INTRODUCTION TO COMPUTING

David Joyner

1st Edition

Mc Graw Hill Education

Introduction to Computing

1st Edition

David Joyner

Mc Graw Hill Education

1 2 3 4 5 6 7 8 9 0 TBA TBA 19 18 17 16

ISBN-13: 978-1-260-08227-2
ISBN-10: 1-260-08227-X

Solutions Program Manager: Craig Bartley
Project Manager: Jennifer Bartell
Cover Photo Credits: © Design Pics/Darren Greenwood

Brief Contents

Unit 5: Object-Oriented Programming

UNIT 1

INTRODUCTION TO COMPUTING

CHAPTER 1.1

Computing

1. What Is Computing?

What is computing? If you ask a dozen different computer scientists, you'll likely get a dozen different answers. At its broadest level, computing is defined as anything that involves computers in some way, from designing the physical components that make up a single device to designing massive systems like the Internet that use computing principles.

In other words, computing is a massive field that touches almost every corner of modern society in some way. With such a massive domain… where do we start?

Introduction to Programming

Fortunately, effectively all of computing has a common foundation: programming. Programming is the act of creating instructions for a computer to carry out. Those instructions might be things like, "Add 5 and 3," "Fetch Google.com," or "Save my document." Chains of these commands create the behaviors of every single computing device you see, from your thermostat to the space station. That's what programming is: writing the commands, called "code," for a computer to perform.

Let's take a simple example of this. You're browsing the Internet and you see a link you'd like to follow—so, you click it. There are lines of code that translate that take that click and figure out what you clicked on based on where the mouse was located. There are lines of code that take the fact that you clicked a link and use it to send out a request to the Internet to retrieve the document. There are lines of code that monitor that retrieval, making sure that the document you requested exists and organizing it as it comes in. There are lines of code that take the document that you received and translate them into pixels on the screen so you can read it. Every stage of the process is governed by some code.

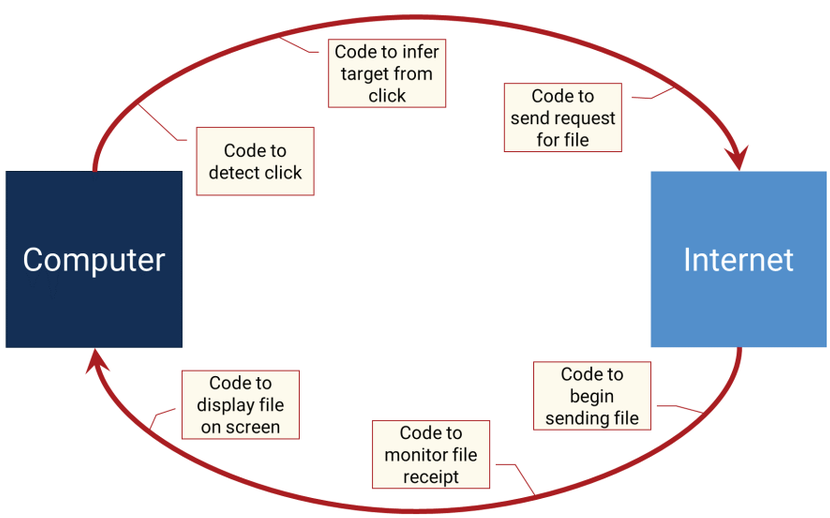


Figure 1.1.1

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Analyze the nature of computing (for the purposes of this introduction) in addition to constructing instructions for a computer to follow;
- Use the basic terminology surrounding computing and be able to correctly identify terms such as input, output, console, GUI, lines of code, compiling, and executing;
- Describe the value of Python as well as how to develop using Python;
- Work on the domain and complete different types of tasks on it.

Programming Is Everywhere

No matter where in computing you end up going, you'll likely be dealing with programming in some way. Even if you're not writing code yourself, you might be designing programs for someone else to code, or designing the hardware which will run code. Being able to program is like being able to speak a language. Just as you need to speak Spanish to communicate in Spain, you need to speak "code" to communicate in computing.

So, this Introduction to Computing aims to give you that language to communicate in the computing world. Our goal is for you to learn not only how to write code, but also know how to communicate in a community that uses code as its language. Just like learning to speak a new language is more than just memorizing vocabulary words, so also working in computing is about more than just understanding how to write singular lines. It's about understanding what writing code allows you to say and do.

Chapter Outline

This initial chapter is meant to provide you with the background necessary to start having these conversations about programming and computing. We'll cover some of the basic vocabulary you need to start reading about programming, like output and compilation. We'll discuss the general nature of different programming languages, their strengths and weaknesses. We'll discuss different places where we might see a program's output, especially the console or graphical interfaces. Finally, we'll discuss what to expect in the rest of the course, as well as the language and domain in which you'll be working.

2. Programming Vocabulary

In order to talk about programming, there are some basic terms we need to know. We'll cover a lot of vocabulary in context throughout this course as well, but there are a few terms we need to understand just to get started.

Programs and Code

We've already covered a couple of these. Code is commands given to a computer to order it to perform some task.

Line of code
A single instruction for the computer to perform.

A **line of code** is generally a single command. Very often, we'll talk in terms of individual lines of code and what each line does. In practice, we'll find a single line could actually set off a sequence of lots of other commands, but generally a single line of code is the smallest unit we're interested in dealing with at this stage.

Program
An independent collection of lines of code that serves one or more overall functions.

A **program**, for our purposes, is a collection of lines of code that serves one or more overall functions. This could be anything from calculating the average of some numbers to running a self-driving automobile. Programs are often what we're interested in building. A program is like a house and lines of code are like individual bricks.

So, when we talk about programming or coding, we're talking about writing lines of code to create programs that accomplish some tasks.

Input and Output

Input
Data that is fed into a program for it to operate upon.

Nearly every program we write is also largely defined by its relationship with its input and its output. **Input** is anything that we put into a program for it to work on, and output is what the program gives us in return. Usually we're not going to write programs that do the exact same thing every time they're used—usually we're going to write programs that process input in some way, providing **output** that corresponds to the input.

Output
What the computer provides in return after running some lines of code.

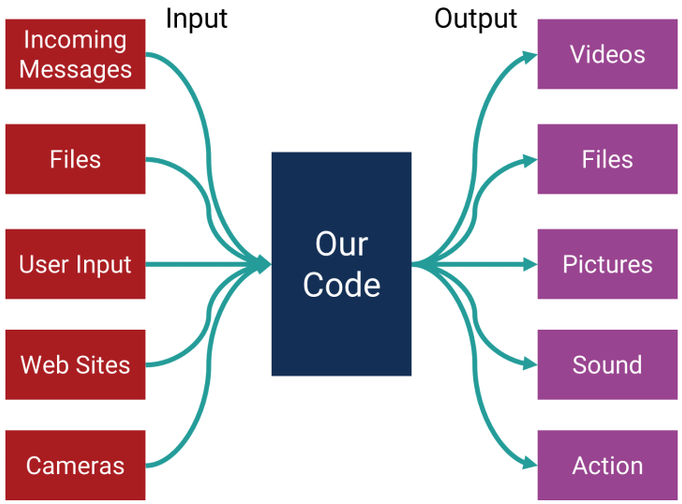


Figure 1.1.2

Input can come from a lot of places. Very often, we're dealing with user input. To take a simple example, think of a basic word processor like Notepad or TextEdit. The user types keys, and in return, the program shows the letters that were pressed. The input is the keys that the user pressed; the output is the letters shown on the screen. Everything a user does on a computer screen is user input, and anything a computer screen shows is output.

Input doesn't have to be from a human, though. When a web browser retrieves a website from the Internet, for example, the contents of the website would be the input, and the display of the site on the screen would be the output. When a word processor opens a file from your desktop, the file's contents would be the input into the program, and the display of the document would be the output.

Output doesn't just have to be to the screen, either. For example, when your phone receives an incoming call, the call information is the input into the phone, and the phone ringing is the output. Or, when you create a new document and press "Save" for the first time, the document contents that you've entered become the input, and the file that is saved is the output.

At a general level, the input into some code is whatever exists before the code is run, and the output is whatever the code produces as a result of running. When we write code, we'll even find that we'll constantly be dealing with input and output between different portions of our own programs. The output of some code that we write becomes the input into some other code.

Compiling and Executing

Finally, the last two terms you need to know before we even get started are compile and run. Compiling and running are two things we do to code that we've written to see if it's working the way we intend.

Compiling is like reading over code and looking for errors in the way we've written it. It's kind of like the proofreading you would do on an essay. You can just look at the text and see if there are problems with it, like misspelled words or comma splices. Code has more strict syntax than an essay, though, so we rely on other computer programs, called compilers, to do this for us. They read in the code and let us know what problems they find. If there aren't any problems, they produce programs that can be run.

Executing is then when the program is actually run. Just because some code compiled into a program doesn't mean it will actually do what we want it to do—it

Compile
To translate human-readable computer code into instructions the computer can execute. In the programming flow, this functions as a check on the code the user has written to make sure it makes sense to the computer.

Execution
Running some code and having it actually perform its operations.

just means that what we told it to do makes sense. For example, imagine we wrote a program that would add two numbers, but instead we accidentally put a subtraction sign instead of an addition sign. The code still makes perfect sense during compilation, it just does the wrong thing.

To use an analogy, imagine giving your friend directions for where to find a form in your office. You write the directions on a piece of paper and hand them to her. She reads over them and checks if they make sense. Perhaps she can't make out something you wrote, or wants extra clarification on a particular step. That's like compilation—she checks to see if the directions make sense before trying to carry them out. Then, when she's satisfied with them, she tries to actually carry them out. That doesn't guarantee she'll be successful, though: maybe the form isn't where you said it would be, or maybe one of the steps that made sense on paper doesn't make sense once she's in the office. That's like executing the code: actually carrying out the steps.

We should note that this description of compiling and executing is from the perspective of how you write code and build programs. In reality, compiling code actually serves a more significant set of purposes than this. Compiling translates the code that you write into the low-level types of commands that the computer actually understands. That level of detail is outside the scope of an Introduction to Computing class, however. For the programming you'll actually do, this definition of compiling and executing should be fine.

You might notice that compiling seems potentially optional. After all, your friend could go and try to follow your directions without ever reading them first. Compilation is more important under the full definition of what it includes, but you're right that it potentially could be skipped. We call languages that require compilation "static" or "compiled" languages, and languages that do not require compilation "dynamic" or "interpreted" languages. Nonetheless, even with dynamic languages, we often mimic the workflow of static languages. You likely won't encounter the differences between the two until much later in your computing studies.

3. Programming Languages

In order to write an essay, you must have a language in which to write it. You could write an essay in English, Japanese, Spanish, or Mandarin, but there must be a language. The same is true for programming: you must have a language in which to write. Just as different written languages have different syntax, different vocabularies, different structures, so also do different programming languages have different syntax, different vocabulary, and different structures.

There are dozens, even hundreds of programming languages out there, with many similarities and many differences. There are lots of ways to categorize programming languages. For example, static languages require a compilation step, whereas dynamic languages do not. High-level languages involve a great deal of abstraction away from the details of the computer like memory, whereas low-level languages require programmers to do more of these things manually.

Why do so many languages exist? Different languages are good for different things. When you're optimizing for performance, as you might with a visually complex video game or a highly complicated mathematical function, you might want to have more control over the details of how things run. If you're more interested in being able to design rapidly, you might be interested in a language that doesn't force you to think about those details.

But Why Do I Care?

But why am I telling you all this? You're just about to set out on learning your first language, why do you need to know about all the others? The reason for this is that as you learn your first language, it's useful to keep in mind where that language sits in the broad spectrum of computing: what it's good for, when it's bad.

Most importantly, though, this is important because this is an Introduction to Computing, not simply an Introduction to Programming or an Introduction to Programming in a certain language. While we talk a lot about programming, it is because we must learn the language we use to discuss computing. However, simply knowing the language is only half the task. The other half is to understand the nature of computing as a whole and its relationship with programming. Toward that end, as we go forward, we will revisit some of the concepts that may differ between languages in order to paint a broader picture of computing as a whole than simply the language you choose to learn.

4. Console vs. GUI

We've discussed how the programs we write can be largely characterized by their input and output. While programs can have many kinds of output, as we learn to write code, we'll deal a lot with output we design specifically to help us understand how programs work. The easiest way to do this is by stripping out as much as possible so that we can focus entirely on what our programs are outputting.

In your experience using computers, you most often interact with Graphical User Interfaces (GUIs). These are systems that involve any kind of output beyond plaintext, and any kind of input beyond pure text entry. These are useful applications, but they are very complicated. As we learn to program, we'll start with console-based programs, and work up to graphical programs.

The Console

Console
An output medium for a program to show exclusively text-based output.

There is a chance you might have used a **console**-like interface before. These are similar to command-line interfaces, like the Terminal on a Mac or the Command window on a PC. Generally, these are methods for input and output based exclusively on plain text: no graphics, no layouts, no input mechanisms besides the keyboard. These are common starting points for learning to program, and most common languages can be written exclusively for the console.

For our purposes, this means that we will start by creating programs that only output text. We can use that text to evaluate how well the program is performing. For example, we know what output we would expect for some input into the program. When the program finishes running, we can print that output and see if it matches

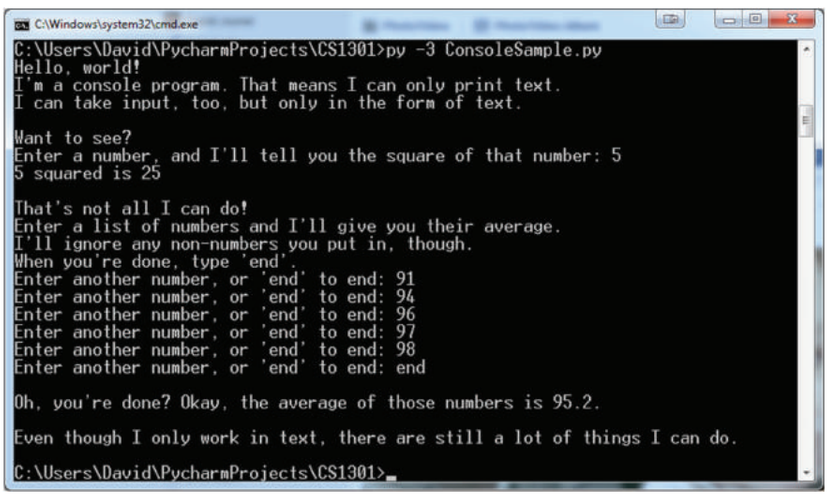


Figure 1.1.3

our expectations. If it does not, we could print some text throughout the program to see where it diverges from our expectations.

In most languages, we can handle input in this realm as well. We can prescribe specific points where the user will input some text into our program. Again, this is all carried out in plain text. The programs that we create that run in this realm are very simple, and in this context, that's a good thing: they help us understand exactly what is going on in our code.

GUIs

Graphical User Interface
An output medium that uses more than just text, like forms, buttons, tabs, and more. More programs are graphical user interfaces.

Of course, very few people use console interfaces for regular tasks nowadays. They have their niche uses, and they're very efficient for experts, but for a wide variety of reasons graphical interfaces are more prevalent.

Graphical user interfaces, like the one in Figure 1.1.4, introduce a lot more complexity into programming. We have to deal with issues of screen layout, font, and color. We have to deal with changing between multiple screens or popping up different windows. We have to understand what portion of a program has been actively selected and where the input should go. There's a lot to deal with.

Despite this, graphical user interfaces are built largely the same way that console programs are built: using lines of code that are executed in some order. Instead of just printing to the screen or taking in user input, these might do things like designate where in a form a certain textbox should be shown, or where a link should lead when clicked. The behavior of these programs is still similar to the console programs we'll design, just far more complex because there are more things to deal with.

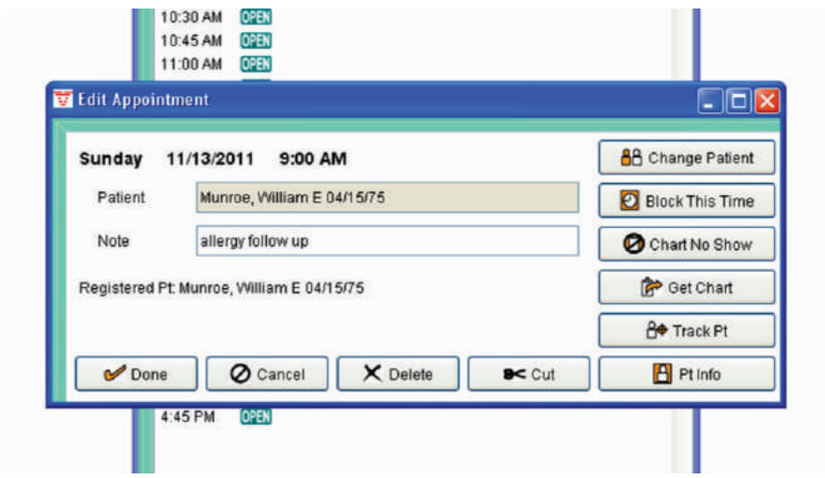


Figure 1.1.4

5. What Is This Book?

With that foundation in computing in mind, let's get started with our introduction to computing. There are lots of places online to learn the basics of computing and programming, but in this one, there are a few new and experimental approaches we're trying. In order to fully appreciate this book, it's useful to keep these unique approaches in mind.

Computing vs. Programming

Notice the title of this book is *Introduction to Computing*. You'll find lots of courses out there that are introductions to programming, and programming is indeed the

foundation of computing. Learning to program is like learning to speak the language of the computer, and so it's true, one of the learning objectives of this book is to learn to program.

However, learning to speak the language of the computer is only a small part of actual *computing*. Computing is about what you use that language to say. It's oftentimes easy to focus too strongly on the programming and miss the underlying concepts and principles of computing as a whole. At the same time, programming isn't useful just for the sake of programming: just as learning to program is learning to speak the computer's language, it's important to also understand what you're using that language to make the computer do!

To try to address this, we've separated the material for this book into three general categories: Foundations, Language, and Domain. Foundations are the core principles of computing that transcend specific programming languages. Those Foundational principles are then implemented in specific programming Languages. You then use those Foundations in a given Language to achieve something within a Domain.

That's the structure we've used to guide the construction of this book: we'll cover Foundational principles, implement them in a specific Language, and apply them to a particular Domain. The result of this will be that you'll actually find yourself going over certain ideas two or three times; this is by design! Just as study material multiple times solidifies your understanding, covering some of the same ideas multiple times in different ways solidifies it as well. We encourage you to embrace this repetition and use it to enhance your computing education.

Code Segments

To demonstrate the Foundations and the Language and to apply them to the Domain, you're going to see a lot of code segments. Within this book, you'll typically see images like the one in Figure 1.1.5, where the code is provided on the left while the output of the code is provided on the right.

On the far left are line numbers. We use line numbers to describe code because it helps us talk about where certain things are happening. In the middle is the code itself; you'll notice it's colored and highlighted. This highlighting emphasizes certain types of code, like variables, function names, comments, and reserved words; we'll cover all of these as we go forward. On the right is the output: when we run the code in the middle, we would receive the output on the right. If there is any user input, it's highlighted in a different color as well, as shown in the output in Figure 1.1.6.

However, a huge part of learning to program is tinkering with code. We encourage you to open any of the code segments on your own and play with them. Modify them and see how the results change. Try to break them, and try to fix them. Programming is a highly procedural skill, and it's only by doing it that you'll ever really know it.

#	DataTypesandVariables-1.py	Output
1	`#Gets the library we need to see the date`	5
2	`from datetime import date`	5.1
3	`#Prints the integer 5`	2016-09-19
4	`print(5)`	
5	`#Prints the number 5.1`	
6	`print(5.1)`	
7	`#Prints today's date`	
8	`print(date.today())`	
9		

Figure 1.1.5

#	ForLoopswithUnknownRanges-1.py
1	`#Creates sum with the value 0`
2	`sum = 0`
3	`#Loop 10 times`
4	`for i in range(1, 11):`
5	`    #Gets the user's number`
6	`    nextNumber = int(input("Enter number #" + str(i) + ": "))`
7	`    #Add the inputted number to the sum`
8	`    sum += nextNumber`
9	`#Print the sum over 10`
10	`print(sum / 10)`
11	
12	
13	
14	

Output
Enter number #1: 91
Enter number #2: 92
Enter number #3: 93
Enter number #4: 94
Enter number #5: 95
Enter number #6: 96
Enter number #7: 97
Enter number #8: 98
Enter number #9: 99
Enter number #10: 100
95.5

Figure 1.1.6

6. Course Outline

This Introduction to Computing is broken into five units. To get a picture of where the course as a whole is going, let's run through them right now.

Unit 1: Basics

This is where you are now. The goal of Unit 1 is to take you from no prior background on computing to the knowledge necessary to begin learning. We've already started that! You now know some basic terminology, like input, output, code, compiling, console, and GUI.

In the remainder of this chapter, we'll cover the basics of your programming language and how to get your own local programming setup ready to go. That way you'll be able to start applying the things you learn on your own. This will depend heavily on the language you're learning, but we'll get to that later. We'll also chat about the domain in which we'll apply these principles.

In the next chapter, we'll talk about the basic flow of programming: writing code, compiling it, executing it, and evaluating the results. This is when you'll get your first taste of actually writing computer code yourself. The process we'll describe here is fundamental to anything you ever do in computing.

In the third chapter of this unit, we'll cover a process called debugging. This is basically resolving errors that arise in your code, either where it won't work or where it works, but doesn't do what you want it to do. In some ways, it's hard to talk about debugging before you have lots of experience programming, but in others, you need to understand how to debug code to really make progress in learning in the first place.

At the conclusion of Unit 1, you'll be prepared to start learning to develop real computer code.

Unit 2: Procedural Programming

Once we've covered the basics, it's time to get started with programming. In the first unit, we'll cover procedural programming. Procedural programming is the basic approach to code, writing sequences of commands that are run by the computer in a specified order.

We'll start by talking about variables. Variables are how computer programs store information to be manipulated. We can use variables to store information from numbers to names to pictures to songs to pretty much anything else you can imagine. The key concept here will be variables and values. A variable is the name of some piece of information, while a value is the information itself. For example, "today's date" would be a variable: it doesn't matter what day it is, the question, "What is the value of today's date?" makes sense. A value, in turn, would be "September 12th." The value can change, but the variable name does not.

Then we'll talk about operators, starting with logical operators. Logical operators are operators that check if certain things are true or false. For example, the

statement, "today is September 12th" is either true or false, and "is" is an operator that compares equality. We have logical operators to compare equality or check if certain values are greater or less than others. We also have logical operators that combine the results of *other* logical operators. For example, "today is September 12th and it is raining" is a statement that is true or false: true if both the date and the weather are accurate, false if either is inaccurate.

Then, we'll talk about mathematical operators. A lot of programming deals with numbers, so we have operators that deal with addition, subtraction, multiplication, division, and remainders. Depending on the language, there might be others as well. If you don't care for math, though, don't worry: you don't need any math knowledge beyond arithmetic to succeed in this material. To be honest, I never personally understood a lot of mathematics until I learned computing. Computing takes a lot of the confusing things in math and makes them clearer.

Unit 3: Control Structures

With variables and operators together, we can then move on to what are called control structures. Control structures are lines of code that *control* other lines of code.

We'll start conditionals. Conditionals are how we build more complex behavior in our programs. We can tell our programs to perform certain tasks only if certain conditions are met, like rejecting a calendar invite if the user is already busy at that time or closing a file if everything in it has been read. Notice the word "if" in both those examples: conditionals are also called "if statements." A conditional runs certain lines of code *if* some condition is true (which is why our logical operators were so important!).

Conditionals then give way to even more complex control structures, called loops. Loops are how we tell our programs to repeat a certain set of commands a certain number of times or until a certain condition is true. We might use loops to change the names of every file in a folder, or to keep waiting for user input until they put something in, or to play a sound a certain number of times.

With loops, we're echoing the idea that if you want to perform certain commands multiple times, it's better to just have the lines of code for those commands in one place and refer to them when you need them rather than writing them multiple times. That gets us to the idea of functions. Functions are like little programs with their own input and output that let us organize our code better.

Finally, we'll revisit the idea of debugging with more sophistication by talking about exceptions. As we develop more complex code, we'll encounter instances where we might not want to fix every error—we might instead want to anticipate and account for them. We want to run certain lines of code *if* an error is encountered; error handling is using conditionals that monitor for errors.

This level of knowledge covers a huge portion of the foundation of computing. In fact, some of the most complex applications you've heard of, from space shuttles to early video games, are written without much more knowledge than we describe here.

Unit 4: Data Structures

Once we know how to create basic procedural programs, it's time to learn about data structures. Data structures are different ways of organizing data for our programs to use. Procedural programming covers coding around basic things like letters and numbers, but with data structures we can start to code more complex behaviors.

We'll start with something called strings. "String" is short for strings of characters, where characters are things like letters, numbers, and punctuation marks. Strings are basically the programmatic term for text. Text is one of the main ways people communicate with programs, so a lot of what we do will be text-based.

Text is also the foundation for files. Files are how we store information between runs of our programs. Imagine you implement a word processor: the user creates some document, then saves it to a file. Then, later, they open it. File input and output radically improves the usefulness of the programs we can create, and it builds on our new understanding of manipulating text.

Just as strings are lists of characters in order, so also we can make lists of any other kind of data in our programs. We might store a list of files to modify, or a list of students for a gradebook, or a list of bookmarks for a web browser. Lists are so useful that there are several ways of implementing and using lists and list-like structures that we'll cover.

When we deal with lists, we're usually dealing with information in some kind of ordered format. There's a first item, a second item, and so on. We can look up those items by searching by their number. But sometimes, we don't care about order so much as we care about easily being able to look up data. For this, we'll talk about data structures called hash tables or dictionaries. Using dictionaries, you can do things like look up a person's profile just by using their name.

These are some of the data structures most languages will give us to use automatically. However, the real power of data structures really arises when we start to create our own.

Unit 5: Objects and Algorithms

The material covered in the first four units of this material form the foundation of computing. Many classes would stop here. However, before we close, we want to preview the next two general concepts in computing: object-oriented programming and algorithms. If you go into areas like designing websites or creating mobile apps, you'll see a lot of object-oriented programming. If you go into areas like computer graphics or computing theory, you'll see a lot of algorithms. If you go into places like virtual reality or video game design, you'll likely see a lot of both!

Object-oriented programming means the ability to create our own data structures. This approach allows us to create our own ways of organizing data, more closely matching both our understanding of the problem and the demands of the program we're writing. So we'll discuss the basics of object-oriented programming and how it can be used to organize together natural ways of thinking about problems. For example, you and I have a very clear idea of the general concept of a chair that includes details like a chair will have some number of legs and some color. We can also imagine individual instances of that general concept of a chair, like the blue one with three legs at the counter or the brown one with four in the living room. That's what object-oriented programming lets us do: create concepts, and then create instances of those concepts.

Then, we'll briefly discuss algorithms. We'll start by discussing the basic vocabulary of describing algorithms, especially their efficiency. When designing algorithms that will run on millions or billions of values, small differences in efficiency can lead to major effects. We'll then discuss a common approach for designing algorithms, called recursion. Recursion is the name for functions that call themselves. Finally, we'll conclude with two of the most valuable types of algorithms, searching and sorting algorithms. Searching algorithms let us find a single item from a long list with greatest efficiency. Sorting algorithms put long lists of items in order according to a certain requirement, like alphabetizing a dictionary.

7. Introduction to Python

This version of this material is provided in terms of the Python programming language. Python is a high-level, dynamic programming language. The language was first created in the early 1990s, and reached a strong degree of popularity in the 2000s. Today, it's one of the more popular languages, especially among beginners.

Python: A High-Level Language

First, we describe Python as a high-level language. High-level here doesn't mean it's more powerful or more advanced; instead, it means it abstracts pretty far away from the core processor and memory of the computer. We don't have to worry about a lot of things like managing memory that we might need to think about in a lower-level language. That also means that the language is more portable: Python can run on PC, Mac, or Linux because there is a separate software to install to provide access to it. Lower-level languages are more likely to be tied only to certain operating systems.

The fact that Python is a high-level language means that we don't have to spend time thinking about several things we don't know about yet anyway, so while it's an important detail to keep in mind, it doesn't make much of a practical difference to us.

Python: An Interpreted Language

The fact that Python is dynamic or interpreted, however, is more significant. This means that Python will run our code line-by-line when we ask it to, without trying to compile it first. That opens up the possibility of using Python in a command-line interface, where we write and execute lines of code one at a time, more like a traditional calculator. The alternative to this is a scripting mode, where we write a bunch of code then run it all at once.

The main takeaway of Python being an interpreted language is that we might not be aware of errors until we try to actually execute those lines. Compiled languages will do some error checking before we try to execute them, but interpreted languages generally don't. However, we can use some additional tools to duplicate some of those functions.

8. Setting Up

As you go through this material, you're going to want to try out the concepts yourself. That means that you're going to need to set up an environment in which to program in Python on your own. If you're using this book as part of a course, chances are that your course has its own preferred development environment, we recommend following that. If you're reading this book independently, though—or if you want something beyond what your course recommends—here are four general options.

Files and the Command Line

The most "pure" way to do Python development is to simply write your code in text files and run it using the command line. This isn't the recommended way for beginners, but it also involves the least overhead, so we'll cover it first.

To do this, the first thing we need to do is install Python. For a brief bit of history, Python was originally created in the early 1990s. The second version, Python 2, came out in 2000, and became extremely popular. The third version, Python 3, came out in 2008. Interestingly, Python 3 *isn't* backwards compatible with Python 2. Code that worked in Python 2 won't work with Python 3, and vice versa. So, it's important to make sure we're using the same version.

This book will use Python 3. Much of the content will work for Python 2 as well, but some won't. To install Python 3, go to https://www.python.org/downloads/ and follow the directions for the latest version. If two versions are offered, make sure to choose the one that stars with the number 3 (e.g., Python 3.5.2), not 2 (e.g., Python 2.7.12).

After following those directions, you should be ready to get started. If you're going to program using raw files and the command line, you can create your files with any text editor. Notepad on PC, TextEdit on Mac, and Emacs or Vim on Linux

are popular native options. However, other tools exist that provide more features, like Notepad++. I'd personally recommend using Notepad++ if you're going to go this route.

Using the text editor, you can create code files the same way you'd create documents or pictures: write the code, and save it with the extension .py, like MyCode.py. Then, open the command line (on PC) or the terminal (on Mac or Linux). Navigate to the folder in which you saved the code, and execute the command `python MyCode.py`. This will open and run MyCode.py, showing the output on the screen (or in a separate window if need be).

Using an IDE

IDE stands for Integrated Development Environment. It's a custom piece of software intended to make coding easier. Depending on the IDE, it might provide lots of features, but at the least it usually provides dedicated windows for writing code, managing files, and viewing output. It also usually allows you to run code just by pressing a single button.

There are lots of IDEs out there for Python: NetBeans, Spider, IDLE, IdleX, Komodo, LiClipse, and PyScripter to name a few. Personally, I prefer one called PyCharm, so we'll provide brief instructions for PyCharm, but you're welcome to use whichever you want.

To obtain PyCharm, visit https://www.jetbrains.com/pycharm-edu/. This actually provides the educational edition, which should be good for any novice programmers. It even has an integrated tutorial on Python, which should be a great complement to this material!

Within PyCharm, even the Edu version, there are a lot of features. Here's the extent of what you need to get started, though, based on the initial configuration of PyCharm Edu. If you open the tool, you'll see:

- On the left, you can manage your files. Each file contains some code to run, just like a single document would contain a single essay. It's possible for code in some files to refer to code in other files, but we won't really need that for the concepts covered in this material.
- On the right, you write your code, one line at a time. The majority of what we do in the class will be here.
- Next to line 1, you'll see a green triangle. Click this to run your code. This tells the computer to actually execute your code line by line.
- On the bottom, you'll see the output of the last run of your code. For us, pretty much everything down here will come from text that we print out. This is the PyCharm equivalent of the console: it's all text. If your code takes input from the user, you'll enter it here as well.

PyCharm has a lot of other features, and we encourage you to explore them! However, the details above are all you *need* to know.

Web-Based IDEs

If you don't want to bother setting up software, though, you'll actually find there are websites that let you run code right in the browser! You wouldn't want to develop a big program that way, but it's totally fine to test out little segments of code, and likely covers the complexity of what we'll cover in our material.

Some popular examples of this include:

- repl.it, a popular and full-featured browser-based environment. Note that its output syntax differs a little from what you'll see in this book, but it shouldn't be too hard to follow. Find it at www.repl.it.
- Holy Cross's Online Python Interpreter, a simple and elegant pairing of a code window and an output window. Find it at mathcs.holycross.edu/~kwalsh/python/.

- Skulpt, an embeddable Python widget for websites that also has a version on its website. Find it at www.skulpt.org.
- Ideone, an online tool that supports dozens of languages. Find it at http://ideone.com/.
- PythonTutor's Visualize tool, an online tool that shows you the line-by-line execution of a Python program. Find it at http://www.pythontutor.com/visualize.html, and make sure to select Python 3.
- CodeAcademy Labs, which shows the code and the output side-by-side. Find it at http://labs.codecademy.com/.
- CodingGround, an in-browser development environment visually similar to PyCharm and other downloadable IDEs. Find it at http://www.tutorialspoint.com/execute_python_online.php.

Interactive Mode

As we'll discuss in the next chapter, Python also has something I call "interactive mode." In interactive mode, instead of writing blocks of code and running them all at once, you can put in one line at a time and see its result. It's very much like a very powerful calculator.

Python by default installs a tool that takes care of interactive mode, called IDLE ("Integrated Development and Learning Environment"). If you run this, you'll find you're able to enter lines of code one-by-one. There are also other web-based tools with this type of interaction as well, including:

- The Python.org shell, a browser-based version of Python's immediate mode. Find it at https://www.python.org/shell/.
- The IPython in-browser instance of PythonAnywhere. Find it at https://www.pythonanywhere.com/try-ipython/.
- CodeAcademy Labs also provides an immediate mode. Find it at http://labs.codecademy.com/.

Many introductory Python classes teach nearly the entire class in terms of interactive mode. In this class, however, we'll focus on writing code and then running it. While interactive mode is a great way to explore, the vast majority of computer science is done with this iterative cycle between coding and running, so we'll focus on that workflow.

9. Introduction to Turtles

As mentioned previously, this book is structured into three interleaved areas: computing Foundations, a particular programming Language, and applications to a specific Domain. In this version of this book, the Domain is turtles.

That probably sounds silly, so let me explain a bit further. Turtles is a popular Python graphics module for learning to program. The name "turtle" comes from the Logo programming language, created to teach programming all the way back in 1966 by Wally Feurzig and Seymour Papert. The goal was to create an environment where (a) the meaning of the code was clear, and (b) learners would receive immediate feedback on exactly what happened. The result was turtles, a graphics module where learners would instruct virtual turtles to perform certain actions, like turning around and walking forward, drawing lines behind them.

Turtle Basics

Generally, to work with turtles, you'll need to be using something on your computer to do your programming, not one of the browser-based options (although there are exceptions). The reason for this is that the window in which the turtles draw is always a separate window. So, you need an environment that can create a second window. For that reason, it's also difficult to show these programs here in this book;

we need at least two large windows, and three when we start adding user input. Some of our turtles scripts are going to end up very long, too. So, for that reason, we are not going to show the code and output for the turtles lessons of this book. Instead, we'll supply you the code separately, and you can run it yourself! So, before proceeding, get your programming environment ready.

Once we have that, we can start interacting with turtles with the first line of our program: import turtle. Then, we can start to give our turtle instructions in our code, as you'll see in TurtleBasics.py. Note that you don't need to worry how this works right now; the goal here is just to show you the way it looks.

In TurtleBasics.py, we have some lines of code, similar to the small code segments we saw earlier. Instead of printing stuff to the output on the right, though, these draw things in this separate window. Here, it draws a square. How does it draw a square? The turtle moves forward a distance of 100, then turns 90 degrees to the right. Then, it goes forward by 100 again, then turns 90 degrees to the right again. It repeats those two things one more time, then moves forward 100 one more time to complete the square.

Turtles and User Interface

Turtles are a great way to learn about a lot of programming concepts. Nearly everything we do can be expressed in terms of turtles. However, as a domain of application, turtles aren't very authentic. So, in this book, we're going to add an extra twist to it. While you're welcome to use turtles to explore the concepts, our running example is going to be creating a program that lets a user control the turtles just by entering keywords and values.

For example, we can tell the turtle to move forward by 100 with the line of code `turtle.forward(100)`. Our goal is to allow a user, not us programmers, to enter a simpler command to control the turtle. That's going to be the domain for this version of this book: graphics and user interfaces, building a user interface to let users control a graphics module.

CHAPTER 1.2

Programming

1. What Is Programming?

Programming is the foundation of computing. Programming is effectively being able to speak the computer's language, to give it directions in a way that it understands. Like any language, computers have vocabulary words and syntax that they understand. A lot of this material will cover exactly that: how to speak the computer's language.

The Programming Flow

Programming is more than just knowing the computer's vocabulary, though. Programming is also the process by which we create computer programs, just like writing is the process by which we create essays. It's not a good idea to write an essay once and submit it without ever revising it, and similarly, we don't write code and have it work perfectly the first time. Programming is an iterative process of writing code, attempting to run it, and evaluating the results.

This three-stage process might seem quite familiar to you. It's how you write an essay, it's how you paint a painting, it's how you solve a math problem; it's effectively how you do anything. You write the essay, you read over it or show it to the teacher, and you evaluate it. You paint a painting, you show it to an audience or a mentor, and you plan how to improve your next one. You try to solve a problem, you look up the answer in the book, and you evaluate whether your method matches. Coding is not that different from writing, painting, or solving math problems.

Chapter Preview

In this lesson, you'll get your first experience writing actual code. Our hope is that you'll see how easy it can be, but if it presents challenges, don't be discouraged.

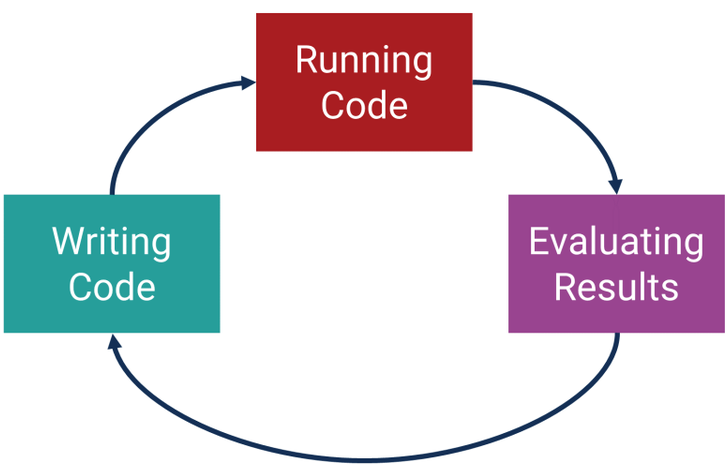


Figure 1.2.1

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Participate in the basic cycle of programming: the three-stage process;
- Differentiate between compiling and debugging and will gain a basic understanding of errors;
- Write basic lines of code in Python, and print statements, variables, and some basic methods;
- Write basic codes, run, and evaluate them within the turtle's library.

Programming
writing code through an iterative process of writing lines of code, attempting to execute them, and evaluating the results.

While computing is similar to writing essays, painting pictures, or solving problems in some ways, it also represents a very different way of thinking in others. It can take some time to get used to. Don't fret! You'll get there.

After your first experience writing code, we'll move on to talking about running code and evaluating the results. We'll also touch a little bit on how to fix things when your code generates errors or when it doesn't perform as expected. It's important to note that we don't expect you to fully understand everything in this chapter before moving on—you'll only truly understand this material once you get some practice writing, running, and evaluating code as you go through this book. However, to really be able to get into that process, it's important to first have some exposure to what the process looks like. This chapter is meant to give you that foundation.

2. Writing Code: Lines

In every language I've ever encountered, the most basic atom of development is the line of code. Lines of code are individual commands to give to the computer. Chains of these lines form complex behaviors or instructions for the computer to carry out.

Chaining Together Instructions

Imagine we are developing a program to print out the roster of students in this class. One command would instruct the program to grab a student's profile from some file or database. Another would instruct the program to grab the student's name from that profile. Another would instruct the program to print that name. Another would instruct the program to repeat those three commands for every student in the class. By chaining these instructions together, the computer can print out the entire class roster.

The Print Statement

There are two important things to note in this example. The first is that third command: print. As we get started with programming, the `print()` command is the first fundamental thing to understand. This is how you print output for you to see while running. In fact, the very first program that you'll develop in the next lesson is just a print statement.

Work in Small Chunks

The second is that we want to develop programs in small chunks. You don't write an entire essay from start to end without reading over the paragraphs as you write them. You likely wouldn't paint a picture from start to finish without pausing to get feedback. So also, we want to develop our programs in small chunks, testing throughout to make sure we're on the right track. In this example, we might first write a program that can print one single student's name before adding the fourth command and printing all of them.

By writing commands in small chunks, we get constant feedback and can detect errors early on in development rather than waiting until they will be much more difficult to fix. We can print things out frequently to check on how things are working as well. You'll try that out in the next lesson.

3. Writing Code: Lines in Python

Let's get started with writing your first Python program. We've talked about how programs are made from lines of code, where each line is basically a command for the computer to do something. We've talked about how the programs we're going to design initially will print to the console. On the left of Figure 1.2.2, we have our code, and on the right, we have our console. So, let's write a program.

#	YourCodeHere.py	Output
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

Figure 1.2.2

Your First Program: Hello, World

In my code, shown in Figure 1.2.3, I'm going to write a simple command: `print("Hello, world")`. This is a simple command that instructs the computer to **print** the text "Hello, world." When I run this little one-line program, the console on the right prints the string of text I asked it to print. Voila! We have our first Python program. The program has one command, to print "Hello, world." So, in our output, we see one action: the computer has printed "Hello, world."

`print(message)`
takes as input a message as a string of characters and prints it to the console.

Print
output some text to the console.

Note a couple things here in Figure 1.2.3. First, note that this line needs to be typed just as you see it here to print this text. The term "print" must be in all lower case letters, followed by an open parenthesis and ended with a close parenthesis: Python looks for that word specifically as a command it understands, and capitalizing it prevents Python from recognizing it. Secondly, note that the text you want to print must generally be surrounded by quotation marks; we'll discuss exceptions to that later. It can be single or double quotation marks, but it has to be surrounded by them: that's how Python recognizes that this is some text to print instead of the name of a variable, which we'll talk about next time.

Third, note that when we list function names like "print," we usually follow them with parentheses, e.g. `print()`. This is to identify that the function is a function as opposed to a variable. This will make more sense later; for now, know that you aren't going crazy by finding the open and close parentheses odd. You'll get used to seeing that, and you'll soon understand what they mean.

It's also worth noting here that Python might be the simplest language to use to get to this point. Other languages require some additional lines of code or some additional setup to get to the point of just printing one line. Python makes this considerably easier. If in the future you switch to Java, C, or some other language, you'll find that even simple print statements like this have to be contained within some broader code structure. That's one of the things that makes Python a great language to learn first.

#	YourFirstProgramHelloWorld.py	Output
1	`print("Hello, world")`	Hello, world
2		

Figure 1.2.3

#	PrintingOtherValues.py	Output
1	`print("Hello, world")`	Hello, world
2	`print(5)`	5
3	`print(5.1)`	5.1
4	`print(True)`	True
5	`print(False)`	False
6		

Figure 1.2.4

Printing Other Values

boolean
a simple True or False value.

In addition to printing strings of characters like "Hello, world," Python also lets us directly print a couple other things. First, we can print numbers like 5 or 5.1 directly using the print statement without quotation marks. Second, we can also print what are called "boolean" values. We'll talk more about **boolean** values later, but for now, just know these are simply either `True` or `False` values.

So, in the code shown in Figure 1.2.4, we're printing all these types of values. We tell the computer to print our string of characters that reads "Hello, world," then we tell it to print the numbers 5 and 5.1, then we tell it to print the values `True` and `False`. Then, when we look at our output, we see the computer executing these commands in this order: it prints "Hello, world," then it prints 5, then it prints 5.1, then it prints True, then it prints False.

The Programming Flow

This lesson is about the overall programming flow from Figure 1.2.1, between writing code, running code, and evaluating the results; here we've focused on that initial process of writing code. Note, though, that we really can't talk about writing code alone. In this lesson, we've covered all three phases. We've written lines of code, we've run the code, and we've evaluated the results. This tiny cycle in which we've engaged is the entire programming flow. Notice also that our prior instructions on coding in small chunks apply here, too: we initially just printed "Hello, world" to make sure we were printing things correctly. Then, we moved on to printing other things as well. That way, if we were making mistakes in how we wrote our print statements, we would detect those mistakes early.

4. Running Code: Compiling vs. Executing

Figure 1.2.5

Now that we've written some code, it's time to move on to trying to run it. Depending on the language and environment in which you're working, running may involve multiple steps: compiling and executing. Earlier, we discussed the definitions of these terms; now, let's go into a little more depth on what they mean in practice.

To explore this, let's use an analogy: imagine you're trying to build a table. You have the parts and you have the instructions. In this analogy, you're like the computer, the instructions are the code, and the parts are like the files or data the program would act on.

Compiling

What do you do first in this case? The first thing you might do is read over the instructions in their entirety. You might check to make sure you understand each individual instruction. You might make sure that all the parts the instructions reference are present. This is analogous to compiling a computer program: reading over the code and making sure everything makes sense. After all, if there are parts of the instructions or the code that don't make sense, there's no reason to proceed: we have

to fix those problems first. If there is a problem in the fifth step or the fifth line of code, **compiling** prevents us from executing the first line until the fifth line is fixed.

Of course, when building our table, there's no requirement that we read over the instructions first. We could just get started, and if there's a problem, we'll encounter it when it comes up. The same is true for programming, and this decision is made at the language level. Some languages, like Java or C, require compilation. These are often described as "static" or "compiled" programming languages. Other languages, like Python and JavaScript, do not require compilation; instead, they just run the lines one-by-one without checking them in advance. These are often described as "dynamic," "interpreted," or "scripting" languages. Even with these languages, some tools can simulate the "compilation" process, checking our code for errors before we actually execute it.

This description covers how compilation works in the practical sense. In the technical sense, compiling is the process of taking all the code that you've written and translating it down into the language the computer can understand. At their core, computers can only process basic commands, and so our high-level coding must be translated down into basic commands before the computer can actually run our code. Understanding that idea is outside the scope of this Introduction; you'll learn about that more if you decide to go into computing more deeply, especially if you decide to focus on developing operating systems like Windows, Mac OS, or Linux.

Compile
to translate human-readable computer code into instructions the computer can execute. In the programming flow, this functions as a check on the code the user has written to make sure it makes sense to the computer.

Executing

Whether you walked through this compilation step or not, we then move on to **execution**. If you're building a table, this means actually starting to follow the instructions and build the table. For code, this means actually running the code and let it do whatever it was designed to do.

When we reach this step, a number of things can happen. First, even if we compiled first, we could still run into errors. In building the table, you could find that the screws won't fit in the holes, or the legs can't support the weight of the top. You couldn't have discovered that during the compilation step. If you didn't compile, this might also be where you discover issues like missing screws. These are errors: fundamental problems that prevent the code from running to completion.

Even if there are no errors, though, that does not guarantee we'll get the results we want. Imagine, for example, that the instructions were incorrectly written for building a chair instead of a table. Checking the presence of all the parts and the logic of each instruction wouldn't catch that. We don't hit any problems while building it. However, at the end, we end up with something different than what we want. The code could run just fine, and still not do what we want it to do.

Third, and ideally, the code could also run just fine, do exactly what we want, and generate the correct results. That's the goal of the programming flow: to ultimately create programs that do what we want them to do.

Execution
running some code and having it actually perform its operations.

5. Executing Code in Python

The Python programming language is a dynamic, interpreted, scripting language. That means that the compilation step isn't required. When I click to run some code, it just starts executing the lines one-by-one. If there is an error on the fifth line, it will execute the first four before telling me about the error.

Encountering Errors

Let's try an example of this. In Figure 1.2.6 are five lines of code. You might notice that the fifth line has an **error**: I've misspelled the word "print." What happens when I run this code? The first four lines run just fine and print their output, but when the computer reaches line 5, it prints the error message shown on the right, under

Error
a problem that prevents code from continuing to run if not handled.

#	ExecutingCodeinPython.py	Output
1	`print("This is Line 1")`	`This is Line 1`
2	`print("This is Line 2")`	`This is Line 2`
3	`print("This is Line 3")`	`This is Line 3`
4	`print("This is Line 4")`	`This is Line 4`
5	`pritn("This is Line 5")`	`Traceback (most recent call last):`
6		`  File "ExecutingCodeinPython.py",`
7		`  line 5, in <module>`
8		`  pritn("This is Line 5")`
9		`NameError: name 'pritn' is not defined`
10		
11		

Figure 1.2.6

'Traceback'. This error message then gives us the information we need to track down and repair the problem we encountered.

"Compiling" Python

Although Python is a dynamic language that does not require a compilation step, some Python development environments supply that anyway. For example, Figure 1.2.7 shows PyCharm, a popular Python Integrated Development Environment, or IDE. PyCharm simulates that compilation step before executing your code. It might do that all at once before you attempt execution, or it can even do it in-line as you type. This is almost like spell-check on a word processor: it can detect possible problems as you type. Note that this is different than the technical definition of compilation, but in terms of our programming workflow, it plays the same type of role, detecting errors before execution.

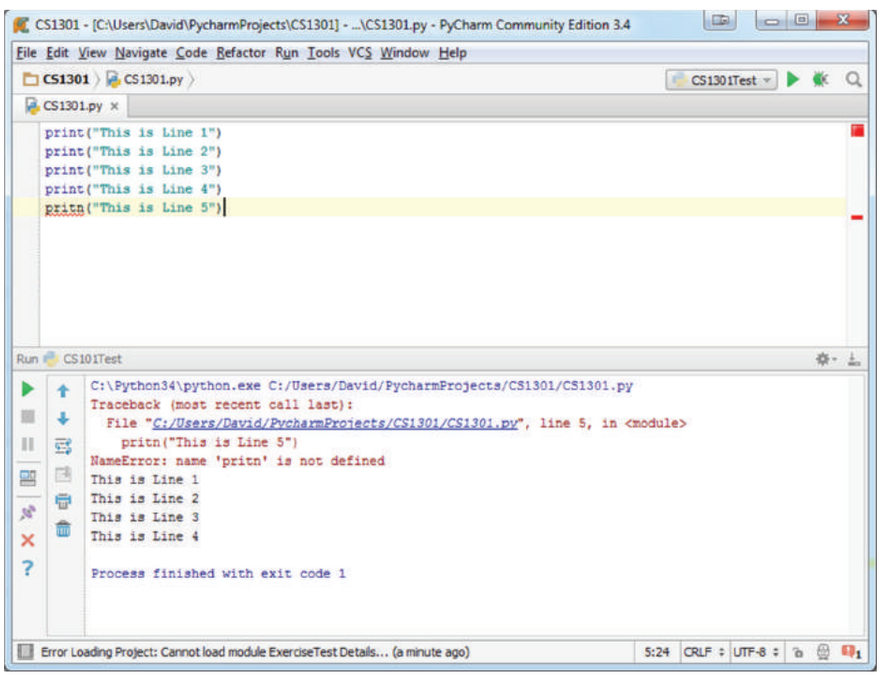


Figure 1.2.7

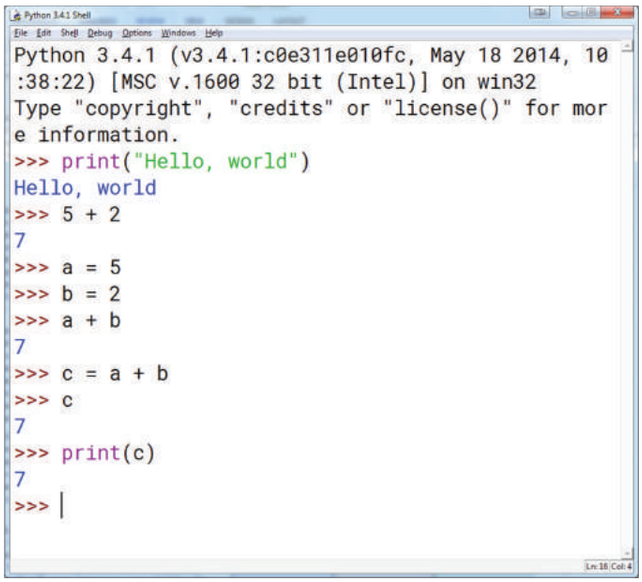


Figure 1.2.8

The Python Interactive Mode

So far, and in the majority of the course, we discuss programming flow in terms of this cycle between writing, running, and evaluating code. In Python, this is called Scripting Mode. However, Python does have a mode that differs from this model, called Interactive Mode. In Interactive Mode, Python works a lot like a really sophisticated calculator: you type commands directly in and get results directly back out. If compiling was like reading over instructions in advance and executing was like running the instructions themselves, then Interactive Mode is like having your friend shout the instructions to you without ever showing them to you all together.

In Figure 1.2.8, we can see Interactive Mode in action. When the Interactive Mode window shows three arrows, it is waiting for us to enter a line of code. When we enter one and press enter, it immediately runs the line and shows us the results. This mode can be quite useful for getting quick feedback or quickly exploring whether certain methods will work. Any Python code can be entered line-by-line into Interactive Mode, and it will generate the same results that the code would have generated had it been run using Scripting Mode.

Some courses teach Python entirely using Interactive Mode. However, we will generally always use Scripting Mode instead. The immediate feedback of interactive mode is very useful in learning the basics of Python and coding, but the goal of this course is to give you an Introduction to Computing. The programming flow between writing code, running code, and evaluating the results is far more common in computing than the more immediate type of interaction provided by Interactive Mode.

6. Evaluating Results

At the evaluation stage, we check the output of our execution and see if it matches our goals. We've discussed three possible outcomes when you reach that evaluation stage: errors, incorrect results, or correct results. If the results are correct, then we're done and we can move forward to the next small chunk of development. So, let's talk about the other two possibilities: errors and incorrect results.

Errors

Errors occur when our code attempts to do something it isn't allowed to do. The program is forced to just stop trying because it can't move forward from that error. In our example of assembling furniture, this is like discovering that the screws are the wrong size or that you don't have a hammer. Until the problem is solved, we can't proceed.

But what do those errors look like in programming? There are lots of possible errors we could encounter, and we'll cover what they might be as they come up in our future chapters. We might, for example, try to open a file that doesn't exist. We might try to add things that can't be added together, like blue plus cabbage. We might try to divide by zero. There are lots of things that can go wrong.

When an error occurs whether during compiling or running, we're generally informed where in our code the error happened; for example, in Figure 1.2.6, the error message said "line 5." So, we can go look at the code and try to figure out what's going on. Oftentimes, that line of code itself doesn't have the problem: instead, the problem occurred because something incorrect happened earlier. For example, if we tried to open a file that doesn't exist, the line of code that tried to open the file might not be the problem; instead, the line of code with the problem might be the one that created the incorrect filename. The computer doesn't know that's a problem, though, until it tries to open that file. We'll cover more about how to resolve this when we talk about debugging.

Incorrect Results

The other likely outcome of our code is for it to run successfully, but not do what we want it to do. In our furniture analogy, this is like following the steps only to discover that they built a chair instead of a table. The steps were fine, but they generated the wrong results.

In programming, this can take on lots of forms as well. For simple examples, maybe we add when we want to subtract. Maybe we sort files alphabetically when we want to sort them by creation date. There are lots of things we could do that would work perfectly fine, but wouldn't generate the results we want. When that occurs, we have to try to trace through our program and discover where the incorrect steps are being taken. That introduces an interesting twist to the programming flow: we might write some new code whose goal is to help us understand the incorrect results, rather than just writing new code that tries to accomplish our original goal. We'll talk more about that, too, when we discuss debugging.

7. Evaluating Results in Python

We've covered the programming flow between writing code, executing code, and evaluating code. We've talked about how we use this cycle to check for problems with our code; it might generate errors, or it might simply not perform how we want it to. We've discussed how the results of an evaluation feed into the next iteration. Now that we know all the principles, let's go through an example of the entire cycle.

In this example, we're going to see some code that is going to look unfamiliar. By the end of the book, you'll understand what everything you see here means. For now, though, don't worry too much about it; focus instead just on the nature of the output and revisions. The goal of this code will be to print the numbers 1 through 9 in order.

Errors in Python

The code shown in Figure 1.2.9 is intended to print the numbers 1 through 9 in order. Right now, you don't know what this code means, but don't worry. All you need to know right now is that the goal is to print these numbers. So, we write the code on

#	ErrorsinPython.py	Output
1	`for i within range(1, 9):`	`File "ErrorsinPython.py",`
2	`    print(i)`	`line 1`
3		`    for i within range(1,9):`
4		`                ^`
5		`SyntaxError: invalid syntax`
6		

Figure 1.2.9

#	IncorrectOutputinPython-1.py	Output
1	`for i in range(1, 9):`	1
2	`    print(i)`	2
3		3
4		4
5		5
6		6
7		7
8		8
9		

Figure 1.2.10

the left in Figure 1.2.9, run it, and get the results on the right. What do we see? The computer spits out a syntax error. Specifically, it tells us that there is an error on line 1, and the caret(^) notes that the problem is with the word "within." The computer is telling us that there's a problem on line 1 at the word "within." This gives us the information we need to start resolving the problem.

So, in this case, what is the problem? The problem is that "within" isn't a word that Python recognizes. Instead, it recognizes the word "in." So, we replace the word "within" with "in" and try again, as shown in Figure 1.2.10. Note how this brings to a close one cycle through the programming flow: we wrote code, we ran it and got some output, we evaluated that output, and we used that evaluation to inform more code revisions. Specifically, it was the output of one run that let us know to replace "within" with "in."

Incorrect Output in Python

Based on that revision, we now run the code again, as shown in Figure 1.2.10. What do we find? Good news! This time, the code runs without generating an error. Bad news! Although it did not generate an error, it didn't do what we wanted it to do. We wanted to print the numbers 1 through 9, but instead, it only printed the numbers 1 through 8.

Correcting this kind of problem can be a bit tougher. When we hit an error, the computer told us specifically where the error occurred. It won't always be as straightforward as the example in Figure 1.2.9, but it will generally at least give us a starting point. With incorrect output, though, we don't have any such feedback because the computer doesn't understand that the result is wrong. The computer doesn't know our intentions, only the commands that we enter.

Fortunately in this case, the resolution is not difficult. When we look at the output of the code in Figure 1.2.10, we see that it is printing one fewer number than we want. We might not fully understand the first line of code, but we might be able to infer that `range(1,9)` in some way specifies the numbers 1 through 9. If the code isn't printing enough numbers, maybe we need to increase the second number to 10.

#	IncorrectOutputinPython-2.py	Output
1	`for i in range(1, 10):`	1
2	`    print(i)`	2
3		3
4		4
5		5
6		6
7		7
8		8
9		9
10		

Figure 1.2.11

We've evaluated the output and come to a conclusion on what revision is necessary, so we return to the first phase of the programming flow and modify our code.

When we make that revision and run that code, we see the correct results now show up in Figure 1.2.11. Fortunately, this was an easy example, but as our code gets more complex, tracking down the cause of the problems can get more difficult. So, in the next chapter, we'll cover more advanced ways of uncovering these kinds of problems, called debugging.

8. Programming with Turtles

So, we've seen a couple of really simple examples of Python programs. In these programs, a couple lines gave us a couple pieces of text output. With turtles, though, those exact same lines can give us more complex graphical output!

Drawing a Square

In TurtleBasics.py, you'll find the code that we saw in our last chapter. At the time, we showed it just enough for you to see the kind of output to expect. Now, let's look at it in more detail.

From the rest of this chapter, we know that the computer is going to run these lines one by one. What does each line do? The first line just tells the computer to go out and get some information it doesn't have automatically. "turtle" is a module that it doesn't see by default, so `import turtle` just tells it, "Hey, go grab the information on turtle."

From there, the commands one by one tell the turtle to do different things. `turtle.forward(100)` tells it to move forward by 100. `turtle.right(90)` tells it to turn right by 90 degrees. By repeating these things four times, it draws a square.

Other Turtle Commands

There are a lot of things we can put here. The turtle module gives us lots of commands to use. We'll mostly use `turtle.forward()` and `turtle.right()` in our work, but you're encouraged to play around with it! You can't break anything; the worst that can happen is your code won't run. So, play around with some of these commands, both now and in the future. In each case, replace `x` with the number you'd like.

- `turtle.forward(x)`: Moves the turtle forward by `x`.
- `turtle.backward(x)`: Moves the turtle backward by `x`.
- `turtle.right(x)`: Turns the turtle to the right by `x` degrees.

- `turtle.left(x)`: Turns the turtle to the left by `x` degrees.
- `turtle.setx(x)`: Sets the turtle's horizontal coordinate.
- `turtle.sety(x)`: Sets the turtle's vertical coordinate.
- `turtle.circle(x)`: Draws a circle with radius `x`.
- `turtle.penup()`: Lifts the pen so the turtle stops drawing whenever it moves.
- `turtle.pendown()`: Puts the pen back down so the turtle resumes drawing when it moves.
- `turtle.pencolor(x)`: Changes the pen color to the color given by `x` (put it in quotes, like `turtle.pencolor("red")` and `turtle.pencolor("brown")`).

Try just entering different combinations of these commands to see how the turtle reacts and what is drawn as a result!

CHAPTER 1.3

Debugging

1. What Is Debugging?

Last chapter, we discussed a little bit about debugging. **Debugging** is trying to find why your code doesn't behave the way you want it to. Maybe it's generating some errors. Maybe it's not generating errors, but the output isn't what you want it to be. Either way, there's a bug in the code: a bug is a mistake or problem that is causing the code to fail at its goal. Debugging is the process of finding and removing those bugs.

Debugging is one of the most fundamental parts of programming. In fact, the programming flow we've described could just as easily be called the debugging flow: it's the flow you go through when you've written some code to check if it behaves as desired—or fix it if it doesn't.

Just as we noted in the previous chapter, this content might seem a bit confusing right now. Until you actually get really into writing code and debugging it, the debugging process might seem very abstract. At the same time, though, as soon as you start programming, you'll likely start to encounter these issues. So, the goal of this chapter is to give you some initial background in debugging so that when you start to encounter bugs in your code, you know how to get started resolving them. Don't feel like you need to pass a test on this chapter alone before moving on.

Debugging in the Programming Flow

In our short example in the previous chapter, we showed how we might go through a tiny debugging loop. We wrote some code, we ran that code, we saw a problem, and we fixed it. Sometimes, that's exactly how this process will go: the output we get when we try to run our code will be all we need to find and fix the bug.

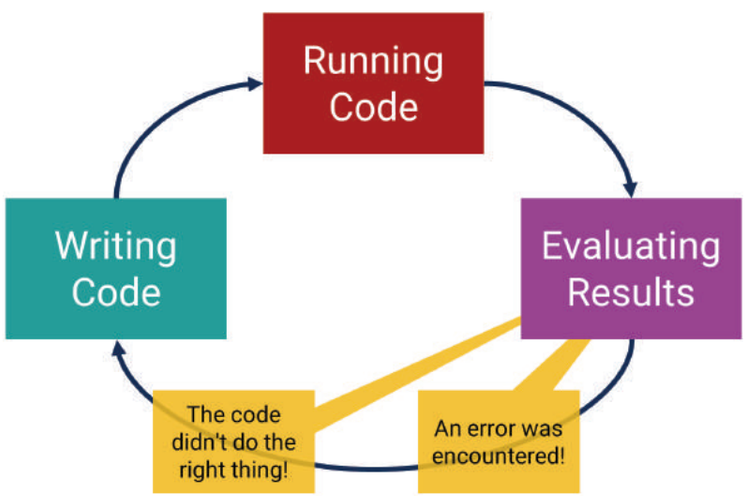


Figure 1.3.1

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Understand the role and workflow of debugging and debug broken code;
- Understand the output that comes from the Python interpreter and how it can be used to aid debugging;

Debugging
Resolving problems in code, whether it be errors thrown in compilation or running or mismatches between the desired and observed output.

However, that won't always be the case. Many times the output will only be one piece of the puzzle in resolving the problems in our code. In this programming flow, don't make the mistake of thinking that the code we write in that first phase will always be code that is important for our program to run. Oftentimes, we might write code in that stage whose only purpose is to help us debug a problem.

Debugging is kind of like doing research on our code. We need to find the information necessary to build our understanding of what's going wrong, and one way to do that is to get more information out of our code while running it—even if that information isn't important to the code's ultimate goal.

2. Types of Errors

The first thing to know about debugging is the kind of errors you'll encounter. We've already talked about the three high-level things that can occur when running some code: it can perform correctly, it can perform incorrectly, or it can fail to perform at all because it generates an error. Let's zoom in on that last one a bit and talk about the two types of errors: compilation errors and runtime errors.

Compilation Errors
Errors that occur during the computer's read through of the code.

Compilation Errors

As you might have guessed from our previous discussion of compiling vs. executing, compilation errors are those that occur when compiling our code in the first place. Not every language has compilation, and so, not every language has compilation errors. Those languages that don't require compilation, like Python and JavaScript, can sometimes have tools that simulate this process, letting us know before we execute our code if there are errors present within it.

For an error to be a compilation error, it has to be a problem inherent within the code. In other words, it has to be an error we can identify just by looking at the code, not an error that only exists in the context of how the code is run. For example, imagine some code whose job is to delete all the files in a folder. If you tried to run it on an empty folder, it might give an error, but that error wasn't a problem with the code itself. It only arose when the code was executed. That would *not* be a compilation error because there's no error inherent in the code.

Errors can differ significantly from language to language, but there are some common ones. For compilation errors, some things you might often encounter are:

- **Syntax errors.** You've written code that just doesn't make sense in the current programming language. This is akin to a grammatical error in an essay.
- **Name errors.** You've written code that tries to use something that doesn't exist. Imagine, for example, I asked you, "Give her the book," but never specified who "her" is. The command doesn't make sense because I'm including a person that doesn't exist.
- **Type errors.** You've written code that tries to do something that doesn't make sense, like requesting the smell of True or the color of the number 5.

Note that depending on the language, some of these might turn up as runtime errors instead. For example, some languages don't specify types within the code; for them, it isn't until runtime that we realize we're requesting the color of the number 5, not the color of a car. Generally, when the error occurs is a lesser concern than why it occurs and how to solve it, so don't worry too much about the difference between compile-time and runtime errors for now.

Runtime Errors
Errors that arise when trying to actually execute the code.

Runtime Errors

Runtime errors are errors that we encounter only when actually running the code. Languages that don't have compilation will only have runtime errors, and even languages that do require compilation can have runtime errors because we can't anticipate every error just by looking at the code.

Runtime errors most often occur because of something specific to the results that code generates when it runs. Some of the common runtime errors you will encounter are:

- **Divide by zero errors.** Your code contains a number being divided by another, but when those numbers actually have values, it turns out you're trying to divide by zero!
- **Null errors.** Null errors are like name errors: you're referring to something that doesn't exist. Here, though, the variable would exist, but it wouldn't have any value. Imagine your code said, "Grab the twelfth book on the shelf," but the shelf only has six books. The request makes sense in the directions until you see the shelf: then you realize you're trying to use something that doesn't exist, but you only "see" that at runtime.
- **Memory errors.** Your computer can only remember a certain amount of stuff at a time. If you try to require it to remember more than that, you'll hit a memory error.

For each of these, and for any other runtime error, it isn't until the computer tries to actually execute these steps that it realizes there's a problem. Runtime errors can be a much more difficult to resolve because instead of just finding the error in the code, you might also have to find the error in the data that the code is acting upon.

3. Types of Errors in Python

Now that you know the types of errors you might encounter, let's show what they look like in Python. The main goal here is just to expose you to these errors so that when you encounter them later, you aren't surprised. Every programmer encounters errors while programming, so learning to deal with them is a key part of learning to code.

NameError

A `NameError` occurs when you use a variable name that doesn't exist. Don't worry too much about what variables are right now, we'll cover those very soon. For now, observe in Figure 1.3.2 that the letters `a`, `b`, `c`, and `d` were all used prior to the print statements on lines 5 and 6. The print statement on line 5, which only requires `a`, `b`, `c`, and `d`, runs just fine and outputs "10," which is the sum of 1, 2, 3, and 4. The print statement on line 6 requires `e`, but `e` was not used previously. So, `e` is a name that Python does not recognize right now, and so it returns a `NameError`.

In the error message, note the information we're given. We're told what line to look at where the message says "line 6," so we know to jump to line 6. The error message specifies "name 'e' is not defined," which tells us exactly what variable it didn't know.

#	NameError.py	Output
1	`a = 1`	`10`
2	`b = 2`	`Traceback (most recent call last):`
3	`c = 3`	`  File "NameError.py", line 6,`
4	`d = 4`	`  in <module>`
5	`print(a + b + c + d)`	`    print(a+b+c+d+e)`
6	`print(a + b + c + d + e)`	`NameError: name 'e' is not defined`
7		

Figure 1.3.2

TypeError

A `TypeError` is one type of our error where we're trying to do something that doesn't make sense. In the code shown in Figure 1.3.3, `len()` will give us the length of whatever is inside the parentheses. It makes sense to ask about the length of a text

#	TypeError.py	Output
1	`print(len("Hello, world"))`	`12`
2	`print(len(5))`	`Traceback (most recent call last):`
3		`  File "TypeError.py", line 2,`
4		`  in <module>`
5		`    print(len(5))`
6		`TypeError: object of type 'int' has`
7		`no len()`
8		

Figure 1.3.3

message—it counts the number of characters, find that it's 12, and prints out 12. It doesn't make as much sense to ask about the length of a number, though.

As with the `NameError`, Python tells us where it encountered this error; here, it's on line 2. Alongside the `TypeError`, we also see an additional message: "object of type 'int' has no len()." We'll talk more about what this message means later. For now, though, if you correctly infer that "int" stands for "integer," then this sentence suggests that there is no "length" of an "integer," which gives us the information necessary to move forward a bit more.

AttributeError

An `AttributeError` is the other result when we try to do something in our code that doesn't make sense. The difference between `TypeError` and `AttributeError` can be a little technical, so don't worry about it for now—we'll cover it in unit 5. Instead, just understand that they're two errors that correspond to times when we try to do things that don't make sense.

The code in Figure 1.3.4 creates two variables, and gives one the value "Hello, world" and the other the value 5. Don't worry too much about what these lines mean; just know that lines 3 and 4 are trying to check whether those two variables end in the letter d. It makes sense to ask if a text message ends in the letter d. It doesn't make sense to ask if a number ends in the letter d. So, Python prints `True` when asked if "Hello, world" ends in d, but prints an `AttributeError` when asked if 5 ends in d. Like the `TypeError`, the message is that an integer (an "int") has no attribute "endswith," meaning that we can't use "endswith" on an integer.

#	AttributeError.py	Output
1	`a = "Hello, world"`	`True`
2	`b = 5`	`Traceback (most recent call last):`
3	`print(a.endswith("d"))`	`  File "AttributeError.py", line 4,`
4	`print(b.endswith("d"))`	`  in <module>`
5		`    print(b.endswith("d"))`
6		`AttributeError: 'int' object has no`
7		`attribute 'endswith`
8		

Figure 1.3.4

SyntaxError

The last common type of error we'll cover for now is the `SyntaxError`. A `SyntaxError` is kind of a catch-all error: it refers to lots of different things that can be done wrong, all based on violating Python's internal grammar.

We can see an easy example of this just with the print statements we've been using so far. We've talked about how Python requires parentheses to surround whatever it's supposed to print. So in Figure 1.3.5, when we write `print(5)`, it prints the number 5. When we write `print 5`, it gives us a syntax error. This is actually an interesting example because old versions of Python (Python 2,

#	SyntaxError.py	Output
1	`print(5)`	`File "MyFile", line 2`
2	`print 5`	`  print 5`
3		`        ^`
4		`SyntaxError: invalid syntax`
5		

Figure 1.3.5

in particular) used the second type of print line, but newer versions use the first. You'll still see lots of Python using the older version, so you might see lines like the second; in this material, however, we assume the newest version of Python.

`SyntaxErrors` can encompass a wide variety of different problems. They can also be somewhat hard to debug. `SyntaxError` means Python couldn't even read what was written, so it can't give the kinds of feedback that it gave on earlier errors.

4. Basic Debugging

You've run your code, but it doesn't behave as expected. Perhaps it generates an error, or perhaps it runs just fine but just doesn't do what you want it to do. In an ideal case, the error message from the computer gives you all the information you need to quickly resolve the problem. In many cases, though, debugging is going to be a longer process.

To help organize your debugging processes, remember that the goal of debugging is to get the information necessary to locate and fix the error. Simply obtaining the right information is important. A common mistake novices make is that they try to just stare at the code until the problem becomes obvious; instead, if the problem isn't immediately obvious, try instead to add code that will help make the problem clearer.

In this lesson, we're going to discuss three general kinds of debugging: print debugging, scope debugging, and rubber duck debugging.

Print Debugging

Print Debugging
A form of debugging where print statements are added throughout the code to check how the program is flowing.

The simplest and most common type of debugging you'll use early on is called print debugging. Some people also refer to this as tracing. With print debugging, you simply instruct your program to print out its status throughout the run process. Looking through these print statements, you can often easily figure out where the program is differing from your expectations.

For example, imagine you've written a short, simple 10-line program, but you're surprised that it never seems to actually finish running. It just runs and runs until you're forced to stop it manually. There would be no error message, and potentially no output. By adding a print statement after each line, you can quickly see what line is taking a long time to run, or what line is executing over and over again.

Using print statements like this let you more easily visualize your program's overall flow and identify when it differs from your expectations.

Scope Debugging

Scope Debugging
A form of debugging where print statements are added to check the status of the variables in the program at different stages to see how they are changing.

Imagine you're writing a program to calculate the average grade in a class. You do this by looping through each student in the class one-by-one, adding their grade to one counter and counting the number of students with a different counter. Then, at the end, you divide the total score by the number of students.

However, you run this code, and it always says the average score is almost 0. You know that isn't true because you know everyone in the class has a positive score.

You use print debugging to track the flow of the program, and you find it is in fact looking at every student. So what do we do now?

Scope debugging is my term for debugging small sections of a program to make sure things have run correctly so far. We still do this with print statements; instead of just printing to trace the program's flow, we print to examine whether a certain section of code behaved correctly. For example, we can check after each student whether the sum is what we expect it to be at that stage.

Rubber Duck Debugging: A form of debugging where the programmer explains the logic, goals, and operations to an inanimate listener to methodically step through the code.

Rubber Duck Debugging

There's an interesting phenomenon documented by nearly every programmer I've ever talked to. You'll often have problems that you can't solve using any of the traditional debugging methods. So, you go online and post on an Internet forum asking for help. There, you have to explain your problem in depth to get a useful answer. Very often, though, the simple act of explaining the problem in depth reveals the answer. By being forced to describe the problem from scratch to a new reader, we get an outside perspective and resolve the problem.

This is where rubber duck debugging comes in. Rubber duck debugging was introduced by the 1999 book *The Pragmatic Programmer*, and it refers to a programmer who carried around a rubber duck to which to explain problems. By explaining things to the duck, the programmer often found the solution. Now, it doesn't have to be a duck; for me, it's my cat. The point is: when faced with a hard-to-solve problem, try explaining it from scratch. You'll often find the solution.

5. Basic Debugging in Python

Let's check out these debugging methods in action in Python. We'll use the same examples shown previously, but we'll specifically talk about how they work in Python.

Print Debugging in Python

Let's start with a short code segment. The goal of the code segment in Figure 1.3.6 is to count from 1 to some number (in this case, 10), and then back from that number to 0. At the end, the code should print out the final number. Don't worry too much about how this code works, though; you don't need to understand this syntax to understand this example.

When we run this code, what happens? …nothing. Nothing happens, although we might notice our computer fan starts working overtime. In evaluating the results of running our code, we note it just runs and runs and runs, never outputting anything. There is no error message, though. Because the code ends on a print statement, we know that it's never reaching that line because it never outputs anything. Where is it getting stuck, though? Right now, we don't know.

The results of that execution tell us what to do next: go back to the first step of the programming flow and write some code that will help us narrow down where the

#	PrintDebugginginPython-1.py	Output
1	`i = 10`	
2	`count = 1`	
3	`while count < i:`	
4	`    count = count + 1`	
5	`while count > 0:`	
6	`    count = count + 1`	
7	`print(count)`	
8		

Figure 1.3.6

#	PrintDebugginginPython-2.py	Output
1	`i = 10`	`Starting first loop...`
2	`count = 1`	`First loop done.`
3	`print("Starting first loop...")`	`Starting second loop...`
4	`while count < i:`	
5	`    count = count + 1`	
6	`print("First loop done.")`	
7	`print("Starting second loop...")`	
8	`while count > 0:`	
9	`    count = count + 1`	
10	`print("Second loop done.")`	
11	`print(count)`	
12		

Figure 1.3.7

error is occurring. In Figure 1.3.7, we've put print statements throughout the code. That way, we can look at the output and see where the code is getting stuck. We see that it completes the first loop, starts the second loop, but never finishes the second loop. Now we know that the problem is on line 8 or 9. That's far easier to solve than not knowing at all where the problem is.

Based on this, we might now notice that when we're supposed to be counting down, we're actually *adding* 1 instead of subtracting it. So, based on this new iteration through the programming flow, we revise our code.

Now we're successful. In Figure 1.3.8, we revised line 9 to subtract instead of add, ran the code again, and received the output we expected. At this point, we might go back and remove our print debugging statements; they were only there to help us resolve the problem, and now it's resolved!

#	PrintDebugginginPython-3.py	Output
1	`i = 10`	`Starting first loop...`
2	`count = 1`	`First loop done.`
3	`print("Starting first loop...")`	`Starting second loop...`
4	`while count < i:`	`Second loop done.`
5	`    count = count + 1`	`0`
6	`print("First loop done.")`	
7	`print("Starting second loop...")`	
8	`while count > 0:`	
9	`    count = count - 1`	
10	`print("Second loop done.")`	
11	`print(count)`	
12		

Figure 1.3.8

Scope Debugging in Python

To chat about scope debugging, let's imagine a little program that calculates the average from a number of grades. The grades themselves are shown in the first line of Figure 1.3.9: 100, 95, 93, and so on. Don't worry too much about how it works for now; we'll talk about everything you see here later. For now, let's focus just on what the bug might be.

In this case, we can manually calculate the average of those grades to know what output we expect: we add those numbers together on a calculator, divide by 12, and we find the average should be 89. However, right now the code in Figure 1.3.9 is outputting 6.833… so something isn't working. How do we find out what's wrong?

If we're using print debugging, we might add print statements to make sure that the program is running through everything as expected, as seen in Figure 1.3.10.

