguard. If we wrote our code incorrectly, there's a potential that we could cause serious damage to a file, especially if the file is preexisting. When manipulating files, you should be double-sure that your code is correct and that the information that you're writing back out to the file is absolutely correct. If you're not sure, you should also make backups of your file before you start to copy it. You could do this by opening the file in read-only mode, then writing a new file with the contents of the old file unchanged, or you can just do it manually in your operating system. If you don't *need* to open a file in a writing mode, then don't; stick to the safe alternative and open the file in read-only mode.

Phew! Now that we've gotten that out of our system, let's talk about the two ways that we can open a file in writing mode. We can open a file either to write to it, or to append to it. While these two methods sound quite similar, they're very different, and one has the potential to really ruin your day. Let's start with appending. As we've seen in the lists section, to append means to add to the end. The same thing applies when manipulating files. Appending to a file means that we're simply adding new material to the end of a file. The original contents of the file remain unchanged.

Conversely, opening a file in write mode is much more dangerous. Unlike appending to a file, writing really means you're **overwriting** the old contents of the file. If you want to keep some of the old contents of the file, you need to read all of it into your program's memory, edit what you need to, then write the *entire* file back out. Anything that you don't write back out will be deleted for good. There's no recycling bin or trash can to catch your work if you accidentally overwrite something that you didn't mean to overwrite. Once it's gone, it's gone for good!

Like `read()` for reading files, writing has its own set of methods for manipulating and editing files. We'll still use the same `open()` function, but instead of passing in `"r"` for read, we're going to pass in other modes: `"a"` for append and `"w"` for write/overwrite. Conversely to the `read()` method in our read-only mode, we can use the `write()` function to write something to the file. You can also read anything from the file. Take a look at the following code, which will append something to the end of the `fileToOpen.txt` file.

```
1 This is a file
2 It has stuff in it
```

```
1 file = open("fileToOpen.txt", "a")
2 for i in file:
3     print(i)
4 file.write("Something to add to the end")
5 file.close()
```

```
1 This is a file
2 It has stuff in it
3 Something to add to the end
```

As you can see, we're opening the file that we've specified in append mode. We're then using our `read()` function to read the current contents of the file before adding a new line to the end of the file using the `write()` method on the file object. Like we mentioned above, it's important to close your file once you're done with it, but it's extra important when you're writing or reading to a file. While you have a file open, it means that no other process can touch that file.

Overwriting a file works in very much the same way. In fact, all we're going to do is change the file mode.

```
1 This is a file
2 It has stuff in it
```

```
1 file = open("fileToOpen.txt", "w")
2 for i in file:
3     print(i)
4 file.write("Something new in the file")
5 file.close()
```

```
1 Something new in the file
```

The code is identical, except for the writing mode on the first line. However, if we were to try and open this file in a text editor, we'd find none of its old contents - they've all been overwritten by our new write line.

8.3 Different Kinds of Files

If you're a runner, cyclist, or swimmer, you've probably used the fitness app Strava® or something similar. If you have, you'll know that you have the option to load in a `.gpx` or a `.tcx` file. These files are really useful if you record your fitness data from, say, a smart watch or from a head unit. These files aren't necessarily special because their extension is `.gpx` or `.tcx`. In fact, these files are bog-standard XML files. They've just been customized with a custom file extension and the structure of these files is designed to be read by specific pieces of software, like Strava®. In fact, you can try it! If you're on Strava®, you should be able to export a GPX file of any of your activities. Open this GPX file in Notepad (on Windows), TextEdit (on macOS), or your OS's text viewer if you're on some other OS like Linux, and you'll see that the first line indicates that the file is a standard XML file.

Similarly, if you go to any webpage (literally, any webpage) and view the source code of that webpage, you can see that the webpage is written and/or rendered in HTML. However, this webpage is just "spicy" XML. It's a type of XML that web browsers are really good at reading. If you're on Google®Chrome®, just insert `view-source:` before the URL in the address bar. If you're on Firefox, just right-click on the web page and select Show Page Source.

All of these files are readable by a standard text editor, like Notepad or TextEdit, and you'd be shocked at how many of these "custom" filetypes are actually just XML (eXtensible Markup Language), JSON (JavaScript Object Notation), or SQLite (Structured Query Language Lite) files, and in fact, you could use Python to crack open these files and edit attributes from them.¹

Using this, you can actually create your own file formats! Let's say that you're editing a plain-text file, and you know that it's going to have a very structured format. For example, the first line of the file will always be the title, the second line will always be the author, and the third line contains the data to be stored. You could store it as a custom filetype by just specifying a different file extension when you write the file out in Python. Since it's a plain-text file, you can still read and write in any text-editing program, including Atom or BBEdit®.

¹Please don't take this as an endorsement to start faking your own `.gpx` files and uploading them to Strava®. You'll probably get flagged, and that's not my problem.

Evaluation Copy

Chapter 9

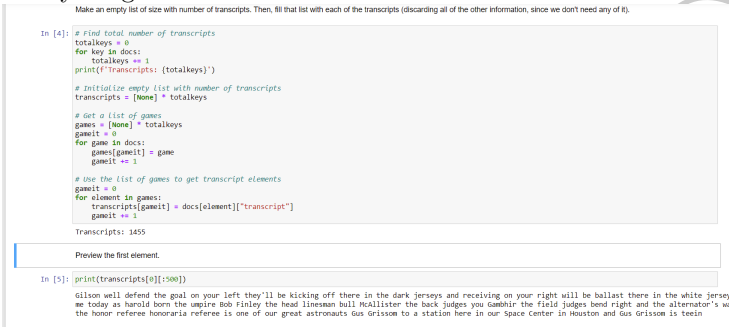
Jupyter Notebooks

Jupyter Notebooks are a very useful data analysis tool, especially among data scientists and Python programmers, since they allow you to run code inline with your document. There are many tools that allow you to create and edit Jupyter Notebooks, such as Google Colab or Anaconda.

9.1 What Are Jupyter Notebooks?

At their most basic, Jupyter Notebooks are just spicy binders. They allow you to write Python code and add notes in a quick and easy way. You'll find that Jupyter Notebooks are actually quite common in academic settings, since they allow academics to share their exact findings with each other in an organized manner.

Jupyter Notebooks always have the file extension `.ipynb` (though the curious among you may try to open these files in a text editor to discover that they're just JSON files). IPYNB stands for Interactive PYTHON NoteBook. It's not uncommon to have multiple notebooks, although some people choose to pile everything into one notebook.



```

Make an empty list of size with number of transcripts. Then, fill that list with each of the transcripts (discarding all of the other information, since we don't need any of it).

In [4]: # Find total number of transcripts
totalkeys = 0
for key in doc:
    totalkeys += 1
print('Transcripts: {totalkeys}')

# Initialize empty list with number of transcripts
transcripts = [None] * totalkeys

# Get a list of games
games = [None] * totalkeys
gameit = 0
for game in doc:
    gameit[gameit] = game
    gameit += 1

# Use the list of games to get transcript elements
gameit = 0
for element in games:
    transcripts[gameit] = doc[element]["transcript"]
    gameit += 1

Transcripts: 1455

Preview the first element.

In [5]: print(transcripts[0][:500])

Gillon will defend the goal on your left they'll be kicking off there in the dark jerseys and receiving on your right will be ballast there in the white jerseys
we today as harold born the umpire Bob Finley the head linesman bull McAllister the back judges you Gambair the field judges bend right and the alternator's wall
the honor referee honoraria referee is one of our great astronauts Gus Grissom to a station here in our Space Center in Houston and Gus Grissom is teeing
  
```

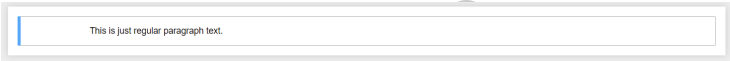
When you're writing in a Jupyter Notebook, you can either write in a markdown block or a code block. Your markdown code will only be rendered, not executed. Even if you specify that a chunk of code should have its syntax highlighting in Python, it will not be executable. Essentially, your markdown code is read-only. However, your code blocks are read/write/executable. You can execute any of the Python code inside of a dedicated Jupyter Notebook code block.

9.2 Basic Markdown Syntax

Jupyter Notebooks support standard markdown syntax, such as what you might use on an Internet forum or on GitHub in `.md` files (like READMEs or Contributing files). If you've never worked with markdown before, it's not too difficult. Markdown is just a way to change the appearance of plain-text while writing in plaintext. When writing markdown, you'll typically write it in a plain-text form, then open the same document in a markdown renderer. When you're working with Jupyter Notebooks, your notebook will be your markdown renderer.