#	ScopeDebugginginPython-1.py	Output
1	`grades = [100, 95, 93, 91, 90, 89,`	6.833333333333333
	`  87, 87, 85, 85, 84, 82]`	
2	`sum = 0`	
3	`count = 0`	
4	`for grade in grades:`	
5	`    count = count + 1`	
6	`    sum = grade`	
7	`print(sum / count)`	
8		

Figure 1.3.9

#	ScopeDebugginginPython-2.py	Output
1	`print("Calculating...")`	Calculating...
2	`grades = [100, 95, 93, 91, 90, 89,`	Adding grade 1...
	`  87, 87, 85, 85, 84, 82]`	Adding grade 2...
3	`sum = 0`	Adding grade 3...
4	`count = 0`	Adding grade 4...
5	`for grade in grades:`	Adding grade 5...
6	`    count = count + 1`	Adding grade 6...
7	`    print("Adding grade ", count, "...", sep="")`	Adding grade 7...
8	`    sum = grade`	Adding grade 8...
9	`print(sum / count)`	Adding grade 9...
10		Adding grade 10...
11		Adding grade 11...
12		Adding grade 12...
13		6.833333333333333
14		
15		
16		

Figure 1.3.10

For example, maybe the problem was that the program was only adding half the grades. So, we add these print statements to count how many grades it adds. We find, however, that the program correctly touches 12 different grades, so it isn't skipping any.

Now we're at a bit of a loss. The code is correctly touching each of the twelve grades, but the sum it's computing is wrong. What's going on here? To check this further, we want to narrow our scope a bit. We know the code is wrong because we checked that the average we calculated by hand differs from the average the code is calculating. Let's now do that throughout the program as well. Instead of just comparing at the end, let's compare every step along the way.

#	ScopeDebugginginPython-3.py	Output
1	`print("Calculating...")`	Calculating...
2	`grades = [100, 95, 93, 91, 90, 89,`	Current sum: 100
	`  87, 87, 85, 85, 84, 82]`	Current sum: 95
3	`sum = 0`	Current sum: 93
4	`count = 0`	Current sum: 91
5	`for grade in grades:`	Current sum: 90
6	`    count = count + 1`	Current sum: 89
7	`    sum = grade`	Current sum: 87
8	`    print("Current sum:", sum)`	Current sum: 87
9	`print(sum / count)`	Current sum: 85
10		Current sum: 85
11		Current sum: 84
12		Current sum: 82
13		6.833333333333333
14		
15		
16		

Figure 1.3.11

In Figure 1.3.11, we're printing the sum each time we add a new grade to it. So, we can calculate what sum should be at any time: 100 after one run, 195 after two, 288 after three, and so on. Our first line appears correct, sum is 100 after one run. After two runs, though, it's 95 instead of 195. After three, it's 93 instead of 288. So, we've found the problem: sum isn't being computed properly. The only place sum is being modified is line 7, so we know the problem is on line 7. Now we can look and see the issue (though we might not understand it until Unit 2): sum isn't *adding* the next grade, it's just being replaced by it!

We fix that in Figure 1.3.12 by adding grade to the current sum, then we run the code again, and voila. We receive the correct result, 89.0. Now we could remove our debugging statements, and we'd be left with code that accomplishes the goal we set out to accomplish.

#	ScopeDebugginginPython-4.py
1	`print("Calculating...")`
2	`grades = [100, 95, 93, 91, 90, 89,`
	`87, 87, 85, 85, 84, 82]`
3	`sum = 0`
4	`count = 0`
5	`for grade in grades:`
6	`    count = count + 1`
7	`    sum = sum + grade`
8	`    print("Current sum:", sum)`
9	`print(sum / count)`
10	
11	
12	
13	
14	
15	

Output
Current sum: 100
Current sum: 195
Current sum: 288
Current sum: 379
Current sum: 469
Current sum: 558
Current sum: 645
Current sum: 732
Current sum: 817
Current sum: 902
Current sum: 986
Current sum: 1068
89.0

Figure 1.3.12

6. Advanced Debugging

Before we move on, there's one last thing we should mention. The workflow we've discussed here for debugging is pretty universal, but the specific methods we've discussed are not. Adding print statements to trace your program's execution pattern or check the status of its data while it runs are great ways to approach debugging as you first start to learn to program. By approaching things this way, debugging is another opportunity to learn to program in addition to a part of the programming process for you to learn.

However, as you get more advanced, you'll find you may need to use more advanced debugging techniques. We don't think you'll reach that point in this class, but you will at some point in the future, and it would be undesirable to stick to these simple ways of debugging when you get to more complex problems. Our goal here is simply to alert you about the other options available so that when you need them, you know that they exist.

Advanced Debugging Methods

Some of the advanced tools available for debugging include:

- **Step-by-step execution.** Some development environments will allow you to run your code one line at a time. You can watch the lines execute, allowing you to visually see the status of the program's execution. So, instead of checking to see if the previous loops were running correctly, you could simply watch. In our print debugging example, you would see the loop running over and over again.

- **Variable visualization.** Some development environments will show you a simple chart of all the data stored in memory; this removes the need for you to print out variables manually and check their values. Combined with step-by-step execution, this would allow you to watch how data is changed as the program runs, removing the need to print out the sum several times in the scope debugging example.
- **In-line debugging.** Some development environments are sophisticated enough to debug simple errors while you're actually writing the code. PyCharm, for example, can underline your code live as you're writing it to call out certain errors, as shown in Figure 1.3.13. This is especially true for syntax errors and name errors; rather than waiting for the code to run to get this kind of feedback, these environments will tell you immediately.

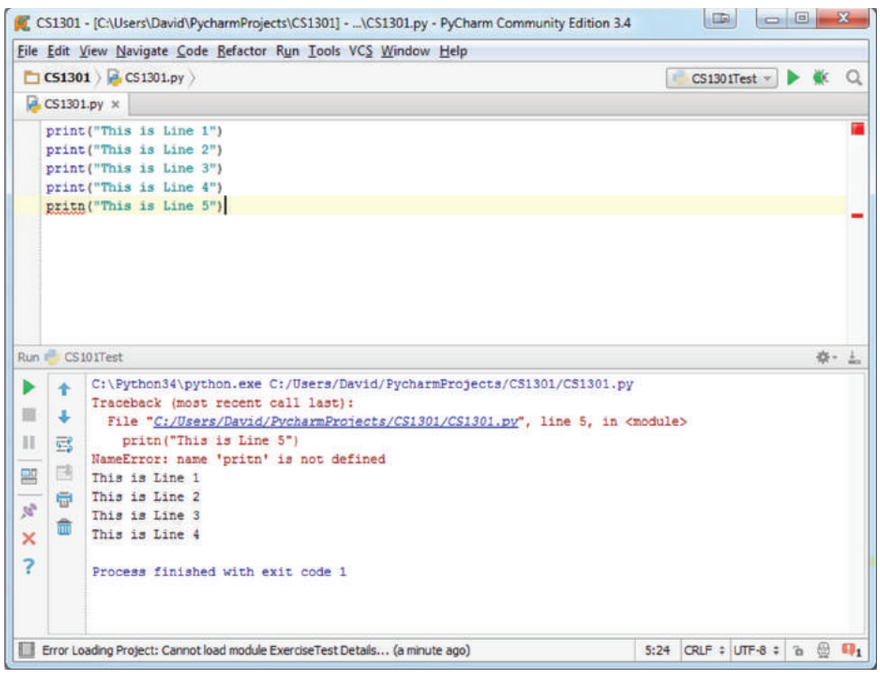


Figure 1.3.13

UNIT 2

INTRODUCTION TO PROCEDURAL PROGRAMMING

CHAPTER 2.1

Procedural Programming

1. What Is Procedural Programming?

It's time we started learning to program in earnest. We've covered the high-level ideas of writing, running, and evaluating. We've talked about the difference between compiling and running. We've talked a bit about what lines of code are and how to debug them. From this point forward, we're going to focus more on actually writing code.

We'll start with what is known as procedural programming. What is procedural programming? At this point, procedural programming is just what you would call… programming. We defined programming as giving instructions to the computer to carry out in order. That's what procedural programming is.

Procedural programming isn't the only kind of programming there is, though. You'll probably notice pretty quickly that a lot of programs you use on a daily basis couldn't be written using only the concepts we'll cover initially here. So, as we get started on procedural programming, let's take a moment to briefly describe the other common programming paradigms so that you understand the limitations of our initial approach. The goal here is to help you see the difference between the programs we're writing and the programs you use on a daily basis, as well as set the right expectations for where this material lies in the broader field of computing.

Functional Programming

Much of programming is based around the idea of "**functions**", in some circumstances referred to as "**methods**." A function is like a little machine: you drop something in, and it spits something out. We refer to what you put in as the "input" and what you get out as the "output", and we usually say that the function or method "returns" the output. Functional programming is a fundamental part of most modern programming languages, and we'll cover writing your own functions at the end of Unit 3. Throughout the unit, we'll also make use of functions and methods to do various tasks like printing text or getting user input, so you should get familiar with the syntax of using functions.

Although syntax can generally vary wildly across multiple languages, most languages have very similar syntax for using functions. The function name is given, followed by a set of parentheses. Inside the parentheses are given any input, if needed. Whatever the function returns essentially replaces this text. For example, in Figure 2.1.1 we see a function named round, which we would assume would round the number given in the parentheses to the nearest integer. This line would thus effectively be replaced by the number 5. The function named round takes a real number as input, and returns the nearest integer as the output.

Object-Oriented Programming

Object-oriented programming will be covered in the fifth unit of this material. Using procedural programming, you can tell the computer to do lots of things. You can tell it to repeat certain instructions a number of times, like repeating an operation for every file in a folder. You can tell it to decide what to do based on some conditions, like only opening a file if it's a specific file type. You can tie together commands to be called when needed. There are a lot of things you can do.

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Understand procedural programming and how it differs from other paradigms of programming;
- Describe procedural programming in the context of Python, as well as read and write comments;
- Evaluate a turtles program from the perspective of procedural programming.

Function
A segment of code that performs a specific task, sometimes taking some input and sometimes returning some output.

Method
A function that is part of a class in object-oriented programming (but colloquially, often used interchangeably with function).

Object-Oriented Programming
A programming paradigm where programmers define custom data types that have custom methods embedded within them.

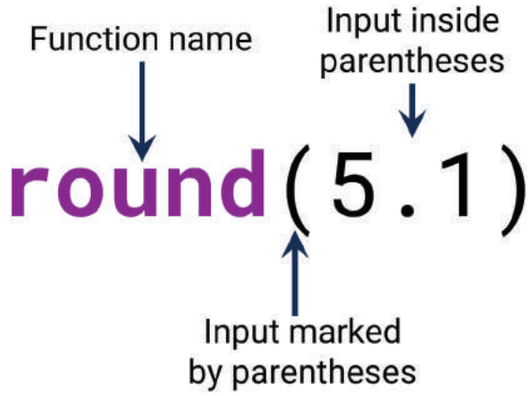


Figure 2.1.1

However, you can't teach the computer a concept it doesn't understand—not very easily, anyway. Imagine we were creating a new gradebook application. We have a concept of student information, such as first name, last name, address, and class enrollments. For us, the idea of a student as a concept with lots of information attached makes a lot of sense. In procedural programming, though, that doesn't exist. Object-oriented programming is the paradigm that lets us create new concepts and teach them to the computer. What it means is that there is some code that isn't executed in order when we execute our code; instead, it's sort of "pre-loaded" and used as needed. That breaks the imperative nature of procedural programming.

Event-Driven Programming

We often describe programs that we write as people. You give your program instructions just as you give a person instructions. However, one kind of instruction we might give a person is: wait for me to come and ask you for help. Think of a receptionist at an office. Much of their time may be spent waiting for a client or customer to come in. During that time, there are no commands to execute. This is similar to **event-driven programming**.

Event-Driven Programming
A type of programming where the program generally awaits and reacts to events rather than running code linearly.

Most programs you use are like this, actually. When you're sitting looking at an empty word processor or an Internet browser, the program is waiting for you to do something. When you do something, it triggers an event, and the event sets off some sequence of code. The code that the event sets off is just like the code we write procedurally, but it's triggered by events rather than happening automatically when we run our program.

There are numerous other programming paradigms out there, but these three are arguably the three you're most likely to think about as you go through this course's material. We'll cover object-oriented programming in Unit 5. Event-driven programming, however, won't be covered in this material because it only truly becomes valuable when you get to graphical user interfaces. The majority of what you learn here is still applicable to event-driven programming; the difference is just what causes the code to start running.

2. Procedural Programming in Python

So far, everything we've done in Python has been pure procedural programming. That's going to remain the case for a while. Our programs can be read as series of instructions executed in order to get the computer to do something. Let's look at a few examples. The syntax you'll see in these examples may not make sense right now, but don't worry; we'll cover it later. The purpose of this lesson is to preview some of the things you'll see later in this unit so that when you see them, it's a little more familiar.

#	HelloWorld-1.py	Output
1	`#Prints the text "Hello, world"`	Hello, world
2	`print("Hello, world")`	
3		

Figure 2.1.2

Hello, World

The simplest example of procedural programming in Python in action is the first program we wrote: Hello, World. Hello, World was a single line, which represented a single command to the computer. When instructed, the computer executed that command. Note that the first line in Figure 2.1.2, preceded by a pound sign or hash mark, is designated a "comment": the computer ignores this. We'll talk more about these in a bit.

A series of these lines would execute that command a series of times. The computer executes each line one by one. If one line is repeated several times, it gets executed several times. In Figure 2.1.3, the line is to print a string of characters. We know it's a string of characters because it's enclosed in quotation marks: this is how we tell the computer to interpret it as text, not as computer code.

#	HelloWorld-2.py	Output
1	`#Prints the text "Hello, world"`	Hello, world
2	`print("Hello, world")`	Hello, world
3	`#Prints the text "Hello, world"`	Hello, world
4	`print("Hello, world")`	Hello, world
5	`#Prints the text "Hello, world"`	Hello, world
6	`print("Hello, world")`	
7	`#Prints the text "Hello, world"`	
8	`print("Hello, world")`	
9	`#Prints the text "Hello, world"`	
10	`print("Hello, world")`	
11		

Figure 2.1.3

Data Types and Variables

Strings of characters aren't the only things we print. Oftentimes, we'll print integers, decimal numbers, and more complex information as well.

These are referred to as data types. Line 4 in Figure 2.1.4, for example, prints the integer 5. 5 is an integer, meaning that it is a whole number (no decimals). 5 has a different data type from "Hello, world"; where "Hello, world" was a string of characters, 5 is an integer. 5.1 is another different kind of data type: it's a decimal

#	DataTypesandVariables-1.py	Output
1	`#Gets the library we need to see the date`	5
2	`from datetime import date`	5.1
3	`#Prints the integer 5`	2016-09-17
4	`print(5)`	
5	`#Prints the number 5.1`	
6	`print(5.1)`	
7	`#Prints today's date`	
8	`print(date.today())`	
9		

Figure 2.1.4

number, which in Python we call a float (which stands for "floating point number"). And there are still more complex data types, like dates and times, that we can access.

Most often, we don't print or use values like those directly. Typically, we store them in variables, and then use the variables in our programming. In Figure 2.1.5, instead of printing 5, 5.1, and today's date directly, we first store them in the variables `myInteger`, `myFloat`, and `myDate`. Lines 3, 7, and 11 are assignment statements: they assign values to those variables. In Python, this also creates the variable. Then, after that, we can use the variable and get its value when we need it.

#	DataTypesandVariables-2.py	Output
1	`from datetime import date`	5
2	`#Creates myInteger with the value 5`	5.1
3	`myInteger = 5`	2016-09-17
4	`#Prints myInteger`	
5	`print(myInteger)`	
6	`#Creates myFloat with the value 5.1`	
7	`myFloat = 5.1`	
8	`#Prints myFloat`	
9	`print(myFloat)`	
10	`#Creates myDate with today's date`	
11	`myDate = date.today()`	
12	`#Prints myDate`	
13	`print(myDate)`	
14		

Figure 2.1.5

The usefulness of this is that we can have a single variable name that changes its value over time. Imagine you have a variable representing a bank account balance, as in Figure 2.1.6. The value of the bank account balance changes over time, but the idea of having a balance is consistent. So, the variable name, `myBalance`, stays the same, but its value can change.

#	DataTypesandVariables-3.py	Output
1	`#Creates myBalance with the value #100.0`	100.0
2	`myBalance = 100.0`	105.0
3	`#Prints myBalance`	
4	`print(myBalance)`	
5	`#Gives myBalance the value 105.0`	
6	`myBalance = 105.0`	
7	`#Prints myBalance`	
8	`print(myBalance)`	
9		

Figure 2.1.6

Logical Operators

The usefulness of variables is in our ability to interact with and modify them, and one of the most fundamental ways we interact with and modify variables is with operators. There are two kinds of operators: logical operators and mathematical operators.

Within logical operators, there are two subtypes of operators: relational operators and boolean operators. Both are reserved words or symbols like > and `or` that perform some operation on variables. Relational operators check for the relationships between values, whether they're contained in variables or not.

Lines 7 and 10 in Figure 2.1.8 contain the relational operator "is greater than." This operator checks if the first number is greater than the second, and responds `True` if so. Otherwise, it responds `False`. Note it can work either on raw values

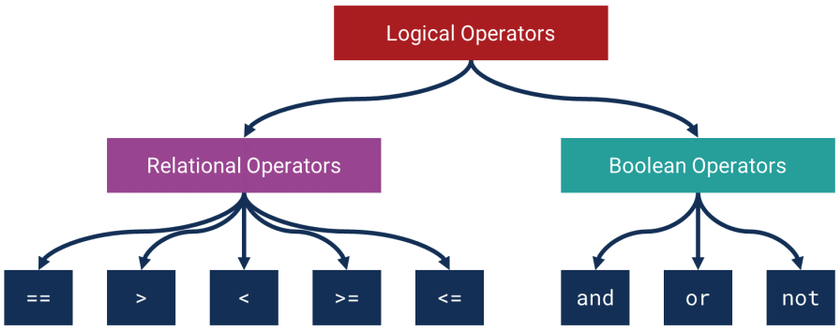


Figure 2.1.7

#	LogicalOperators-1.py	Output
1	`#Creates myNum1 with the value 1`	`True`
2	`myNum1 = 1`	`True`
3	`#Creates myNum2 with the value 2`	
4	`myNum2 = 2`	
5	`#Prints True if 2 is greater than 1,`	
6	`#False otherwise`	
7	`print(2 > 1)`	
8	`#Prints True if myNum2 is greater`	
9	`#than myNum1, False otherwise`	
10	`print(myNum2 > myNum1)`	
11		

Figure 2.1.8

(like 2 and 1 in line 7) or values stored in variables (like `myNum2` and `myNum1` in line 10). Relational operators all check for the existence of certain relationships and say `True` if the relationship exists, or `False` if it doesn't. The most common relational operators are "greater than", "greater than or equal to", "less than", "less than or equal to", and "is equal to."

Boolean operators are operators that themselves only act on `True` and `False` values (called boolean values). For example, the code in Figure 2.1.9 slightly alters the previous block to check if `myNum2` is between two other numbers; or, in other words, if it's greater than one number *and* less than another number. That `and` is a boolean operator; it operates on the results of the two relational operators on either side.

#	LogicalOperators-2.py	Output
1	`#Creates myNum1 with the value 1`	`True`
2	`myNum1 = 1`	`True`
3	`#Creates myNum2 with the value 2`	
4	`myNum2 = 2`	
5	`#Creates myNum3 with the value 3`	
6	`myNum3 = 3`	
7	`#Prints True if 2 is between 1 and 3,`	
8	`#False otherwise`	
9	`print(2 > 1 and 2 < 3)`	
10	`#Prints True if myNum2 is between myNum1`	
11	`#and myNum3, False otherwise`	
12	`print(myNum2 > myNum1 and myNum2 < myNum3)`	
13		

Figure 2.1.9

Mathematical Operators

Like boolean operators, mathematical operators operate on variables. Specifically, they perform mathematical operations. We can add to our running example to show the role of mathematical operators:

#	MathematicalOperators.py	Output
1	`#Creates myNum1 with the value 1`	`False`
2	`myNum1 = 1`	`False`
3	`#Creates myNum2 with the value 2`	
4	`myNum2 = 2`	
5	`#Creates myNum3 with the value 3`	
6	`myNum3 = 3`	
7	`#Prints True if 3 > 1 + 2,`	
8	`#False otherwise`	
9	`print(3 > 1 + 2)`	
10	`#Prints True if myNum3 > myNum1 + myNum2,`	
11	`#False otherwise`	
12	`print(myNum3 > myNum1 + myNum2)`	
13		

Figure 2.1.10

In Figure 2.1.10, instead of just comparing two numbers like 1 and 2, the code compares the sum of two numbers to a third number. 3 is not greater than 1 + 2 (rather, it's equal to it), and so `False` is printed both times. We can do similar operations for differences, products, quotients, and some more exotic operations as well.

Don't worry if you're fuzzy on everything we've talked about so far: you're supposed to be! The goal here has been to preview what you'll see later. The only thing you should come away with from this lesson is: there exist these things called variables and operators, and they interact together in useful ways.

3. Comments and Documentation

Comments
Notes from the programmer supplied in-line alongside the code itself, designated in a way that prevents the computer from reading or attempting to execute them as code.

Documentation
Collected and set-aside descriptions and instructions for a body of code.

Before we move on, there's one more fundamental notion of programming that we should cover: **comments** and **documentation**. When we write code, we're writing in the computer's language; we're translating our understanding of what we want it to do into the computer code. However, later on, someone else might need to come along and modify our code. Or, maybe in the future, we'll return and need to remember how the code works. Because it's written in the computer's language, though, it isn't easy for us to understand. We have to translate it back into human language.

That's where commenting and documentation come in. Comments are a way of explaining how our code works *alongside* the code itself. A comment is essentially some text written into the code with a certain character or label instructing the computer to ignore it. It's only there for people to read, so the computer pays no attention to it. When another human comes along to modify the code, though, it gives an explanation they can understand more easily. These comments are like notes to future developers. Imagine you were showing your code to someone: what would you say about it beyond what is there in the code? That's the kind of explanation you want to put in comments.

Comments

There are two general kinds of comments that we'll see. First, in-line comments are comments that are placed right alongside the code that we write to explain what it does at a very detailed level. They might explain why we wrote some code a certain way, or translate some strange reasoning into a description more understandable by humans. They might contain notes or questions; it isn't uncommon for programmers to include comments suggesting that certain code should be rewritten or has some known bugs.

The other kind of comment is larger. Before big segments of code, at the beginning of files, or before longer functions, we might also put larger explanations for what the code that follows tries to do. This is especially useful in bigger programs, where it's important to understand not just how individual lines of code

work, but how the entire program fits together. There even exist some languages for creating comments like these that can be automatically extracted into manuals and explanations.

Self-Documenting Code

Explaining with comments isn't the only way to make your code more readable by people, though. Another idea in programming is called **self-documenting code**. We have enough leeway in how we write code that we can write it in ways that make it more clear what the code does. We can title our variables, our functions, our files any way we want. If you had a variable that represents how many words are in a file, why call it `a` when you can call it `numWords`? That's the notion of self-documenting code: writing code that shows what it means.

Self-Documenting Code
Code whose variables and functions are named in a way that makes it clear what their underlying content and operations clear to the reader.

4. Comments and Documentation in Python

Before we move on to writing a lot of code, let's quickly look at comments and documentation and how they work in Python.

In-Line Comments

In-line comments give line-by-line descriptions of what a segment of code does. For example, let's go back to one of our earlier blocks of code.

In Figure 2.1.11, we have a conditional statement that tells the computer to change what it does based on the value of a certain variable. Again, don't worry too much about understanding this code just yet. For now, let's focus on the comments. Notice that the presence of commented lines doesn't change the operation of the code. Anything that appears after the pound sign is ignored by the computer. By convention, in Python we usually put comments on their own lines before the code that they describe (as in lines 1, 4, and 7), but you can also put a comment after the end of a line of code (as in lines 3 and 6). Both are shown in Figure 2.1.11.

This way, we can explain to classmates, teammates, graders, and our own future selves how the code we write works. Writing comments is a habit to get into as early as possible; it will make both your and your colleagues' lives much easier.

#	InLineComments.py	Output
1	`#Creates the variable i`	`Greater than 0.`
2	`i = 1`	
3	`if i > 0: #Checks if i is greater than 0`	
4	`    #Prints if i is greater than 0`	
5	`    print("Greater than 0.")`	
6	`if i > 1: #Checks if i is greater than 1`	
7	`    #Prints if i is greater than 1`	
8	`    print("Greater than 1.")`	
9		

Figure 2.1.11

Code Block Comments

We use the exact same syntax in Python for explaining larger blocks of code as well. For example, let's revisit the function we wrote in the last lesson.

Above the function in Figure 2.1.12, we've written three comments that explain what the function does. This way, a person coming along and viewing our code later could easily figure out what the code segment was intended to do and more easily revise it if necessary. In large programs with lots of functions, lots of different files, and lots of people working together, this kind of documentation is essential.

```
#  CodeBlockComments.py
1  #Countdown function
2  #Prints the numbers from 1 to i
3  #on a single line
4  def countdown(i):
5      for j in range(1,i + 1):
6          print(j, end = " ")
7      print()
8  countdown(10)
9  countdown(5)
10 countdown(2)
11
```

```
Output
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5
1 2
```

Figure 2.1.12

Self-Documenting Code

Finally, wherever possible, we should write code that documents itself based on the names we choose for things. The function in Figure 2.1.12 is an example: we could have called the countdown function `functionA`, or `foo` or `davidsFunction`, but none of these suggests what the function does. `countdown` does suggest what the function does. An even better name might have been `countUpToANumber`. There's no reason to be brief!

```
#  SelfDocumentingCode-1.py
1  i = 5
2  for j in range(0, i):
3      print("Hello, world")
4
5
6
7
```

```
Output
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
```

Figure 2.1.13

We can extend this to variables as well. For example, Figure 2.1.13 we're using two variables, `i` and `j`, but `i` and `j` are simply generic labels. Just reading their names doesn't explain what they mean at all. So, what do we want to do to make this code easy to read instead?

In Figure 2.1.14, we've changed `i` to `numberOfTimesToPrint`. That immediately reveals that the variable determines how many times the message is printed. Someone reading our code could immediately understand what it does.

What about `j` though? We replaced it with just an underscore—how is that any clearer? This is an interesting Python convention; you don't have to do this, but it's generally accepted that if you have a variable that (a) must be created and (b) will never be used, you call it `_`. It's a strange little Python thing. You can still name the variable anything you want, but calling it `_` is a way of telling the reader, "Hey, you won't see this variable used anywhere else."

```
#  SelfDocumentingCode-2.py
1  numberOfTimesToPrint = 5
2  for _ in range(0, numberOfTimesToPrint):
3      print("Hello, world")
4
5
6
7
```

```
Output
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
```

Figure 2.1.14

As we explore writing code, notice how we use comments and self-documenting code throughout. This is our way of making this material easier for you to understand, but it also sets a good example of how you should structure and write your own code.

5. Procedural Programming and Turtles

We've seen code running simple sets of lines in order, like printing "Hello, world" several times. Now let's see procedural code in action with the Python turtle graphics library. Here, we'll show two examples of how running a series of lines of code in order can lead to interesting results. For now, don't worry too much about how these lines work; focus instead on how the lines run one-by-one, executing the given command.

Drawing a Hexagon

Let's start with something very simple: drawing a hexagon. If you're drawing a hexagon by hand, you do so by drawing six lines, each one at a certain angle from the previous one. That's exactly how our turtle graphics library draws a hexagon as well, as shown in DrawingaHexagon.py.

One-by-one, the computer executes these lines of code. When it encounters the `forward()` function, it pushes the turtle forward by the given distance, drawing a line behind it. When it encounters the `right()` function, it rotates the turtle to the right by the given number of degrees. So, one command at a time, it draws the hexagon.

`turtle.forward(distance)`
Takes as input distance as a float and moves the turtle forward the given distance.

`turtle.right(angle)`
Takes as input an angle as a float and rotates the turtle the given number of degrees.

Drawing a Rainbow

Let's draw something more complex. How about a rainbow? Or, at least, something that resembles a rainbow. With a relatively small number of lines of code, we can have Python use the turtle graphics library to draw a series of somewhat concentric circles of different colors, as shown in DrawingaRainbow.py.

Voila! One line of code at a time, we've drawn something resembling a rainbow. Each line of code served a purpose, whether it was changing the pen's color, changing the pen's size, or drawing a circle of a certain radius.

This was not the most efficient way to do this, however. As we go on in this material, we'll revisit this and see how to draw this same figure in only 5 lines of code instead of 15. Note that aiming for as few lines of code as possible is not the goal, but we can certainly be more efficient than this.

CHAPTER 2.2

Variables

1. What Is a Variable?

Variables are possibly the most fundamental element of programming. There really isn't much you can do without variables. You're probably familiar with variables from your days learning algebra. Variables aren't really any different here. A variable is a name that holds a value. The name stays the same while the value can change. In algebra, that variable was often x.

To use that variable, you'd give it a value. The equation would then do some stuff to that value, and you'd get a result for y. If you didn't enjoy math, though, don't worry; to be honest, I never fully understood concepts like variables and functions in math until I learned them in computer science. You don't need to know any math to learn about computing. To me, writing code is more like writing an essay than doing a math problem.

Examples of Variables

A variable is a name that holds some **value**. It's like a question with an answer. The question always exists, but the answer to the question might change. Let's think of some examples:

- How many kids do I have? The variable would be the count of David's kids, and the value would be one. That might change, though.
- What's the current stock price of Microsoft? The variable would be Microsoft stock price, and the current value would be 53.74.
- What color shirt am I wearing? The variable would be David's shirt color, and the current value would be blue.

Notice how in each case, there's a variable and a value. The variable can take on different values. Notice also how unlike in math, variables don't have to represent numbers. You could have variables that represent any number of things, such as names, colors, and dates.

Variables are the heart of programming. Nearly everything we do involves manipulating variables. We'll use variables to represent the information in which we're interested, like stock prices or usernames. We'll use variables to control how our programs run, like counting repeated actions or checking if something has been found.

2. Variables in Python

You can't solve y = 2x + 1 unless I give you a value for x or y. Similarly, in order to actually use a variable in Python, it has to have a value. The value can change later, but it needs a value to start with. In Python, we create the variable by giving it a value.

In Figure 2.2.1, I've created a variable on line 2 called x by giving it the value 5. This is called an assignment statement: it assigns a value to a variable. When I print the variable on line 4, it shows me its value. That's an important thing to remember: when we're writing code, we deal with variables; when we run code, we see the values that go into these variables.

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Define variables and values and describe the relationship between them;
- Declare and initialize variables in Python;
- Use variables to write dynamic turtle programs.

Variables
Alphanumeric (letters and numbers) identifiers that hold values, like integers, strings of characters, and dates.

Value
The content of some variable. The variable `myAge` might hold the value 29. The variable `yourName` might hold the value "Adelene".

#	VariablesInPython.py	Output
1	`#Create the variable x with the value 5`	5
2	`x = 5`	
3	`#Print the value of the variable x`	
4	`print(x)`	
5		

Figure 2.2.1

Different Kinds of Variables

Just as we talked about lots of different kinds of variables previously, so also Python variables can be used to store lots of different kinds of information. Let's take a few examples:

#	DifferentKindsofVariables.py	Output
1	`from datetime import date`	-2
2	`#Creates aNumber with the value -2`	7.1
3	`aNumber = -2`	Hello, world!
4	`#Creates aDecimal with the value 7.1`	True
5	`aDecimal = 7.1`	2016-09-22
6	`#Creates aString with the value "Hello,`	
7	`world"`	
8	`aString = "Hello, world!"`	
9	`#Creates aBoolean with the value True`	
10	`aBoolean = True`	
11	`#Creates aDate with the value of today's date`	
12	`aDate = date.today()`	
13	`print(aNumber)`	
14	`print(aDecimal)`	
15	`print(aString)`	
16	`print(aBoolean)`	
17	`print(aDate)`	
18		

Figure 2.2.2

In Figure 2.2.2 we see five variables, named `aNumber`, `aDecimal`, `aString`, `aBoolean`, and `aDate`. Remember, we could have named these anything we wanted, but we gave them names that made it clear what they are. Think of this like giving names to a pet: you *could* name a cat "Fido" and a dog "Tiger," but you wouldn't be surprised if these names confused people. The same way, you could name your variables `phineas`, `celery`, and `sovngarde`, but these names wouldn't tell the reader anything about what they are.

Each of these five variables is assigned a value, but notice that the values themselves differ. `aNumber` is given an integer, `aDecimal` a number with a decimal, `aString` a message in text, `aBoolean` a true or false value, and `aDate` a representation of today's date. We'll talk more later about the kinds of data that variables can hold—for now, just notice how variables can holds lots of different kinds of data.

Python and Typing

There are a few things worth noting here before we move on to that, though. First, notice that if you didn't see lines 1 through 12, you wouldn't know what kind of data a certain variable held. If you just saw the line `print(aNumber)` without seeing the lines that preceded it, you wouldn't know if `aNumber` was holding a number, a string of characters, a date, or something else entirely.

Second, notice that we didn't actually name what kind of information the variables were holding. We didn't say, "`aNumber` is going to hold an integer, and here's what integer it's going to hold." The fact that `aNumber` is holding an integer was defined by the fact that we put an integer on the right side of the assignment statement. That means that we can change what kind of information it's

holding just by changing what's on the other side of the assignment statement, as seen in Figure 2.2.3.

#	PythonandTyping.py	Output
1	`#Create a variable and give it the value 5`	`5`
2	`aVariable = 5`	`I'm not an integer anymore!`
3	`#Print the variable's value`	
4	`print(aVariable)`	
5	`#Reassign the variable to a different type`	
6	`aVariable = "I'm not an integer anymore!"`	
7	`#Print the variable's new value`	
8	`print(aVariable)`	
9		

Figure 2.2.3

The first time we print `aVariable`, it's an integer. The second time, it's a string of characters. The type of value it held changed because what we assigned to it changed. This is a unique feature of Python; in other languages, you have to choose what kind of data a variable will hold when you create the variable.

Naming Rules and Conventions

We don't need to name variables with simple letters like x and y in math. In fact, we shouldn't. When we're choosing variable names, there are two things we need to keep in mind: a set of rules, and some conventions.

First, there are a few rules that cover what variable names we can use. They are:

- Variable names can contain only letters, numbers, and underscores. No spaces or special characters.
- Variable names must start with letters. Underscores are technically allowed, too, but we usually only use these in certain situations.
- Variable names must not duplicate certain reserved words. We'll talk about the reserved keywords later.

When you violate these rules, you'll usually get a syntax error, as shown in Figure 2.2.4. Sometimes, though, the computer will give you a different error depending on how it interpreted your input.

#	NamingRulesandConventions-1.py	Output
1	`my$variable = 5`	`File "NamingRulesandConventions-1.py", line 1`
2		`    my$variable = 5`
3	`my variable = 5`	`      ^`
4		`SyntaxError: invalid syntax`
5	`5variable = 5`	
6		
7	`for = 5`	
8		

Figure 2.2.4

However, beyond these rules, there are certain conventions we want to follow. The computer isn't going to yell at us if we don't follow these, but other people that have to work with our code might. The main convention is that our code should be self-documenting. What that means is we should be able to read the variable name and know what kind of data it's holding. When I see variable names like `a`, `foo`, and `qwerty` in Figure 2.2.5, I don't know what they're holding. They might be integers, strings, times, or something else. Even if I knew the type was an integer, I wouldn't know if they were storing an age, a shoe size, or a height in centimeters. So instead of using vague variable names like `a`, `foo`, and `qwerty`, we might use variable names like `age`, `firstName`, or `catCount`. Just by reading these variable names, we can infer `age` is storing someone's age, `firstName` is storing a name like David or Lucy, and `catCount` is storing how many cats the person has.

```
# NamingRulesandConventions-2.py
1  #Bad variables
2  a = 30
3  foo = "David"
4  qwerty = 2
5
6  #Good variables
7  age = 30
8  firstName = "David"
9  catCount = 2
10
```

Figure 2.2.5

3. Assigning Variables

We've said that variables are like questions, and values are like their answers. The variable sticks around while the value changes. There are a few principles that go along with this, though.

Give Values to Variables

The first is that we have to make sure that the order of that relationship is correct. We can't give variables to values—we always give values to variables. We don't want to ask, "What is 2?" and have the answer be, "It's the number of cats I have!" Unless you're on Jeopardy, that order doesn't make sense. Instead, you want to ask, "How many cats do I have?" and get the answer, "2." In nearly every programming language, we put the variable on the left, followed by the assignment operator, followed by the value we're assigning or giving to the variable. We'd say x equals 5, not 5 equals x. That says, "Give x the value 5."

Assign Values before Using Variables

Second, to use variables, we generally must have already assigned them values. When we use a variable without actually giving it a value, we trigger some kind of name or null error, depending on the programming language. One way to break the rules of programming is by trying to use a variable that doesn't have a value yet.

Imagine if I asked you, what color shoes am I wearing? The variable is shoe color, and you know the value would be some color, like black or brown. You don't know the answer, though, so you can't answer the question. Now, imagine if I were to tell you: "Please paint this wall the same color as my shoes." If you don't know what color my shoes are, you're unable to complete my instruction. That's what happens when we use a variable in programming that doesn't have a value. The program can't proceed, so it throws up an error and stops.

In a few places, we can use this to our advantage. Instead of just using the variable as if it has a value, we can actually check: Does this variable have a value? The computer has a way of saying, "I don't know!", called "null." **Null** is the value for any variable that hasn't been assigned a value otherwise. So, if a variable's value is "null," then we know it hasn't received a value, and we can check that before moving forward.

Null
The "value" a variable has when it doesn't actually have a value.

4. Assigning Variables in Python

Let's see what happens when we make some of the mistakes we've talked about in Python. We've already talked about some of the naming requirements in Python: variables can only have letters, numbers, and underscores; they must start with letters; and they must not duplicate any keywords. Now we've also talked about two more rules: we must give values to variables, not give variables to values, and we may not use variables before they have been given values. Let's tackle the first one.

Giving Variables to Values (Wrong!)

In Figure 2.2.6, when we give `myNumber` the value 5 on line 1, things work fine! We print `myNumber` and see its current value, 5. When we try to give `myNumber` to 5 on line 3, however, we get an error. `SyntaxError` means that the way we've written a statement doesn't work, and "can't assign to literal" is the computer's way of saying we can't give a variable to a value.

#	AssigningVariablestoValues-1.py	Output
1	`myNumber = 5`	`5`
2	`print(myNumber)`	`File "AssigningVariablestoValues-1.py", line 3`
3	`5 = myNumber`	`    5 = myNumber`
4		`    ^`
5		`SyntaxError: can't assign to literal`
6		
7		
8		

Figure 2.2.6

We can see this happen in other ways, too. We also can't assign values to one another, or set other kinds of values equal to variables. Whenever we have a value on the left side of the assignment statement, we'll get that syntax error. Both lines 1 and 2 of Figure 2.2.7 would generate a `SyntaxError`.

#	AssigningVariablestoValues-2.py	Output
1	`3 = 5`	`File "AssigningVariablestoValues-2.py", line 1`
2	`"Hello, world!" = myString`	`    3 = 5`
3		`    ^`
4		`SyntaxError: can't assign to literal`
5		
6		
7		

Figure 2.2.7

The other rule is that we can't use variables before they've been assigned values. This is actually an easier problem to avoid in Python than in other languages because in Python, we create variables by assigning them values. In Figure 2.2.6, we created the variable `myNumber` by setting it equal to the value 5. There's no way to accidentally create a variable without assigning it a value. However, there are a couple of other ways we can accidentally try to use variables that don't yet have values.

#	AssigningVariablestoValues-4.py	Output
1	`print(myUnassignedNumber)`	`Traceback (most recent call last):`
2		`File "AssigningVariablestoValues-4.py", line 1, in <module>`
3		`    print(myUnassignedNumber)`
4		`NameError: name 'myUnassignedNumber' is not defined`
5		
6		
7		
8		

Figure 2.2.8

In Figure 2.2.8, we've skipped that initial assignment statement. We jump straight to trying to print a variable that doesn't yet exist! When that happens, we get this `NameError`, and it tells us that the name `myUnassignedNumber` is not yet defined. So, if you accidentally use a variable name without creating it, you'll see this error. This can happen for a number of reasons you might not expect. For example, if you misspell a variable name, you might encounter this error because the misspelled version hasn't been created yet.

```
# AssigningVariablestoValues-5.py
1 myNullVariable = None
2
```

Figure 2.2.9

It's also possible to manually create a variable without a value. We can do so by setting it equal to null manually, as seen in Figure 2.2.9. However, when we do this, it behaves a little differently from a variable we never initialized in the first place. None is a different value from just no value, as seen in Figure 2.2.10.

```
# AssigningVariablestoValues-6.py
1 myNullVariable = None
2 print(myNullVariable)
3 print(myUndeclaredVariable)
4
5
6
7
8
9
```

```
Output
None
Traceback (most recent call last):
File "AssigningVariablestoValues-6.py",
line 3, in <module>
    print(myUndeclaredVariable)
NameError: name 'myUndeclaredVariable'
is not defined
```

Figure 2.2.10

In Figure 2.2.10, printing `myNullVariable` prints `None`, indicating it has no value. Printing `myUndeclaredVariable` throws an error, indicating that the variable name has never been created in the first place. Nonetheless, we cannot use `myNullVariable` for anything except comparisons to check if its value is `None` or not.

5. Data Types

Data Type
The type of content a variable holds, like an integer or a string of characters.

Every variable has a name, and it should have a value as well if we're using it. There's a third thing that variables have: **types**. Every variable has a type of information it's storing.

Basic Data Types

There are some basic types you'll see very often:

- Integers, or whole numbers.
- Real numbers, or numbers that can have decimals.
- Characters, like individual letters and numbers.
- Strings, which are collections of characters in a row.
- Booleans, which just hold either true or false.

However, not all variables hold these basic types. Variables can hold nearly any type of information you can imagine. You could have variables that hold colors or pictures. Later in the class, we'll even talk about how you can create your own data types. For example, you could create a `Person` type that includes a person's name, birthday, and favorite food. We'll talk about that later.

Importance of Data Types

The type of a variable is very important in a lot of ways. First, we generally have difficulty comparing variables of different types. For example, if the value of `a` is 3 and the value of `b` is 5, then it's easy for me to ask: what's bigger, `a` or `b` (3 or 5)? But if `a` represents the number 3 while `b` represents the color Blue, it's hard for me to ask that same question. What's bigger, 3 or Blue?

The type of variable is also important because there might be certain operations that we can only do on certain types of data. For example, we can multiply integers easily. $3 \times 5 = 15$. How do you multiply strings of characters? What's "Hello, world!" × "Hi, I'm David!"? The operations only make sense in the context of certain types of data.

6. Data Types in Python

In Python, a variable gets its type from the value which is assigned to it. If you assign an integer to a variable, the variable takes on the type integer. If you then assign a string of characters to it, the variable then takes on that new type. So, variables really just hold values, and the values themselves have types.

Common Types

In Python, there are a handful of common **primitive** data types we will deal with frequently. You've seen them before and they're shown in Figure 2.2.11, but let's list them more clearly:

- `int`, short for "integer." An `int` can be nearly any positive or negative non-decimal number. For example, 0, –1, and 911294 would all be type `int`.
- `float`, short for "floating point number." Essentially, a `float` is a real number, including a decimal value. For example, 5.1, –6.9, and 0.00005 would be of type `float`. Note also we might have a `float` like 1.00 if we need extra precision or if we might need decimal values later.
- `str`, short for "string of characters." Any textual message is a string. When we're coding, strings are surrounded by quotation marks, either single or double. For example, "Hello, world" and 'Hello, world' are both strings. Note that a string could be holding all numbers, like "911294"—numbers are, after all, themselves characters.
- `bool`, short for "boolean." A boolean holds either `True` or `False`, nothing else. We use `bool` a lot to perform logical statements in our programs. For example, the logic, "If this file exists, then open it" would be represented by something like `if(fileExists)`, where `fileExists` is either `True` or `False`.

Author's Note
Note that if you learn a language like C or Java, the term 'primitive' for data types will take on a stricter meaning. In the technical sense, Python does not have 'primitive' data types in the way these other languages do, but the data types we describe here are often used in the same way as primitives in other languages.

#	CommonTypes.py	Output
1	`myInteger = 5`	`<class 'int'>`
2	`print(type(myInteger))`	`<class 'float'>`
3	`#Adding the .0 forces this to become a float`	`<class 'str'>`
4	`myFloat = 5.0`	`<class 'bool'>`
5	`print(type(myFloat))`	
6	`#Putting it in quotes makes it a string`	
7	`myString = "5.0"`	
8	`print(type(myString))`	
9	`#The words 'True' and 'False' are reserved`	
10	`#for boolean values`	
11	`myBoolean = True`	
12	`print(type(myBoolean))`	
13		

Figure 2.2.11

Most other languages have additional primitive data types. For example, most languages have a `char` data type that represents a single character, but in Python, this is instead simply represented by strings with only one character. Other languages also differentiate more ways of storing numbers based on how big the number will get, but Python takes care of this automatically.

Python supplies numerous other data types as well, like complex numbers, dates, lists, and many more. Some of these will be covered in this material, while you may look for others elsewhere. These four primitive types, however, will form a significant amount of the time we spend dealing with variables.

The type() Function

Python also gives us a handy function for checking a variable's type: the `type()` function. When we enter `type(variable)`, we're given the type of the variable

`type(variable)`
Takes as input some variable or value directly and returns the type of the variable such as an integer or string of characters.

in parentheses. We'll talk more about functions and how they're used in the future, but for now, just pay attention to how we're using it in Figure 2.2.12.

#	TheTypeFunction.py	Output
1	`myVariable = 5`	`<class 'int'>`
2	`print(type(myVariable))`	`<class 'str'>`
3	`myVariable = "Hello, world!"`	`<class 'float'>`
4	`print(type(myVariable))`	`<class 'bool'>`
5	`myVariable = 7.2`	
6	`print(type(myVariable))`	
7	`myVariable = False`	
8	`print(type(myVariable))`	
9		

Figure 2.2.12

Before even thinking about the `type()` function, notice first that this is the first time we've used multiple functions at the same time! Remember, `print()` is a function, and we're telling `print()` to print `type()`. How does this work? Python reads from left to right the way most of us do, so it first tries to execute the `print()` function. However, it discovers it can't execute the `print()` function until it executes the `type()` function: it's only after executing the `type()` function that it knows what to print. So, it executes the `type()` function, and whatever it outputs takes the place of the function itself. So, Python essentially translates `print(type(myVariable))` on line 2 into `print("<class 'int'>")`. After that translation, `print()` can run.

Each time we assigned a different type of data to `myVariable` in Figure 2.2.12, the type changed. <type 'int'> means that initially, the variable was an integer. <type 'str'> means that then, the variable was a string of characters. <type 'float'> is the computer's way of saying the variable is a number with a decimal; float stands for "floating point." <type 'bool'> means that the variable is holding either `True` or `False` as its value.

Mixing Types

In the next chapter, we'll discuss operators. Operators are the most basic way we act on variables. Some of the basic operators are mathematical, things like addition and subtraction, so let's briefly talk about the interaction between types and operators. For example, what happens when we try to multiply two variables with different types? Figure 2.2.13 shows the answer.

#	MixingTypes-1.py	Output
1	`myNumber1 = 5`	`15`
2	`myNumber2 = 3`	`Traceback (most recent call last):`
3	`myString = "Hello, world"`	`File "MixingTypes-1.py", line 6, in <module>`
4	`myFloat = 5.1`	`    print(myFloat * myString)`
5	`print(myNumber1 * myNumber2)`	`TypeError: can't multiply sequence by non-int of type 'float'`
6	`print(myFloat * myString)`	
7		
8		

Figure 2.2.13

For now, don't worry too much about what these operators do; think of them just in the arithmetic sense, multiplying numbers together. As we can see, when we multiply two integers together on line 5, things work just fine. When we try to multiply a float by a string on line 6, we receive a `TypeError`, which means that the data types we're using aren't compatible—at least, not with the operator we're using.

Python has some unexpected operations, though. For example, we just saw that we can't multiply a float by a string. That means we can't multiply an integer by a string either, right?

#	MixingTypes-2.py	Output
1	myNumber1 = 5	Hello, worldHello, worldHello,
2	myString = "Hello, world"	worldHello, worldHello, world
3	print(myString * myNumber1)	
4		

Figure 2.2.14

Check out Figure 2.2.14. Surprise! We can multiply a string by an integer, as seen on line 3. It duplicates the string that number of times given by the number. That's one of the strange things Python does, and it's a good example of how it can be good to just play around with the language and see what happens—frankly speaking, I only discovered that the multiplication operator works this way when trying to write some code that generates an error!

We cannot, however, multiply a string by another string. In Figure 2.2.15, we receive another `TypeError`, indicating that we're not using these data types in a compatible way.

#	MixingTypes-3.py	Output
1	myString1 = "Hello, world"	Traceback (most recent call last):
2	myString2 = "World, hello"	File "MixingTypes-3", line 3, in
3	print(myString1 * myString2)	<module>
4		print(myString1 * myString2)
5		TypeError: can't multiply sequence
6		by non-int of type 'str'
7		

Figure 2.2.15

7. Type Conversions in Python

There will be times when we need to convert among the different data types. For example, imagine writing a program that reads some text in from a file. It encounters the text "5554321." Should this be stored as the string of text "5554321" or as the number 5,554,321? Or, imagine it encounters the worse "False". Should this remain as the text "False" or be stored as the boolean value `False?` The correct answer will depend on the purpose of the code, and so we need a way to convert between the different types.

Converting to Strings

Fortunately, Python provides a very simple way of accomplishing this. First, let's look at converting numbers into strings. It will become more clear as we go on in the material why this is useful, but for now we'll just focus on how to do it.

#	ConvertingtoStrings-1.py	Output
1	myNumber = 5	<class 'int'>
2	print(type(myNumber))	<class 'str'>
3	myNumberAsString = str(myNumber)	
4	print(type(myNumberAsString))	
5		

Figure 2.2.16

In Figure 2.2.16, we create a variable `myNumber`, and give it the value 5. Then we check the type of `myNumber` on line 2, and sure enough, it is an integer, or `int`. Then, we create a new variable `myNumberAsString`, and we assign it to `str(myNumber)` on line 3. `str()` is a function (we'll talk about what functions are later) that converts whatever is inside the parentheses into a string, if possible. So, when we check the type of `myNumberAsString` on line 4, it's a string. Note that we could also use the `str()` function on a number by itself as well; for example, `fiveAsString = str(5)` would assign the string "5" to `fiveAsString`.

`str(variable)`
Takes as input some variable and returns a string representation of the variable's value.

The `str()` function will work to convert nearly any kind of variable to some kind of string. For example, the somewhat complex code shown in Figure 2.2.17 grabs today's date. The `str()` function used on line 3 returns a string representation of the date.

#	ConvertingtoStrings-2.py	Output
1	`from datetime import date`	2016-09-22
2	`myDate = date.today()`	
3	`myDateAsString = str(myDate)`	
4	`print(myDateAsString)`	
5		

Figure 2.2.17

In practice, you may not always need to do this; when you put a non-string value into Python's `print()` function, Python implicitly converts the value to a string. The code in Figure 2.2.18, for example, does the same thing as the code in Figure 2.2.17 without the string conversion. Python implicitly calls `str()` on `myDate` in order to convert it something it can print.

#	ConvertingtoStrings-3.py	Output
1	`from datetime import date`	2016-09-22
2	`myDate = date.today()`	
3	`print(myDate)`	
4		

Figure 2.2.18

However, there are instances where Python does not perform this conversion automatically. For example, we often want to provide labels behind what we're printing; this makes our output much easier to read. Instead of just printing the date, we might want to print it with the label "Today's date:". However, that requires `myDate` to be a string before it's printed in order to combine (or "concatenate") it with the label. Otherwise, we receive a `TypeError` as shown in Figure 2.2.19.

#	ConvertingtoStrings-4.py	Output
1	`from datetime import date`	Traceback (most recent call last):
2	`myDate = date.today()`	File "ConvertingtoStrings-4.py", line 3, in <module>
3	`print("Today's date: " + myDate)`	print("Today's date: " + myDate)
4		TypeError: Can't convert 'datetime.date' object to str implicitly
5		
6		
7		
8		

Figure 2.2.19

There are several ways we can do this, all shown in Figure 2.2.20. In lines 2 through 4, we store the date in `myDate`, then convert the date to the string `myDateAsString`, and then print the label "Today's date:" with `myDateAsString`. In lines 6 and 7, we skip storing myDate on its own and jump straight to passing

#	ConvertingtoStrings-5.py	Output
1	`from datetime import date`	Today's date: 2016-09-22
2	`myDate = date.today()`	Today's date: 2016-09-22
3	`myDateAsString = str(myDate)`	Today's date: 2016-09-22
4	`print("Today's date: " + myDateAsString)`	Today's date: 2016-09-22
5		Today's date: 2016-09-22
6	`myDateAsString = str(date.today())`	
7	`print("Today's date: " + myDateAsString)`	
8		
9	`myDate = date.today()`	
10	`print("Today's date: " + str(myDate))`	
11		
12	`print("Today's date: " + str(date.today()))`	
13		
14	`print("Today's date:", date.today())`	
15		

Figure 2.2.20

`date.today()` into `str()`. In lines 9 and 10, we skip storing `myDateAsString` explicitly, and instead do the string conversion inside the print statement with `str(myDate)`. In line 12, we skip storing anything at all and just print `str(date.today())` directly. In line 14, we skip the explicit conversion, and instead use a comma instead of a +, which tells the `print()` function to convert each comma-separated piece to a string automatically before trying to put them together. There are trade-offs here between clarity and efficiency or brevity. Learning which approach to take and when is one of the skills you'll develop as you learn computing. Generally, we'll use the approach shown on line 14.

`date.today()`
After importing date from datetime, returns a date object representing the current date.

Converting from Strings

Practically speaking, converting from strings to other values is just as easy. Just as there was a `str()` function for converting to a string, so also there are `int()`, `bool()`, and `float()` functions for converting to other data types.

`int(variable)`
Takes as input some variable and attempts to convert it to an integer, returning the integer if successful or raising a `ValueError` if unsuccessful.

`bool(variable)`
Takes as input some variable and attempts to convert it to a boolean, returning the boolean value if successful or raising a `ValueError` if unsuccessful.

`float(variable)`
Takes as input some variable and attempts to convert it to a float, returning the float if successful or raising a `ValueError` if unsuccessful.

#	ConvertingfromStrings-1.py
1	`myIntAsString = "5"`
2	`myInt = int(myIntAsString)`
3	`print(type(myInt))`
4	
5	`myFloatAsString = "5.1"`
6	`myFloat = float(myFloatAsString)`
7	`print(type(myFloat))`
8	
9	`myBooleanAsString = "True"`
10	`myBoolean = bool(myBooleanAsString)`
11	`print(type(myBoolean))`
12	

Output

```
<class 'int'>
<class 'float'>
<class 'bool'>
```

Figure 2.2.21

In Figure 2.2.21, we have three strings, each of which hold text that could be easily converted to a different data type: an `int` on line 1, a `float` on line 5, and a `bool` on line 9. Each one is converted using the corresponding function, and we see that the type is what we want it to be.

There is an important note here compared to the string conversion function. Whereas almost anything can be converted to a string, not everything can be converted to integers, booleans, and floats. For example, if a string holds an integer, we can still convert it to a float with `float()` because Python just assumes its decimal is .0, as seen in lines 1 through 3 of Figure 2.2.22. However, if a string holds a float, we cannot convert it to an int with `int()` because Python does not know what to do with the decimal, as seen in lines 5 through 7 of Figure 2.2.22. We receive a `ValueError` when trying to perform type conversions that are not permitted.

#	ConvertingfromStrings-2.py
1	`myIntAsString = "5"`
2	`myIntAsFloat = float(myIntAsString)`
3	`print(type(myIntAsFloat))`
4	
5	`myFloatAsString = "5.1"`
6	`myFloatAsInt = int(myFloatAsString)`
7	`print(type(myFloat))`
8	

Output

```
<class 'float'>
Traceback (most recent call last):
  File "ConvertingfromStrings-2.py",
line 5, in <module>
    myFloatAsInt = int(myFloatAsString)
ValueError: invalid literal for int()
with base 10: '5.1'
```

Figure 2.2.22

User Input

Now is a good time to talk about getting input from the user in console applications. Imagine you're writing a program that returns the square of a number that the user puts in. Without letting the user edit the code, how do you write this?

Here's how. Python has a function called `input()`. Whatever text you supply to input is given to the user as a prompt. The program then waits for the user to type something into the console and press enter; whatever the user types is stored in the

`input(prompt)`
Takes as input some string to use as a prompt for user input, and returns as a string the text the user enters.

#	UserInput-1.py	Output
1	`myUserInput = input("Enter an integer: ")`	`Enter an integer: 5`
2	`print(myUserInput * myUserInput)`	`Traceback (most recent call last):`
3		`File "UserInput-1", line 2, in <module>`
4		`print(myUserInput * myUserInput)`
5		`TypeError: can't multiply sequence by non-`
6		`int of type 'str'`
7		
8		

Figure 2.2.23

variable. In Figure 2.2.23, the user is asked to put in a number, and that number is then multiplied by itself and printed, right? Not exactly. As you can see, Python throws up an error. In the error, Python says that it can't multiply by a string. Why does it say that if the user entered an integer?

By default, Python (as of version 3) takes anything the user enters as a string. After all, Python doesn't know if the user is entering a string, an integer, a float, a boolean, or something else; all it knows is that the user's input is text, and text can always be interpreted as a string. Many times, numbers should be interpreted as strings; imagine a PIN or a phone number. There's little sense in interpreting those as 1,234 or 4,045,551,234.

#	UserInput-2.py	Output
1	`myInt = input("Enter an integer: ")`	`Enter an integer: 5`
2	`print(type(myInt))`	`<class 'str'>`
3		`Enter a float: 5.1`
4	`myFloat = input("Enter a float: ")`	`<class 'str'>`
5	`print(type(myFloat))`	`Enter a string: Hello, world!`
6		`<class 'str'>`
7	`myString = input("Enter a string: ")`	
8	`print(type(myString))`	
9		

Figure 2.2.24

You can see that behavior in Figure 2.2.24. We prompt the user for an integer, a float, and a string, and each time the user obeys the instruction; however, Python always interprets the user's input as a string regardless. In order to use an integer as an integer or a float as a float, we must convert them to integers or floats.

#	UserInput-3.py	Output
1	`myUserInput = input("Enter an integer: ")`	`Enter an integer: 5`
2	`myUserInputAsInt = int(myUserInput)`	`25`
3	`print(myUserInputAsInt * myUserInputAsInt)`	
4		

Figure 2.2.25

So what do we do? Simple: we convert the user's input to an integer. Before printing `myUserInput * myUserInput` on line 3 of Figure 2.2.25, we convert `myUserInput` to an integer on line 2. Then, we can be assured that we can successfully print its square… unless the user enters the wrong type of data. In that case, we would receive an error when we try to perform the conversion on line 2. Later in the material (in Chapter 3.5, on error handling), we'll discuss how to deal with this. Specifically, we can anticipate those errors and handle them within the code instead of letting the code crash.

8. Reserved Keywords in Python

In an earlier lesson, we noted that one of the rules around what names you can use for variables is that you can't use some of the reserved keywords that Python has. What this means is that there are certain words to which Python attaches special meaning and uses to understand what you're telling it to do. If you try to use these words to mean something different, Python gets confused.

This occurs in the real world in natural conversation, too. For example, if you named your dog "for" or "some," people would get confused when you would say things like "Can you take for for a walk?" or "Can you pour some some food?"

Python's Reserved Words

For now, you don't need to worry too much about what the reserved words do and why they're reserved. Right now, let's just list what these words are.

#	PythonsReservedWords.py	Output
1	`import keyword`	`['False', 'None', 'True',`
2	`print(keyword.kwlist)`	`'and', 'as', 'assert',`
3		`'break', 'class', 'continue',`
4		`'def', 'del', 'elif', 'else',`
5		`'except', 'finally', 'for',`
6		`'from', 'global', 'if',`
7		`'import', 'in', 'is',`
8		`'lambda', 'nonlocal', 'not',`
9		`'or', 'pass', 'raise',`
10		`'return', 'try', 'while',`
11		`'with', 'yield']`
12		

Figure 2.2.26

Fortunately, Python gives us a handy way of checking all its reserved keywords. Using the two lines of code in Figure 2.2.26, Python will print a list of them. Let's break these keywords into a few categories so you know when to expect to learn more about each one:

- Importing Libraries. `import`, `from`.
- Logical Operators. `and`, `is`, `not`, `or`, `False`, `True`, `None`.
- Control Structures. `as`, `break`, `continue`, `if`, `elif`, `else`, `for`, `in`, `while`, `pass`, `with`.
- Functions. `def`, `return`.
- Object-Oriented Programming Syntax. `class`.
- Error Handling. `except`, `finally`, `raise`, `try`.

The remaining words (`assert`, `del`, `global`, `lambda`, `nonlocal`, and `yield`) are outside the scope of our material. Note that if you see additional words like `print` and `exec`, it means you're working in an older version of Python. Based on these words, you can get a nice feeling for what we'll spend a lot of our time on: the remaining lessons in this unit are on logical and mathematical operators; Unit 3 covers control structures, functions, and error handling; and Unit 5 covers object-oriented programming. Units 2 and 3 are largely about learning to speak the computer's languages, which is why most of these keywords are covered in thess units.

Misusing Reserved Words

So what happens when we misuse reserved words? Let's try using a reserved word as a variable name in Figure 2.2.27. When we use the reserved word **"`pass`"** as a variable name, we're given a syntax error.

What about when we use it as a function name, as in Figure 2.2.28? Same thing—we're given a syntax error. This isn't how Python expects to see the reserved word `except` used. The same thing would happen if we used a reserved word as a

#	MisusingReservedWords-1.py	Output
1	`pass = 5`	`File "MisusingReservedWords-1.py",`
2		`line 1`
3		`    pass = 5`
4		`         ^`
5		`SyntaxError: invalid syntax`
6		

Figure 2.2.27

#	MisusingReservedWords-2.py	Output
1	`def except(num):`	`File "MisusingReservedWords-1.py", line 1`
2	`    return num`	`    def except(num):`
3		`              ^`
4		`SyntaxError: invalid syntax`
5		
6		

Figure 2.2.28

variable name. So, if you receive a syntax error, one thing to check is if you're using a reserved word improperly. Most Python development environments will highlight reserved words, making it easier for you to tell.

What about functions?

But wait! Haven't we seen other reserved words already? For example, wasn't `print` a reserved word? Let's find out!

#	WhatAboutFunctions.py	Output
1	`print = 5`	`Traceback (most recent call last):`
2	`print(print)`	`File "WhatAboutFunctions.py", line 2, in <module>`
3		`    print(print)`
4		`TypeError: 'int' object is not callable`
5		
6		
7		

Figure 2.2.29

On line 1 of Figure 2.2.29, Python lets us assign `print` a value just as if it's a variable! `print` = 5 is a fine line as far as Python is concerned. However, when we've done that, it can no longer use `print()` as a function name on line 2. This is an early introduction to the idea of scope. If we have a function name, we can override it with a variable name; if we have a variable name, we can override it with a function name. What this means is that when we encounter strange errors like the one shown here, one thing we might want to check is if we're using the same term as both a function and a variable. These aren't reserved in the same way as the reserved keywords; instead, they just have a certain temporary meaning, but that temporary meaning can be changed.

9. Dot Notation in Python

When we move on to object-oriented programming, we'll talk a lot about how a single variable can actually contain a lot of information. However, before we get to object-oriented programming, we're going to see several instances of using variables that have more data contained within them. So, let's look real quick at the syntax for that, called dot notation.

Variables with Lots of Data

We've talked in the past about another variable type, called a `date`. This goes beyond our primitive integers, floats, strings, and booleans. A `date`, however, is really three variables packaged into one with a special meaning: it's a year, a month, and a day. That's three integers. How can one variable hold *three* integers?

In Figure 2.2.30, we create a variable named `myDate`, and assign it to today's date. The date contains three integers: a year, a month, and a day. We access each of those by using dot notation: we take the variable name `(myDate)`, type a dot (.), then type the specific part of `myDate` we want to receive. In Figure 2.2.30, this gives

#	VariableswithLotsofData-1.py	Output
1	`from datetime import date`	2016
2	`myDate = date.today()`	9
3	`print(myDate.year)`	23
4	`print(myDate.month)`	
5	`print(myDate.day)`	
6		

Figure 2.2.30

us the year on line 3, the month on line 4, and the day on line 5. We're actually using dot notation in line 2 as well: `date` is a library (sort of a type of variable), `today()` is a method (sort of like a function) in that library, and the dot lets us access that part of the library.

What variables or functions are available is dependent on the particular variable you're working with. For example, it makes sense to ask the year, month, and day of a date, but it doesn't make sense to ask for the hour, minute, or second of a date. There is no answer to the question, "What's the current time on May 15th?" However, if we had a time object, it would make sense to ask for the hours, minutes, and second, as shown in Figure 2.2.31.

#	VariableswithLotsofData-2.py	Output
1	`import datetime`	9
2	`#Don't worry about why this line has`	15
3	`#the extra datetime for now`	31
4	`currentTime = datetime.datetime.now()`	
5	`print(currentTime.hour)`	
6	`print(currentTime.minute)`	
7	`print(currentTime.second)`	
8		

Figure 2.2.31

As noted above, for now, don't worry too much about this syntax. We won't get into it in depth until we get to Unit 4. For now, it's mostly just useful to be able to understand what you're seeing when you see it.

10. Variables with Turtles

Now let's talk about using variables with our turtles. Throughout our material, we're going to talk about programming with turtles in two contexts: first, with regard to just making fun things, and two, with regard to creating a more full-featured script that allows the user to dynamically control the turtle from the command line.

Simple Drawings with Variables

First, let's draw something simple with variables alone in SimpleDrawingswithVariables.py. We'll use two variables: length and turn angle. We'll draw five lines, each of the given length, turning by the given angle between each turn. To do this, we'll make use of two simple functions from the turtles library: `forward()` and `right()`.

`turtle.forward(distance)`
Takes as input distance as a float and moves the turtle forward the given distance.

In this code, we draw a small arc with five segments, turning by 30 degrees (`myTurnAngle`) between each 50-pixel (`myLength`) line. Play around with this a little bit. Notice how just by changing the variables once, you can radically change the picture that is generated. Try to draw a five-pointed star—it's possible to draw with five lines, rotating between a constant value each time.

`turtle.right(angle)`
Takes as input an angle as a float and rotates the turtle the given number of degrees.

User-Controlled Turtles: Getting Started

Let's get started developing a script that will let the user running the script control the turtle. For right now, our goal is simple: let the user input a length and a degree

of rotation, then execute them. This will draw one line in the direction and length of the user's choosing—a simple start, but an important one.

In this code in UserControlledTurtlesGettingStarted.py, the user inputs a turn angle and a length; these are then converted into integers, and Python then uses these numbers to draw a turtle's movement.

It's important to note the limitations here that we'll be addressing as we go forward. First, the user is forced to enter only one angle and one length, in that order. They are not able to select their commands: we'll add that functionality when we talk about conditionals. They are not able to input multiple commands: we'll add that functionality when we talk about loops. If they enter bad input, the program crashes: we'll fix that when we discuss exception handling.

Lines of Code: Efficiency and Readability

It is important to note that the number of lines of code is not a good measurement of code complexity or clarity. Packing everything into a tiny number of lines of code can make it hard to read, but extending it out into several lines can *also* make it hard to read. Keeping in mind that using as few lines of code as possible is not a good goal to have in software development, it is worth noting that figuring out ways to perform the same operations in fewer lines is a good way to test your coding knowledge. The code presented in LinesofCodeEfficiencyandReadability-1.py in six lines can be executed in two. Try to shorten the code to four lines, and then to two lines only.

We've supplied ways of shortening the code to both four lines and two lines in LinesofCodeEfficiencyandReadability-2lines.py. In one sense, the two-line code is harder to read: it packs a lot of information into repeatedly nested parentheses. In another sense, though, it's more efficient. We're not going to use the angle and length again later, so why bother saving them in variables? Take them, use them, and get rid of them.

In practice, the four-line version is probably the best way to do this, as shown in LinesofCodeEfficiencyandReadability-4lines.py. By storing the user's input in variables, someone reading our code is more able to recognize what we're saving and how we're using it. However, it doesn't really aid readability to save the converted version in its own variable; any reader could immediately recognize that the conversion is merely to make the types match up.

CHAPTER

2.3

Logical Operators

1. What Are Logical Operators?

To ask what logical operators are, we must first ask: what are **operators**? Operators are the simplest way to act on data, typically on simple data types like integers, strings, and floats. In the next couple chapters, we'll discuss two different types of operators: logical operators in this chapter, and mathematical operators in the next. But first, we'll discuss the types of operators in general; then, we'll get in more depth with logical operators and how they're used in your language.

Mathematical Operators

Mathematical operators are the more familiar but less important category of operators. Most languages have at least five mathematical operators: addition, subtraction, multiplication, division, and modulus. The first four of these are the same operations you've been discussing for years, since early arithmetic. The fifth, modulus, is the remainder function, the remainder when one integer is divided by another. Some languages might have dedicated operators for exponents or other common operations, while others might use functions for those tasks.

Most often, these mathematical operators will operate on numbers like integers and floats. Rarely, though, we'll find we can use operators on other data types as well. For example, in some programming languages (like Python), if you multiply a string by an integer, the string is duplicated a number of times. In other languages (like Java), adding two strings together might "concatenate" them, or put them together; but we'll cover that later.

Logical Operators

Logical operators are likely less familiar, but they play a bigger role in computing. Some of these are used for comparisons, like checking if two values are equal to one another or if one is greater than the other. Others are used for chaining together logical decisions like these, checking if multiple conditions are true or if one of many conditions is true.

Logical operators are incredibly powerful in computing. They are how we check to see if complex sets of conditions are met, and they are fundamental to how we control the execution of our programs. Every condition and loop that we cover in the next unit is governed in some way by logical operators.

We'll start with logical operators given their relatively high importance, then come back to mathematical operators in the next chapter.

2. Relational Operators

The ultimate goal of all logical operators is to assess whether certain statements are true or false. They only have two possible outcomes, true or false. A lot of complicated reasoning can go into that, though. The good thing is that we actually think in terms of logical operators every day, so we can pretty quickly jump to understanding these operators in our code.

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Describe different types of logical operators and use them to implement different relationships among data;
- Write lines of code to chain different decisions together using Python logical operators;
- Use truth tables to simulate the results of different Boolean operators.

Operators
Specific, simple functions that act on primitive data types, like integers and strings.

Mathematical Operators
Operators that perform mathematical functions, like adding numbers together or assigning values to variables.

Logical Operators
Operators that perform logical operations, such as comparing relative values, checking equality, checking set membership, or evaluating combinations of other logical operators.

Relational Operators
Operators that check the relationships between multiple variables, such as checking if they are equal or if one is greater than another.

Generally, there are two kinds of logical operators: **relational operators** and boolean operators. Relational operators check if things are true in the world or in our data. Boolean operators typically check the combination of multiple relational operators. Don't let the complex terminology here scare you: you'll see soon this reasoning is very natural, and in fact, you engage in it every day.

Relational operators check the relationships between multiple things. Let's first take some examples from the real world, and then look at some examples more core to programming.

Numeric Comparisons

Let's say you go to the kitchen for a snack. You open the refrigerator to get some grapes, but you find that there aren't any. So, you add grapes to your grocery list. That was a relational operator: you compared the number of grapes to some number in your head, in this case 0, and reasoned that it is true that the number of grapes in the refrigerator equals zero. That's the relational operator between a variable, the number of grapes, and some number you have in mind.

So instead of grapes, you decide you'll have a piece of fruit. There are apples and oranges, and there are more oranges than apples, so you decide to have an orange. That's another relational operator: you compared the number of apples to the number of oranges, decided it was false that there were more apples, and so you decided to have an orange. Notice in both these cases, logical operators are very closely entwined with a conditional statement: you compare something, generate a conclusion, and decide to take action based on the conclusion. The comparison is the logical operator, while the decision is the conditional.

Non-Numeric Equality Comparisons

Relational operators aren't just numeric comparisons, though. We can compare non-numeric equality as well. You grab your orange and sit down in front of the TV. You check the listings to see what's on, and you see your favorite show is on two different channels. You check what episodes are playing, and see both channels are playing the same episode, so you just choose which one to select randomly. That's a non-numeric equality comparison: you judged that it was true that the two episodes were "equal", so you chose randomly.

Set Operators

Right as you're about to sit down, though, you realize you didn't grab a drink. You go back to the fridge and see water, apple juice, and milk. You have a dairy allergy, though, so you select between water and apple juice. That's another relational operator: you checked to see if each beverage was in the set of things to which you're allergic, and chose one that gave the answer "false."

That's a rundown of the relational operators we'll use in a nutshell: we might do numeric comparisons, check for non-numeric equality, or we might check if something is a member of a certain type or list of things. Pay attention in your everyday life—you'll likely notice a huge amount of your reasoning uses these operators. Note that depending on the languages, some of these might not be implemented as operators; they might be implemented as methods or functions. That just means that performing these operations is a little more complex.

3. Relational Operators in Python

Let's see how we execute these operators in Python. We'll take some simple syntactical examples, then build toward a larger example of these principles in action.

Numeric Equality Comparisons

Our numeric comparison statements are very simple. If we want to compare whether two numbers are equal, we just write the numbers (or the variables holding the numbers) with **two** equals signs between them.

#	NumericEqualityComparisons-1.py	Output
1	`print(5 == 3)`	`False`
2	`print(5 == 5)`	`True`
3	`a = 5`	`True`
4	`b = 5`	`False`
5	`c = 3`	
6	`print(a == b)`	
7	`print(a == c)`	
8		

Figure 2.3.1

In Figure 2.3.1, we're printing the results of the numeric comparison so that we can see what's happening. The first line asks, "5 equals 3?", to which the computer replies, `False`! The second asks, "5 equals 5?", to which the computer replies, `True`! The last two statements repeat this with the numbers assigned to variables instead.

You might be wondering: why two equals signs instead of one? The middle three lines of Figure 2.3.1 show the difference. When we use only one equals sign, Python *assigns* the value to the variable. Line 3 doesn't check if a equals 5, it sets a equal to 5. If we tried to just print `a = 5`, we'd get an error. A single equal sign *sets* two things equal to each other (if possible), while two equal signs *checks* if they're *already* equal to each other.

Figure 2.3.2 shows what happens if you use the wrong operator here. When we use the double-equals on line 2, it correctly prints `True`. When we use the single-equals on line 3, we receive a `TypeError`.

#	NumericEqualityComparisons-2.py	Output
1	`a = 5`	`True`
2	`print(a == 5)`	`Traceback (most recent call last):`
3	`print(a = 5)`	`File "NumericEqualityComparisons-2.py", line 3, in <module>`
4		`    print(a = 5)`
5		`TypeError: 'a' is an invalid keyword argument for this function`
6		
7		
8		
9		

Figure 2.3.2

Numeric Value Comparisons

In addition to checking if numbers are equal, we can check if they're greater than or less than one another. We do this the way you might expect, by simply using the greater-than and less-than characters

In Figure 2.3.3, we see the same principles applied with greater-than and less-than. We can also use greater-than-or-equal-to and less-than-or-equal-to by adding an equals sign after the greater or less than sign, as seen on lines 4 and 12.

Non-Numeric Equality Comparisons

With numbers, it's easy to look at relative value. It makes sense to ask, "Is 3 greater than 5?" It doesn't make sense to ask, "Is blue greater than red?" With colors, names, and lots of other data types, there are no relative values. However, equality still exists in these areas. It doesn't make sense to ask, "Is blue greater than red?",

```
#  NumericValueComparisons.py
1  print(3 > 5)
2  print(3 < 5)
3  print(3 < 3)
4  print(3 <= 3)
5
6  a = 7
7  b = 9
8  c = 9
9  print(a < b)
10 print(b < a)
11 print(c > b)
12 print(c >= b)
13
```

```
Output
False
True
False
True
True
False
False
True
```

Figure 2.3.3

but it does make sense to ask, "Is blue the same as red?" In Python, we can often use the double-equals to mean equality among non-numeric data types. Most often, this will be strings.

```
#  NonNumericEqualityComparisons-1.py
1  a = "Hello, world"
2  b = "Hello, world"
3  c = "Hello, "
4  d = "world"
5  print(a == b)
6  print(a == c)
7  print(a == c + d)
8
```

```
Output
True
False
True
```

Figure 2.3.4

In Figure 2.3.4, we create four strings, `a`, `b`, `c`, and `d`. We give the same value to `a` and `b`, and sure enough, the == operator on line 5 shows that they're equal. It correctly notes that `a` is not equal to `c` on line 6, however. Line 7 is also the first look we have at string concatenation, or putting two strings together. `c + d` squeezes "Hello," and "world" into one string, which ends up equal to `a`.

Whether or not the == operator works this way with the data types you're using is dependent on a number of factors. Before relying on it, you should check to see if it operates as expected. The general point here is that for almost any data type, it does make sense to check for equality.

```
#  NonNumericEqualityComparisons-2.py
1  a = "Hello, world"
2  b = "Hello, world"
3  c = "Hello, "
4  d = "world"
5  print(a > b)
6  print(a >= b)
7  print(a >= d)
8
```

```
Output
False
True
False
```

Figure 2.3.5

It's worth noting that the greater-than and less-than operators do work for strings as well, and they operate based on sorting the strings alphabetically. Strings that are earlier alphabetically are treated as "less" than strings that are later alphabetically, as shown in line 5 through 7 of Figure 2.3.5. However, note that (a) this method treats capital and lower-case letters separately, meaning that capital Z would be sorted as "less" than lower-case a, and (b) I don't think I've ever had an occasion to use this method except when writing a string sorting method.

Set Operators

Finally, the other major Python operator we'll use very often is the `in` operator. The `in` operator checks to see if something is contained within a list of things. The `in` operator is somewhat unique to Python; most languages (like Java and C++) supply the same ability, but with a function or method rather than simply with an operator. Python just makes it a little easier.

#	SetOperators-1.py	Output
1	`myString = "Hello, world"`	`True`
2	`print("H" in myString)`	`True`
3	`print("lo, w" in myString)`	`False`
4	`print("oll" in myString)`	
5		

Figure 2.3.6

For example, we can use the `in` operator to see if there's a certain message contained within a longer message. In Figure 2.3.6, the first two print statements say `True` because "H" and "lo, w" are character sequences within `myString`. "oll" is `"False"` because it does not occur within `myString`. This is effectively like saying, "'oll' is in `myString`?" and having the computer reply, `False!`

We haven't talked much yet about lists, but `in` is also very useful in lists. Line 1 in Figure 2.3.7 defines a list; we'll talk more about lists in Unit 3, but for now, just know that this is a list with three strings: "and," "or," and "not." Line 2 asks, "'and' is in that list?", to which the computer replies, `True!` Line 3 asks, "'else' is in that list?", to which the computer replies, `False!`

#	SetOperators-2.py	Output
1	`pythonBooleanOperators = ["and", "or", "not"]`	`True`
2	`print("and" in pythonBooleanOperators)`	`False`
3	`print("else" in pythonBooleanOperators)`	
4		

Figure 2.3.7

Operators in Action

Let's look at a longer example of these operators in action. Imagine we're writing a program that validates a credit card scan. It would need to go through a number of different checks: it must check that the balance is sufficient to cover the total, it must check that the cardholder is the person trying to make the purchase, and it must check that the transaction is occurring at a trusted vendor. What would that look like in code?

#	OperatorsinAction.py
1	`...`
2	`if transactionAmount < accountBalance:`
3	`    if customerName == cardholderName:`
4	`        if vendor in trustedVendors:`
5	`            acceptTransaction()`
6	`        else:`
7	`            rejectTransaction("Untrusted Vendor")`
8	`    else:`
9	`        rejectTransaction("Invalid cardholder")`
10	`else:`
11	`    rejectTransaction("Insufficient funds")`
12	`...`

Figure 2.3.8

Figure 2.3.8 is the longest and most complex chunk of code you've seen so far, but don't let it intimidate you. Learning to interpret code that you can't fully read is a valuable skill. Note that we're using variables (`transactionAmount`, `accountBalance`, etc.) and functions (`acceptTransaction()`, `rejectTransaction()`) that aren't defined here; in reading this, we assume that this is just a chunk of code from a larger program, and values have been supplied for those variables elsewhere. This is why self-documenting code is so valuable: the variable names tell us what would have been contained here. We could have just as easily called these variables `foo` and `spud`, but that would make this code harder to read.

This code is built around some conditional statements which we'll cover in the next chapter. For now, though, just note that the first three lines each use a different relational operator: the first checks relative numeric value, the second checks non-numeric equality, and the third checks set membership.

4. Boolean Operators

We've now covered operators for checking a number of different relationships among data. Now, we want to chain these decisions together into higher levels of reasoning. We do that with **boolean operators**, operators that themselves work on combinations of `True` and `False` values, and generate either `True` or `False` themselves. Like our relational variables, these are very familiar in our everyday reasoning.

Boolean Operators
Operators like "`and`" and "`or`" that act on pairs of boolean (`true` or `false`) values, or that act on single boolean values, like "`not`".

And

And
An operator that acts on two boolean (`true` or `false`) values and evaluates to "`true`" if and only if *both* are `true`.

onList = grapes are on shopping list
inStock = grapes are in stock at store
onList **and** inStock

The first operator is `and`. An `and` statement is true if every part of the statement is true. For example, previously we used the example of going to get grapes out of the refrigerator, but finding none. So, you added them to your shopping list. Now imagine you're at the store. You buy grapes if they're on your list and if they're in stock. That's two relational operators: "grapes" must be in the set of items on your shopping list, and "grapes" must be in the set of items the grocery store has. You purchase grapes if they're *both* on your list *and* in stock at the store. If they're in stock but not on your list, or if they're on your list but not in stock, you don't buy them; only one part of the statement needs to be false to render the entire statement false.

Or

Or
An operator that acts on two boolean (`true` or `false`) values and evaluates to "`true`" if and only if *at least one* is `true`.

onList = grapes are on shopping list
onSale = grapes are on sale for a discount
onList **or** onSale

The second operator is `or`. An `or` statement is true if at least one part of the statement is true. We can tweak our previous example to make it an `or` statement: at the store you purchase grapes if they're on your shopping list *or* they're on sale for a discount. If they're on sale for a discount, you buy them whether they're on your list or not. If they're on your list, you buy them whether they're on sale for a discount or not. Only one part of the statement needs to be true to make the entire statement true. Note that `or` has a closely related but rarely used additional operator, **exclusive or** `xor`, that resolves to true if *exactly* one of the variables is true, but false if both are true. Not all languages supply the `xor` operator, but it can always be implemented using a combination of `and`, `or`, and `not`.

Xor
An operator that acts on two boolean (`true` or `false`) values and evaluates to "`true`" if and only if *exactly one* is `true`.

Not

rotten = grapes are rotten
not rotten

The third operator is `not`. `not` simply alters the value of a part of the statement. For example, even if grapes were on your list, even if they were in stock, and even if they were on sale for a discount, you would not purchase them if they were rotten. The question, "They are rotten?", however, would return `True`, which means that if it were used in an `and` or `or` statement, being rotten would contribute to the trueness of the statement. `not` allows us to switch the value to make sense in the logical context of our statement. So, we would say that if the grapes are not rotten, you would purchase them.

Not
An operator that acts on one boolean (`true` or `false`) value and evaluates to the opposite value (`false` becomes `true`, `true` becomes `false`).

Combining Boolean Operators

onList = grapes are on shopping list
onSale = grapes are on sale for a discount
rotten = grapes are rotten
inStock = grapes are in stock at store

(onList or onSale) and inStock and not rotten

The power of these operators really arises when we put them together in longer, more complex chains. We've discussed four elements of your decision to buy grapes: are they on your list?; are they in stock?; are they on sale for a discount?; are they rotten? How do we combine these into one larger statement?

We can use parentheses to group together parts of our logical statement into a broader conclusion. We can try to parse the statement in the margin from either the outside in or the inside out. Let's start with the inside out. First, your initial decision will be to purchase the grapes if they are either on your list `(onList)` or on sale `(onSale)`; if neither is true, then the innermost statement is false. We notice that as we go outward, the boolean operators are both `and`, so if the innermost statement is false, the entire statement must be false.

If the innermost statement is true, however, then we proceed outward. If the grapes are not `inStock`, the status of that innermost statement is irrelevant; they can't be purchased. If the innermost statement is true and they are in stock, it still does not matter if they are rotten; if `not rotten` is false, it negates the rest of the statement, too.

This kind of logic forms an enormous portion of the philosophy of computing, artificial intelligence, and a field called predicate calculus.

5. Boolean Operators in Python

Python makes the usage of these boolean operators simple: while other languages (like Java and JavaScript) use symbols like &, |, ~, or ! to represent them, Python simply uses the words `and`, `or`, and `not`. This makes the language quite readable.

In order to explore these operators, let's define some boolean variables based on our previous example. Just as variables can take numeric values or strings of characters, so also they can take `True` or `False` as values. This is why `True` and `False` are reserved words in Python, meaning that these terms can't be used as variable names or function names. You will see the four lines in Figure 2.3.9 at the beginning of every block of code below.

#	BooleanOperatorsinPython.py	Output
1	`onList = True`	
2	`inStock = True`	
3	`onSale = False`	
4	`rotten = False`	
5		

Figure 2.3.9

And

Our original statement was to check if grapes were both on your list and in stock at the store. How do we check this?

In Figure 2.3.10 on line 6, we print the result of `onList and inStock`. If both are true, the statement is true. If either is false, the statement is false. Note that while

#	And-1.py	Output
1	`onList = True`	`True`
2	`inStock = True`	
3	`onSale = False`	
4	`rotten = False`	
5		
6	`print(onList and inStock)`	

Figure 2.3.10

some languages can only perform "and" and "or" on two operators at a time, Python can process multiple, moving from left to right; it evaluates the first pair, replaces the pair with the value, evaluates the value with the next variable, and so on.

Figure 2.3.11 shows evaluating three variables using a pair of `and` operators. The logic here is effectively, "Buy grapes if they're on my list, in stock, and on sale for a discount." Here, Python first evaluates `onList and inStock`, and receives the result `True`. It then evaluates `True and onSale`," replacing the first part of the original statement with `True`. `onSale` is `False`, though, which negates the statement. Processing this way, a single `False` in a series of `and` operations will always negate the entire series, causing it to be `False`.

#	And-2.py	Output
1	`onList = True`	`False`
2	`inStock = True`	
3	`onSale = False`	
4	`rotten = False`	
5		
6	`print(onList and inStock and onSale)`	
7		

Figure 2.3.11

Or

Our second example was to check if grapes were either on our list or on sale for a discount. How do we check that?

In Figure 2.3.12, we just print the result of `onList or onSale`. If either is `True`, the result will be `True`. `or` works the opposite way as `and`: if we chain together multiple `or` operators, one single `True` will render the entire statement `True`.

#	Or-1.py	Output
1	`onList = True`	`True`
2	`inStock = True`	
3	`onSale = False`	
4	`rotten = False`	
5		
6	`print(onList or onSale)`	
7		

Figure 2.3.12

Line 6 in Figure 2.3.13 represents the strange reasoning that you will buy grapes if they're on your list, on sale for a discount, or rotten. I don't know why you're buying grapes specifically if they're rotten even if they're neither on sale nor on your shopping list, but that's what this line says you're doing!

#	Or-2.py	Output
1	`onList = True`	`True`
2	`inStock = True`	
3	`onSale = False`	
4	`rotten = False`	
5		
6	`print(onList or onSale or rotten)`	
7		

Figure 2.3.13

Not

Our third example was to check if grapes were rotten. How do we check that?

#	Not-1.py	Output
1	`onList = True`	`True`
2	`inStock = True`	
3	`onSale = False`	
4	`rotten = False`	
5		
6	`print(not rotten)`	
7		

Figure 2.3.14

To accomplish this, we just print the results of `not rotten` in Figure 2.3.14. `rotten` is `False`, so `not rotten` is `True`. We can also use `not` on the results of other logical statements as well.

In Figure 2.3.15, we use our `not` operator in two ways. In line 6, `not` specifically applies to `onSale`, changing the value of the first half from `False` to `True`. Then, line 6 applies the `and` operator with `rotten`, which is `False`. This equates to `True and False`, which is `False`. In line 7, however, the `not` is applied to the result of `onSale and rotten`. `onSale and rotten` are both `False`, so `onSale and rotten` is also `False`. The `not` operator is outside the parentheses, meaning it applies to the result of the inner operation, turning the `False` to `True`.

Don't worry if you're a little confused. Working with boolean operators is very much a skill to be learned, not a set of facts to memorize. When in doubt, try to think through the reasoning as if it was a decision you were making in your everyday life.

#	Not-2.py	Output
1	`onList = True`	`False`
2	`inStock = True`	`True`
3	`onSale = False`	
4	`rotten = False`	
5		
6	`print(not onSale and rotten)`	
7	`print(not(onSale and rotten))`	
8		

Figure 2.3.15

Combining Operators

So, let's use these operators and combine them into our original statement, that you should buy grapes if they are not rotten, in stock, and either on your shopping list or on sale.

Line 6 in Figure 2.3.16 chains together all these operations. Working from the outside-in, we see that if the grapes are rotten, it will negate anything else because this is half of an `and` statement. If that does not negate the statement, then `inStock` being `False` would negate the rest because it is also half of an `and` operation. If

#	CombiningOperators.py	Output
1	`onList = True`	`True`
2	`inStock = True`	
3	`onSale = False`	
4	`rotten = False`	
5		
6	`print(((onList or onSale) and inStock) and not rotten)`	
7		

Figure 2.3.16

both `not rotten` and `inStock` are `True`, then the entire statement is `True` if either half of the inner statement is `True`. In this case, `not rotten`, `inStock`, and `onList` are all `True`, and so you purchase the grapes.

Note that we're using parentheses here to organize our operators. This is not totally necessary. If we removed the parentheses, we would get the same result. However, including the parentheses makes the code more readable and predictable. I would recommend using parentheses to organize the order in which the computer interprets your boolean operators whenever you are using a mixture of them, just in case.

Simplifying Conditionals

Previously, we looked at the complex code in Figure 2.3.17 for verifying transaction information. We'll cover conditionals later, but for now, let's look real quick at how boolean operators let us simplify this code.

```
# OperatorsinAction.py
1  ...
2  if transactionAmount < accountBalance:
3      if customerName == cardholderName:
4          if vendor in trustedVendors:
5              acceptTransaction()
6          else:
7              rejectTransaction("Untrusted Vendor")
8      else:
9          rejectTransaction("Invalid cardholder")
10 else:
11     rejectTransaction("Insufficient funds")
12 ...
13
```

Figure 2.3.17

In Figure 2.3.18, we see that rather than having each condition in its own statement and line, we can use boolean operators to chain them together. Here, we check all the conditions together on one line. However, there is a trade-off: first, line 2 is harder to read than the sequence of conditionals in lines 2 through 4 of Figure 2.3.17. Second, we can no longer give feedback specifically based on which condition was false. This is a good example of the trade-offs in programming between simplicity and readability. There is no right answer; the best choice will depend on the program you're writing. If it is important to have feedback on why the transaction was rejected, the first way is necessary. If this is not needed, I might argue the second is just as good.

```
# SimplifyingConditionals-2.py
1  ...
2  if (transactionAmount < accountBalance) and (customerName ==
     cardholderName) and (vendor in trustedVendors):
3      acceptTransaction()
4  else:
5      rejectTransaction("Transaction invalid")
6  ...
7
```

Figure 2.3.18

6. Truth Tables

Dealing with boolean operators is one of the foundational parts of computing. This spans across languages, domains, operating systems, and more: these are parts of the very core of computer science.

Truth Tables
Tables that map out the results of a statement in boolean logic (that is, using boolean operators) depending on the values of the individual variables.

To explore these further, we use something called **truth tables**. Truth tables are visualizations that help out with simulating and understanding the results of different boolean operators. Let's start with a couple of simple ones, then move on to some more complex ones.

And, Or, and Not

To visualize the simplest truth tables, let's start with the three basic Boolean operators, `and`, `or`, and `not`.

Value of A	Value of B	Value of (A and B)
True	True	True
True	False	False
False	True	False
False	False	False

Figure 2.3.19

In a truth table, we have a number of different columns and rows. A single truth table corresponds to a single boolean statement. We then have a column for each variable (the two on the left of Figure 2.3.19, and column for the result (the one on the right). In Figure 2.3.19, we have the truth table for the overall statement `a and b`: columns for `a` and `b`, and a column for the result. Each row of the table assigns the variables different values, until all possible combinations of values have been listed; here, we have two variables, so four rows are needed to capture all the combinations. The number of rows will always be $2^{\text{number of variables}}$—four rows for two variables, eight rows for three variables, sixteen rows for four variables, etc.

The truth table shows what happens for each combination of variables. Whenever either of them is `False`, the result is `False`. When both are `True`, the result is `True`.

Value of A	Value of B	Value of (A or B)
True	True	True
True	False	True
False	True	True
False	False	False

Figure 2.3.20

In Figure 2.3.20, `or` is the opposite result: whenever *either* variable is `True`, the result is `True`. Whenever *both* variables are `False`, the result is `False`.

Value of B	Value of (not B)
True	False
False	True
True	False
False	True

Figure 2.3.21

In Figure 2.3.21, the `not` operator gives us the simplest truth table of all: only one variable, and when it's `True`, the result is `False`; when it's `False`, the result is `True`. If this sounds strange, then think of it in terms of a real example: if it's true that the grapes *are* rotten, then it's false that the grapes are *not* rotten.

More Complex Truth Tables

The power of these truth tables shows up when we evaluate more complex statements. For example, in Figure 2.3.22 we have the statement `a or (b and not c)`. That's a complicated statement, but the truth table allows us to organize the different possibilities and go through them one by one. The table has eight rows because there are eight possible combinations of values for the three variables (2^3 is 8).

Value of A	Value of B	Value of C	Value of A or (B and not C)
True	True	True	True
True	True	False	True
True	False	True	True
True	False	False	True
False	True	True	False
False	True	False	True
False	False	True	False
False	False	False	False

Figure 2.3.22

In the first row, all the variables are `True`. So, we can start with the innermost parenthesis: `b and not c`. This is an `and` statement, so both `b` and `not c` must be `True` for the statement to be `True`. `b` is `True`, but `not c` is `False` (because `c` itself is `True`). So, this inner parenthetical is `False`. The statement now reads `a or false`; `a` is `True`, and an `or` requires only one part to be `True`. Therefore, the first row is `True`.

We might also notice here that the results of the inner parenthetical don't matter as long as `a` is `True`; since `a` is half of an `or` statement, we can go ahead and say any row where `a` is `True` resolves to `True`.

That just leaves us with the rows where `a` is `False`. We know these rows will be `False` if the parenthetical is `False`, which we know it will be if `b` and `c` are both true from our work on the first row. So, we can go ahead and mark `False` on row 5. Now, we would need only to resolve the last three rows.

Properties of Boolean Operators

Let's look at some of the properties of boolean operators to explore truth tables a bit more. In looking at these examples, look both at how the truth tables are being resolved and at the ultimate conclusions they carry.

First, individual boolean operators are commutative, as shown in Figure 2.3.23. That means that it doesn't matter the order in which the variables are presented if only one type of operator is used: `a and b` presents the same result as `b and a`. This applies also to multiple of the same operator: `a or b or c` produces the same result as `c or b or a`. Note that this does *not* apply to a mixture of operators: `a and b or c` is not guaranteed to produce the same output as `c or b and a`. That's why it's important to use parentheses to tell the computer in what order to read more complex logical statements.

Commutative Property

Value of A	Value of B	Value of (A and B)	Value of (B and A)
True	True	True	True
True	False	False	False
False	True	False	False
False	False	False	False

Figure 2.3.23

Distributive Property

Value of A	Value of B	Value of C	Value of A and (B or C)	Value of (A and B) or (A and C)
True	True	True	True	True
True	True	False	True	True
True	False	True	True	True
True	False	False	False	False
False	True	True	False	False
False	True	False	False	False
False	False	True	False	False
False	False	False	False	False

Figure 2.3.24

Second, boolean operators are distributive, as shown in Figure 2.3.24. This is hard to describe in abstract terms, so let's describe it concretely: here, we're comparing `a and (b or c)` with `(a and b) or (a and c)`. We've "distributed" the `a and` to the individual operators inside the parentheses, and the result of the statement is the same. This applies to any other combinations of `and` and `or` operators as well—any `not` operator would stay with the variable to which it was attached.

Finally, an interesting application of truth tables is to something called "de Morgan's Law," shown in Figure 2.3.25. The law says that the negation `(not)` of an `and` operation between two variables is the same as an `or` operation between each variable's negation on its own. That sounds complicated, so let's put it in more practical terms with our variables `inStock` and `onList` from our grapes example. If it's true that grapes are *not both* in stock and on our list, then it's true that *either* grapes are *not* in stock *or* they are *not* on our list. One of the two parts of the statement must be false if the entire statement is false. So, we can transform our `not and` statement into `not or not`: they're *not* both in stock *and* on our list becomes they're either *not* in stock *or* they're *not* on our list.

de Morgan's Law applies in the other direction as well. The negation of an `or` operation between two variables is the same as an `and` operation between each variable's negation on its own. Let's put this in terms of our `onList` and `onSale` operations. If they're *not* either on our list *or* on sale, then they're *not* on sale *and not* on our list.

De Morgan's Law (And → Or)

Value of A	Value of B	Value of not (A and B)	Value of not A or not B
True	True	False	False
True	False	True	True
False	True	True	True
False	False	True	True

Figure 2.3.25

De Morgan's Law (Or → And)

Value of A	Value of B	Value of not (A or B)	Value of not A and not B
True	True	False	False
True	False	False	False
False	True	False	False
False	False	True	True

Figure 2.3.26

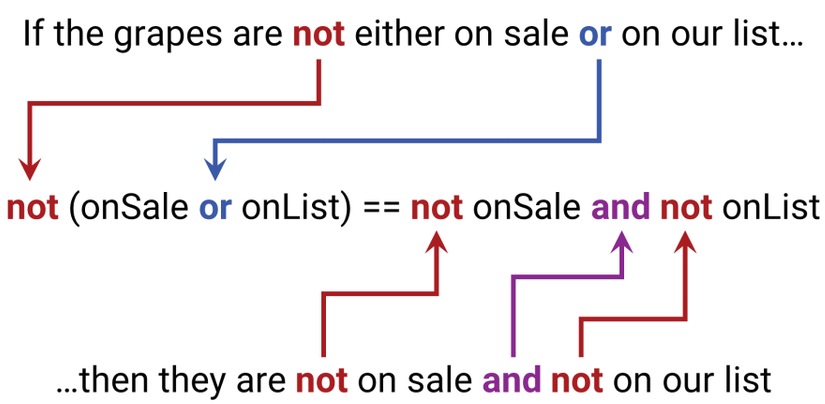


Figure 2.3.27

If this is still confusing, the diagram in Figure 2.3.27 may help. Note how the "`and`," "`or`," and "`not`" in our natural language description of this situation map to the "`and`," "`or`," and "`not`" in our logical statement. The two statements are expressing the same content. And if this is still confusing, don't fret: this is a skill to be practiced, not a formula to be memorized. As you do more work with boolean operators, this will start to feel far more natural.

CHAPTER 2.4

Mathematical Operators

1. What Are Mathematical Operators?

Previously, we discussed logical operators. Logical operators took as input one or two boolean values and produced a boolean as output. Mathematical operators work in much the same way: they take one or more numbers as input, and produce a number as output. Some mathematical operators are also able to work on input besides numbers; we'll talk about that later as well.

The Assignment Operator

The first mathematical operator—if we can call it mathematical, it's a bit of a special case—is one we've actually already seen before. It's the **assignment operator**. The assignment operator is the operator we use to give, or assign, a value to a variable. It populates the variable with a value in the first place.

The reason I include the assignment operator as a mathematical operator (even though it's also used to assign other values to variables) is that, besides populating variables with their initial values, it is very often used in the context of mathematical operations. When we perform an addition, subtraction, or other mathematical operation we need to store the results somewhere. Typically, we do so by assigning the outcome to a variable. Often, we'll actually assign the outcome to one of the variables used in the calculation, which will feel odd, but you'll see the usefulness soon!

Mathematical Operators

Aside from the assignment operator, the other **mathematical operators** are borrowed directly from arithmetic. As I've said before, if you don't enjoy math, don't worry; as we get started in programming, you don't need any more mathematical background than basic arithmetic. Almost every programming language supplies at least five operators: addition, subtraction, multiplication, division, and modulus. The first four of these should be straightforward.

The fifth operator, **modulus**, is the remainder function. When you divide two integers and one does not go evenly into the other, you have a remainder. For example, 3 goes into 7 twice, with 1 left over. So, 7 modulus 3 would give a result of 1: it returns only the remainder of the division, not the result itself. This is actually extremely useful. For example, how do you check if a number is even? The simplest way is to check if the number modulus 2 is 0; if there is no remainder when the number is divided by 2, then it is even. For a more authentic example, imagine you are programming a website that laid out images in rows of 10. To know when to start a new row, you would need to know when the number of pictures in the current row is 10. You could do this by using the modulus operator to check if the total number of pictures shown so far is a multiple of 10. If so, you would start a new row.

Additional Operators

Some languages supply additional operators as well. For example, some languages, like Python, have dedicated operators for using exponents. Others, like Java, require separate functions to accomplish the same goal. Functions and operators can be

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Use different mathematical operators in programming and describe real-world scenarios in the context of incrementing and self-assignment;
- Write lines of code that leverage the seven Python operators;
- Use self-assignment operation and the method of incrementing the values of the variable;
- Write turtle programs that leverage mathematical operators.

Assignment Operator
An operator that takes the output of an expression and assigns it to a variable.

Mathematical Operators
Operators that mimic basic mathematical functions, like addition and multiplication.

Modulus
The remainder function, returns the remainder of one number divided by another.

len()
A function that takes as input a variable with a length, such as a string of characters or a list of items, and returns its length.

very similar in that both take one or more variables or values as input and return some output. Operators are distinct in that they're usually represented by a dedicated character or keyword, like a plus sign for addition or `not` for a negation, rather than by a function name like `print()` or `len()`.

2. Mathematical Operators in Python

You've seen the assignment operator in the past, and you continue to see it in nearly every segment of code we look at. With the assignment operator, we place a variable on the left, and something on the right that resolves to a value. It could just be a value on its own, or it could be an operator, function, or expression that results in a value. We'll continue to see this as we go on.

So, here, let's focus first on the four simple mathematical operators, then on the modulus operator, and then on a couple of special operators that Python supplies.

The Basic Mathematical Operators

The basic mathematical operators are addition, subtraction, multiplication, and division. They work just about as you might expect.

#	TheBasicMathematicalOperators-1.py	Output
1	`#Create three and set it equal to 3`	12
2	`three = 3`	6
3	`#Create nine and set it equal to 9`	27
4	`nine = 9`	3.0
5		
6	`#Print the sum of 9 and 3`	
7	`print(nine + three)`	
8	`#Print the difference between 9 and 3`	
9	`print(nine - three)`	
10	`#Print the product of 9 and 3`	
11	`print(nine * three)`	
12	`#Print the quotient of 9 and 3`	
13	`print(nine / three)`	
14		

Figure 2.4.1

Note in Figure 2.4.1 we're doing something a little clever: we're making our variable names the spelled-out version of the values they hold to make it easier to see the results are what we expect them to be. Note that the value of `three` doesn't *have* to be 3, but we're doing that to make our code a little easier to follow.

We see about what we expect: 9 + 3 is 12, 9 – 3 is 6, 9 × 3 is 27, and 9 ÷ 3 is 3, with * and / for multiplication and division, respectively. Each operation does just what we'd expect. Note, however, there is something interesting that happens here: when we add, subtract, and multiply two integers, our result is an integer as well; that's true in math itself as well as in programming. However, note that when we divide two integers, we get a float (3.0), even if one number went into the other one evenly. That's because unlike addition, subtraction, and multiplication, division does not guarantee an integer result even if it applies to integers. So, division always gives a float, with .0 as the decimal if the numbers did in fact divide evenly.

Note that this sets Python apart from other languages. Most languages, when dividing integers, round down automatically. They can be thought of as, "How many *complete* times does this number go into that one?" Python, however, implements division more in the way to which we're accustomed: a decimal value is returned just in case one number does not go evenly into the other.

Typically, rather than printing the results of these operators outright, we'll save them to some variable. In Figure 2.4.2, we see the assignment operator used that way: it is used to assign the sum of nine and three to the variable name `mySum` on line 7,

```
#  TheBasicMathematicalOperators-2.py
1  #Create three and set it equal to 3
2  three = 3
3  #Create nine and set it equal to 9
4  nine = 9
5
6  #Create mySum with the sum of nine and three
7  mySum = nine + three
8  #Create myDifference with the difference between nine and three
9  myDifference = nine - three
10 #Create myProduct with the product of nine and three
11 myProduct = nine * three
12 #Create myQuotient with the quotient of nine and three
13 myQuotient = nine / three
14
```

Figure 2.4.2

the difference to `myDifference`, and so on. Note this code has no output because it has no print statements; it still, nonetheless, runs as expected.

Modulus

While those four operators are relatively straightforward from your days in arithmetic, the fifth common operator, modulus, is a little stranger. The modulus operator executes the remainder function: it divides two integers, but instead of returning the quotient, it returns the remainder of the integer division. In Python, modulus is represented by the percent (%) character. Let's see an example.

There's a lot of print statements in Figure 2.4.3, but don't let them startle you—we're using so many just to show the pattern. The first batch (lines 1 through 4) shows the remainders when dividing numbers by 2, while the second batch (lines 6 through 11) shows the remainders when dividing numbers by 3. Starting from line 1: 2 goes into 3 once with 1 left over, so 3 % 2 is 1. 2 goes into 4 twice with 0 left over, so 4 % 2 is 0. 2 goes into 5 once with 1 left over, so 5 % 2 is 1. When dividing by 2, we would alternate between 0 and 1. When dividing by 3 in lines 6 through 11, the pattern shifts a bit: we alternate between 0, 1, and 2.

```
#  Modulus.py                                          Output
1  print(3 % 2)    #Print the remainder of 3 / 2       1
2  print(4 % 2)    #Print the remainder of 4 / 2       0
3  print(5 % 2)    #Print the remainder of 5 / 2       1
4  print(6 % 2)    #Print the remainder of 6 / 2       0
5  print()                                             
6  print(4 % 3)    #Print the remainder of 4 / 3       
7  print(5 % 3)    #Print the remainder of 5 / 3       1
8  print(6 % 3)    #Print the remainder of 6 / 3       2
9  print(7 % 3)    #Print the remainder of 7 / 3       0
10 print(8 % 3)    #Print the remainder of 8 / 3       1
11 print(9 % 3)    #Print the remainder of 9 / 3       2
12                                                     0
13
```

Figure 2.4.3

The modulus operator is extremely useful in many areas of programming. We described two earlier: creating rows of length 10 and creating alternating patterns. Modulus is also used extensively in information security and cryptography among other areas.

Other Operators

Floor Division
Division that rounds the result down to the nearest integer.

Python supplies a couple of other operators as well. The first is a little strange: it's called **floor division**. Floor division mimics the way integer division works in many other languages; it effectively divides two integers and rounds the result down to the nearest integer. It's represented in Python by a double slash (//).

Note the differences in division and floor division in Figure 2.4.4. When dividing nine by three, division on its own results in a float in line 5. As we noted earlier, in Python, division automatically results in a float even if you're dividing two integers. Floor division, on the other hand, always returns an integer. It returns the number of times one number goes into another one, ignoring how much is left over. So, using the floor division operator on 9 over 3 results in 3 in line 8.

#	OtherOperators-1.py	Output
1	`three = 3`	3.0
2	`nine = 9`	3
3		
4	`#Print the quotient of nine over three`	
5	`print(nine / three)`	
6	`#Print the quotient of nine over three,`	
7	`#rounded down to an integer`	
8	`print(nine // three)`	
9		

Figure 2.4.4

When we use the floor division operator with numbers that are not evenly divisible by each other, we see the difference more clearly. The result of the division operator in lines 6 and 12 of Figure 2.4.5 includes the decimal; the floor division operator in lines 9 and 15 rounds down. So, with division in line 6, 5 / 2 is 2.5, whereas with floor division in line 9, 5 // 2 is 2. 2 only goes into 5 two complete times, so 5 // 2 is 2. Note that when using negative numbers, floor division still rounds *down*, not toward zero. So, –5 // 2 in line 15 is –3, not –2. I can't honestly say I've ever had a reason to use floor division or modulus with negative numbers, though.

#	OtherOperators-2.py	Output
1	`two = 2`	2.5
2	`five = 5`	2
3	`negativeFive = -5`	
4		-2.5
5	`#Print the quotient of five over two`	-3
6	`print(five / two)`	
7	`#Print the quotient of five over two,`	
8	`#rounded down to an integer`	
9	`print(five // two)`	
10	`print()`	
11	`#Print the quotient of negativeFive and two`	
12	`print(negativeFive / two)`	
13	`#Print the quotient of negativeFive and two,`	
14	`#rounded down to an integer`	
15	`print(negativeFive // two)`	
16		

Figure 2.4.5

The other somewhat unique mathematical operator Python supplies is the exponentiation operator, for using exponents. Most languages I've seen use a function to supply this capability, but Python reserves the double-asterisk (**) for exponentiation as shown in Figure 2.4.6. Line 6 shows 5^2. Line 8 shows 2^5. Line 10 shows -5^2. Line 12 shows 2^{-5}.

#	OtherOperators-3.py	Output
1	`two = 2`	25
2	`five = 5`	32
3	`negativeFive = -5`	25
4		0.03125
5	`#Prints the result of five to the two power`	
6	`print(five ** two)`	
7	`#Prints the result of two to the five power`	
8	`print(two ** five)`	
9	`#Prints the result of negativeFive to the two power`	
10	`print(negativeFive ** two)`	
11	`#Prints the result of two to the negativeFive power`	
12	`print(two ** negativeFive)`	
13		

Figure 2.4.6

3. Self-Assignment and Incrementing

In many ways, variables and mathematical operators work the same way in programming as they do in algebra. However, one of the most common usages of variables in programming isn't common in most mathematics: **self-assignment**.

Self-Assignment
A common programming pattern where a variable is assigned to the output of an expression that included the variable itself.

Self-Assignment

Self-assignment means assigning a variable to a value that is in part determined by the current value of the variable itself. In other words, instead of assigning the variable to the output of an operator on other values (like addition) or a function (like the length of some string), the variable is set to a *new* value that is based on the *current* value of the variable.

This is hard to describe in abstract terms, so let's take an example. Imagine you receive a paycheck that is automatically deposited into your bank. What is your new balance? Well, your new balance is your old balance plus your paycheck's amount. The calculation of the new balance is based on the value of your current balance: newBalance = oldBalance + paycheck. But really, you only have one bank account balance: it just changes over time. So, it makes sense to only have one variable to represent your balance, and just update that variable over time: balance = balance + paycheck. The details of this situation, however, mean that *updating* that balance involves using the *current* balance in the calculation. This is self-assignment: the variable *receiving* the value on the left is *used* in the calculation on the right.

Incrementing

Increment
Repeatedly adding a constant, typically one, to a variable.

The most common place this principle is likely used is with **incrementing**. Incrementing refers to having a variable whose job is to count something. Imagine, for example, you want to count the number of instances of a certain word in a document. So, you create a variable to hold the number of times you've found it so far, perhaps calling it `wordCount`. Every time you encounter the word, you add one to `wordCount`. That means every time you encounter the word, you reassign the variable `wordCount` to the value that results from `wordCount` plus one. In other words, you say, "My new value for `wordCount` should be whatever my old value was, plus one."

In practice, this can feel very strange at first. In most languages, you would write this as `wordCount = wordCount + 1`. It feels weird to have `wordCount` on both sides of the assignment. However, this makes more sense when you consider how the computer executes this statement. Let's imagine our current value of `wordCount` is 5, and we've just encountered another instance of the word, so we run `wordCount = wordCount + 1` (or your language's equivalent). The computer

starts by interpreting the right side of that assignment statement: the value of `wordCount` is 5, the value of 1 is 1, and 5 + 1 = 6. So, the computer effectively replaces `wordCount + 1` with 6. Now, it runs the new line, `wordCount = 6`, which changes the value of `wordCount` to 6. The line is done, and `wordCount's` value is now 6.

4. Self-Assignment and Incrementing in Python

Let's see how this works in Python. Self-assignment is common enough that we can demonstrate it pretty straightforwardly. For incrementing, we'll have to briefly preview a topic that we'll get to more fully in Chapter 3.3, loops.

Self-Assignment

We saw in Figure 2.4.2 an example of assignment with mathematical operators in Python: we would have a mathematical expression on the right, and a variable on the left to which to assign the results of the expression (e.g. x = 5 – 2). Self-assignment isn't actually any different; the only thing that makes self-assignment unique is that a single variable appears on both sides of the statement. To the computer, it actually isn't different at all because it interprets the right side of the statement before looking at the variable to which to assign its result.

In Figure 2.4.7, we create a variable `myNum` and assign it the value 5 on line 1. We print it on line 3 to ensure the value really is 5. Then on line 5, we run `myNum = myNum + 1`, or "Set the value of `myNum` to its current value plus 1." When we print it again on line 7, we see that sure enough, its value has increased by 1. Running the line again on line 9 increases it by 1 again. Replacing 1 with 5 on line 13 increases `myNum` by 5.

#	SelfAssignment-1.py	Output
1	`myNum = 5`	5
2	`#Print myNum's current value`	6
3	`print(myNum)`	7
4	`#Add 1 to myNum, and give the result to myNum`	12
5	`myNum = myNum + 1`	
6	`#Print myNum's current value`	
7	`print(myNum)`	
8	`#Add 1 to myNum, and give the result to myNum`	
9	`myNum = myNum + 1`	
10	`#Print myNum's current value`	
11	`print(myNum)`	
12	`#Add 5 to myNum, and give the result to myNum`	
13	`myNum = myNum + 5`	
14	`#Print myNum's current value`	
15	`print(myNum)`	
16		

Figure 2.4.7

Self-assignment works the same way with any of the mathematical operators, or with more complex chain of operators. In Figure 2.4.8, we show it working with all seven mathematical operators, concluding with division to save the conversion to a float until the end.

Note that the self-assignment operator works the same way no matter what is going on in the right side of the line of code; it doesn't just have to be a mathematical operator. In Figure 2.4.9, we create a string called `myString` on line 2, and print it on line 4. Then, we set the value of `myString` to the string version of its own length on line 7. `len(myString)` returns the number of characters in `myString`, which is originally 12. So, `len(myString)` is replaced by the integer 12. `str(12)` then changes the integer 12 to the string "12," and sets `myString` equal to "12." Then on line 11, we repeat that, this time with "12" as the value of `myString`, which ends with `myString` having the value "2," since there are two characters in "12."

#	SelfAssignment-2.py	Output
1	`#Create myNum and set it equal to 5`	5
2	`myNum = 5`	8
3		5
4	`print(myNum)`	15
5	`#Add 3 to myNum and give the result to myNum`	5
6	`myNum = myNum + 3`	125
7	`print(myNum)`	2
8	`#Subtract 3 from myNum and give the result to myNum`	0.6666666666666666
9	`myNum = myNum - 3`	
10	`print(myNum)`	
11	`#Multiply 3 by myNum and give the result to myNum`	
12	`myNum = myNum * 3`	
13	`print(myNum)`	
14	`#Divide myNum by 3 and give the result to myNum`	
15	`myNum = myNum // 3`	
16	`print(myNum)`	
17	`#Raise myNum to the 3rd and give the result to myNum`	
18	`myNum = myNum ** 3`	
19	`print(myNum)`	
20	`#Divide myNum by 3 and give the remainder to myNum`	
21	`myNum = myNum % 3`	
22	`print(myNum)`	
23	`#Divide myNum by 3 and set the result equal to myNum`	
24	`myNum = myNum / 3`	
25	`print(myNum)`	

Figure 2.4.8

#	SelfAssignment-3.py	Output
1	`#Set myString to "Hello, world"`	Hello, world
2	`myString = "Hello, world"`	12
3		2
4	`print(myString)`	
5	`#Find the length of myString, convert the length to a string,`	
6	`#and set myString equal to the result`	
7	`myString = str(len(myString))`	
8	`print(myString)`	
9	`#Find the length of myString, convert the length to a string,`	
10	`#and set myString equal to the result`	
11	`myString = str(len(myString))`	
12	`print(myString)`	
13		

Figure 2.4.9

The important takeaway here is that, as far as the computer is concerned, it really doesn't matter if the variable receiving a new value is used in the calculation *of* that new value. The computer evaluates the right side of the expression to get the value, then assigns that value to the variable. Having the same variable on both sides doesn't affect that process.

Self-Assignment Shortcuts

Self-assignment is very common in programming. We are constantly using a single variable to keep track of a changing value, and it's very common for that changing value to be based in some way on the current value. Counting, incrementing, tracking a balance, measuring lag time, and lots of other tasks will involve self-assignment. So, most languages—Python included—actually supply a little shortcut. You never *need* to use this shortcut because the syntax in Figure 2.4.8 will always work, but these tasks are common enough that simplifying them a bit can be helpful.

The self-assignment shortcuts all work basically the same way: you enter the variable receiving the value, then the mathematical operator you want to use, then the equal sign, then the expression that gives the value to use with the operator. It's simpler than it sounds, as seen in Figure 2.4.10.

#	SelfAssignmentShortcuts-1.py	Output
1	`#Create myNum and set it equal to 5`	5
2	`myNum = 5`	6
3		7
4	`print(myNum)`	12
5	`#Add 1 to myNum, and set the result equal to myNum`	
6	`myNum += 1`	
7	`print(myNum)`	
8	`#Add 1 to myNum, and set the result equal to myNum`	
9	`myNum += 1`	
10	`print(myNum)`	
11	`#Add 5 to myNum, and set the result equal to myNum`	
12	`myNum += 5`	
13	`print(myNum)`	
14		

Figure 2.4.10

This code is functionally equivalent to the code from Figure 2.4.7. `myNum += 1` translates to "Add 1 to `myNum`", or "Set `myNum` to the sum of `myNum` and 1." In many ways, this might makes self-assignment more intuitive because we see it as an operation on the variable itself, not an operation that treats the variable like it's two different things. So, the line `myNum = myNum + 1` is the same as `myNum += 1`.

These shortcuts apply to every mathematical operator. The code in Figure 2.4.11 is the equivalent of the code shown in Figure 2.4.8 as well.

#	SelfAssignmentShortcuts-2.py	Output
1	`#Create myNum and set it equal to 5`	5
2	`myNum = 5`	8
3		5
4	`print(myNum)`	15
5	`#Add 3 to myNum, and give the result to myNum`	5
6	`myNum += 3`	125
7	`print(myNum)`	2
8	`#Subtract 3 from myNum, and give the result to myNum`	0.6666666666666666
9	`myNum -= 3`	
10	`print(myNum)`	
11	`#Multiply 3 by myNum, and give the result to myNum`	
12	`myNum *= 3`	
13	`print(myNum)`	
14	`#Divide myNum by 3, and give the result to myNum`	
15	`myNum //= 3`	
16	`print(myNum)`	
17	`#Raise myNum to the 3rd power, and give the result to myNum`	
18	`myNum **= 3`	
19	`print(myNum)`	
20	`#Divide myNum by 3, and give the result to myNum`	
21	`myNum %= 3`	
22	`print(myNum)`	
23	`#Divide myNum by 3, and give the result to myNum`	
24	`myNum /= 3`	
25	`print(myNum)`	

Figure 2.4.11

This also works for those strange, nonmathematical applications of these operators. For example, strings use the + operator to put two strings together. So, when we run `myString += "!"` on line 5 in Figure 2.4.12, it appends `"!"` to the end of `myString`, as if we had said `myString = myString + "!"`.

#	SelfAssignmentShortcuts-3.py	Output
1	`#Create the string "Hello, world"`	Hello, world!
2	`myString = "Hello, world"`	
3		
4	`#Add '!' to the end of myString`	
5	`myString += "!"`	
6	`print(myString)`	
7		

Figure 2.4.12

Incrementing and Loops

The most common place we use incrementing is with loops. Loops are segments of code that are run multiple times, either a predetermined number (e.g., "run this ten times"), a number for each item in a list (e.g., "run this for each character in this string"), or while a condition is true (e.g., "run this until you find the word I'm looking for").

We'll talk about loops more extensively in Chapter 3.3, but for now, we'll use a simple example to demonstrate incrementing. Imagine we wanted to measure the length (number of characters) of a string, and we didn't realize the `len()` function existed. How would we do that?

#	IncrementingandLoops.py	Output
1	`#Create letterCount and set it equal to 0`	12
2	`letterCount = 0`	
3	`#Run this code for each letter in the string`	
4	`for character in "Hello, world":`	
5	`    #Add one to letterCount`	
6	`    letterCount += 1`	
7	`print(letterCount)`	
8		

Figure 2.4.13

Line 4 of this code, beginning with `for`, basically says "Repeat the code indented below for each character in the string." The string has twelve characters, and so `letterCount += 1` runs 12 times. So, in the end, `letterCount` equals 12. This is a trivial example, of course, but it is the foundation of more complex examples. For example, imagine you wanted to find the number of students in a class receiving As, Bs, etc. You'd loop through every student in the class, and if they were receiving an A, you'd increment a counter for A; if they were receiving a B, you'd increment a counter for B; and so on.

5. Operators Together

We've now covered both types of operators: logical and mathematical. Did you know, though, we can use them together? We'll do this more when we reach conditionals next in Chapter 3.2, but in the meantime, let's look at some of the ways we can use logical and mathematical operators together.

Checking If a Triangle Exists

In geometry, there's a rule about triangles that says: three side lengths can form a triangle if and only if the sum of the two shorter sides' lengths is longer than the longest side's length. Don't worry if that doesn't sound familiar, you don't need any background in this to understand this example. Imagine if we want to write some code that takes three side lengths from the user and checks if a triangle can be made out of them.

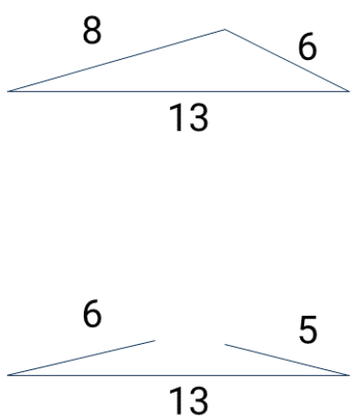


Figure 2.4.14

There's a lot going on in Figure 2.4.15, so let's take this one step at a time. First, in lines 2, 4, and 6, notice how we're getting input from the user using the `input()` function, and then converting it to an integer in the same line with the `int()` function. We could have done this in two lines for each side, but since we never need the string version of the user's input, we might as well just convert it to an integer immediately. If that's confusing, glance back at 2.2.7.

Once we have these three sides, we move on to line 9. Here, we create a variable `result` to store the results of our question. We then ask: The sum of `side1` and `side2` is larger than `side3?` `True` would mean yes, `False` would mean no.

```
#  CheckingIfaTriangleExists.py
1  #Get the first side from the user
2  side1 = int(input("Enter the shortest side: "))
3  #Get the second side from the user
4  side2 = int(input("Enter the next shortest side: "))
5  #Get the third side from the user
6  side3 = int(input("Enter the longest side: " ))
7
8  #Check if the shorter sides' sum exceeds the longer side
9  result = (side1 + side2) > side3
10 #Print the result
11 print("'These sides can form a triangle' is", result)
12
```

```
Output
Enter the shortest side: 1
Enter the next shortest side: 2
Enter the longest side: 3
'These sides can form a triangle'
is False
```

Figure 2.4.15

How does the computer evaluate this? It starts with the innermost parentheses, so it evaluates `side1 + side2`, which gives 3. It then effectively replaces (`side1 + side2`) with 3. Now, it evaluates 3 > `side3`, where `side3` has the value 3. 3 is not greater than 3 (it's equal), and so this statement is `False`. `False` is then stored as `result`, as seen when we print `result` on line 11.

The print statement on line 11 here is somewhat new, but it's extremely useful: we're labeling the print statement with some indicator of what it means. The computer implicitly converts `result` to a string, and then puts the statement in quotes and the text of `result` together.

Composition of Operators

The example in Figure 2.4.15 is here for another reason: it shows how we can put together functions, such as type conversions, and operators in a single line. We're starting to move into some more advanced ways of writing lines of code.

Imagine, for example, we wanted to write some simple code that could be used to grab two numbers from the user and print their sum. Figure 2.4.16 shows one way we could do that.

```
#  CompositionofOperators-1.py
1  #Create num1 and give it the value provided by the user
2  num1 = input("Enter the first number: ")
3  #Create num2 and give it the value provided by the user
4  num2 = input("Enter the second number: ")
5  #Create num1AsInt and give it the result of converting
6  #num1 to an integer
7  num1AsInt = int(num1)
8  #Create num2AsInt and give it the result of converting
9  #num2 to an integer
10 num2AsInt = int(num2)
11 #Create sum and give it the sum of num1AsInt and num2AsInt
12 sum = num1AsInt + num2AsInt
13 print(sum)
14
```

```
Output
Enter the first number: 1
Enter the second number: 2
3
```

Figure 2.4.16

This works perfectly fine, but does it really take us six lines of code to perform one of the most simple programs with user input we could write? As we've said before, it's not good to strive for the fewest lines possible when you're working in computing, but it's a good exercise to test your coding ability. Try to shorten this code down to four lines, then to three lines, and then to one line.

Borrowing from our earlier examples, you may quickly see that lines 2, 4, 7, and 10 could have been smooshed into two by performing the type conversion alongside the input, as we did in Figure 2.4.15. You may have also recognized you could just print the sum instead of saving it in the variable `sum`. That would take you to three lines, but that last jump to one line might be tough.

Figure 2.4.17 shows how it could be done, though. I should note that this isn't something you'd want to do in a real programming environment; this line is hard to read. However, it's useful to show what we can do. Let's step through how the computer runs this line. Like we've said, it starts with the innermost parentheses. In

```
# CompositionofOperators-2.py
1 print(int(input("Enter the first number: ")) + int(input("Enter the second number: ")))
2
3
4
5
```

Figure 2.4.17

this case, there are two parentheses at equal levels: the two sets of parentheses following the two `input()` functions. That means that input gets executed first, from left to right. So, the program first gets the user's input.

So, the two input functions are now basically replaced by "1" and "2" from the user's input. Note that the input function gets the user's input as a string, so right now, our effective line of code reads, `print(int("1") + int("2"))`. Now, the computer again starts with the innermost parentheses, which now are the parentheses following the `int()` functions, the type conversions. It runs `int("1")` to get 1 and `int("2")` to get 2, and those integers now replace those function calls. So, now the effective line is `print(1 + 2)`. Now, it evaluates the innermost parentheses again, finds that 1 + 2 resolves to 3, and prints 3.

This is a complex example of how the computer interprets this line. If this was confusing, don't worry. This takes practice to really understand.

6. Operators and Context in Python

One of the unique things about operators is that they can be designed to react intelligently to the types of data they're asked to act on. We've already seen a little bit of this: when we divided two integers together, Python knew to return the result as a float since the result would typically have a decimal attached to it. Let's check out a few examples of operators reacting differently based on their context (i.e., the type of data they're operating on).

Integers and Floats

First, let's look at how the different operators react to being asked to work on a combination of integers and floats.

In Figure 2.4.18 on line 6, when the addition operator is used on two integers, the result is an integer. This makes sense; if both numbers are integers, decimals aren't going to appear out of nowhere. When the addition operator is used on an integer and a float on line 8, the result is a float. This also makes sense; we can always easily convert an integer to a float by adding .0, but we cannot always easily convert a float to an integer without losing some information (like dropping the decimal point). Sometimes we can (and here, we could), but it's safer to assume that the result should be a float if any of the operands were floats.

```
# IntegersandFloats-1.py
1 myInt1 = 1
2 myInt2 = 2
3 myFloat = 3.0
4
5 #Print the sum of two integers
6 print(myInt1 + myInt2)
7 #Print the sum of an integer and a float
8 print(myInt1 + myFloat)
9
```

```
Output
3
4.0
```

Figure 2.4.18

Python operates similarly across all the mathematical operators, as shown in Figure 2.4.19; whenever we mix an integer and a float, the result is a float. This is generally unsurprising, except for potentially where we use the floor division

#	IntegersandFloats-2.py	Output
1	`myInt = 2`	5.0
2	`myFloat = 3.0`	-1.0
3		6.0
4	`#Print the sum of an integer and a float`	0.6666666666666666
5	`print(myInt + myFloat)`	8.0
6	`#Print the difference of an integer and a float`	0.0
7	`print(myInt - myFloat)`	2.0
8	`#Print the product of an integer and a float`	
9	`print(myInt * myFloat)`	
10	`#Print the quotient of an integer and a float`	
11	`print(myInt / myFloat)`	
12	`#Print the result of an integer raised to a float`	
13	`print(myInt ** myFloat)`	
14	`#Print the rounded quotient of an integer and a float`	
15	`print(myInt // myFloat)`	
16	`#Print the remainder of an integer and a float`	
17	`print(myInt % myFloat)`	
18		

Figure 2.4.19

operator. Even when operating on floats, the floor division operator is guaranteed to return a whole number because it counts whole times one number goes into another; however, when it is used on a float, it still returns a float.

String Operators

Interestingly, there are also instances where these operators can be used on strings. That might seem strange since most of these operators heavily suggest something numeric. The mathematical operators obviously work on numbers, and the relational operators typically focus on numerical relationships as well. However, some of our operators work on strings.

#	SelfAssignmentShortcuts-3.py	Output
1	`#Create the string "Hello, world"`	Hello, world!
2	`myString = "Hello, world"`	
3		
4	`#Add '!' to the end of myString`	
5	`myString += "!"`	
6	`print(myString)`	
7		

Figure 2.4.20

We've seen one example of this already. The addition operator, when applied to strings, puts two strings together into one new string. In Figure 2.4.20, we start with "Hello, world", and we add "!" to it, ending up with "Hello, world!". This is the only mathematical operator that works on pairs of strings. What about relational operators, though?

#	StringOperators-1.py	Output
1	`myAString = "A"`	True
2	`myBString = "B"`	False
3	`myCString = "C"`	True
4		
5	`print(myAString < myBString)`	
6	`print(myCString < myAString)`	
7	`print(myAString == myAString)`	
8		

Figure 2.4.21

Perhaps surprisingly, our relational operators work fine on strings as well. It's not too surprising that the equality operator works on line 7 of Figure 2.4.21 since

it makes sense to check if two strings have the same characters, but what does it mean for one string to be "greater" than the other? In this context, it means the string comes later alphabetically. One string is greater than another if, when sorted alphabetically, it would come after it. In Figure 2.4.21, "A" is less than "B" in line 4, and "C" is not less than "A" in line 5. We can use this to sort strings alphabetically, although beware that Python sorts uppercase and lowercase strings separately.

In Figure 2.4.22, we see that as expected, "A" is less than "B" because it comes first alphabetically. Perhaps surprisingly, "A" is also less than "a"; in Python's interpretation, all uppercase letters are less than all lowercase letters. That explains the last line: "a" is not less than "B."

#	StringOperators-2.py	Output
1	`myAString = "A"`	`True`
2	`myaString = "a"`	`True`
3	`myBString = "B"`	`False`
4		
5	`print(myAString < myBString)`	
6	`print(myAString < myaString)`	
7	`print(myaString < myBString)`	
8		

Figure 2.4.22

Finally, one last surprising instance of operators understanding their context happens when we try to multiply a string by an integer. That doesn't make any logical sense to us: how do you multiply text by a number? Python, however, has an interpretation for it, shown in Figure 2.4.23: it duplicates the string the given number of times. Interestingly, this is the only operator that works this way for strings, and it only works with a string times an integer. A float times a string will not have the same result, and will instead give an error.

#	StringOperators-3.py	Output
1	`myAString = "A"`	`AAAAA`
2	`myInt = 5`	
3		
4	`print(myAString * myInt)`	
5		

Figure 2.4.23

7. Operators and Turtles

Now that we've covered lots of how operators can work, let's put them into action with our turtles domain.

Exponential Circle Growth

First, let's check out a simple application of the exponentiation operator. In ExponentialCircleGrowth-1.py, we create an initial radius `myInt`, and then we draw increasing sizes of circles based on raising `myInt` to increasing powers. So, we see the circles growing at an exponential rate: they start pretty small, but quickly they get extremely larger. We haven't reached loops yet, but we talked about them a little when we talked about incrementing, so let's take a look real quick at how this code could be simplified using a loop to show off incrementing.

We can cut those eleven lines down to six lines by using a while loop. A while loop basically says, "Keep running the code below until something is true." In ExponentialCircleGrowth-2.py, it continues to run the code indented below it until `myExponent` is at least 10, which will happen eventually since `myExponent` is incremented each time the loop runs in the last line. Note also that while this cuts

this code from 11 lines to 6 lines, these 6 lines could draw any number of circles just by changing the number at which the code stops. If we changed `myExponent < 10` to `myExponent < 100`": the loop would run 100 times. I don't recommend that, though: on my computer, that code would take 212 million years to run to completion, and would generate a circle with the radius roughly equivalent to that of the observable universe.

Visualizing Modulus

We can also visualize our modulus function using our turtles library. Let's write a program, VisualizingModulus.py, that will divide two numbers, and draw how much of the division is the rounded quotient and how much is the remainder.

First the code gets two numbers from the user. It then draws the dividend, the whole number, as a black line on the bottom. The turtle then moves to draw the top line. It then draws a line with length `((dividend // divisor) * divisor)`. What this does is finds the number of whole times the divisor goes into the dividend, then draws a line that covers all those times in red. For example, if the divisor goes into the dividend three times, it draws a line with length three times the divisor. This covers the times the divisor goes into the dividend in its entirety. It then uses `dividend % divisor` to find the remainder after that, and draws a red line with that width.

In the abstract, this is tough to follow, so let's talk about this exact example. Try entering the numbers 17 and 6. That means we want to divide 17 by 6. How many whole times does 6 go into 17? It goes in two times: 6×2 is 12, which is less than 17, but 6×3 is 18. So, 6 goes into 17 twice, with some remainder. The remainder is 5 because $17 - 12$ is 5. First, this code draws a line with length 17 to show the entire number we're dividing. Then, the turtle moves up to draw the quotient-and-remainder line. It grabs the quotient, which is 2, and multiplies it by the divisor, 6, to get 12. So, it draws a red line with length 12. This covers the times 6 completely goes into 17. Then, it grabs the remainder, which is 5, and draws a blue line to cover the rest of the way. The top line is always guaranteed to be the same length as the bottom line, only divided between the red quotient and the blue remainder.

UNIT 3

CONTROL STRUCTURES

CHAPTER 3.1

Control Structures

1. What Are Control Structures?

At this point, we've covered the basic design of procedural computer programs. Every program we've seen, except for the glimpses forward we've taken, has run a linear series of lines of code in order from first to last, generating output. In doing so, we've been able to do some powerful things, but at the same time, what we can do is somewhat limited. For example, we can draw a hexagon in around twelve lines of code, but what if we want to draw an octagon? So far, that means writing new lines of code, lengthening the program. Wouldn't it be great if we could just say, "I'd like a shape with 8 sides" and get an octagon instead of having to write the code?

What Do Control Structures Do?

Control structures are where we start to have that capability. Control structures let us loop over certain lines of code multiple times, changing the data they act on each time. Control structures let us branch our code based on the result of some conditional statement, like returning one message if a customer has sufficient money to make a purchase and another if they don't. Control structures let us repackage code that is commonly used into functions, like validating a customer's information or making a series of turns in a vehicle. Control structures let us anticipate certain errors and react gracefully instead of crashing.

The content of these control structures will largely be the same kinds of code we covered with procedural programming; control structures, in many ways, simply control what lines of code will be run in what order. This makes what we can create orders of magnitude more powerful.

2. The Control Structures

We'll generally break our conversation about control structures into four types of structures: conditionals, loops, functions, and exception handling.

Conditionals

One of the first types of structures we'll cover to add to our programming toolbox is **conditional statements**. Conditionals basically tell the computer to make a decision. Depending on that decision, it might execute some code or skip that code; or, it might choose between two different blocks of code to execute.

Conditionals build on our logical operators that we covered last unit. In fact, almost every conditional statement reacts to the result of a logical expression. For example, imagine we're writing code to validate a transaction at the store. We might write, "If the customer has a sufficient balance for the purchase, then permit the purchase; otherwise, reject the purchase." This is a conditional statement based on the result of the relational evaluation of the customer's balance and the purchase price. Depending on the result of that logical expression, the code will do different things.

Conditionals can be used to make very complex code structures. For example, you could nest several conditional statements one after the other to check the

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Identify the types of control structures and illustrate them with examples;
- Demonstrate the use of indentation with control structures;
- Identify code blocks and a variable's scope in Python.

Control Structures
Statements that control the flow of execution of the program. Or, more simply, lines of code that control when other lines of code run.

Conditional Statements
Programming statements that control what code is executed based on certain conditions; usually of the form "`if`", "`else if`", and "`else`".

customer's balance, the retailer's authenticity, and the cardholder's identity. You can also write single conditional structures that react to multiple conditions; for example, if a purchase is rejected, you might want your code to *then* check for fraud if other suspicious purchases have been attempted. In this way, conditionals are powerful tools for creating complex code.

Loops

Loop
A programming control structure that executes a segment of code multiple times.

A **loop** involves executing certain lines of code multiple times. Multiple times might be a certain number of times; it might be for every item in a list, like for every file in a folder; or it might be while some condition remains true, like reading from a file as long as you haven't reached the end yet.

We can think of our example of programming a cash register in terms of loops. First, the register would loop through every item that the customer is purchasing. For every item, it would execute several lines of code: it would update the customer's running total, it would update the store's internal inventory, it would calculate and add tax to the product's value, and so on. The same lines of code would be executed for each item the customer is purchasing. What's more, that loop, as well as operations for getting the payment method and validating the purchase, would be run while the store still has customers waiting in line. So, that's another loop, this time one operating on each customer in order.

Like conditionals, loops can be complex and nested. In fact, the example above would be a nested loop: we would run the loop over a single customer's items for every customer in line. We can also use loops in other complicated ways as well. For example, if we wanted to write a program that would consistently listen for some interruption from a server, we could simply tell it to loop indefinitely until a message was received.

Function

Function
A segment of code that performs a specific task, sometimes taking some input and sometimes returning some output.

A **function** is a way of packaging together multiple lines of code in a way that allows them to be easily used wherever needed. In effect, it removes the need to copy and paste lines of code around our program when needed because instead of copying them, we can just "call the function" that contains them. Practically speaking, this is like dynamically inserting the lines of code from the function into the rest of the code and running them right there.

You've already seen some examples of functions. This isn't because we were trying to get ahead of ourselves, but rather it's because functions are so fundamental to modern programming that it's difficult to show anything without using some functions. Even something as fundamental as printing to the console is usually run through a function. In practice, functions behave just like operators: they take some input and produce some output. Operators are simply low-level, extremely common functions, but every operator could be rewritten as a function.

The real power of functions is their ability to take lots of different pieces of input and produce some output. You could have, for example, a single function called `validatePurchase()` that takes as input a customer's name, credit card number, purchase amount, current balance, and retailer name, and returns either `True` or `False` to indicate whether the purchase is valid. This goes far beyond just adding or subtracting a couple of numbers; functions can handle complex operations.

Exception Handling

Earlier in our material, we covered the idea of errors. Errors occur when your code tries to do something it can't do such as accessing files that don't exist or dividing by zero. So far, we've usually talked about errors in the context of debugging. However, can we actually use errors in the design of our programs?

Exception handling tries to do exactly this. With **exception handling**, we anticipate certain errors ("exceptions," in this case) might arise, and we program in a way to recover from them. In many ways, they're extremely similar to conditionals; you can think of nearly every instance of exception handling as saying, "If an error is encountered in the following code, then…"

When would we want to anticipate and handle errors instead of just avoiding them? Think of our example of loading information from a file. Let's say we prompt the user to enter a filename. We have no guarantee the filename they enter is valid. So, we need to check it first before trying to load it, right? That's one way we could do it. However, we know that if the file doesn't exist, we should get an error that says "file not found." So instead of checking if the file exists before trying to load it, we could instead just try to load it, and prepare our code to handle a "file not found" error if it arises.

Exception
An error that a program might want to anticipate and catch instead of outright avoiding.

Exception Handling
A control structure that catches certain anticipated errors and reacts to them accordingly.

3. Indentation and Control Structures in Python

The fundamental idea of control structures is that certain lines of code tell the computer how to interpret or when to execute other lines of code. With a conditional statement, for example, certain lines of code only run if the logical expression is true. However, that means the code needs some way of telling the computer which lines of code apply. In most languages, this is taken care of with reserved characters like brackets around the lines of code; Python, interestingly, uses **indentation**.

Indentation
Spaces at the beginning of a line that are used to group together blocks of code. All consecutive lines of code at the same level of indentation are in a single code block.

Indentation and Conditionals

Let's look at this with a simple example of a conditional statement. We'll talk more about conditionals in the next lesson, but for now, just know that the third line of Figure 3.1.1 says, "if `myNum1` is less than `myNum2`, do the indented line of code below."

#	IndentationandConditionals-1.py	Output
1	`myNum1 = 1`	`myNum2 is greater than myNum1!`
2	`myNum2 = 2`	`Execution complete!`
3	`#Checks if myNum1 is less than myNum2`	
4	`if myNum1 < myNum2:`	
5	`    #Prints this if so`	
6	`    print("myNum2 is greater than myNum1!")`	
7	`#Prints this regardless`	
8	`print("Execution complete!")`	
9		

Figure 3.1.1

Notice in Figure 3.1.1 that both print statements ran. That's because the conditional statement on line 4 said, "If this expression is true, run the indented code below." When we change the results of the conditional statement by changing the values of `myNum1` and `myNum2`, check out what happens in Figure 3.1.2.

#	IndentationandConditionals-2.py	Output
1	`myNum1 = 3`	`Execution complete!`
2	`myNum2 = 2`	
3	`#Checks if myNum1 is less than myNum2`	
4	`if myNum1 < myNum2:`	
5	`    #Prints this if so`	
6	`    print("myNum2 is greater than myNum1!")`	
7	`#Prints this regardless`	
8	`print("Execution complete!")`	
9		

Figure 3.1.2

Here, the indented code on line 6 did not run because the conditional statement was false (because `myNum1` is now 3, not 1). In effect, the conditional statement "controls" the indented line below it; that's why we call it a control structure. However, the important thing to note here is that the non-indented line of code (line 8) *did* run. It's outside the indentation, so it executes regardless.

This applies to multiple indented lines as well. None of the indented lines in Figure 3.1.3 executed. They are one code block that runs only if the conditional statement is true. This organizational method applies to every code structure we'll talk about in this chapter: conditionals, loops, functions, and exception handling all group together code through indentation, and all mark their blocks of code with a colon at the end of the preceding line (here, if `myNum1 < myNum2:`).

#	IndentationandConditionals-3.py	Output
1	`myNum1 = 3`	`Execution complete!`
2	`myNum2 = 2`	
3	`#Checks if myNum1 is less than myNum2`	
4	`if myNum1 < myNum2:`	
5	`    #Prints all these if so`	
6	`    print("myNum2 is greater than myNum1!")`	
7	`    print("Yes it is!")`	
8	`    print("Yes it is!")`	
9	`#Prints this regardless`	
10	`print("Execution complete!")`	
11		

Figure 3.1.3

Nested Indentation

Indentation can be nested as well; this is how we create nested conditionals or nested loops. Look at Figure 3.1.4, with three numbers.

#	NestedIndentation-1.py	Output
1	`myNum1 = 1`	`myNum2 is greater than myNum1!`
2	`myNum2 = 2`	`myNum3 is also greater than myNum1`
3	`myNum3 = 3`	`Execution complete!`
4	`#Checks if myNum1 is less than myNum2`	
5	`if myNum1 < myNum2:`	
6	`    #Prints this if so`	
7	`    print("myNum2 is greater than myNum1!")`	
8	`    #Checks if myNum1 is less than myNum3`	
9	`    if myNum1 < myNum3:`	
10	`        #Prints this if so`	
11	`        print("myNum3 is also greater than myNum1")`	
12	`#Prints this regardless`	
13	`print("Execution complete!")`	
14		

Figure 3.1.4

Line 9 is also indented under line 5, so line 9 is controlled by line 5. Here, the conditional statement in line 5 is `True`, so the computer reaches line 9. The conditional statement in line 9 is also `True`, so the computer runs line 11. Notice what happens if the conditional statement in line 5 is `False` when we switch `myNum1` and `myNum2` in Figure 3.1.5.

`myNum1` is still less than `myNum3`, so *if* the computer had reached line 9, it would have evaluated to `True` and printed line 11. However, line 9 is controlled by line 5, and line 5 was `False`, so the computer skips over the indented block (lines 6 through 11) containing line 9 and just runs the print statement in line 13.

#	NestedIndentation-2.py	Output
1	`myNum1 = 2`	`Execution complete!`
2	`myNum2 = 1`	
3	`myNum3 = 3`	
4	`#Checks if myNum1 is less than myNum2`	
5	`if myNum1 < myNum2:`	
6	`    #Prints this if so`	
7	`    print("myNum2 is greater than myNum1!")`	
8	`    #Checks if myNum1 is less than myNum3`	
9	`    if myNum1 < myNum3:`	
10	`        #Prints this if so`	
11	`        print("myNum3 is also greater than myNum1")`	
12	`#Prints this regardless`	
13	`print("Execution complete!")`	
14		

Figure 3.1.5

4. Scope

Nearly every programming language has some concept of scope. Scope most often describes what portions of a program can see a particular variable. It's like your program's short-term memory: what is it remembering at a given time? You can't access something it's no longer remembering. We can extend the idea of scope to functions, classes, and other advanced concepts that we'll get to later, but most often for our material, **scope** refers to which parts of a program can see the variables that you've declared.

Scope
The portion of a program's execution during which a variable can be seen and accessed.

Examples of Scope

At a certain level, scope is obvious. You're seeing code written on the page in this book. Can your code access the variables I'm writing here? Of course not (unless you copied the code into your code, but then it's your code, not my code). What about if you have two code files on your computer; can one see the variables in the other? There may be ways to tell one file how to see the other, but that would have to be done explicitly; just having two files of code on one computer that mention the same variable doesn't mean they'll see one another.

So, a lot of time scope is pretty easily-defined. You expect the scope of your program to have an outer limit of all the code you're running at a given time. That's a characteristic of nearly every language I've encountered. A second aspect of scope is a little more complex, but still makes some logical sense: scope is often defined linearly. In other words, the computer doesn't know what that a variable exists until it encounters it. If you create a variable named `myVariable` on line 10, you can't refer to it until you reach line 10. The scope begins when the variable is first created.

Within these general limits, though, things can get a little trickier. Some languages behave differently. Most languages, however, use control structures to define scope.

Control Structures and Scope

So why do we discuss scope when we discuss our introduction to control structures? In many languages, control structures define the scope. A single variable usually "lives" within a control structure's definition, and once the computer leaves that control structure, the variable is lost. In most (but not all) languages, this is especially true for conditionals and loops: the code inside of a conditional's code block can see variables that were defined *before* the conditional, but the code that runs *after* the conditional can't see any variables that were created *inside* its code block.

Why is this? The reason is that the computer can't guarantee that the contents of a conditional statement's code block actually ran. If you create `myVariable` inside some conditional, then the code outside the conditional doesn't know that the line

of code creating `myVariable` was ever run. `myVariable` might not exist. But this only applies to variables *created* inside the conditional; if a variable was created *before* the conditional, it can be referenced both *inside* and *after* the conditional. Some languages will let you use `myVariable` after the conditional's code block anyway, but you risk causing an error.

Functions define scope even more narrowly. A function's scope is, generally, just the variables sent to it or created within it. Functions don't automatically see anything created before they're run; they only see those things that are sent into them intentionally. Functions can see some variables that are defined even more globally—it's possible to define variables in a way that forces them to be visible everywhere in a program. For now, though, we'll focus on the more natural forms of scope. Don't worry about understanding everything here right now, though; we're just previewing the general idea of scope so we can return to it in each individual chapter of this unit.

5. Scope in Python

Python's scoping rules are actually simpler than many languages'. Part of this is because Python is an interpreted language, not a compiled language—it can live with certain things being unknown. Python doesn't mind if it can't tell if a line that creates a variable will be executed or not.

Simple Scope in Python

Let's start with a simple example. Figure 3.1.6 is a revised version of the conditional we saw in Figure 3.1.1. The revision makes one change: instead of printing inside the conditional (the `if` statement), it saves the result to a string called `result`. It then prints `result` after the conditional has executed. What happens?

#	SimpleScopeinPython.py	Output
1	`myNum1 = 1`	`myNum2 is greater than myNum1!`
2	`myNum2 = 2`	`Execution complete!`
3	`#Checks if myNum1 is less than myNum2`	
4	`if myNum1 < myNum2:`	
5	`    #Saves this if so`	
6	`    result = "myNum2 is greater than myNum1!"`	
7	`#Prints the result`	
8	`print(result)`	
9	`print("Execution complete!")`	
10		

Figure 3.1.6

Well, `myNum1` *is* less than `myNum2`, so the contents of the conditional on line 4 run. The variable `result` is created, and it is then printed. Python didn't care that result was created inside the conditional's code block. This makes it different than many languages; many languages only define variables as existing within that code block.

In some ways, this makes programming in Python simpler. If we know that we're going to create result at some point, we don't have to worry about creating it at the wrong point. However, it isn't all good news.

The Dangers of Scope in Python

Scope in Python also presents a danger. Take a look at the simple tweak in Figure 3.1.7 to the code from Figure 3.1.6. All we've done is change the values of `myNum1` and `myNum2` so that now the conditional statement on line 4 doesn't trigger. That means line 6 never runs, which means `result` is never created. So, what happens

#	TheDangersofScopeinPython-1.py	Output
1	`myNum1 = 2`	`Traceback (most recent call last):`
2	`myNum2 = 1`	`File "...", line 8, in <module>`
3	`#Checks if myNum1 is less than myNum2`	`print(result)`
4	`if myNum1 < myNum2:`	`NameError: name 'result' is not defined`
5	`    #Saves this if so`	
6	`    result = "myNum2 is greater than myNum1!"`	
7	`#Prints the result`	
8	`print(result)`	
9	`print("Execution complete!")`	
10		

Figure 3.1.7

when we run this code? Line 8 gives us an error. It hits an error because `result` was never created because line 6 was never run. This goes back to the idea that a variable's scope begins when it is created; if the variable is never created, it has no scope.

This is the danger of scope in Python. When everything is working correctly, scope in Python is relatively easy to understand because you can think of the control structures as just determining what lines of code run in what order; the result is those lines of code running as if they had been written that way in the first place. However, if a control structure is going to start interfering with whether or not variables get created, then you might run into some issues.

You can avoid this by creating variables outside the control structures just in case, as shown in Figure 3.1.8. Here, we create `result` initially before the conditional, so that even if the conditional doesn't execute, `result` is still created. Generally, for our purposes, knowledge of scope is most useful in debugging; when you encounter errors, one of the first things to check is whether the error is due to scope problems. Are you trying to access a variable that was created inside a conditional that didn't run? Then you have a scope error.

#	TheDangersofScopeinPython-2.py	Output
1	`myNum1 = 2`	`Result was unchanged.`
2	`myNum2 = 1`	`Execution complete!`
3	`#Creates an initial value for result`	
4	`result = "Result was unchanged."`	
5	`#Checks if myNum1 is less than myNum2`	
6	`if myNum1 < myNum2:`	
7	`    #Saves this if so`	
8	`    result = "myNum2 is greater than myNum1!"`	
9	`#Prints the result`	
10	`print(result)`	
11	`print("Execution complete!")`	
12		

Figure 3.1.8

If you go into computing, one day you'll learn other languages as well. This knowledge of scope will also be useful for that transition. When working in Python, we take this for granted because it makes scope relatively easy, but in many other languages, it's a more significant topic.

CHAPTER

3.2

Conditionals

1. What Are Conditionals?

The term **"conditional"** comes from the idea that sometimes we want to run some code *conditionally*; in other words, we only want to run it if something is true. For example, *if* the user has modified a document since the last saved, *then* we want to ask them if they want to save before closing the program. *If* a user has entered a different password than the one saved, *then* we want to ask if they want to update the saved password. *If* a user's bank balance is insufficient to cover a purchase or the retailer is not trusted, *then* we want to reject the purchase.

Modern programming couldn't exist without conditionals like these. They're a relatively simple principle (but don't worry if you don't get them at first), but they're extremely powerful.

If-Then

The most fundamental form of conditional is the simple `if-then` statement. *If* something is true, *then* do something. We think in terms of conditionals every day. Consider:

- If it's cold outside, then wear a long-sleeved shirt.
- If highway traffic is bad, then take surface streets.
- If you have a test tomorrow, then study.
- If you're a vegetarian, then order the vegetarian entrée.

Each of these is easily phrased in terms of an `if-then` statement. You check if some condition is true, and if so, you take some action. The "action" could actually be several actions. You could imagine, for example, that if it's cold outside, then you wear warmer clothing, start the car early so it can warm up, and makeg some hot coffee.

If-Then-Else

A slightly more complicated version of this includes a third part: an `else`. The `else` is a different series of actions to perform if the condition wasn't true in the first place. With an `if-then-else` structure, you'll always do one thing or the other.

We can rewrite our real-world examples above in terms of if-then-else:

- If it's cold outside, then wear a long-sleeved shirt; else, wear a t-shirt.
- If highway traffic is bad, then take surface streets; else, take the highway.
- If you have a test tomorrow, then study; else, go to a party.
- If you're a vegetarian, then order the vegetarian entrée; else, order the meat entrée.

Just like the original `then`, there could be multiple actions that follow an `else`. The `else` following the conditional on whether it's cold outside could be: wear a t-shirt, make some iced coffee, plan to go to the park after school or work, and pack a water bottle. The important thing here is that `if-else` structures create two alternatives, one of which will always be chosen.

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Describe the structure of different types of conditional statements and their usage with mathematical, relational, and Boolean operators;
- Implement different types of conditional statements with operators and analyze the effect of conditionals on the scope of a variable;
- Write a script using conditional statements to control the turtle with user input.

Conditional Statements
Programming statements that control what code is executed based on certain conditions; usually of the form "`if`", "`else if`", and "`else`".

If-Then Statement
A conditional control structure that runs a block of code only if a certain condition is `true`.

Else Statement
A conditional control structure that runs a block of code if all preceding `if-then` and `else-if` statements have been false.

Else-If Statement
A conditional control structure that runs a block of code if all preceding `if-then` and `else-if` statements have been false *and* some other conditions are met.

If-Then-Else-If

Sometimes, though, our reasoning might be more complex. We might need multiple pathways depending on different checks. In this case, we might employ an `else-if` statement. Like an `else`, an `else-if` only runs if the original `if-then` did not. Unlike an else, however, an `else-if` has its own conditions to check; if the conditions aren't met, it doesn't run either.

Consider this more complex version of our weather example: *If* it's raining, then wear a raincoat; *else, if* it's cold, then wear a long-sleeved shirt; *else*, wear a t-shirt. Here, we check two things: whether it's raining, and whether it's cold. If it's raining, we don't need to bother checking if it's cold: we wear a raincoat regardless. Otherwise, or *else*, we need to check if it's cold, and if so, wear a long-sleeved shirt. That's what makes this an `else-if`: it only runs if the original `if` wasn't true, but it still has its own conditions.

Multiple Else-Ifs

We can chain together multiple `else-if`s as well. For example, we could say: *if* it's raining, then wear a raincoat; *else, if* it's cold, then wear a long-sleeved shirt; *else, if* it's hot, then wear a t-shirt; *else, if* it's windy, then wear a jacket; *else*, wear a collared shirt. We must start with `if`, and we can have at most one `else`, but we can have any number of `else-if`s in between. With this kind of structure, each `else-if` and `else` will only execute if *no previous* condition has executed. If it was cold, then this logic wouldn't check if it was hot or windy. A collared shirt would only be the result if *every* previous statement was false.

This might be easier to visualize using a flowchart. From here, we can see that once one of the conditions is true, it changes our path and sends us to one of the results. So, we don't even check the other questions because we've already reached our decision of what to do next. Flowcharts like this can be useful ways of planning out your code if you're having trouble keeping track of it in text.

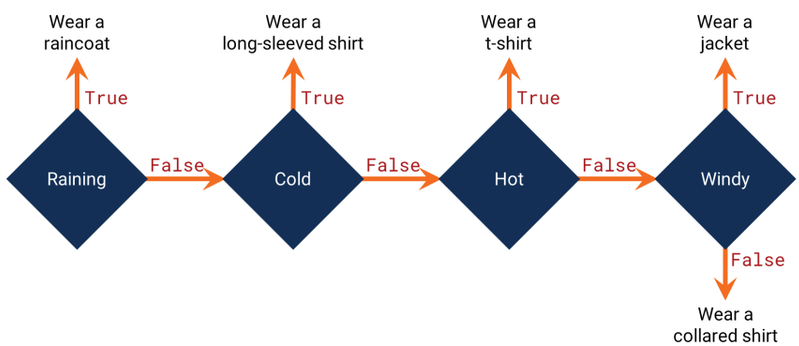


Figure 3.2.1

If we wanted to guarantee we check multiple things, we would just put multiple `if-then` structures one after the other. For example, imagine we said: *if* it's cold, *then* wear a long-sleeved shirt; *if* it's raining, *then* wear a raincoat. With this logic, we could end up wearing both a long-sleeved shirt and a raincoat if it's cold and raining; the second statement doesn't begin with `else`, so this reasoning checks if it's raining whether it's cold or not.

Just as we didn't have to end an `if-then` statement with an `else`, we also don't have to end an `if-then-else-if` with an `else`. For example, consider this reasoning: *If* you have a test tomorrow, then study; *else, if* you have class early, then

go to bed early. If neither of these conditions is true, then this block doesn't need to prescribe what you do. We can have an `if-then-else-if` without having a final `else`.

Conditionals Recap

So, to recap: our basic conditional structure is the `if-then` structure; it checks if some condition is true, and runs some code if so. We can augment our `if-then` structure with `else-if` and `else`. `else-if` checks additional conditions *if* the earlier ones were false. `else` always performs some actions if no previous `if` or `else-if` was true.