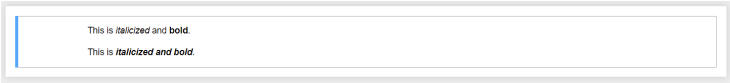
The most basic of the markdowns is paragraph text. Paragraph text is written with no extra or special symbols.

```
1 This is just regular paragraph text.
```



You can add additional emphasis styling to your paragraph text, just like you would in any other word processor: italics and bold. To italicize something, use single asterisks to mark the beginning and end of the italicized area. To bold something, use double asterisks to mark the beginning and end of the bolded area. You can also use combinations of asterisks to denote both italicized and bolded text. To create a line break, add an extra blank line, as a single line break won't add an extra line. Your symbols cannot span multiple lines, so if you want more than one line to be emphasized, you need to write more than one set of symbols for the respective emphasis symbol sequence.

```
1 This is *italicized* and **bold**.  
2  
3 This is ***italicized and bold***.
```



Another common block to use is writing snippets of code. This is different from writing code in a Jupyter Notebook code block. Code in a markdown block cannot be run; only code in code blocks can be run. You can choose the syntax highlighting that is used when writing in a code block. To write in a code block, use three backticks to mark the beginning and end of your code block. To write inline code, use single backticks.

Where's the backtick?

The backtick is the character right next to the number 1 key on most US-layout ANSI and ISO keyboards. Shift-Backtick normally gives you tilde ~.

```

1 This is some inline code: 'print()'
2
3 '''
4 # No syntax highlighting
5 print("Hello, world!")
6 print("This is Python!")
7 '''
8
9 '''python
10 # Python syntax highlighting
11 print("Hello, world!")
12 '''
13
14 '''cpp
15 # C++ syntax highlighting
16 std::cout << "Hello, World!" << endl;
17 std::cout << "This is C++!" << endl;
18 '''

```

This is some inline code: print()

```

1 # No syntax highlighting
2 print("Hello, world!")
3 print("This is Python!")

```

```

1 # Python syntax highlighting
2 print("Hello, world!")

```

```

1 // C++ syntax highlighting
2 std::cout << "Hello, World!" << endl;
3 std::cout << "This is C++!" << endl;

```

¹The exact coloring and style of your Jupyter Notebook code will depend on the renderer that you're using.

¹These colors have been simulated.

At the top of your project and to divide each of the subsections, you might want to include a header. There are six header sizes, which are determined by how many # symbols are added before your heading text.

```
1 # The largest heading
2 ## The second largest heading
3 ##### The smallest heading
```

You can also include links in your code. To create a hyperlink, wrap the link text in brackets [] followed by the URL in parentheses ().

```
1 We're writing code in [Python](https://www.python.org/).
```

The best way to get to know how to use and write markdown is to just write more text in markdown. Before long, it will become second nature!

9.3 The Interactive Python Shell

Let's get to the "interactive" part of IPYNBs. The killer feature of Jupyter Notebooks is their ability to run code inline. Jupyter Notebooks that are opened in compatible software actually have their own Python interpreter that can run Python code, including loading modules and reading files.

In order to use this interpreter, you need to add a code block to your Jupyter Notebook. Every piece of software has a different way of doing this, but once you've gotten your code block in your notebook, you can then write any Python code inside of that block. One of the really cool things about Jupyter Notebooks is your ability to create multiple code blocks in one notebook. Even when you have multiple code blocks, your notebook will still keep all of the variables and states that you created in earlier blocks. For example, let's say that you created a variable first thing in your Jupyter Notebook. Then, three code blocks later, you decide that you need to use that variable. The variable is still active until the Jupyter Notebook Python runtime is reset or the notebook is closed.

Once you've written your code in a Jupyter Notebook, you can run it by either using a keyboard combination (typically Shift + Enter) or by pressing the Start button on the left side of the code block. Once the code block has finished its execution, you'll be able to use any of the variables that you've declared in future code blocks.

When you are running code in a Jupyter Notebook, it is entirely likely that you will make a mistake when writing your code. You might accidentally make an infinite loop, or perhaps your program isn't asking for user input correctly. In these cases, it is important to know how to *interrupt* the flow of your program. In the console, the easiest way to send an interrupt (specifically, a **KeyboardInterrupt**) is to press Ctrl+C on the keyboard. This interrupt was designed to be sent by a keyboard during the DOS computer days (and before), when the keyboard was the only method a programmer had of interrupting the execution of a program. In fact, modern "stop" buttons simply emulate a Ctrl+C in the IDE.

Chapter 10

Data Analysis with Pandas

In all honesty, Python makes a bad web development language and an even worse desktop development language¹. However, one of Python's biggest strengths is in scientific computing and in statistics. If you've used R before, you'll find that Python isn't that much different. In fact, both R and Python are built on the C programming language! It does use a different set of packages, but you'll find that with a little bit of statistical translation, you can do everything that you could do with a dedicated statistics programming language, such as R or SAS.

¹This is not a fact, the author just really hates Python for desktop and web development.

10.1 Pandas are Not Bears

While R has much of its functionality baked right into Base-R (as a dedicated statistics language), Python requires you to load in the appropriate libraries. The hands-down most popular library is Pandas, and it's an incredibly powerful package that's appropriate for all sorts of data analysis. There are many other packages out there that are actually built on Pandas, like NumPy (pronounced Num-Pie, *not num-pee*), TensorFlow, Scikit-Learn, Matplotlib, Seaborn, and many others. Much of the inner workings of economics services are built using Pandas, too, like Robinhood, Quandl, Morningstar, and Google Finance.

Pandas's Etymology

Pandas actually stands for something: Python ANalysis for DATA. It was originally created for financial research by Wes McKinney at AQR Capital Management.

Take your mind back to chapter 8, when we were working with a new library called `csv`. Well, like `csv`, Pandas is also a library. In fact, we can use the same `import` statement as we used with `csv`.

```
1 import pandas
```

However, when you look at most code that uses Pandas, you'll notice that it's *aliased*. That means that the library name `pandas` has been assigned a nickname `pd` that can be used to reference library methods at any point in that script. The common `pandas` alias is `pd`, and we can create a library alias by using the Python keyword `as`.

```
1 import pandas as pd
```

Now, we can access the `pandas` method by using `pd`. For example, instead of typing out `pandas.DataFrame()`, we can just call `pd.DataFrame()`.

Use `pd`, not something else

Avoid creating a different alias for pandas than "pd", which has become the default alias. Using a different alias may cause confusion in readers.

Aliases With Other Libraries

Other libraries have well-known aliases, too. "numpy" is aliased "np", "scikit-learn" is aliased "sk", and "BeautifulSoup4" is aliased "bs4".

10.2 New Datatypes

Learning how to use Pandas means that you need to learn how to use two new datatypes: the series and the dataframe. These datatypes were built expressly for the purposes of data analysis while using Pandas, and they build upon the initial structure of the dictionary and the list.

These datatypes are only available when you use Pandas, so make sure you import the Pandas library before trying to instantiate a new series or dataframe.

10.3 Series

Series aren't used nearly as much as dataframes are in Python data analysis. Series are most akin to lists in Python, but they differ in the way that the data is stored and the methods that are available to use. However, it is possible to typecast from a Python list to a Pandas series and vice-versa.

That being said, series are still important to cover, as they are the fundamental building blocks of dataframes, which we'll cover in the next section. Cast your mind back to when we looked at lists. A list might look like the following.

```
1 var1 = [56, 52, 38, 62]
```

If we look at the type of this list, we see that it's a type list.

```
1 type(var1)
```

```
1 <class 'list'>
```

Now, let's make this Python list into a Pandas series. Remember, a list looks like a series.

```
1 var1 = pd.Series(var1)
2 type(var1)
```

```
1 <class 'pandas.core.series.Series'>
```

We have actually typecast the `var1` object from a Python list into a Pandas series. Observe how both series and lists are one-dimensional. However, what makes series special is how their indices are inherent and modifiable.

To make this list, we used the Pandas **Series** method. Because **Series** is in the Pandas class, we need to call **Series** as a method on Pandas, but as shown, we're using its alias `pd` instead of typing out `pandas`. We could just as easily type out the full class as `pandas` instead of `pd`.

```
1 var1 = pandas.Series(var1)
```

However, since the `pd` moniker is so well known, we'll just use its alias.

In order to make a series, we will always use the **Series** method from the Pandas class. If we wanted to create an empty series, we would just pass nothing into the method.

```
1 emptySeries = pd.Series()
```

`emptySeries` is of type series, but it has nothing in this series. We can also pass in a single list, as we did above.

```
1 var2 = pd.Series([122, 139, 185, 115])
```

Variables Versus Literals

Notice how in `var1`, we passed in a variable with a list, while in `var2`, we passed in a list literal. As long as the datatype is correct, Python will make the series.