Right now, I would predict you feel like you kind-of get this and kind-of don't. If you're unsure, don't worry. We've covered a lot in this lesson, but the remainder of this chapter is applying just these concepts to different contexts and combining it with what we learned last unit. If you find yourself stuck, try to think about the principles in terms of those real-world decisions instead of coding conditionals.

2. Conditionals in Python

Now that we've covered the basic principles of conditional statements, let's see them in action. To demonstrate these, let's use the same running example. Imagine we're writing some code that will make a recommendation for what someone will wear. Part of this reasoning will be receiving today's weather as a string, stored in `todaysWeather`. Our code will print what the user should wear.

If-Then

Let's start with the simple example: *if* it's raining, *then* the user should wear a raincoat and rainboots. This reasoning is shown in Figure 3.2.2.

#	IfThen-1.py	Output
1	`#Creates todaysWeather and sets it`	raincoat
2	`#equal to "raining"`	rainboots
3	`todaysWeather = "raining"`	Done!
4		
5	`#Checks if todaysWeather equals "raining"`	
6	`if todaysWeather == "raining":`	
7	`    #Prints "raincoat" if so`	
8	`    print("raincoat")`	
9	`    #Prints "rainboots" if so`	
10	`    print("rainboots")`	
11	`#Prints "Done!" when complete`	
12	`print("Done!")`	
13		

Figure 3.2.2

In line 3, we're creating the variable to store `todaysWeather`; if we were actually developing a program to do this, we would probably load this value from the Internet, but for testing we would give it a value manually to test the rest of our code. Then on line 6, we use the relational equality operator, ==, to check if `todaysWeather` is equal to "raining." Here, it is, so "raincoat" and "rainboots" are printed from lines 8 and 10. What if `todaysWeather` didn't equal "raining?"

In Figure 3.2.3, `todaysWeather` isn't "raining," so the conditional on line 6 is false, and so "raincoat" and "rainboots" don't get printed. Note the syntax in the code on line 6: we start with the word `if`, followed by a space. You might sometimes see it followed by an open parenthesis instead; either is fine. Sometimes parentheses will be used if the logical expression is more complex to make it easier to read, but for simple ones like this, parentheses aren't necessary. Either way, we then put in the

#	IfThen-2.py	Output
1	`#Creates todaysWeather and sets it`	Done!
2	`#equal to "cold"`	
3	`todaysWeather = "cold"`	
4		
5	`#Checks if todaysWeather equals "raining"`	
6	`if todaysWeather == "raining":`	
7	`    #Prints "raincoat" if so`	
8	`    print("raincoat")`	
9	`    #Prints "rainboots" if so`	
10	`    print("rainboots")`	
11	`#Prints "Done!" when complete`	
12	`print("Done!")`	
13		

Figure 3.2.3

condition we're checking. You'll notice we're using the equality (==) operator we introduced last unit; conditional statements are a big reason why logical operators were so important. Finally, we end the line with a colon: a colon is Python's sign that an indented code block is beginning.

The following line is indented, meaning that it is "under" or "controlled by" the conditional statement on line 6. The indented code is the "then" code; it's the code that runs *if* the conditional is true. Anything indented directly under that conditional statement will be controlled by that statement. Here, that's why line 10 is still controlled by the conditional statement; it's also indented under it. This is also why line 12 is *not* controlled by the conditional; it is *not* indented. So, even when the conditional on line 6 is false, line 12 still runs because it is not indented under line 6.

So, this is our fundamental `if` statement: the word `if`, some logical statement that resolves to `True` or `False`, a colon, and some indented code. Next, let's make it more complex.

If-Then-Else

Right now, the code just checks if it's raining, and recommends a raincoat and rainboots if so. Let's say that if it's not raining, we want to recommend a t-shirt and shorts. How do we do that?

We add an `else` block, and under the `else` block, we place the lines of code to print "t-shirt" and "shorts." In Figure 3.2.4, `todaysWeather` does not equal "raining," and so the conditional on line 6 is `False`, and the code it controls

#	IfThenElse-1.py	Output
1	`#Creates todaysWeather and sets it`	t-shirt
2	`#equal to "cold"`	shorts
3	`todaysWeather = "cold"`	Done!
4		
5	`#Checks if todaysWeather equals "raining"`	
6	`if todaysWeather == "raining":`	
7	`    print("raincoat")`	
8	`    print("rainboots")`	
9	`#If todaysWeather didn't equal "raining",`	
10	`#do the following`	
11	`else:`	
12	`    print("t-shirt")`	
13	`    print("shorts")`	
14	`print("Done!")`	
15		

Figure 3.2.4

does not run. The `else` code runs if the corresponding conditional statement was `False`, so here, the `else` code block (lines 12 and 13) runs and prints "t-shirt" and "shorts."

Notice the syntax in line 11, it's important to get this right: the keyword `else` must be at the same level of indentation as the original `if`; this is what tells Python which `else` corresponds to which `if` (which will matter later in this chapter). Logically, this makes sense: the `else` block code runs if the `if` block code did *not* run; if `else` was part of the `if` block code (that is, indented under it), it wouldn't run either! As before, `else` must also be followed by a colon, Python's sign that an indented code block is beginning.

What happens if the first `if` statement was true? Then the code under the `if` statement runs, as shown in Figure 3.2.5. The `else` code block only runs if the `if` code block did *not* run, and so here, the `else` code does not run. The final line that prints "Done!," however, lies outside either code block, so it runs in both Figure 3.2.4 and 3.2.5.

#	IfThenElse-2.py	Output
1	`#Creates todaysWeather and sets it`	raincoat
2	`#equal to "raining"`	rainboots
3	`todaysWeather = "raining"`	Done!
4		
5	`#Checks if todaysWeather equals "raining"`	
6	`if todaysWeather == "raining":`	
7	`    print("raincoat")`	
8	`    print("rainboots")`	
9	`#If todaysWeather didn't equal "raining",`	
10	`#do the following`	
11	`else:`	
12	`    print("t-shirt")`	
13	`    print("shorts")`	
14	`print("Done!")`	
15		

Figure 3.2.5

If-Then-Else-If-Else

Now let's throw our `else-if` statements into the mix. We'll start with just two checks: raining or cold.

The majority of the code in Figure 3.2.6 is the same as the code from Figure 3.2.4, but we've put something in between the `if` and the `else`. Line 9 is nearly identical to the original `if` in line 5, but it starts with a slightly different keyword: `elif`. This is Python's keyword for else-if. Other than the "el" at the beginning, it

#	IfThenElseIfElse-1.py	Output
1	`#Creates todaysWeather and sets it equal to "cold"`	long-sleeved shirt
2	`todaysWeather = "cold"`	scarf
3		Done!
4	`#Checks if todaysWeather equals "raining"`	
5	`if todaysWeather == "raining":`	
6	`    print("raincoat")`	
7	`    print("rainboots")`	
8	`#Otherwise, checks if todaysWeather equals "cold"`	
9	`elif todaysWeather == "cold":`	
10	`    print("long-sleeved shirt")`	
11	`    print("scarf")`	
12	`#If todaysWeather didn't equal "raining", do the following`	
13	`else:`	
14	`    print("t-shirt")`	
15	`    print("shorts")`	
16	`print("Done!")`	
17		

Figure 3.2.6

perfectly matches the original `if`. The only necessity for an `elif` statement is that it must come after an `if` and before any `else` at that level of indentation. And, as we said before, we can have more than one, as shown in Figure 3.2.7.

IfThenElseIfElse-2.py

```
1  #Creates todaysWeather and sets it equal to "windy"
2  todaysWeather = "windy"
3
4  #Checks if todaysWeather equals "raining"
5  if todaysWeather == "raining":
6      print("raincoat")
7      print("rainboots")
8  #Checks if todaysWeather equals "cold"
9  elif todaysWeather == "cold":
10     print("long-sleeved shirt")
11     print("scarf")
12 #Checks if todaysWeather equals "windy"
13 elif todaysWeather == "windy":
14     print("jacket")
15 #Checks if todaysWeather equals "snowy"
16 elif todaysWeather == "snowy":
17     print("snowboots")
18 #If todaysWeather didn't equal any of these, then...
19 else:
20     print("t-shirt")
21     print("shorts")
22 print("Done!")
23
```

Output

```
jacket
Done!
```

Figure 3.2.7

How does the code in Figure 3.2.7 run? First it creates `todaysWeather` on line 2 and gives it the value "windy." Then it checks on line 5 if `todaysWeather` equals "raining." It doesn't, so it skips the conditional's code block (lines 6 and 7). Then it checks the first `elif` on line 9. `todaysWeather` doesn't equal "cold," so it skips this code block (lines 10 and 11), too. Then it checks the second `elif` on line 13. `todaysWeather` *does* equal "windy," though, so it runs the contents of that code block (line 14) and prints "jacket." Now that one of the parts of the if-then-else-if-else block *has* run, it doesn't check the rest. From the start, it goes to the first `True` conditional it finds, runs its code block, and skips the rest. In this case, that means it skips the `elif` on line 16 and the `else` on line 19.

We can preview a later lesson to examine this; later, we'll talk about using operators along with conditionals. The code in Figure 3.2.8 approaches the same issue twice; note the difference.

IfThenElseIfElse-3.py

```
1  #Creates todaysWeather and sets it equal to "cold"
2  todaysWeather = "cold"
3
4  #Checks if todaysWeather equals "cold"
5  if todaysWeather == "cold":
6      print("scarf")
7  #Otherwise, checks if todaysWeather equals "windy" or "cold"
8  elif todaysWeather == "cold" or todaysWeather == "windy":
9      print("jacket")
10 print("Done! (First block)")
11 print()
12 #Checks if todaysWeather equals "cold"
13 if todaysWeather == "cold":
14     print("scarf")
15 #Checks if todaysWeather equals "windy" or "cold"
16 if todaysWeather == "cold" or todaysWeather == "windy":
17     print("jacket")
18 print("Done! (Second block)")
19
```

Output

```
scarf
Done! (First block)

scarf
jacket
Done! (Second block)
```

Figure 3.2.8

In Figure 3.2.8, we're trying to print "scarf" if it's cold and "jacket" if it's either cold *or* windy. In the first segment (lines 4 through 10), what happens? The conditional on line 5 triggers (or is `True`), so its code block runs and prints "scarf." The second condition (on line 8) is an `else-if`, so it doesn't trigger if the first one

runs. So, even though the condition in the `elif` statement is true, it doesn't run because the first `if` ran. It's an *else*-if; it only runs as an *alternative* to the preceding conditionals.

In the second segment (lines 13 through 17), we resolve this. Instead of making it an `elif`, we just make it another `if`. It's not indented under conditional beginning on line 13, so it runs either way. The second segment checks both conditionals because neither one is an `else-if` for the other. So, only use `else-if` if you want the conditional to be skipped if a previous part of the structure was true.

Common Errors

Finally, note that a common error in programming conditionals is to "orphan" the `else` or the `else-if` conditionals, as shown in Figure 3.2.9.

#	CommonErrors.py
1	`#Creates todaysWeather and sets it equal to "cold"`
2	`todaysWeather = "cold"`
3	
4	`#Checks if todaysWeather equals "cold"`
5	`if todaysWeather == "cold":`
6	`    print("scarf")`
7	`print("Done!")`
8	`#Otherwise, checks if todaysWeather equals "windy" or "cold"`
9	`elif todaysWeather == "cold" or todaysWeather == "windy":`
10	`    print("jacket")`
11	`print("Done!")`
12	`#This else is unattached because the block`
13	`#was already broken!`
14	`else:`
15	`    print("t-shirt")`
16	`print("Done!")`
17	

Output

```
File "CommonErrors.py", line 9
    elif todaysWeather ==
"cold" or todaysWeather ==
"windy":
       ^
SyntaxError: invalid syntax
```

Figure 3.2.9

The code in Figure 3.2.9 code gives us a `SyntaxError`. Why? Between the `elif` on line 9 and the `if` on line 5, there is a line (line 7) at the same level of indentation as the `if`. That breaks the code block of the `if` on line 5. The `elif` on line 9, however, has to follow an `if`; or more specifically, must immediately follow the indented code block that follows an `if`. As far as Python is concerned, the `elif` on line 9 is orphaned; it has no corresponding `if` because line 7 broke the code block of the `if` on line 5. The same occurs on line 11 before the `else`, which similarly must follow an `if` or `elif` block directly.

3. Conditionals and Operators

We've already seen that conditional statements usually use logical expressions built around logical operators to decide what to do. This isn't always the case; sometimes we might store the result of a logical expression in a boolean, and simply use that boolean inside the conditional instead of the expression itself. Either way, though, conditionals are often used with logical expressions in some way.

So far, we've only looked at the equality operator. However, conditionals are used in other ways as well.

Relational and Mathematical Operators

In addition to the equality operator (whether used mathematically or more generally with strings), it is common to use the other relational and mathematical operators with conditionals. We've covered one example several times: comparing bank balances. That is a relational expression that would generate `True` or `False` based on whether one number is greater than another.

We can embed other mathematical operators within these statements as well. For example, if we wanted to compare a person's bank balance to a purchase price plus its sales tax, we could perform that mathematical operation right there within

the conditional rather than performing it separately and storing it for later comparison in a conditional.

Boolean Functions

We've mentioned functions a few times now; we'll get to them more later, but for now, we know that functions are like custom, more complex operators that take some input and return some output. For example, we've mentioned before that some languages (such as Python) have a `len()` function before, which takes as input something with a length (like a string or a list of items) and produces as output the length of that input (like 12 when the input is "Hello, world").

Functions can return booleans as well, which means we can use functions in conditionals. For example, we could have a function that takes a filename and checks if the file exists. So, our conditional would basically say, "if this file exists, then…" A lot of the complexity and power around conditionals comes when we start writing custom functions to return booleans.

Boolean Operators

Finally, boolean operators allow us to take other operators and functions and combine them into far more complex conditionals. We can check multiple different conditions, or multiple combinations of conditions. We could have very complex statements, although in practice we generally want to break complex conditionals down into multiple, simpler, nested conditionals.

Returning to our weather and clothing example, we would likely have certain articles of clothing that are worn in multiple kinds of weather. We might wear a jacket in either cold or windy weather, for example. Boolean operators would let us check either of those conditions within a single line: if cold or windy, then wear a jacket.

4. Conditionals and Operators

Let's take a look at some of the ways we can use conditionals along with operators in Python. We'll keep these examples simple: mostly `if-then-else` statements and few `elif` statements, but note that these can be combined with the advanced structures covered above.

Relational Operators

We've covered before the simple way we can use relational operators in conditionals, but let's look again. What if we wanted to check to see if a buyer has enough funds on a card to make a purchase?

The greater-than-or-equal-to operator returns `True` if the first number is greater than or equal to the second, `False` if it is not. Since it returns `True` or `False`, we can use it in conditional statement in line 8 of Figure 3.2.10. Here, we see that

#	RelationalOperators.py	Output
1	`#Creates balance and sets it equal to 20.0`	Purchase possible!
2	`balance = 20.0`	Done!
3	`#Creates purchasePrice and sets it equal to 19.0`	
4	`purchasePrice = 19.0`	
5		
6	`#Checks if balance is greater than or equal`	
7	`#to purchasePrice`	
8	`if balance >= purchasePrice:`	
9	`    print("Purchase possible!")`	
10	`else:`	
11	`    print("Purchase not possible!")`	
12	`print("Done!")`	
13		

Figure 3.2.10

`balance` is greater than `purchasePrice`, so the operator returns `True`, and the code block under the `if` statement runs.

Relational and Mathematical Operators

On their own, mathematical operators return other numbers, so they can't be used on their own in a conditional. The statement "if 3 + 5, then…" doesn't make sense because 3 + 5 returns 8, not a `True` or `False`.

However, we can use mathematical operators along with relational operators. Imagine in our above example if we wanted to compare the balance to the purchase price with sales tax. How would we do that?

In Figure 3.2.11, we've created a variable `salesTax` and given it the value 1.08, which mathematically is the multiplier for an 8% sales tax. Then, in the conditional, we multiply `purchasePrice` by `salesTax`. The computer automatically does this before checking the relational operator because of its internal order of operations. In this way, we can use mathematical operators within conditional statements.

#	RelationalandMathematicalOperators.py	Output
1	`#Creates balance and sets it equal to 20.0`	`Purchase not possible!`
2	`balance = 20.0`	`Done!`
3	`#Creates purchasePrice and sets it equal to 19.0`	
4	`purchasePrice = 19.0`	
5	`#Creates salesTax and sets it equal to 1.08`	
6	`salesTax = 1.08`	
7		
8	`#Checks if balance is greater than or equal`	
9	`#to purchasePrice times salesTax`	
10	`if balance >= purchasePrice * salesTax:`	
11	`    print("Purchase possible!")`	
12	`else:`	
13	`    print("Purchase not possible!")`	
14	`print("Done!")`	
15		

Figure 3.2.11

Set Membership Operators

You might remember that one of the things that makes Python unique is easy access to functions that check if something is a member of another set. So, where many languages would have this next example as an example of a boolean function, in Python it's a unique kind of operator.

Figure 3.2.12 shows a more complicated check for weather and clothing. Instead of checking each type of weather one-by-one and printing the corresponding articles of clothing, we could instead create lists of the weather conditions for each piece of clothing. On line 2, we see one of them: `jacketWeather` is a list of types of weather that suggest the user should wear a jacket: cold, windy, raining, and snowing.

#	SetMembershipOperators.py	Output
1	`#Creates the list jacketWeather`	`jacket`
2	`jacketWeather = ["cold", "windy", "raining", "snowing"]`	`Done!`
3	`#Creates todaysWeather and sets it equal to "raining"`	
4	`todaysWeather = "raining"`	
5		
6	`#Checks if todaysWeather is in jacketWeather`	
7	`if todaysWeather in jacketWeather:`	
8	`    print("jacket")`	
9	`print("Done!")`	
10		

Figure 3.2.12

The conditional on line 7 checks to see if `todaysWeather` is one of the items in `jacketWeather`, and if so, prints jacket on line 8. If we wanted to add another weather condition to the list of conditions that dictate wearing a jacket, we just have to add it to the list `jacketWeather`. Similarly, we could have lists like this for jackets, scarves, t-shirts, etc., and easily check them. Note that if this is confusing, don't worry: we haven't gotten to lists yet. Python's syntax is accessible enough that you might understand this just based on the natural meaning of the word "in," but don't worry if that's not the case. We'll talk more about this later.

Boolean Functions

If a function returns a boolean, then we can use it in a conditional statement. For example, in Python, there is a function (well, technically a method, but don't worry about the difference for now) called `isdigit()` that returns `True` if the string represents a number, `False` if it does not. Figure 3.2.13 shows this in action.

#	BooleanFunctions.py	Output
1	`myNumericString = "12345"`	The first string is numerical.
2	`myNonNumericString = "ABCDE"`	The second string is non-numerical.
3		
4	`#Checks if myNumericString is purely numeric`	
5	`if myNumericString.isdigit():`	
6	`    print("The first string is numerical.")`	
7	`else:`	
8	`    print("The first string is non-numerical.")`	
9	`#Checks if myNonNumericString is purely numeric`	
10	`if myNonNumericString.isdigit():`	
11	`    print("The second string is numerical.")`	
12	`else:`	
13	`    print("The second string is non-numerical.")`	
14		

Figure 3.2.13

You might initially be confused about why `isdigit()` is after the variable name (`myNumericString.isdigit()`) instead of the way we've seen functions before (`isdigit(myNumericString)`). The reason for this is that it's a method, not a function—but again, we'll get to the difference later. For now, just know that `myNumericString.isdigit()` returns `True` if `myNumericString` is a number, `False` if it is not; and, any string can use `.isdigit()` the same way.

So, within the conditional on line 5 of Figure 3.2.13, we have `myNumericString.isdigit()`. "12345" is all numbers, so the conditional on line 5 is `True`, and so it prints on line 6 that the string is numerical. The function (well, method) returns `True`, so the conditional is true, so the first code block runs. In lines 10 through 13, the opposite happens: "ABCDE" is not numeric, so the conditional is `False`, so the second code block (line 13, after the `else`) runs instead. If you're curious, there are similar methods for checking if a string is all letters (`.isalpha()`), all letters or numbers (`.isalnum()`), all lowercase (`.islower()`), all uppercase (`.isupper()`), or all whitespace (`.isspace()`).

Boolean Operators

Finally, our boolean operators—`and`, `or`, and `not`—can be used to combine any of these logical expressions together. Let's look at this with two examples: a simple one from our weather example, and a complex one from our purchasing example.

Figure 3.2.14 is a simplified version of one of our previous examples of `elif`, this time using just one `if`. Here, the conditional checks if `todaysWeather` is either cold *or* windy. Note the syntax here: to check if one *or* the other is true, we simply put the word `or` between the two logical expressions. Here, the first one

evaluates to `True` and the second one evaluates to `False`, and `True or False` resolves to `True`: it's true that the weather is either cold or windy.

BooleanOperators-1.py

```
1 todaysWeather = "cold"
2
3 #Checks if todaysWeather equals "windy" or "cold"
4 if todaysWeather == "cold" or todaysWeather == "windy":
5     print("jacket")
6 print("Done!")
7
```

Output

```
jacket
Done!
```

Figure 3.2.14

In Figure 3.2.15, had we used `and` instead of `or`, then the conditional would have been `False`. It would not be true that `todaysWeather` equals both "cold" and "windy" (and in fact, the way we've written this, that would be impossible since `"cold" == "windy"` itself is `False`).

BooleanOperators-2.py

```
1 todaysWeather = "cold"
2
3 #Checks if todaysWeather equals "windy" and "cold"
4 if todaysWeather == "cold" and todaysWeather == "windy":
5     print("jacket")
6 print("Done!")
7
```

Output

```
Done!
```

Figure 3.2.15

Now let's try a more complex example. Previously, we've mentioned in the context of our purchasing code the idea of checking several conditions: Is the balance sufficient? Is the cardholder the person making the purchase? Is the vendor a trusted vendor?

Figure 3.2.16 shows a complex chunk of code that tests this. We're putting together three logical expressions in one conditional with a pair of `and` operators. We check if the balance is greater than the purchase price plus sales tax, *and* the cardholder is the current customer, *and* the vendor is a trusted vendor. Only if all three of those things are `True` do we approve the purchase.

BooleanOperators-3.py

```
1  #Sets up general information about the balance, tax, cardholder, and
2  #trusted vendors. Generally, the information in these lines would be
3  #sent into our program, not created here. Here, we create it manually
4  #to test out our code
5  balance = 20.0
6  salesTax = 1.08
7  cardholderName = "David Joyner"
8  trustedVendors = ["Maria's", "Bob's", "Vrushali's", "Ling's", "Tia's"]
9
10 purchasePrice = 19.0
11 customerName = "David Joyner"
12 vendor = "Vrushali's"
13
14 #This long conditional checks whether the balance is greater than the
15 #total price, whether the cardholder is also the customer, and whether
16 #the vendor is trusted.
17 if balance > purchasePrice * salesTax and cardholderName == \
18         customerName and vendor in trustedVendors:
19     print("Purchase approved!")
20 else:
21     print("Purchase not approved!")
22 print("Done!")
23
```

Output

```
Purchase not
approved!
Done!
```

Figure 3.2.16

This can get even more complicated. We might have logical expressions with boolean operators within the larger expression. Observe Figure 3.2.17 and note how it runs.

```
#  BooleanOperators-4.py
1  #Sets up general information about the balance, tax, cardholder, and
2  #trusted vendors. Generally, the information in these lines would be
3  #sent into our program, not created here. Here, we create it manually
4  #to test out our code
5  balance = 20.0
6  salesTax = 1.08
7  cardholderName = "David Joyner"
8  trustedVendors = ["Maria's", "Bob's", "Vrushali's", "Ling's", "Tia's"]
9
10 purchasePrice = 19.0
11 customerName = "David Joyner"
12 vendor = "Vrushali's"
13 overdraftProtection = True
14
15 #This long conditional checks whether the balance is greater than the
16 #total price or overdraft protection is available, whether the cardholder
17 #is also the customer, and whether the vendor is trusted.
18 if (balance > purchasePrice * salesTax or overdraftProtection) \
19         and cardholderName == customerName and vendor in trustedVendors:
20     print("Purchase approved!")
21 else:
22     print("Purchase not approved!")
23 print("Done!")
24
```

```
Output
Purchase not approved!
Done!
```

Figure 3.2.17

First, a syntactical note. The conditional statement on lines 18 and 19 looks weird, doesn't it? It ends in a slash, the next line is double-indented, and the colon isn't until after line 19. In Python, this is how we tell the computer, "Interpret these two lines as one line." Breaking the code between two lines makes it more readable for us as humans, but the computer needs to see it as all one line. So, this line lets us do both. The slash says, "Copy the next line, and put it where this slash is."

Anyway, on line 13 we've added an additional variable: `overdraftProtection`. Overdraft protection (for this example, anyway) allows the customer to charge more than their balance and pay it off later. If it's available, then it doesn't matter if the balance is greater than the purchase price. So, here we have a nested `or` within our longer `and` statements. The computer should evaluate whether the balance is sufficient *or* overdraft protection is available. If either is `True`, then the first part of the condition is `True`.

Note that we put parentheses around this `or` expression on line 18 to force the computer to evaluate it first. In this case, we didn't actually have to. The computer will automatically evaluate logical operators from left to right. However, it's always good to use parentheses for human readability, as well as for safety. For example, if we had put the `or` expression at the end without parentheses, it would have changed the results. So, it's always good to use parentheses to be clear on the order in which things should be evaluated.

As we said before, we could take these principles and combine them with the complex if-then-else-if-else statement structures from earlier.

5. Nested Conditionals

In our example of evaluating whether a purchase would be approved, there was a weakness. We evaluated whether or not multiple conditions were all true, and if they all were, then we approved the purchase; if not, we rejected it. However, this doesn't tell us *why* the purchase was rejected. We know that if it was approved, all the conditions were true, but if it was rejected, we don't know which part caused the rejection. We can resolve this by using a more complex structure: a nested conditional.

Nested Conditional
A conditional statement that is itself controlled by another conditional statement. More simply, an `if-then` statement within another `if-then` statement.

Nested Conditionals

A **nested conditional** isn't a special type of control structure like `else-if` or `else`. Rather, it's just one way of applying an existing control structure. If a conditional is true, it runs the code block that the conditional controls. That code block can be anything we want it to be, which means that code block can *itself* contain conditionals.

Our original reasoning was, "If the balance is sufficient and the customer is the cardholder and the vendor is trusted, approve the purchase; if not, reject it." We can revise this reasoning a bit to allow us to make decisions based on those individual conditions. This is a bit difficult to explain in paragraph form, so if this is confusing, don't worry; we'll use a flowchart in a moment, then code. We might say instead: if the balance is sufficient, check the cardholder; else, reject because of insufficient balance. If the cardholder is the customer, check the vendor; else, reject because of invalid cardholder. If the vendor is trusted, then accept the purchase; else, reject because of untrusted vendor.

Nested Conditionals in a Flowchart

If that was confusing, don't worry: this kind of branching reasoning is tough to explain in linear text. Instead, let's take a look at two flowchart views of this.

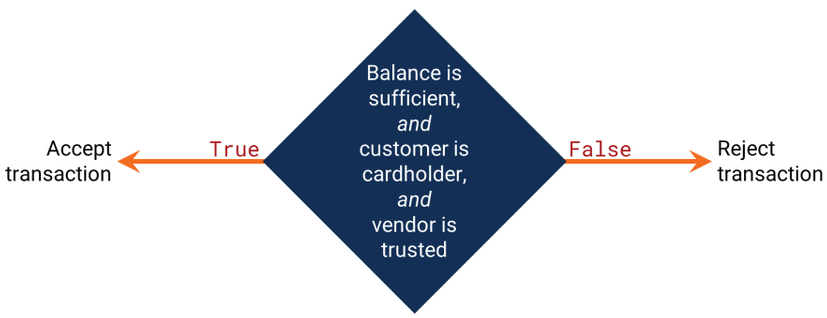


Figure 3.2.18

Figure 3.2.18 was our original approach: one big decision with multiple conditions. If all are true, we go one way; if one is false, we go a different way.

Figure 3.2.19 is our new approach. Each individual decision is separate. If one is `True`, we go on to the next decision; if one is `False`, we go to the dedicated output for *that* decision. Each conditional governs whether we move on to the next conditional or just exit. In some ways, this is similar to the `else-if`; however, where an `else-if` only runs if the previous `if` was `False`, a nested `if` only runs if the previous `if` was `True` because it's part of the code block that only runs if the `if` statement was `True`.

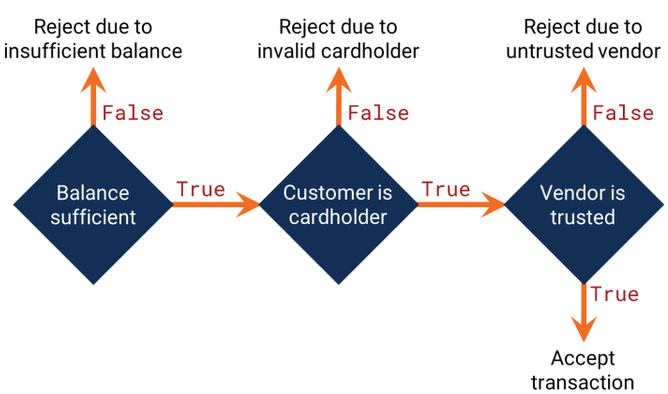


Figure 3.2.19

6. Nested Conditionals in Python

Let's take a look at what our previous purchase validation code would look like with nested conditionals.

Ifs Within Ifs

Note that while one major benefit of nested conditionals is that we can take care of more combinations of conditions, another benefit is that in many ways, this code is more readable. Take a look at Figure 3.2.20.

IfsWithinIfs.py

```
1  balance = 20.0
2  salesTax = 1.08
3  cardholderName = "David Joyner"
4  trustedVendors = ["Maria's", "Bob's", "Vrushali's", "Ling's", "Tia's"]
5
6  purchasePrice = 19.0
7  customerName = "David Joyner"
8  vendor = "Freddy's"
9  overdraftProtection = True
10
11 #This nested conditional checks whether the balance is greater than
12 #the total price or overdraft protection is available, then whether
13 #the cardholder is the customer, and then whether the vendor is trusted.
14 if balance > purchasePrice * salesTax or overdraftProtection:
15     if cardholderName == customerName:
16         if vendor in trustedVendors:
17             print("Purchase approved!")
18         else:
19             print("Purchase not approved; untrusted vendor.")
20     else:
21         print("Purchase not approved; invalid customer.")
22 else:
23     print("Purchase not approved; no funds or overdraft protection.")
24 print("Done!")
25
```

Output

```
Purchase not approved; untrusted vendor.
Done!
```

Figure 3.2.20

Remember how we had to break one line of code between two lines just for readability in Figure 3.2.16? With these nested conditionals, we no longer have to do that. Instead, we have three short, simple conditional statements, one under the other on lines 14, 15, and 16. Each is indented under the previous one, meaning each only runs if the previous one also ran. That means the purchase is only approved if the first conditional *and* the second conditional *and* the third conditional are all `True`, which makes it functionally equivalent to our original statement.

However, with this structure, each individual conditional can have its own dedicated `else` block, meaning we can print exactly why the purchase failed. On line 8, I've changed the vendor to an untrusted vendor, and so the code runs until it checks the third conditional on line 16. This condition is `False`, so it jumps to *this* conditional's `else` block (line 19) and says the vendor was untrusted. This tells us a lot more than our earlier code: it tells us the vendor was untrusted, and the fact that it reached this line also tells us that both the balance was sufficient and the cardholder was valid because this conditional was controlled by those previous conditionals.

Ifs Within Elses

This nesting applies on both sides of the structure as well. We can write code that is functionally equivalent to the above with a completely different structure by nesting our conditionals in the `else` blocks instead. Check it out in Figure 3.2.21.

This code performs exactly the same, but all the nesting is inside the `else` portions of the conditional. That's because instead of checking whether the purchase *passes* each requirement on lines 15, 18, and 21, it checks whether the purchase *fails* each requirement; notice how the logical expressions have changed compared to Figure 3.2.20. If the purchase fails a condition, it prints that it's failed and why; if not, it moves on to the next check. This is like saying that a purchase is approved if *none* of the checks *fail*, rather than if *all* of them *pass*: these mean the same thing, but they're organized a little differently.

#	IfsWithinElses.py
1	`balance = 20.0`
2	`salesTax = 1.08`
3	`cardholderName = "David Joyner"`
4	`trustedVendors = ["Maria's", "Bob's", "Vrushali's", "Ling's", "Tia's"]`
5	
6	`purchasePrice = 19.0`
7	`customerName = "David Joyner"`
8	`vendor = "Freddy's"`
9	`overdraftProtection = True`
10	
11	`#This nested conditional checks whether the balance is less than or`
12	`#equal to the total price and overdraft protection is not available;`
13	`#otherwise, whether the cardholder is not also the customer; and`
14	`#otherwise, whether the vendor is not trusted.`
15	`if balance <= purchasePrice * salesTax and not overdraftProtection:`
16	`    print("Purchase not approved; no funds or overdraft protection.")`
17	`else:`
18	`    if not cardholderName == customerName:`
19	`        print("Purchase not approved; invalid customer.")`
20	`    else:`
21	`        if not vendor in trustedVendors:`
22	`            print("Purchase not approved; untrusted vendor.")`
23	`        else:`
24	`            print("Purchase approved!")`
25	`print("Done!")`
26	

Output
Purchase not approved; untrusted vendor. Done!

Figure 3.2.21

7. Conditionals and Scope

We used conditionals as our example for scope in the previous chapter, so you've already seen a bit about how these interact. Now that you know what conditionals are, however, let's revisit this. In Python, the scope of a variable starts when it is created, and ends when one of a number of terminations happen. For now, the only termination you need to know is the program ending: when the program ends and closes, the computer forgets the variables that were created while it was running. There are other times when the scope of a variable ends or is suspended, but for now, you only need to worry about the scope ending when the code ends.

Accessing Variables within Conditionals

So, let's return to our earlier example of scope in a conditional now that we know what conditionals are, shown here in Figure 3.2.22.

Recall that this code creates the variable `result` outside the conditional. The scope of `result` is from line 4 until the program stops running. So, when line 6 inside the conditional comes up to change the value of `result`, `result` is still available. This line is within the scope of this variable. Within what we know now (and what we'll learn until we get to functions), the scope of a variable is from the point at which it is created until the end of the program.

#	AccessingVariableswithinConditionals.py
1	`myNum1 = 1`
2	`myNum2 = 2`
3	`#Creates an initial value for result`
4	`result = "Result was unchanged."`
5	`if myNum1 < myNum2:`
6	`    result = "myNum2 is greater than myNum1!"`
7	`print(result)`
8	`print("Execution complete!")`
9	

Output
myNum2 is greater than myNum1! Execution complete!

Figure 3.2.22

Creating Variables within Conditionals

Note, however, that there is a risk. Imagine if you create a variable within an `if` statement's code block, but then that code block doesn't run. That means the variable was never created, and so if you try to access the variable outside the code block, your code will crash. You can see this happening in Figure 3.2.23.

#	CreatingVariableswithinConditionals-1.py	Output
1	`myNum1 = 2`	`Traceback (most recent call last):`
2	`myNum2 = 1`	`File "...", line 5`
3	`if myNum1 < myNum2:`	`  print(result)`
4	`  result = "myNum2 is greater than myNum1!"`	`NameError: name 'result' is not defined`
5	`print(result)`	
6	`print("Execution complete!")`	
7		

Figure 3.2.23

If we set the values of `myNum1` and `myNum2` such that the conditional doesn't run and we don't create `result` outside the conditional, then our code encounters an error. The scope of `result` is from the moment it is created until the end of the program, but if line 4 never runs, it is never created; so, when the computer tries to print it in line 5, it is out of scope.

The best way to resolve this, in my opinion at least, is to never create variables *inside* a conditional that will need to be accessed *outside* the conditional. In fact, most languages that I know of won't even *let* you do what Python is letting you do here; Java, for example, defines a variable's scope as the current code block, so once the conditional is concluded, the program forgets `result` altogether even if the code block ran. I, personally, recommend following that convention. There is another way around this issue, though.

If we really want to create a variable inside a conditional to use outside of it, the least we can do is create it within each branch of the conditional, including an `else`. The if-then-else shown in Figure 3.2.24 guarantees that either the `if` code block *or* the `else` code block will run. Since `result` is created in both, we guarantee `result` will have been created when we reach line 7. Again, I personally recommend that you avoid creating variables in conditionals that you need to access outside, but if you do, you can use this to guarantee they're created.

#	CreatingVariableswithinConditionals-2.py	Output
1	`myNum1 = 1`	`myNum2 is greater than myNum1!`
2	`myNum2 = 2`	`Execution complete!`
3	`#Checks if myNum1 is less than myNum2`	
4	`if myNum1 < myNum2:`	
5	`  result = "myNum2 is greater than myNum1!"`	
6	`else:`	
7	`  result = "myNum2 is not greater than myNum1!"`	
8	`print(result)`	
9	`print("Execution complete!")`	
10		

Figure 3.2.24

8. Conditionals and Turtles

Now that we have conditionals at our disposal, we can really start to create a way for the user to control turtles with their input alone. Let's create an interface with two possible commands the user can give: turn and forward. Let's also assume the user is going to enter two such commands, so we'll run the same code twice.

Turn and Forward

So what does this look like in code? It's actually surprisingly short, as seen in TurnandForward.py.

We let the user input two commands, and so we've copied all the code twice (and left the comments out the second time); later, we'll learn how to do this more efficiently. For now, it means we're mostly interested in lines 4 through 17; line 1 just sets up our turtle, and lines 19 through 27 just repeat lines 4 through 17. So, let's walk through this code piece by piece.

First, line 4 gets the command from the user as a string and stores it in command. Then, line 7 runs a simple logical expression for string equality to see if the command the user put in was "turn." If so, then line 9 asks the user to put in an angle, and line 11 executes that turn.

If the user didn't input "turn" as the command, then the code skips lines 9 and 11 and checks in line 13 if the command was "forward." If so, line 15 asks the user to input a distance, and line 17 executes the move forward. Then, lines 19 through 27 repeat the process.

Notice a couple of things here. First, notice that this code reuses `command` in lines 19 through 27. There's no need to create a second variable to store the command because the previous command will be overwritten. The same goes for `angle` and `distance`; if the user enters two forward commands, then `distance` in the first one will be overwritten the second time.

You can extend this code in a lot of ways: you could add more commands, for example. You can find the list of commands available for the turtle graphics package at https://docs.python.org/3.5/library/turtle.html.

Turn, Forward, or Error

However, with the current design of this code, what happens if the user enters invalid commands? Try running TurnandForward.py, but enter words like "up" and "down" instead of "turn" and "forward".

What happens? Nothing! Why is that? Well, the words "up" and "down" (or any other words besides "turn" and "forward") don't cause any of the conditionals to resolve to `True`, so they're skipped. In part, that's a good thing: it means our code doesn't crash if we enter invalid commands (as it will right now if we enter strings for `angle` or `distance`, but we'll handle that later). But it's also a bad thing: the user doesn't know why the code doesn't do anything! How do we repair this?

Simple! We just add an `else` at the end of each conditional that prints that the command was invalid, as shown in TurnForwardorError.py. For usability, it's good to give the user feedback on what exactly they *could* have done as well, so we print the commands the code could have understood, "turn" and "forward". So, this is one way we can improve this code.

There remains lots of room for improvement in this code, of course. For one, why only execute two commands? With these two commands alone, the user can only ever draw a single line in one direction. Would it be better for the user to be able to execute as many commands as they want until they choose to exit? We'll cover that when we come to loops next lesson. Second, why only these primitive little options like "forward" and "turn?" Wouldn't it be nice if we could have singular commands for "octagon" or "star?" We'll cover that when we come to functions. Third, the code crashes if the user doesn't put in a valid number for "distance" or "angle." Wouldn't it be better if it gave them feedback the way it does if they enter an invalid angle? We'll cover that when we come to error handling. By the end of this unit, you'll have a script that can run any number of user-inputted commands, and the ability to write custom commands like "octagon" yourself!

CHAPTER

3.3

Loops

1. What Is a Loop?

What is a **loop**? A loop is a control structure that repeats some lines of code until a certain condition is met. The important word there is: repeats. A loop repeats lines of code. This is extremely valuable. Most trivially, it means that we don't have to just copy and paste chunks of code if we want to do something more than once: but that's only a small part of the power of loops. Their real power is their ability to repeat code a dynamic number of times based on some conditions.

For example, imagine if you were writing some code to change the names of all the files in a folder. The code for changing the name of each file is probably pretty similar, and you just want to repeat it for each file in the folder. With loops, it doesn't matter if there are two files in the folder or two million: you could write code that would repeat the renaming function for every file in the folder.

Generally speaking, there are two common kinds of loops: `for` loops and `while` loops. They do have some variations within each type as well, but these are the most common categories of loops. As we'll describe later, anything you can do with a `for` loop, you can also do with a `while` loop; however, each one is better suited to certain kinds of tasks.

For Loops

A `for` loop repeats some code a certain number of times: "for 7 times, do this." For example, imagine you have an exercise routine that has you do ten push-ups, ten sit-ups, and five pull-ups. We could describe this in terms of three loops: the first would run ten times, and with each **iteration** of the loop, you run the function `pushup()`. Or, imagine you were shopping, and there were seven items on your shopping list. So, seven times, you (a) read the next item on the list, (b) walk to that item in the store, and (c) put the item in your basket. Those three tasks would run seven times; or, in other words, you'd "loop over" them seven times.

Sometimes, we'll know exactly how many times a loop should run in advance. For example, imagine you were writing the code for a blog, and you knew you wanted to show ten posts on the front page. You might run a loop ten times that grabs the next posts and displays it. Other times, the desired number of iterations (i.e., repetitions) might vary. In that same example, imagine we want to show *every* post in the blog on one page. In that case, we'd run the same loop, where the number of times we run it equals the number of posts in the blog. As more posts are written, the loop will need to run more times.

Some languages, like Python, supply a special kind of `for` loop called a `for-each` loop. `for-each` loops come from the observation that a large number of times that we use `for` loops, we're looping over items in a list, like blog posts in a blog or items on a shopping list. These loops generally always take the same form: (a) we find the total number of items in the list and run the loop that many times, and (b) at the beginning of each iteration, we grab the next item on the list and store it in some variable. A `for-each` loop just simplifies this. Instead of saying, "for seven times, read the next item, go get it, and put it in my basket", it's like saying, "for each item, go get it and put it in my basket." Functionally these are the same, but just a little easier to write and more natural to think about.

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Describe the concept of loops and analyze the usage of `for` loops and `while` loops in different situations;
- Write different types of loops, including `for` loops with known and unknown ranges, `for-each` loops, `while` loops, `infinite` loops, and `nested` loops;
- Use `continue`, `break`, and `pass` keywords with loops and define the scope of a variable in the context of a loop;
- Write a user-facing script for controlling the turtle graphics module using `while` loops and `for` loops.

Loop
A programming control structure that executes a segment of code multiple times.

For Loop
A loop control structure that runs a block of code a predetermined number of times.

Iteration
A single execution of a repeated task or block of code.

While Loops

While Loop
A loop control structure that runs a block of code until a certain logical expression is satisfied.

`for` loops run with some advanced knowledge about how many times the loop will run: "for 7 times, do this." `while` loops, on the other hand, run *while* something remains true. For example:

- While the screw is still loose, keep screwing it in.
- While there is food on the plate, take a bite.
- While there are still emails in your inbox, read the next one.
- While there are still items on your grocery list, get the next one.

We mentioned earlier that anything we express in a `for` loop can also be expressed in a `while` loop, and we see that with this last example. We can think of this as, "do this for each item on my list," or we can think of this as, "do this while there are still items on my list." It's also technically true that pretty much every `while` loop can be rewritten as a `for` loop, but in practice, usually this is far more of a hack than just a different way of approaching the problem.

Some languages, like Java, also have a special kind of `while` loop called a `do-while` loop. We can think of the standard `while` loop above as a "`while-do`" loop: while a condition is true, do something. A `do-while` loop is identical, except that it guarantees the "something" will be run at least once: it does it before checking the condition the first time.

Notice also how `while` loops are heavily dependent on logical expressions, just like conditionals were. Just as we filled a conditional with a logical expression to decide whether to run its code block, so also we fill a `while` loop with a logical expression to decide whether to *keep* repeating its code block. In many ways, you can think of a `while` loop as a repeated conditional: it repeats *while* the condition is true, rather than running once *if* the condition is true.

2. Traditional For Loops in Python

Let's start with the traditional `for` loop. In practice, the traditional `for` loop is actually used less in Python than the `for-each` loop; however, in computing as a whole, the traditional `for` loop is probably more fundamental. In fact, technically Python does not even have a traditional `for` loop: even the `for` loops are technically `for-each` loops.

For Loops with Known Ranges

Let's start with a loop where we know how many times we want to run it. Imagine we want to write a loop that will print the numbers from 1 to 10. How do we do that?

Figure 3.3.1 shows the syntax for this simple loop. Let's walk through it part by part. First, just like we started with `if` to do a conditional, we start with `for` to do a `for` loop on line 2. Then, we define a variable name. Generally, self-documenting code is the goal, but we use loops so commonly that it's not uncommon to use a single character—usually `i` or `n`—here. We call this the **loop control variable**.

Loop Control Variable
A variable whose value is the number of times a loop has run. It is used to check if the loop should keep running (e.g. if it has run as many times as it's supposed to).

#	ForLoopswithKnownRanges-1.py	Output
1	`#Loop this for the numbers 1 through 10`	1
2	`for i in range(1, 11):`	2
3	`    #Print the current number`	3
4	`    print(i)`	4
5		5
6		6
7		7
8		8
9		9
10		10
11		
12		
13		

Figure 3.3.1

This is a variable we'll be able to access inside the loop to see how many times the loop has run so far.

Next, we see `range(1, 11)`. This is a little confusing in Python, so I won't get into the nitty-gritty of how exactly it works. All you need to know is that in this kind of loop, the first time it runs, the variable (in this case, `i`) will take the first number specified in `range()` as its value (in this case, the 1 in `range(1, 11)`). Then, each time the loop runs, the variable will increase by 1. The loop will stop running when the variable equals the second number (in this case, 11). Note that when the variable equals that number, it will *not* run the loop again; it runs *until* the variable *equals* the second number, not until the variable *exceeds* the second number.

`range()`
Takes as input two variables, a first number and a last number, and provides the list of numbers for a for loop to iterate over in a for loop.

Then, the `for` loop ends with a colon, Python's indicator that some indented code block is coming. Then, each line of code indented under the loop statement will be run, in order, each time the loop runs. In this case, these are lines 3 and 4.

So, let's trace this one. When the computer hits `for i` on line 2, it creates a variable `i`. Then, it assigns that variable to the first number in the range; in this case, it assigns it to 1. It then checks to see if the number equals the second number. 1 doesn't equal 11, so it runs the code and prints the current value of `i`, 1. When it's done with the code block, it jumps back to the top and increments `i`. So, now `i` is 2. It checks to see if `i` equals 11. It doesn't, so it runs it again, then jumps to the top and increments `i`. It continues like this until `i` equals 10. At this point, it prints `i` (10), increments `i` (to 11), and checks if `i` equals 11. It does now, so it stops running the loop.

One interesting (but non-essential) note: if you modify `i` within the body of the loop, the modified value will be used for the rest of *that* iteration of the loop; however, when that iteration ends, it restores the previous value of `i`. The *contents* of the loop can't control the loop itself. You likely won't ever encounter this, but it's an interesting idiosyncrasy of the Python language; in Java and other languages, the contents of the loop *can* control the loop itself.

#	ForLoopswithKnownRanges-2.py	Output
1	`#Loop this for the numbers 1 through 10`	Before addition: 1
2	`for i in range(1, 11):`	After addition: 2
3	`    #Print i with a label`	Before addition: 2
4	`    print("Before addition:", i)`	After addition: 3
5	`    #Adds 1 to i`	Before addition: 3
6	`    i += 1`	After addition: 4
7	`    #Prints i with a label`	Before addition: 4
8	`    print("After addition:", i)`	After addition: 5
9		Before addition: 5
10		After addition: 6
11		Before addition: 6
12		After addition: 7
13		Before addition: 7
14		After addition: 8
15		Before addition: 8
16		After addition: 9
17		Before addition: 9
18		After addition: 10
19		Before addition: 10
20		After addition: 11
21		
22		
23		
24		
25		
26		

Figure 3.3.2

For Loops with Unknown Ranges

One common use of the `for` loop is to average numbers. So, let's try that out. Let's write some code in Figure 3.3.3 that will have the user enter 10 numbers, and then print the average. We'll start with a known range, but then we'll look at an unknown range.

First, we need to create `sum` on line 2, outside the loop with a value of 0 since we haven't started adding numbers to it. Interesting thought experiment, though: why do we have to create it outside the loop? Couldn't we create it inside like declaring variables inside our conditionals? The reason is that every time the loop runs, we

need to set `sum` equal to the previous `sum` plus the new number; if `sum` didn't exist before, there is no previous `sum`, and so the program crashes because we're not in `sum`'s scope the first time it runs!

#	ForLoopswithUnknownRanges-1.py	Output
1	`#Creates sum with the value 0`	`Enter number #1: 91`
2	`sum = 0`	`Enter number #2: 92`
3	`#Loop 10 times`	`Enter number #3: 93`
4	`for i in range(1, 11):`	`Enter number #4: 94`
5	`    #Gets the user's number`	`Enter number #5: 95`
6	`    nextNumber = int(input("Enter number #" + str(i) + ": "))`	`Enter number #6: 96`
7	`    #Add the inputted number to the sum`	`Enter number #7: 97`
8	`    sum += nextNumber`	`Enter number #8: 98`
9	`#Print the sum over 10`	`Enter number #9: 99`
10	`print(sum / 10)`	`Enter number #10: 100`
11		`95.5`
12		
13		
14		

Figure 3.3.3

Then, we run the same loop we ran previously starting on line 4, 10 times. Each time, we get the next number from the user, store it in `nextNumber`, and then add `nextNumber` to `sum`. Really, we don't even need `nextNumber`—we could add the input directly (after converting it to an integer). I'm using `nextNumber` here mostly to show off having multiple lines controlled by our `for` loop. Then, at the end, we print the `sum` divided by the number of numbers we added, 10.

However, what if we didn't want to just average 10 numbers? What if we wanted to let the user decide how many numbers to average? We can do that pretty easily, actually: we just have to get the number from the user, then run the loop to that number instead of to 11:

In Figure 3.3.4, instead of just jumping straight to `range(1, 11)`, we instead first prompt the user for the count of numbers to average. Then, we run the loop from 1 to `numCount + 1`—the plus 1 is so the loop *runs* the number of times the user inputted instead of stopping one short. Remember, a loop from 1 to 11 runs 10 times, so if the user wants to run 5 times, we need a loop from 1 to 6.

#	ForLoopswithUnknownRanges-1.py	Output
1	`#Creates sum with the value 0`	`How many numbers will you average? 5`
2	`sum = 0`	`Enter number #1: 91`
3	`#Get the number of numbers to average`	`Enter number #2: 92`
4	`numCount = int(input("How many numbers will you average? "))`	`Enter number #3: 93`
5	`#Loop numCount times`	`Enter number #4: 94`
6	`for i in range(1, numCount + 1):`	`Enter number #5: 95`
7	`    #Gets the user's number`	`93.0`
8	`    nextNumber = int(input("Enter number #" + str(i) + ": "))`	
9	`    #Add the inputted number to the sum`	
10	`    sum += nextNumber`	
11	`#Print the sum over numCount`	
12	`print(sum / numCount)`	
13		

Figure 3.3.4

For-Each Loop
A loop control structure that runs a block of code a predetermined number of times, where the number of times comes from the length of some list and the items in the list are automatically loaded into a variable for usage in the block of code.

Iterate
To repeat code a number of times. For example, if a loop runs for each item in a list, the loop "iterates" over the list. Each time the code is repeated is a single iteration.

3. For-Each Loops in Python

As I mentioned earlier, I see `for-each` loops more commonly than I see `for` loops in Python. Python is specifically set up to make dealing with lists of items easier, and since so many `for` loops exist to **iterate** over lists of items, `for-each` loops are very common. We saw this in our examples above: two of our examples of `for` loops could be easily described as `for-each` loops:

- For each email in your inbox, read it.
- For each item on your shopping list, go get it.

These are a little simpler to write than the `for` loops that we saw earlier. A `for-each` loop runs the same way as a `for` loop, but it streamlines the process of creating a variable, getting the length of the list, and grabbing the next item during each iteration.

For-Each and Lists

We haven't covered lists, but you've seen them a couple times by now. We'll cover them more in Unit 4. For now, though, let's perform the same task as before, but instead of getting numbers one-by-one from the user, let's have them provided in a list.

In Figure 3.3.5, line 3 creates a list of numbers. The list is called `listOfNumbers`, and inside the brackets on the right, we list numbers separated by commas. The result is a list of 10 numbers, 91 through 100. We'll talk about this more in Unit 4. As before, we then create `sum` and give it the value 0.

#	ForEachandLists-1.py	Output
1	`#Creates listOfNumbers and assigns it to a list of ten`	95.5
2	`#numbers, 91 through 100`	
3	`listOfNumbers = [91, 92, 93, 94, 95, 96, 97, 98, 99, 100]`	
4	`sum = 0`	
5		
6	`#Runs this loop once for each item, assigning the current`	
7	`#item to the variable 'currentNumber'`	
8	`for currentNumber in listOfNumbers:`	
9	`    sum += currentNumber`	
10	`#Divides sum by the number of items in the list`	
11	`print(sum / len(listOfNumbers))`	
12		

Figure 3.3.5

Line 6 is our `for-each` loop. Notice that it doesn't use the word "each." The computer knows this is a `for-each` loop because `listOfNumbers` is a list instead of a range. The computer infers from the context what it should do, like when it inferred from the data types we were using how to use addition and multiplication operators. Just like `i` took on each number in a range in a `for` loop, `currentNumber` is given each value in sequence from the list.

So, let's trace through this. On the first run of the loop, `currentNumber` is given the first value from `listOfNumbers`, which is 91. 91 is added to `sum`, which was previously 0. So, now the value of `sum` is 91. That iteration of the loop is now done. The code jumps back to the top, and `currentNumber` is now assigned to the next value on the list, 92. That value is then added to `sum`, so `sum` now equals 183. This continues for each item in the list. Then, the program prints `sum` divided by the number of numbers that were added, in this case expressed as the length of the list of numbers, or `len(listOfNumbers)`

To see the equivalence of these different types of `for` loops, note that the `for-each` loop in Figure 3.3.5 is identical to the `for` loop in Figure 3.3.6. In Figure 3.3.6, we run the `for` loop from 0 to the length of the list of numbers. We start at 0 because the computer sees the first item in the list as the "zeroth" item, for reasons we'll describe in Unit 4. Then, for each iteration of the list, the first thing we do is grab item `i` (e.g., item 4) from the list, assign it to `currentNumber`, and then add `currentNumber` to `sum`. It works exactly the same, it just introduces some more manual work: we have to manually get the length of the list and the current number.

#	ForEachandLists-2.py	Output
1	`#Creates listOfNumbers and assigns it to a list of ten`	95.5
2	`#numbers, 91 through 100`	
3	`listOfNumbers = [91, 92, 93, 94, 95, 96, 97, 98, 99, 100]`	
4	`sum = 0`	
5		
6	`for i in range(0, len(listOfNumbers)):`	
7	`    currentNumber = listOfNumbers[i]`	
8	`    sum += currentNumber`	
9	`#Divides sum by the number of items in the list`	
10	`print(sum / len(listOfNumbers))`	
11		

Figure 3.3.6

For-Each and Other Types

We'll use `for-each` a lot in Unit 4 because many of the data types we'll discover are different forms of lists. Lists, tuples, and dictionaries are all data types that in some way implement a list-like structure. Another one, however, you've already seen: strings. Strings are, effectively, ordered lists of individual characters. So, we can use a `for-each` loop with a string as well.

For example, imagine we wanted to count the number of words in a string. We might infer that we can do that by counting the number of spaces in the string: the number of words should be the number of spaces, plus one. How do we do this?

In Figure 3.3.7, we're doing something more complicated than what we've done before: we're using a conditional *inside* a `for-each` loop. We check each character (`currentCharacter`) of the string to see if it equals a space on line 7; if it does, it increments `numSpaces` on line 8. Then, at the end on line 9, it prints `numSpaces + 1`, which would be the number of words.

#	ForEachandOtherTypes.py	Output
1	`myString = "There are seven words in this string."`	There are 7 words in the string.
2	`numSpaces = 0`	
3		
4	`#Runs this loop for each character in the string`	
5	`for currentCharacter in myString:`	
6	`    #Checks if the character is a space`	
7	`    if currentCharacter == " ":`	
8	`        numSpaces += 1`	
9	`print("There are " + str(numSpaces + 1) + " words in the string.")`	
10		

Figure 3.3.7

Notice a few things here. First, notice that technically, characters and strings aren't different data types in Python. They are in many other languages, but in Python, a character is just a string with a length of 1. So, when it iterates over the characters in the string, it's just iterating over smaller strings of length 1 that make up the bigger string.

Second, notice the nested structures here. The conditional on line 7 is *inside*, or *controlled by*, the `for` loop on line 5, so it's indented under the `for` loop. Line 8, which increments `numSpaces`, is controlled by both the `for` loop (it could repeat for each character) and the conditional, so it's indented under both (and double-indented as a result).

Third, notice the explicit conversion in our print statement on line 9. Python won't just add a number to a string, so we have to convert it to a string: `str(numSpaces + 1)`. However, notice that we're adding 1 to `numSpaces` right here inside the explicit conversion to a string. This isn't changing `numSpaces`; if we printed it afterward, it would still equal 6. That's because we're not using the assignment operator; we're not saying `numSpaces +=` 1, we're just asking for the string version of `numSpaces` + 1. We're saying, "what is `numSpaces` plus one?" instead of "set `numSpaces` equal to itself plus one." Note that we can bypass this by using commas instead of addition signs: this tells Python to use the implicit conversion to a string instead, which is what we'll usually use.

Fourth and finally, notice that this code doesn't *really* work the way it's intended to. It works when we have predictable strings, but it would generate a lot of wrong results. It would say an empty string has 1 word (0 spaces + 1 = 1 word). If a string had back-to-back spaces, it would assume there was a word between them even if there wasn't. These are what we might call **edge cases**. They're outside of what we usually expect, but when they happen, they demand some special processing. For example, to improve this, we might tell our code to ignore consecutive spaces, or assume the length of an empty string is zero words.

Edge Case
A rare situation that requires special processing to handle.

4. While Loops in Python

`for` loops run code blocks a certain number of times. This might be the number of items in a list, or a preset number of iterations. A `while` loop, on the other hand, runs some code as long as some situation or condition remains true.

Any of our `for` loops could have been rewritten as `while` loops. However, `while` loops are good for code where we don't know how many times we'll need to repeat in advance. For example, imagine we're playing a number-guessing game with the user. We don't know how many guesses it will take to get the correct number. So, we want to repeat while their guess is incorrect.

Simple While Loops

Let's take a simple example of this. You might notice what we're about to write is similar to something we wrote previously with `for` loops. That's exactly right: any `for` loop can be written as a `while` loop. Figure 3.3.8 shows some code to count up to a number with a `while` loop.

#	SimpleWhileLoops.py	Output
1	`i = 1`	1
2	`#Repeat while i is less than 11`	2
3	`while i < 11:`	3
4	`    print(i)`	4
5	`    i += 1`	5
6		6
7		7
8		8
9		9
10		10
11		
12		

Figure 3.3.8

On line 1, we create `i` and set it equal to 1. We need to create it in advance so we can check it when we get to the loop. Then we write our `while` loop. We start with the word `while`, which is a reserved word like `if` and `for`. We then write the logical expression that will control the loop. Notice this is just like an `if` statement: in many ways, a `while` loop is an `if` statement that repeats until the condition is `False` instead of just running once if it was `True`. As before, we end with a colon to mark off the code block the loop will control.

Inside the `while` loop, we print the current value of `i`, and then add 1 to it. So, let's trace this code. When the program reaches the `while` loop the first time, `i` is 1, which is less than 11. So, it prints `i` (1), then adds 1 to `i` to make it 2. Then, it jumps back to the top and checks the logical expression again: is 2 less than 11? Yes, it still is. So, it runs the body again: it prints `i`, now 2, and then adds 1 to `i`, making it 3. It repeats this until `i` is 11. At that point, the logical expression is no longer true, and so it stops running. Notice that we increment `i` *after* we print it. If we incremented it first, the loop would run the same number of times, but it would print 2 to 11 instead of 1 to 10.

Notice how similar this is to our `for` loops. With our `for` loop, we noted that when the code jumps back to the top, it increments `i` or whatever variable we use to control the loop. Here, we increment `i` manually on line 5. We noted that whenever the code jumps to the top, it checks if `i` is still less than the second number in the range. Here, this check is what governs the `while` loop on line 3. So, anything a `for` loop can do, a `while` loop can do, too. A `for` loop—especially a `for-each` loop in Python—is just a little more efficient to write and often more natural to think about.

While Loops and Number Guessing

Let's try to create a small game using `while` loops. In this game, we'll randomly generate a number from 1 to 100, and we'll ask the user to guess the number. While their guess is wrong, we'll give them feedback on which direction to guess and repeat until it's correct. The code for this is shown in Figure 3.3.9.

```
#  WhileLoopsandNumberGuessing.py
1  import random
2  #Get a random number from 1 to 100
3  hiddenNumber = random.randint(1, 100)
4  #Create userGuess and give is a value that can't be correct
5  userGuess = 0
6  #Repeat until the guess is correct
7  while not userGuess == hiddenNumber:
8      #Get the user's next guess as an integer
9      userGuess = int(input("Guess a number: "))
10     #Check if the guess is too high
11     if userGuess > hiddenNumber:
12         print("Too high!")
13     #Check if the guess is too low
14     elif userGuess < hiddenNumber:
15         print("Too low!")
16     #The guess must be right!
17     else:
18         print("That's right!")
19
```

```
Output
Guess a number: 50
Too low!
Guess a number: 75
Too low!
Guess a number: 87
Too high!
Guess a number: 81
That's right!
```

Figure 3.3.9

We start with `import random` on line 1 just so we can get random numbers. You can read more about Python's random library and what you can do with it at https://docs.python.org/3/library/random.html. Then on line 3, we create a random number from 1 to 100 using `random.randint(1, 100)`. Here, that integer was 81, but the user doesn't know that initially. We then create `userGuess` on line 5 and give it the value 0; we use 0 because `hiddenNumber` is greater than or equal to 1, so using 0 guarantees that we won't accidentally set `userGuess` to the right answer.

`random.randint(min, max)`
Returns a random integer greater than or equal to min and less than or equal to max.

Then, we get started on the `while` loop on line 7. The `while` loop repeats as long as the user's guess is not correct. Initially it's guaranteed not to be correct, so the contents of the `while` loop definitely run the first time. It gets the user's guess (for now, we assume the user correctly enters an integer), and runs a conditional on the guess:

- If the guess is too high, it prints "Too high!".
- If the guess is too low, it prints "Too low!".
- If the guess is neither too high nor too low, it must be correct, so it prints "That's right!"

That's the end of the `while` loop's contents, so how does it know whether to repeat again? At the end of the loop's contents, `userGuess` is whatever number the user entered. So, when the loop ends, it jumps back to the top, line 7. Then, it checks `userGuess` to see if it's equal to `hiddenNumber`. If it *is* equal, then this statement is `False` because of the `not` in front, and so if it *is* equal, the loop will not repeat. If it's not equal, the loop repeats again.

Infinite Loops

You might notice in the previous example that something risky could happen. What if the user never enters the right answer? What if they just enter "1" over and over and over again? The program will never end! Now what if instead of waiting on user input in the loop, the loop could just run as fast as it wanted? This is called an **infinite loop**: a loop that, by its design, will never end.

Infinite Loop
A loop that will never end because the conditions for ending the loop will never be met.

These are very easy to create. This simple loop in Figure 3.3.10, for example, will never end. We create `i`, our loop control variable, and assign it the value 1. Then, we create a `while` loop that repeats as long as `i` is greater than 0. Every time it repeats, we just print `i`. What will the result be here? Simple: it will just print 1

#	InfiniteLoops.py	Output
1	`i = 1`	
2	`while i > 0:`	
3	`    print(i)`	
4		

Figure 3.3.10

over and over and over again, millions of times per second if our computer allows it. If the loop repeats so long as `i` is greater than 0 and the value of `i` never changes, then the loop will never terminate.

Infinite loops are errors, but they're interesting errors in that they generally don't give us any actual error messages. When we use a variable outside its scope or divide by zero, the computer tells us, "Hey, you can't do that!" But when we have an infinite loop in our code, it just runs and runs and runs. If your code is complex, it can be tough to tell if it's just running normally and taking a while, or encountering an infinite loop. `print()` statements can help with this: if you use print debugging and see the same thing printed over and over again, it often means you're looping over the same thing over and over. I can say, personally, I've done that *many* times, usually just by forgetting to add that last line that increments some loop control variable.

5. Advanced Loops: Nesting in Python

We discussed previously that we can nest conditionals within one another. We can do the same thing with loops. We can have `for` loops inside `for` loops and `for` loops inside `while` loops. We can have `while` loops inside `while` loops and `while` loops inside `for` loops. We can have `for` loops inside `while` loops inside conditionals and conditions inside `while` loops inside `for` loops. Before this book becomes too much like a Dr. Seuss poem, let's take a look at some examples.

Nested For Loops

Earlier, we used `for-each` loops to loop over lists of strings and characters in a string. Now, let's try doing both at once. Imagine we have a list of strings of multiple words, and we want to count the words in the list. To do that, we need to iterate over each character in each string; there are two "each"es in that statement, which means two `for-each` loops.

Figure 3.3.11 shows the code that does this. First, in lines 3 through 5, note that we have another way to tell the computer to interpret multiple lines as one line for human readability. If we're creating a list and separating our list items with commas, we can create a new line and continue the list. Python knows that the list isn't over until it sees that closed bracket, so it knows we're still creating the list—that lets us break the line up into multiple visual lines so that we can read it more easily.

#	NestedForLoops.py	Output
1	`#Creates listOfStrings and assigns it a list of strings each with`	There are 25 words in these strings.
2	`#multiple words`	
3	`listOfStrings = ["This is the first string", "This is the second string",`	
4	`                 "This is the third string", "This is the fourth string",`	
5	`                 "This is the fifth string"]`	
6	`numSpaces = 0`	
7	`#Loops over each string in listOfStrings`	
8	`for currentString in listOfStrings:`	
9	`    #Loops over each character in currentString`	
10	`    for currentCharacter in currentString:`	
11	`        #Checks if the current character is a space`	
12	`        if currentCharacter == " ":`	
13	`            numSpaces += 1`	
14		
15	`numWords = numSpaces + len(listOfStrings)`	
16	`print("There are", numWords, "words in these strings.")`	
17		

Figure 3.3.11

Starting on line 8, we loop over each of the strings in `listOfStrings`. Each time we loop, the next string in the list gets assigned to the variable `currentString`. So, the first time this loop (the "outer loop") runs, `currentString` is assigned the value "This is the first string." The second time it runs, `currentString` is assigned the value "This is the second string." And so on.

Each time the outer loop iterates once, the inner loop on line 10 runs. This one iterates over each character in `currentString`. For each character, it checks if the character is a space, and increments `numSpaces` if so. Notice that `numSpaces` is created outside *both* loops. If we created it inside the inner loop, the code would run, but with each iteration of the outer loop, its value would be reset. So, by the end of it, we'd have a list of spaces in the *last* string, not all the strings.

After this code has iterated over every character in every string, it then creates `numWords` and sets it equal to `numSpaces + len(listOfStrings)` on line 15. Why does it do this? Recall that previously, we added 1 to `numSpaces` because one space meant two words. Here, that's true for each string individually: each string's first space suggests two words. So, we want to add one for each string in the list. Granted, there are other ways we could do this, too: we could run an additional `for-each` loop at the end to add 1 for each string, or we could simply add 1 to `numSpaces` each time the outer loop runs, assuming there is at least one word in the string.

Nesting Both Loops

The previous example showed nesting `for` loops (and as an added bonus, nesting a conditional inside a `for` loop). What about nesting `while` loops inside `for` loops, or `for` loops inside `while` loops? Let's try that out. Let's take our previous `while` loop-enabled number-guessing game and extend it to allow the player to decide in advance how many games they'd like to play. Note that we're about to get pretty complicated, so if you find yourself confused, don't worry: try to trace through the code line-by-line to understand how it's running.

#	NestingBothLoops-1.py	Output
1	`import random`	How many games would you like to play? 5
2	`numGames = int(input("How many games would you like to play? "))`	Game start!
3	`#Repeats this the number of games the user chose`	Guess a number: 50
4	`for i in range(0, numGames):`	Too high!
5	`    print("Game start!")`	Guess a number: 25
6	`    #Get a random number from 1 to 100`	Too low!
7	`    hiddenNumber = random.randint(1, 100)`	Guess a number: 37
8	`    #Create userGuess and give is a value that can't be correct`	That's right!
9	`    userGuess = 0`	Game start!
10	`    #Repeat until the guess is correct`	Guess a number: 50
11	`    while not userGuess == hiddenNumber:`	Too high!
12	`        #Get the user's next guess as an integer`	Guess a number: 25
13	`        userGuess = int(input("Guess a number: "))`	Too low!
14	`        #Check if the guess is too high`	Guess a number: 37
15	`        if userGuess > hiddenNumber:`	Too low!
16	`            print("Too high!")`	Guess a number: 43
17	`        #Check if the guess is too low`	Too low!
18	`        elif userGuess < hiddenNumber:`	Guess a number: 47
19	`            print("Too low!")`	Too low!
20	`        #The guess must be right!`	Guess a number: 48
21	`        else:`	
22	`            print("That's right!")`	
23		
24		
25		

Figure 3.3.12

This code is shown in Figure 3.3.12. Notice a few things here: first, notice that we keep the `import` statement at the top. That's a common convention, to put all the `import` statements at the top. For now, don't worry too much about what these do: just know that sometimes with things like `random` and `datetime`, we have to import something before we can use it.

Second, notice that in my outer `for` loop on line 4, I'm running the loop from 0 to `numGames` instead of 1 to `numGames + 1`. These will run the same number of

times, and if I don't care to print or use the value of `i` inside the loop, the fact that it starts at 0 isn't as confusing. Even if it was, I could simply print `i + 1` whenever I wanted to use it to translate it from Python's interpretation to ours.

Third and most importantly, though, notice that it took only two lines (lines 2 and 4) of code to change this game from running once to running a user-defined number of times. In fact, we could have shrunk it even more to only one line; we could replace `numGames` in line 4 with the input function call itself. That's the power of loops: tiny segments of code can radically change the behavior of the finished product.

This is still a little bit of an odd design, though. What if the user decides halfway through they want to quit? Or what if they reach the end and decide they want to keep playing? We can take care of that by switching this to nested `while` loops, as shown in Figure 3.3.13.

#	NestingBothLoops-2.py
1	`import random`
2	`keepPlaying = "y"`
3	`#While keepPlaying is "y"`
4	`while keepPlaying == "y":`
5	`    print("Game start!")`
6	`    #Get a random number from 1 to 100`
7	`    hiddenNumber = random.randint(1, 100)`
8	`    #Create userGuess and give is a value that can't be correct`
9	`    userGuess = 0`
10	`    #Repeat until the guess is correct`
11	`    while not userGuess == hiddenNumber:`
12	`        #Get the user's next guess as an integer`
13	`        userGuess = int(input("Guess a number: "))`
14	`        #Check if the guess is too high`
15	`        if userGuess > hiddenNumber:`
16	`            print("Too high!")`
17	`        #Check if the guess is too low`
18	`        elif userGuess < hiddenNumber:`
19	`            print("Too low!")`
20	`        #The guess must be right!`
21	`        else:`
22	`            print("That's right!")`
23	`    #Update keepPlaying`
24	`    keepPlaying = input("Play again (y for yes, n for no)? ")`
25	

Output
Game start!
Guess a number: 50
Too high!
Guess a number: 25
That's right!
Play again (y for yes, n for no)? y
Game start!
Guess a number: 50
Too high!
Guess a number: 25
Too low!
Guess a number: 37
That's right!
Play again (y for yes, n for no)? n

Figure 3.3.13

Instead of asking for the number of games in advance and running a `for` loop, we could instead ask the user at the end of each game if they want to continue. Then, we run a `while` loop while their answer to the final question is "y" for "yes." To design it this way, we would set the initial value of `keepPlaying` to "y" on line 2 so that the loop is guaranteed to run the first time. Other languages might do this with a `do-while` loop, which similarly guarantees the loop will run at least once, but Python does not have a `do-while` loop.

6. Advanced Loops: Keywords and Scope in Python

Before we end our discussion of loops, there are two final things we should discuss. First, Python has a couple of additional keywords that govern loops. So far we've covered `for`, `while`, and `in`. We also have `continue`, `break`, and `pass`. In practice, these are used relatively rarely (in my experience, anyway), but they're worth covering in case you come across them.

Second, we discussed scope earlier in our material. Scope is important `for` loops as well, so we'll briefly discuss the scope of variables used in loops.

Advanced Loop Keywords

There are three final keywords with loops that are worth covering, although to be honest, I find relatively few opportunities to use these. They're covered here for the sake of completeness, but don't be surprised if you rarely see these in the wild.

The first is the `continue` statement. The `continue` statement forces the current iteration of the loop to stop, skipping over any remaining code inside the loop, as shown in Figure 3.3.14.

#	AdvancedLoopKeywords-1.py
1	`#Runs this loop 20 times`
2	`for i in range(1, 21):`
3	`    #Checks if i is even`
4	`    if i % 2 == 0:`
5	`        #Skips the rest of the code block if so`
6	`        continue`
7	`    #Prints that i is odd`
8	`    print(i, "is odd.")`
9	`print("Done!")`
10	
11	
12	
13	
14	

Output
1 is odd.
3 is odd.
5 is odd.
7 is odd.
9 is odd.
11 is odd.
13 is odd.
15 is odd.
17 is odd.
19 is odd.
Done!

Figure 3.3.14

This code runs 20 times and prints the odd numbers. It does this by checking if `i`, the loop control variable, is even. If it's even, it runs the `continue` statement, which skips the remainder of the loop's code and continues to the next iteration. The keyword `continue` is used inside a loop's code block to skip the rest of the code block for the current iteration, and return to the loop itself.

The second advanced structure is the `break` statement. Like the `continue` statement, the `break` statement forces the entire loop to terminate; it will not iterate any further even if the conditions governing the loop remain `True`, as shown in Figure 3.3.15.

#	AdvancedLoopKeywords-2.py
1	`#Runs this loop 20 times`
2	`for i in range(1, 21):`
3	`    #Checks if i is even`
4	`    if i % 2 == 0:`
5	`        #Skips the rest of the loop if so`
6	`        break`
7	`    #Prints that i is odd`
8	`    print(i, "is odd.")`
9	`print("Done!")`
10	

Output
1 is odd.
Done!

Figure 3.3.15

If we replace the `continue` statement with a `break` statement, then just like before, the loop terminates before it prints `i` for the second iteration. Unlike with the `continue` statement, however, with `break`, the loop terminates altogether. It jumps straight to whatever line follows the loop's code block, as opposed to back up to the loop itself. It breaks the loop.

Finally, the third advanced loop keyword we might use is the word `pass`. The keyword `pass` exists by necessity. How do you have a loop with an empty code block? A blank line isn't interpreted as a line by Python; something needs to be there and indented. The keyword `pass` is simply what you put if you want to run a loop that does nothing. It can also be used when you want to run a conditional, function, or exception handler with no code block underneath it as well. Why would you want to do either of these things? That mystery is left to you, dear reader. (Also, we'll see why later in this unit.)

Scope and Loops

Finally, we should briefly talk about scope in the context of loops. As we said with conditionals, the scope of a variable is the area of the program's execution where the

variable can be seen. In Python, a variable's scope is effectively anything after the variable is created. It doesn't matter if it's declared inside a loop or outside a loop; anything after the variable is created is within the variable's scope.

With conditionals, this presented a challenge if a variable was created inside a conditional, then referred to outside the conditional. If the conditional didn't run (which, by design, it sometimes shouldn't—if it always should, why have a conditional?), then the variable was not created, and referring to it later would create an error. This isn't as big a deal in loops. We rarely create loops that might not run at all. It can happen, but it isn't a common decision in my experience.

However, loops create a different issue for creating variables inside loops. The first time you assign a value to a variable, Python creates the variable. The second time, Python just changes the variable's value. That means if you create a variable inside a loop, then every time the loop runs, its value is *replaced*, as shown in Figure 3.3.16. Many times that's perfectly fine; if you only need the value of that variable within one iteration of the loop, then there's no problem replacing its old value with the new one.

ScopeandLoops-1.py

```
1  #Get the number of numbers to average
2  numCount = int(input("How many numbers will you average? "))
3  #Loop numCount times
4  for i in range(1, numCount + 1):
5      #Creates sum with the value 0
6      sum = 0
7      #Gets the user's number
8      nextNumber = int(input("Enter number #" + str(i) + ": "))
9      #Add the inputted number to the sum
10     sum += nextNumber
11 #Print the sum over numCount
12 print(sum / numCount)
13
```

Output

```
How many numbers will you average? 5
Enter number #1: 91
Enter number #2: 92
Enter number #3: 93
Enter number #4: 94
Enter number #5: 95
19.0
```

Figure 3.3.16

However, if you're going to reference the variable outside the loop, *usually* it's because you wanted that variable to persist across different iterations of the loop. If the variable was created inside the loop, then referencing it after the loop will just give you the value the variable received the last time the loop ran, which is rarely what we want. So, the same advice I had in conditionals applies here, too: generally, if you need to refer to a variable outside a loop, don't create it inside a loop. Create it before the loop, as shown in Figure 3.3.17.

ScopeandLoops-2.py

```
1  #Creates sum with the value 0
2  sum = 0
3  #Get the number of numbers to average
4  numCount = int(input("How many numbers would you like to
5  average? "))
6  #Loop numCount times
7  for i in range(1, numCount + 1):
8      #Gets the user's number
9      nextNumber = int(input("Enter number #" + str(i) + ": "))
10     #Add the inputted number to the sum
11     sum += nextNumber
12 #Print the sum over numCount
13 print(sum / numCount)
14
```

Output

```
How many numbers would you like to average? 5
Enter number #1: 91
Enter number #2: 92
Enter number #3: 93
Enter number #4: 94
Enter number #5: 95
93.0
```

Figure 3.3.17

7. Loops and Turtles

Last time in our work with the turtles graphics module, we created a little segment of code that would allow the user to enter two commands (turn or forward) two times. That was rather limited, though: with only two iterations, we could draw at most one line in any direction. With loops, though, we've already seen how we can loop some code until the user is done. Let's apply that to our turtles with a `while` loop, and then use a `for` loop to create an interesting portion of an image for the user to draw.

While Loops for Repeated Commands

We noted before that it only took a couple of extra lines to make some code repeat with a `while` loop. We'll see the same thing in WhileLoopsforRepeatedCommands.py.