If we tried to print the list before we typecast `var1`, we would end up with something like this.

```
1 [56, 52, 63, 38]
```

It prints just like any other list, complete with square brackets and commas. However, when we print our series, we see something a little bit different.

```
1 print(var1)
```

```
1 0    56
2 1    52
3 2    63
4 3    38
5 dtype: int64
```

In our second column, we see the values that we typecast from our list into our series, but in the first column, we also see index values, starting at zero. Using this index, we can actually extract data from the series just as we did when we were working with lists. We will use square brackets to indicate which index we want.

```
1 print(var1[1])
```

```
1 52
```

Indices start at zero

Reminder: indices still start at zero, even in Pandas series!

We could also create our own indices for a series. Recall how we made our `var1` series. We passed in a list only. In order to create an index, we can also pass in the argument `index`, which should be of type `list`.

```
1 var3 = pd.Series([3, 2, 2, 3], index = ['a', 'b', 'c',
    'd'])
```

Now, if we attempt to look at `var3`, we'll notice how it still has our values in the second column, but the first column has the index that we specified as a list.

```
1  a      3
2  b      2
3  c      2
4  d      3
5  dtype: int64
```

a, b, c, and d became our index instead of the default 1, 2, 3, and 4.

Check your indices!

Make sure that your index has the same number of elements as the data that you are putting into the array. If your indices are mismatched, you will end up with a syntax error.

Getting data out of a series with an explicitly set index is similar to how we get data out of a dictionary. We still use square brackets, and we put the index that we defined. For example, if I wanted the `c`'th element of the `var3` series, I could just refer to the `c`'th element as a string.

```
1  print(var3['c'])
```

```
1  2
```

Because `c` is of type string, we had to put our index into our square brackets inside of quotes, similar to how we need to put keys of a dictionary into quotes.

After learning how to change the index, you may have realized that a series can act like both a list *or* a dictionary! If not, now you know. When we referenced our series without an explicit index, we could just refer to a series element by that index number.

```
1  print(var1[1])
```

```
1  52
```

However, when we refer to a series with an explicit index, we refer to a series element by that index value that we set.

```
1  print(var3['d'])
```

```
1  3
```

Let's look at a more concrete example. Consider this list which has been typecast to a series.

```
1 speeds = pd.Series([84, 93, 66, 89, 58, 59])
```

If we wanted to refer to the n 'th element of the `speeds` series, we would just refer to the index number as n .

```
1 print(speeds[n])
```

Now, let's consider a list which has been typecast to a series, but which has an explicit index.

```
1 agility = pd.Series([90, 90, 96, 86], index = ['Cristiano Ronaldo', 'Lionel Messi', 'Neymar', 'Luis Suarez'])
```

In order to get any of the data out of the `agility` series, we need to know the indices or we need to just print the entire series.

```
1 print(agility)
2 print(agility['Neymar'])
```

```
1 Cristiano Ronaldo    90
2 Lionel Messi        90
3 Neymar              96
4 Luis Suarez         86
5 dtype: int64
6 96
```

If we didn't want to create a dictionary-like series as two distinct lists (one list with the data, one list with the indices), we can actually pass in a dictionary into the `Series` method and leave out the `index` argument altogether. Consider the following dictionary.

```
1 composeure = {'Cristiano Ronaldo': 86,
2              'Lionel Messi': 94,
3              'Neymar': 80,
4              'Luis Suarez': 84}
```

If we pass this dictionary into the `Series` method, Pandas will typecast the dictionary into a series using the keys as the index and the values as the data.

```
1 composeure = pd.Series(composeure)
2 print(composeure)
```

```
1 Cristiano Ronaldo    86
2 Lionel Messi        94
```

```

3 Neymar                80
4 Luis Suarez           84
5 dtype: int64

```

Just like in a dictionary and like above, we can refer to a data value in this series by its index (the equivalent to a key in a dictionary, if a index were a string).

```

1 print(composure['Lionel Messi'])

```

```

1 94

```

By now, you should have also noticed that when we print a full series, we also have a little line at the bottom that starts with `dtype`. This tells us the type of data that is in our series. If all of the datatypes in a series are the same, the `dtype` will represent that. Pandas tries to store data in the simplest method possible, just like Python. So, if you pass it in an integer, it'll try to store that value as an integer, rather than a float.

Like a list, we can mix our datatypes between each element in a series. You can mix floats with integers, strings with floats, booleans with strings, and every other combination out there. However, when we mix datatypes, the `dtype` that is shown somehow has to represent all of the data. Because of this mixed data, Pandas will just return a datatype of "object." However, Pandas will maintain the datatype in that series element, meaning that filling an element with a string will keep it a string, even if the `dtype` of the entire series is listed as `object`.

```

1 mixeddtype = pd.Series(['spinach', 48, 9.1])
2 print(mixeddtype)
3 print(type(mixeddtype))
4 print(mixeddtype[0], mixeddtype[1], mixeddtype[2])
5 print(type(mixeddtype[0]), type(mixeddtype[1]), type(
    mixeddtype[2]))

```

```

1 0    spinach
2 1         48
3 2         9.1
4 dtype: object
5 <class 'pandas.core.series.Series'>
6 spinach 48 9.1
7 <class 'str'> <class 'int'> <class 'float'>

```

Observe how in the first output (where we print the entire series), we see that the `dtype` is of type `object`. This means that the elements inside of our series are mixed. However, the complex datatype is still a series. Just how we

grabbed data out of our series before, we can do the same by referring to individual elements by index number. We can also print the datatypes of individual elements by using the `type` function, where we see that all of the datatypes that we initially gave Python are maintained (string, integer, and float).

10.4 Dataframes

Now that we've seen series, we can begin to explore dataframes. A dataframe is yet another complex datatype, but it is among the most powerful of complex datatypes out there because of its flexibility and the sheer number of methods that exist to manipulate that data.

Most data comes to data scientists as a CSV file. Whether it's been cleaned or not, they tend to follow the same form.

id	var1	var2	var3
1	56	122	3
2	52	139	2
3	38	185	2
4	62	115	3

The general form behind most datasets is that you have some ID to keep track of your entries and a set of variables that hold the data for each entry. This wouldn't really fit neatly into any preexisting data structure, and while we could probably write our own class to handle this data, we don't have to, since Pandas comes with its own data structure that was designed to hold CSVs and other datasets: the Pandas *dataframe*.

A dataframe is made up of Pandas series (hence why we covered those first). Each column of a dataframe is composed of a single Pandas series, and the process of putting series side-by-side in a certain order creates a dataframe. However, there are two major things to note: because a dataframe is a composite structure, you cannot edit the index of one column without altering the index of all of the columns, and the column names do not correspond to variables as they do in a pure series. This means that in the above series, we cannot simply call the `var1` series independently of the dataframe. We instead need to call the column as a part of the dataframe.

You can think of a dataframe as a data structure that emulates the format of a two-dimensional spreadsheet. If you are familiar with other programming languages such as C++ or C#, then you can associate the dataframe with a two-dimensional array.

Pandas dataframes require you to use a certain data format in order for the data to be read and associated properly. When you import your data, you should set it up with your individual variables on row 1 and each of your trials or data entries indexed at column 1. This will allow Python to determine the datatype of an entire column. On the surface, this seems trivial, but making sure that your data is formatted correctly before you import it will mean that you'll be able to analyze it in a somewhat standard manner.

If your data isn't stored in the format that you required but there is some standardization to the format of the data, then consider using your already-known standard file reading and writing skills to read the file into memory,

then rewrite it in the format that you require.

10.4.1 Getting Data

The most common operation that you'll be doing with a Pandas dataframe is retrieving data. There are a few ways to retrieve this data: we can retrieve the entire dataset as a dataframe, typecast it to another type, or just get a subset of the data. For this section, we will cover two ways of getting data: entering it naively, and importing it from a CSV. We'll also briefly cover pickles.

Making a Dataframe Naively

Let's say that you want to create a new, empty dataframe, then put data into that empty dataframe. You might have several lists that you want to turn into a dataframe, or you might be reading data in from an input/output device that needs stored in a dataframe.

Pandas allows us to create a dataframe by calling the `.DataFrame()` method in the Pandas library, and by default, it returns an empty Pandas dataframe.

```
1 df = pd.DataFrame()  
2 type(df)  
3 print(df)
```

```
pandas.core.frame.DataFrame  
Empty DataFrame  
Columns: []  
Index: []
```

As you might have inferred, this is not that dissimilar for how we might have created an empty list or dictionary by calling the `list()` or `dict()` functions in Python. When we first create an empty dataframe, it is assigned no size in either the row or column direction. That means that, strictly speaking, it is a zero-dimensional object.

The way dataframes are structured make them conducive to adding rows over columns, just like how when we make a spreadsheet in Microsoft Excel or Google Sheets, we will create the columns, then fill rows with different data. Because of this, it is much easier to define the column-wise dimension of your dataframe than the row-wise dimension when we create the dataframe in the first place. We can do this by passing a list type into the `columns` argument of the `DataFrame()` method. This will tell Pandas to create columns with the names of the elements in the list that you passed in. It won't tell Pandas anything about the datatypes, but at least we'll have our column-wise dimensions for our dataframe.

For example, let's create a dataframe for some ice hockey data with the column names `name`, `goals`, and `toi`, which will stand for player name, number of goals scored, and the amount of time on the ice that they've spent, respectively.

We know what our columns names will be, so we can pass these in as elements of a list into the `columns` argument.

```
1 hockey = pd.DataFrame(columns = ["name", "goals", "toi"  
    "])  
2 print(hockey)
```

```
1 Empty DataFrame  
2 Columns: [name, goals, toi]  
3 Index: []
```

Check your commas!

Just a reminder to check the positions of your commas. This is just a regular list, and since we're passing in strings, our commas must fall outside of the string, otherwise Python will consider the comma to be part of the string with no demarcation between elements, which will result in a syntax error.

Our dataframe is still empty because there isn't any data in the row-wise dimension, but we now see that we have three columns when we print out the dataframe.

Reading a CSV

Pandas has its own method for reading in a comma-separated values, or CSV, file. Using the Pandas method will allow you to put the data directly into a Pandas dataframe, rather than having to shoehorn it into a Python datatype, then typecast it to a Pandas dataframe.

Save time, use a package!

Pandas is so well universally utilized at this point that there is almost certainly a package out there for the type of data that you want to import, whether it's a R dataset (.rda or .rds), Excel worksheet (.xls or .xlsx), or OpenDocument spreadsheet (.ods). If your data is not in an easy-to-read binary file (as opposed to a plaintext file, like a CSV or TSV (tab separated values) file, consider finding a library in Pip to open the file for you and put the values into a Pandas dataframe.

The built-in Pandas method for reading a CSV file is `.read_csv()`, and it returns a Pandas dataframe. The default arguments for the `read_csv()` method are to read a CSV file that was created from a Microsoft Excel, Libreoffice Calc, or Google Sheets file. All three pieces of software create CSV files that adhere to some form of "typical" (although there is no standard for CSV files). For example, some CSV files may use different demarcations for strings, cells, headers, and any number of other changes.

- The separator default is a comma `,`, though some software separates cells using a tab (in the case of TSV files), period, or some other character.
- The header default is to infer whether there is one. Pandas will look at the datatypes of the potential column and evaluate what datatype it is compared to the datatype of the potential header. If the datatypes match with the header, it will infer that you have no header, but if all of your header row is all strings, Pandas will likely infer that you do have a header.
- Pandas will assume that you have no indexing column and it will make its own for you.

Because these are the default arguments for the `read_csv()` method, we don't need to explicitly set these arguments when we use the method. In fact, we only need to pass in one argument: the actual CSV file itself.

Because this is a required argument, we don't even need to specify the position of the argument in the method call. We can instead just pass in the location of the file as a string in the first position of the method call. Let's say that my file was called `skaters.csv` and it was located in the current working directory. We could just run the following line to put `skaters.csv` into a dataframe called `skaters`, then print the head of the `skaters` dataframe.

```
1 skaters = pd.read_csv("skaters.csv")
2 print(skaters.head())
```

	rank	player	playerid	...	floss	fopct
0	1	Calen Addison	addisca01	...	0	0.0
1	2	Andrew Agozzino	agozzan01	...	2	33.3
2	3	Jack Ahcan	ahcanja01	...	0	0.0
3	4	Sebastian Aho	ahose01	...	268	52.7
4	5	Sebastian Aho	ahose02	...	0	0.0

[5 rows x 62 columns]

Where am I?

To find what your current working directory is, you can just run the `pwd` command in the Python shell. `pwd` stands for "print working directory."

Getting Data in Subdirectories

If your CSV file is in a subdirectory of your current working directory, you can just use slashes to indicate subdirectories. For example, if `skaters.csv` were a file inside of the `data` directory, which was inside inside of `nhlskaters`, which was inside of `Downloads`, and `Downloads` was my current working directory, I could just pass in `"nhlskaters/data/skaters.csv"`

From the Top

Sometimes, it's easier to just give the entire file string. On Unix-like systems, this is easy: just preface the first directory with `./`. Your home directory is located in `/home/yourname/`. On Windows, start with `C:/` instead. Your home directory is located in `C:/Users/yourname/`.

Reading a Pickle

Pickles are a method of storing a data in non-volatile memory that represent an entire Pandas object. When a Pandas object is pickled, it is stored and recalled exactly the same at the time of pickling. Consider some of the ways that we can fail to read a CSV file. If the data were created with tabs instead of commas, it would be possible to read the data incorrectly. Plus, we can't tell Pandas that the first column is an index, meaning that anyone that imports the data down the line might create a new index, thus breaking your code. A pickle avoids these problems by storing the object as a whole, including indices, column names, and other metadata. This data is read when someone attempts to read the pickle, and the process of reading the pickle recreates the object exactly as it existed from whoever wrote the picklefile. Essentially, writing a pickle freezes an object in time, and anyone who uses that pickle down the line

will get that object exactly as it stood when it was frozen.

10.4.2 Adding Data

Consider the table presented in section 10.2: it looks like a table! With that view comes the advantage that we can index this table by rows and columns.

Recall

We first saw numerical indexing when we looked at lists. Remember that in Python (and in Pandas), indices start at 0, not at 1!

10.4.3 Removing Data

Other times, you'll want to permanently remove data. When we want to remove data, we need to choose exactly which data we want to remove, and this is called data selection or data location. Pandas has its own dedicated methods for data selection, including `.at`, `.iat`, `.loc`, and `.iloc`. Once you know how to select the cell that you want, it's much easier to remove that data, and it also helps you manipulate dataframes in general.

10.4.4 Modifying Data

When we work with data, we often have a need to change individual cells. Perhaps the data is incorrect or the type was created incorrectly (for example, it's a string instead of a float). This is where dataframe data modification comes into play.

10.4.5 Transforming Data

What if we don't want to edit just one column, but we'd rather change the entire shape of the data? This is called *data transformation*.²

²Note that the term "transformation" means something different here than it does in traditional statistics. While in statistics, transformation means to alter a variable to change its value by using some function (like taking the square root) in order to fit some assumption (like normality), we take "transformation" to mean changing the shape of how the data are represented in the dataframe.

10.5 Cleaning Up Data

One of the major strengths of Python is its ability to clean data. The very structure of Python means that it's great at very program-oriented tasks, such as procedurally cleaning and sorting data.

The simplest of data cleansing tasks is ***data normalization***. Normalization is the process of making all of the data of a certain variable into a similar form. For example, if you had a variable called "age" that contained both integer and floating point values, then you might want to normalize the data so that all of the data is an integer (if you only care about the integer value of the age) or so that all of the data is a float (if you need more specificity).

Data normalization can clarify confusing things in your data. Let's go back to our "age" variable. Let's say you had the value 7 as one of the values. Does this mean 7 years exactly or some time between 7 and 8 years? Rather, we could specify that the value should be 7.0 - this much more clearly indicates that we're looking at exactly 7 years, not the latter.

10.6 Calculating Summary Statistics

This book isn't really designed to be a statistics book, but what we can do is go over some basic statistics concepts and apply them to our Pandas and Python skills. If you want to do more with statistics, you should go take a statistics course.

This is not a stats book!

This textbook is not a statistics book. This is a Python programming book. We cannot possibly cover all of the details of these statistical tests and models in this one chapter - there are entire textbooks dedicated to this topic. Rather, this book should help you translate your skills in R, SAS, or SPSS to Python.

10.7 Basic Hypothesis Tests with statsmodels

This is not a stats book!

This textbook is not a statistics book. This is a Python programming book. We cannot possibly cover all of the details of these statistical tests and models in this one chapter - there are entire textbooks dedicated to this topic.

Alone, Pandas and Python make quite the deadly duo, but add in statsmodels and scipy and you've got yourself a Triforce of Stats! The statsmodels library adds many statistical tests that can be used to evaluate a dataset. Again, this isn't a statistics book, so we won't delve into the theory and details of the tests, but we can cover what the tests do and how to run the tests in R. For more detail, we recommend you check out the OpenIntro Statistics textbook (or just take a statistics class!).