As we see, it's actually far less code to loop over this with this `while` loop than it was to repeat it twice manually! After deleting the repeated code, we only made the following changes:

- Created `command` outside the loop on line 2 so we could use it in the loop's logical expression. This way, the user just enters the command "end" to stop the loop.
- Add the `while` loop line to repeat until command is "end" on line 5
- Add a separate `elif` to check for "end" on line 21. We need this because otherwise, if the user entered "end," it would be caught by the `else` on line 24 and register as an invalid command. However, we don't need to *do* anything inside that `elif` because the loop won't repeat again if command is "end" (because of the condition for the `while` loop; we just have to prevent the `else` from running.

Interestingly, we could have also used `break` or `continue` where we use `pass` here. If the user enters "end," our goal is to stop repeating the loop. Using `break` would force the loop to stop altogether. Using `continue` would skip the `else` (which wouldn't run anyway) and return to the logical expression governing the loop, which would end the loop, too, since the command now is "end." It's important to note as well that we could restructure this loop to not have to use `pass`, `continue`, or `break` at all.

With this relatively simple change, we now have some code that will allow the user to keep entering commands until they type end. They could run 200 commands if they wanted to and draw pretty complex figures.

For Loops for Drawing Shapes

We've seen how a `while` loop can be used to let the user keep entering multiple commands. Now let's see how we could use a `for` loop to create some more complex commands. We're going to implement a shape command. A shape command lets the user enter a number of sides and a side length, and it will draw the regular polygon that they entered. If they entered 4 and 100, for example, it would draw a square with side length 100, starting from the current direction it's facing. What's remarkable is this actually won't take many lines of code!

The results of this implementation are in ForLoopsforDrawingShapes.py. In six lines (not including comments), we created this ability. Let's walk through them.

- In line 21, we check if the command entered was "shape," just like the other commands.
- In line 23, we get the number of sides. For now, we just assume the user enters a valid number of sides (an integer greater than 2).
- In line 25, we get the length of each side. Again, we assume the user enters a valid number (any integer).
- In line 27, we run a loop for however many sides the user wrote. Here, we use a convention in Python: if we're never going to *use* the loop control variable (what we usually call `i`), then we name it `_`, the underscore character. Python sees this just as a normal variable name, but visually, it's a nice indicator to other humans that this variable is never used besides its role controlling the loop.
- In line 28, we simply move the turtle forward by the side length.
- In line 29, we use a little principle from math to complete our shape. To draw a complete shape, we need to rotate 360 total degrees. So, we divide 360 degrees by the number of sides, and rotate that length at the end of each side. This way, we're guaranteed to draw a complete shape.

With only these six lines, the user can now enter a number of sides and side length, and Python will automatically draw the corresponding shape. The code starts with the turtle in whatever direction it was facing originally, so we can rotate to draw shapes in different orientations. For example, if issued the command "turn" with angle 45, then "shape" with `numSides` 4 and side length 50, we'd draw a diamond with sides of length 50.

It's cool that we can do that, but if we wanted to add a lot of commands like that, this code would get really long. It would get hard to differentiate the high-level organization from the details of drawing individual shapes or patterns. So, in the next chapter, we'll talk about functions. Among several other benefits, functions are ways of separating out often-used code so that we can refer to it from different places without cluttering our code.

CHAPTER

3.4

Functions

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Explain functional programming paradigm analogically, including function definitions and function calls;
- Write functions and implement function calls, including those with return statements and different types of parameters;
- Recognize Python function errors, including parameter mismatch and scope errors;
- Equip their user-facing turtle script with functions to build more complex programs.

Function
A segment of code that performs a specific task, sometimes taking some input and sometimes returning some output.

Function Call
A place where a function is actually used in some code.

1. What Is a Function?

A **function** is like a little program on its own. Like full programs, a function takes some input and produces some output. In that sense, functions are very simple. We could slap a function declaration (the line of code that tells the computer that the code that follows is a function) on top of any code we've written so far and call it a function. In fact, some languages (like Java) can't do *anything* outside a function.

Power of Functions

Despite their simplicity, functions are extremely powerful. We've talked about loops, which let us repeat some lines of code without repeating them. A loop sat in a single place in our code, though. What happens if we wanted to repeat the same loop in two different places? We would have to just copy the loop's code to the second place! Not only is that inefficient, it means that if later we have to change that code, we have to remember to change it in two different places.

Functions change that. Functions let us take some code and package it up into a mini-program. Then, whenever we need to use that code, we just **"call" that function;** to call a function means to use it in some other code. Take our running example of validating a purchase. Instead of having to put that long, complex series of conditional statements anywhere we need to validate a purchase, we could instead create a function named `validatePurchase()` and call that function whenever we need to validate a purchase. Inside that function would be the same conditional statements, but we would only need to put it in one place: once it was there, we could refer to it from anywhere else.

To be honest, this topic is probably my favorite topic in this entire book. Functions are the first step that allow us to transition from the clever little bits of code we've been writing to writing real complete programs. In real programs, nearly every single segment of code will have a function call in it somewhere. The power of functions to support organization and reuse of code drastically increases what we can easily create.

Function Terminology: Calls and Definitions

Functions are likely the most complex topic we've discussed so far, and so they come with their own terminology. In order to discuss function terminology, let's imagine a simple function for addition. The function would take as input two numbers, and produce as output their sum. This function is pretty trivial, but it will be useful to explain the concept: this function called `add()` will take two numbers as input, and return as output their sum. Notice that we're describing this function the same way we described programming itself, lines of code that take in input and return output. Functions are like mini-programs, and we build big, complex programs out of lots of little simple "programs." These little simple programs are functions.

We've already mentioned one term describing functions: a function call. A function call is the place where we actually use, or "call," a function in our code. In our `add()` example, it's the place where we say, "hey, add these two numbers!" You've no doubt seen function calls before in our material depending on the lan-

guage. Things like printing text or converting between data types are often done via functions. These are function calls: you weren't sure how the functions worked, but you didn't need to know. To call the function, all you needed was to know what it would accomplish, what input to give it, and what to call it.

The opposite, in some ways, of the function call is the function definition, made of a function header and a function body. The function definition is what actually creates the function so that it can be called from other parts of code. It's the place where we tell the computer, "Hey, if someone wants to add two numbers, here's what input you'll get, here are the steps to take, and here's the result to give back to them."

Going back to our analogy of functions as mini-programs, we can think of the function definition as the program's code, and the function call as actually running the mini-program. Writing the function definition is like writing the code; it hasn't run yet, but it's there ready to be used. Calling the function is like running some code, and for that, you don't necessarily have to understand how it works: you just have to know what it will do.

Parts of a Function Definition

Function Definition
A segment of code that creates a function, including its name, parameters, and code, to be used by other portions of a program.

Parameter
A variable for which a function expects to receive a value when called, whose scope is the function's own execution.

Function Header
The name and list of parameters a function expects, provided as reference to the rest of the program to use when calling the function.

Function Body
The code that a function runs when called.

Return Statement
The line of code that defines what output will be send back at the end of a function.

The structure of **function definitions** differs from language to language, but most have some commonalities. First, most are made of a header and a body. The header names the function and states what input it will expect; the input is shown as a list of **parameters**. For a `print()` function, for example, the name is "print" and there is one parameter, the text to print. For our addition function, the name might be "add", and it would have two parameters: the two numbers to add. In some languages, the function itself and each parameter will be given a type. This is all the **function header**: it defines the function as far as the rest of the program is concerned. It tells the rest of the program what name to call when it needs the function, and what information to pass along. That is all the rest of the program will need to know to use the function: it doesn't have to know how the program works, it just needs to know what to call it and what to give it.

In most cases, the majority of the function definition is the code itself, called the **function body**. These are the actual lines of code that dictate what should be done when the function is called. These are the lines of code that would say, "Add these two numbers and store the result in `sum`, then return `sum`." Generally, the function body isn't seen by the rest of the code you're writing outside the function. That code doesn't need to see the body: it just needs to know what input to give and what the result will be. For our addition function, our main code doesn't need to know *how* the addition function adds two numbers; it just needs to trust that it does so accurately and returns the result.

The word `return` is key in that statement: it's the other unique piece of a function. The `return` statement is where the function sends something back to the main program as output. When the code says, "Hey, add 5 and 2," the function replies, "Hey, the answer is 7!" It returns its output, the number 7, to the main program. A `return` statement terminates that function and returns execution to the main program. When the addition function says, "Hey, the answer is 7!", it's done running.

So, these are the main parts of a function.

- A definition or header, which defines its name and what input it should expect. With our addition function, this might be the name "add" and the expectation of two numbers as parameters for input.
- A body, which are the actual lines of code that run when the function is called. With our addition function, this would be the lines of code to add two numbers and store the result.
- A return, which tells the function what to send back to the main program as output. With our addition function, this would tell the function to return the sum of the two numbers.

Parts of a Function Call

Once we have that, we can call the function. Calling a function means that when our code reaches the line where we call the function, it jumps into and runs the function's code. It runs the function's code until it finds a return statement or the function otherwise ends, and then it returns and picks up where it left off in the regular code. So, in some ways, a function all is like saying, "Go to that other code over there, bring this input with you, and tell me what the output is!" The execution of the code then goes to the function, runs the function's code, and returns to the main code with the output.

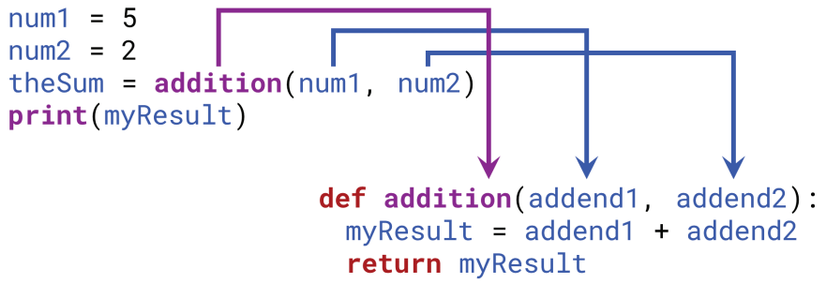


Figure 3.4.1

To call a function, we would write the function's name, and then give it the input to provide the function's parameters. We call the input provided in the function call **"arguments."** This is nearly identical to variables and values. Variables are names that are assigned values. Parameters are variables specifically for a function, and they are assigned values, called arguments, when the function is called. In our addition example, the "addition" function had two parameters: addend1 and addend2. When we say "Hey, add 5 and 2!", 5 and 2 are arguments, which are loaded into the parameters. So, when the function has the code `addend1 + addend2`, these are variables that are loaded with the values 5 and 2.

Arguments
Values passed into parameters during a function call. Essentially, these are the values assigned to the function's dedicated variables (i.e., parameters).

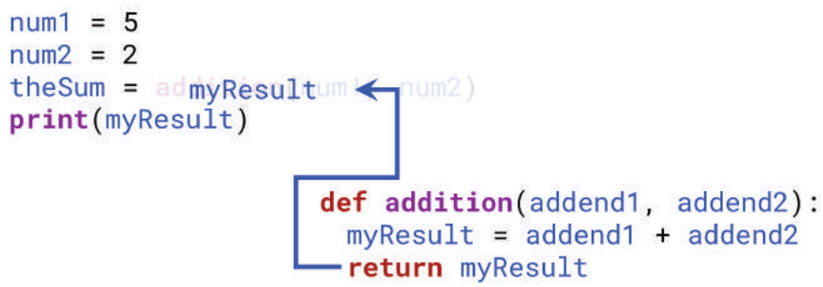


Figure 3.4.2

Then, the function would return the sum, 7. This would jump the execution back to our main code, where the function call asking the function `add()` to add 5 and 2 would be replaced by the value 7. These are the parts of the function call: calling the function by name and providing the arguments, and then being replaced by the output.

Note that not all functions in all languages have return statements; some might not need to return any output to the main program. For example, a function to save a document might not need to tell the main program anything, it might simply need to write to a file on its own. Similarly, not all functions require any input; for example, a print function might be called without any input, and the computer knows to just print a blank line.

Finally, our notion of scope still applies to functions. In order to call a function, it must be within the scope of the code we're writing. Different languages define this differently, but it's worth keeping in mind.

2. An Analogy for Functions

Functions are powerful, but complicated. They fundamentally change the way we think about programming. We're no longer writing just linear sequences of instructions, with small branches or repetitions. Instead, we're now structuring entire programs. When writing a function, we need to think about how it might be used by the rest of the program, not just what the needs are for our current line of code. The power of functions means that they're critical to understand, but they're also a fundamental change in how we think about structuring programs. As a result, they can be very confusing. So, I think it's useful to think of functions with an analogy to an office.

Setting Up the Analogy

In this analogy, you're the main code that's running. You have a specific job. Functions are your coworkers. They also have specific jobs. When you need a job done, you ask your coworker to do it. You might give them some information to do it with, and they might give you some answer back.

In this analogy:

- Your coworker is a function.
- Their name is a function name.
- The declaration by the company of what their job will be is the function definition.
- The information you give them is the input to the function.
- What they do on their own with that input is the body of the function.
- The result they return to you is the output of the function.

The Function Definition

Let's use this analogy to actually define a function. You have a coworker named Addison. Your boss tells you that Addison's sole job is to add two numbers, and to get him to do so, you should shout his name and the numbers. So, Addison (the person) is a function. "Addison" is the name of the function. The statement, "You can give Addison two numbers to add" is the list of parameters to the function. All this information is the function header. Addison knows how to add two numbers; that's the body of the function. When he's done adding them, he hollers back the answer; that's the output of the function.

Note a few features of this analogy. First, note that Addison hasn't done anything yet. You've been informed of *what* he does, and how to get him to do it. You're told that he adds two numbers, and you're told that to get him to add two numbers, you holler his name and then the numbers you want him to add. This is all the definition, defining how the function is *used*.

Second, notice that you don't know *how* Addison adds two numbers. Does he do it by mental calculation? Does he do it on paper? Does he use a calculator? You don't know, and you don't care. Remember, in this analogy, you're the main program. You don't need to know how Addison does his job. All you need to know is how to call him, what input to give him, and what output to expect.

The Function Call

So, you're sitting in your office today, and you say, "Hey Addison, add 5 and 2!" That's the function call: you call out to Addison to do something, and you give him the input into what he does. He expects two numbers as input, so you give him

two numbers; those are the arguments, and they go into his parameters (like values into variables). Addison hears the two numbers and adds them. That's running the body of the function. It was defined before, but now it's being used. You, the main program, don't know what's going on inside that function. You're not sure if he's adding mentally, using pen and paper, or using a calculator. When Addison is done, he hollers back, "7!" That's the output of the function. Now, wherever you needed the result of 5 plus 2, you instead use 7.

Notice a couple of things here as well. First, when you hollered for Addison to add 5 and 2, your work stopped. You didn't continue until you got the output back. In the same way, when a program calls a function, it hands control over to that function. It waits until the function is done running to continue.

Second, notice that we said Addison is in your office. You probably have other coworkers as well, and they might need numbers added, too. If the company didn't have Addison, you could have added those two numbers yourself in your office, but then *everyone* in the office has to do their own addition. By having one coworker, one function, that can do it, everyone can just call him to do it. If he comes up with a more efficient way to do it, then everyone benefits instead of everyone having to learn the more efficient way themselves. If later we wanted to record every time he added two numbers, we would only need to ask him to record them instead of teaching everyone to record them. These are all connected to the benefits of using functions: by packaging code together and calling it when needed, we make it easier to revise and enhance our programs over time.

Bigger Functions

Now, of course, with addition this is almost a silly example. In practice, you wouldn't write a function for addition (in fact, the addition operator is basically a function on its own), just as you wouldn't have a coworker whose sole function is to add numbers. This has been a simple example, but you might be able to easily generalize it to more authentic applications.

For example, most mid-sized companies have a public relations person that handles press releases. When someone at the company has something to announce, they call the public relations person and give them the information to announce. Let's call ours Riley. Riley is a function, her parameter would be the information to announce, and her function body would be the steps to announce it. Calling the Riley is calling the function, and providing the information is passing in the argument. This is a good example of a time when a function might not return anything: Riley's job might not be to tell the coworker any result, but rather just to announce something to the world. Either way, though, this should make it more clear (a) why there is value in encapsulating this job within a particular person or function, and (b) what it means to say that you, when calling a function, don't care *how* it works, just *that* it works.

We can also think of an example of a job that needs no input. For example, imagine you're working at a retail store. A common question in such a store is, "What was today's total sales?" A company might have a person whose job it is to tally that when needed. Let's call him Terry. Terry would be the function, and Terry's function body would be the steps to tally the sales, but they don't have any parameters. Terry just automatically has access to the day's sales. Asking Terry for the day's sales numbers would be calling the function, but there would be no arguments to pass in. Terry does return a value, though: the day's sales. There might also be jobs that require no input and return no value; for example, Lola's job might be to lock up at the end of the day. No input might be needed, and no information is returned, but a function is performed.

We've spent a very long time now just talking about functions in abstract terms. That's because functions are not just another control structure; they're an entire paradigm of programming. They're a different way to think about structuring the

code that we write. So, it's important to understand not just the syntax, but also the philosophy behind functional programming. The main takeaways of what we've discussed so far are:

- Functions are like miniature programs, with their own input and output, for tasks we need to do a lot.
- Functions are defined by a header that states their name and the input they expect, like a `add()` function having two parameters, addend1 and addend2.
- Functions possess a body that dictates what to do with that input and what output to return.
- Complicated programs are built out of lots and lots of smaller functions.

3. Simple Functions in Python

Functions are a bit of a chicken-and-egg problem when learning to program. You must have defined a function to be able to call it, but you must be able to call a function to test defining it. So, where do we start? Fortunately, we've seen some examples of function calls already: `len()`, `str()`, and `random.randint()` for example. So, we've seen what's involved in a function call. Let's get started defining functions. First, we'll define a very general function with no input or return, a function that just performs a task. Then, next lesson we'll define a function that does return some value. Finally, we'll define a function that has some parameters *and* returns some value.

The Function Definition

Imagine you're writing some code that will be used by an online store that does business in multiple countries. That means the web site must be able to show prices with local currency symbols. You generally only have your own local currency on your keyboard, though, which means accessing the other symbols could be frustrating. So, we want to write a function that will print out a certain local currency's symbol. Let's go with the symbol for Japanese yen, ¥.

To start with, let's just write a simple function `printYen()` that will print the character ¥; no input, no output. How do we do that?

#	TheFunctionDefinition.py	Output
1	`#Defines the function "printYen"`	¥5
2	`def printYen():`	
3	`    #Prints "¥", without the new line afterward`	
4	`    print("¥", end='')`	
5		
6	`#Calls the function "printYen"`	
7	`printYen()`	
8	`print(5)`	
9		

Figure 3.4.3

Figure 3.4.3 shows the code to print the character ¥. Let's trace through it. Line 2 is the function header. It starts with the keyword `def`. The keyword `def` is a reserved word in Python for creating functions. Then, we have the function name, `printYen()`. Then, we have an open and close parenthesis. If we were defining parameters, they would go in these parentheses; even if we have no parameters, however, we have to leave these parentheses because they tell Python explicitly that the function needs no input. This is also why we often show function names with parentheses after them when referring to them; for example, "the `len()` function returns the length of a list." Then, as with all other control structures, we end with a colon, telling Python to expect an indented line.

Inside the body (the indented part, lines 3 and 4) of the function, we write the one line: `print("¥")`. Actually, we're doing something extra: the `, end = ""` inside the print statement tells Python not to create a new line after printing like it usually does. We'll talk about why later; for now, just know that including this little extra bit prevents Python from going to the next line, so the next thing printed will be on the same line. This is why 5 appears on the same line as ¥ in the output.

The Function Call

Then still in Figure 3.4.3, we're back outside the function again. On line 7, we call the function: `printYen()`. Line 2 told the computer, "Hey, there exists a function called `printYen()`, so when it gets called, come up here!" So, when `printYen()` is called on line 7, execution of the program goes to line 3, the first line inside the function `printYen()`. The computer runs line 3 and prints the symbol. That's the end of the function, so it comes back out to the main program. Line 7 is now done, so it runs line 8, and prints the number 5. The result: the computer prints "¥5." The number 5 is on the same line as the symbol ¥ because of the `, end = ""` part of line 7.

Notice a couple of things here. First, as we've said before, Python starts at line 1 and runs the lines of code one by one. That means, on line 2, it runs, `def printYen():`. What does running that line mean? Running that line on its own means remembering, "Hey, there exists a function called `printYen()`." The function doesn't run; running this line merely *defines* the function. It's the equivalent of you being told, "Hey, Addison adds two numbers": you haven't asked him to add two numbers yet, but you know he's there. The program then skips any lines inside the function because the function hasn't been called; it's just been defined.

Then the program runs line 7. Line 7 says, "Hey, remember that function `printYen()`? Run it!" Now the program jumps to line 3, the first line inside `printYen()`, and runs it. The body of the code is run when the function is *called*, not when it's *defined*. After it runs line 3, that's the end of the `printYen()` method, so execution returns to where it left off in the main program. Line 7 is now done, so it runs line 8.

Notice that we've touched on the idea of scope for methods in this. The scope of the `printYen()` method begins on line 2 when it's defined. So, anything after line 2 can see `printYen()`. Had we tried to call `printYen()` on line 1, it would have failed because `printYen()` wasn't yet defined. This is different from many other languages, where the language goes through and compiles functions before trying to run the code.

4. Functions with Returns and Parameters in Python

So, we've now seen the general syntax for defining a function: the keyword `def`, a name for the function, parentheses, a colon, then the function body. Now let's add two layers of complexity to this: a return statement and some parameters.

A Function with a Return

The function we defined in Figure 3.4.3 is of limited usefulness. Chances are, we're not usually trying to print the ¥ symbol to the console; we usually want to use it in some other program we're writing. So, instead of printing it directly, we want a function that will *return* that symbol.

In Figure 3.4.4, we've changed line 4. Instead of printing ¥, we return it. What does this mean? It means that wherever we call `returnYen()`, it gets replaced by the string "¥". That was our goal, after all: to make it so we didn't have to keep going and finding the symbol. Now, instead, whenever we need to use it we can just call `returnYen()`.

#	AFunctionwithaReturn-1.py	Output
1	`#Defines the function "returnYen"`	¥5
2	`def returnYen():`	
3	`    #Returns "¥"`	
4	`    return "¥"`	
5		
6	`#Prints the output of returnYen()`	
7	`print(returnYen(), end = "")`	
8	`print(5)`	
9		

Figure 3.4.4

Let's trace through how this code executes. Just like before, when the computer runs line 2, it loads the knowledge of the existence of `returnYen()` into memory. Now it knows that if `returnYen()` is called, it should come back here and run this function's body. Then, it skips line 4 because it's in the body of the function, and the function hasn't been called yet. Then, it runs line 7. Line 7 asks to print the output of `returnYen()` (again, without the linebreak). To do this, it has to call `returnYen()`. So, it does. Execution jumps to line 4, the body of `returnYen()`. Line 4 says return "¥". This means that "¥" is sent back to the main program to replace the function call. `returnYen()` is replaced by the string "¥". So, this line becomes `print("¥", end="")`, and so the program just prints ¥. Then, line 8 runs and prints 5.

The outcome is exactly the same, but notice that this design means we could have used `returnYen()` in other ways. Specifically, we could cut the two print statements down to one, as shown in Figure 3.4.5.

#	AFunctionwithaReturn-2.py	Output
1	`#Defines the function "returnYen"`	¥5
2	`def returnYen():`	
3	`    #Returns "¥"`	
4	`    return "¥"`	
5		
6	`#Prints the output of returnYen(), then 5`	
7	`print(returnYen(), 5, sep = "")`	
8		

Figure 3.4.5

Because `returnYen()` returns "¥" to replace the function call instead of just printing "¥" by itself, we can use it in other lines, too. In Figure 3.4.5, we're using it to print the same text as before, but imagine using this to generate the price tag to be put into a web site, or the price listing for a pricing database. In those cases, having this kind of access is valuable.

A Function with a Parameter

The function from Figure 3.4.5 just returns the "¥" symbol. However, are we ever going to use this symbol without an amount of currency following it? …well, we might, but for a moment let's pretend we won't. So, instead of forcing the main program to always add the amount separately, why don't we instead make this function simply return the string version of the amount of currency preceded by the ¥ symbol? To do that, we need to send in the amount of currency to use as input into the function, as shown in Figure 3.4.6.

We've updated the function definition and put some new information in the header: `amount`, in parentheses. `amount` is a parameter of this function. This is like creating a variable specifically for this function: inside the function body, we can refer to `amount` as a variable. The value for this variable is the argument passed into the function when it is called. So, when the computer runs line 2 and defines

#	AFunctionwithaParameter-1.py	Output
1	`#Defines the function "returnYenAmount"`	¥5
2	`def returnYenAmount(amount):`	
3	`    #Returns "¥" with the amount`	
4	`    return "¥" + str(amount)`	
5		
6	`#Prints the output of returnYenAmount(5)`	
7	`print(returnYenAmount(5))`	
8		

Figure 3.4.6

the function, it defines it with the knowledge, "Hey, to call this function, you need to pass in one argument."

As before, execution then skips line 6 because the function `returnYenAmount()` is being defined, not called. So, execution proceeds to line 7, where it prints `returnYenAmount(5)`. 5 is the argument being passed into the function. So, the value 5 is assigned to the variable `amount`. This is just as if the first line of the function was `amount = 5`. Execution jumps to the body of `returnYenAmount`, which returns `"¥" + str(amount)`. Since `amount` is 5, this resolves to "¥5". So, the string "¥5" is returned and replaces `returnYenAmount(5)` in line 7. This line becomes `print("¥5")`, so the computer does so.

So far, all our arguments have been values themselves, but we can (and usually will) use variables themselves as arguments. In this case, the value of the variable becomes the value of the parameter. For example, let's convert this to a user-facing program that asks the user to input the amount they want added to the currency symbol.

#	AFunctionwithaParameter-2.py	Output
1	`#Defines the function "returnYenAmount"`	Enter the amount: 5
2	`def returnYenAmount(amount):`	¥5
3	`    #Returns "¥"`	
4	`    return "¥" + str(amount)`	
5		
6	`inputAmount = int(input("Enter the amount: "))`	
7	`#Prints the output of returnYenAmount(inputAmount)`	
8	`print(returnYenAmount(inputAmount))`	
9		

Figure 3.4.7

On line 6 of Figure 3.4.7, the user enters a number, when prompted by line 6 and it's stored in the variable `inputAmount`. The variable `inputAmount` is then used as the argument to the function `returnYenAmount`. This assigns the value of `inputAmount` to the parameter `amount`, to be used in the function `returnYenAmount`. So, in this case, the user types in 10, which is stored in `inputAmount`. `inputAmount` is then passed into `returnYenAmount` as an argument, meaning the value of `inputAmount` is assigned to the parameter `amount`. Now, `amount` in the function `returnYenAmount` has the value 10, so when `return "¥" + str(amount)` is called, it becomes `return "¥" + str(10)`. This resolves to `return "¥10"`, and so the function call is replaced with "¥10".

A Function with Multiple Parameters

If a function is defined with multiple inputs, then it is assumed that the order of the arguments in the function call matches the order of parameters in the function definition. Let's make our running example from Figure 3.4.6 a little more complicated to check this out. Let's write a function that doesn't just handle the ¥ symbol, but the £ and $ symbols as well.

In Figure 3.4.8, we've revised the function definition to change the name (now `currencyAmount`) and have two parameters: `currency` and `amount`. The body of the function is a series of conditionals that checks to see if the currency is one of three expected types: JPY for Japanese yen, USD for US dollars, or GBP for British

#	AFunctionwithMultipleParameters.py	Output
1	`#Defines the function "currencyAmount"`	£5
2	`def currencyAmount(currency, amount):`	
3	`    if currency == "JPY":`	
4	`        return "¥" + str(amount)`	
5	`    elif currency == "USD":`	
6	`        return "$" + str(amount)`	
7	`    elif currency == "GBP":`	
8	`        return "£" + str(amount)`	
9	`    else:`	
10	`        return str(amount)`	
11		
12	`#Prints the output of currencyAmount("GBP", 5)`	
13	`print(currencyAmount("GBP", 5))`	
14		

Figure 3.4.8

pounds. If `currency` isn't one of these, it simply returns `amount` as a string. Notice here how we have multiple returns, but only one will be reachable at a time based on the value of `currency`.

When we call the function, we now use two arguments: "GBP" and 5. The order of the arguments matches the order of the parameters, so the parameter `currency` receives the argument "GBP" and the parameter `amount` receives the argument 5. The body of the function runs, finding that `currency == "GBP"` is `True`, so it returns `"£" + str(amount)`. This resolves to "£5", so it returns the string "£5", which replaces the function call. So, line 11 resolves to `print("£5")`, so it does so.

Before we move on, it is also worth calling attention to the subtle but powerful structure here. We have the function `print()`, and we have the function `currencyAmount()`. The function `print()` takes a string as input and prints it. How, then, can `print()` take `currencyAmount()`, a function, as input? Remember, when we call a function, we effectively replace the function with its output, then run the line of code again. When we run `print(currencyAmount("GBP", 5))`, the computer starts with the innermost set of parentheses (which may be attached to a function) and evaluates it. Here, that's `currencyAmount("GBP", 5)`. After evaluating that function call, it is replaced by the string "£5". Now, the computer tries the line again: `print("£5")`. There is nothing left to evaluate, so it just runs this line as-is. There are two important takeaways here: (a) function calls are effectively evaluated and replaced by whatever they output, and (b) as a result, we can use function calls as arguments to other function calls as long as they will output the right kind of data. Later, we'll even look at functions that use *themselves* as arguments, like `reprint(reprint("WHAT?!"))`.

5. Common Function Errors in Python

We've covered how functions are defined and called; now let's discuss some of the function-specific things that can go wrong. All of the errors we've described in the past can apply to functions, too. For example, if we have a parameter to a function (like `amount` in `currencyAmount()`) that is treated as an integer inside the function but we pass in a string, then we get the same error as if we had just tried to do math on a string in a regular program as well. The function doesn't change that. However, there are a couple of function-specific errors we can anticipate.

Parameter Mismatch

When we define a function, part of that definition is the declaration of how many arguments should be passed in. This comes in the form of the parameter list. Our `currencyAmount()` function above had two parameters: `currency` and `amount`.

To call `currencyAmount()`, the program had to supply arguments for both parameters (that is, values for both variables). What happens if we don't?

ParameterMismatch.py

```
1  #Defines function "currencyAmount"
2  def currencyAmount(currency, amount):
3      if currency == "JPY":
4          return "¥" + str(amount)
5      elif currency == "USD":
6          return "$" + str(amount)
7      elif currency == "GBP":
8          return "£" + str(amount)
9      else:
10         return str(amount)
11
12 #Prints currencyAmount(5)'s output
13 print(currencyAmount(5))
14
```

Output

```
Traceback (most recent call last):
  File "ParameterMismatch.py", line 13
    print(currencyAmount(5))
TypeError: currencyAmount() missing 1
required positional argument: 'amount'
```

Figure 3.4.9

Figure 3.4.9 shows this error in action. The error is `TypeError`, which we've seen before. The feedback from the error gives us plenty of information: "missing 1 required positional argument: 'amount.'" This basically says, "no value was given for amount." The only reason `amount` specifically didn't receive a value is because, as we described before, the arguments are assumed to go in the same order as the parameters. The first argument is 5, so it's assigned to the first parameter, `currency`. This is wrong, of course, but that's the way the computer interprets it. It then looks for an argument for `amount`, but doesn't find one. So, it throws up that error.

Scope Error

We've covered the scope of functions themselves; basically, a function must be defined before it is called in some code, the same way a variable had to be created before it was used. However, what about the variables we use inside functions? What are their scopes?

ScopeError.py

```
1  #Defines the function "currencyAmount"
2  def currencyAmount(currency, amount):
3      if currency == "JPY":
4          resultString = "¥" + str(amount)
5      elif currency == "USD":
6          resultString = "$" + str(amount)
7      elif currency == "GBP":
8          resultString = "£" + str(amount)
9      else:
10         resultString = str(amount)
11
12 #Runs currencyAmount(5)
13 currencyAmount("GBP", 5)
14 print(resultString)
15
```

Output

```
Traceback (most recent call last):
  File "ScopeError", line 14
    print(resultString)
NameError: name 'resultString' is
not defined
```

Figure 3.4.10

The code in Figure 3.4.10 tests that out. Instead of returning something from `currencyAmount`, here we just set the result equal to `resultString`. If the scope of a variable created inside a function extended to after the function has run, then this code should work: by the time line 12 runs, line 8 will have run, and `resultString` will have been created. Instead, though, we receive an error saying that `resultString` is not defined. Variables defined inside functions only exist inside those functions. The same goes for parameters: we can't use `currency` or `amount` outside of `currencyAmount()` because these are variables specifically for that function.

This is the first major exception we've encountered to our general rule that the scope of a variable is the remainder of the program's execution. Variables created

inside functions only exist inside that function, and only until the function is done running; if we called `currencyAmount` twice, it would be as if `resultString` was never created. In this way, functions really are like little programs; the scope of a variable is the remainder of the program, and a function is like its own little program, so the scope of a variable created inside a function is the remainder of the function. Note that this doesn't interfere with the scope of other variables outside the function; a variable declared before a function call is still available after the function call, but not inside the function that is called.

6. Advanced Python Functions

Different languages extend the general idea of functions in different ways. In Python, there are a couple of advanced details regarding functions that are worth covering. In fact, Python functions can get very complex, but we're most interested in keyword parameters. Keyword parameters are not terribly common in what you'll write, but they do add a lot of power to your toolbox.

Using Keyword Parameters

Keyword Parameters
A special kind of optional parameter to which the program may choose to assign an argument during a function call, or may ignore. Typically, keyword parameters have a default value that is used if it is not overridden by a function call.

To understand **keyword parameters**, remember two things we've said. First, remember that we said Python assumes that arguments come in the order that parameters are defined in a function definition. Second, remember that at one point, we included a weird extra bit of code in a print statement: we said `print("¥", end = "")`. I promised to come back to this later, and now is later!

For required parameters, what we've said so far is true with regard to assuming arguments come in the order that parameters are defined. However, sometimes that can be a bit limiting. One instance of that is that oftentimes, we might have parameters for which we want to *assume* one value, but allow the program to override this if they want.

The `print()` function is a good example of this. The `print()` function takes as input one or more strings to print in order. We haven't seen it printing multiple strings before, but we can see it now in Figure 3.4.11.

#	UsingKeywordParameters-1.py	Output
1	`print("A", "B", "C")`	A B C
2	`print("D", "E", "F")`	D E F
3		

Figure 3.4.11

When we give `print()` multiple strings, it prints them one at a time in the order they're given, separated by spaces. Each `print()` function call automatically ends the line, which is why D E F appears on a different line from A B C; `print()` puts a space between each character and a new line at the end of each line. Technically, new line is just a character that isn't shown, but instead tells the computer to print the next text on a new line.

The `print()` function assumes we want spaces between the individual strings, and a new line at the end of each `print()` call. What if we don't want that? What if we want no spaces between strings, and no new line at the end of each line? Then, we use keyword parameters, as shown in Figure 3.4.12.

#	UsingKeywordParameters-2.py	Output
1	`print("A", "B", "C", sep = "", end = "")`	ABCDEF
2	`print("D", "E", "F", sep = "", end = "")`	
3		

Figure 3.4.12

A keyword parameter is a parameter that has an assumed value, but that the function call can override. Overriding it looks like declaring a variable: we take the name of the parameter, and assign it to a different value. Here, the parameter `sep` (for "separator") is used to define what character will be used between each string in the `print()` call. It is assumed to be a space, but if we include the argument `sep = ""`, it is assigned an empty value. Then, the print statement puts nothing in between the characters. Similarly, the parameter `end` holds what character to put at the end of the line. By passing the argument `end = ""`, we override that with a blank character, forcing everything onto one line.

If this is confusing, let's try to override `sep` and `end` with visible characters in Figure 3.4.13. This looks messier, but it might be easier to understand. Including the argument `sep = "#"` in the first line replaces those spaces from the original output with # symbols. Including the argument `end = "?"` replaces the new line character with a question mark. We can also see here how using different symbols in line 2 causes different symbols to appear among the D, E, and F.

#	UsingKeywordParameters-3.py	Output
1	`print("A", "B", "C", sep = "#", end = "?")`	A#B#C?D%E%F!
2	`print("D", "E", "F", sep = "%", end = "!")`	
3		

Figure 3.4.13

These are keyword parameters. We don't assume the program will define them because oftentimes they won't; but we want to give the program the ability to define them if need be. If we simply list them as normal parameters, they become required, and throw up that `TypeError` from Figure 3.4.9 if the program doesn't specify them. We don't want the programmer to have to specify what separator to use every time they use the `print()` function, though, since most of the time it will be the same. So, we use keyword parameters, which give the program a way to specify alternate values when needed, but a way to ignore them in favor of default values when alternate values are not needed.

Creating Keyword Parameters

So, how do we create keyword parameters? In our function declaration, we include the parameter name, but assign it a value the same way we do when overriding it. For example, let's assume we want our `currencyAmount` function to assume US dollars unless the program specifies otherwise.

In Figure 3.4.14, we've revised our function declaration to say `currency = "USD"` in the parameter list instead of just `currency`. We switched the order because keyword parameters must come after regular (also called positional) parameters.

#	CreatingKeywordParameters.py	Output
1	`#Defines the function "currencyAmount"`	$5
2	`def currencyAmount(amount, currency = "USD"):`	£5
3	`    if currency == "JPY":`	
4	`        return "¥" + str(amount)`	
5	`    elif currency == "USD":`	
6	`        return "$" + str(amount)`	
7	`    elif currency == "GBP":`	
8	`        return "£" + str(amount)`	
9	`    else:`	
10	`        return str(amount)`	
11		
12	`#Prints the output of currencyAmount(5)`	
13	`print(currencyAmount(5))`	
14	`#Prints the output of currencyAmount(5, currency = "GBP")`	
15	`print(currencyAmount(5, currency = "GBP"))`	
16		

Figure 3.4.14

We assigned `currency` a value of "USD", which basically says, "Assume currency is 'USD' unless the function call says otherwise." Beyond that, the function definition is the same.

Now, we call `currencyAmount` in two different ways. On line 13, we just give it the argument 5. Based on the position, the computer assumes 5 is the `amount`. No argument is given for `currency`, so it maintains the assumed value of 'USD' and runs accordingly. On line 15 we call `currencyAmount`, we specifically override the parameter `currency` with the value "GBP", and as a result, the code uses "GBP" as the value for that variable.

7. Functions and Turtles

Functions give us a powerful new way to add new functionality to our turtle project while keeping it somewhat organized. To close out our lesson on functions, let's do two things: first, let's take our prior work on a shape command and spin it off into a function, and second, let's create a new function for something even more interesting.

The Shape Function

While our code to draw a custom shape wasn't too long (five lines), it was still a good bit longer than other commands, which were only two lines. If we had commands with even longer code, this could get big and disorganized fast. Functions give us a way to keep things more organized by separating out different areas of the program.

To start, let's take that shape command's code and make it a separate function. There are two ways we could do this: we could continue to get the user's input in the main program, then pass it as input into the function to draw the shape. Or, we could just combine all that code into the single function and call it; then, the user input would be entered during the function call. The benefit of the first approach is that it allows us to draw a shape with `numSides` and `sideLength` regardless of whether the user entered these or if they came from somewhere else (like reading them from a file), so that's probably the better design.

TheShapeFunction.py shows the code to do this the first way. Our changes are relatively simple:

- Defined a function `drawShape()` on line 5 at the start with parameters `numSides` and `sideLength` (still after the import statement, though).
- Copied the three lines of code (from the 'shape' branch) after getting the `sideLength` into the function `drawShape()`, line 6 through 8.
- Called the function `drawShape()` with the user's inputted `numSides` and `sideLength` as arguments on line 35. Note that here, the arguments are variables, and their names (numSides and sideLength) *happen* to match the parameter names (also numSides and sideLength), but this doesn't have to be the case.

So, now we've successfully spun `drawShape()` off into its own function. In the process, we've shortened the code inside the main reasoning of the program down to just the input lines and one line to actually do the drawing, like the other commands. One of the benefits of this is that it keeps our code more organized. The bigger benefit, though, is that it lets us call that function in more flexible ways. Let's see how.

The Snowflake Function

To see this in action, we're going to write a function that will call `drawShape()` multiple times. We couldn't do that previously: we would have had to copy the `for` loop into a different area of the program to use it in a different place. Because it's a function now, though, we can call it wherever we want.

So, in TheSnowflakeFunction.py, I've created another command: snowflake (line 46). Snowflake asks the user for three numbers: the number of sides (line 48), the length of each side (line 50), and the rotation angle (line 52). It stores these in the corresponding variables, and passes them as arguments into the `drawSnowflake()` function (line 54), jumping to line 13. The `drawSnowflake()` function first uses the `rotationAngle` to figure out how many times it will have to rotate to cover a full 360 degrees (line 14). It then runs a `for` loop that many times (line 15). Each time the `for` loop runs, it calls (line 16) `drawShape()` with the given number of sides and side length, jumping to line 5. Once the shape is drawn (repeating lines 6 and 7 several times), the execution jumps back to line 17. There, it rotates the turtle by the rotation angle, and repeats drawing the shape.

The result? A snowflake effect, most of the time anyway. The same shape is drawn repeatedly, rotating a little bit about the center point each time. This can lead to some radically different results based on the number of sides and the rotation angle; try out some different combinations to see. Because of the `while` loop, we can use this to draw multiple things one over the other. Note that this is this easy (in terms of number of lines of code in `drawSnowflake()`, 5) because we first put `drawShape()` into its own function. That allowed us to create `drawSnowflake()`, which calls `drawShape()`. That's the power of functional programming: not only does it keep our code more organized but it also lets us reuse and recycle segments of code in varied ways.

CHAPTER

3.5

Error Handling

1. What Is Exception Handling?

Early in our conversations, we covered the idea of errors. Errors were specific times when our code tried to do something it wasn't able to do. For example, we can't divide by zero: telling our code to divide by zero would cause an error. Similarly, we can't open a file that doesn't exist: telling our code to open a file that doesn't exist would cause an error. In some languages, these are also referred to as exceptions; and in some languages, there are subtle differences between errors and exceptions. Generally, we'll use the words interchangeably until we get to your language-specific material.

So far, we've most often used errors for debugging: when an error arises, we know we need to go and figure out what caused it and prevent it from happening in the future. However, that's not the only purpose of errors. Sometimes, instead of preventing errors, we want to use the fact that an error arises to direct or control our program. In other words: some errors might be expected and even purposeful, and when they arise our program should know how to deal with them.

"Catching" Errors

We call this "**catching**" an error. An **uncaught error** (also called an unhandled error) will crash our program. A caught error will let our program keep running, and we can add code that specifically runs if an error was caught. We can even add different blocks of code that react to different types of errors.

Let's take an example. Imagine we asked the user to put in a list of numbers. We would then run a `for-each` loop over all the numbers, add them up, and divide the sum by the length of the list of numbers. What happens if the user doesn't put in any numbers, though? That's not a problem for our `for` loop: it still runs for "each" item in the list, there just aren't any. It would be like going to the grocery store with an empty shopping list: it's silly, yes, but it wouldn't cause your reasoning to crash.

However, what happens when it reaches the end of the loop? It attempts to divide the sum, 0, by the length of the list, also 0. 0 divided by 0 triggers a divide-by-zero error, which would crash our program. The user shouldn't be able to crash the program, though, so we need to avoid this. There are two ways we can avoid this error: one, we could simply run a conditional before calculating the average to make sure the length of the list was not zero. If the length is not zero, we calculate the average; else, we show a message to the user saying, "You can't average an empty list!"

There's nothing wrong with that method, but let's talk about the other approach: catching the error. We can look at this code and know: the only possible way to encounter a divide-by-zero error is if the list was empty. So, if a divide by zero error arises, that must mean the list was empty. So, instead of checking if the list was empty first before trying to calculate the average, we can instead just tell the computer, "Hey, try to calculate the sum, but don't crash if you can't: instead, just tell the user they can't average an empty list." This is catching the error: telling the computer not to crash if the error is encountered, as well as giving it some step to take instead.

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Illustrate the purpose of error handling, including `try`, `catch`, and `finally` blocks;
- Correctly use Python's error handling structures and integrate them with other control structures;
- Implement error handling structures in their programs to catch and respond to invalid user input.

Catching Errors
Using error handling to prevent a program from crashing when an error is encountered.

Uncaught Error
An error that is not handled by error handling code, and thus usually forces the program to crash.

When to Catch Errors

In the example above, we had two pretty equivalent options: we could use a conditional to check if an error would occur in advance, or we could catch errors after they already occurred. What's the usefulness of catching errors if they could be avoided with conditionals?

First, part of this is more about our thought process when creating programs. Catching errors lets us first emphasize the code itself, and later think about what errors might arise. You don't want to do that in big programs, but for individual functions, that can help you focus on the actual reasoning of the function first. Along these same lines, when catching errors, some of the code will run until an error arises, whereas with the `if` statement, either all the code will run or not of it will. That can be useful, too.

Second, catching errors can create more organized code. If you have a segment of code that could have multiple expected errors, then you would need either (a) one long conditional that checks every possible error, or (b) a series of conditionals each checking a different error. Either of these can get messy. When you're catching errors, you can generally just wrap up one long code block and list the errors that could arise at the end.

Third and perhaps most importantly, if you're writing programs for people to actually use (which, presumably, is the goal of learning to program), you never want a program to crash on the user. The goal is error-free code, but in big programs, that's nearly impossible; big applications such as games or operating systems may handle dozens of errors per second. Many of these are expected, but some aren't. Catching errors localizes the damage even of unexpected errors. For example, if you were writing a document in a word processor and it hit an error loading a font, you wouldn't want the program to just crash; you'd want it to say, "Well, it's better to not show that font than to crash altogether." Error handling allows us to write programs where, even if there are unanticipated errors, their damage is localized and minimized.

2. Try-Catch-Finally

We're covering error handling in the control structures unit of this course because the actual structure of error-handling is, itself, a control structure. There are three common structures for error-handling: the `try`, `catch`, and `finally`.

`Try`
A control structure that sets aside a block of code in which an error might occur so that the computer will look for error handling capabilities.

The Try Block

The `try` block of the error-handling control structure is the simple one. It marks off the code in which an error is anticipated to arise. On its own, it doesn't actually do very much; it's more of a marker, so the computer knows what code might have its errors handled later on. The computer will run the code in the `try` block until an error arises; if an error arises, the code will skip the rest of the code in the `try` block and jump to the code in the next block, the `catch` block.

Earlier we mentioned that error handling is like a conditional statement. We could handle errors with a conditional by saying, "If an error is going to arise, don't run this code; else, do run it." In that structure, we would put the code we actually want to run in the `else` portion of the structure. The `try` block is thus similar to the `else` block: it's a block of code marked off to run if some other block didn't run. However, the `try` block is different in that it will always *start* to run, and only stop when an error is encountered.

`Catch`
A control structure that designates what error it anticipates in a `try` block and provides the code to execute if that error arises.

The Catch Block

When the computer encounters a `try` block, it makes a "mental" note that if an error occurs during the block, it should jump forward to the `catch` block. The `catch`

block contains the code the computer should run if an expected error was encountered in the `try` blcok.

The `catch` block has one additional detail declared with it: the type of error to be expected. We can tell the computer exactly what kind of error to catch. For example, we could write code that would catch a divide-by-zero error only, but would let any other errors through and crash the program. We could also design our code to have multiple catch blocks, allowing it to react differently to different kinds of errors; for example, it might warn the user about averaging an empty list on a divide-by-zero error, and it might send a message to the programmers if a different error was encountered. In some languages, we can even skip the `catch` block altogether: a `try` without a `catch` just tells the computer not to crash if the code inside the `try` block raises an error.

The `catch` block is where the bulk of the interesting reasoning in error handling occurs. If the error was expected in some way, the `catch` block might tell the user why the error occurred and how it can be fixed. If the error was not specifically expected, the `catch` block could print the reason the error occurred to the console, or trigger an error report to be submitted to the developers. Any type of code can be placed into both a `try` and `catch` block, so we could do complex reasoning like determining if the user has opted in to reporting errors.

The Finally Block

Finally (pun intended), some languages have a `finally` **block**. The `finally` block contains code that should be executed after the code in the `try` block whether it succeeded or not. If the code in the `try` block ran without errors, then execution will jump to the `finally` block when the `try` block is done. If the code in the `try` block hit an error, execution will (in most languages) run what is in the `catch` block next, and then will always run the code in the `finally` block. For this reason, we generally want to be very confident in what we put in the `finally` block since, if it raises an error, too, we aren't prepared to catch it.

`Finally`
A control structure that designates some code to run after a `try` and `catch` structure regardless of whether or not an error arose.

The `finally` block is typically used for code that absolutely needs to run, even if other things have gone wrong. For example, imagine some code to close an application. If there is code in the close function that can raise an error, then the application can't exit at all! We would want to enclose the close function in a `try-catch-finally` structure, and in the `finally` block, we would want to ensure that the application really does close.

3. Try and Except in Python

So far, we've been avoiding errors in our code. Now, we get to add them intentionally, so that we can learn how to catch them. We'll start just by using `try` blocks to prevent code from crashing if errors are encountered; then, we'll catch these errors and react accordingly.

The Try Statement

To experiment with catching errors, we need an error to catch. Let's start with something simple: trying to convert a non-numeric string to an integer. What happens if we do this without any error handling?

In Figure 3.5.1, we're given an error on line 3, the attempted type conversion. Notice that line 2 still runs and prints this statement, but line 5 does not run because the error arises on line 3. For now, all we want to do is prevent our code from crashing when it tries (see why it's called a `try` statement?) to do perform this type conversion. However, Python is not a language that allows `try` without `catch` (or alternatively, `finally`), so we need to include both. We'll talk more extensively about the `catch` in the next section.

#	TheTryStatement-1.py	Output
1	`myString = "This string is not a number!"`	`Converting myString to int...`
2	`print("Converting myString to int...")`	`Traceback (most recent call last):`
3	`myInt = int(myString)`	`  File "...", line 3`
4	`print(myInt)`	`    myInt = int(myString)`
5	`print("Done!")`	`ValueError: invalid literal for int() with base 10: 'This string is not a number!'`
6		
7		
8		
9		
10		

Figure 3.5.1

In Figure 3.5.2, we've added the line `try:` on line 3, following the comment. Like all other control structures, the colon indicates a code block for this statement to control, and all indented code following this line falls within this block. So, the computer starts executing the lines of code in the `try` block. It runs line 4, then attempts to run line 5. Line 5 still generates an error. Instead of crashing as before, the computer instead knows it's inside a `try` block, and so it should look to see if the `catch` on line 8 actually accepts this error; `except` is Python's word to indicate a `catch` block. As we'll see next section, this does catch the error, so it jumps into line 9, the first line inside the `catch` block. Line 9 simply tells the computer to continue with the keyword `pass`, and so it runs line 10, then closes. We haven't resolved the error in line 5, but we've allowed our program to recover from it.

#	TheTryStatement-2.py	Output
1	`myString = "This string is not a number!"`	`Converting myString to int...`
2	`#Run the code below until an error occurs`	`Done!`
3	`try:`	
4	`    print("Converting myString to int...")`	
5	`    myInt = int(myString)`	
6	`    print(myInt)`	
7	`#If an error occurs, jump to here`	
8	`except:`	
9	`    pass`	
10	`print("Done!")`	
11		

Figure 3.5.2

This is the essence of the `try` block. If an error occurs inside of it, the computer checks if that type of error is caught. If so, it jumps into the `catch` block and runs the code there, then continues as if no error occurred. If not, it crashes as usual.

Catching Any Error

In Python, the `catch` block (which we'll now call the `except` block) starts with the keyword `except`. In speech, we can think of this as saying, "try this, except if this error happens…" We had an `except` block in the previous code by necessity: Python won't allow a `try` block without an `except` block. However, it didn't do anything; we used the keyword `pass` to skip on to the next line of code. Let's now make it actually do something.

As we can see in Figure 3.5.3, adding the print statement on line 9 into the body of the `except` block runs it when the code hits an error. The code runs until line 5 and hits the same error as before. It checks line 8 to see if the error was properly handled. It is, so it runs line 9, then proceeds to line 10. The code in line 9 runs if any error is encountered.

#	CatchingAnyError.py	Output
1	`myString = "This string is not a number!"`	Converting myString to int...
2	`#Run the code below until an error occurs`	Can't convert; myString not a number.
3	`try:`	Done!
4	`    print("Converting myString to int...")`	
5	`    myInt = int(myString)`	
6	`    print(myInt)`	
7	`#If an error occurs, jump to here`	
8	`except:`	
9	`    print("Can't convert; myString not a number.")`	
10	`print("Done!")`	
11		

Figure 3.5.3

Catching a Specific Error

Note that the way we've written Figure 3.5.3, *any* error will be caught by this `except` block. If we don't specify a type of error to catch, the `except` block will catch any error. So, take a look at what happens if we add a different error earlier in the `try` block in Figure 3.5.4.

#	CatchingASpecificError-1.py	Output
1	`myString = "This string is not a number!"`	Converting myString to int...
2	`#Run the code below until an error occurs`	Can't convert; myString not a number.
3	`try:`	Done!
4	`    print("Converting myString to int...")`	
5	`    print("String #" + 1 + ": " + myString)`	
6	`    myInt = int(myString)`	
7	`    print(myInt)`	
8	`#If an error occurs, jump to here`	
9	`except:`	
10	`    print("Can't convert; myString not a number.")`	
11	`print("Done!")`	
12		

Figure 3.5.4

Line 5 here has a different error: it tries to cast an integer to a string implicitly instead of explicitly. If you don't understand why this is an error, glance back at Chapter 2.2, but you're also safe to continue as long as you remember that line 5, as written, will cause an error. To write line 5 successfully, we would need to write `str(1)` or "1", not just the number 1 (unless we use commas instead of addition signs). However, our error handling code was written to catch *any* error, and so this error is caught, too. The message printed by the `except` block is inaccurate because this error clearly wasn't the one we expected. To avoid this, we should instead catch a *specific* error. Line 6 would generate a `ValueError`, so if that's the error we expect, let's catch that one specifically, as shown in Figure 3.5.5.

#	CatchingASpecificError-2.py	Output
1	`myString = "This string is not a number!"`	Converting myString to int...
2	`#Run the code below until an error occurs`	Traceback (most recent call last):
3	`try:`	File "...", line 5
4	`    print("Converting myString to int...")`	print("String #" + 1 + ": " + myString)
5	`    print("String #" + 1 + ": " + myString)`	TypeError: Can't convert 'int' object to str implicitly
6	`    myInt = int(myString)`	
7	`    print(myInt)`	
8	`#If a ValueError occurs, jump to here`	
9	`except ValueError:`	
10	`    print("Can't convert; myString not a number.")`	
11	`print("Done!")`	
12		

Figure 3.5.5

The error that arose on line 5 wasn't a `ValueError` (but rather, a `TypeError`), and so this `except` block didn't run. Adding the term `ValueError` after `except` tells it to only run this `except` block for a `ValueError`. Because a `TypeError`

occurs before a `ValueError`, the `try` block stops executing before a `ValueError` occurs. If we remove the line that gives us the `TypeError`, this code still runs as before, as shown in Figure 3.5.6. The error that arose, a `ValueError`, *was* caught by the `except` statement on line 8.

#	CatchingASpecificError-3.py
1	`myString = "This string is not a number!"`
2	`#Run the code below until an error occurs`
3	`try:`
4	`    print("Converting myString to int...")`
5	`    myInt = int(myString)`
6	`    print(myInt)`
7	`#If a ValueError occurs, jump to here`
8	`except ValueError:`
9	`    print("Can't convert; myString not a number.")`
10	`print("Done!")`
11	

Output

```
Converting myString to int...
Can't convert; myString not a number.
Done!
```

Figure 3.5.6

We can also take this a step further: if we're handling unexpected errors or we want to know a little more about why the error arose, we can further extend this to print information about the error itself, as shown in Figure 3.5.7.

#	CatchingASpecificError-4.py
1	`myString = "This string is not a number!"`
2	`#Run the code below until an error occurs`
3	`try:`
4	`    print("Converting myString to int...")`
5	`    myInt = int(myString)`
6	`    print(myInt)`
7	`#If a ValueError occurs, jump to here`
8	`except ValueError as error:`
9	`    print(error)`
10	`print("Done!")`
11	

Output

```
Converting myString to int...
invalid literal for int() with base 10: 'This string is not a number!'
Done!
```

Figure 3.5.7

An error is a data type like integers or strings, and so when we catch it, we can actually grab it as a variable. Adding `as error` to the end of the `except` statement means that inside the `except` block (but not after it, its scope is only inside the `except` block), we can treat the error as a variable, named `error` (or whatever variable name we placed after `as`). We can save it to a file, print it to the console, or access other information about the error. Now instead of printing our prewritten statement, the `except` block prints whatever it would have printed to the console while crashing by printing `error`. We get the same information.

Catching Multiple Specific Errors

This `except` block is a lot like saying, "if a `ValueError` was detected, then…" It's similar to a conditional. Remember, with conditionals, we could also chain together multiple `elif` statements to check multiple conditions. We can do that here, too. Let's bring back the line that triggered a `TypeError`, and catch both in Figure 3.5.8.

#	CatchingMultipleSpecificErrors-1.py
1	`myString = "This string is not a number!"`
2	`#Run the code below until an error occurs`
3	`try:`
4	`    print("Converting myString to int...")`
5	`    print("String #" + 1 + ": " + myString)`
6	`    myInt = int(myString)`
7	`    print(myInt)`
8	`#If an error occurs, check if it's a ValueError`
9	`except ValueError as error:`
10	`    print(error)`
11	`#If an error occurs, check if it's a TypeError`
12	`except TypeError as error:`
13	`    print(error)`
14	`print("Done!")`
15	

Output

```
Converting myString to int...
Can't convert 'int' object to str implicitly
Done!
```

Figure 3.5.8

Just like chaining together `elif` statements, we can chain together except statements, as seen on lines 9 and 12. Here, we catch either a `TypeError` (line 12) or a `ValueError` (line 9). However, if a different kind of error still occurs, it remains uncaught, as shown in Figure 3.5.9.

#	CatchingMultipleSpecificErrors-2.py	Output

1	`myString = "This string is not a number!"`	`Converting myString to int...`
		`Traceback (most recent call last):`
		`  File "...", line 5`
		`  print(1 / 0)`
		`ZeroDivisionError: division by zero`

2	`#Run the code below until an error occurs`	
3	`try:`	
4	`  print("Converting myString to int...")`	
5	`  print(1 / 0)`	
6	`  print("String #" + 1 + ": " + myString)`	
7	`  myInt = int(myString)`	
8	`  print(myInt)`	
9	`#If an error occurs, check if it's a ValueError`	
10	`except ValueError as error:`	
11	`  print(error)`	
12	`#If an error occurs, check if it's a TypeError`	
13	`except TypeError as error:`	
14	`  print(error)`	
15	`print("Done!")`	
16		

Figure 3.5.9

If we add a random division-by-zero on line 5, the code still crashes because `ZeroDivisionError` is not one of the types of errors our `except` statements can handle. Note, however, that if this uncaught type of error were to occur *after* a caught type of error occurs, it would not be a problem because the line causing the uncaught error would never run. The code jumps to the `except` statements the *first* time an error is encountered, and does not come back. So, if we move it to the end, we see the code will end gracefully again, as shown in Figure 3.5.10.

#	CatchingMultipleSpecificErrors-3.py	Output

1	`myString = "This string is not a number!"`	`Converting myString to int...`
		`Can't convert 'int' object to str implicitly`
		`Done!`

2	`#Run the code below until an error occurs`	
3	`try:`	
4	`  print("Converting myString to int...")`	
5	`  print("String #" + 1 + ": " + myString)`	
6	`  myInt = int(myString)`	
7	`  print(myInt)`	
8	`  print(1 / 0)`	
9	`#If an error occurs, check if it's a ValueError`	
10	`except ValueError as error:`	
11	`  print(error)`	
12	`#If an error occurs, check if it's a TypeError`	
13	`except TypeError as error:`	
14	`  print(error)`	
15	`print("Done!")`	
16		

Figure 3.5.10

Note as well that there are a couple more advanced ways we can handle this. We could, for instance, have single `except` blocks that handle multiple kinds of errors, but not *all* kinds of errors. We could also have a series of `except` blocks that handle specific errors, followed by a catch-all `except` block at the end that handles any others. Figure 3.5.11 shows both.

In Figure 3.5.11, line 11 catches either a `TypeError` or a `ValueError`. We can specify multiple errors to catch by listing them in parentheses, separated by commas. We use the same syntax for then assigning the error to a variable. So, in Figure 3.5.11, a `TypeError` occurs first (on line 5), and so the `except` statement on line 11 activates.

#	CatchingMultipleSpecificErrors-4.py	Output
1	`myString = "This string is not a number!"`	`Converting myString to int...`
2	`#Run the code below until an error occurs`	`A ValueError or TypeError occurred.`
3	`try:`	`Done!`
4	`    print("Converting myString to int...")`	
5	`    print("String #" + 1 + ": " + myString)`	
6	`    myInt = int(myString)`	
7	`    print(myInt)`	
8	`    print(1 / 0)`	
9	`#If an error occurs, check if it's a ValueError or`	
10	`#TypeError`	
11	`except (ValueError, TypeError) as error:`	
12	`    print("A ValueError or TypeError occurred.")`	
13	`#Check if some other type of error occurred`	
14	`except Exception as error:`	
15	`    print("Some other type of error occurred.")`	
16	`print("Done!")`	
17		

Figure 3.5.11

In Figure 3.5.12, we moved the divide-by-zero error back up to line 5. As a result, it's encountered first. It's not a `TypeError` or `ValueError`, so the `except` statement in line 11 ignores it, and the computer moves on and checks the `except` statement in line 14. This `except` block catches any other error, so this one is activated, and the computer prints that some other error occurred. Note that this line also shows you how to catch any kind of error and assign it to a variable: instead of specifying an error type like `ValueError`, just use the word `Exception`. The line `except Exception:` works the same as `except:` on its own because `Exception` is the "umbrella" over all the different kinds of errors. Using it, however, lets us add `as error` to the end.

#	CatchingMultipleSpecificErrors-5.py	Output
1	`myString = "This string is not a number!"`	`Converting myString to int...`
2	`#Run the code below until an error occurs`	`Some other type of error occurred.`
3	`try:`	`Done!`
4	`    print("Converting myString to int...")`	
5	`    print(1 / 0)`	
6	`    print("String #" + 1 + ": " + myString)`	
7	`    myInt = int(myString)`	
8	`    print(myInt)`	
9	`#If an error occurs, check if it's a ValueError or`	
10	`#TypeError`	
11	`except (ValueError, TypeError) as error:`	
12	`    print("A ValueError or TypeError occurred.")`	
13	`#Check if some other type of error occurred`	
14	`except Exception as error:`	
15	`    print("Some other type of error occurred.")`	
16	`print("Done!")`	
17		

Figure 3.5.12

4. Else and Finally in Python

At this point, we've tried some code and caught any errors that arose while that code was running. In some languages, that's all there is. In many languages, there's an additional block called `finally`, which runs some code whether an error occurred or not. Python also adds an additional option: remember `else` from conditionals? We can use `else` here as well!

Else for Error Handling

To use an `else` with error handling, we add it after all the `except` blocks. Figure 3.5.13 shows an example of what this looks like.

Note that here, I've changed `myString` to actually hold a number, specifically so that an error does *not* arise. The goal of this code is to show that an `else` block at the end of a sequence of `try` and `except` blocks runs some code if and only if *no* errors arose. Colloquially, we can think of each `except` block as, "if this error occurs, then…," and the `else` block at the end is like the `else` after a series of `elif`

#	ElseforErrorHandling.py	Output
1	`myString = "1"`	Converting myString to int...
2	`#Run the code below until an error occurs`	1
3	`try:`	No errors occurred!
4	`    print("Converting myString to int...")`	Done!
5	`    myInt = int(myString)`	
6	`    print(myInt)`	
7	`#If an error occurs, check if it's a ValueError or`	
8	`#TypeError`	
9	`except (ValueError, TypeError) as error:`	
10	`    print("A ValueError or TypeError occurred.")`	
11	`#Check if some other type of error occurred`	
12	`except Exception as error:`	
13	`    print("Some other type of error occurred.")`	
14	`#If no errors occurred, then do the following`	
15	`else:`	
16	`    print("No errors occurred!")`	
17	`print("Done!")`	
18		

Figure 3.5.13

blocks. The `else` is in reply to the `except` blocks: if any of them run, the `else` block won't run. If none of them run, the `else` block will run.

You might ask, as I did when I was first learning Python, why we need an `else` block—why not just include that code inside the `try` block itself? Much of the time, we can without making a practical difference in how our program runs. However, we can use this more stylistically. In many languages, it's normal to have huge blocks of code in a `try` block, even though the expected errors are only in one or two places. The `else` block lets us restrict our `try` block to only the code that we expect to generate an error. The `else` block will *only* run if no errors were encountered, so we can trust everything that was written in the `try` block ran successfully.

Else and File Input

A good example of this is file input. Whenever we load some data from a file, we want to enclose the attempt to load the file in a `try` block because file input commonly raises errors; some languages even require file input to happen inside a `try` block. Figure 3.5.14 shows what that looks like without an `else` statement; this code loads a file, then prints everything in the file.

#	ElseandFileInput-1.py	Output
1	`try:`	
2	`    #Open InputFile.txt in read-only mode`	
3	`    inputFile = open("InputFile.txt", mode = "r")`	
4	`    #For each line in the file`	
5	`    for line in inputFile:`	
6	`        #Print the line`	
7	`        print(line)`	
8	`    #Close the file`	
9	`    inputFile.close()`	
10	`#Catch an IOError`	
11	`except IOError as error:`	
12	`    print("An input error occurred!")`	
13		

Figure 3.5.14

Before talking about the `try` and `catch`, let's take some time just to understand file input. We'll talk about it more in Unit 4, but we can discuss it a little in the meantime. The `open()` function on line 3 takes as input a filename. Optionally, it can also take a mode as a keyword parameter: here, the mode "r" means read-only, which means we can read the contents of the file but not write to it. Once we've loaded the file into a variable (another data type, file!), we can read one line at a time with a `for-each` loop.

Here, we catch an input–output error on line 11, called an `IOError`, if an error arises inside the `try` block. However, the only place where an `IOError` can happen is when we read from or write to a file. That only technically happens on line 3; this

`open(filename):`
Takes as input a filename and returns the file. Once returned, the file can be read line-by-line or written to, depending on the mode. Mode is set with the keyword parameter "mode", "r" for read, "w" for write, "a" for append.

line loads the file into the program, and lines 5 and 7 just print it. So, really, we only need to catch an `IOError` that arises on line 3; it can't arise elsewhere. So, while the code we have right now is fine, it would be great to narrow down where the error could have arisen.

#	ElseandFileInput-2.py	Output

```
1  try:
2      #Open InputFile.txt in read-only mode
3      inputFile = open("InputFile.txt", mode = "r")
4  #Catch an IOError
5  except IOError as error:
6      print("An input error occurred!")
7  else:
8      #For each line in the file
9      for line in inputFile:
10         #Print the line
11         print(line)
12     #Close the file
13     inputFile.close()
14
```

Figure 3.5.15

The code in Figure 3.5.15 will do the same thing, but it's a little bit better organized. The `try` block contains only those lines of code that *need* to be in the `try` block, and lines that rely on that code are in the `else` block. What happens if we try to run this with a filename that doesn't exist?

#	ElseandFileInput-3.py	Output

```
1  try:                                                    An input error
2      #Open InputFile.txt in read-only mode               occurred!
3      inputFile = open("FakeInputFile.txt", mode = "r")
4  #Catch an IOError
5  except IOError as error:
6      print("An input error occurred!")
7  else:
8      #For each line in the file
9      for line in inputFile:
10         #Print the line
11         print(line)
12     #Close the file
13     inputFile.close()
14
```

Figure 3.5.16

As shown in Figure 3.5.16, the error is caught! We see the text in the output came from line 6 in the code. Because the error was caught, the `else` block doesn't execute, so we *don't* see any attempt to read the non-existent file in the output. This could be read as, "If an `IOError` occurs, print 'An input error has occurred!'; else, print the file using this loop."

Finally

Finally, we come to the finally block. As mentioned previously, the `finally` block is for code that needs to run regardless of whether an error was detected or not. With this block, we are now able to cover every possible situation:

- The `try` block contains the code to attempt.
- The `except` blocks contain the code to run if and only if an expected error type occurs.
- The `else` block contains the code to run if and only if no errors occur.
- The `finally` block contains the code to run regardless of whether or not an error occurred.

When would we need a `finally` block? Imagine if we expected a file to just contain numbers. When we originally load a line of text from this file, the line of text is stored as a string, and we want to convert it to an integer. Then, the error we

#	Finally.py	Output
1	`#Open InputFile.txt in read-only mode`	1
2	`inputFile = open("NumberFile.txt", mode = "r")`	2
3	`try:`	3
4	`    #For each line in the file`	4
5	`    for line in inputFile:`	5
6	`        #Print the line`	No errors occurred!
7	`        print(int(line))`	
8	`#Catch an IOError`	
9	`except ValueError as error:`	
10	`    print("A value error occurred!")`	
11	`else:`	
12	`    print("No errors occurred!")`	
13	`finally:`	
14	`    #Close the file`	
15	`    inputFile.close()`	
16		

Figure 3.5.17

would anticipate would be a `ValueError`, which is what would arise on line 7 of Figure 3.5.17 if one of the lines of the file was *not* an integer. In practice, we would also want to anticipate the `IOError` from before, but we'll remove that for now to demonstrate the `finally` block. However, even if a `ValueError` occurs, we still want to close the file! So, regardless of whether an error arises or not, we put the `close()` method call in the `finally` block, as shown in Figure 3.5.17.

`close()`:
A method that closes the file of which it's a member.

Notice a few things here. First, notice we moved the line opening the file (now line 2) outside the `try` block. We're no longer expecting an error here (for now, we're ignoring the error we might expect here), so it doesn't need to be in the `try`. Notice that we kept the loop (lines 4 through 7) inside the `try` block, even though the error can only occur when we're performing the type conversion. Technically, we could put the `try` block *inside* the loop! We'll try that later.

The main takeaway here, though, is that the code in the `finally` block is run regardless of whether any errors occurred or not. This code will call `inputFile.close()` regardless of whether an error was encountered converting the file or not. Now, this might leave you with a question: couldn't we instead just put the code we want to run regardless *after* the error handling blocks? Won't the code just jump back out after it's done and hit that line of code anyway? In other words, how is using the `finally` block any different from just putting `inputFile.close()` on line 13, unindented, as shown in Figure 3.5.18? The answer is that `finally` has a special behavior when it comes to uncaught errors.

#	FinallyandUncaughtErrors-1.py	Output
1	`#Open InputFile.txt in read-only mode`	1
2	`inputFile = open("NumberFile.txt", mode = "r")`	2
3	`try:`	3
4	`    #For each line in the file`	4
5	`    for line in inputFile:`	5
6	`        #Print the line`	No errors occurred!
7	`        print(int(line))`	
8	`#Catch an IOError`	
9	`except ValueError as error:`	
10	`    print("A value error occurred!")`	
11	`else:`	
12	`    print("No errors occurred!")`	
13	`#Close the file`	
14	`inputFile.close()`	
15		

Figure 3.5.18

Finally and Uncaught Errors

After the computer tries the code in the `try` block (lines 4 through 7 in Figure 3.5.18 and runs the code in either the `except` block (line 10, if there was an error) or the `else` block (line 12, if there wasn't an error), won't it just proceed to run the

`inputFile.close()` line? The answer is: kind of. The `finally` block has one special feature. If there were errors in the `try` block that were *not* handled by the `except` blocks, then the `finally` block *still* runs. Here, we're catching a `ValueError` on line 9, but no other types of errors; if a `TypeError` were to occur inside the `try` block, it would not be caught, but the `finally` block would *still* run.

Let's go back to our example from Figure 3.5.10. Remember when we had except blocks for `TypeErrors` and `ValueErrors`, but a `ZeroDivisionError` occurred? Our code still crashed in Figure 3.5.10; and even with a `finally` block, it will still crash. However, a `finally` block lets us do some things first.

The result of using a `finally` block is shown in Figure 3.5.19. The `ZeroDivisionError` isn't caught, but because it still occurred inside a `try` block, our `finally` block *still* runs. In the output, you see the error message, starting with "Traceback", but you also see the text printed by line 16 *after* the error is printed. You might notice also that this is almost the same result as including `except Exception:` at the end, as shown in Figure 3.5.13. However, with `except` and `finally`, our code still crashes after it runs the code inside the `finally` block. This is useful during debugging: when debugging, we want to know that our code crashes, but we want the opportunity to find out why, too. The `finally` block lets us print why, then crashes anyway.

#	FinallyandUncaughtErrors-2.py	Output
1	`myString = "This string is not a number!"`	`Converting myString to int...`
2	`#Run the code below until an error occurs`	`Traceback (most recent call last):`
3	`try:`	`File "...", line 5`
4	`    print("Converting myString to int...")`	`    print(1 / 0)`
5	`    print(1 / 0)`	`ZeroDivisionError: division by zero`
6	`    print("String #" + 1 + ": " + myString)`	`An unknown error occurred!`
7	`    myInt = int(myString)`	
8	`    print(myInt)`	
9	`#If an error occurs, check if it's a ValueError`	
10	`except ValueError as error:`	
11	`    print(error)`	
12	`#If an error occurs, check if it's a TypeError`	
13	`except TypeError as error:`	
14	`    print(error)`	
15	`finally:`	
16	`    print("An unknown error occurred!")`	
17	`print("Done!")`	
18		

Figure 3.5.19

Nested Try-Catch-Else-Finally

Earlier we said we would temporarily remove the check for `IOError` to show off `finally`. However, in practice we would still want to check that while opening the file, while also checking for a `TypeError` while reading and converting the file. If an `IOError` occurred, we don't even want to try reading or closing the file; but, if a `TypeError` occurred, we still want to close the file. How do we do this?

By now, you've seen nested control structures several times, so we won't belabor the point. The conclusion is: we can put a `try` block inside another `try` block, as shown in Figure 3.5.20. Our outer `try` block, starting on line 1, checks whether or not the file was successfully opened on line 3; if it wasn't then an `IOError` is raised, and so we just need to print that the file was not opened in the `except` block on lines 18 and 19. If it was successfully opened (i.e., if line 3 didn't cause an error), then the inner `try` block checks if the conversions were successfully run. If they were, that means that no errors were encountered, and so it reports that they were converted on line 13; if they weren't, it means an error was encountered, so it reports that they weren't converted on line 11. Either way, it needs to close the file, so it does so on line 16 inside the `finally` block. In Figure 3.5.20, we see the code running with a file of all integers, so no errors occur.

```
#  NestedTryCatchElseFinally-1.py
1  try:
2      #Open InputFile.txt in read-only mode
3      inputFile = open("NumberFile.txt", mode = "r")
4      try:
5          #For each line in the file
6          for line in inputFile:
7              #Print the line
8              print(int(line))
9      #Catch a ValueError
10     except ValueError as error:
11         print("A value error occurred!")
12     else:
13         print("No errors occurred converting the file!")
14     finally:
15         #Close the file
16         inputFile.close()
17 #Catch an IOError
18 except IOError as error:
19     print("An error occurred reading the file!")
20
```

```
Output
1
2
3
4
5
No errors
occurred
converting the
file!
```

Figure 3.5.20

Notice that line 16 is only reachable if the file was opened successfully; if it wasn't opened successfully on line 3, then execution would be kicked to line 18, skipping lines 4 through 16. By placing the `inputFile.close()` function call here, we guarantee we only try to close the file if it was previously opened.

Figure 3.5.21 shows the code running with a file of non-integers. An error occurs in the inner `try` block and is caught as a `ValueError` when the code tries to convert a string without an integer into an integer. So, "A value error occurred!" is printed, but the file is still closed by line 16 because of the `finally` block.

```
#  NestedTryCatchElseFinally-2.py
1  try:
2      #Open InputFile.txt in read-only mode
3      inputFile = open("InputFile.txt", mode = "r")
4      try:
5          #For each line in the file
6          for line in inputFile:
7              #Print the line
8              print(int(line))
9      #Catch a ValueError
10     except ValueError as error:
11         print("A value error occurred!")
12     else:
13         print("No errors occurred converting the file!")
14     finally:
15         #Close the file
16         inputFile.close()
17 #Catch an IOError
18 except IOError as error:
19     print("An error occurred reading the file!")
20
```

```
Output
A value error
occurred!
```

Figure 3.5.21

Finally, Figure 3.5.22 shows the code running with an input file that doesn't exist. The error occurs on line 3, which is in the outer `try` block, so it is caught as an `IOError` by line 18. So, "An error occurred reading the file!" is printed. The file doesn't need to be closed because it was never successfully opened in the first place, and similarly, no `ValueError`s could occur because the `try` block quit before reaching line 8.

#	NestedTryCatchElseFinally-3.py	Output

```
1  try:
2      #Open InputFile.txt in read-only mode
3      inputFile = open("FakeFile.txt", mode = "r")
4      try:
5          #For each line in the file
6          for line in inputFile:
7              #Print the line
8              print(int(line))
9      #Catch a ValueError
10     except ValueError as error:
11         print("A value error occurred!")
12     else:
13         print("No errors occurred converting the file!")
14     finally:
15         #Close the file
16         inputFile.close()
17 #Catch an IOError
18 except IOError as error:
19     print("An error occurred reading the file!")
20
```

```
An error occurred
reading the file!
```

Figure 3.5.22

5. Error Handling and Other Control Structures

Recall that early in our material, we covered some common types of errors such as `TypeError` and `NameError`. Since then, we've encountered some others, like `IOError` and `ValueError`. Remember, when we first went over these, we stated that you wouldn't necessarily understand them all right away: rather, they were provided early so you could keep going back to them. I'd advise going back to them now as well with your new knowledge of programming and errors in general. You may also read a complete listing of Python's error types here: https://docs.python.org/3/library/exceptions.html.

As we close our conversation on control structures, let us look at how error handling integrates with the other control structures that we have seen.

Error Handling and For Loops

Recall as briefly mentioned earlier that because a `for` loop was itself enclosed in a `try` block, one single error on any iteration of the loop would cause the execution of the program to jump to the error handling statements. So, the code would read from the file until it found a non-integer line, and then it would quit, as shown in Figure 3.5.23. This file contains some lines with integers, then some without.

#	ErrorHandlingandForLoops-1.py	Output

```
1  try:
2      #Open InputFile.txt in read-only mode
3      inputFile = open("NumberAndLetterFile.txt", mode = "r")
4      try:
5          #For each line in the file
6          for line in inputFile:
7              #Print the line
8              print(int(line))
9      #Catch a ValueError
10     except ValueError as error:
11         print("A value error occurred!")
12     else:
13         print("No errors occurred converting the file!")
14     finally:
15         #Close the file
16         inputFile.close()
17 #Catch an IOError
18 except IOError as error:
19     print("An error occurred reading the file!")
20
```

```
1
2
A value error occurred!
```

Figure 3.5.23

As we see in Figure 3.5.23, the code runs just fine for the first two lines of the file, which have integers (1 and 2). The third line of the file does not have an integer, so it jumps to the `except` block, which ends execution and prints that a

`ValueError` has occurred. What if we wanted it to only skip the current iteration, though, and then keep reading the file? To do that, we could switch the order: instead of putting the `for` loop inside the `try` block, we could put the `try` block inside of the `for` loop, as shown in Figure 3.5.24.

#	ErrorHandlingandForLoops-2.py
1	`try:`
2	`    #Open NumberAndLetterFile.txt in read-only mode`
3	`    inputFile = open("NumberAndLetterFile.txt", mode = "r")`
4	`    #For each line in the file`
5	`    for line in inputFile:`
6	`        try:`
7	`            #Print the line`
8	`            print(int(line))`
9	`        #Catch a ValueError`
10	`        except ValueError as error:`
11	`            print("A value error occurred!")`
12	`        else:`
13	`            print("No errors occurred converting this line!")`
14	`    #Close the file`
15	`    inputFile.close()`
16	`#Catch an IOError`
17	`except IOError as error:`
18	`    print("An error occurred reading the file!")`
19	

Output
1
No errors occurred converting this line!
2
No errors occurred converting this line!
A value error occurred!
A value error occurred!

Figure 3.5.24

How will Figure 3.5.24's execution differ? Recall that when our code runs, the code inside the `try` block will run. If an error arises, it will jump into the `except` block. If no error is found by the conclusion of the `try` block, it will jump into the `else` block. Either way, it will then run the `finally` block, if present. What happens after that? After that, execution moves on to the next line of code outside of the `try-except-else-finally` structure.

However, in Figure 3.5.24, that `try-except-else` structure is in a loop. When we reach the end of an iteration of the loop, execution jumps back to the loop and asks, "Are the loop's conditions fulfilled?" If so, the loop ends. If not, it does not. Whether an error was raised or not, the loop is not done. The `try`, `except`, and `else` blocks were all inside the loop, so when the code jumps to the `except` block, it's still jumping inside the loop. Previously, when it jumped to the `except` block, it was jumping out of the loop. Now, it's jumping within the loop, so an error does not interfere with the loop touching each line of the file. In Figure 3.5.24, we can tell this is happening because the code continues running after hitting an error: specifically, it encounters *two* errors because the third and fourth lines *each* have non-integer contents, so *each* cause an error. Previously, encountering an error terminated the loop, so it would be impossible to encounter two errors.

Error Handling and Functions

What happens if an error arises in a function that you write? There are two ways we might handle that: we could handle it inside the function body, or we could handle it in the code that makes the function call. Let's look at both, using a silly function we'll write specifically to create errors: `divideByZero()`.

#	ErrorHandlingandFunctions-1.py
1	`#Attempts to divide by zero`
2	`def divideByZero():`
3	`    try:`
4	`        print(1 / 0)`
5	`    except Exception as error:`
6	`        print(error)`
7	
8	`print("About to encounter an error...")`
9	`divideByZero()`
10	`print("We just encountered an error!")`
11	

Output
About to encounter an error...
division by zero
We just encountered an error!

Figure 3.5.25

In Figure 3.5.25, we catch the error inside the function. When the function is called, it attempts to execute line 4, fails, and jumps into the `except` block. It prints the error on line 6. Then, it jumps back to the main program, and runs the final print statement on line 10.

What happens if we put the error handling directly in the code that calls the function? As shown in Figure 3.5.26, we get effectively the same result. The error occurs inside the function, but because it isn't caught inside the function, it comes back out to the main program. There, it does get caught. That's a pretty advanced principle, so don't worry if it's a bit confusing. The point is that if an error happens in a function, it will keep "rising" until it is handled. If it's never handled, the program crashes.

#	ErrorHandlingandFunctions-2.py	Output
1	`#Attempts to divide by zero`	About to encounter an error...
2	`def divideByZero():`	division by zero
3	`    print(1 / 0)`	We just encountered an error!
4		
5	`print("About to encounter an error...")`	
6	`try:`	
7	`    divideByZero()`	
8	`except Exception as error:`	
9	`        print(error)`	
10	`print("We just encountered an error!")`	
11		

Figure 3.5.26

6. Error Handling and Turtles

We've been developing code that allows a user using the command line to control the turtle in the turtle window. However, we've noted a couple times that there was a weakness. While our code could intelligently react if the user entered an invalid command, it could not intelligently react if the user entered an invalid argument. When prompted for distance, angle, or number of sides, if the user entered a non-numerical input, the program crashed.

We now have the means to fix that. We'll keep things relatively simple and say that if a user enters an invalid argument, they are kicked back out to the first menu.

Error Handling and Turtles

Our goal to start with is to rerun the loop from scratch if an error is encountered. We don't want to quit the entire program if the user enters invalid input, but right now we're not worried about just repeating the same questions until we get the right answer. So, in that case, we can wrap the entire series of conditionals in one giant `try` block, as shown in ErrorHandlingandTurtles.py on line 24.

We added only a couple lines–line 24, 64, and 65, as well as indenting the lines after 24–but their impact is powerful. Now, if the user accidentally enters a letter instead of a number, the code doesn't just quit and crash; it tells them that the input was invalid, but it lets them try again. That's immensely powerful. Now, the only way to exit the program is to type `end` when prompted. It's far less likely for someone to do *that* by accident than accidentally enter a letter when they should enter a number.

Error Handling and Functions with Turtles

However, note that this still isn't ideal. The ideal approach would be to instead keep repeating that one specific query until the user puts in some valid input. If a user selects the snowflake command and enters "5," "100," and then accidentally types "3p" instead of "30," it should not send them all the way back to the beginning to enter "snowflake," "5," and "100" again. Instead, it should simply ask them to try again on that last prompt.

With what we have in ErrorHandlingandTurtles.py, though, that's extremely difficult. We could wrap each individual input statement in a `try` block, but that would add a lot of mess to our code. More importantly, it still wouldn't help; after executing the `except` block for these `try` blocks, the code would continue. It wouldn't be able to draw without getting the argument and it couldn't go back and get the argument, so it would return to the beginning of the loop again.

So, how can we force the user to keep answering a *single prompt* until an integer is correctly entered? Beware, things are about to get complicated. Don't worry if this confuses you at first. This might not make total sense for a long time. You can see the implementation of this in ErrorHandlingandFunctionswithTurtles-Recursion.py.

We've actually only made two general changes to create ErrorHandlingand-FunctionswithTurtles-Recursion.py from ErrorHandlingandTurtles.py:

1. We added the `getIntegerInput()` function at the top.
2. We replaced all the calls to `input()` inside our main code with calls to `getIntegerInput`().

We also removed a couple type conversions that aren't needed anymore, and our error handling code which isn't needed anymore either. However, even though these are only a couple changes, what we've changed is profound. First, instead of relying on Python's built in `input()` function, we've built our own. It still uses Python's `input()` function, we've built some reasoning around it. And because we've put it in a function, we can refer to it wherever we were referring to Python's `input()` function. This has the practical effect of being *like* putting these `try` and `except` blocks all over our program: we keep referring to a function that has these `try` and `except` blocks built in.

The more profound thing we've done here, though, is how we've structured our `getIntegerInput`() function. We get the user's input, try to convert it to an integer, and then *if it works*, we return it. So, if we return from here, we know it's an integer, and the user's input was valid.

What happens if the user enters invalid input, like a letter? That generates an error on the second line of the `try` block. That means the program jumps down to the `except` block. It tells the user to enter an integer, and then it does something clever: it *runs* `getIntegerInput()` *again*, with the same prompt. Don't worry if this is confusing; we're previewing the advanced topic of recursion from the last unit of our course.

Essentially, when execution reaches the second line of the `except` block, it creates *another copy* of `getIntegerInput()`, and runs that. So, we repeat exactly what we just did. If the user enters valid input the second time, then the second copy of the function returns that input. The first copy basically says, "return whatever the second copy returns." If the user enters invalid input the second time, too, then execution creates a *third* copy, and the second copy says, "return whatever the third copy returns." So, as long as the user keeps entering invalid input, it keeps creating an extra copy, and each copy returns whatever the copy it creates returns.

It's like a `while` loop, and in fact, we *could* implement this in a `while` loop. ErrorHandlingandFunctionswithTurtles-While.py shows how.

In ErrorHandlingandFunctionswithTurtles-While.py, we're doing effectively the same thing: getting input from the user, checking if it's an integer, and repeating the request if not. The main difference is that instead of a function creating another copy of itself, this relies on us knowing that there is an `isdigit()` function that checks if a string holds a digit. So, we've actually *removed* error handling from this code by checking if an error will arise preemptively.

This is the most complicated thing we've covered so far (especially the first version of doing this), so don't worry if you're a little lost. The main takeaway here is the type of complexity we're starting to build into our program. Lots of functions, helper functions, replacing built-in functions—we're getting pretty advanced.

UNIT 4

DATA STRUCTURES

CHAPTER

4.1

Data Structures

1. What Are Data Structures?

So far, we've done some pretty interesting programming with a relatively limited number of data types. We've used numbers, some characters, and some strings. Is it really possible, though, that all the complex reasoning we see on computers can be built out of these simple types of data?

Advanced Data Types

In a sense, yes; in fact, everything on your computer is distilled down to 1s and 0s for your computer to process with a relatively small set of commands, at a rate of billions of commands per second. Our programs get translated down through the layers until the computer can execute them in terms it understands.

But for building programs at our level, do we really build them out of these control structures and simple data types? No; we need some more complicated **data structures** to build the reasoning that we see in computers today. In this unit, we'll cover a few of these data structures. Specifically, we'll focus on the data structures that contain multiple pieces of other information, like lists and strings—remember, strings are like lists of individual characters

Lists and List-Like Structures

So far, everything we've covered has involved a one-to-one mapping between variables and values. Every variable could have one and only one value. What happens, then, if you need to keep track of the names of all the students in a class? Do you create a different variable for each student? What if you're building a roster program where you don't know in advance how many students to expect; do you just make sure to create more variables than you need?

This is where list-like structures come into play. **List-like structures**, also called sequences, give multiple values to a single variable name. The individual values are then accessed through some kind of index. For example, if we have a variable named `roster`, we could specifically ask for the first name on the roster, the seventh name, or the twelfth name. The variable `roster` has one name, but there are multiple values associated with it, accessed via numbers called **indices** (plural for index). Sometimes (specifically, with dictionaries, the topic of Chapter 4.5) it won't be numbers that we use to access these values, though; it might be strings or other data variables.

Lists can generally hold any kind of data, including other lists. Using that, you can make some pretty complex data structures. You could make a list of classes in a school, where each class is actually a list of students. That could go deeper: each student could be a list of grades, or you could have a list of schools where each school is a list of classes. That's moving toward the more complex data structures that we'll talk about in Unit 5.

Unit Outline

In this unit, we'll cover four different general classifications of data structures. We'll begin with **strings**. You've seen strings a lot before, but only because it's difficult to

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Use advanced data structures and ways to pass values of a variable, including pass by value and pass by reference;
- Demonstrate the use of pass by value and pass by reference and leverage the concept of mutability;
- Differentiate between methods and functions and analyze the effect of a method call on a program.

Data Structures
Approaches to organizing abstract data types, such that the data can be accessed efficiently.

List-Like Structures
Also referred to as sequences and collections, a data structure that holds multiple individual values gathered together under one variable name, accessed via indices. Includes to lists, arrays, and tuples. Lists are simultaneously a type of data structure and a specific type in some languages.

Index
A number used to access a particular element from a list-like data structure. Traditionally, most programming languages assign the first item of a list-like data structure the index 0.

String
A data structure that holds a list, or a string, of characters.

do anything in programming without the ability to print text. We've only scratched the surface of what strings can do and how they can be used. Strings are actually relatively advanced data structures, both because they technically store a list of other items (in this case, a list of individual characters) and because they have complex reasoning built into them to deal with human language.

Then, we'll cover **lists**. Lists are single variable names that can store several different items of data, accessed by a numeric index. For example, we can ask for the 7th value of the list named `myList`. Different languages handle lists very differently, and many languages have multiple different ways of handling lists, so we'll briefly cover the general concepts before jumping into your language's instantiation of the concept.

Lists
A data structure that holds multiple individual values gathered together under one variable name, accessed via indices. Similar to arrays and tuples.

Then, we'll cover **file input and output**. File input and output is how we persist information across multiple sessions of a program. In many ways, it resembles and builds on list-like structures. When writing files, we'll very often write all the items in a list one-by-one. When reading a file, we'll often treat the individual lines like strings in a list. This isn't always the case, but the similarity is sufficient to make this a good time to cover file input and output.

File Input and Output
The complementary processes of saving data to a file and loading data from a file, generally such that the state of the memory of the program is the same after saving and loading have occurred.

Finally, we'll cover more advanced list-based structures, like **Dictionaries** and HashMaps. These data structures are similar to lists in that they store multiple values, but they differ in that the values can be accessed not just through numeric indices, but also through strings or even other data structures. You could, for example, have a data structure that stores multiple students and their grades, and to look up my grades, you'd look up "David" instead of having to know that I'm the 7th student in the list.

Dictionaries
A data structure comprised of key-value pairs, where a key is entered into the dictionary to get out a value. Similar to or synonymous with Maps, Associative Arrays, HashMaps, and Hashtables.

Before we get there, though, there are a couple concepts regarding advanced data structures that are important to understand: passing by value, passing by reference, and mutability. These are complex principles that may not come up often in your routine programming, but they're fundamental concepts in computing.

2. Passing by Value vs. Passing by Reference

The first concept we want to understand is the difference between **passing by value** and **passing by reference**. This distinction specifically applies to writing and using functions. This is one of the more complicated concepts we've covered so far, and the terminology is a bit unfamiliar. Let's start by talking about the difference between the two, and then move on to talking about where the terms "by value" and "by reference" actually come from.

Passing by Value
An approach for passing arguments into a function where the function is not able to modify the variable whose value was getting passed, only its local parameter that accepts the argument.

Passing by Reference
An approach for passing arguments into a function where the function is able to modify the variable whose value was getting passed, changing it for both the function and the code that called the function.

Passing by Value: An Analogy

Let's return to the analogy we used when describing functions. You worked in an office, and you had a co-worker named Addison. Addison's job was to add two numbers. Any employee in the office could give Addison two numbers, and Addison would return the result.

Let's bring variables into this. On your desk, you have two files: File A and File B. On File A and File B are written numbers, 5 and 2 respectively. These are variables and values: the variable A has the value 5, and the variable B has the value 2. Last time, you wanted to add these two numbers together, but we didn't really talk about what we would do with the result. This time, let's imagine we want to add B to A; or, in other words, to add 2 to 5. You want to set A equal to the sum of A and B. Our end result should be that A should have the value 7.

Previously, the way we described this is that you shouted across the office, "Hey Addison [the function call], 5 and 2 [the arguments]!" Addison shouted back, "7 [the return]!" So, you erase the number 5 on File A and write the number 7. You set A equal to the sum of A and B, according to the function `Add(ison)`.

This is passing by value. You hollered to Addison the values themselves, 5 and 2. Addison didn't know 5 and 2 were stored in File A and File B. Then, Addison shouted back the value corresponding to the result. He didn't know what you'd do with that value. All he knew was he needed to shout the value back. The key here is that it is the values themselves being shouted back and forth: Addison never knows your variables, and you never know Addison's variables.

Passing by Reference: An Analogy

This interaction could have gone differently, though. Instead of shouting the values of File A and File B across the office, you could have instead walked by Addison's desk and handed him the two files. Rather than simply telling you the value, now Addison himself erases the original value of File A and puts in the new value. He then brings the files back to you. The result is the same: File A holds 7, File B holds 2 as it did at the beginning. Or, maybe he brings the values back to you as they were, and tells you verbally that the answer is 7. Either way, he had access to your variables and could modify them.

This is passing by **reference**. The major difference is that you and Addison are accessing the same variables. Note that the variables didn't necessarily have to have the same name: Addison could have a different name he uses to refer to File A. What's important is that during his operation, Addison could write to *your* variable. When passing by reference, you're handing the variable itself to the function, and the function can modify the variable's value if it wants. When passing by value, you're simply handing off the value, and the function can't actually change the value of your variable.

Reference
An alias to a variable that already exists. Either the reference or the variable name can be used to access the value stored in that variable.

Why does this matter? Imagine if Addison had a strange way of adding numbers. Instead of just adding them, he instead changes the sign of one number, then subtracts it instead. So, when Addison receives 5 (File A) and 2 (File B), he changes them to 5 and −2, and performs 5 – (−2). The result is still 7. However, if you passed by reference, you handed Addison access to File B. He changed File B from 2 to −2. Now, when he hands you the files back, File B has been modified.

The difference between passing by value and passing by reference comes down to whether or not you want the function to be able to change the values of the variables directly, or if you simply want it to receive the values themselves without being able to access the variables. If you shout "5 and 2!" to Addison, he can't change what you have written down; if you hand the papers to him, he can.

Terminology: By Reference

The term "passing by value" makes some sense. You're passing some data into a function, and you're doing it *by* passing a variable's *value*. You're passing the data by passing its value.

What does passing "by reference" mean, then? To understand that, we need to understand a little bit about the way a computer works on the inside. Imagine your computer like a giant file cabinet. It has thousands and millions of files. These files are the computer's memory: they store everything that it knows at a given time. To access some data, you have to know where in the file cabinet the data is located. So, in our analogy, to access File A and File B, you have to know where they are.

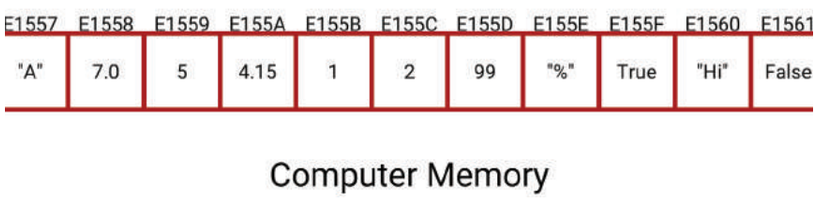


Figure 4.1.1

So, are they just labeled "File A" and "File B"? No; they're labeled with some far more cryptic identifier like E1557. These identifiers are systematic and ordered; that's why they're relatively easy to navigate. We know that E1557 will come after E1556 and before E1558. These identifiers are tough to use, though. So, we create more accessible names, like "File A", and somewhere we have a key indicating, "File A can be found at E1559."

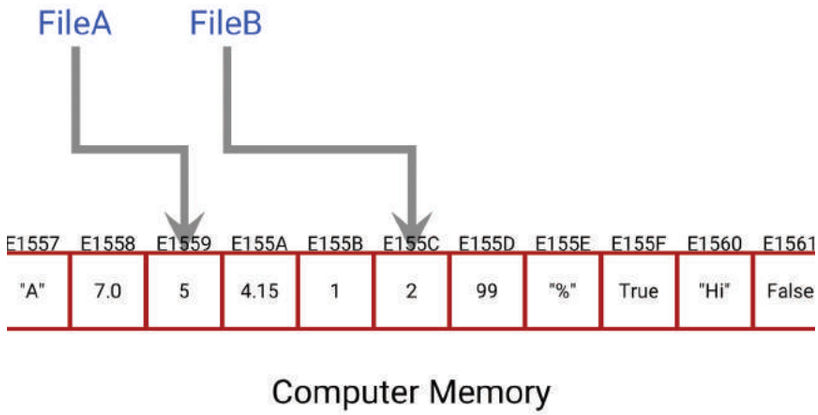


Figure 4.1.2

That cryptic identifier is called a reference (or a memory address). It tells you where the variable itself can actually be found. When we pass by value, we grab the variable name (File A), find its reference (E1559), use the reference to find the value (5), and then tell the function the value ("Hey Addison, 5 and…"). The function never knows where the value came from.

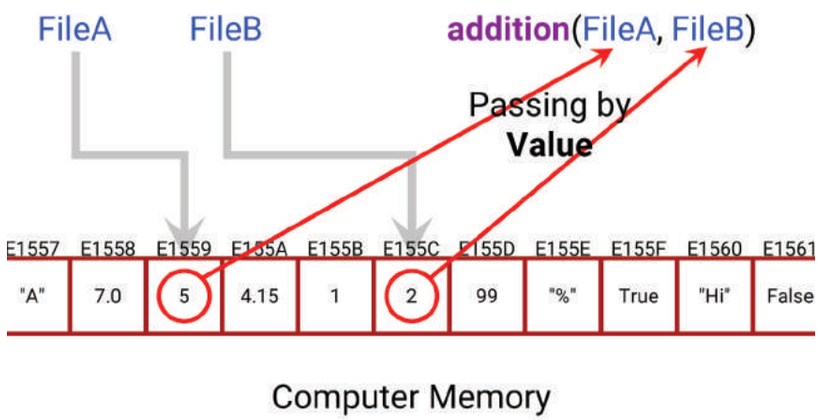


Figure 4.1.3

When we pass by reference, we grab the variable name (File A), find its reference (E1559), and pass that reference *directly* to the function ("Hey Addison, the value stored at E1559 and…"). The function then looks up the value on its own, but because it knows the reference, it can change the value if it wants to. It doesn't have to, but it can.

That's why these two approaches are called pass by value and pass by reference. With pass by value, we simply tell the function what value to operate on; with pass by reference, we tell it where to find the value, such that it could change the value if it wants to.

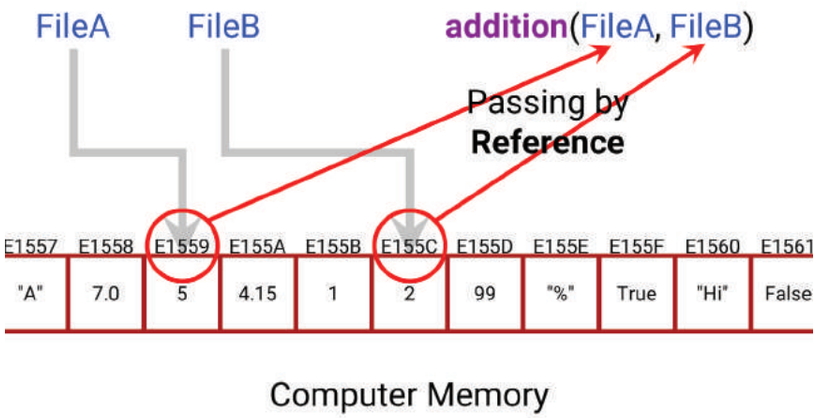


Figure 4.1.4

3. Passing by Value and Reference in Python

Many languages let you define whether you want to pass an argument by value or by reference based on the way you write the function. In Visual Basic, there are reserved keywords `ByVal` and `ByRef` that tell the computer whether to send in the value or the reference. In C, the asterisk (*) character means "address of" or "reference to" a variable, so passing in the variable name with an asterisk means "the reference to" a variable, while passing in the variable name without an asterisk means "the value of" a variable.

In other languages, like Java, whether an argument will be passed by value or by reference is determined in the language. In Java, primitive data types like integers and characters are always passed by value, while advanced data structures like lists are always passed by reference.

How does Python work? We could just answer that, or we could actually take a look and see.

Integers: By Value or by Reference?

Note first that in Figure 4.1.5, we've started to include a label in our print statements. We'll do that going forward to make our output a little easier to follow, even though it will make our code a little tougher to read.

We start by defining a function `addOne()` on line 2. `addOne` has one parameter, `anInteger`. It takes whatever argument is passed into `anInteger` and adds one to it. It does *not* return the result; it simply sets `anInteger` equal to `anInteger` + 1.

Outside the function, we create `myInteger` and give it a value of 5 on line 7. We print it on line 8 to make sure its value really is 5 (it is). We then call `addOne()`, where `myInteger` becomes an argument to the `anInteger` parameter

#	IntegersByValueorByReference.py
1	`#Add one to anInteger`
2	`def addOne(anInteger):`
3	`    anInteger = anInteger + 1`
4	`    print("anInteger:", anInteger)`
5	
6	`#Create myInteger with the value 5`
7	`myInteger = 5`
8	`print("myInteger before addOne:", myInteger)`
9	`#Call addOne on myInteger`
10	`addOne(myInteger)`
11	`print("myInteger after addOne:", myInteger)`
12	

Output
myInteger before addOne: 5
anInteger: 6
myInteger after addOne: 5

Figure 4.1.5

in `addOne()`. `anInteger` is then incremented by one. We print `anInteger` on line 4 to make sure it now has the value 6 (it does).

Then, after the function call is over, we print `myInteger` one more time on line 11. If it was passed by value, it would now be 5: `addOne()` would never have access to the original variable, so it couldn't change the value of `myInteger`; it only operates on `anInteger`, which was given the value of `myInteger` at the time of the function call, but from then on exists as its own variable. If `myInteger` was passed by value, it was as if `addOne()` made a copy of the value.

If `myInteger` was passed by reference, though, it would be 6 after the call to `addOne()`, because `addOne()` would be modifying the value of the original variable itself; because it knows the references, it knows where to find it. If `myInteger` was passed by reference, it was as if `addOne()` was working on the same variable as the code that called it.

What is the result? When printed on line 11, `myInteger` retains the value 5, meaning that Python *seems* to have passed this by value rather than by reference. Note that later, we'll see that Python actually does something slightly different: however, the result is functionally the same as passing by value.

Other Data Types: By Value or by Reference?

We noted above, though, that Java treats different data types differently. What about Python? Let's check how it handles strings in Figure 4.1.6.

#	OtherDataTypesByValueorByReference-1.py	Output
1	`#Add an exclamation to aString`	`myString before addExc: Hello, world`
2	`def addExc(aString):`	`aString: Hello, world!`
3	`    aString = aString + "!"`	`myString after addExc: Hello, world`
4	`    print("aString:", aString)`	
5		
6	`myString = "Hello, world"`	
7	`print("myString before addExc:", myString)`	
8	`addExc(myString)`	
9	`print("myString after addExc:", myString)`	
10		

Figure 4.1.6

Just like Figure 4.1.5, we create a string `myString` on line 6, pass it into a function that will modify it on line 8, and check to see if that modification persists in the main program on line 9. We see that `addExc` successfully adds an exclamation point to `aString`, its local variable, because there is an exclamation point in the text printed by line 3. However, `myString` remains unchanged when printed on line 9. So, Python seems to have passed this string by value, too. Modifying `aString`, `addExc`'s local copy of `myString`'s value, does not change the actual value f `myString`.

When we described Java, though, we mentioned it was Java's primitive types that were passed by value. Strings are somewhat primitive in Python, so let's see how Python treats something more complex, like a list. We don't know much about lists yet, but this example doesn't need much understanding of them. Still, if this is confusing, revisit this after reading Chapter 4.3.

#	OtherDataTypesByValueorByReference-2.py	Output
1	`#Add an item to aList`	`myList before addItem: ['One', 'Two', 'Three']`
2	`def addItem(aList):`	`aList: ['One', 'Two', 'Three', 'New Item!']`
3	`    aList.append("New Item!")`	`myList after addItem: ['One', 'Two', 'Three', 'New Item!']`
4	`    print("aList:", aList)`	
5		
6	`myList = ["One", "Two", "Three"]`	
7	`print("myList before addItem:", myList)`	
8	`addItem(myList)`	
9	`print("myList after addItem:", myList)`	
10		

Figure 4.1.7

Figure 4.1.7 shows the result of this same kind of code running on a list instead of an integer or string. Now things are getting interesting, though. When we did this with integers and strings, the values of `myInteger` or `myString` were identical before and after the function was run, and only the value inside the function was different. Now with a list, though, the values of the list inside the function and after the function match, as shown by the second and third lines of the output. The new item in the list is still in the list out in the main program. You don't need to worry about how lists work right now; all you need to notice is that the output of printing the list inside `addItem()` on line 4 matches the output of printing the list after `addItem()` on line 9.

This means that Python seems to pass primitive data types such as integers and strings by value, and advanced data types such as lists by reference, right? Practically speaking, yes. Accurately speaking, no! The ultimate effect is that it's *as if* Python is passing these primitive data types by value, but in reality, something different is going on: Python has immutable data types. That gets tricky, though, so we'll talk about that next lesson. For now, it's sufficient to know: there are some data types that Python *effectively* passes "by value", but for the majority of data types, Python passes by reference.

Variable Assignments

This is a good time to briefly look at a related dynamic in how variables are assigned in Python. There are some variable assignments that function similarly to how these function calls work

In Figure 4.1.8, we create `myInt1` and give it the value 5 on line 1. Then we assign `myInt2` to `myInt1` on line 2. Then we change `myInt1` to 7 on line 3. What is the result? `myInt1` now has the value 7, its new value, as shown in the output of line 5. `myInt2`, though, keeps the value 5, as shown in the output of line 6. So, `myInt2` isn't set to equal `myInt1` on line 2; it's set to equal the *current value* of `myInt1` on line 2. This is similar to our notion of pass-by-value, although this is assignment-by-value. We assign `myInt2` to the current value of `myInt1`. If the value of `myInt1` changes, it doesn't change `myInt2` because it was only set to the value of `myInt1` at one point in time.

#	VariableAssignments-1.py	Output
1	`myInt1 = 5`	`myInt1: 7`
2	`myInt2 = myInt1`	`myInt2: 5`
3	`myInt1 = 7`	
4		
5	`print("myInt1:", myInt1)`	
6	`print("myInt2:", myInt2)`	

Figure 4.1.8

What happens when we try that with a list? The same thing happens in Figure 4.1.9 that happened with our function calls with lists in Figure 4.1.7. We create `myList1` on line 1, then set `myList2` equal to `myList1` on line 2. We then change `myList1` on line 3. When we print both lists, we find that `myList2` *still* equals `myList1` on lines 5 and 6. It's been set equal to the *reference* to `myList1`. In other words, `myList1` and `myList2` now point to the same data: if we change `myList1`,

#	VariableAssignments-2.py	Output
1	`myList1 = ["One", "Two", "Three"]`	`myList1: ['One', 'Two', 'Three', 'Four']`
2	`myList2 = myList1`	`myList2: ['One', 'Two', 'Three', 'Four']`
3	`myList1.append("Four")`	
4		
5	`print("myList1: ", myList1)`	
6	`print("myList2: ", myList2)`	
7		

Figure 4.1.9

we're actually changing the data that `myList1` and `myList2` both point to. It's kind of like `myList2` is following `myList1` around, copying its data, but really, it's not even copying it: it's pointing to the exact same data.

To stick with our previous analogy, this is like you and Addison remember the location of a certain file in two different ways. You remember, "Oh, that's file E1557", while Addison remembers, "Oh, that's the file in the second drawer of the third cabinet eight folders back." These are two different "names" for the same file. If you modify file E1557, then you're also modifying Addison's second-drawer-third-cabinet-eighth-folder's-file.

Or, to take an alternate analogy: you likely refer to your mother as "mom," while others refer to her by her name. These are two different identifiers that point to the same underlying "data". Saying "Margaret is my mom" means that anything that happens to "Margaret" happens to "my mom."

4. Mutability in Python

Mutability
Whether or not a variable can have its value changed after being declared.

Mutable Variable
A variable whose value can change after it has been declared.

Immutable Variable
A variable whose value cannot change after it has been declared.

Mutability is a simple idea with complicated implications. A variable is said to be **mutable** if its value can change after it has been assigned; all the variables we've used so far have *appeared* to be mutable because we can always change their values by reassigning them. Inversely, a variable is said to be **immutable** if its value cannot change. You may create the variable and assign it a value, but once that value has been assigned, it cannot be changed again.

Mutability vs. Passing by Reference

Python passes all arguments by reference. We noted above that, for all practical purposes, Python seems to pass certain data types by value, and indeed, there's really no functional difference between what it actually does and passing data types by value. However, our goal here is to learn computing, not just programming, so we should know what's going on "under the hood."

What's going on in this case is that integers, floats, strings, and some other data types in Python are actually immutable. In fact, every data type that appeared to be passed by value is immutable. That means that once a variable of these types is created with a value, its value cannot be changed. To go back to our analogy on passing by reference, this is like handing Addison File A with the number 5 written in permanent marker. Yes, he knows where the file is located, but he can't erase the current value; it's written permanently.

So, the values of `myInteger` and `myString` in Figures 4.1.5 and 4.1.6 didn't change because `myInteger` and `myString` were immutable. Their values can't change. Simple, right? Not exactly.

Reassigning Immutable Data Types

The idea that integers and strings are immutable seems to be contradicted in the very same segment of code that is meant to demonstrate that they're immutable. Let's look at it again, shown here in Figure 4.1.10.

#	ReassigningImmutableDataTypes-1.py	Output
1	`#Add one to anInteger`	`myInteger before addOne: 5`
2	`def addOne(anInteger):`	`anInteger: 6`
3	`    anInteger = anInteger + 1`	`myInteger after addOne: 5`
4	`    print("anInteger: ", anInteger)`	
5		
6	`#Create myInteger with the value 5`	
7	`myInteger = 5`	
8	`print("myInteger before addOne:", myInteger)`	
9	`#Call addOne on myInteger`	
10	`addOne(myInteger)`	
11	`print("myInteger after addOne:", myInteger)`	

Figure 4.1.10

`myInteger` has the same value in lines 8 and 11 because `myInteger` is immutable, and therefore cannot be changed by `addOne()`. But wait: isn't that very principle being contradicted inside `addOne()`? Isn't `anInteger` also an integer, and isn't its value being changed in line 3?

Or, to make this even simpler, we can consider just the block of code shown in Figure 4.1.11. We're changing `myInteger` right there on line 2, and the print statement on line 3 confirms its value changed! So how can integer be an immutable data type? We *just* saw its value change in one of the simplest programs we can imagine.

#	ReassigningImmutableDataTypes-2.py	Output
1	`myInteger = 1`	2
2	`myInteger = 2`	
3	`print(myInteger)`	
4		

Figure 4.1.11

Brace yourself, this is about to get weird.

In Python, we can't change the value of an immutable variable. We can, however, change the reference of an immutable variable to point to a different value. That's what's happening here. We're not technically changing the value of `myInteger`; we're telling `myInteger` to point to the address of a different value.

This will hopefully make more sense in terms of our analogy. We have File A, and on File A is written 5 in permanent marker. We can't change the number 5 on File A. However, imagine we want to change the value of File A to 7. What do we do? We take out a new sheet of paper, write the number 7 in permanent marker, and we say to ourselves, "Okay, this is now File A." We never changed what was written on the original sheet of paper, we just changed what we named it. We wanted File A to be 7, so we said, "File A now refers to this other sheet of paper, on which is written the number 7." If that feels like cheating, then you're understanding it pretty well.

That's exactly what Python is doing in Figure 4.1.11. When we call line 1, it creates the value 1 in memory, and points `myInteger` at 1. When we change `myInteger` to 2, it doesn't change the same spot in memory where 1 was stored. Instead, it grabs a new spot in memory, puts the number 2 in it, and says that `myInteger` now points to the new spot. So, the 1 is still there.

Let's tie this back to Addison. We drop File A and File B on Addison's desk. They have the numbers 5 and 2 written in permanent marker on them. Addison wants to change the value of File A to 7. So, what does he do? He pulls out a sheet of paper, writes the number 7, and says to himself, "This is now my File A." However, he doesn't get to change what file *we* call File A. As far as we're concerned, the original file with 5 written on it is still File A. That's why the code in Figure 4.1.10 behaved the way it did: `addOne()` only changes what its variable `anInteger` points to, not what `myInteger` pointed to.

Immutable Data Types: Functions vs. Local Assignments

Let's alter the code a little bit to trace through this entire process. In Figure 4.1.12, we've added lines at the bottom to legitimately change the value of `myInteger`, and then print it again (lines 12 and 13).

Running it, what do we see? When we attempt to change the value of `myInteger` in our main code at the bottom, it works! When we attempt to change it in `addOne()`, it only works on the local variable within `addOne()`, `anInteger`. Now, let's trace through this and see why this happens.

As before, first we define the `addOne()` function on line 2. Then, in our main program code, we create `myInteger` and assign it the value 5 on line 7. That means that Python creates a memory spot and plops the value 5 in it, and then points `myInteger` to that memory spot. Then, on line 10, we call `addOne()`.

#	ImmutableDataTypesFunctionsvsLocalAssignments.py	Output
1	`#Add one to anInteger`	myInteger before addOne: 5
2	`def addOne(anInteger):`	anInteger: 6
3	`    anInteger = anInteger + 1`	myInteger after addOne: 5
4	`    print("anInteger:", anInteger)`	myInteger after increment: 6
5		
6	`#Create myInteger with the value 5`	
7	`myInteger = 5`	
8	`print("myInteger before addOne:", myInteger)`	
9	`#Call addOne on myInteger`	
10	`addOne(myInteger)`	
11	`print("myInteger after addOne:", myInteger)`	
12	`myInteger = myInteger + 1`	
13	`print("myInteger after increment:", myInteger)`	
14		

Figure 4.1.12

Python technically passes by reference, so `anInteger` is assigned to point to the same memory spot as `myInteger`. Were `myInteger` not immutable, that means if we changed `anInteger`, it would also change `myInteger` because they're pointing at the same spot in memory.

However, `myInteger` and `anInteger` are both immutable. So, when line 3 runs inside `addOne()`, the computer doesn't change the number stored at that memory spot to 6. Instead, it grabs a new memory spot, puts in the number 6, and tells `anInteger` to point to *that* memory spot instead. So, the 5 is still there, `anInteger` just isn't pointing to it anymore. So, when we print `anInteger`, we're printing the value of the new location to which it's pointing, which is now 6.

However, telling `anInteger` to point at 6 instead of 5 doesn't change where `myInteger` points. `myInteger` is still pointing at 5. So, when we exit the function, printing `myInteger` still prints 5. In line 12, though, we reassign `myInteger` to point at 6 as well. So, when we print `myInteger` in line 13, we now see it show the value 6.

Printing Memory Addresses

That all brings us to one last interesting thing we can do in Python. We mentioned above how every variable name actually points to a spot in memory, and when we change the value of an immutable variable (like an integer), we're actually changing where the variable name is pointing. To make this a little easier, we *can* print what spot of memory each variable is pointing at, as shown in Figure 4.1.13.

#	PrintingMemoryAddresses-1.py	Output
1	`myInt1 = 5`	1407565448
2	`#Print the spot in memory to which`	
3	`#myInt1 is pointing`	
4	`print(id(myInt1))`	
5		

Figure 4.1.13

The `id()` function tells us what spot in memory a variable is pointing to. Here, it shows us that the variable `myInt1` is pointing to the location in memory labeled 1407565448. If a data type is immutable, it just means that while the program is running, the data stored in that spot can't be changed. It's written in permanent marker, so to speak. Using this new function, we can see the difference between mutable and immutable data structures, as shown in Figure 4.1.14.

When we make a change to the immutable variable `myInt1` on line 5, the memory address to which it points changes, as shown on line 8. Notice that the first two `print()` statements in this code print `myInt1`'s memory address, but the memory address is different between the two. Line 5 changes what spot in

#	PrintingMemoryAddresses-2.py	Output
1	`myInt1 = 5`	1407565448
2	`#Print the spot in memory to which`	1407565464
3	`#myInt1 is pointing`	40885824
4	`print(id(myInt1))`	40885824
5	`myInt1 = 6`	
6	`#Print the spot in memory to which`	
7	`#myInt1 is pointing`	
8	`print(id(myInt1))`	
9		
10	`myList = ["One", "Two", "Three"]`	
11	`#Print the spot in memory to which`	
12	`#myList is pointing`	
13	`print(id(myList))`	
14	`myList.append("Four")`	
15	`#Print the spot in memory to which`	
16	`#myList is pointing`	
17	`print(id(myList))`	
18		

Figure 4.1.14

memory `myInt1` points to. `myInt1` is an immutable type, so to change its value, we have to change where it's pointing in memory.

When we make a change to the mutable variable `myList` on line 14, however, the memory address to which it points doesn't change. Notice that the `print()` statements on lines 13 and 17 print the same number, 40885824. Note, however, that this only applies to *changing* the mutable variable. If we reassign it, it also gets a new memory address, as shown in Figure 4.1.15.

#	PrintingMemoryAddresses-3.py	Output
1	`myList = ["One", "Two", "Three"]`	6217280
2	`#Print the spot in memory to which`	6217280
3	`#myList is pointing`	6239056
4	`print(id(myList))`	
5	`myList.append("Four")`	
6	`#Print the spot in memory to which`	
7	`#myList is pointing`	
8	`print(id(myList))`	
9	`myList = ["Five", "Six", "Seven"]`	
10	`#Print the spot in memory to which`	
11	`#myList is pointing`	
12	`print(id(myList))`	
13		

Figure 4.1.15

Running `append()` on line 5 changes the value of the variable, but it doesn't change its memory address. This is confirmed by the `print()` statements on lines 4 and 8 printing the same memory addressed. Reassigning it, however, does: notice that after running line 9 and assigning `myList` to a new list, the memory address for `myList` is changed, as shown by the `print()` statement on line 12. In this way, integers and strings really don't behave all that differently; they just don't have methods that change their values without reassigning them.

Finally, it's this strange oddity of Python that makes the code in Figure 4.1.16 print "True." `myInt1` and `myInt2` are separately assigned to the value 5; yet, because Python creates 5 in memory, it simply assigns both variables to point at the same spot. So, not only do `myInt1` and `myInt2` have the same value, but they also refer to the same spot in memory. This only works because integers are immutable: otherwise, every time two variables coincidentally took on the same value, they would start interfering with each other.

#	PrintingMemoryAddresses-4.py	Output
1	`myInt1 = 5`	`True`
2	`myInt2 = 5`	
3	`#Prints true if myInt1 and myInt2`	
4	`#point to the same spot in memory`	
5	`print(id(myInt1) == id(myInt2))`	
6		

Figure 4.1.16

Mutable variables will behave differently, as shown in Figure 4.1.17. Just like `myInt1` and `myInt2`, `myList1` and `myList2` are assigned the same value. Because they are mutable, though, Python creates that value twice in memory, assigning them to point at the different memory locations; so, it is `False` that their memory locations are equal, as shown by line 4. This is what allows us to call `append()` on `myList2` without affecting `myList1`, as shown by lines 6 through 9: had they pointed at the same memory location, changing the value via one variable name would have changed the value for the other variable name.

#	PrintingMemoryAddresses-5.py	Output
1	`myList1 = ["One", "Two", "Three"]`	`False`
2	`myList2 = ["One", "Two", "Three"]`	`['One', 'Two', 'Three']`
3		`['One', 'Two', 'Three', 'Four']`
4	`print(id(myList1) == id(myList2))`	
5		
6	`myList2.append("Four")`	
7		
8	`print(myList1)`	
9	`print(myList2)`	
10		

Figure 4.1.17

5. A Brief Introduction to Methods

So far, we've glossed over a bit of syntax and promised to return to it later. For example, in the previous lesson we saw it with `myList.append()`. We noted earlier that the dot separates the variable, `myList`, from something resembling a function, `append()`. We've seen similar things in other places as well, like `myString.isdigit()`. We noted at the time to just remember what each individual example of this kind of syntax does, and not worry too much about the syntax.

Going forward into more advanced data structures, however, this syntax is going to become more common. We'll talk about this more extensively when we discuss encapsulation in Chapter 5.1, but because we'll see it more often in this unit, this is a good time to pause and comment a little more on what this is.

Functions vs. Methods

Previously we noted that this dot syntax behaved essentially like a function, and that's still true. Just like functions, they have names, parameters, some internal operations or code, and they may or may not return some value. The way we've used them so far, they're pretty indistinguishable from functions.

Methods
Functions that are contained within data types.

The difference is subtle. The functions we've defined have been defined at the top level of our programs. We've created them before actually writing the body of our programs, and that's why they're visible. **Methods**, on the other hand, are *contained within data types*. Instead of just calling them directly like functions, we have to first say which variable we're referring to, and *then* call the method inside them. So, a method is a function contained inside a data structure.

Methods in Practice

What does this mean for us in practice? So far, we've mostly been manipulating our variables through operators. Operators work on simple data types. Going forward, we're mostly going to be manipulating our variables using methods and functions. The reasoning necessary to manipulate advanced data types is too complex to be taken care of by simpler operators.

For now, though, you don't need to worry too much about how methods work or the meaning behind their syntax; we'll cover that in Chapter 5.1. For now, you really just need to understand what the effect of a method call will be. Let's take a simple string method we've seen in the past: `isdigit()`.

`isdigit()` is a boolean method that is part of the string data structure. It checks if the string represents a number. Note that just like functions, methods can have types as well: `isdigit()` returns a `True` or `False` value, but other methods could return integers, strings, or more complex data types just like functions. If we try to think of `isdigit()` as a function, though, we're faced with a problem: how does `isdigit()` know *which* string to check? It doesn't have any parameters! We can imagine an `isdigit()` function with the header `def isdigit(aString)`, where it would check if `aString` is all digits. But the method `isdigit()` as defined has no parameters; how does it know what string to check?

The answer lies in the fact that `isdigit()` is contained *within* the string data type. That means every string has access to the `isdigit()` method. When called, the `isdigit()` method acts on whatever string called it, as shown in Figure 4.1.18. When we call `myNumericString.isdigit()`, `isdigit()` looks at `myNumericString`. When we call `myNonNumericString.isdigit()`, `isdigit()` checks `myNonNumericString`.

#	MethodsinPractice.py	Output
1	`myNumericString = "12345"`	`True`
2	`myNonNumericString = "ABCDE"`	`False`
3		
4	`#Prints True if myNumericString is digital`	
5	`print(myNumericString.isdigit())`	
6	`#Prints True if myNonNumericString is digital`	
7	`print(myNonNumericString.isdigit())`	
8		

Figure 4.1.18

In the past, we've used the example of Addison, where Addison was a function. Now let's take another example. Imagine Dana is another co-worker, but let's treat Dana as a variable of type `Coworker`. The `Coworker` type might have a method `getMyName()`, which is the equivalent of asking a co-worker for their name. When someone asks Dana her name, she doesn't have to holler to some other function in the workplace and ask, "Hey, what's my name?" She knows her name; it's "Dana." In fact, every co-worker knows their own name, but for every co-worker the answer is different. They all have the same method, `getMyName()`, which requires no input and returns a different result for each co-worker. The reason why is when asked their name, each co-worker knows to return their *own* name.

That's like a method. A method is a function defined within a data type that must be called from a particular variable. When it's called, it knows to look at the variable from which it's called. `myNumericString.isdigit()` knows to check `myNumericString`, not `myNonNumericString`, just as Dana knows to give her own name, not Vrushali's name when asked.

Equivalent Syntax

If this is still confusing, don't worry. Like I've said, you don't really need to understand this too deeply until we get to Chapter 5.1. This is good exposure, but you're safe to move forward without understanding this fully.

If this is still confusing but functions made sense, then there's a little equivalent syntax you can think of that will be true for the rest of this unit. A method is like a function that takes the variable referencing it as its first parameter. `myString.isdigit()` can be thought of largely as the same as `isdigit(myString)`, where `isdigit()` is a function that checks if `myString` is all digits, as shown in Figure 4.1.19. This won't work when we reach Chapter 5.1, but it's sufficiently equivalent to let you proceed with Unit 4.

#	EquivalentSyntax.py	Output
1	`import string`	True
2	`#Return True if inString contains all digits`	True
3	`def isdigit(inString):`	
4	`    #Check each character one-by-one`	
5	`    for character in inString:`	
6	`        #Check if the character is a digit`	
7	`        if not character in string.digits:`	
8	`            #Return false if not`	
9	`            return False`	
10	`    #Return true if we reached here`	
11	`    return True`	
12		
13	`myString = "52672"`	
14	`print(isdigit(myString))`	
15	`print(myString.isdigit())`	
16		

Figure 4.1.19

CHAPTER

4.2

Strings

1. What Are Strings?

At this point, we've already used strings a lot. It's hard to even start writing programs that can be followed or do interesting things without using strings—at least in their simple form. We speak in natural language, so strings let us have our programs output natural language for us to understand.

However, **strings** are also pretty complex data structures; we've barely scratched the surface of how they're used and why they're valuable. At a fundamental level, strings are lists of individual characters; some languages actually represent them as such, while others represent individual characters as strings with length one. Either way, though, strings are often treated as lists of individual characters.

String and Alphabets

What makes strings complex, then? It seems like a list of letters, numbers, and symbols would be a simple data structure. What makes strings complex is of us pesky humans. Strings represent human alphabets and human languages. The alphabet is surprisingly complex. Take, for example, something as simple as capital and lower-case letters. We pretty easily see 'A' and 'a' as two characters for the same letter, to be used in different contexts. We know that we would use 'A' if we're starting a sentence, starting someone's name, or writing a book title; in other situations, we know to use 'a'.

To the computer, though, 'a' and 'A' are entirely different **characters**, as different as b and Q. You might have seen that before: if you declared a variable with one capitalization scheme in one place, like `myInteger`, and another in another place, like `MyInteger`, the computer saw those as entirely different variables. It has no understanding that "m" and "M" are the same character unless we build such understanding into it.

This is where a lot of string complexity comes into play: we want to manipulate strings based on the way we actually represent human language, even though the computer has little knowledge of human language. For example, if we want to sort things alphabetically, we want to the computer to view 'A' and 'a' as equivalent characters On its own, it doesn't do so, and this can lead to strings getting sorted with all uppercase letters before any lowercase letters. Things like converting letters to uppercase or removing trailing spaces in some text make perfect sense to us, but are arbitrary rules to the computer.

Unicode Characters

The relationship with the human alphabet is only half the story, however. Ask yourself: what are characters? You might rightly say they're the keys on your keyboard: letters, numbers, and some symbols. Those are certainly characters, but they aren't all the characters. You might also rightly say that a symbol doesn't have to be on the keyboard to be a character; ∞, Δ, →, •, and ÷ are all characters as well that you could include in your strings. Even emojis are technically characters: 😁, 👩, and 🌀 are characters, too.

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Examine the complexities involved in strings, including the role of Unicode in establishing character standards and use of special characters;
- Write programs to declare strings using multiple declaration styles, perform slice and concatenate operations, and use special characters as escape sequences;
- Write programs to perform string search and implement methods to modify strings;
- Modify turtle program to accept string input and generate processed output.

String
A data structure that holds a list, or a string, of characters.

Character
A single letter, number, symbol, or special character.

Unicode
A computing industry standard that sets what hexadecimal codes correspond to what characters, so that text appears consistent across platforms.

Hexadecimal
A short-hand expression of the ones and zeroes that comprise computer data, comprised of 16 characters, 0 through 9 and A through F.

Characters are defined by **Unicode**, a computing industry standard for handling text. Unicode defines hundreds of different characters in terms of their codes in **hexadecimal**, which is close to ones and zeroes inside the computer. Think about it: if a file is distilled all the way down to ones and zeroes, then when you receive a file, how does your computer know to render an "A" where the original author wrote an "A"? The answer is that both computers use the Unicode standard: they know when they see a character with the hexadecimal code 0041, that should translate to an A. That A might be rendered in different fonts or with different style, but it's still the character A. The same way, every computer knows that 263A should be a simple smiley emoticon. It might be displayed differently based on whether you're on Facebook, Twitter, or Microsoft Word, but the underlying standard is for that character to be a smiley face.

Special Characters

We're not talking about this just so that we can start to use emojis in our code, of course (though feel free!). The reason this is complex is that *everything* plaintext-related in computing is communicated through Unicode characters. By plaintext, we mean without formatting like font face, font size, bold and italics, color, and so on. We mean anything that can be expressed in a plaintext editor, like Notepad.

Yet, there are a lot of things contained there that you wouldn't immediately recognize as characters. For example, you press Enter or Return to start a new line; how does that computer know to start a new line? Technically, there is a **newline character**. It's invisible, you don't see it, and yet it's there: just as the computer shows the "a" character as the letter a, it shows a new line character as a break down to the next line.

Newline Character
A Unicode character, either LF (line feed) or CR (carriage return), that is rendered as the beginning of a new line of text.

There are lots of these "invisible" characters. Tabs, for instance, are characters that are rendered to have a certain width of whitespace. Paragraphs are their own characters, different from newline characters. There are actually two newline characters: carriage returns and line feed. Have you ever created a plain text file on Mac OS or Unix and then opened it on Windows, only to find everything was on one line? That's because Mac OS uses the line feed character to represent its new line, while Windows uses both carriage returns and line feeds. When Windows sees just line feeds, it doesn't render them the same way.

InputFile.txt	Output
A	A
B	
C	B
D	
E	C
	D
	E

Figure 4.2.1

Why does all this matter? The fact that these are stored as characters can alter the way we do some of our string manipulations. For example, let's say we load five lines from a file, and print them each on their own line. Lo and behold, we might find that there's a blank line between each pair of lines. Why? Because when we loaded the lines from a file, there was a new line character as *part* of the line. When we then printed each on its own line, it printed an additional new line character: the one in the string, and the one we printed between lines.

So, the conclusion is: strings are simply lists of characters. However, characters themselves are quite complex, between the relationships within the human alphabet and the special characters supplied by Unicode, and that can make string formatting pretty complex. The very fact that our code is also written in text is a good example of this complexity as well: we differentiate text in our code from the code itself using quotation marks, but what if we want to actually print quotation marks? The fact that we write code in text makes this an interesting little challenge.

2. Declaring Strings in Python

I know what you're probably thinking: we've declared lots of strings, why do we need to cover this again? So far, we've declared strings in a natural way, but we haven't really talked about what we're actually doing, or about how to deal with some interesting edge cases. So, let's look at string declaration in a little more detail.

Three Ways to Declare Strings

So far, we've always used one method to declare strings, as shown in Figure 4.2.2.

#	ThreeWaystoDeclareStrings-1.py	Output
1	`myString = "12345"`	12345
2		
3	`print(myString)`	
4		

Figure 4.2.2

`myString` on line 1 is the variable name, and to tell it that its data type is a string, we enclose the value assigned to it on the right in quotation marks. If we didn't, it would become an integer with value 12345. Simple enough, let's move on… except, there's a special case we can't handle here. What do we do if we want to put a quotation mark *inside* a string? Right now the string's value is the text 12345; what if we wanted its value to be "12345", where the quotation marks were actually *part* of the value? If we just write the obvious declaration, we get Figure 4.2.3.

#	ThreeWaystoDeclareStrings-2.py	Output
1	`myString = ""12345""`	`File "ThreeWaystoDeclareStrings-2.py", line 1`
2		`    myString = ""12345""`
3	`print(myString)`	`                       ^`
4		`SyntaxError: invalid syntax`
5		

Figure 4.2.3

In Figure 4.2.3, Python gives us an error. How come? A quotation mark opens and closes the string. How could the computer know the second quotation mark was supposed to be a part of the string rather than the end of the string? Here, Python sees this as an empty string "", followed by the number 12345, followed by another empty string "", with no operators. It doesn't know how to interpret that.

So, how can we include quotation marks in the string? Fortunately, Python gives us three ways to define strings, and we can use whichever way is compatible with the string content we want, as shown in Figure 4.2.4.

#	ThreeWaystoDeclareStrings-3.py	Output
1	`myString1 = '"12345"'`	`"12345"`
2	`myString2 = "'12345'"`	`'12345'`
3	`myString3 = '''"'12345'"'''`	`"'12345'"`
4	`myString4 = "'''12345'''"`	`'''12345'''`
5		
6	`print(myString1)`	
7	`print(myString2)`	
8	`print(myString3)`	
9	`print(myString4)`	
10		

Figure 4.2.4

We can use quotation marks, apostrophes, or triple-apostrophes to declare strings. If we want our string to include quotation marks, we can instead declare it with apostrophes. If we want our string to include apostrophes, we can instead declare it with quotation marks. If we want our string to include both quotation marks and apostrophes, then we can declare it with triple-apostrophes (or triple quotation marks, even!). No matter what we use to start the string, the string does not end until we encounter that character or character sequence again.

Special Characters

We mentioned previously that everything in text is technically a character, including things like tabs and line breaks. Can we then include these in our strings? Yes,

```
#  SpecialCharacters-1.py
1  myStringWithNewline = "12345
2                        67890"
3  print(myStringWithNewline)
4
5
6
7
```

```
Output
File "SpecialCharacters-1.py",
line 1
    myStringWithNewline = "12345
                                ^
SyntaxError: EOL while scanning
string literal
```

Figure 4.2.5

though we have to know how. If we try to just put a newline character into the middle of a string, Python isn't sure what to make of it, as shown in Figure 4.2.5.

Python sees line 1 as ending before the string is closed because Python interprets the newline character in terms of the code, not as part of the string. In other words, Python sees the newline character at the end of line 1 as terminating line 1 with an unclosed string, and it sees line 2 as starting with an unopened string. We need a character that tells Python, "Hey, when running this code, include this newline as part of this string." It's shown in Figure 4.2.6.

```
#  SpecialCharacters-2.py
1  myStringWithNewline = "12345\n67890"
2
3  print(myStringWithNewline)
4
```

```
Output
12345
67890
```

Figure 4.2.6

The character sequence `\n` in line 1 is translated by Python into the newline character. Whenever it sees `\n` it prints a newline. So, here, the newline character appears between "12345" and "67890" within the string, so Python prints a line break between the 5 and the 6.

Escape Sequence
A sequence of characters that, when occurring in a string, is interpreted to have a meaning beyond the characters themselves. The most common example is "\n", which is interpreted by many languages as representing a newline character.

This actually represents a general principle: inside a string, the forward slash is called an "escape" character, and it starts an **escape sequence**. When Python sees the forward slash, it tries to interpret it and the next character as a special sequence that carries special meaning. The `\n` sequence carries the special meaning "newline." A couple of others are shown in Figure 4.2.7.

```
#  SpecialCharacters-3.py
1  myString = "12345\n67890\tabcde\"fghijklm\\no"
2
3  print(myString)
```

```
Output
12345
67890    abcde"fghijklm\no
```

Figure 4.2.7

In Figure 4.2.7, we see four escape sequences:

- `\n`, which inserts a new line.
- `\t`, which inserts a tab.
- `\"`, which inserts a quotation mark without terminating the string (another way to include quotation marks and apostrophes inside strings).
- `\\`, which inserts a forward slash without interpreting it as an escape character (note that otherwise, the \n at the end of the string would have been a new line).

Note that it's also alright to simply put a tab directly into the string by pressing Tab inside of typing `\t`. However, most Python development environments will translate this into four spaces instead of a tab, so use `\t` to have a true tab character.

Note also that our triple-apostrophe method would let us simply write the new line directly, as shown in Figure 4.2.8. Python interprets strings started by triple apostrophes differently, and allows new lines in them.

#	SpecialCharacters-4.py	Output
1	`myString = '''12345`	12345
2	`67890'''`	67890
3		
4	`print(myString)`	
5		

Figure 4.2.8

3. String Concatenation and Slicing in Python

We now have the ability to define lots of strings. What can we do with them? Here, we'll talk about two common operations we want to do on strings: concatenation and slicing.

String Concatenation

String concatenation means putting multiple strings together. It comes from the word "concatenate," which simply means to link things together in an ordered series or chain. We actually can just do this with the + operator, as shown in Figure 4.2.9.

String Concatenation
The process of putting two or more strings together in order to form one string made of the individual strings. For example, concatenating "A" with "B" would give "AB".

```
#  StringConcatenation-1.py
1  myString1 = "12345"
2  myString2 = "67890"
3  myString3 = myString1 + myString2
4  print("Assignment Concatenation: " + myString3)
5  print("In-Line Concatenation: " + myString1 + myString2)
6  myString1 += myString2
7  print("Self-Assignment Concatenation: " + myString1)

Output
Assignment Concatenation: 1234567890
In-Line Concatenation: 1234567890
Self-Assignment Concatenation: 1234567890
```

Figure 4.2.9

In Figure 4.2.9, we see three kinds of concatenation. We declare `myString1` and `myString2` on lines 1 and 2, then concatenate them and assign the result to `myString3` on line 3. The result is the two strings squished together into one, as shown when we print `myString3` on line 4. Then, on line 5, we do the same thing in-line: they don't have to be assigned to a separate variable to do this. Then, on line 7, we also do this with self-assignment concatenation: we set `myString1` equal to the concatenation of itself and `myString2`, and line 8 confirms the results are the same. So, generally, the + operator tacks each string on to the previous string. Note that technically, we're seeing string concatenation with the labels, too. Line 4 technically concatenates "Assignment Concatenation: " and the value of `myString3` together to print them as one string, "Assignment Concatenation: 1234567890".

Notice how this works if we throw in a newline character in Figure 4.2.10. When we print a string with a newline character inside it on line 3, we get an extra blank line: `print()` automatically ends with a newline character, so having one inside the string adds a second newline. Then, when we concatenate `myString1` and its

```
# StringConcatenation-2.py
1 myString1 = "12345\n"
2 myString2 = "67890"
3 print(myString1)
4 print(myString2)
5 print("In-Line Concatenation: " + myString1 + myString2)
```

```
Output
12345

67890
In-Line Concatenation: 12345
67890
```

Figure 4.2.10

newline character with `myString2` on line 5, printing them together breaks the second half onto its own line; technically, we're printing "In-Line Concatenation: 12345\n67890", and Python interprets the `\n` as the escape sequence for a newline.

String Slicing: Individual Characters

String Slicing
The Python term for obtaining substrings from within a string based on character indices.

String slicing is Python's term for finding substrings within a broader string. Imagine you have a string: how do you get just the first 5 characters? The last 5? The middle 5? String slicing is the answer.

Let's start simple. How would you get just a single character out of a string? Python has some dedicated syntax for that: brackets. Whenever you have a data structure that is a list of multiple items, you can follow the variable name with brackets and a number, called the index, to get a certain item from the list.

```
# StringSlicingIndividualCharacters-1.py
1 myString = "ABCDE"
2 #Print the 0th (1st) character
3 #of the string
4 print(myString[0])
5 #Print the 1st (2nd) character
6 #of the string
7 print(myString[1])
8
```

```
Output
A
B
```

Figure 4.2.11

The tricky thing here is that Python treats the first item in a string as the "0th" item. So, to get the first item, you ask for the 0th item, as shown in line 4 of Figure 4.2.11. This is called zero-indexing: the indices of a list start with 0. It also means that the last item in the list is one fewer than the length of the list, and if we try to access a list item that doesn't exist, we get an error, as shown in line 7 of Figure 4.2.12.

```
# StringSlicingIndividualCharacters-2.py
1 myString = "ABCDE"
2 #Print the 4th (5th) character
3 #of the string
4 print(myString[4])
5 #Print the 5th (6th) character
6 #of the string
7 print(myString[5])
8
```

```
Output
E
Traceback (most recent call last):
File "...", line 5
    print(myString[5])
IndexError: string index out of range
```

Figure 4.2.12

`IndexError`s are the error that arises when we try to access an index that doesn't exist for a particular list. In Figure 4.2.12, the index 5 does not exist because the characters in a five-character string are numbered 0, 1, 2, 3, and 4. Remember also, strings are immutable, which means we cannot use this syntax to change individual characters, as shown in line 2 of Figure 4.2.13.

#	StringSlicingIndividualCharacters-3.py	Output
1	`myString = "ABCDE"`	`Traceback (most recent call last):`
2	`myString[4] = "F"`	`File "...", line 2`
3		`    myString[4] = "F"`
4		`TypeError: 'str' object does not`
5		`support item assignment`
6		

Figure 4.2.13

Why **zero-indexing**? This goes all the way back to the early days of computing. Remember, a variable points to a place in memory where the value is stored. Early on, lists were technically just consecutive locations in memory. The index told the list how many places in memory to skip. To get the first item in a list, you wouldn't skip any places in memory, so your index would be 0. Saying "skip 5 items" with a 5-item list would skip the entire list, triggering the `IndexError`.

Zero-Indexing
A convention in most programming languages where the first item of a list of items is considered the "0th" item, not the 1st item.

String Slicing: Substrings

But what if you want to create a string made up of a part of another string? What if you wanted to grab the first three characters and create a new string? Remember, we can traverse a string with a for-each loop, as shown in Figure 4.2.14.

#	StringSlicingSubstrings-1.py	Output
1	`myString = "ABCDE"`	`mySubstring: ABCDE`
2	`mySubstring = ""`	
3	`#Run a loop for each character in myString`	
4	`for character in myString:`	
5	`    #Add character i to mySubstring`	
6	`    mySubstring += character`	
7	`print("mySubstring: " + mySubstring)`	
8		

Figure 4.2.14

The fact that we can traverse each character of a string that way is a useful takeaway on its own. However, there's no straightforward way to stop the `for-each` loop on line 4 before it reaches the end of the string, and in this example, we wanted to grab just the first three characters. So, we could do this the hard way, with a regular `for` loop from 0 to 2, as shown in Figure 4.2.15.

#	StringSlicingSubstrings-2.py	Output
1	`myString = "ABCDE"`	`First three characters: ABC`
2	`mySubstring = ""`	
3	`#Run a loop from i = 0 to i = 2`	
4	`for i in range(0, 3):`	
5	`    #Add character i to mySubstring`	
6	`    mySubstring += myString[i]`	
7	`print("First three characters: " + mySubstring)`	
8		

Figure 4.2.15

We run a `for` loop from 0 to 2, grabbing characters 0, 1, and 2 from `myString` and adding them to `mySubstring`. That was a lot of work to do that; there has to be a better way. One better way would be to create a function that just takes as parameters the string and the number of characters, and in fact, that's how many languages

```
# StringSlicingSubstrings-3.py
1 myString = "ABCDE"
2 start = 0
3 end = 3
4 print("First three characters: " + myString[start:end])

Output
First three characters: ABC
```

Figure 4.2.16

do this. However, Python makes this even simpler with its string slicing syntax, as shown in Figure 4.2.16.

In Figure 4.2.16, we're using the same brackets on line 4 that we used before, but instead of just putting a single number, we put two numbers with a colon in-between. The number before the colon is the start index, and the number after is the end index. The substring will be the characters from start (inclusive) to end (exclusive). In other words, it will include the character with the starting index, and stop before the character with the end index. So in line 4 of Figure 4.2.16, the substring is from 0 to 3, so characters 0, 1, and 2 are included.

One nice thing about this is that we don't need to use variables: we can put in the indices directly, as shown in lines 2 and 3 of Figure 4.2.17. That's much simpler than our for loop!

```
# StringSlicingSubstrings-4.py
1 myString = "ABCDE"
2 print("First three characters: " + myString[0:3])
3 print("Characters 1 through 3: " + myString[1:4])
4

Output
First three characters: ABC
Characters 1 through 3: BCD
```

Figure 4.2.17

Python also takes this a step further by allowing us to omit either the start or end, as shown in Figure 4.2.18. If we skip the "start" number as shown in line 2, Python assumes 0. If we skip the "end" number as shown in line 3, Python assumes "to the end." If our "end" number is beyond the length of the string as shown in line 4, then Python stops at the end of the string.

```
# StringSlicingSubstrings-5.py
1 myString = "ABCDE"
2 print("First three characters: " + myString[:3])
3 print("Characters from 3 to the end: " + myString[3:])
4 print("Characters from 3 to 10: " + myString[3:10])

Output
First three characters: ABC
Characters from 3 to the end: DE
Characters from 3 to 10: DE
```

Figure 4.2.18

Negative Indices

Those methods all covered getting things like the "first five characters" or "characters three through eight." How do we get the last several characters in a string, though? What if we wanted to get the last two letters of a string?

We could do it the hard way. The hard way would be to use the length of the string to figure out what index to start at to get the last few characters. For example, if we wanted the last two characters, we'd start at the length of the string minus two, as shown in Figure 4.2.19.

```
# NegativeIndices-1.py
1 myString = "ABCDE"
2 print("Last two characters: " + myString[len(myString)-2:])

Output
Last two characters: DE
```

Figure 4.2.19

That works, but that's really complex. We have to find the end of the string, count backward by two, then count forward again. It works just fine, and in many languages this is exactly what you have to do. Python tries to make things easier, though, with negative indices. Negative indices (i.e., using a negative number as an index) count backward from the end of the string, as shown in Figure 4.2.20.

```
# NegativeIndices-2.py
1 myString = "ABCDE"
2 print("All but the last two characters: " + myString[:-2])
3 print("Last two characters: " + myString[-2:])

Output
All but the last two characters: ABC
Last two characters: DE
```

Figure 4.2.20

Line 2 here says to start at the beginning (because there is no "start" index in the brackets), and go until 2 characters *from* the end; this is the meaning of -2 as the "end" index. So, an index of 2 for "end" would mean to include the characters until 2 from the *start*, and an index of –2 for "end" would mean to include the characters until 2 from the *end*. Similarly, if our "start" index is –2 as on line 3, it means to *start* two indices from the end.

4. String Searching in Python

Back when we talked about logical operators, we talked about string equality. Specifically, we talked about how two strings are equal if they have the same characters, and one string is "greater" than another if it comes later alphabetically. Earlier in this chapter, we briefly noted that natively, many languages—Python included—process uppercase and lowercase letters separately, meaning that all the uppercase letters will be sorted among themselves before any lowercase letters.

Those rules generally take care of string comparisons, whether two strings are equal or if one is "greater" than another. However, there's more we can do with strings, including checking to see if a substring is present in a string and, if so, where.

The In Operator

When we covered operators, we talked about the `in` operator in Python. The `in` operator is unique; it seems to take on different meanings when used in a `for` loop (e.g., `for i in range(0, 3):`) and in a conditional (`if "Bob" in myList:`).

The `in` operator can be used with strings to check if a substring is part of a string, as shown in Figure 4.2.21. The conditional on line 3 correctly identifies that "BC" is in the string "ABCDE", and the conditional on line 8 correctly identifies that "GH" is *not* in the string.

#	TheInOperator-1.py	Output
1	`myString = "ABCDE"`	BC was found!
2		GH was not found!
3	`if "BC" in myString:`	
4	`    print("BC was found!")`	
5	`else:`	
6	`    print("BC was not found!")`	
7		
8	`if "GH" in myString:`	
9	`    print("GH was found!")`	
10	`else:`	
11	`    print("GH was not found!")`	
12		

Figure 4.2.21

In reality, we would probably package this together as a function, as shown in Figure 4.2.22. So, we can use the `in` operator to check to see if a string is present in another string.

#	TheInOperator-2.py	Output
1	`#Checks if searchString is in checkString`	BC was found!
2	`def checkInString(checkString, searchString):`	GH was not found!
3	`    if searchString in checkString:`	
4	`        print(searchString + " was found!")`	
5	`    else:`	
6	`        print(searchString + " was not found!")`	
7		
8	`myString = "ABCDE"`	
9	`checkInString(myString, "BC")`	
10	`checkInString(myString, "GH")`	
11		

Figure 4.2.22

Similarly, we could use `not in` to check the inverse, as shown in Figure 4.2.23. Same result here, but the reasoning is in reversed. If it's *not* in the string, we print that it's not in the string; else, we print that it is.

#	TheInOperator-3.py	Output
1	`#Checks if searchString is not in checkString`	BC was found!
2	`def checkNotInString(checkString, searchString):`	GH was not found!
3	`    if searchString not in checkString:`	
4	`        print(searchString + " was not found!")`	
5	`    else:`	
6	`        print(searchString + " was found!")`	
7		
8	`myString = "ABCDE"`	
9	`checkNotInString(myString, "BC")`	
10	`checkNotInString(myString, "GH")`	
11		

Figure 4.2.23

The Find Method

find(text, [start], [end])
A method of the string data type that will find the first instance of the value of text within the string calling the method. Optionally, also takes parameters start and end to mark where to search in the string.

Sometimes, though, we're not just interested in finding out *if* a string is in another string. Oftentimes, we want to know *where* it was found. That's where the powerful Find method comes in handy. The Find method, `find()`, is a member of the string type, and it takes as input the substring to find, then returns the index where it was found or –1 if it was not found.

In Figure 4.2.24, "CDE" starts at index 2 in `myString` (due to zero-indexing, "A" is 0, "B" is 1, and "C" is 2, so "CDE" starts at 2), so `myString.find("CDE")` on line 4 returns 2. "ACE" is not found in `myString`—while each individual character is in "ABCDE", the continuous string "ACE" is not found. So, `myString.find("ACE")` on line 6 returns –1.

#	TheFindMethod-1.py	Output
1	`myString = "ABCDE"`	2
2		-1
3	`#Prints the index of "CDE" in myString`	
4	`print(myString.find("CDE"))`	
5	`#Prints the index of "ACE" in myString`	
6	`print(myString.find("ACE"))`	
7		

Figure 4.2.24

Note that in this way, `find()` subsumes all the reasoning of the `in` operator, as shown in Figure 4.2.25. If the result of `find()` is positive on line 3, it means that the substring was found; if it's negative, it means it wasn't found. What happens, though, if the string we're trying to find is in two places in the string that we're searching?

#	TheFindMethod-2.py	Output
1	`#Checks if searchString is in checkString`	BC was found!
2	`def checkInString(checkString, searchString):`	GH was not found!
3	`    if checkString.find(searchString) >= 0:`	
4	`        print(searchString + " was found!")`	
5	`    else:`	
6	`        print(searchString + " was not found!")`	
7		
8	`myString = "ABCDE"`	
9	`checkInString(myString, "BC")`	
10	`checkInString(myString, "GH")`	
11		

Figure 4.2.25

As shown on line 4 of Figure 4.2.26, `find()` only finds the first index; after all, it can only return one number. We'll talk in a moment about how to use `find()` more flexibly.

#	TheFindMethod-3.py	Output
1	`myString = "ABCDEABCDE"`	2
2		
3	`#Prints the index of "CDE" in myString`	
4	`print(myString.find("CDE"))`	
5		

Figure 4.2.26

Before that, though, note also that the `find()` method is case-sensitive, as shown in Figure 4.2.27. Remember, the computer doesn't see "c" and "C" as the same character; they're as different as "b" and "Q". So, searching for "cde" on line 4 won't turn up anything in "ABCDEABCDE".

#	TheFindMethod-4.py	Output
1	`myString = "ABCDEABCDE"`	-1
2		
3	`#Prints the index of "cde" in myString`	
4	`print(myString.find("cde"))`	
5		

Figure 4.2.27

Parameters of the Find Method

We can extend `find()` by using some of its optional parameters. Optionally, we can supply two additional arguments to the `find()` method: `start` and `end`. `start` tells `find()` where to start looking, and `end` tells it where to stop looking. If it isn't found after `start` and before `end`, it returns –1.

Figure 4.2.28 shows five different `find()` calls on `myString` using "CDE". On line 4, it finds the first index of "CDE" at 2. On line 6, it searches only after the index 5; the first occurrence of "CDE" after the index 5 is at 7. On line 8, it searches only after the index 13; "CDE" doesn't occur after 8, though, so it returns –1 to say the string was not found. On line 10, it searches only between the indices 4 and 10; the first occurrence there is at 7. In this way, it skips both the first and last overall appearances and only gets the one in the middle. Then, on line 12, it searches between 3 and 6, but finds nothing and returns –1.

#	ParametersoftheFindMethod-1.py	Output
1	`myString = "ABCDEABCDEABCDE"`	2
2		7
3	`#Prints the first index of "CDE" in myString`	-1
4	`print(myString.find("CDE"))`	7
5	`#Prints the first index of "CDE" in myString after 5`	-1
6	`print(myString.find("CDE", 5))`	
7	`#Prints the first index of "CDE" in myString after 8`	
8	`print(myString.find("CDE", 13))`	
9	`#Prints the first index of "CDE" in myString between 4 and 10`	
10	`print(myString.find("CDE", 4, 10))`	
11	`#Prints the first index of "CDE" in myString between 3 and 6`	
12	`print(myString.find("CDE", 3, 6))`	
13		

Figure 4.2.28

We can use `find()` to build a list of all the appearances of a particular string within another string. We'll talk about making it a list later; for now, let's just print out all the indices. Figure 4.2.29 shows the code to do this—we'll make the string we're searching a little longer and more complicated to make things interesting.

#	ParametersoftheFindMethod-2.py	Output
1	`myString = "ABCDEABCDEABCDEFGHIJFGHIJABCDEABCDEFGHIJ"`	CDE found at 2
2	`findString = "CDE"`	CDE found at 7
3	`#Find findString in myString and assign its index to currentLocation`	CDE found at 12
4	`currentLocation = myString.find(findString)`	CDE found at 27
5		CDE found at 32
6	`#While currentLocation is positive; e.g. while findString is found`	
7	`while currentLocation >= 0:`	
8	`    #Print the index`	
9	`    print(findString, "found at", currentLocation)`	
10	`    #Get the next index, or -1 if there are no more`	
11	`    currentLocation = myString.find(findString, currentLocation + 1)`	
12		

Figure 4.2.29

We first create the string to search on line 1, `myString`, and the string to search for on line 2, `findString`. Then on line 4, we get the first location of `findString` in `myString` and assign it to `currentLocation` to get our loop started. Then starting on line 7, we run a loop while `currentLocation` is greater than 0, and on line 11 at the end of each iteration of that loop, we search `myString` again *starting at* the last found location (as given by the `currentLocation + 1` parameter). Since `find()` always finds the first instance after a certain index, that guarantees we'll find

the instances in order. When we run out, that means `findString` wasn't found, so `find()` will return –1, terminating the while loop.

So, let's step through this. Initially, `currentLocation` is assigned to 2, the first location of "CDE". 2 >= 0 is `True`, so the `while` loop runs, printing that "CDE" was found at 2. Then, it searches again, starting at index 3. Because the search starts at index 3, it won't find the instance starting at 2 again. Instead, it finds the next one, at 7. 7 >= 0 is still `True`, so it prints and finds the next location. This continues for 12, 27, and 32. After 32, though, this code calls `find()` starting at index 33. There are no instances of "CDE" after 32, though, so this call to `find()` returns –1. Now, it's not `True` that –1 >= 0, so the `while` loop terminates.

Note that if `findString` is not found anywhere in `myString`, then the initial assignment to `currentLocation` will be –1, and the while loop will never run even once, as shown in Figure 4.2.30. In this way, we can build a segment of code that gathers every instance where a particular string was found in another string.

#	ParametersoftheFindMethod-3.py	Output
1	`myString = "ABCDEABCDEABCDEFGHIJFGHIJABCDEABCDEFGHIJ"`	CDE found at 2
2	`findString = "BOB"`	CDE found at 7
3	`#Find findString in myString and assign its index to currentLocation`	CDE found at 12
4	`currentLocation = myString.find(findString)`	CDE found at 27
5		CDE found at 32
6	`##While currentLocation is positive; e.g. while findString is found`	
7	`while currentLocation >= 0:`	
8	`    #Print the index`	
9	`    print(findString, "found at", currentLocation)`	
10	`    #Get the next index, or -1 if there are no more`	
11	`    currentLocation = myString.find(findString, currentLocation + 1)`	
12		

Figure 4.2.30

Note that we can also use a different method, `count()`, to simply count the instances without finding them, as shown in Figure 4.2.31. This confirms there are five instances of "CDE" in `myString`. `count()` can also take the same parameters as `find()`, `start` and `end` to mark off within what portion of the string it should count. Note also this is a somewhat complex `print()` statement on line 3: we have our label, "Count of", which is concatenated with the value of `findString` and a colon. That is then also concatenated with the result of `myString.count(findString)`.