We'll cover three statistical tests: the Z-test for proportions, the T-test for differences in means, and the Kruskal-Wallis test for differences in medians.

10.7.1 Z-Test for Proportions

To run a Z-test for proportions, we can use the `proportions_ztest()` method in the `proportion` module in the statsmodels library.

The purpose of the Z-test for proportions is to determine whether the proportions between two groups are the same. For example, suppose we wanted to test the idea that out of a sample of 50 men and 50 women, the proportion of men whose favorite Pokémon is Pikachu is the same as the proportion of women whose favorite Pokémon is Pikachu.

10.7.2 T-Test for Differences in Means

To run a T-test for differences in means, we can use the `ttest_ind()` method in the `weightstats` module in the statsmodels library.

10.7.3 Kruskal-Wallis Rank Test for Differences in Medians

To run the Kruskal-Wallis test for differences in medians, we can use the `kruskal()` method in the `stats` module in the scipy library.

10.8 Building Basic Models

This is not a stats book!

This textbook is not a statistics book. This is a Python programming book. We cannot possibly cover all of the details of these statistical tests and models in this one chapter - there are entire textbooks dedicated to this topic.

A big part of statistics is models. Statisticians use models to evaluate their data all the time, and it's something that Python and Pandas are quite good at doing.

For this book, we'll cover two models: the simple linear regression model, and the logistic regression model.

In order to understand why we're doing any of this, we must first understand what a model is. Consider a dataset with a few columns. One of these columns is our response variable, and the other columns are our indicator variables. If we just look at this one dataset, we know the data that was collected, but how does this generalize? That is, how can we make a prediction for our response variable given a new situation? This generalization is called a model.

As a more concrete example, consider a dataset of 25 high-school age soccer players in a specific league. We might have the variables speed, agility, age, and goals. Given the data, we can build a simple linear model to predict how many goals a player might score given their speed, agility, and age. This allows us to generalize the dataset (with its real data) to the entire league of, say 200 player, even though they might not have been sampled (assuming that our dataset is representative of the league).³

³This is a dramatic simplification of the process of making a model. You should refer to a statistician for more details.

10.8.1 Simple Linear Regression

10.8.2 Logistic Regression

10.9 Basic Graphs with Matplotlib

This is not a stats book!

This textbook is not a statistics book. This is a Python programming book. We cannot possibly cover all of the details of these statistical tests and models in this one chapter - there are entire textbooks dedicated to this topic.

The most glamorous part of statistics is probably the graphs. So far, almost everything that we've done has been done in the console, and while it's great for data density, it doesn't look great if we're being perfectly honest. However, we're about to leave the realm of the console and start to play with a new library: Matplotlib. This library will introduce us to tools that will open windows with graphs, since trying to view graphs in a terminal just isn't fun.

Matplotlib's Etymology

Matplotlib was derived from MATLAB as a library for Python designed to look like MATLAB. It stands for MATlab PLOtting LIBrary (think: mat-plot-lib).

Matplotlib is an incredibly complex and powerful library that allows you to create plots that are both static and animated! However, we'll only be looking at static plots.

Evaluation Copy

Chapter 11

Requests, Web Scraping, and BS4

As we begin to work with Pandas and data, so will the need to scrape data emerge. ***Data scraping*** is the act of pulling data off of a webpage without having to manually copy that data over. Instead, scraping data allows you to automate the process for a relatively low investment. Since you are now familiar with Pandas, you can use the full power of its data structures, plus the power of Beautiful Soup to scrape webpages efficiently.

11.1 Webpage Structure

To understand how we can extract data out of a webpage using web scraping, we first must understand how to read a webpage and how it is constructed. Webpages are much more than the rendered version that you see when you click on a link or navigate to a website. Webpages are actually another type of code called **HTML**, or HyperText Markup Language. HTML is not a *programming* language persay, since it does not support any of the features of a high- or low-level language. Instead, it is considered a markup language, which uses textual elements to dictate how something should be rendered by a web renderer. A HTML renderer is equivalent to the Python interpreter: it takes the code that you write and turns it into a pretty rendering. However, you can still refer to HTML as *code* or as *markup code*. How HTML is rendered is not dissimilar to how you can use markdown language in your .

Webpages can be constructed in several different ways, depending on the developer. However, most of the data that you will be scraping comes from HTML tables, and this is what we will focus on in this textbook.

11.1.1 HTML

The most obvious part of is its nested nature. Objects in HTML are nested inside of larger objects, and these larger objects can dictate how a sub-object is rendered. HTML describes the structure of pages using this nesting concept.

```
1 <html>
2   <head>
3     <title>My Webpage</title>
4   </head>
5   <body>
6     <h1>Star Wars</h1>
7     <h2>Episode 4</h2>
8     <p>Did you hear that? They've shut down the
        main reactor. We'll be destroyed for sure.
        This is madness! We're doomed! There'll be
        no escape for the Princess this time. What's
        that? Artoo! Artoo-Detoo, where are you?
        At last! Where have you been? They're
        heading in this direction.</p>
9     <p>I want her alive! There she is! Set for
        stun! She'll be all right. Inform Lord
        Vader we have a prisoner. Hey, you're not
        permitted in there. It's restricted.</p>
10    <p>The circle is now complete. When I left you
        , I was but the learner, now I am the
        master. Only a master of evil, Darth. Your
```

```

11         powers are weak, old man.</p>
12     <h2>Episode 6</h2>
13     <p>Command station, this is ST 321. Code
        Clearance Blue. We're starting our approach
        . Deactivate the security shield. The
        security deflector shield will be
        deactivated when we have confirmation of
        your code transmission. Stand by...</p>
14     <p>Let's go back and tell Master Luke. Tee
        chuta hhat yudd! Goodness gracious me!
        Artoo Detoowha bo Seethreepiowha ey toota
        odd mischka Jabba du Hutt. I don't think
        they're going to let us in, Artoo. We'd
        better go.</p>
15     <p>Do they have a code clearance? It's an
        older code, sir, but it checks out. I was
        about to clear them. Shall I hold them? No.
        Leave them to me. I will deal with them
        myself. As you wish, my lord.</p>
16 </body>
</html>

```

In the above markup code, take a look at the tags, written in blue. The tags include things like `<p>` and `<body>`. Notice how all of the tags start with a `<` and end with a `>`. Also observe how every tag that is created has a corresponding tag at the end. This is called a closing tag, and it can be distinguished because the first character after the opening left angle bracket `<` is a forward slash: `</body>`, `</p>`, and `</h3>`. All of the text is shown in black.

The above code renders to the following:

Star Wars

Episode 4

Did you hear that? They've shut down the main reactor. We'll be destroyed for sure. This is madness! We're doomed! There'll be no escape for the Princess this time. What's that? Artoo! Artoo-Detoo, where are you? At last! Where have you been? They're heading in this direction.

I want her alive! There she is! Set for stun! She'll be all right. Inform Lord Vader we have a prisoner. Hey, you're not permitted in there. It's restricted.

The circle is now complete. When I left you, I was but the learner, now I am the master. Only a master of evil, Darth. Your powers are weak, old man.

Episode 6

Command station, this is ST 321. Code Clearance Blue. We're starting our approach. Deactivate the security shield. The security deflector shield will be deactivated when we have confirmation of your code transmission. Stand by...

Let's go back and tell Master Luke. Tee chuta hhat yudd! Goodness gracious me! Artoo Detoowha bo Seethreepiowha ey toota odd mischka Jabba du Hutt. I don't think they're going to let us in, Artoo. We'd better go.

Do they have a code clearance? It's an older code, sir, but it checks out. I was about to clear them. Shall I hold them? No. Leave them to me. I will deal with them myself. As you wish, my lord.

Elements are usually made up of two tags: an opening tag and a closing tag. Each tag tells the renderer something about the information that sits between each of the tags. You can think of a tag like a container. The material that falls between the opening tag and the closing tag is put inside of the container that the tag created, and it is rendered according to the rules that the tag dictates.

Some common tags include:

Tag	Description
<html>	Encloses the entire HTML document
<head>	Encloses special document details that are not rendered
<body>	Encloses the actual rendered document
<h1> through <h6>	Headers 1 through 6
<i>	Italicize
	Bold
<a>	Hyperlink
	Image
	Unordered (bullet pointed) List
	Ordered (numbered) List
	List element
<table>	Encloses a table's contents
<thead>	Encloses a table header
<tbody>	Encloses a table body
<tr>	Table row
<td>	Table data, or one cell in a table

Not all elements have two tags, and that's okay. However, these are special cases. For example, the `<input>` and `
` elements only have one tag: their opening tag. This is because all of the information that is needed to render the tag are included in the tag's *attributes*. HTML tag attributes give additional information to the renderer about how the tag should be rendered, what the tag is, and if there's any data that needs to be dealt with. Attributes can be placed on any element, regardless of whether the tag has a closing tag or not, but they can only be placed on *opening* tags, never on *closing* tags.

There are some common attributes, including `id`, `class`, and `name`. However, as you can see in the above HTML code snip, you don't strictly *need* to include any attributes.

11.1.2 CSS and JavaScript

This is not a web design course, but it is important to understand how we get from the very simple HTML documents presented above to the websites of today.

CSS stands for Cascading Style Sheets, and it is responsible for styling the webpage. By providing specific rules on the size, color, shape, and placement of different elements, it is possible to create pretty webpages. Essentially, it's like putting a Snapchat filter on your boring HTML code. The HTML code provides the *substance*, and the CSS provides the *style*.

CSS can be placed in several ways: inline, in the document, or in a dedicated style sheet. If there are conflicting rules, the renderer should look at inline CSS first, then document CSS, then at the dedicated style sheet.

Inline styles are put in the HTML attribute `style=`. Inside of the style

attribute, you can write CSS code that will apply *only* to the element that the styling is written on, as well as any subelements. For example, if you applied a style attribute to a `div`, which typically acts as a container to enclose other elements, the style attributes on that `div` would trickle down to everything inside of it. However, if that `div` were inside of another `div`, it would only apply to the inside `div`, not the parent.

Minification

When you write inline CSS, you are doing something called "minification". Unlike Python, CSS doesn't rely on new lines to tell when one statement has ended and another has begun. Instead, it uses semicolons. So you can just stack a bunch of style rules next to each other, separated by semicolons.

Document CSS is written at the beginning or in the middle of an HTML document (in HTML5). Document CSS only applies to the document that the CSS is written in, but it applies to the entire document. Document CSS is placed inside of special `<style></style>` tags, and it resembles what might be placed in a dedicated CSS document. In our actual document, we refer to the rules that we wrote in our `<style>` tags using another HTML attribute: `class=`.

Not all classes are created equal

Classes in HTML are very different from classes in Python! Be careful to not confuse the two.

In CSS, every group of rules is placed inside of a CSS class, which can be applied to HTML elements using the `class` attribute. If all of the elements on your page are styled in a similar way, it might be possible to select what you need using the `class` attribute.

11.1.3 The requests Library

In order to get the HTML into Python, we need to use a special library that can make a web request to a web server. The web server responds with what has been requested. In fact, whenever you use a website in a standard web browser, you are making requests whenever you click. There are two types of requests that are of concern to us: `POST` and `GET` requests.

`POST` requests are used to send data to a remote server. If you imagine a web server as a bulletin board, a `POST` request is asking for permission to put

something on the bulletin board. The web server might deny the request, but at the very least, it must acknowledge that it received the request.

GET requests, as their name suggests, get data from a remote server. In our bulletin board analogy, it's equivalent to asking someone to read a listing on a bulletin board out loud to you. Again, the web server might deny the request, but it must acknowledge that you asked to get data.

Common pitfall: authentication

Some websites require authentication. Whether this is through an username-password combination or an authentication token, this should be checked if you're having issues getting or posting data. A basic test to see if authentication is required is to just open a new Incognito, InPrivate, or Private window in a web browser, then try to directly access the resource by pasting its URL into the address bar.

With that, it might seem like we'd only want to use GET requests. However, the inner workings of modern web frameworks are much more complicated, and they often involve asking for specific amounts of data. It's a negotiated process: you might ask for the format of the data, the web server responds with the format, you then ask if you need permissions and the web server responds with a yes, so you send the authentication token and the web server asks you what data you want, you finally send a request for what you want and the web server responds with the data. In this highly simplified scenario, we've already placed eight requests. Modern webpages might involve dozens of requests!

The `requests` library abstracts all of this for us. We don't need to know all of the details of which requests to place and what kind of data to expect in response, since `requests` already does this for us.

11.1.4 Parsing with Beautiful Soup

Now that we have our HTML in Python, we need to know how to break it apart. We could break the structure apart in a naive method, by writing a series of `if` statements and iterating over the contents of the HTML until we find tags. However, why would we do this when we can use Beautiful Soup instead?

Beautiful Soup's job is to break apart and parse webpages in a way that is easy for you to understand. As opposed to creating a parser and writing `if/else` statements, Beautiful Soup comes with many tools that are incredibly powerful. The power of Beautiful Soup lies in its ability to pull data out of extraordinarily complex webpages in just a few lines of code.

11.1.5 APIs

An *application programming interface*, or **API**, returns formatted data based on a request that is made to it. While on the surface, an API call looks like any other web resource call, its return is very different, typically an XML or JSON form. You can experiment with open APIs using an API testing tool, like Postman. Postman can also help you form authentication strings, if your API of choice requires authentication.

If an API will give you the information you need, use this instead, since it's much faster to develop for.

There is an API for everything, and many of these APIs are free (though some still require you to get a free authentication key)! Here, we'll look at Animechan, an API that returns a random quote from a list of animes. Animechan doesn't require any authentication, so we can simply make a request to <https://animechan.vercel.app/api/random>. If you were to just type this into a web browser, you'd get a response in the form of JSON, which can be parsed using the JSON library in Python. We'll cover this in the next section.

11.1.6 Reading a JSON response

Consider the Animechan API in the previous section. Let's make an example request. Based on the API documentation, we know that the API returns data in JSON form, so we can expect this in our response.

```
1 import requests
2 response = requests.get("https://animechan.vercel.app/
    api/random").json()
3 print(response)
```

Here's an example response (responses are random, but you should get a response with the same form).

```
{"anime": "Naruto", "character": "Kiba Inuzuka", "quote": "
Akamaru, what's wrong boy? Have you forgotten my
scent? We've always been together haven't we? We
grew up together. Akamaru please, somewhere in
there, there has to be a part of you that remembers
. Show me that you remember. AKAMARU! Forgive me.
Can you? I know that I've brought you nothing but
pain and suffering. I broke my word. I swore I'd
always protect you. Akamaru I'm sorry. Sorry I wasn
't a better master. I'm here. Here for you. Forever
."}
```

By default, the `requests.get()` method returns an object of type `Response`, which isn't terribly useful to us on its own.

Observe in the request how we added the `.json()` method to the end of the `requests.get()` statement. Again, because we know that the API returns its data as a JSON object, we can just tell Python that we want to automatically typecast the data into a Python dictionary. The requests library will process the response as a JSON and parse it into a Python dictionary. This saves us a step down the line, explicitly parsing a JSON string into a Python dictionary. However, we still included this as an option below, since sometimes, JSON string parsing using the requests library doesn't work as we might expect.

We printed the data in its dictionary form, hence why we got the curly braces, keys, and values. Just like with any other Python dictionary, we can use all of our regular methods on dictionaries on this one. In our case, the dictionary is stored in a variable `response`.

```
1 print(response["anime"])
2 print(response["character"])
```

```
Naruto
Kiba Inuzuka
```

11.1.7 Parsing JSON

This is in a form called JSON, or JavaScript Object Notation. It looks similar to a Python dictionary, and it also supports nesting. In fact, if you were to bring this into your Python script as a JSON-formatted string, you could easily parse it into a Python dictionary using the `json` library.

The `json` library allows us to typecast JSON data into a list or dictionary, depending whether the JSON object has keys or not. If the JSON object doesn't have any keys, then the `json` library will typecast the object into a Python list with an auto-assigned index. If the JSON object does have keys, like our Animechan response, then the `json` library will typecast the object into a Python dictionary using the keys and values from the JSON object.

To do this typecasting, we can use the `load()` or `loads()` method. They are different in the type of object they are expecting. For our purposes, we will almost always use the `loads()` method. The `s` in `loads()` stands for string, and this particular method is used to parse a string containing JSON data into a Python dictionary or list.

Let's make another request, but this time, let's specify that we want the string form of the data, rather than letting the requests library parse the JSON object for us. As we did when we were getting the contents of the HTML webpage, we can specify this by specifying that we want the `text` attribute from the `response` object.

```
1 response = requests.get("https://animechan.vercel.app/
    api/random").text
2 print(type(response))
```

```
3 print(response)
```

```
str
{"anime":"Death Note","character":"Light Yagami","
  quote":"This is the first time in my life that I've
  been provoked to hit a woman."}
```

This time, our response is left in the string form that it was given to us by the API. We can parse this using the `loads()` method in the `json` library. `loads()` typically only takes one argument, the string with the JSON-formatted string.

```
1 response = json.loads(response)
2 print(type(response))
3 print(response)
```

```
<class 'dict'>
{'anime': 'Death Note', 'character': 'Light Yagami', '
  quote': 'This is the first time in my life that I'
  ve been provoked to hit a woman.'}
```

We can now see that the type is no longer `str` but is a dictionary object instead.