#	ParametersoftheFindMethod-4.py	Output
1	`myString = "ABCDEABCDEABCDEFGHIJFGHIJABCDEABCDEFGHIJ"`	Count of CDE : 5
2	`findString = "CDE"`	
3	`print("Count of", findString, ":", myString.count(findString))`	
4		

Figure 4.2.31

5. Useful String Methods in Python

Last chapter, we covered methods at a high level, just to familiarize ourselves with method syntax. The reason was that although we have seen a few methods before, now we're going to start seeing them even more frequently. That starts here with strings. Strings in Python have lots of useful methods for us to use. Let's take a look at several of them.

Split()

The `split()` method divides the string up into several substrings based on the separator character. The simplest case, when no arguments are given to `split()`, is that it splits the string up by spaces, as shown in Figure 4.2.32.

In Figure 4.2.32, `myString` is a long string with 13 words. When we call `myString.split()` on line 3, it splits it up by the space character. The result is

`split([separator])`
A method of the string data type that will split a string up into a list of smaller strings. If a separator string is given, that string will be used to determine where to split; if not, the string will be split by spaces.

```
# Split-1.py
myString = "This is my text. It has thirteen " \
           "words. It also has three sentences."
print(myString.split())
```

```
Output
['This', 'is', 'my', 'text.', 'It', 'has', 'thirteen', 'words.',
'It', 'also', 'has', 'three', 'sentences.']
```

Figure 4.2.32

a list of 13 strings, each one word from the original string. It removes the spaces themselves as well.

We can also specify our own unique separator as an argument to `split()`, as shown in Figure 4.2.33. Here, instead of splitting the string up into 13 strings based on spaces, it splits it into 4 strings based on periods. Note, though, a couple issues. First, the spaces at the beginning of each sentence are still there. Second, there's an empty string at the end: Python sees the period at the end and splits between two strings, regardless of the fact that nothing comes after that period.

```
# Split-2.py
myString = "This is my text. It has thirteen " \
           "words. It also has three sentences."
print(myString.split("."))
```

```
Output
['This is my text', ' It has thirteen words', ' It also has three
sentences', '']
```

Figure 4.2.33

We can resolve this by instead splitting on the entire "." string, as shown in Figure 4.2.34. This still isn't perfect; this means that Python removes the period character from the first two sentences (since it's part of the "." string being used to split), but not the third. Still, we're closer now to what we wanted.

```
# Split-3.py
myString = "This is my text. It has thirteen " \
           "words. It also has three sentences."
print(myString.split(". "))
```

```
Output
['This is my text', 'It has thirteen words', 'It also has three
sentences.']
```

Figure 4.2.34

This `split()` method is especially useful when we deal with comma-, tab-, or newline-separated lists. It's common to ask users to enter multiple options separated by commas. For example, imagine we were asking the users to enter the first names of each person they want to e-mail as part of an e-mail application. We could have them enter the names one at a time until they type "exit" or something similar, or we could have them enter the names all at once separated by commas. Then, we can use the `split()` method with "," as the argument to pull out the individual names, as shown in Figure 4.2.35.

Note that we don't necessarily know if the user will put spaces after commas or not. We don't want to split based on ", " (with a space) because then it will not split

```
# Split-4.py
1 names = input("Enter a list of names: ")
2 print(names.split(","))

Output
Enter a list of names: David,Vrushali,Lucy,Ping,Jasmine
['David', 'Vrushali', 'Lucy', 'Ping', 'Jasmine']
```

Figure 4.2.35

if they don't, but we don't want to include the spaces either if they do. The easiest way to do this will be to strip out the whitespace at the beginning and end of each string after calling `split()`, which we'll cover next.

Useful String Methods

The Python string class has a lot of utility methods for modifying our strings in predictable and useful ways. Here are a few of them:

- `capitalize()`. Makes the first letter of the string uppercase and all the rest lowercase, and returns the result.
- `lower()`. Returns a version of the string with all uppercase letters changed to lowercase.
- `upper()`. Returns a version of the string with all lowercase letters changed to uppercase.
- `title()`. Returns a version of the string with each word (e.g., letter following a space) capitalized.
- `strip()`. Returns a version of the string with any whitespaces (spaces, line breaks, etc.) at the beginning and end of the string removed. `rstrip()` and `lstrip()` apply this `strip()` method only to the right or left sides of the string.
- `replace(old, new)`. Replace all occurrences of the substring old with the substring new.
- `rfind(findString)`. Just like `find()`, but returns the *last* index of `findString` instead of the first.
- `join(list)`. Creates a string where each item in the list is followed by the string, and returns the result.

Figure 4.2.36 shows a demonstration of those methods in action. Notice that `myString` is the same on line 13 as on line 2; these methods return the result of the change, but they leave the original string unaffected. So, to store the results of these method calls, we would need to assign the result to a new string, e.g., `newString = myString.lower()`.

```
# UsefulStringMethods-1.py
1  myString = "this is MY test string!   "
2  print(myString)
3  print(myString.capitalize())
4  print(myString.lower())
5  print(myString.upper())
6  print(myString.title())
7  print(myString.strip())
8  print(myString.replace("MY", "YOUR"))
9
10 myList = myString.split()
11 print("-".join(myList))
12
13 print(myString)
14
```

```
Output
this is MY test string!
This is my test string!
this is my test string!
THIS IS MY TEST STRING!
This Is My Test String!
this is MY test string!
this is YOUR test string!
this-is-MY-test-string!
this is MY test string!
```

Figure 4.2.36

Among these, likely the most interesting and most confusing is the `join()` method. Note that on line 11, we're calling the join method not on `myString`, but on the string "-". `myString` is an argument to `join()` here, not the string calling `join()`. The `join()` method takes a list of items and merges them into one string, joined by the character on which you're calling the `join()` method. In that way, the `join()` method is like the reverse of the `split()` method, as shown in Figure 4.2.37.

#	UsefulStringMethods-2.py	Output
1	`myStringToSplit = "My-string-to-split"`	`My-string-to-split`
2	`print(myStringToSplit)`	`['My', 'string', 'to', 'split']`
3		`My-string-to-split`
4	`#Splits myStringToSplit and assigns it`	
5	`#to mySplitString`	
6	`mySplitString = myStringToSplit.split("-")`	
7	`print(mySplitString)`	
8		
9	`#Joins mySplitString using "-" and assigns`	
10	`#it to myJoinedString`	
11	`myJoinedString = "-".join(mySplitString)`	
12	`print(myJoinedString)`	
13		

Figure 4.2.37

In Figure 4.2.37, we initially split `myStringToSplit` into its individual parts based on where the hyphens are located. We print it and see a list of each individual string that had previously been separated by hyphens. Then, we call `join(mySplitString)` on the simple string "-", and Python merges the items on the `mySplitString`, joining them with the "-".

Additionally, Python has some interesting boolean methods that simply check if certain criteria are `true` about the string, returning `True` if so and `False` if not:

- `endswith(suffix)`. Returns `True` if the string ends with suffix, `False` if not.
- `startswith(prefix)`. Returns `True` if the string starts with prefix, `False` if not.
- `isalnum()`. Returns `True` if the string is all letters and numbers, `False` if not.
- `isalpha()`. Returns `True` if the string is all letters, `False` if not.
- `isdecimal()`. Returns `True` if the string represents an integer or decimal number, `False` if not.
- `isdigit()`. Returns `true` if the string is all numbers (e.g., represents an integer), `False` if not. `isnumeric()` is similar, but supports fraction and other characters as well (which are rarely used).
- `islower()`. Returns `True` if the string contains no uppercase letters, `False` if not.
- `isupper()`. Returns `True` if the string contains no lowercase letters, `False` if not.
- `istitle()`. Returns `True` if the string is in title case, meaning each word is capitalized, and `False` if not.

`turtle.write(message, [move], [align], [font])`
A method of the turtle library that will write the given message on the canvas. If move is `True`, it will move the turtle along with the text. Align determines whether the text is left, right, or center aligned, and font is a three-tuple that contains the font face, size, and style.

6. Turtles and Text

In this chapter we've been talking all about strings, but it would seem that wouldn't have a lot to do with our turtles program. Sure, we accept user input in text, but those are pretty simple commands that just get fed into our conditional. Can we really do anything more complicated than that?

That would be true, except for the `turtle.write()` method. The `turtle.write()` method takes as input a string (and, optionally, an alignment, a font, and

a setting whether to move to the end of the written text), and in turn writes the given message to the screen. Now, suddenly, anything we've been doing with our strings can be written by our turtles! So, let's create a simple function for this, one that will just write the user's message at the current point. We'll find, though, that it isn't quite that simple if we want the user to put in multiple lines of text.

The Text Function

After revising our code to allow a new "text entry" option, we get TheTextFunction.py. Run it, enter the "text" command, and type "Hello, world" to see this option in action.

First, let's do a little cleanup. The print statement where we ask the user to enter a command is getting pretty long. So, we've switched it to print the options on one line (line 37) and get the input on the next (line 38). After that, we add a new `elif` to our conditional on line 70: "text". If the user enters "text", they're prompted to enter the message that they'd like to print on line 72. After entering the message, it's printed on the canvas on line 74.

This is a single method call, so we don't really gain anything by wrapping it in a function. For now, we've forced certain options as well: we've set the font size to 16, the font face to Arial, and the style to normal to keep things simple.

Penup and Feedback

Running TheTextFunction.py, though, we quickly run into an issue: while it's good for the turtle to move to the end of the text it drew (so we can write multiple messages in a row), that draws a line under the text, which could get a little ugly. So, in PenupAndFeedback.py, we've added two new commands here on lines 78 through 88: **penup and pendown**. These simply turn drawing off and on. If the turtle moves with the pen up, it won't draw a line, so now our users can move the turtle around the canvas without necessarily drawing the entire trail!

`turtle.penup()` and `turtle.pendown()`
Two methods of the turtle library that toggle off and on, respectively, whether the turtle draws lines as it moves.

Notice, however, that the penup and pendown commands wouldn't have any automatic feedback to the user; how do they know they've worked? Other commands draw lines or show rotation, but these cause no immediate visible change. So, to help the user know the command registered, we have added print statements here to confirm the action was executed. This is the valuable principle of feedback at work: the user should be able to immediately tell that their input was received and correctly recognized.

The Text Function and Newlines

Now let's take a quick closer look at this. We mentioned earlier the escape sequence `\n`. Could the user enter "\n" to write text on multiple lines? Try it out and see: enter a string into the text command with "\n" in the middle.

As you'll see, no! If the user enters "\n" in their text, Python just prints the characters "\" and "n". Why is that? Why doesn't Python interpret this as the escape sequence? Let's find out. Let's print the user's input after the text entry line to see how Python is interpreting it. We won't include a file for this here, but try printing `message` after line 72. What do you see?

When we print the message directly, we see something similar: Python prints "\n". So, the string is properly included. But wait: earlier when we included "\n" in a string, Python translated it as a newline… unless the slash was preceded by another slash! When we included "\\n" in our string, Python printed "\n", but when we included just "\n", it printed a newline. So, if it's printing the "\n" here, it must be storing it as "\\n"! Python must be automatically converting the user's "\n" into "\\n" so that it will print the string as the user inputted it.

So, if we want the user to use "\n" in their input to separate lines, we need to replace any instances of "\\n" with "\n". So, before printing the message, we add a simple call to the `replace()` method on line 74, and the result is shown in TheTextFunctionandNewlines.py.

By replacing "\\n" with "\n", we achieve the result we wanted, and the user is able to enter text on two lines. We should add a note for the user about that as well: that keeps the option discoverable. Interestingly, to include that note in our prompt, we have to use the double-slash again: "use \\n to designate a newline" will render only one slash.

CHAPTER

4.3

Lists

1. What Are Lists?

List-like structures are data structures that themselves store multiple values. For example, think of a file folder (a real one, in a file cabinet, not one on your computer). In one sense, you could think of that folder as a singular object. In another sense, you could think of it as a container of several other objects, each individual page inside the folder. If someone were to ask you for the folder, you could give them the folder; yet, the folder on its own is not data, it just contains other data.

That's effectively what a list is: a single variable that contains multiple values. Actually, this definition is even a little too broad: this would encompass other data structures that we'll talk about later. More specifically, lists are variables that contained ordered lists of values, accessed via numbers called indices (plural for index).

Properties of Lists

We're starting to get into the more advanced areas of computing, and so as we go forward, fewer and fewer things will be common across multiple languages. Lists are a good example of this. Nearly every programming language has some concept of a list, but the terminology and specific details differ significantly.

One major way different languages differ in their implementations of lists is mutability. Mutability for a list can involve two things: one, whether the values of the list can be changed, and two, whether the size of the list can be changed. Some list implementations will freely let you keep appending new items to the end, while others require you to state in advance how many items can fit in the list. Generally, the latter is more common for lower-level languages like C, where we manage memory more deliberately.

Another major way list-like structures may differ is whether they accept multiple types. This is called **homogeneity** in lists. In some list implementations, every item in a list must be of the same type: you can have a list of integers, a list of strings, or a list of dates, but you can't have a single list that has integers, strings, dates, and other types. Other languages don't have this restriction.

There are benefits to these different properties, so many languages will provide multiple ways to use list-like structures. What, then, defines a list? A list is a data structure that contains multiple values, accessed via an ordered numerical index; in other words, lists will have a first value, a seventh value, and so on. This is in contrast to Dictionaries, HashTables, and other data structures that also contain multiple values, but that access them via non-numeric keys.

List Synonyms

List-like structures go by various names: lists, arrays, tuples, vectors, tables, and more. The specific terms used depend on the language. Java, for example, uses "array" and "list" to refer to different things. "Tuple" is most commonly used in Python. Oftentimes, "array" is used to refer to a more primitive structure that only supports changing the existing values, whereas "list" refers to more complex data structures that support sorting, inserting, and other operations. "Tuples" are often immutable.

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Recall the concept of list data structure, its properties, and advanced list-like structures, including stacks, queues, and linked lists;
- Create tuples in their programs, including nested tuples and perform read and write operations on them;
- Write programs to implement lists, perform iterations, and to learn the difference between lists and tuples;
- Use the tuples and lists to pass information around a turtle program and record information about it.

List-like Structures
Also referred to as sequences and collections, a data structure that holds multiple individual values gathered together under one variable name, accessed via indices. Includes to lists, arrays, and tuples. Lists are simultaneously a type of data structure and a specific type in some languages.

Homogeneity
A property of lists determining whether they can accept multiple types of variables. A homogenous list can only accept one type of variable; a non-homogenous or heterogenous list can accept multiple types.

Homogenous Lists

5	"These"	5.1
17	"Are"	7.2
23	"All"	9.0
1	"One"	1.1
53	"Word"	4.3
5	"Strings"	5.6

Heterogenous Lists

5	"All"	4.3
"These"	9.0	53
5.1	23	"Word
17	"One"	5.6
"Are"	1.1	5
7.2	1	"Strings"

Figure 4.3.1

Tuple
An immutable form of a list-like structure in Python.

Lists
A mutable form of a list-like structure in Python.

As noted above, which specific implementation you'll use depends on your language. The commonality among these list-like structures is that they contain an ordered series of values accessed via numeric indices.

2. Tuples in Python

Python has two major list-like structures: **tuples** and **lists.** Tuples are immutable, lists are mutable. The immutable type tends to be simpler because it can't be modified after it's declared, although for the same reason it tends to be less useful. So, let's start with tuples, then move on to more advanced lists.

Declaring Tuples

To declare a tuple, we set a variable equal to a comma-separated series of values or variables, as shown on line 2 of Figure 4.3.2. Here, we're creating a tuple (called a 3-tuple because it has three values) with the values 1, 2, and 3. We've done this here in terms of values, but if we use variables, it still pulls out the values and creates a tuple with these values, as shown on lines 7 and 8 in Figure 4.3.3.

#	DeclaringTuples-1.py	Output
1	`#Creates myTuple with the tuple (1, 2, 3)`	`(1, 2, 3)`
2	`myTuple = (1, 2, 3)`	
3		
4	`print(myTuple)`	
5		

Figure 4.3.2

#	DeclaringTuples-2.py	Output
1	`myInt1 = 1`	`(1, 2, 3)`
2	`myInt2 = 2`	
3	`myInt3 = 3`	
4		
5	`#Creates myTuple and assigns it the tuple`	
6	`#(myInt1, myInt2, myInt3)`	
7	`myTuple = (myInt1, myInt2, myInt3)`	
8	`print(myTuple)`	
9		

Figure 4.3.3

Note that the parentheses are optional in both Figure 4.3.2 and Figure 4.3.3; the same result would occur from `myTuple = 1, 2, 3` and `myTuple = myInt1, myInt2, myInt3`. The parentheses are always optional except for in those places where they would make a difference, such as differentiating passing a tuple to a function from passing three arguments. However, I recommend including the parentheses. They never hurt, and they sometimes help.

Tuples can have multiple data types within them. For example, we could create a tuple of a string, a float, and an integer, as shown in Figure 4.3.4. Here, the first value of the tuple is a string, the second is a float, and the third is an integer. This follows Python's general procedure of being loosely typed: Python never really cares what data types you're using until you try to do something that doesn't make sense. Imagine, though, if you assumed every value in the tuple was a string and attempted to call `isupper()` on each to check if they were uppercase. That would crash on the second item in the tuple because it isn't a string, so we need to be careful to know what data types we're dealing with.

#	DeclaringTuples-3.py	Output
1	`myString = "Hello, world!"`	`('Hello, world!',`
2	`myFloat = 5.1`	`5.1, 5)`
3	`myInteger = 5`	
4		
5	`#Creates myTuple as a tuple with the values`	
6	`#of myString, myFloat, myInteger`	
7	`myTuple = (myString, myFloat, myInteger)`	
8		
9	`print(myTuple)`	
10		

Figure 4.3.4

Reading Tuples

To access the individual items of a tuple, we use the same syntax we used for accessing individual characters in a string. As far as Python is concerned, strings, tuples, and lists are all pretty much the same.

#	ReadingTuples-1.py	Output
1	`myString = "Hello, world!"`	`Hello, world!`
2	`myFloat = 5.1`	`5.1`
3	`myInteger = 5`	`5`
4		
5	`#Creates myTuple as a tuple with the values`	
6	`#of myString, myFloat, myInteger`	
7	`myTuple = (myString, myFloat, myInteger)`	
8		
9	`#Prints the first element of myTuple`	
10	`print(myTuple[0])`	
11	`#Prints the second element of myTuple`	
12	`print(myTuple[1])`	
13	`#Prints the third element of myTuple`	
14	`print(myTuple[2])`	
15		

Figure 4.3.5

As shown in lines 10, 12, and 14 of Figure 4.3.5, to read an individual value from a tuple, we simply place the index in brackets after the variable name to access that specific element. This analogy to strings extends to slicing as well. We can access individual parts of the tuple using the same syntax we used with strings. Figure 4.3.6 shows a long tuple with some examples of slicing it the way we sliced strings on lines 11 through 19. All the same splicing syntax we used for strings works here for tuples as well.

#	ReadingTuples-2.py	Output
1	`myString = "Hello, world!"`	`('Q', -1)`
2	`myFloat = 5.1`	`('Hello, world!', 5.1)`
3	`myInt1 = 5`	`(5.1, 5)`
4	`myCharacter = "Q"`	`(5, 'Q', -1)`
5	`myInt2 = -1`	`('Hello, world!', 5.1)`
6		
7	`myTuple = (myString, myFloat, myInt1, myCharacter,`	
8	`myInt2)`	
9		
10	`#Prints myTuple's values from #4 to the end`	
11	`print(myTuple[3:])`	
12	`#Prints the first two values of myTuple`	
13	`print(myTuple[:2])`	
14	`#Prints the second and third values of myTuple`	
15	`print(myTuple[1:3])`	
16	`#Prints the last three values of myTuple`	
17	`print(myTuple[-3:])`	
18	`#Prints all but the last three values of myTuple`	
19	`print(myTuple[:-3])`	
20		

Figure 4.3.6

Tuples also have a handy syntax for unpacking them, shown in line 11 of Figure 4.3.7. While technically this doesn't allow us to do anything we couldn't do before (since we could have referred to the tuple parts by index whenever we needed them), it allows us to easily unpack the parts of a tuple into variables with self-documenting names again rather than remembering what kind of data is held at each index.

```
# ReadingTuples-3.py
1  myString = "Hello, world!"
2  myFloat = 5.1
3  myInt1 = 5
4  myCharacter = "Q"
5  myInt2 = -1
6
7  #Packs these five variables into myTuple
8  myTuple = (myString, myFloat, myInt1, myCharacter, myInt2)
9
10 #Unpacks myTuple into these variables
11 (myNewString, myNewFloat, myNewInt1, myNewCharacter,  myNewInt2) = myTuple
12
13 print(myNewString)
14 print(myNewFloat)
15 print(myNewInt1)
16 print(myNewCharacter)
17 print(myNewInt2)
```

```
Output
Hello, world!
5.1
5
Q
-1
```

Figure 4.3.7

Usefulness of Tuples

Given that tuples are immutable, they lack some of the value of traditional lists. We would not use tuples to store a list of students on a class roster, for example, because students might enroll or drop, and a tuple could not add or remove those students. Similarly, we would not use tuples to store data that we want to sort because the element order cannot be changed.

In that case, what are tuples good for? You might be tempted to say, "A tuple lets me use fewer variable names!", but that's not inherently a good thing. Remember, we want our variables to be self-documenting, and having to remember which index corresponds to which value isn't self-documenting.

Rather, the real value of tuples comes in places where we can only pass one variable back and forth, but we want to pass multiple values. The most prominent example of that is the return statements of functions. A function or method can only return one

```
# UsefulnessofTuples-1.py
1  #Returns a tuple containing the quotient and remainder
2  def quotientAndRemainder(dividend, divisor):
3      #Gets the quotient
4      quotient = dividend // divisor
5      #Gets the remainder
6      remainder = dividend % divisor
7      #Returns the tuple of the quotient and remainder
8      return (quotient, remainder)
9
10 myDividend = 11
11 myDivisor = 4
12 tupleResults = quotientAndRemainder(myDividend, myDivisor)
13
14 #Prints the first element of tupleResults
15 print("Quotient:", tupleResults[0])
16 #Prints the second element of tupleResults
17 print("Remainder:", tupleResults[1])
18
```

```
Output
Quotient: 2
Remainder: 3
```

Figure 4.3.8

value, but if it returns a tuple, it can actually return multiple values contained within that list. The receiving code merely needs to know what is located at each index of the tuple.

Let's try an example of this: imagine we want to tie together floor division and modulus so that we can call one function and get both the quotient and remainder for a division operation.

In lines 2 through 8 of Figure 4.3.8, we define a function `quotientAndRemainder` that takes as parameters a dividend and a divisor. It calculates the quotient and remainder, then returns a tuple containing the quotient in the first spot and the remainder in the second on line 8. The main code of the program on lines 15 and 17 knows (because we told it) that it can find the quotient in the first spot and the remainder in the second, so it prints them. Thus, we've returned two values with one function by packaging them as a tuple.

We can actually make this code shorter and more readable respectively by performing the operations in the tuple assignment and by using the unpacking trick we saw earlier, as shown in Figure 4.3.9. Here, we skip creating new temporary variables to hold the quotient and remainder by calculating them on line 4, and we make our code more readable by unpacking the resultant tuple into `myQuotient` and `myRemainder` on line 10 instead of referring to these values as `tupleResults[0]` and `tupleResults[1]`. This gives tuples what appear to be some considerable power. However, we'll later see that Dictionaries are actually even better for this, at least in some ways.

UsefulnessofTuples-2.py

```
1  #Returns a tuple containing the quotient and remainder
2  def quotientAndRemainder(dividend, divisor):
3      #Returns the tuple of the quotient and remainder
4      return (dividend // divisor, dividend % divisor)
5
6  myDividend = 11
7  myDivisor = 4
8  #Assigns the first value of the result to myQuotient and
9  #the second to myRemainder
10 (myQuotient, myRemainder) = quotientAndRemainder(myDividend, myDivisor)
11
12 print("Quotient:", myQuotient)
13 print("Remainder:", myRemainder)
14
```

Output

```
Quotient: 2
Remainder: 3
```

Figure 4.3.9

Nesting Tuples

Finally, tuples can be nested. You can have a tuple of tuples. First, we could do this by creating tuples, and then creating a tuple that contains those tuples, as shown in lines 1 through 5 in Figure 4.3.10.

NestingTuples-1.py

```
1  myTuple1 = (1, 2, 3)
2  myTuple2 = (4, 5, 6)
3  myTuple3 = (7, 8, 9)
4
5  mySuperTuple = (myTuple1, myTuple2, myTuple3)
6
7  print(mySuperTuple)
8
```

Output

```
((1, 2, 3), (4, 5, 6),
(7, 8, 9))
```

Figure 4.3.10

The nested sets of parentheses in the output of Figure 4.3.10 show that these are nested tuples: the outer set of parentheses defines the tuple as a whole, and the inner sets of parentheses define each smaller tuple. You might notice that it looks like this syntax could be used in-line as well, and you would be correct, as shown in Figure 4.3.11.

#	NestingTuples-2.py	Output
1	`mySuperTuple = ((1, 2, 3), (4, 5, 6), (7, 8, 9))`	`((1, 2, 3), (4, 5, 6), (7, 8, 9))`
2		
3	`print(mySuperTuple)`	
4		

Figure 4.3.11

How would you then access individual values of this nested tuple? The first index you provide would determine which of the three tuples you grab, and the second index you provide would determine which element from that tuple you get. So, to get the first item of the second tuple, you ask for `mySuperTuple[1][0]`), as shown in Figure 4.3.12.

#	NestingTuples-3.py	Output
1	`myTuple1 = (1, 2, 3)`	`4`
2	`myTuple2 = (4, 5, 6)`	
3	`myTuple3 = (7, 8, 9)`	
4		
5	`mySuperTuple = (myTuple1, myTuple2, myTuple3)`	
6		
7	`#Prints the first item of the second tuple`	
8	`print(mySuperTuple[1][0])`	
9		

Figure 4.3.12

Remember, tuples are like strings and all other list-like structures in Python: they're zero-indexed. So, to get the first item, you request the item at index 0. This is why requesting `mySuperTuple[1][0]` yields 4: `mySuperTuple[1]` yields the second tuple, and `mySuperTuple[1][0]` yields the first item of the second tuple.

3. Lists in Python

For the most part, anything you can do with a tuple, you can also do with a list. Most of the syntax is the same, even. Lists are created similarly, sliced the same way, accessed the same way, and nested the same way. The major difference is that lists are mutable, which means we can add items to them or remove items from them. So, let's start by quickly demonstrating the extent to which lists are the same as tuples, then find out what makes them different.

#	ListsasTuples-1.py	Output
1	`myList1 = [1, 2, 3]`	`[1, 2, 3]`
2	`print(myList1)`	
3	`print()`	
4	`myInt1 = 1`	`[1, 2, 3]`
5	`myInt2 = 2`	
6	`myInt3 = 3`	`['Hello, world!', 5.1,`
7	`myList2 = [myInt1, myInt2, myInt3]`	`5, 'Q', -1]`
8	`print(myList2)`	
9	`print()`	
10	`myString = "Hello, world!"`	`Hello, world!`
11	`myFloat = 5.1`	`5.1`
12	`myInt1 = 5`	`5`
13	`myCharacter = "Q"`	
14	`myInt2 = -1`	`['Q', -1]`
15	`myList3 = [myString, myFloat, myInt1, myCharacter, myInt2]`	`['Hello, world!', 5.1]`
16	`print(myList3)`	`[5.1, 5]`
17	`print()`	`[5, 'Q', -1]`
18	`print(myList3[0])`	`['Hello, world!', 5.1]`
19	`print(myList3[1])`	
20	`print(myList3[2])`	
21	`print()`	
22	`print(myList3[3:])`	
23	`print(myList3[:2])`	
24	`print(myList3[1:3])`	
25	`print(myList3[-3:])`	
26	`print(myList3[:-3])`	

Figure 4.3.13

Lists as Tuples

Nearly everything we did with tuples, we can also do with lists. Figure 4.3.13 runs through all the different creation, unpacing, and slicing methods we covered with tuples, showing they work with lists as well.

Notice that the output of each block throughout this process matches the output of the corresponding block from the tuple examples in the previous lesson. The only difference is that lists are defined with brackets (as shown on line 7 of Figure 4.3.13) instead of parentheses.

```
# ListsasTuples-2.py
1  #Returns a tuple containing the quotient and remainder
2  def quotientAndRemainder(dividend, divisor):
3      #Returns the tuple of the quotient and remainder
4      return [dividend // divisor, dividend % divisor]
5
6  myDividend = 11
7  myDivisor = 4
8  #Assigns the first value of the result to myQuotient and the
9  #second to myRemainder
10 [myQuotient, myRemainder] = quotientAndRemainder(myDividend, myDivisor)
11
12 print("Quotient:", myQuotient)
13 print("Remainder:", myRemainder)
14
Output
Quotient: 2
Remainder: 3
```

Figure 4.3.14

The handy way of unpacking tuples into individual variables even works with lists, as shown in Figure 4.3.14. And nested lists work the same way as nested tuples, as shown in Figure 4.3.15. You could even mix lists and tuples to have a list of tuples or a tuple of lists, or a list with both lists and tuples or a tuple with both tuples and lists. Python is very flexible.

```
# ListsasTuples-3.py
1  mySuperTuple = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
2
3  print(mySuperTuple)
4
Output
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Figure 4.3.15

So, everything you learned about tuples still applies to lists. The declarations are the same (just with brackets instead of parentheses), accessing elements is the same, slicing is the same, unpacking is the same, nesting is the same. What's different, then?

List Functions

What differentiates lists is that they're mutable. That means there are a lot of interesting things we can do with lists that we haven't been able to do with strings or tuples. Let's run through a few of them:

- `append(item)`. Adds `item` to the end of the list, lengthening the list by one.
- `insert(index, item)`. Inserts `item` at the given `index`. For example, `insert(1, newItem)` would make `newItem` the second item on the list (because lists are zero-indexed).
- `sort()`. Sorts the items in the list in place. This will change the values associated with each index in the list.

- `reverse()`. Reverses the current order of the list. This will also change the values associated with each index in the list.
- `copy()`. Returns a copy of the list.
- `index(item)`. Returns the index of the first element in the list whose value matches `item`.
- `count(item)`. Returns the count of elements in the list whose values match `item`.
- `remove(item)`. Removes the first element in the list whose value matches `item`.
- `pop()`. Removes and returns the last item on the list.
- `clear()`. Removes all items from the list.
- `extend(incomingList)`. Appends all the items in `incomingList` to the current list. Note that this adds the items in `incomingList` one-by-one to the current list; it doesn't add `incomingList` itself as an item to the current list. `incomingList` could be a list or a tuple.

In addition, there are a couple of other ways we can examine and modify lists. The `in` operator can still be used to check to see if a particular value is present in a list; it will return `True` if the value is present, `False` if the value isn't. We can also use the reserved word `del` to delete items from the list by their index. We can combine `del` with our syntax for slicing lists to delete any slice that we want to remove.

That's a lot of methods we just ran through, so Figure 4.3.16 runs through them all one-by-one. Remember, because lists are mutable, if a method modifies the list, that modification will carry forward. That differs from our string methods because strings were immutable.

To try to make this easier to follow, we're going to do something a little unconventional: we're going to put the string labels on the right so that you can line up the lists and see the differences one on top of the other. So, walk through it one by one. Read the operation on the top to see what was done, then compare the lists to see what happened. Then, revisit the method name to see why this happened. If you're confused, try running them yourself!

ListFunctions.py

```
#A list of the numbers 1 through 20
myList = [9, 2, 4, 3, 1, 0, 7, 8, 6, 5]
print(myList, ": Original list")
myList.sort()
print(myList, ": After sorting")
myList.append(11)
print(myList, ": After appending 21")
myOtherList = [15, 13, 14]
myList.extend(myOtherList)
print(myList, ": After extending with myOtherList")
myList.insert(7, 12)
print(myList, ": After inserting 25 at the index 15")
myList.remove(15)
print(myList, ": After removing 26")
myList.sort()
print(myList, ": After sorting again")
myList.reverse()
print(myList, ": After reversing")
myList.pop()
print(myList, ": After popping")
del myList[-5:]
print(myList, ": After deleting the last five items")
print(myList.index(7), ": Index of 7")
print(myList.count(5), ": Count of 5")
print(6 in myList, ": 6 in myList")
print(25 in myList, ": 25 in myList")
```

Output

```
[9, 2, 4, 3, 1, 0, 7, 8, 6, 5] : Original list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] : After sorting
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11] : After appending 11
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 13, 14] : After extending with myOtherList
[0, 1, 2, 3, 4, 5, 6, 12, 7, 8, 9, 11, 15, 13, 14] : After inserting 12 at the index 7
[0, 1, 2, 3, 4, 5, 6, 12, 7, 8, 9, 11, 13, 14] : After removing 15
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14] : After sorting again
[14, 13, 12, 11, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] : After reversing
[14, 13, 12, 11, 9, 8, 7, 6, 5, 4, 3, 2, 1] : After popping
[14, 13, 12, 11, 9, 8, 7, 6] : After deleting the last five items
6 : Index of 7
0 : Count of 5
True : 6 in myList
False : 25 in myList
```

Figure 4.3.16

4. Lists, Loops, and Functions

A significant portion of the value of lists (and sometimes tuples as well) comes when we integrate them with loops. Additionally, lists are our first example of mutable data structures, so it's useful to remind ourselves what happens when we pass a mutable data structure into a function. So, let's briefly look over three things: iterating over a list with a `for` loop, iterating over multi-dimensional lists, and using lists with functions.

Iterating Over a List

Remember, to iterate over a sequence means to execute some block of code for each item in the sequence. So, iterating over a list means executing a segment of code for each individual item in a list. The same can be done for a tuple as long as we're not trying to modify it.

Recall that earlier when we talked about loops, we had a program where the user would enter a bunch of grades, and it would average those grades. Let's revise that program to instead average the numbers in a list. In fact, let's make it a function, so that it could be used in other ways by a hypothetical program.

`IteratingOveraList.py`

```
1  #Averages the numbers in a list
2  def average(inList):
3      sum = 0
4      for number in inList:
5          sum += number
6      return sum / len(inList)
7
8  myList1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
9  myList2 = [97, 87, 91, 83, 85, 91, 95, 99, 81, 85]
10
11 print("The average of myList1 is:", average(myList1))
12 print("The average of myList2 is:", average(myList2))
13
14
```

Output

```
The average of myList1 is: 5.0
The average of myList2 is: 89.4
```

Figure 4.3.17

In Figure 4.3.17, we define the function `average()` on lines 2 through 6, which takes as input a list. It initializes `sum` to 0, and then for each `number` in the list, it adds `number` to `sum`. It then returns `sum` divided by the number of items in the list. In the main program starting on line 8, we create two lists: `myList1` and `myList2`. We then print their averages on lines 11 and 12.

We can do this kind of iteration for any kind of list. Each time the loop iterates, `number` (or whatever variable fills in the blank in `for ___ in list)` takes on the value of the next item in the list. The first time the loop runs on `myList1`, `number` has the value 1, and 1 is added to `sum`. The second time, it has the value 2, and 2 is added to `sum`; it repeats this *for each* item in the list.

Generally, when dealing with lists, we use `for` loops. We can technically use `while` loops, but `for` loops just tend to be a little bit easier: we don't have to worry about manually checking the length of the list or incrementing a loop control variable, and if we're using a `for-each` loop, we need no loop control variable anyway. The constraint on `for` loops was that we needed to know the number of iterations in advance, but we do know that with a list, as long as we don't make the mistake of modifying its length while the loop is running.

Multi-Dimensional Lists
A list-like structure where the items in a list are themselves lists, such that the practical effect is a multi-dimensional list.

Iterating Over a 2-Dimensional List

Let's take that a step further. We've mentioned that we can have **multi-dimensional lists,** or lists of lists. How do we iterate over those? Let's imagine that instead of just averaging one set of numbers, we were interested in averaging several sets, each within themselves. Let's call this a gradebook. In the gradebook, we have a list of lists of grades: each list of grades corresponds to one student, so we want to average each individual student's grades. The result of this should be a separate list where each item in the list is the average of the corresponding item from the list of lists of grades. In other words, item #1 in the result should be the average of the numbers in list #1.

#	IteratingOvera2DimensionalList.py	Output
1	`#Averages the numbers in a list`	`Averages: [90.4, 85.2,`
2	`#Averages each list in in2DList`	`82.2, 88.4, 92.8]`
3	`def TwoDAverage(in2DList):`	
4	`    result = []`	
5	`    #For each list in the list of lists`	
6	`    for numList in in2DList:`	
7	`        sum = 0`	
8	`        #For each number in the current list`	
9	`        for number in numList:`	
10	`            sum += number`	
11	`        #Append this list's average to result`	
12	`        result.append(sum / len(numList))`	
13	`    return result`	
14		
15	`my2DList = [[91, 95, 89, 92, 85],`	
16	`            [85, 87, 91, 81, 82],`	
17	`            [79, 75, 85, 83, 89],`	
18	`            [81, 89, 91, 91, 90],`	
19	`            [99, 91, 95, 89, 90]]`	
20		
21	`print("Averages:", TwoDAverage(my2DList))`	

Figure 4.3.18

Let's start with the main program this time so we can have some real numbers when we discuss the function. In Figure 4.3.18, we define `my2DList` on lines 15 through 19 as a two-dimensional list: a list of lists. We start the outer list with a bracket, and then immediately start the first inner list with another bracket. Python lets us split the list definition into multiple lines to keep things organized, so we can see the two-dimensional structure here: each line is a list, and `my2DList` is a list where each of these lists is one of the items.

Then, we print the result of `TwoDAverage(my2DList)`. So, the code jumps up to `TwoDAverage()` on line 4. It initially defines an empty list on line 4 that will hold the results; each time we get a new result, we'll add it to this list. Then, we start what we call the outer loop on line 6: the outer loop is the loop that wraps around the other (inner) loop. Each run of our outer loop will grab the next list from `my2DList` and assign it to `numList`. So, on the first iteration, the value of `numList` is [91, 95, 89, 92, 85], which is the first list in `my2DList`.

Now, we run the inner loop over `numList`. This portion is identical to our previous `average()` function: create `sum` and set it equal to 0 on line 7, iterate over each number adding it to `sum` on lines 9 and 10, then divide `sum` by the length of the list to get the average on line 12. Instead of returning it, though, we add it to `result`: `result` contains all the averages we want to return, so we want to wait until the end to return it. So, the outer loop runs for each of the five lists, and the inner loop runs within each of the five lists. In the end, `result` contains five numbers, each the average of one of the lists in `my2DList`.

This kind of nested loop structure is how we iterate over a two-dimensional list. We could take this further: we could have a list of lists of lists, where we would need three nested loops to iterate over everything.

Lists and Functions

Lists are mutable. That means that if we pass a list to a function or method, the function or method *can* change the values of the list, and those changes will persist out to the main program. Sometimes, this can be problematic. Imagine, for example, if the function from Figure 4.3.18 "popped" the grades in each list rather than iterating over them, as shown in Figure 4.3.19.

ListsandFunctions-1.py

```
1  #Averages each list in in2DList
2  def TwoDAverageWithPop(in2DList):
3      result = []
4      #Repeat until in2DList is empty
5      while len(in2DList) > 0:
6          #Remove and assign the last item of in2DList to numList
7          numList = in2DList.pop()
8          sum = 0
9          count = 0
10         #Repeat until numList is empty
11         while len(numList) > 0:
12             #Remove and save the last item of numList to number
13             number = numList.pop()
14             sum += number
15             count += 1
16         #Insert this average at the beginning of result
17         result.insert(0, sum / count)
18     return result
19
20 my2DList = [[91, 95, 89, 92, 85],[85, 87, 91, 81, 82],
21             [79, 75, 85, 83, 89],[81, 89, 91, 91, 90],
22             [99, 91, 95, 89, 90]]
23
24 print("Averages:", TwoDAverageWithPop(my2DList))
25 print("my2DList:", my2DList)
```

Output

```
Averages: [90.4, 85.2, 82.2, 88.4, 92.8]
my2DList: []
```

Figure 4.3.19

The `pop()` method *removes* and returns the last item in a list. It's good when you want to empty and process everything in a list. If it was used here, though, it's actually changing the list itself. That's why we have to manually count the number of items on line 9 and 15: when we reach line 17 to calculate the sum, the length of the list is now empty.

So, when dealing with lists and any other mutable data type, we have to be careful that we understand that all modifications we make to the list will persist. Most of the time, though, this will be useful: we can call functions to modify lists in place. We've actually seen an example of this. Compare the string methods to the list methods in Figure 4.3.20.

ListsandFunctions-2.py

```
1  myString = "Hello, world"
2  myList = [4, 1, 2, 3]
3
4  print("myString before upper():", myString)
5  print("myList before sort():", myList, "\n")
6
7  #Returns an uppercase version of myString
8  myString.upper()
9  #Sorts myList in place
10 myList.sort()
11
12 print("myString after upper():", myString)
13 print("myList after sort():", myList)
14
```

Output

```
myString before upper(): Hello, world
myList before sort(): [4, 1, 2, 3]

myString after upper(): Hello, world
myList after sort(): [1, 2, 3, 4]
```

Figure 4.3.20

String's `upper()` method on line 8 makes the string uppercase. List's `sort()` method on line 10 sorts a list. Yet, after we call `myString.upper()` and `myList.sort()`, only `myList` has changed, as shown by the `print()` statements on lines 12 and 13. Lists such as `myList` are mutable, meaning that a method can actually change its values (or in this case, the order of its values). Strings such as `myString` are immutable, meaning that a method cannot change its value. To get the uppercase version of `myString`, we have to assign `myString.upper()` to a variable; it returns what happens *if* we converted the string. To get the sorted version of `myList`, we just have to call `myList.sort()`; it sorts it *in place*. To go back to our example of Addison, `myString.upper()` is like asking someone to read to you the results of the capitalizing the string; `myList.sort()` is like handing someone the list and asking them to hand you back a sorted list.

Lists vs. Tuples

Based on our description, lists can do everything tuples can do and more. They can be declared, sliced, nested, and unpacked the same ways, with the added bonus of being mutable. That means we can add new items, remove items, sort items, and more.

So, you might wonder: Why would you ever use tuples? That's actually an interesting question, and if you ask a bunch of people, you'll get a bunch of different answers. Generally, though, there are a couple of conventions we follow to decide whether to use lists or tuples.

First, we often use tuples when we're definitely dealing with a predetermined number of items. We use lists when the ability to add or remove from the list would actually be useful and relevant. Consider our `quotientAndRemainder()` function from Figure 4.3.14, though. Would there ever be a need to add additional values to the return from that function? Not really; they represent two qualitatively different things, a quotient and a remainder. So, we use a tuple in that case.

Common Conventions for Lists and Tuples:

- Use tuples when the number of items is known in advance.
- Use lists when the number of items may change.
- Use tuples for heterogenous collections.
- Use lists for homogenous collections.

That connects nicely to the second convention. When we think of a list of things, we typically think of the list as somewhat homogenous. That is, we think of the things on the list as being qualitatively similar. Take our examples of real-world lists. A grocery list is a list of items. A to-do list is a list of tasks. We would generally consider it strange to put "Call Becky" on a grocery list or "oranges" on a to-do list because we think of lists as having a homogenous collection of things. So, if we're grouping together different *kinds* of information (even if they're the same data *types*), we often use tuples.

That convention can be made even stronger. One common convention is that lists are used for *like* data types. Python technically lets you create a list that has integers, floats, strings, and other data types in the same list, but by convention we rarely do that. This is largely because lists are so often iterated upon with `for` loops: when we iterate over a list, we execute a block of code on each item in the list. For a single block of code more complex than just a print statement to work on every item, the items generally need to be of the same type. Tuples are more often used to represent information that is unpacked into unique variables, like quotients and remainders, so it's more natural for them to have a different data types.

These are all just conventions, though. That means that while other programmers with whom you share your code will often expect these conventions to be followed, Python isn't going to stop you from doing things one way or the other. It's good to get in the habit of following these conventions, though.

5. Advanced List-Like Structures

Before we move on, let's quickly cover three special kinds of list-like structures. These structures are like lists in that they contain multiple items in a certain order. However, they provide unique constraints on how the items in the list are actually accessed. In some ways, these are additional data types; many languages have

dedicated types for these list-like structures. In other ways, these are simply ways of interacting with lists, and are constraints we could apply on ourselves depending on the type of program we're writing.

Stacks

A **Stack** is a list-like structure that limits the ways in which you can add or remove information from the list. Rather than being able to insert or grab information from anywhere in the list, you can only add new information on top, and you can only access the information that is currently on the top. To add new data, we "push" it onto the top of the stack; to remove data, we "pop" it off the top of the stack. This is sometimes referred to as Last-In-First-Out, or LIFO. The last item added to the list is the first item removed from the list.

Stack
A list-like structure that follows the "Last-In-First-Out" paradigm, where we can only access the most recently-added item on the list and can only access it by removing it from the list.

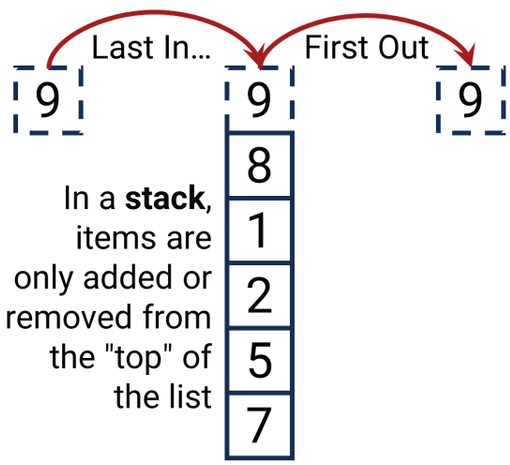


Figure 4.3.21

Stacks are LIFO:
Last-In-First-Out

Figure 4.3.22

This makes a lot of sense when we're dealing with physical objects: typically putting a new item on top blocks access to the ones below, so that should be the one we grab first when we need one. What about when there is no physical need to do so, though?

Imagine you were programming a robot to search for your keys in your house or apartment. You would want it to search each room, and within each room search each piece of furniture, and within each piece of furniture search each drawer. So, what would you do? You'd initially give it three commands: `search-Kitchen`, `search-Bedroom`, and `search-Bathroom`. `search-Kitchen`, though, would be unpacked to find `search-Counters`, `search-Drawers`, and `search-Cabinets`. You would want the robot to do all three of these things before moving on to search the bedroom. So, you'd push those three commands on top of the stack of orders, then pop them off one by one. That way, you would ensure the robot would not move on to the bedroom until it had finished checking the kitchen.

This is actually analogous to an advanced computing topic called depth-first search that comes from the advanced data structure trees. These are both outside the scope of this material, but you'll get to them in a future computing class.

Queues

Queues work in exactly the opposite way of stacks: rather than Last-In-First-Out, queues are First-In-First-Out, or FIFO. Queues work the way most lines you experience in the real-world work: you are served in the order in which you got in line.

Queue
A list-like structure that follows the "First-In-First-Out" paradigm, where we can only access the least recently-added item on the list and can only access it by removing it from the list.

Your call is answered in the order it was received. With queues, we're restricted to removing items from the front of the list and adding items to the end of the list. We refer to this as "enqueuing" (adding an item to the end of the list) and "dequeuing" (removing an item from the start of the list).

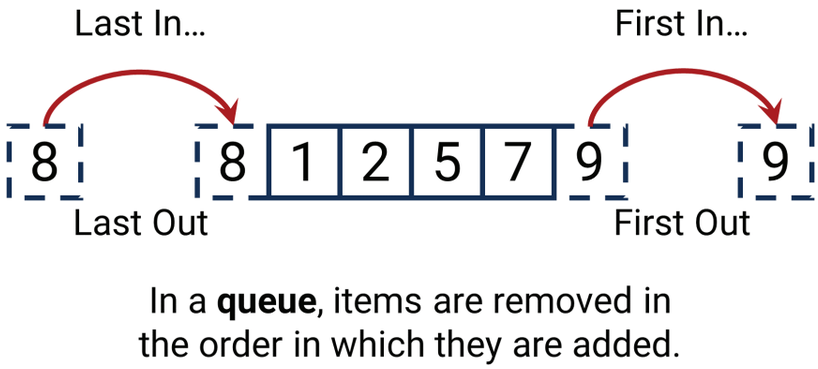


Figure 4.3.23

Queues are FIFO:
First-In-First-Out

When designing programs, we might use queues to process tasks in the order they were launched. Interestingly, however, stacks end up being used more often in programming, for both theoretical and practical reasons. Theoretically, stacks lend themselves to the way in which programs are executed, as demonstrated by the analogy to a depth-first search above—you don't need to understand the analogy right now, but rather just know that there exist concepts in computing that lend themselves nicely to stacks. For practical reasons, stacks tend to be a little more efficient as well: when using a queue, every time we dequeue an item, we have to update the indices of every other item in the list to decrease them by one. When using advanced computers that isn't a big issue, but when you use computers of earlier generations, which are comparatively slower, that could present a major difficulty.

Note that when we iterate over a list using a simple `for` loop, we're processing things in the same order we would process them with a queue. That doesn't make that a queue, though: queues and stacks are characterized by the requirement to *remove* an item from the list in order to really access it.

Linked List

Linked List:
A list-like structure where the location of each item in the list is contained in the previous item in the list.

Linked lists are a special kind of implementation of a list-like structure. In order to understand linked lists, though, we have to briefly talk about how lists are usually stored.

In lower-level languages (i.e., languages that are closer to the real functions of a computer), memory is typically allocated in contiguous blocks. That's why lists in these languages have to be declared with their lengths in advance: If you need a list with 40 memory spots, the computer has to find an area of memory with 40 consecutive spots open. If you then need a 41st, there's no guarantee the 41st spot is open.

Higher-level languages abstract over this process. While they're written to mimic interacting with a contiguous block of memory, in reality there might be pointers to different areas of memory. A list with five items might store those five items in various different places in memory, and when you request the second item, it goes and looks up the next item's memory location. With older computers, though, even those look-ups could take some notable time, especially if you were doing a lot of them. Inserting was a particularly high-intensity operation: if you wanted to insert an item in the middle of a list, the computer had to go through and change its locations for every item after it in the list.

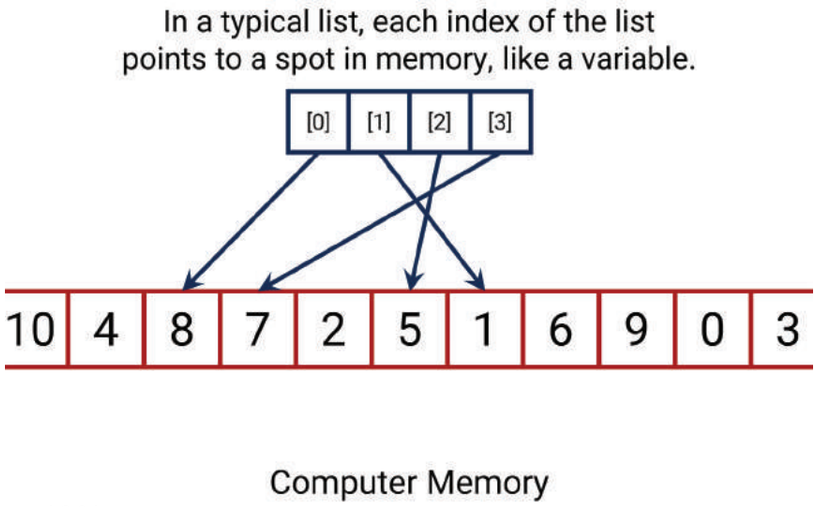


Figure 4.3.24

A linked list operates differently. Instead of one list of memory pointers, each item in the list would contain both its value *and* a pointer to the next item in the list. So, if you wanted to get item #6, you would find its location from item #5. This makes certain operations significantly more efficient. Iterating, for example, wouldn't require the computer to return to its list of memory locations each time it wanted to move to the next item: the next item's location was stored with the current item. Inserting was a breeze as well: instead of having to update every item's index, only the previous item had to be updated. If we wanted to insert a new item #6, we would just have to change item #5 to point to the new item #6; the new item #6 would then point to the old item #6, which is now #7.

Of course, other operations are significantly less efficient. In a linked list, the computer has to iterate through *every* previous item to find an item with a given index; there's no way to jump straight to item #7 like there is with a regular list. But if we're iterating and inserting far more than we're jumping into the center of the list, then a linked list could grant some significant efficiency gains. Many of these issues have gone away as computers have gotten faster, but the underlying concepts are still part of the core of computing theory.

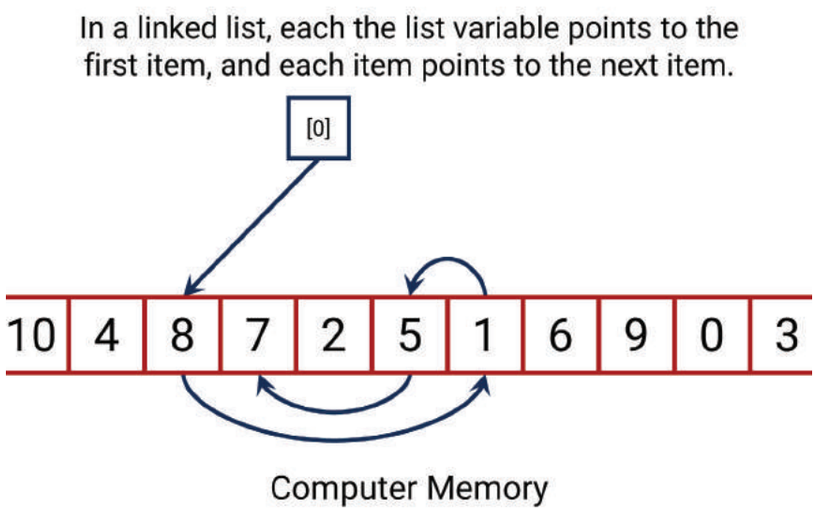


Figure 4.3.25

6. Lists and Turtles

We have a new tool in our toolbox now: lists. How can we use lists to improve or expand our user interface for controlling turtles? Let's do two things: first, let's simplify the process by which we're listing commands that the user has available. Second, let's create a new function: a record function that would allow the user to enter multiple commands, and a number of times to repeat them.

It's worth noting that here, the complexity of our turtles applications is going to rise tremendously. Don't worry at all if you get lost; we're covering very advanced structures, syntax, and ideas. Try to step through the code one line at a time, following how the programs execute. Try to predict what will happen if you enter certain commands, and then follow through and see if you were right. Most importantly, don't get discouraged. It's taken me a long time to develop this example, and part of its purpose *is* to show the complexity possible with the concepts we've learned. If you understand the rest of the course besides these turtle examples, you're still doing very well.

Listing Commands

When we covered strings, we covered a `join()` method that could use the string to join together multiple items in a list into a longer string. Right now, we're having to update the line that lists the commands for users manually each time we add a new command. This can be slightly easier if we maintain a list of all the valid commands at the top of the program, and simply join together the valid commands when they're needed. This way, if we need to list the commands in multiple places, we don't need to worry about adding to the list in multiple places.

We've only made two changes in ListingCommands.py. First, we've created a tuple early in the program: `VALID_COMMANDS`, on line 3. This is written in all caps to specify it as a constant, a variable whose value won't be changed. Granted, a tuple's value can't be changed anyway, but we're also not going to set `VALID_COMMANDS` equal to a different tuple either. The goal of writing it in all caps is to communicate that this variable will retain the value it was assigned at the beginning throughout the program.

Then, later, we've replaced the line that lists the available commands with line 39, which calls ", ".`join(VALID_COMMANDS)`. This takes the individual strings in `VALID_COMMANDS` and combines them into one string, with ", " separating each pair. This way, when we add a new command, we need only remember to add it to the tuple at the beginning, not to any line that lists commands. So, let's go ahead and change the "Invalid!" message to use `VALID_COMMANDS`, too, on line 96.

Preparing for the Record Function

Our next goal is to create a command that will allow the user to enter *multiple* commands, as well as a number of times to repeat these commands, to then be executed repeatedly. In other words, instead of just entering one command at a time, we want the user to enter a *list* of commands, followed by a number of times to repeat these commands.

That's going to mean, however, our user needs to be able to enter commands in two ways: one, the normal way we have now, to execute them directly, and two, a new way, to enter them into a "recording". This is going to mean a pretty significant restructuring of our program. We now need to differentiate commands that are to be run immediately from commands that are to be added to a list. We also need to differentiate commands that are run based on direct user input from commands that are run based on executing a list. In other words, we need to split up our command input and our command execution. To do that, we need to create a data structure that will hold commands and their parameters. We'll use tuples for that in PreparingfortheRecordFunction.py; remember, tuples are good for heterogenous lists.

We haven't yet added our record function yet, right now we're just laying the groundwork. To be able to record commands to be repeated later, we need to get commands from the user without executing them right away. We also need to execute commands that weren't just entered by the user. So, we've split up the process: We have separate functions for getting commands from users and to execute commands.

Previously, however, the arguments we received from users were stored in local variables, like angle and distance. Now, however, we need to store commands and their arguments separately. This means combining qualitatively different kinds of information into one variable whose value should not change once entered. In other words, perfect time for a tuple!

So, we've revised our code: we now have a `getCommandFromUser()` function starting on line 16. It runs the same conditionals we ran previously, but instead of immediately executing the command the user enters, it instead stores it in a tuple with the command name itself as the first item. That tuple is returned at the end of the function, so we'll be able to correctly direct it to be executed or recorded. Right now, we're only executing, so immediately after getting a command from the user, we run the execute function. It runs the same conditional based on what command was in the first spot of the tuple, but instead of getting any user input, it just executes the command directly.

Note one special feature here: for our commands without arguments (penup, pendown, end), we have a comma after the command name in the tuple declaration. That's how Python knows to treat these as tuples: otherwise it just treats them as strings. Notice also how this has radically shortened the actual control loop of the function, down on lines 86 through 89. It could be shortened even more by having the `while` loop operate on `commandTuple[0]` directly, but we're about to need the command in the main function anyway.

So, we've now separated the act of getting the command from the user and executing the command we received. Now we're ready to start differentiating whether the command should be executed immediately or recorded for repetition.

The Record Function

Now let's make things complicated. All our new reasoning here is going to be inside our `while` loop. Why? The list of recorded moves (the list `recordList`) and the memory of whether we're recording (the boolean `recording`) need to persist across multiple executions of the while loop. There are other ways we could handle this, of course, but for now let's do it the way shown in TheRecordFunction.py.

Every step of the `while` loop's execution, it now does a couple extra things; all of the following reasoning takes place in the `while` loop at the bottom of the program. It first checks if it's currently recording. If it's not, it executes the next command. If that command is to *start* recording, then it clears the list of recorded commands (in case this isn't the first time we've recorded something), and sets the boolean recording to `True`. So, if we're not recording and we didn't just start recording, `executeCommand()` runs like normal.

If we *are* recording, then the loop first checks to see if the command was to stop. If so, it stops recording, then runs two loops. The outer loop repeats the request number of repetitions. The inner loop runs each command one-by-one. So, if the user asked to run the sequence five times, the outer loop would run five times, and each time it ran, all the commands would run one-by-one.

If we're recording and the command *isn't* stop, then it just adds the command to the list and continues as normal. Once we're done recording, the program returns to its default state: any new commands are executed immediately unless we record again.

Our program has now gotten very complex. This little explanation shouldn't be enough to understand it. Instead, you should try it out. Try different sequences of commands to see what happens. Modify the code and see what breaks, and how. This is a very complex program, one that took someone with almost two decades of programming experience considerable time to write and debug; it's okay to find it intimidating. Step through it line-by-line, see why it operates the way it does, and try it out with different input.

How else could we have implemented this? If we wanted to avoid having to check for the recording and stop commands in the while loop, we could have repeatedly passed the list `recordList` and the boolean recording back and forth between the main code and the functions themselves. I prefer not to do that, however, since that requires dealing with them every step of the way rather than only dealing with them when recording is turned on and off. We could also force the functions to see global copies of the variables, but that's outside of the scope (no pun intended) of what we've covered so far.

As of now, for anyone, the recording function requires some pretty complex reasoning within our while loop. In the future, we may find a way to streamline this.

CHAPTER

4.4

File Input and Output

1. What Is File Input and Output?

So far, the major weakness of everything we've written is that every time we run our code, it's like we're running it for the very first time. Nothing persists, or is saved, across different runs of our code. If we want to change some data, we have to change it in the code itself. Needless to say, this isn't how real programs operate. Nearly every program we use on a daily basis persists some information across multiple runs of the same program, whether it's login information, the user preferences, or the files that we create. This is taken care of by the complementary processes of file input and file output.

Output Complements Input

We usually refer to "**file input and output**," but let's start with output because it's what we'll need to do first in the programs we've been writing. File output is the process of taking what's currently stored in memory and writing it to a persistent file on the hard drive. We might not want to write everything in memory to a file, but anything we would want the next time we run the program would need to be written. This is file output: outputting the current data in the program to a file.

The complementary process to this is file input. File input is reading that data from the file into our program's active memory. Ideally, these processes are reverses of one another: whatever data was outputted to the file should end up loaded into the program in the same way when inputting from the file. Imagine a program that had three variables, `a`, `b`, and `c`, with the values 5, 3, and 1, respectively. When outputting to the file, the program would write 5, 3, and 1. When inputting the file, the goal would be for 5 to be loaded as the value of `a`, 3 the value of `b`, and 1 the value of `c`. Ideally, it would not be clear after loading that any outputting and inputting occurred at all: loading should restore the state of the program to just as it was when it saved (for most programs, anyway).

Some of the data we're referring to is obvious. It might be the preferences the user has set, or the document that they've created; when they load it again, it should load the document exactly as it was saved. Other data can be a little subtler, though. Recent versions of Microsoft Word, for example, save not only the document, but also the scroll position, so reopening the document actually shows you the same area of the document as you saw when you saved.

File Types

As you've noticed using computers in your everyday life, most files have types. There are .pdf files, .docx files, .jpg files, .png files, and thousands of other types. Each type of file specifies rules about how the program should read the data within it.

You may have noticed that you can open any type of file with a plain text reader, like Notepad on Windows, TextEdit on Mac, Emacs or Vim on Linux, or many more. While technically you can open these files, you usually won't be able to read much in them: they're not stored in just plain text, but rather there's a lot of

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Recall the fundamentals of file input and output, including the concepts of reading, writing, and appending files;
- Create programs that write to and read from files;
- Write programs to save commands to a file and load them from the file for execution.

File Input and Output
The complementary processes of saving data to a file and loading data from a file, generally such that the state of the memory of the program is the same after saving and loading have occurred.

built-in, type-specific encoding. That encoding is properly unpacked by a program that knows how to read the file.

Think of encoding as the organizational scheme for a friend or co-worker. If you walk into a co-worker's office, you might see file folders labeled with different colors and numbers. You don't know what those colors and numbers mean, so you can't fully understand the information stored there the way they can. That's what a file type is: a set of rules for how to interpret a file. A program can only correctly interpret the file if it knows these rules. You could learn the rules, and similarly, we could develop new programs to read existing file types, but only if the rules are made available.

There do exist plaintext file formats, like .txt, .csv, .html, .xml, and more. These are like walking into your co-worker's office and seeing an organizational scheme so simple or so well-documented, anyone could understand it. When you open these, what you see is what you get: you can read them just as the text itself. We'll stick close to these plaintext types for our conversations.

2. Reading, Writing, and Appending

With plaintext files, there are three general concepts we need to understand: reading from the files, writing to the files, and appending to the files. Regardless, though, it all starts with opening files and ends with closing files.

Getting Started: Opening and Closing Files

File input can vary pretty dramatically from language to language. Generally, however, it follows a certain high-level workflow. First, the file is opened and assigned to a variable that represents the opened file. This is not always an easy step: if we try to open a file that doesn't exist, our program will usually throw an error, and as a result, crash if the error was not handled. For that reason, many languages require that file input and output be enclosed within a `try` block. In many languages, when opening a file, we need to specify a mode: read-only, write, or append, which we'll cover in the next section.

Opening the file, in most languages and operating systems, locks the file down in the operating system. Other programs are not permitted to modify the file while our program has it opened. For that reason, we need to also close the file when we're done. Closing the file indicates to the operating system that we're done modifying it, and it can be modified by other programs again.

Reading, Writing, Appending

Once we've opened a file, there are three general modes we'll usually use for interacting with it: reading, writing, and appending. Reading simply means that we're looking at the file's contents and reading it into our program. We're not changing the file's contents at all, just reading it.

Writing, on the other hand, means we're writing to the file from scratch. Have you ever accidentally saved over a file on your computer with a different file? The reason that's so disastrous is that when "writing" to a file, we by default erase it and write it completely from scratch. With writing, we assume the file is a snapshot of the current state of our program, not a running log of the history of its changes. Usually that's the case, but it does mean we need to be careful. We should never use sensitive data when testing out our input and output because we could easily find ourselves overwriting it.

The third mode, appending, is safer, although often not as useful. Appending also writes to the file, but it starts on the last line of the file. Nothing is overwritten; new data is just added to the end.

Beyond writing or appending, there are some pretty advanced ways of writing content to files. For example, serialization is an often automatic process

of grabbing all the variables in a program and automatically storing them in a way that can be loaded directly in the future. That's further down the road, though. For our purposes, we're going to discuss file input and output primarily in terms of reading and writing lines one-by-one. This is a good place to get started because it keeps things pretty close to the individual variables we're storing.

3. Writing Files in Python

Let's start by writing files. The main reason to start here is that to have something to load, we must have something saved! We'll go through two examples: a simple one, where we just output a handful of variables, and a complex one, where we output a list using a loop.

Simple File Writing

To write to a file in Python, we open the file, write our data, then close the file. Python makes this relatively easy, as shown in Figure 4.4.1.

#	SimpleFileWriting-1.py	OutputFile.txt
1	`myInt1 = 12`	122334
2	`myInt2 = 23`	
3	`myInt3 = 34`	
4		
5	`#Open OutputFile.txt in write mode`	
6	`outputFile = open("OutputFile.txt", "w")`	
7		
8	`#Write myInt1 to outputFile`	
9	`outputFile.write(str(myInt1))`	
10	`#Write myInt2 to outputFile`	
11	`outputFile.write(str(myInt2))`	
12	`#Write myInt3 to outputFile`	
13	`outputFile.write(str(myInt3))`	
14	`#Close outputFile`	
15	`outputFile.close()`	
16		

Figure 4.4.1

We open the file using Python's general function `open()` on line 6. We give it a filename and, optionally, a `mode`: "w" for write, "r" for read, and "a" for append. We then assign the open file to a variable. This variable now contains our open file. However, the variable itself isn't really data: what the variable does is it allows us to run methods that will take care of reading and writing the file.

Right now we want to write, so we call `outputFile.write()` on lines 9 through 13. `outputFile.write()` can only write strings, so we have to convert our variables to strings before writing them, so our argument to `outputFile.write()` on line 9 is `str(myInt1)`. Then, after writing all the variables, we `close()` the file on line 15. After running this, we could go to the folder where this code lives, and we would find OutputFile.txt. Opening it, we would find that it contains the text shown on the right side of Figure 4.4.1: 122334.

We probably want each number to appear on its own line, though. Otherwise, we would receive the same file from printing number sets like 12, 23, 34; 123, 3, 34; 12, 334, 2; and so on. Remember our escape sequence `\n`? If we add that to the end of each line, we'll accomplish our goal, as shown in Figure 4.4.2. Here, if we open the file, we'll see the numbers 12, 23, and 34 each on a separate line. That would make it easier to load in our loading stage.

`open()`
A function that takes as input a filename and, optionally, a write mode ("r" for read, "w" for write, "a" for append), and opens the file for access.

`write(text)`
A method of a variable with type file, writes the text to the file.

`close()`
A method of a variable with type file, closes the file from further reading or writing.

#	SimpleFileWriting-2.py	OutputFile.txt
1	`myInt1 = 12`	12
2	`myInt2 = 23`	23
3	`myInt3 = 34`	34
4		
5	`#Open OutputFile.txt in write mode`	
6	`outputFile = open("OutputFile.txt", "w")`	
7		
8	`#Write myInt1 to outputFile`	
9	`outputFile.write(str(myInt1) + "\n")`	
10	`#Write myInt2 to outputFile`	
11	`outputFile.write(str(myInt2) + "\n")`	
12	`#Write myInt3 to outputFile`	
13	`outputFile.write(str(myInt3) + "\n")`	
14	`#Close outputFile`	
15	`outputFile.close()`	
16		

Figure 4.4.2

Writing Lists

That form of simple writing works just fine if we know exactly how many values we want to store and exactly the order in which to store them. As we'll see later, when loading from that file, we'll assume the first line holds `myInt1`, the second holds `myInt2`, and the third holds `myInt3`.

Most of the interesting applications we'll write, though, don't have a predictable number of variables. What if we want to write a list of items to a file? Let's pretend we're writing a list of names, for example. How would we do that? The most obvious way would be to iterate over the list, writing them to the file one by one, as shown in Figure 4.4.3. The `for` loop on line 8 repeats for each name in `myList`, calling `outputFile.write(name + "\n")` on each one to print the name and a line break.

#	WritingLists-1.py	OutputFile.txt
1	`myList = ["David", "Lucy", "Vrushali", "Ping",`	David
2	`"Natalie", "Dana", "Addison", "Jasmine"]`	Lucy
3		Vrushali
4	`#Open OutputFile.txt in write mode`	Ping
5	`outputFile = open("OutputFile.txt", "w")`	Natalie
6		Dana
7	`#For each name in myList`	Addison
8	`for name in myList:`	Jasmine
9	`    #Write the name to the file on its own line`	
10	`    outputFile.write(name + "\n")`	
11		
12	`outputFile.close()`	
13		

Figure 4.4.3

This could be even easier, though. The reason we cover lists before file output (or at least, one of the reasons) is because Python has a handy way for writing lists to a file, as shown on line 8 of Figure 4.4.4.

The `writelines()` method writes every item in the list to a file. Unfortunately, we're back to our old problem: `writelines()` doesn't append line breaks, so all the names are squished together! So what do we do? Well, we could put the newlines directly into the names, but that seems a little inelegant: what if we need to use these names for something else? Instead, we could merge the list into one string with the newlines built in, and then write *that* to a file, as shown in line 8 of Figure 4.4.5.

```
# WritingLists-2.py
myList = ["David", "Lucy", "Vrushali", "Ping",
          "Natalie", "Dana", "Addison", "Jasmine"]

#Open OutputFile.txt in write mode
outputFile = open("OutputFile.txt", "w")

#Writes every string in myList to a file
outputFile.writelines(myList)

outputFile.close()
```

```
OutputFile.txt
DavidLucyVrushaliPingNa
talieDanaAddisonJasmine
```

Figure 4.4.4

```
# WritingLists-3.py
myList = ["David", "Lucy", "Vrushali", "Ping",
          "Natalie", "Dana", "Addison", "Jasmine"]

#Open OutputFile.txt in write mode
outputFile = open("OutputFile.txt", "w")

#Joins myList using \n, then writes it to a file
outputFile.write("\n".join(myList))

outputFile.close()
```

```
OutputFile.txt
David
Lucy
Vrushali
Ping
Natalie
Dana
Addison
Jasmine
```

Figure 4.4.5

This code embeds the newline escape sequence, "\n", between every pair of names in `myList`; if you're unsure how this works, peek back at the `join()` method from Chapter 4.2. When it prints this one large string, the newlines are built in, as shown by the contents of OutputFile.txt in Figure 4.4.5.

Note that we can alternate between writing variables directly and writing via loops, as well as print via multiple loops or nested loops. Writing is just like printing: whenever the computer encounters a `write()` call, it writes to the file in that order.

Figure 4.4.6 shows an example of printing variables directly followed by printing a list. This code will write 12, 23, and 34 to the first three lines of the output file, then the strings David through Jasmine on the next eight lines.