The key with the `load()` and `loads()` methods is that they work on any JSON-formatted data, not just data that comes from a web request. For example, consider if we imported a JSON as a file and read it into a string variable, as we showed in Chapter 8. We could then use the `loads()` method to parse this string into a Python dictionary.

Evaluation Copy

Labs

This chapter contains labs and rubrics for each lab. Your instructor might assign certain labs to you and not others.

Any modifications that your instructor makes should take precedence over the lab provided here. You should take care to follow your course's style guide, if your instructor has one. This style guide should give you important information regarding naming conventions, line spacing, whitespace, and other notes like these. Remember to follow your course's style guide!

Lab 1: Error Messages

Description

An important part of programming is being able to see how your changes change the behavior of the program. For this exercise, you'll be deliberately making errors and figuring out what lines of code result in these issues.

Task

Begin with the following code. Confirm that it works correctly.

```
1 import pandas as pd
2 def main():
3     ser = pd.Series()
4     n_ethereum = int()
5     value_per_coin = float()
6     total_value = float()
7     n_ethereum = int(input("Enter the number of
8         Ethereum in wallet."))
9     print('You entered: ' + str(n_ethereum) + '\n')
10
11     value_per_coin = float(input("Enter the dollar
12         value of one Ethereum."))
13     print('You entered: ' + str(value_per_coin) + '\n')
14
15     total_value = value_per_coin * n_ethereum
16     print('Total value in wallet is ' + str(
17         total_value) + ' dollars.')
18
19     ser[0] = total_value
20
21 if __name__ == "__main__":
22     main()
```

Attempt the following tasks. In between each task, revert your program back to its original state.

- Put an extra space in between `pandas` and `as` in line 1.
- Remove `as pd` from line 1.
- Remove the opening quote from line 7 inside of the `input` function.
- Replace the opening quote on line 7 with a single quote.
- Remove the backslash `\` on line 8.

- Remove the `[0]` on line 16.
- Remove the underscores `_` on line 18.

For each task, you will report on what happened and why you think that happened.

Turn In

You will submit a lab writeup in a Jupyter Notebook. Your lab writeup should contain the following sections.

- Introduction: What were you given? What was the purpose of the lab?
- Code Description: What does the code (as given to you) do when it is executed?
- Tasks: What did each of the items for the list of tasks do? What happened when you made the change as directed? Why do you think that happened?
- Conclusion: What did you learn during this lab?

Lab 2: Payroll Calculation

Description

An employee is paid at a rate of \$20.68 per hour for the first 40 hours worked in a week. Any hours over that are paid at the overtime rate of one and one half times that. From the worker's gross pay, 6% is withheld for social security tax, 14% is withheld for federal income tax, 6% is withheld for state income tax, and \$12 per week is withheld for union dues. If the worker has three or more dependents, then an additional \$35 is withheld to cover the extra cost of health insurance beyond what the employer pays.

Your job will be to write a program that calculates the values above, given the hours worked and the number of dependents.

Task

Write a program that will prompt the user for the following information.

- The number of hours worked in a week
- The number of dependents

The program will then output the worker's gross pay, each deduction amount, and the net take-home pay for the week. Write your program so that it allows the calculation to be repeated as often as the user wishes.

All decimal point numbers that represent money must be outputted with two digits after the decimal point - no more and no less. For example, print 2.50 instead of 2.5.

Turn In

You will submit a lab writeup in a Jupyter Notebook. Your lab writeup should contain the following sections.

- Introduction: What is the program supposed to do? If you were provided with any code, where did it come from?
- Code Description: In your head, break down your program into logical sections. Then, write a subheading for each of your code's sections, include the code (in a code block) what it does, and why you included it.
- Issues: Did you experience any issues while writing this lab? If so, what issues did you run into? If not, what are some issues that you could foresee another student making, and how did you avoid these issues? If you were creating this lab, how would you change or improve it?
- Completed Program: Include one big code block that contains your program in its entirety.

- Test Runs: Provide the entire output from your program when you ran it using the trial data from below.
- Conclusion: What did you learn during this lab? How did you apply some of the skills that you've learned to this lab?

Trial Data

Trial	Hours	Dependents
1	15	1
2	40	4
3	53	3
4	2	5

Sample Output

This sample output is provided to guide you to your solution. You should follow the instructions provided to include all of the functionality that is shown below.

```
This program will ask you how many hours you worked ,
and calculate your
taxes , dues , gross pay , and net pay .
```

```
How many hours did you work? 20
How many dependents do you have? 1
```

```
Regular hours: 20.00 (at $16.68 an hour)
Overtime hours: 0.00 (at $25.02 an hour)
Total hours: 20.00
Gross pay is $333.60
Social Security tax: $20.02
Federal taxes: $46.70
State taxes: $16.68
Union Dues: $10.00
Total Deductions: $93.40
Net Pay: $240.20.
```

```
Would you like to calculate another week's pay? (y or
n) y
```

```
How many hours did you work? 48
How many dependents do you have? 4
```

```
Regular hours: 40.00 (at $16.68 an hour)
Overtime hours: 8.00 (at $25.02 an hour)
Total hours: 48.00
```

Gross pay is \$867.36
Social Security tax: \$52.04
Federal taxes: \$121.43
State taxes: \$43.37
Union Dues: \$10.00
Family Health Insurance: \$35.00 (additional insurance
premiums for your family)
Total Deductions: \$261.84
Net Pay: \$605.52.

Would you like to calculate another week's pay? (y or
n) y

How many hours did you work? 3
How many dependents do you have? 4

Regular hours: 3.00 (at \$16.68 an hour)
Overtime hours: 0.00 (at \$25.02 an hour)
Total hours: 3.00

Gross pay is \$50.04
Social Security tax: \$3.00
Federal taxes: \$7.01
State taxes: \$2.50
Union Dues: \$10.00
Family Health Insurance: \$35.00 (additional insurance
premiums for your family)
Total Deductions: \$57.51

Your dues and insurance obligations outstripped your
pay by \$-7.47.

Would you like to calculate another week's pay? (y or
n) n

Thank you for using this program.

Grading Table

Requirement	Possible Points
Correct output on required trial data	60
Dollar amounts have the right float value (2 decimal places)	10
Appropriate code formatting, good use of whitespace	5
Meaningful variable names	5
Descriptive comments at the top	5
Descriptive comments to label sections	5
Jupyter Notebook is constructed well and with care	10
Total	100

Lab 3: Bad Programmer!

Description

For many programming projects in the real world, you'll be using or working with code that someone else wrote, rather than writing code from scratch. A good programmer will be able to intelligently break down someone else's code, and if you can read someone else's bad code, it'll be a piece of cake to read someone else's good code.

For this lab, you will be writing using terrible programming practices, then you'll be trying to decipher someone else's program, who has also used terrible programming practices.

Task

For this lab, you will need a partner.

On your own (without your partner), write a program that takes a student's name, year (as an integer), major(s), minor(s), and dormitory. Optional: make the majors and minors multiple choice questions. You can write this in any programming language you'd like as long as it's Python 3.6. Then, print a summary of that student using the format: Your name is [name] and you are a [year]. Your major(s) is in/are [major1] and your minor(s) is in/are [minor1 and minor2]. You live in [dormitory].

Your program should correctly choose whether to use the term "major" or "majors," "minor" or "minors," "is" or "are," and whether you need to use commas or an "and" for multiple majors or minors.

Now, here's the kicker. You know what good programming practices are. Now, break every good programming practice you can without actually breaking valid syntax. That means that your program should properly execute, but it shouldn't be readable by a human. Use bad indentation practices (without breaking Python), bad/nonexistent comments, confoundingly constructed if/then statements and loops, the wrong data structures, obfuscation, complexity and anything else to make your program difficult to read. The only thing you can't do is hide a file. All of your script must fall in one file. If anyone except for you can read your program, you have not done a good job.

Trade programs with your partner. You should've left them with a horrible mess that is virtually unusable. Your partner's job is simple: comment the code that you wrote, and write a summary of the code that you wrote, all without asking you any questions. This summary should contain things like what datatypes are being used for what variables, the logical flow of the program (like where decisions are being made), and how this code might be fixed (though you don't actually have to fix it).

Turn In

You will submit this program inside of a Jupyter Notebook. You should include both your's and your partner's script (unmodified) inside of code blocks. You should also include your partner's commented and annotated script, as well as your best summary of their code.

Trial Data

Trial	Name	Year	Majors	Minors	Dormitory
1	Alice Alexander	Sophomore	Economics	Statistics	Pless Hall
2	Ben Loch	Junior	Env. Science		McAlester Apartment
3	Sam Williams	Senior	Biology, Chemistry	Computer Science	Raphael Dormitory

Grading Table

Requirement	Possible Points
Correct output on required trial data on your script	20
Inappropriate code formatting, bad use of whitespace in your script	30
Poor variable names in your script	5
Poor commenting or no comments, other challenges to readability in your script	5
Excellent commenting of partner's code	10
Correct description and interpretation of partner's code	20
Jupyter Notebook is constructed well and with care	10
Total	100

Note: This means that you will *receive* points for writing bad (but syntactically correct) code yourself *and* for properly analyzing and breaking down your partner's bad code.

Lab 4:

Evaluation Copy

End Matter

Conclusion

With this book, you have only started to scratch the surface of what is possible with Python. It is an incredibly versatile and powerful programming language with many features that will prove useful to you as you embark further on your programming journey. Furthermore, it prepares you to learn other programming languages. If you decide to pick up a language like C++, Swift, JavaScript, PHP, or any other language, you'll immediately begin to notice similarities. Sure, the syntax is different, but the ideas and structures that exist in Python also exist in nearly every other programming language. Now that you know what a variable is, what the datatypes are, how a function works, and other concepts like these, you are well equipped to adapt this knowledge to new languages: a function in C++ fundamentally does the same thing as a Python function, even though they might look different.

Programming is useful for other reasons, too. For one, it teaches you how to be a critical thinker and problem solver. With your programming mindset, you have become tuned to hunting down and fixing issues and developing cohesive solutions to complicated problems, and those skills are increasingly important in our modern age of information. Even if you don't choose to continue programming formally, consider continuing to work on these skills, and consider using programming as a means to achieve great things.

Further Projects

As mentioned, we have only scratched the surface of Python's immense capabilities. Here are some projects to try and exercise your newfound Python skills, as well as some new things to explore.

- Code a Tic-Tac-Toe game in the Python console. You'll have to figure out how to make your program interactive.
- Learn how to use Turtle. Turtle is a graphics library that allows you to draw using Python.
- Learn how to use Tkinter (after Turtle). Tkinter will allow you to develop desktop applications using Python.
- Learn how to use Flask. Flask is a web-server designed for Python, and it'll allow you to flex your new HTML and CSS skills.
- Code a Towers of Hanoi simulation. This is a great way to learn how to use recursion.
- Learn JavaScript or R. All are interpreted languages (like Python), but they use a different syntax. Each of these languages have advantages and disadvantages, but either will make you a better programmer and problem solver.
- Learn Java, C, C++, C#, Swift, or Objective-C. All are compiled languages and they use a similar syntax, but they have some advantages over Python. Learning either will make you a much better programmer.

Sample Style Guide

This sample style guide is provided to you as a student or instructor as a framework for your own style guide.

Use descriptive names, even if it increases line length slightly. `count` is more descriptive than `c`.

In general, avoid using single character variable names, since they are often difficult to follow and read. Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), 'I' (uppercase letter eye), '1' (number one), or '0' (number zero) as single character variable names. Avoid using 'L' (uppercase letter el) when possible. In some fonts, these characters are indistinguishable from the numerals one and zero.

Capitalization Conventions

- Variables and objects: `camelCase` (`count`, `numRuns`)
- Functions: `snake_case` (`sum`, `sum_of`, `get_result`)
- Classes: `CapCase` (`GameScore`, `Runs`)
- Constants: `ALLCAPS` (`PI`, `FIELDLENGTH`)

Whitespace

Use whitespace wisely. Remember, whitespace takes the form of both horizontal whitespace (spaces and indentation) and vertical whitespace (blank lines). Both too much and too little whitespace make your source code difficult to read.

Leave one space around initializations and boolean operators.

```
1 runs = 1 # Good
2 if (runs >= 10): # Good
3 runs=3 # Bad
```

Observe how the equal sign in line 1 is surrounded by spaces. This is an example of space around initialization.

Also observe how the greater than/equal to sign in line 2 is surrounded by spaces without a space between components of the boolean operator. This ensures that the syntax is correct for the entire boolean operator (the `>=` is one unit, not a separate `>` and `=`) while still providing adequate whitespace. This is an example of space around a boolean operator.

Also leave space before and after comment demarcations, as shown in lines 1-3. The comment demarcation in Python is a `#`, and there is a space before and after.

Leave an extra space between function arguments. Do not leave an extra space before or after function parentheses.

```
1 atlRuns = GetRuns('ATL') # Good
2 ariWins = GetWins('ARI', 'away') # Good
3
4 bosRuns = GetRuns( 'BOS' ) # Bad, too much space around args
5 chiWins = GetWins( 'CHI', 'home' ) # Bad, too much space
    around args
6 dalWins = GetWins('DAL','away') # Bad, no space between args
```

Indentation

In connection with whitespace, make sure you follow indentation conventions for your language. Python enforces indentation, so make sure you use consistent indentation.

Indent using one tab, which should indent two spaces.

Indent anything nested, including function contents, logic statement bodies, loops, and nested objects (mainly arrays, lists, and dictionaries).

Do not put a space before a colon in a conditional or logic statement.

```
1 if (numRuns > 3): # Good
2     print("More than three runs!") # Good, two spaces of
      indentation
3 else : # Bad
4     print("Not more than three runs.") # Bad, inconsistent
      indentation (3 spaces)
```

Soft-wrap lines in your editor, not by manually splitting a line into multiple lines. Not everyone's editor window size and font size is the same as yours.

Errata

No errata exist for the previous edition.

Evaluation Copy

Evaluation Copy

Index

- alias, 154, 155
- application programming interface, 181
 - Animechan, 181
 - API, 181
- arithmetic, 56, 66, 74, 75
 - exponents, 76
 - integer division, 77
 - modulo, 77
- basic input/output system, 14
- Boolean, 22, 38, 39
- characters, 28
 - chars, 28
- compilation, 16
- complex datatypes, 82
- concatenation, 32
- data normalization, 169
- data scraping, 175
- data transformation, 168
- datatype, 19, 81
 - dictionary, 81
 - list, 81
 - element, 85
 - set, 81, 98
 - string, 28
 - literal, 28
 - tuple, 81, 96
- errors
 - logical, 55, 57
 - syntax, 55
- escape characters, 33
- f-string, 70
- firmware, 14, 15
- float, 26
- floating-point, 26
- function
 - arguments, 126
- functions
 - arguments, 31
 - method, 88
 - return, 127, 128
 - scope, 121
 - global, 121
 - void, 128
- HTML, 176
 - attributes, 178
- human interface devices, 14
 - HIDs, 14
- index, 86
- input device, 14
- interpretation, 16
- interrupt, 152
- Jupyter Notebooks, 176
- memory, 13
 - primary, 13
 - secondary, 14
 - space, 13, 37
 - tertiary, 14
- Nonetype, 39, 90, 95
- object code, 16
- object-oriented programming, 134
- Pandas
 - dataframe, 163
 - series, 157
- programming languages, 15
 - high-level, 15
 - low-level, 15
- pseudocode, 17
- software, 15

- source code, 16
- statement, 7, 8
- strongly typed, 20

- typecast, 20, 86, 112

- variable
 - declaration, 37
 - initialization, 37

- weakly typed, 20
- whitespace, 7
 - line breaks, 7
 - spacing, 7

Sejin Kim '22 is a student at Kenyon College studying scientific computing. He serves as a lead tutor for introductory programming courses in the Department of Scientific Computing. He can be contacted at `kim3[AT]kenyon.edu`.

Evaluation Copy

Evaluation Copy

Version Number: 0.1.9023

License: MIT License

Citation: The suggested BibTeX entry is as follows.

```
@book{scientificpython,  
      title = {Python for Scientists, version 0.1.9022},  
      author = {Kim, Sejin},  
      edition = {Zeroth},  
      year = {2022},  
      address = {Gambier},  
}
```

Evaluation Copy

Python for Scientists

Sejin Kim

Python in Purple is an open-source Python textbook designed for introductory courses in computer science or scientific computing. As opposed to other textbooks, which only cover the fundamentals of coding itself, Python in Purple also provides students with good programming and development skills. Written for students at Kenyon College, it should also serve students at similar, small liberal arts institutions with its emphasis in Python as a tool in scientific computing.

This edition includes:

- Fundamentals of programming
- Good coding practices
- Python in scientific computing using Jupyter Notebooks
- Data manipulation with Pandas and plotting with Matplotlib
- Basic statistical tests with statsmodels
- Web scraping tables with Beautiful Soup 4

Version Number: 0.1.9023

License: MIT License