```
# WritingLists-4.py
myInt1 = 12
myInt2 = 23
myInt3 = 34
myList = ["David", "Lucy", "Vrushali", "Ping",
          "Natalie", "Dana", "Addison", "Jasmine"]

#Open OutputFile.txt in write mode
outputFile = open("OutputFile.txt", "w")

#Write myInt1 to outputFile
outputFile.write(str(myInt1) + "\n")
#Write myInt2 to outputFile
outputFile.write(str(myInt2) + "\n")
#Write myInt3 to outputFile
outputFile.write(str(myInt3) + "\n")
#Joins myList using \n, then writes it to a file
outputFile.write("\n".join(myList))

outputFile.close()
```

```
OutputFile.txt
12
23
34
David
Lucy
Vrushali
Ping
Natalie
Dana
Addison
Jasmine
```

Figure 4.4.6

Another Way to Output

Python also gives another way to output files that might be a little more intuitive. The `print()` function that we've been using for a long time has a keyword parameter `file` that, when defined, writes to the specified file instead of the console, as shown in Figure 4.4.7.

AnotherWaytoOutput.py

```
1  myList = ["David", "Lucy", "Vrushali", "Ping",
2            "Natalie", "Dana", "Addison", "Jasmine"]
3
4  #Open OutputFile.txt in write mode
5  outputFile = open("OutputFile.txt", "w")
6
7  #For each name in myList
8  for name in myList:
9      #Write the name to the file on its own line
10     print(name, file = outputFile)
11
12 outputFile.close()
13
```

OutputFile.txt

```
David
Lucy
Vrushali
Ping
Natalie
Dana
Addison
Jasmine
```

Figure 4.4.7

We still open the file the same way on line 5, but instead of writing using `outputFile.write()`, we write using `print()` on line 10, and specify with the `file` parameter that the target of our print statement is `outputFile`. The added bonus here is that `print()`, by default, appends the newline character. It also lets us use our other keyword parameters like `end` and `sep`, so we could change the newline character to something else, or write multiple variables separated by a space at the same time.

Personally, I usually see people using the first approach (the `write()` method), but I prefer using the second approach (including the `file` parameter with the `print()` function). Either will work, though.

Appending to Files

In all of the above examples, we used the "w" argument to indicate we were writing to the file. "w" tells the computer to open the file and rewrite it from the beginning; the original file contents are overwritten. If we instead use the "a" argument, the computer opens the file in "append" mode. It's ready to write, but starting at the end of the file instead of the beginning, keeping the original contents.

What happens if we run the previous code blocks with append mode instead of write mode? Let's assume we're starting with OutputFile.txt empty, and that we run our code from Figure 4.4.2 twice. If we were in write mode, the second run would merely overwrite the first. What about in append mode?

AppendingtoFiles-1.py

```
1  myInt1 = 12
2  myInt2 = 23
3  myInt3 = 34
4
5  #Open OutputFile.txt in append mode
6  outputFile = open("OutputFile.txt", "a")
7
8  #Write myInt1 to outputFile
9  outputFile.write(str(myInt1) + "\n")
10 #Write myInt2 to outputFile
11 outputFile.write(str(myInt2) + "\n")
12 #Write myInt3 to outputFile
13 outputFile.write(str(myInt3) + "\n")
14 #Close outputFile
15 outputFile.close()
16
```

OutputFile.txt (after 2x)

```
12
23
34
12
23
34
```

Figure 4.4.8

Figure 4.4.8 shows the results of running this code twice in append mode. If we run this code twice, then each of the three lines is printed twice. If we run it three more times, then each of the three lines will be printed three more times.

Does this work for the alternate method of writing we covered? Let's again assume that OutputFile.txt is empty when we get started, and that we run the code twice, as shown in Figure 4.4.9. This works, too! So, if we want to add to the end of an existing file instead of writing it from scratch, we can open it in append mode. In practice, I haven't found that many users for this: it's most useful for logging across multiple runs of a program.

#	AppendingtoFiles-2.py	OutputFile.txt (after 2x)
1	`myList = ["David", "Lucy", "Vrushali", "Ping",`	David
2	`"Natalie", "Dana", "Addison", "Jasmine"]`	Lucy
3		Vrushali
4	`#Open OutputFile.txt in append mode`	Ping
5	`outputFile = open("OutputFile.txt", "a")`	Natalie
6		Dana
7	`#For each name in myList`	Addison
8	`for name in myList:`	Jasmine
9	`    #Write the name to the file on its own line`	David
10	`    print(name, file = outputFile)`	Lucy
11		Vrushali
12	`outputFile.close()`	Ping
13		Natalie
14		Dana
15		Addison
16		Jasmine
17		
18		
19		
20		
21		

Figure 4.4.9

4. Reading Files in Python

We've now written some data to a file. Our goal is now to complete the symmetry between output and input: we want to load these files back into our program such that the values of the variables are the same after loading as they were before.

Simple File Reading

In our first example in Figure 4.4.2, we wrote three integers to a file, each on their own line. Now, let's write a program to load these back into `myInt1`, `myInt2`, and `myInt3`. First, though, let's just see how Python opens the file and what it sees when it does in Figure 4.4.10.

#	SimpleFileReading-1.py	Output
1	`inputFile = open("OutputFile.txt", "r")`	`<_io.TextIOWrapper`
2	`print(inputFile)`	`name='OutputFile.txt' mode='r'`
3	`inputFile.close()`	`encoding='cp1252'>`
4		

Figure 4.4.10

Here's how we open a file to read: we use the same function, `open()`, and supply the "r" argument to indicate we want to read it. What happens if we just print the file directly? Python prints a reference to what type of file is and what mode it's in, not the contents. To print the contents, we have to actually read the file, as shown in 4.4.11.

#	SimpleFileReading-2.py	Output
1	`inputFile = open("OutputFile.txt", "r")`	12
2		
3	`#Print the next line of inputFile`	23
4	`print(inputFile.readline())`	
5	`#Print the next line of inputFile`	
6	`print(inputFile.readline())`	34
7	`#Print the next line of inputFile`	
8	`print(inputFile.readline())`	
9		
10	`inputFile.close()`	
11		

Figure 4.4.11

`readline()`
A method of a variable of type file, reads and returns the next line of the file as a string.

To read files, we use the `readline()` method in the `outputFile` variable, as shown on lines 4, 6, and 8. `readline()` reads to the next line break, then returns the string that results. This moves the reader forward to the start of the next line, so when we call `readline()`, the next line is passed permanently; if we don't save the results in a variable, we can't go back and read it again without completely reopening the file from scratch. Reading the file reads through it once straight through; it doesn't jump around or repeat. Note also that the line breaks are included in what is read from the file, as indicated by the extra blank line between numbers in the output. If we want to store or print the lines without the line breaks (as we often might), we need to strip the whitespace off of them, as shown in lines 4, 6, and 8 of Figure 4.4.12.

#	SimpleFileReading-3.py	Output
1	`inputFile = open("OutputFile.txt", "r")`	12
2		23
3	`#Print the next line of inputFile`	34
4	`print(inputFile.readline().strip())`	
5	`#Print the next line of inputFile`	
6	`print(inputFile.readline().strip())`	
7	`#Print the next line of inputFile`	
8	`print(inputFile.readline().strip())`	
9		
10	`inputFile.close()`	
11		

Figure 4.4.12

This is deceptively complicated actually. We're calling a method on another method. We have `inputFile`, a variable that represents the file we're reading. We call `inputFile.readline()` to get the next line of `inputFile`. We then want to strip the whitespace (spaces and newlines) off of the string; but if `readline()` returns a string, then we can call `strip()` directly on top of `readline()`. The result is that we print the lines one-by-one with the extra newline characters removed.

If that's still confusing, let's step through it bit by bit. We tell the computer to print `inputFile.readline().strip()`. When we chain calls together like this, they're run left-to-right—after all, it doesn't know what it's calling `strip()` on until it calls `readline()`. So, it grabs `inputFile()` and calls `readline()`. The first time this happens, `readline()` returns "12\n". This effectively replaces `inputFile.readline()` with "12\n". So, this line effectively becomes `"\n".strip()`. "12\n" is a string, so it has access to the `strip()` method. `strip()` removes spaces and newlines, so it removes "\n", leaving only "12".

#	SimpleFileReading-4.py	Output
1	`inputFile = open("OutputFile.txt", "r")`	myInt1: 12
2		myInt2: 23
3	`#Read the next line of inputFile, cast it`	myInt3: 34
4	`#to int, and assign it to myInt1`	
5	`myInt1 = int(inputFile.readline())`	
6	`#Read the next line of inputFile, cast it`	
7	`#to int, and assign it to myInt2`	
8	`myInt2 = int(inputFile.readline())`	
9	`#Read the next line of inputFile, cast it`	
10	`#to int, and assign it to myInt3`	
11	`myInt3 = int(inputFile.readline())`	
12		
13	`print("myInt1:", myInt1)`	
14	`print("myInt2:", myInt2)`	
15	`print("myInt3:", myInt3)`	
16		
17	`inputFile.close()`	
18		

Figure 4.4.13

So, if we want to store these lines in our variables as we load them, we have to assign the result of each call to `readline()` to some variable, as shown in lines 5, 8, and 11 of Figure 4.4.13.

Each time we call `inputFile.readline()`, it reads the next number from the file. By default, `readline()` returns strings, so we have to convert them to integers. The `int()` function is smart enough to ignore the whitespace in the strings, so we can skip the `strip()` method. Then, we print them out one-by-one on lines 13 through 15.

However, the most important thing is that at the end of this execution, `myInt1`, `myint2`, and `myInt3` hold the same values that they held before we saved and closed the program from Figure 4.4.2: 1, 2, and 3 respectively. We've thus completed the symmetry between output and input. Their values before saving and after loading are the same.

Loading into Lists

Writing lists was relatively easy: we just iterated over the list, writing each line to a file just the same way we would print it. The reason this was easy was that we knew in advance how many items in the list there were. However, that presents a challenge for reading from a list. When we're reading, we don't know in advance how many lines there are to read. How do we get around this?

Well, we could go back and change our output, telling it to first print the length of the list. That's a little inelegant, though. Instead, it would be great if Python had a mechanism for reading all the lines in a file until the end. Fortunately, it does have a couple ways of doing this, as shown in Figure 4.4.14.

#	LoadingintoLists-1.py	Output
1	`myList = []`	['12', '23', '34']
2		
3	`#Open OutputFile.txt in read mode`	
4	`inputFile = open("OutputFile.txt", "r")`	
5		
6	`#For each line in the file`	
7	`for line in inputFile:`	
8	`    #Add the line to myList, stripping out whitespace`	
9	`    myList.append(line.strip())`	
10		
11	`print(myList)`	
12		
13	`inputFile.close()`	
14		

Figure 4.4.14

Python's files can use `for-each` syntax to read each line in the file. On lines 7 and 9, we say for each line in the file, append the line to `myList` (after calling `strip()` to strip out whitespace and newlines). If the only thing stored in our file is the list's data, then this is all we need.

What happens, though, if our file was a mixture of lists and variables? For example, in Figure 4.4.6, we printed out `myInt1`, `myInt2`, `myInt3`, and `myList` all in one file. How do we load just the integers into the three `myInt` variables, and the list items into the list?

To do this, we do have to know which lines are going to be which kinds of data. In Figure 4.4.15, we know that the first three lines of the file are going to be the integers, so we can load these the same way on lines 6 through 8. Fortunately, though, the `for-each` loop by default starts with the next *unread* line, not at the beginning of the file. So, we just have to start the `for` loop after reading the first three integers.

At the end of Figure 4.4.15, the status of all four variables—the three `myInt` variables and `myList`—is the same as it was when we saved in Figure 4.4.6. Note also that we don't *have* to read line-by-line: we can also use the `read()` method (rather than the `readline()` method) to read the entire remaining contents of the file into one string, newlines and all.

#	LoadingintoLists-2.py	Output
1	`myList = []`	12 23 34 ['David', 'Lucy', 'Vrushali', 'Ping', 'Natalie', 'Dana', 'Addison', 'Jasmine']
2		
3	`#Open OutputFile.txt in read mode`	
4	`inputFile = open("OutputFile.txt", "r")`	
5		
6	`myInt1 = int(inputFile.readline())`	
7	`myInt2 = int(inputFile.readline())`	
8	`myInt3 = int(inputFile.readline())`	
9	`#For each line in the file`	
10	`for line in inputFile:`	
11	`    #Add the line to myList, stripping out whitespace`	
12	`    myList.append(line.strip())`	
13		
14	`print(myInt1)`	
15	`print(myInt2)`	
16	`print(myInt3)`	
17	`print(myList)`	
18		
19	`inputFile.close()`	
20		

Figure 4.4.15

Save and Load Functions

Generally, of course, we don't write programs whose sole purposes are to save and load random variables. We write programs to do other things, and we need to save and load data as part of what they do. For that reason, we typically write save and load as functions or methods to be called when needed. Let's see what that would look like real quick.

Figure 4.4.16 is just one way we could have done created these `save()` and `load()` functions; there are others. Here, lines 2 through 8 define a `save()` function that takes as input a filename and a list, and prints the list to the file. Lines 11 through 18, in turn, define a `load()` function that takes as input a filename, and then creates and returns a list, `inList`, populated by the lines of the file. The point is that we typically write save and load functions that mirror one another: loading data from a file loads it into the same variables from which it was saved, unless there's a special reason not to.

SaveandLoadFunctions.py

```
#Saves inList to the file
def save(filename, inList):
    outputFile = open(filename, "w")

    for item in inList:
        print(item, file = outputFile)

    outputFile.close()

#Loads from filename and returns a list of the contents
def load(filename):
    inputFile = open(filename, "r")
    inList = []

    for line in inputFile:
        inList.append(line.strip())
    inputFile.close()
    return inList

myList = ["David", "Lucy", "Vrushali", "Ping", "Natalie",
          "Dana", "Addison", "Jasmine"]
save("OutputFile.txt", myList)
newList = load("OutputFile.txt")

print(newList)
```

Output

```
['David', 'Lucy',
'Vrushali', 'Ping',
'Natalie', 'Dana',
'Addison', 'Jasmine']
```

Figure 4.4.16

5. Files and Turtles

Now that we've covered our save and load functions, let's make that the next thing we implement in our turtles. Let's create the ability to save all previously executed commands to a file, and load these commands from a file. That will let users share what they created! For now, we'll keep this simple: the save command will automatically save all previous commands, and the load command will automatically load and execute all previous commands.

We're going to be making several revisions this time around; instead of sharing each intermediate state, let's instead just look at the finished product, FilesandTurtles.py. We can talk through each individual change as we go.

Preparing to Save and Load

One way to do this would be to automatically save any command issued to a file. However, we don't want to just automatically save every session. We want to specifically give the user the option to save when they want to. To save something, we need to have it stored in memory, and right now, we lose our commands once they're run. So, the first thing we need to do is create a list of all the commands in order, to have available when we want to save.

So, to prepare for saving and loading, we need to keep that log of the commands that have been entered. We do that by creating `allCommandsList`, near the end of our program. Then, whenever the user executes a command, it's appended to `allCommandsList`; we want to save the commands as they're executed rather than as they're entered so that we don't have to worry about saving record and stop commands (instead, we'll save the "unpacked" version).

However, this presents an issue. `allCommandsList` is declared in the main part of our program. However, save will be a command that the user enters, so it will be processed inside `executeCommand()`. `executeCommand()`, however, can't see the variables of the main program. This was similar to our problem with `recordList` and `recording`: for `executeCommand()` to work on them, it needed to see them, but they were defined outside `executeCommand()`.

So, the way our program is structured right now, we would have to take care of saving and loading in that main while loop at the bottom as well. But this is getting messy: we want all our commands to be run through `executeCommand()`, not divided up between two places. We can do this, we just have to introduce a new principle: global variables.

Global Variable
A variable whose scope is the entire program; it is visible within any function or method in the program.

Global Variables

A **global variable** is a variable that can be seen across the entire program. We force it to have a large scope. It can be seen inside the functions even if it wasn't passed to them as an argument. Functions can be global as well, actually, and we've seen some: why were we able to see functions like `print()` and `open()` even though we never declared them anywhere? They were declared globally. We can declare our own variables and functions globally, too. In fact, it isn't hard to do in Python.

By default, any variable declared in our main program has the *potential* to be global. The problem lies with our functions. When we create a function, by default it only sees the parameters with which it was declared. So, our `executeCommand()` function, by default, sees only `commandTuple`. Any variables we declare inside are assumed to be local to the function, *unless* we specifically tell the function to go look outside of itself for the variables.

We do that on the first two lines: `global recording` and `global recordList`. These tell the function, "From here on, whenever you see a reference to `recordList`, go look at `recordList` in the main program". This is a subtle difference, but it's significant. This means that `executeCommand()` can now change whether the main program is recording or not.

So, when the user enters "record," `recording` is set to `True` here in `executeCommand()`. When the user enters "stop," the recorded commands are run inside `executeCommand()`. Now, the only "special" reasoning our main program needs is to (a) determine whether the newest command should be executed or recorded, and (b) to bypass that check if the command is "stop". Without the latter reasoning, a "stop" command would merely be added to the list of recorded commands; instead, the latter reasoning directs "stop" to be executed immediately.

Now we're prepared to actually create our save and load commands. Defining things globally will help us create our save command, but we'll also soon see that restructuring the recording process to use global variables will allow it to be loaded more easily as well.

The Save Command

With a global variable available, saving becomes actually a pretty easy process. We've already added reasoning to our program that saves every command as it comes in to `allCommandsList`. Now all we need to do is save `allCommandsList` to a file, and allow the user to execute that command.

So, we add three things:

- Save as an option in `getCommandFromUser()`. This will also allow the user to enter a filename to which to save. For now, we'll assume that the filename they enter is valid.
- Save as an option in `executeCommand()`. This will actually call the `save()` function if the user enters save as their command.
- The `save()` function itself.

Because everything we're saving is tuples, we're just going to save the tuples directly. This means we won't have to worry when reading these commands back in about how many parameters each has. So, when we save, we'll find that every line of the file represents one of the tuples.

The Load Command

Now for the slightly harder part: the load command. There are different ways we can do this. For now, let's construct the load command to load the commands from a file into a list and return the list. Let's also update our save command to ignore load commands, too, and let's keep assuming the file entered will be valid.

Adding our load function involves calling a library and method we've never seen before. The library is `ast`, and the method is `literal_eval()`. Don't worry about what this does for now, just know that this reads a line from a file and interprets it according to Python syntax; in this case, it correctly interprets a string that represents a tuple as a tuple. Note also you should generally do all your imports at the start of the program; I'm including the import here just to highlight it.

So, now we have a list of commands contained within `loadedCommands`. What do we do with it? Simple: we run them! Remember, we saved commands as they were *executed*, not as they were *entered*. That means that we don't need the reasoning for dealing with recording and stopping; if a set of three commands was repeated five times, then we saved all fifteen commands individually. So, after retrieving our loaded commands, we just iterate over them, executing them one by one.

The final result: it actually took relatively little work to create these save and load commands. Excluding the conversion of recording and stopping to global variables, we've only added 29 lines:

- Four lines to give the save and load options to `getCommandFromUser()`.
- Six lines to execute the save and load commands in `executeCommand()`.
- Four lines to build `allCommandsList`.
- Eight lines to build the save function itself.
- Seven lines to build the load function itself.

Notice that the way we've structured this, if we load some commands from a file, then save our own commands, the commands we loaded are saved as well; so, we can add other commands to a new program, and save them without having to save the original file.

Looking Forward

Note that this has gotten extremely complex. The goal here is not necessarily for you to understand every part. The goal is for you to understand the general idea of complexity and program flow. Trace through some executions of this code and notice how execution passes back and forth from function to function, how loops run over saved commands or recorded commands, and how tuples are used to communicate commands around. Notice how I've referenced some methods and specifically told you not to worry how they work, like `ast.literal_eval()`: that's part of programming, using methods you don't understand because you know their result.

There are also lots of places this program could still be improved:

- Files are still unchecked for validity. The program will crash if the user saves to an invalid filename, or loads from an absent file.
- Recording can't be nested. Since recording is stored globally, we can only ever have one recording at a time. It would be cool if we could include a recording within a recording, like a nested loop.
- Right now, we're looping over commands in three places: the main while loop, the loop repeating recorded commands, and the loop repeating commands loaded from the file. Ideally, we would only have one such loop. We could restructure the program to have only one `executeCommands()` function that takes as input a list of commands and executes them.

This is indicative of the overall process of writing programs: there's always room for expansion, improvement, and refinement. This is the programming life cycle: evaluating our code also gives us ideas for how to improve it even more going forward.

CHAPTER

4.5

Dictionaries

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Understand the nature of dictionaries as collections of keys and values, similar to variables and values, and the power therein;
- Construct dictionaries, as well as complicated reasoning centered around dictionaries simulating objects;
- Read and describe using function calls as values in maps.

1. What Is a Dictionary?

A dictionary is a book where, if you have a word, you can look for its definition. But you probably weren't wondering about a dictionary in the real world, you were probably wondering about a dictionary in computing. So, replace "book" with "data structure," "word" with "key," and "definition" with "value," and you have the definition of a **dictionary** in terms of computing: a dictionary is a data structure where, if you have a key, you can look for its value.

Dictionaries vs. Variables

Values are the same here as they have been everywhere else: a value is an actual data value, like "Hello, world" or 5 or 5.1. What is a **key**, then? A key is the name that brings up that value. Now you might be thinking: wait, isn't that just a variable? A variable is a name that, when referenced, brings up a value. And you'd be exactly right. Just as there is a 1:1 connection between variables and values, so also there is a 1:1 connection between keys in a dictionary and **values**.

So what makes them different? A dictionary is like a compilation of multiple key-value pairs that you can pass around together. Recall that one challenge of functions is that you can generally only return one value. A dictionary would let us return multiple values from a function. We've been able to do that before with lists, tuples, arrays, or some other structure like that, but a dictionary would *preserve* the ability to give names to those values by way of keys. So, in one way, we can think of a dictionary as a holding structure for several variables and their values.

Dictionaries vs. Lists

Like lists, dictionaries can hold multiple values. Depending on the language, these values might have an order to them. They might be sortable, or they might arrive in random order whenever they're accessed. Either way though, both lists and dictionaries store multiple values.

The key distinguishing factor of a list, however, was that the values of a list had to be accessed via a *numeric* identifier called an index. A list would have a first value, a seventh value, and a twelfth value. The only way to access the seventh value was to ask the list for the seventh value.

Imagine, though, that each item in the list was a student, with their name and their current average; maybe this was stored as a tuple, a list, an array, or even a custom object like what we'll discuss next unit. Now imagine that you knew you wanted to access David's grades. How would you do that with a list? You would have to iterate over the list one item at a time looking for David.

Dictionaries use keys instead of numeric indices. What that means is that you access the values of the dictionary by putting in the key, not by putting in an index. If you want David's grades, you use "David" as the key, and the grades pop right out. Keys are like non-numeric indices, but because they're non-numeric, they can take on intuitive meanings, similar to variable names.

Imagine, for example, storing calendar items. The key for each list of calendar items could be the date itself, so if we wanted the calendar items for September 12th,

Dictionaries
A data structure comprised of key-value pairs, where a key is entered into the dictionary to get out a value. Similar to or synonymous with Maps, Associative Arrays, HashMaps, and Hashtables.

Dictionary Key
A value then, when passed into a dictionary, returns a corresponding value, like a word and its definition. Similar to a variable.

Dictionary Value
A value returned in response to a key in a dictionary. Similar to a value of a variable outside a dictionary.

we could use September 12th as the key instead of trying to calculate what numeric index corresponded to September 12th.

Dictionary Terminology

I keep referring to this data structure as a dictionary because the original language of this material refers to it as a dictionary, but different languages have different names for structures like these. They can sometimes have subtle differences, but generally the concept is the same:

- "Dictionary," our current term, is so named because it echoes the idea of looking up words (keys) to find their definitions (values). Dictionaries suggest (but do not require) that the keys will be strings. Python, Swift, and the .NET languages use this term.
- "Map" is essentially a synonym to Dictionary. Some languages use the term "Map" because it echoes the mapping between keys and values and reinforce that keys need not be strings. Java and C++ use "Map".
- "Associative Array" is another synonym. This term echoes the idea of two arrays with associations between them. PHP, JavaScript, and others sometimes use this term.
- "Hash," "Hashtable," and "HashMap" are implementations of Dictionaries or Maps. These terms are somewhat synonymous, although they contain an extra layer of detail; the term "Hash" refers to the way in which the data structure is constructed, not just how it is used. Perl, Lisp, and Ruby use some of these terms.

So, if you see any of these terms, know that they refer to effectively the same thing: a data structure comprised of a collection of keys mapped to values.

2. Dictionaries in Python

One of the things that makes Python unique as a programming language is the accessibility of its dictionaries. In most languages, dictionaries are a little clunky to use; they can only be declared with special constructors (we'll talk about that next unit), and they're only usable through methods. Python, however, gives us an in-line method for defining dictionaries.

Creating and Accessing Dictionaries

For example, let's imagine we're creating an inventory program. In this case, our keys would be product names, and our values would be the current stock of that item.

How would we declare a dictionary with product names as keys and inventories as values? Well, we used brackets for lists and parentheses for tuples, I bet you can guess what we'll use for dictionaries.

```
#  CreatingandAccessingDictionaries-1.py
1  #Creates myDictionary with sprockets=5, widgets=11, cogs=3, and gizmos=15
2  myDictionary = {"sprockets" : 5, "widgets" : 11, "cogs" : 3, "gizmos": 15}
3  print(myDictionary)
4
5
Output
{'sprockets': 5, 'cogs': 3, 'widgets': 11, 'gizmos': 15}
```

Figure 4.5.1

To define a dictionary, we enclose it in braces, as shown on line 2 in Figure 4.5.1. There are actually several other ways to declare them, too, but we really only need one, and this one is most intuitive in my opinion.

For dictionaries, we need to declare both the key and the value. In lists and tuples, we could jump straight to the values because the indices were inferred: the

first item was 0, the second was 1, and so on. With dictionaries, however, our keys have to be supplied manually. So, each key-value pair in our dictionary is defined with the syntax `key:value`, as shown in line 2. Here, the key "sprockets" has the value 5, the key "widgets" has the value 11, and so on.

Functionally, this is the equivalent of creating four variables (`sprockets`, `widgets`, `cogs`, and `gizmos`), and assigning each the corresponding value (5, 11, 3, 15, respectively). However, because they're contained within the dictionary, we can pass this list to a function or method, and it would retain access to every key and its associated value. Notice also that the order of the pairs is different when we print the dictionary on line 3 from the way we declared it: dictionaries in Python aren't guaranteed to preserve the order of their values the way tuples, lists, and strings do.

Keys in dictionaries must be immutable: strings, integers, and floats can be keys. Tuples can also be keys, if and only if each individual item *in* the tuple is itself immutable (such as strings, integers, floats, or other immutable tuples). The reason for this is that if the key changes, the dictionary won't know what value is associated with it; so it must guarantee the keys cannot change.

Values, on the other hand, can change. In fact, if this program was to be used for inventory management, we would absolutely need to change the values. We can do this by accessing individual items from the dictionary, as shown in Figure 4.5.2.

```
# CreatingandAccessingDictionaries-2.py
1 #Creates myDictionary with sprockets=5, widgets=11, cogs=3, and gizmos=15
2 myDictionary = {"sprockets" : 5, "widgets" : 11, "cogs" : 3, "gizmos": 15}
3 print(myDictionary)
4 myDictionary["sprockets"] += 1
5 print(myDictionary)
Output
{'sprockets': 6, 'cogs': 3, 'widgets': 11, 'gizmos': 15}
```

Figure 4.5.2

The `print()` statements on lines 3 and 5 show that the operation on line 4 *does* change the value associated with the key "sprockets" in `myDictionary`. This also shows how we access individual items from a dictionary: using the same syntax as with lists, strings, and tuples, but with keys inside the brackets instead of indices.

Adding to and Removing from a Dictionary

So, we've created a dictionary in Figure 4.5.2. How do we add new items to it? You might be tempted to try to call an `append()` method like we used with lists, but in dictionaries, we don't need to. We create new key:value pairs in dictionaries the same way we create new variables: by assigning a value to a new key. Figure 4.5.3 shows this in action.

```
# AddingtoandRemovingfromaDictionary-1.py
1  #Creates myDictionary with sprockets=5, widgets=11, cogs=3, and gizmos=15
2  myDictionary = {"sprockets" : 5, "widgets" : 11,
3                  "cogs" : 3, "gizmos": 15}
4  print(myDictionary)
5
6  #Creates the new key "gadgets" with value 1
7  myDictionary["gadgets"] = 1
8  print(myDictionary)
9  del myDictionary["gadgets"]
10 print(myDictionary)
11
12
Output
{'widgets': 11, 'cogs': 3, 'sprockets': 5, 'gizmos': 15}
{'widgets': 11, 'gadgets': 1, 'cogs': 3, 'sprockets': 5, 'gizmos': 15}
{'widgets': 11, 'cogs': 3, 'sprockets': 5, 'gizmos': 15}
```

Figure 4.5.3

If we wanted to create a variable called gadgets and assign it the value 1, we would just say `gadgets = 1`. Similarly, if we want to create a new key in `myDictionary` called "gadgets" and assign it the value 1, we would just say `myDictionary["gadgets"] = 1`, as shown on line 7 of Figure 4.5.3. This creates the new key if it doesn't already exist, or reassigns it if it does already exist, just as if it was a variable. Similarly, we can delete a key by using that special `del` operator, as shown on line 9. The `print()` statements on lines 4, 8, and 10 confirm that the key "gadgets" was added, then removed.

The reason we create and use keys this way is that a dictionary cannot have multiple copies of the same key; if it did, it wouldn't know which value to return for that key. In other languages, the method for adding new keys is differentiated from the method for assigning values to existing keys, and those languages would throw an error if we tried to add a previously existing key again. Here, though, we have no way of adding a key besides assigning it as if it already exists.

Sometimes, though, we might want to check to see if a key exists before assigning it. For example, imagine if we were building a phonebook app, and if the user tries to create a key that already exists, we want to prompt them for a different key instead of changing the phone number of the existing key. We can do that by checking if a key exists before referring to it, as shown in Figure 4.5.4.

```
# AddingtoandRemovingfromaDictionary-2.py
1  #Creates myDictionary with David=4045551234, Lucy=4045555678,
2  #Vrushali=4045559101
3  myDictionary = {"David" : "4045551234", "Lucy" : "4045555678",
4                  "Vrushali" : "4045559101"}
5  print(myDictionary)
6
7  #Checks if "David" is a key in the dictionary
8  if "David" in myDictionary:
9      print("David is already in myDictionary!")
10     myDictionary["David2"] = "4045551121"
11 else:
12     myDictionary["David"] = "4045551121"
13 print(myDictionary)
14
Output
{'David': '4045551234', 'Vrushali': '4045559101', 'Lucy': '4045555678'}
David is already in myDictionary!
{'David': '4045551234', 'Vrushali': '4045559101', 'Lucy': '4045555678', 'David2':
'4045551121'}
```

Figure 4.5.4

The `in` operator, by default, operates on the keys of the dictionary. In Figure 4.5.4, "David" is already a key in the dictionary, so line 8 returns `True`, causing the code to try again with "David2" as the key. We can also use the keyword `in` to avoid the error caused when we look up a key that isn't in the dictionary, as shown in Figure 4.5.5.

If we're uncertain if a key will appear in the dictionary, we should either (a) check if it appears using `in` before trying to access it, or (b) be prepared to catch the `KeyError` shown in Figure 4.5.5.

```
# AddingtoandRemovingfromaDictionary-3.py
1  #Creates myDictionary with #David=4045551234, Lucy=4045555678,
2  #Vrushali=4045559101
3  myDictionary = {"David" : "4045551234", "Lucy" : "4045555678",
4                  "Vrushali" : "4045559101"}
5  print(myDictionary["Dana"])
6
Output
Traceback (most recent call last):
  File "AddingtoandRemovingfromaDictionary-3.py", line 3, in <module>
    print(myDictionary["Dana"])
KeyError: 'Dana'
```

Figure 4.5.5

Traversing Dictionaries

As noted, the main benefit of dictionaries is that they give us useful keys so we can jump straight to the value we want. However, there will still be lots of times we want to traverse every item in a dictionary. For example, in our inventory program, perhaps we want to make sure to order more of any item that drops below 5. There are a number of ways we could do this. First, if we don't care what key gives us these values, we could iterate over the `values()` directly, as shown in lines 5 through 7 of Figure 4.5.6.

`values()`
A method of the dictionary type that returns a list of all the values of the dictionary.

```
# TraversingDictionaries-1.py
1 #Creates myDictionary with sprockets=5, widgets=11, cogs=3, gizmos=15,
2 #gadgets=1
3 myDictionary = {"sprockets" : 5, "widgets" : 11, "cogs" : 3, "gizmos" : 15,
4                 "gadgets" : 1}
5 for value in myDictionary.values():
6     if value < 5:
7         print("A value less than 5 was found:", value)
8
9

Output
A value less than 5 was found: 1
A value less than 5 was found: 3
```

Figure 4.5.6

More commonly, though, we'll want the key and the value. So, instead, we could iterate over the keys, and then grab the value from the dictionary based on the key, as shown in lines 5 through 8 of Figure 4.5.7. Notice that here we're iterating over `myDictionary.keys()` in line 5, but this has the same effect as iterating over `myDictionary` itself; it assumes it should use keys if we don't tell it otherwise.

`keys()`
A method of a dictionary type that returns a list of all the keys in that dictionary.

```
# TraversingDictionaries-2.py
1 #Creates myDictionary with sprockets=5, widgets=11, cogs=3, gizmos=15,
2 #gadgets=1
3 myDictionary = {"sprockets" : 5, "widgets" : 11, "cogs" : 3, "gizmos" : 15,
4                 "gadgets" : 1}
5 for key in myDictionary.keys():
6     value = myDictionary[key]
7     if value < 5:
8         print(key, "is less than 5:", value)
9

Output
cogs is less than 5: 3
gadgets is less than 5: 1
```

Figure 4.5.7

It's also possible, however, to iterate over the keys and values simultaneously. This is a shortcut similar to the `for-each` loop itself where multiple parts of the sequence can be loaded into variables. Here, it's assumed that the order is key, value, as shown in line 5 of Figure 4.5.8. So, we can iterate over the keys in the dictionary, the values in the dictionary, and the keys and values together.

```
# TraversingDictionaries-3.py
1 #Creates myDictionary with sprockets=5,
2 #widgets=11, cogs=3, gizmos=15, gadgets=1
3 myDictionary = {"sprockets" : 5, "widgets" : 11,
4                 "cogs" : 3, "gizmos" : 15, "gadgets" : 1}
5 for (key, value) in myDictionary.items():
6     if value < 5:
7         print(key, "is less than 5:", value)
8
9

Output
gadgets is less than 5: 1
cogs is less than 5: 3
```

Figure 4.5.8

3. Dictionary Applications

Dictionaries are extremely powerful data structures, even when just used the way we've discussed them so far. However, some of the benefits of dictionaries are that they allow us to create a low-overhead version of object-oriented programming. Object-oriented programming is our next chapter, and it's a big and important topic, so we'll preview it here.

Simple Dictionary Applications

With a single dictionary, there are a lot of things you can do. For example, imagine you want to count the most common words in a book. If you have the book in plaintext, how would you do that? First, you'd likely replace all the punctuation marks and other symbols with spaces, so that you don't get stuck treating a word with a period after it as a different word from its other appearances. Then, you'd probably change the entire thing to lower or upper case so you don't have to worry about capitals.

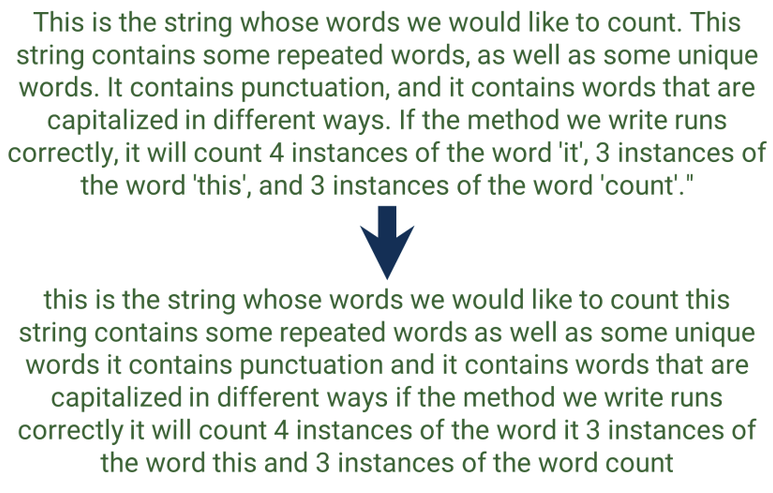


Figure 4.5.9

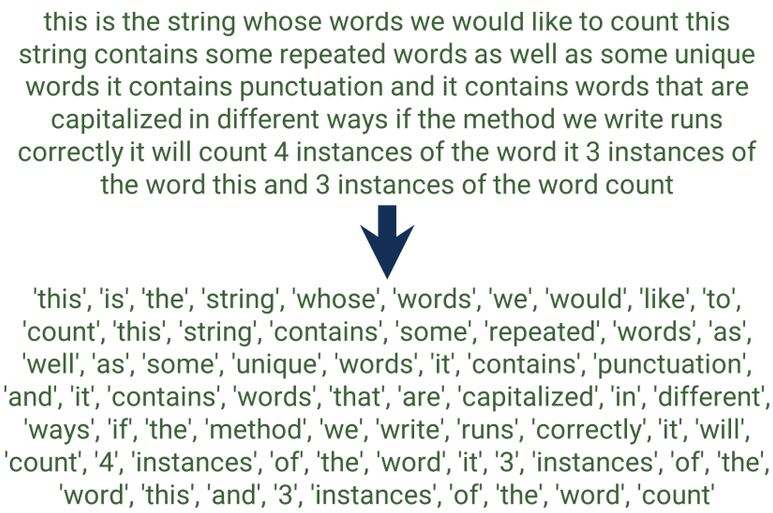


Figure 4.5.10

After that, though, what would you do? You very likely might split the book by spaces, then start iterating over each individual word. When you find a word you haven't seen before, you add it as a key to your dictionary with a value of 1. When you find a word you have seen before, you look for it in the dictionary and increment its value. In the end, your keys are all the words in the book, and your values are all the counts of each word.

Alternatively, imagine you were creating a seating chart for a wedding. How would you do this? You could have a list of all the seats, or a list of tables each with a list of seats. However, this would mean you would need to know the table and seat number to look who was sitting there. In all likelihood, that's backwards: you're probably not looking for, "Who's at table 5 seat 3?", you're looking for, "Where is Addison sitting?" So, you could do this with a dictionary instead: the keys are the individuals' names, and the values are the seat assignments, as either strings (e.g. "5-3") or as tuples (5, 3). That gets into our more complex application of dictionaries: merging dictionaries with lists, tuples, arrays, or other dictionaries.

Wedding Seating Chart Dictionary

Names are keys; table assignments are values

David	3	Lucy	3	Dana	2
Addison	2	Vrushali	1	Bilbo	3
Sarah	1	Lugos	1	Mireia	1
Partha	2	Venijamin	1	Terra	2
Tryphon	3	Gevorg	1	Raza	3
Rein	3	Sofia	2	Perle	2

Figure 4.5.11

Dictionaries and Lists

A big part of the usefulness of dictionaries comes when we use them in conjunction with other lists or dictionaries. For example, imagine we're keeping track of students across a school at a given time of day. We might have a dictionary of classrooms, where the keys are the classroom numbers and the values are lists of students in these classrooms. That way, at any given time we can look for how many students are in a class as well as which students are in a class.

Alternatively, imagine we're building a more comprehensive address book program. We want each person in the address book to have a name, an address, a phone number, and an email address. We want to access them by their names. So, the keys would be the names, and the values could be a tuple or list, containing their address, phone number, and email address.

However, that design still poses a problem: because the address, phone number, and email address are stored in a list or tuple, we have to remember which index corresponds to which kind of data. If we forget, we may load the phone number as the email address or vice versa. However, dictionaries let us use non-numeric keys instead of numeric indices, and that is immensely powerful.

Dictionaries as Simple Object-Oriented Programming

We'll cover object-oriented programming in the next unit. For now, though, what you need to know is that an object is a custom data type that can contain multiple individual variables and methods, each with its own name. That's powerful; we could, for example, create a Person object that would let us store a first name, last name, and phone number together in one data type, but each individually accessible.

However, dictionaries actually already let us do that. In this address book example, we had a dictionary of tuples or lists, where the three items on the list represented addresses, phone numbers, and email addresses. Instead of collecting these in a list, however, we could collect them in a dictionary, where the keys are the types of information ("address," "phone number," and "email address"), and the values are that individual person's values.

The unique twist here is that in this case, we would have lots of dictionaries, each with only a few items. Most importantly, though, each dictionary would have the *same keys*. So, we could iterate over every person in the address book and get their email address just by requesting their email address by name from their individual dictionary. This is immensely powerful.

4. Dictionary Applications in Python

Before moving on to these super-advanced types of dictionaries, let's explore some of the more accessible applications a little bit more.

Simple Examples of Dictionaries

First, let's see some code that counts the words in a string using a dictionary. This is shown in Figure 4.5.12.

#	SimpleExamplesofDictionaries-1.py	Output
1	`myString = "This is the string whose words we would like to count. This string contains some repeated words, as well as some unique words. It contains punctuation, and it contains words that are capitalized in different ways. If the method we write runs correctly, it will count 4 instances of the word 'it', 3 instances of the word 'this', and 3 instances of the word 'count'."`	`{'method': 1, '4': 1, 'in': 1, 'count': 3, 'instances': 3, 'if': 1, '3': 2, 'words': 4, 'and': 2, 'the': 5, 'string': 2, 'whose': 1, 'punctuation': 1, 'runs': 1, 'well': 1, 'word': 3, 'different': 1, 'will': 1, 'repeated': 1, 'are': 1, 'correctly': 1, 'ways': 1, 'of': 3, 'like': 1, 'is': 1, 'contains': 3, 'that': 1, 'write': 1, 'some': 2, 'to': 1, 'as': 2, 'this': 3, 'we': 2, 'capitalized': 1, 'would': 1, 'unique': 1, 'it': 4}`
2		
3	`myString = myString.replace(".","") #Remove periods`	
4	`myString = myString.replace(",","") #Remove commas`	
5	`myString = myString.replace("'","") #Remove apostrophes`	
6	`myString = myString.lower() #Make all lower case`	
7	`mySplitString = myString.split() #Split by spaces`	
8		
9	`wordDictionary = {} #Create empty dictionary`	
10	`for word in mySplitString:  #For each word in the split string`	
11	`    if word in wordDictionary:  #If it's already been found...`	
12	`        wordDictionary[word] += 1   #Add one to its count`	
13	`    else:   #Otherwise...`	
14	`        wordDictionary[word] = 1 #Create it with value 1`	
15		
16	`print(wordDictionary)`	
17		

Figure 4.5.12

We start by defining the string, `myString`, on line 1. We then modify it on lines 3 through 7 it so that it contains only the words: we remove punctuation (lines 3, 4, and 5), make it lower case (line 6), then split it by spaces so that we have a list of words (line 7). We then iterate over that list of words starting on line 10. For each word that isn't yet in the dictionary, the conditional on line 11 is `False`, so we add it to the dictionary with a value of 1 on line 14 because 1 instance has been found. For each word that is already in the dictionary, the conditional on line 11 is `True`, so we just increment its counter on line 12. In the end, our dictionary contains all the words in the string as keys, and the count of these words as their corresponding values.

For our second example, let's define the seating chart for a wedding. First, we'll show how this would be used to look up a particular person's seat. Second, we'll show how we would use this to find a listing of all the people at each table. For example, we'll keep it simple and assume we're only assigning people to tables, not to specific seats at tables. We'll assume 3 tables, 6 seats each. This code is shown in Figure 4.5.13.

SimpleExamplesofDictionaries-2.py

```
1  seatingChart = {"David" : 3, "Lucy" : 3, "Dana" : 2,
2                  "Addison" : 2, "Vrushali" : 1, "Bilbo" : 3,
3                  "Sara" : 1, "Lugos" : 1, "Mireia" : 1,
4                  "Partha" : 2, "Venijamin" : 1, "Terra" : 2,
5                  "Tryphon" : 3, "Gevorg" : 1, "Raza" : 3,
6                  "Rein" : 3, "Sofia" : 2, "Perle" : 2}
7
8  #For each name, table pair in the seating chart
9  for (name, table) in seatingChart.items():
10     #Print the table for the name
11     print(name, " is seated at table #", table, sep="")
12
13 print()
14 #For each table number
15 for i in range(1, 4):
16     print("The guests at table #", i, " are: ", sep="", end="")
17     #For each name, table pair
18     for (name, table) in seatingChart.items():
19         #If the table numer is this number
20         if i == table:
21             #Print the name
22             print(name, end=" ")
23     print()
24
25
```

Output

```
Sofia is at table #2
Lucy is at table #3
Raza is at table #3
Dana is at table #2
Terra is at table #2
Lugos is at table #1
Venijamin is at table #1
Tryphon is at table #3
Gevorg is at table #1
Mireia is at table #1
...

The guests at table #1 are:
Lugos Venijamin Gevorg
Mireia Vrushali Sara
The guests at table #2 are:
Sofia Dana Terra Partha
Addison Perle
The guests at table #3 are:
Lucy Raza Tryphon Rein Bilbo
David
```

Figure 4.5.13

First, we create the seating chart with 18 key:value pairs on line 1; each pair is a name and a table number. Notice how we can separate the dictionary declaration onto multiple lines after commas. Then, we iterate over each name:table pair in the dictionary starting on line 9 and print the name and its table; `seatingChart.items()` returns these pairs as tuples. Then on lines 15 through 23, we iterate over the three table ID numbers 1, 2, and 3. Within each iteration, we iterate over each person on the seating chart on lines 18 through 22 and check if their table number matches the current table number on line 20. If so, we print their name on line 22.

`items()`
A method of the dictionary type that returns all the items in the dictionary as (key, value) tuples.

Dictionaries and Lists

For our classroom roster, we first define a dictionary of classes, as shown in lines 1 through 5 of Figure 4.5.14. Each class has a name as the key and a list of students (as defined by the brackets) as the value. Notice here how easily we can use lists as

DictionariesandLists-1.py

```
1  classes = {"Math" : ["David", "Lucy", "Dana"],
2             "Physics" : ["Addison", "Vrushali", "Bilbo"],
3             "Chemistry" : ["Sara", "Lugos", "Mireia", "Perle"],
4             "Computing" : ["Partha", "Venijamin", "Terra", "Sofia"],
5             "History" : ["Tryphon", "Gevorg", "Raza", "Rein"]}
6
7  print("Students in Computing:", classes["Computing"])
8  #Add Francis to History
9  classes["History"].append("Francis")
10 print("Students in History:", classes["History"])
11
12
13
14
```

Output

```
Students in Computing: ['Partha', 'Venijamin', 'Terra', 'Sofia']
Students in History: ['Tryphon', 'Gevorg', 'Raza', 'Rein', 'Francis']
```

Figure 4.5.14

values instead of just simpler values on their own; we just put the same brackets and declaration as we usually would to define a list.

Then, we look up all the students in Computing with `classes["Computing"]`. This returns a list of students, which we print on line 7. Then, we decide to add Francis to History. We get the list corresponding to History with `classes["History"]` on line 9, and because this is a list, we can call `append("Francis")` on it. Then, when we print the History class roster on line 10, we see Francis is included.

For our address book, we sub out the lists for tuples, as shown in Figure 4.5.15. Here, we see on line 5 that we can print David's complete information by printing his tuple, or we can print Dana's phone number alone on line 6 by knowing that the phone number is at index 1. But that last note is exactly where we find some real power in dictionaries: why should we have to remember that the phone number is at index 1 when we can instead use a dictionary and store the phone number with key "phone number"?

```
# DictionariesandLists-2.py
addressBook = {"David": ("555 Home St", "4045551234", "david@david.com"),
               "Lucy" : ("555 Home St", "4045555678", "lucy@lucy.com"),
               "Dana" : ("123 There Rd", "4045559101", "dana@dana.net")}

print("David's Information:", addressBook["David"])
print("Dana's Phone Number:", addressBook["Dana"][1])
```

```
Output
David's Information: ('555 Home St', '4045551234', 'david@david.com')
Dana's Phone Number: 4045559101
```

Figure 4.5.15

Dictionaries as Objects

One of the most powerful parts of using dictionaries is the ability to have multiple dictionaries with the same keys, but different values. This data structure is a low-overhead version of object-oriented programming, which we'll cover in the next chapter. To start, let's convert that address book code to use these nested dictionaries in Figure 4.5.16.

```
# DictionariesasObjects-1.py
addressBook = {"David": {"address" : "555 Home St", "phone" : "4045551234",
                         "email" : "david@david.com"},
               "Lucy" : {"address" : "555 Home St", "phone" : "4045555678",
                         "email" : "lucy@lucy.com"},
               "Dana" : {"address" : "123 Here Rd", "phone" : "4045559101",
                         "email" : "dana@dana.net"}}

print("David's Information:", addressBook["David"])
print("Dana's Phone Number:", addressBook["Dana"]["phone"])
```

```
Output
David's Information: ('555 Home St', '4045551234', 'david@david.com')
Dana's Phone Number: 4045559101
```

Figure 4.5.16

In this case, our code has gotten a bit longer: instead of the inferred numeric indices, we now supply actual keys for the dictionaries, as shown on lines 1 through 6. Compare Figure 4.5.16 to Figure 4.5.15 to see the difference. However, the trade-off is that this is now much easier to use. We don't need to remember the order in which the different types of data are contained inside the items of the dictionary; we just need to remember that phone number is stored with key "phone", address with key "address," and email address with key "email". Don't underestimate that trade-off: it might seem easy to remember this for these three values, but imagine if you were dealing with several different dictionaries with several different types of data, or with one dictionary with twenty different fields. With lists, you would need to remember the index for each type of data, as well as remember to declare them in the right order. With dictionaries, you merely need to remember the name for each type of data, similar to how you would remember a variable.

Note that this merely implements something resembling object-oriented programming in the simplest terms. In real object-oriented programming, we can have methods as well as variables contained within each item, as well as other advanced features. Still, this approach gives a glimpse of the power of object-oriented programming, as well as a way to realize some of its benefits with relatively low overhead.

To show this off, let's take one more example. Imagine we're teaching a class with five students, and we gave a multiple choice test with five questions. Now, we want to grade the test. We could do that with dictionaries, as shown in Figure 4.5.17.

```
#  DictionariesasObjects-2.py
1  ANSWER_KEY = {"1" : "A", "2" : "B", "3" : "C", "4" : "D", "5" : "A"}
2
3  students={}
4  students["David"] = {"1" : "A", "2" : "B", "3" : "A", "4" : "B", "5" : "C"}
5  students["Terra"] = {"1" : "A", "2" : "B", "3" : "C", "4" : "D", "5" : "A"}
6  students["Lugos"] = {"1" : "A", "2" : "C", "3" : "C", "4" : "D", "5" : "A"}
7
8  #For each student and their answers
9  for (student, answers) in students.items():
10     grade = 0   #Start grade at 0
11     #For each question and answer
12     for (question, answer) in answers.items():
13         #If the answer matches ANSWER_KEY's answer...
14         if answer == ANSWER_KEY[question]:
15             grade +=1   #Increment their grade
16     #Create a new key "grade" and assign it their grade
17     students[student]["grade"] = grade
18 #For each student and their answers
19 for (student, answers) in students.items():
20     #Print the name and grade
21     print(student, ": ", answers["grade"], sep = "", end = "; ")
22
Output
Lugos: 4; David: 2; Terra: 5;
```

Figure 4.5.17

Here, we create three students and give them five answers to the five questions each on lines 4, 5, and 6. We also create an answer key on line 1. Then with the `for` loop on line 9, for each student in the dictionary of students, we grab their answer dictionary and store it in `answers`. Then, for each question in the dictionary `answers`, we grab its `answer` with the `for` loop on line 12. If the value for `answer` to the key `question` matches `ANSWER_KEY`'s value (answer) for the corresponding key (question) in the conditional on line 14, we add one to `grade` on line 15. Then, at the end on line 17, we create a new key-value pair between "grade" and their score, stored in the variable `grade`.

The loop at the end (lines 19 to 21) isn't totally necessary here; we could have printed out the grades in the first `for` loop. However, it's here to demonstrate the

power of knowing that every element in the dictionary will have the same key. We know every student will have a "grade" key because we created it for each student on line 17, so we can run a loop over all students, getting their grade.

5. Dictionaries and Turtles

Over the past several lessons, our turtles code has gotten more and more complex. In some ways, that's not bad: it's a complicated program, and its goal is to show you how the simple rules you've been learning can be chained together to form massively complex programs.

In other ways, though, it's a bad thing. Giant multi-layer if statements are often frowned upon because they're difficult to read and navigate. We might also notice that our commands are, in some ways, like keys to a dictionary: it's just that in this case, the values are the lines of code that run in response, not variables. Could we create dictionaries whose values are actually references to functions? Yes! Follow along in DictionariesandTurtles.py to see how.

Replacing Conditionals with Dictionaries

Let's start by trying to replace the giant series of conditional statements that get the user's input with something dictionary-based. Take a look at that conditional inside `getCommandFromUser()`: notice anything? Every branch follows the same pattern: get arguments, then create `commandTuple` with the command in the first spot. Instead of this complex branching series, we could instead create little functions for each, and then just call the function based on the command name as established by a map. Let's try that out.

You might have to trust me for now when I say this code is better. It seems incredibly strange. What have we done? We've taken the entire body of `getCommandFromUser()` and `executeCommand()`, and we've split each branch into its own function, named things like `getSnowflakeCommand()` and `executeTextCommand()`. Why did we do that?

Take a look at `COMMAND_DICTIONARY`. Here, we see every command the user can enter, from turn to end. This is a dictionary, so the command names themselves are keys. What are the values? Other dictionaries! What are the keys of these other dictionaries? Every one of these dictionaries has the keys `"get"` and `"execute"` What are the values? Brace yourself: the values are *the functions themselves*. Functions are variables. So, we can put the function names here.

So what happens? Well, the `while` loop that drives the program is unchanged. So, let's take a look at `getCommandFromUser()`. Previously, this would get a command from a user, then run a big conditional statement checking the command's value. It would choose which arguments to get based on the command's value. It was a long function. What about now? It's only 6 lines! How did we replace over 30 lines with 6?

The first two lines are the same: we get a command from the user. However, instead of using that command in a conditional, we instead use it as the key to our `COMMAND_DICTIONARY`. If it's not a key in the dictionary, our `commandTuple` is just the "Invalid" tuple. If it *is* a key in the dictionary, though, the magic happens.

Let's take a close look at the line here: return `COMMAND_DICTIONARY[command]["get"]()`. What a crazy line. It starts with return, so we know what comes out of the rest of it must be our `commandTuple`. We start from left to right. `COMMAND_DICTIONARY[command]` takes the command and uses it as the key to the dictionary. We know the command is valid because of the earlier conditional. So, what comes out of `COMMAND_DICTIONARY`?

Let's assume the user's inputted command was 'forward'. So, our line is now `COMMAND_DICTIONARY["forward"]["get"]()`. The result of `COMMAND_DICTIONARY["forward"]` is another dictionary. This other dictionary has two keys, "get" and "execute". So, `COMMAND_DICTIONARY`["forward"]["get"] returns the value associated with "get" in the second dictionary. What value is that? `getForwardCommand`. It's the name of the function. Then, the end of the line is (), which tells Python to actually run the function.

So, `COMMAND_DICTIONARY[command]["get"]()` tells Python to run the "get" function associated with the command the user entered. That function does the same thing it did before: it gets the user's input (in this case, just a distance), creates the `commandTuple` that holds the command and the arguments, and returns it. Since we're returning the result of `COMMAND_DICTIONARY[command]["get"]()` as well, we return the `commandTuple` all the way back out to the main program.

After all that work, the result is exactly the same. We could have made these changes and our users would never even notice. Notice that `executeCommand()` is the same. Instead of a big conditional, it uses a map to find the right execute command to run. Here, though, it actually passes the `commandTuple` itself so that the execute commands can have their arguments. Other than that, the only difference is that we have to define recording, `recordList`, and `allCommandsList` as global within the execute functions for record, stop, and save.

As I've said repeatedly before, don't worry if this is confusing. In fact, if this *isn't* confusing, please contact me so that I can hire you to teach this class. This is an incredibly complex example of Python reasoning and syntax. It's taken me several hours to compile this to this point, and I've learned a lot in the process. The goal is not for you to understand exactly how this works. The goal is for you to see an example of a complex, authentic Python program as it is developed. Do try to understand it as best as you can, but don't be discouraged by it.

Why Is This better?

The big question you might have is: In what way is this actually better? "I understood the earlier way!" you might say. "You're just making it more complex on purpose to show off these principles!"

While that's partially true, this actually is better. Why is it better? Good programming practice is typically built around lots of small, reusable functions. Originally, our main program was extremely large. When we converted `getCommandFromUser()` and `executeCommand()` to functions, we shrunk down the size of the main program, but we replaced it with two even bigger functions.

Now look at our code, though. No function over 8 lines. Our main program code is only 7 lines, 11 if you include the initializations of things like command and recording. Our functions are highly reusable: we could copy `executeDrawSnowflake()` to any program that stored arguments the same way we do.

Consider also how we would add new commands to this now. We would add them to the `COMMAND_DICTIONARY`, then also create dedicated functions for them. However, we don't have to trace through the program code to find the right place to add the functions or their reasoning. We don't have to create a new branch of a giant conditional. Just by adding them to the dictionary and creating their functions, we add them to the program. We could copy and paste someone else's functions directly into our code without worrying about integrating it beyond just adding it to our list of supported commands.

That's what makes this code better: small, modular functions with very few interdependencies.

UNIT 5

OBJECT-ORIENTED PROGRAMMING

CHAPTER

5.1

Objects

1. What Are Objects?

At this point, we've covered the basics of computing. Congratulations! So far, you've covered the principles and methods that put a man on the moon. The techniques you now know covered the state of the art of computing for its first several decades of existence.

Surprisingly, though, the principles needed to put a man on the moon were far simpler than the principles needed to load up that gif of a cat falling down on your phone. Modern cloud computing, virtual reality, and even typical web development require more complex frameworks and paradigms than the ones responsible for the early days of the space program.

Unit 5 is our "advanced topics" unit. In this unit, we're going to preview the next topics you'll cover if you decide to continue your education in computing. We're going to focus on two topics: objects and algorithms. In many ways, these two topics cover two different directions you could choose to go in computing. If you want to go into developing websites, video games, or other portions of the design side of computing, you'll be using objects a lot. If you want to go into machine learning, theory, or the more mathematical side of computing, you'll spend a lot of your time developing algorithms.

What Are Objects?

That brings us to the topic for this chapter. What *are* **objects**? Surprise! You've actually been interacting with objects throughout the past several chapters; we've just glossed over them and promised to come back to them later. Well, later is now.

Objects are custom data types that you get to create. We usually refer to the data types themselves as **Classes**. Just like you've been using data types to represent numbers, letters, and strings, you can create data types or classes to represent people, places, and items. In creating these, you specify multiple variables to be wrapped up into one data type.

One of the biggest strengths of programming with objects is that it lets us write programs the way we think about things in the real world. We're naturally predisposed to think about the world in terms of generic objects. For example, you have the concept in you of a person. A person would be a single entity that has lots of variables assigned to them. They have a first name, a last name, and a middle name. They have a height, a hair color, and an eye color. They have a phone number and an e-mail address.

These are all variables that we could wrap up into one data type, and call that data type a `Person` class. The class tells us what variables should be specified for that type of object. A `Person` object would almost certainly have a first name and a last name, and that's the reason why we know that asking, "What's your name?" is a logical question to ask a new person. We know they should have a name because these are variables in that type. By that same principle, if we had a `Chair` object, we would find it very silly to ask, "What's your name?" to a chair because "name" probably isn't a variable we'd associate with a chair type.

Lesson Learning Objectives

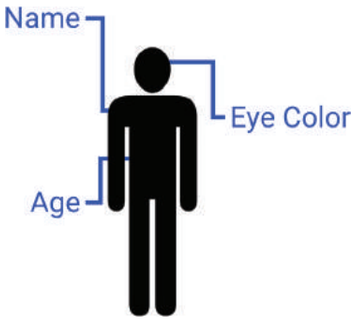
By the end of this chapter, students will be able to:

- Describe the use of objects and recognize the difference between an object and an instance;
- Explain the principles of object-oriented programming;
- Define objects in Python and use principles of object-oriented programming for declaring methods and combining classes.

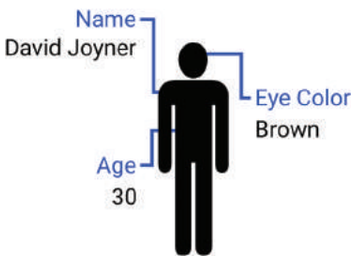
Object-oriented programming
A programming paradigm where programmers define custom data types that have custom methods embedded within them.

Object
An object is a custom data structure that organizes and encapsulates variables and methods into a single data type. It is used near-interchangeably with "instance."

Class
A custom data type comprised of multiple variables and/or methods. Instances or objects are created based on the template provided by the class.



Instance
A single set of values of a particular class. Classes may be comprised of multiple variables; an instance is a set of values for these variables. The term "instance" is often used interchangeably with the term "object".



Objects vs. Instances

So, a class is a generic structure for a certain kind of data. If we have a `Person` class, we expect it to have a name, a height, and an eye color. These are variables we expect to exist about people.

An **instance**, on the other hand, is a *specific* person. A `Person` class is a data type with variables for name, age, and eye color. David Joyner is an *instance* of a `Person` object with name "David Joyner", age 30, and eye color brown. "David Joyner", 30, and brown are values for these variables, just as David Joyner is an instance of an object of type `Person`. It might seem weird to differentiate the name "David Joyner" from the `Person` David Joyner, but ask yourself: if I change my name, am I suddenly a different person? No; having a name is a variable, my actual name is that variable's value, but even if I change that variable's value, the person is the same. It's just difficult to refer to people without using their names! Think about this with something like chairs, though: a chair has a serial number, which is an unchanging name of that specific chair, differentiating it from other chairs of the same type. Even if we remove a leg and change the color, the serial number remains the same.

Classes are the general description of the types of variables associated with the type. An instance of the class is a particular example of an object of that type. You know what a person is, and you know that David Joyner is an example—an instance—of a person. Similarly, you know what a chair is in general, and the chair you're probably sitting on right now is an instance of a chair. You know that chairs can have lots of variables, like a number of legs, a color, and a material; similarly, you know that the chair you're sitting on is white, wooden, and has four legs. You know that not all chairs are white with four legs, just as you know that not all people are named "David Joyner". The general concepts are classes, and the specific examples are instances of these classes.

2. Objects and Instances in Python

To get started, let's talk just about how to define classes and instances in Python. We've actually already seen these in some places, but we didn't refer to them by these terms. For this lesson, let's stick to the running `Person` example. A `Person` should have a first name, a last name, an eye color, and an age. We'll represent the names and eye color as strings and the age as a number.

Declaring a Class

Figure 5.1.1 shows how we would define our `Person` class. Line 2 here is the line that starts off the creation of a new object. This is similar to what we've seen in the past with loops and conditionals: the reserved word `class` tells the computer that it's about to see a code block contained within the class, after the class name (Person) and a colon as usual. The contents of the class is indented.

#	DeclaringaClass-1.py	Output
1	`#Define the class Person`	
2	`class Person:`	
3	`    #Create a new instance of Person`	
4	`    def __init__(self):`	
5	`        #Person's default values`	
6	`        self.firstname = "[no first name]"`	
7	`        self.lastname = "[no last name]"`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		

Figure 5.1.1

What do we see next? On line 4, we see… a function! A function with the strange name `__init__` and the strange parameter `self`. We'll talk more about `__init__` later, but for now, what you need to know is that `__init__` is called when we first create a new instance of this class. Think back to when we would define strings: The string class actually had an `__init__` method that was called whenever we created a new string. It just executes some code that will be needed for the rest of the program. Here, when we create a new `Person`, we want to give some default values to the names and eye color that emphasize that the real values haven't yet been supplied.

When a function is defined *inside* a class, we call it a **method**. It still works the same way: when the method is called, the lines of code are executed in order. In the `__init__()` method here in Figure 5.1.1, there are four lines. Notice that each line starts with the variable name `self`. `self` is a little strange: it tells Python to define the following variable (like `firstname`) for the instance as a whole. If we leave off `self`, the variable has the same scope as a variable normally would in a function: it stops existing when the function is over. So, writing `self.firstname` says, "Any time we look at this instance's `firstname`, it should be the same one!" Every method declared inside a class should have `self` as the first parameter, and every variable for the class should be preceded by `self` every time the variable is used inside the class. `self` is a little like saying 'my'; it collects together the class's variables.

Method
A function defined inside of a class.

Self
A keyword in Python classes that is used to refer to the instance itself. It defines the scope of variables and methods that methods in the class can see.

Defining the `Person` class works just like defining the functions we've seen in the past. Seeing the line `class Person` tells the computer, "Hey, you need to know this concept of a `Person`." Later on, we can actually use this concept in our code. Think of declaring a class like teaching someone a concept. If you knew someone wasn't familiar with books, you would teach them that every book has a title and an author; then when you give them a book, they would know to look for the title and author. Teaching them the idea of a book having a title and author is like declaring a class; handing them a copy of *Introduction to Computing* by David Joyner is like having them create an instance.

Once defined, though, classes work like any data type. You can use instances of them as values in lists or tuples. If some of their variables are lists, you can loop over these lists. You can even use classes as values for *other* classes. For example, we could create a `Name` class that has two variables, `firstname` and `lastname`, as shown in Figure 5.1.2.

#	DeclaringaClass-2.py	Output

```
1  #Define the class Name
2  class Name:
3      def __init__(self):
4          self.firstname = "[no first name]"
5          self.lastname = "[no last name]"
6
7  #Define the class Person
8  class Person:
9      #Create a new instance of Person
10     def __init__(self):
11         #Person's default values
12         self.name = Name()
13         self.eyecolor = "[no eye color]"
14         self.age = -1
15
```

Figure 5.1.2

In Figure 5.1.2, an instance of the `Name` class supplies the `firstname` and `lastname` to the `Person` class. We can use this to create extremely complex data structures that are nonetheless easy to use because everything is organized in logical ways.

Creating Instances

Now that we have the class declared, we can use it in our program! We declare a new person with the line `myPerson = Person()`, as shown on line 12 of Figure 5.1.3. Notice that this syntax looks like we're calling a function because of the parentheses after `Person`. This line is effectively like calling `Person.__init__()`, but Python is smart enough to let us leave out `__init__` when it's written exactly like that. So, calling `Person()` is like saying to the computer, "Give me a new instance of `Person`!" As a result, the computer creates a new instance, runs `__init__` to do the initial setup, and then returns that new instance so that we can assign it to a variable, all on line 12.

#	CreatingInstances-1.py	Output
1	`#Define the class Person`	`[no first name]`
2	`class Person:`	`[no last name]`
3	`    #Create a new instance of Person`	`[no eye color]`
4	`    def __init__(self):`	`-1`
5	`        #Person's default values`	
6	`        self.firstname = "[no first name]"`	
7	`        self.lastname = "[no last name]"`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		
11	`#Create a new Person and assign it to myPerson`	
12	`myPerson = Person()`	
13	`#Print myPerson's values`	
14	`print(myPerson.firstname)`	
15	`print(myPerson.lastname)`	
16	`print(myPerson.eyecolor)`	
17	`print(myPerson.age)`	
18		

Figure 5.1.3

Then, we can use `myPerson` like a normal variable. By calling `myPerson.firstname` on line 14, we access `self.firstname` from within the instance. Notice that this is similar to calling something like `myString.isupper()` to check if a string is uppercase: `firstname` and `isupper()` are contained *within* the instance, and calling them gives an answer specific to *that* instance. The difference is that `firstname` is a variable while `isupper()` is a method, but we'll talk about that in detail later.

Once we've created an instance of a class, we can also modify its variables just as if they were variables in our own program, as shown in Figure 5.1.4. We can print these variables by calling `print(myPerson.firstname)`, or we can modify

#	CreatingInstances-2.py	Output
1	`#Define the class Person`	`[no first name]`
2	`class Person:`	`David`
3	`    #Create a new instance of Person`	
4	`    def __init__(self):`	
5	`        #Person's default values`	
6	`        self.firstname = "[no first name]"`	
7	`        self.lastname = "[no last name]"`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		
11	`#Create a new Person and assign it to myPerson`	
12	`myPerson = Person()`	
13	`#Print myPerson's firstname`	
14	`print(myPerson.firstname)`	
15	`#Change myPerson's firstname to David`	
16	`myPerson.firstname = "David"`	
17	`#Print myPerson's firstname`	
18	`print(myPerson.firstname)`	
19		

Figure 5.1.4

them just by reassigning `myPerson.firstname`. Here on line 16, we reassign it to "David", then print it on line 18 to see the change. What about in our more complex example, though, where we had separate classes for `Name` and `Person`?

In Figure 5.1.5, instead of accessing `firstname` directly from `myPerson` (`myPerson.firstname`), we instead access `name` from `myPerson` (`myPerson.name`), and *then* access `firstname` from `name` (`myPerson.name.firstname`). `name.firstname` means "get `name`'s `firstname`". So, `myPerson.name.firstname` "get `myPerson`'s `name`'s `firstname`." We'll stick with just one class for the most part, but know we can combine them like this as well.

#	CreatingInstances-3.py	Output
1	`#Define the class Name`	`[no first name]`
2	`class Name:`	`David`
3	`    def __init__(self):`	
4	`        self.firstname = "[no first name]"`	
5	`        self.lastname = "[no last name]"`	
6		
7	`#Define the class Person`	
8	`class Person:`	
9	`    def __init__(self):`	
10	`        self.name = Name()`	
11	`        self.eyecolor = "[no eye color]"`	
12	`        self.age = -1`	
13		
14	`#Create a new Person and assign it to myPerson`	
15	`myPerson = Person()`	
16	`#Print myPerson's name's firstname`	
17	`print(myPerson.name.firstname)`	
18	`#Change myPerson's name's firstname to David`	
19	`myPerson.name.firstname = "David"`	
20	`#Print myPerson's name's firstname`	
21	`print(myPerson.name.firstname)`	

Figure 5.1.5

The usefulness of objects is that we can create multiple instances, each with their own values. For example, we could create mutliple variables to represent multiple people, and give each person a unique name, as seen in Figure 5.1.6.

#	CreatingInstances-4.py	Output
1	`#Define the class Person`	`myPerson1: David`
2	`class Person:`	`myPerson2: Vrushali`
3	`    #Create a new instance of Person`	
4	`    def __init__(self):`	
5	`        #Person's default values`	
6	`        self.firstname = "[no first name]"`	
7	`        self.lastname = "[no last name]"`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		
11	`#Create two new Persons and assign them to`	
12	`#myPerson1 and myPerson2`	
13	`myPerson1 = Person()`	
14	`myPerson2 = Person()`	
15	`myPerson1.firstname = "David"`	
16	`myPerson2.firstname = "Vrushali"`	
17		
18	`print("myPerson1: " + myPerson1.firstname)`	
19	`print("myPerson2: " + myPerson2.firstname)`	
20		

Figure 5.1.6

In Figure 5.1.6, we define two instances of the `Person` class, `myPerson1` and `myPerson2`. Changing the first name of `myPerson1` doesn't impact `myPerson2`, as confirmed by the `print()` statements on lines 18 and 19. We could thus create lists of lots of instances of `Person` to represent class rosters or directories, and modifying one instance wouldn't affect any of the others.

Objects vs. Dictionaries

Recall that at the end of the Chapter 4.5 on dictionaries, we briefly discussed using dictionaries to create object-like structures. Specifically, we said that if we used the same keys across multiple dictionaries, we were effectively creating objects. Check out the similarity in Figure 5.1.7.

ObjectsvsDictionaries.py

```
1  #Define the class Name
2  class Name:
3      def __init__(self):
4          self.firstname = "[no first name]"
5          self.lastname = "[no last name]"
6
7  #Define dictionaries with keys firstname and lastname
8  myNameDict = {"firstname" : "David", "lastname" : "Joyner"}
9
10 #Define instances of Name
11 myNameInst = Name()
12 myNameInst.firstname = "David"
13 myNameInst.lastname = "Joyner"
14
15 print("Dictionary: " + myNameDict["firstname"])
16 print("Instance: " + myNameInst.firstname)
17
```

Output

```
Dictionary:
David
Instance:
David
```

Figure 5.1.7

In each case, we define an instance (of type dictionary on line 8 and of type `Person` on lines 11 to 13) that represents David Joyner. For the dictionary, it's with the key `firstname` and the value "David", and for the class it's with the class variable `firstname` and the value "David". So what's the benefit of using class instead of dictionaries? First, by defining classes, we *guarantee* (rather than *assume*) the keys are what we expect them to be. Second, classes can have methods as well as variables, while dictionaries have only variables; in other words, dictionaries only store data, whereas classes can also act on data.

In Figure 5.1.7, it would seem that the benefit of dictionaries is that we can define everything in one line, but there's a way we can do that with classes, too, which we'll cover next.

3. Encapsulating Methods in Classes

Part of the power of classes and instances is that they let us create data types with logical combinations of variables. We could create a `Person` class with a person's name, eye color, age, address, and telephone number. We could create a `Chair` class with a chair's color, material, number of legs, and style. We could create a `Student` class with a student's name, student ID number, and a list of their course enrollments. These course enrollments could themselves be instances of a `CourseData` class, which would have a list of the student's grades in that class. We can use objects to create complicated schemes of data that are still relatively easy to use because they're organized logically.

That's only half the power of object-oriented programming, though. The other half is that classes can contain methods—their own dedicated functions—as well as variables.

Encapsulation
The ability to combine variables and methods into class definitions in object-oriented programming. It helps avoid modification or misuse of data by other functions or programs.

Encapsulating Methods

Encapsulation is the principle of object-oriented program that describes organizing variables and methods together into custom structures. I've saved defining the term until here because while it applies just to variables as well, methods are what make encapsulation truly powerful.

A method is a function defined inside of a class. It has all the same properties: a name, a list of parameters, some code, and optionally a return statement. The scope inside the method is defined as the normal scope of a function for that language (typically the method's parameters and any variables it defines in its own code), *plus* any variables that are visible in the instance of the class as a whole (accessed via that `self` variable). So, in the case of a `Person` class with class variables `first-name` and `lastname`, the class might have a `getFullName()` method that would return the first and last name together. Because the first and last name exist inside the instance, the method `getFullName()` could see them and manipulate them before returning the name.

Common Method Types: Constructors and Destructors

Methods can be used for anything we use functions to do. For example, if we had a class representing a person's bank account called `BankAccount`, it might have a variable to represent the current balance, and methods to check the balance and attempt a transaction.

There are four common types of methods, however, that tend to come up a lot in the implementation of classes. The first two are **constructors** and **destructors**. Oftentimes, these both have special syntax in a language to set them apart so that the computer can recognize them. A constructor contains code that will run every time a new instance is created; it's kind of like a "setup" method. For example, if a class has a list, the list needs to be initialized before it can actually be used; this is done in the constructor because it guarantees it is performed before it's needed. Or, in our `BankAccount` example, we would typically assume every newly-created bank account has no balance, so in the constructor we would initially set the balance to 0.

Since constructors are methods, they can take arguments for parameters as well. In other words, when someone is first creating an instance of a `Person` class, they could supply the first name and last name as arguments directly to the constructor. The constructor, then, would initialize the instance with these initial values. So, a constructor is a method that is automatically run whenever you create a new instance of a class; any code you want to run before the instance can be used should be placed in the constructor.

A destructor, on the other hand, is a method that deletes the instance. This is most pertinent in languages where the programmer is asked to do a lot of manual memory management, and for the most part these principles are outside the scope of this material. Generally, though, destructors can be useful if you're dealing with massive quantities of data and find yourself running out of memory: you can free some up by destroying instances when you're done with them.

Constructor
A common type of method in writing classes that specifies some code to run whenever a new instance of the class is created. The constructor often has parameters that provide values to initialize the variables defined by the class.

Destructor
A common type of method in writing classes that specifies how the instance of a class is to be destroyed, such as releasing its memory back to the computer.

Common Method Types: Getters and Setters

Getters and **setters** are simple method structures that allow code to interact with the variables inside an instance. A getter simply returns the value of a certain variable, and a setter changes the value of a variable.

Why do we need to do these through methods, though? Can't we just access these variables directly? Many times, it's best to design our code such that the variables inside it cannot be modified directly. For example, imagine again that we're writing a class to represent a `BankAccount`. Imagine this class is going to be accessed by different banks and customers. We don't want them all to just be able to change the value of `myBalance` at will: they should only be able to change it in certain ways, like in a transaction. Methods like getters and setters let us dictate the rules under which class variables can be accessed and changed.

Even if we're writing a class where we *don't* mind if the variables are accessed and changed haphazardly, we still might want to know about these changes. Because getters and setters are methods, we can build whatever code into them that we want. Imagine that we want to build a log every time someone accessed or changed a

Getter
A common type of method in writing classes that returns the value of a variable contained within the class. They are commonly used to allow other processing to occur whenever the variable is accessed, like logging.

Setter
A common type of method in writing classes that sets a variable contained within the class to a new value. They are commonly used to allow other processing to occur whenever the variable is changed, like logging.

certain piece of data. By using getters and setters, we allow ourselves to run some logging code every time these methods are run. If other code was accessing the variables directly, we wouldn't be able to log when the data was accessed.

Some languages go so far as to dictate the use of getters and setters. Some languages have a concept of "privacy" in the variables and methods they encapsulate: they can determine whether a variable or method is public, where it can be accessed from outside the instance, or private, where it can only be accessed by methods within the class. So, in our `BankAccount` example, we might state that `myBalance` is a private variable, meaning it can only be accessed by methods inside the class. `getAccountBalance()`, on the other hand, would be a public method that returns `myBalance`. In that way, `myBalance` is still accessible, but it can't be modified, and `BankAccount` can log every time it is accessed.

4. Encapsulating Methods in Python

In many programming languages, class variables are defined outside of any particular method. Python, interestingly, doesn't support that: if you define a variable outside of a method, then that variable exists once and has the same value for every single instance of that class (which is called a "static" variable). Sometimes that's useful, but we won't really talk about these times here; instead, we'll focus on how to use class variables and methods the more typical way.

Constructors in Python

We've already seen a constructor in Python, and in order to define class variables in Python, we *have* to define them inside a method, preceded by that `self` variable. So, recall the code shown again in Figure 5.1.8.

On line 4, `__init__` is Python's convention for identifying constructors. Whenever a new instance of a class is created, Python goes and searches for the class's `__init__` method and runs it if it exists. If it doesn't exist, that's alright; Python just creates the instance without running any initial code. Here, running this constructor creates the variables `firstname`, `lastname`, `eyecolor`, and `age` to be seen later.

#	DeclaringaClass-1.py	Output
1	`#Define the class Person`	
2	`class Person:`	
3	`    #Create a new instance of Person`	
4	`    def __init__(self):`	
5	`        #Person's default values`	
6	`        self.firstname = "[no first name]"`	
7	`        self.lastname = "[no last name]"`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		

Figure 5.1.8

Notice, however, that `__init__` is a method, and that it has a parameter list given in parentheses on line 4. Because it's a method in a class, its first parameter must always be `self`; this is what lets the method see the variables defined for the instance. After that, however, we can define parameters as normal. If we wanted to be able to create a new person and define that person's first name and last name from the start; we could write the code shown in Figure 5.1.9.

When `Person("David", "Joyner")` is run on line 11, Python automatically goes looking for `__init_` in the `Person` class. It finds it, and pairs the first argument "David" with the first parameter `firstname`, and the second argument "Joyner" with the second parameter `lastname`. It then creates its own class variable

#	ConstructorsinPython-1.py	Output
1	`#Define the class Person`	David
2	`class Person:`	Joyner
3	`    #Create a new instance of Person`	
4	`    def __init__(self, firstname, lastname):`	
5	`        self.firstname = firstname`	
6	`        self.lastname = lastname`	
7	`        self.eyecolor = "[no eye color]"`	
8	`        self.age = -1`	
9		
10	`#Creates a person with names David and Joyner`	
11	`myPerson = Person("David", "Joyner")`	
12	`print(myPerson.firstname)`	
13	`print(myPerson.lastname)`	
14		

Figure 5.1.9

`self.firstname` on line 5 and sets it equal to `firstname` from the parameter list, and then does the same for `lastname` on line 6. Note here that it can tell the difference between `self.firstname` and `firstname`. `self.firstname` tells it to go and check for a variable named `firstname` that is persistent for the instance, and if it doesn't find one, create one; `firstname` without `self` preceding it is known to only exist within the method, so it checks the variables defined in the parameter list.

After calling that constructor, the values of `firstname` and `lastname` within the instance `myPerson` are assigned to "David" and "Joyner," so when we print them on lines 12 and 13, these values still print.

Now that we've done this, though, can we still create an instance without these arguments? As shown in Figure 5.1.10, no! "Positional" arguments are mandatory in functions and methods, meaning that we must supply them to run the method. If we want to preserve the ability to skip supplying arguments, we need to make the parameters optional, as shown in Figure 5.1.11.

#	ConstructorsinPython-2.py	Output
1	`#Define the class Person`	Traceback (most recent call last):
2	`class Person:`	File "...", line 11
3	`    #Create a new instance of Person`	`    myPerson = Person()`
4	`    def __init__(self, firstname, lastname):`	TypeError: __init__() missing 2 required positional arguments: 'firstname' and 'lastname'
5	`        self.firstname = firstname`	
6	`        self.lastname = lastname`	
7	`        self.eyecolor = "[no eye color]"`	
8	`        self.age = -1`	
9		
10	`#Creates a new person`	
11	`myPerson = Person()`	
12	`print(myPerson.firstname)`	
13	`print(myPerson.lastname)`	
14		

Figure 5.1.10

#	ConstructorsinPython-3.py	Output
1	`#Define the class Person`	[no first name]
2	`class Person:`	David
3	`    #Create a new instance of Person`	Vrushali
4	`    def __init__(self, firstname="[no first name]",`	
5	`                 lastname="[no last name"):`	
6	`        self.firstname = firstname`	
7	`        self.lastname = lastname`	
8	`        self.eyecolor = "[no eye color]"`	
9	`        self.age = -1`	
10		
11	`myPerson1 = Person()`	
12	`print(myPerson1.firstname)`	
13	`myPerson2 = Person(firstname = "David")`	
14	`print(myPerson2.firstname)`	
15	`myPerson3 = Person("Vrushali")`	
16	`print(myPerson3.firstname)`	
17		

Figure 5.1.11

In Figure 5.1.11, we've defined the parameters as optional by giving them default values in the parameter list. If a given argument isn't supplied, the code assumes it should use the value from the parameter list (such as "[no first name]" for `firstname` on line 4). So, the `Person` instance on line 11 supplies no arguments, and so `myPerson1`'s first name is "[no first name]". The second `Person` instance, `myPerson2` on line 13, supplies `firstname = "David"` as an argument, and so the `firstname` parameter gets the value "David". The third `Person` instance, `myPerson3` on line 15, shows that if an argument is given where no positional parameter is located, the program assumes the argument is for the next parameter; so, even though `firstname =` is not included on line 15, the code nonetheless assumes "Vrushali" is the value for `firstname` since `firstname` is the first parameter in the list.

Destructors do exist in Python, but because Python does so much memory management on its own, you likely won't need to use them until you get to much more advanced programs.

Getters and Setters

Interestingly as well, Python does not provide privacy options for its variables and methods. There is no way to bindingly mark a variable or method in a Python class as private, meaning that other code can always access variables directly. By convention, we often precede variables that we don't *want* other classes or functions to access with a double underscore; however, this is only a convention, meaning that other classes or functions are still able to access the data. The double underscore simply informs them that they are not intended to access the data in this way.

Part of this is "Pythonic" style, which focuses on easy access to data. This is related to the ease of Python's lists and dictionaries; other languages supply these as more traditional classes with tougher syntax. Generally, when you're developing classes in Python, it's alright to directly access variables. That's a major taboo in some languages (like Java), but it's accepted in Python.

That said, getters and setters have other purposes as well. Recall that part of the benefit of getters and setters was that they allow us to run some code whenever a variable is accessed or modified. That might be useful in a simple logging behavior, for example. To demonstrate this, let's finally implement that `BankAccount` class we've been talking about.

First, Figure 5.1.12 shows the class itself. Notice a few things. First, notice that to create an account, we *must* have a name, but the balance is optional, as shown on lines 4 (defining the class) and 22 (creating an instance). If no balance is

GettersandSetters.py

```
1  #Define class BankAccount
2  class BankAccount:
3      #Initialize balance to 0
4      def __init__(self, name, balance = 0.0):
5          self.log("Account created!")
6          self.name = name
7          self.balance = balance
8
9      def getBalance(self): #Getter for balance
10         self.log("Balance checked at " + str(self.balance))
11         return self.balance
12
13     def setBalance(self, newBalance): #Setter for balance
14         self.log("Balance changed to " + str(newBalance))
15         self.balance = newBalance
16
17     def log(self, message): #Logging method
18         myLog = open("Log.txt", "a")
19         print(message, file = myLog)
20         myLog.close()
21
22 myBankAccount = BankAccount("David Joyner")
23 myBankAccount.setBalance(20.0)
24 print(myBankAccount.getBalance())
25
```

Output

```
20.0
```

Log.txt

```
Account created!
Balance changed to 20.0
Balance checked at 20.0
```

Figure 5.1.12

supplied, the computer assumes the balance is 0.0. Notice that we say 0.0 to force the computer to see this as a float, not an integer.

Second, notice we've supplied a getter and setter on lines 9 through 15. Within each, we have the obvious lines `return self.balance` and `self.balance = newBalance`, which get and set `balance` respectively. Notice, however, that we precede these with a call to `log()` on lines 10 and 14 with a message. This is why getters and setters can still be valuable: they allow us to trace or log program execution.

Third, notice that when we're calling `log()`, we still precede it with `self.`. `self` is still how we allow different parts of an instance to see other parts. To let the `setBalance()` method see the `log()` method, it needs to call `self.log()` to basically say, "my `log()` method". Fourth, notice that we open our file in "append" mode inside `log()` on line 18. This allows us to build a log over time in the file without storing the log in a variable within our program.

Down in the main code, we first create a bank account on line 22. We supply a name to the constructor, but no balance, so the balance is assumed to be 0. We then set the balance to 20.0 on line 23; we could have set it in the constructor, of course, but we're demonstrating the setter and getter. Then, we print the balance on line 24, seeing that it has been correctly set to 20.0. If we opened Log.txt (as shown in the bottom right), we would see three lines:

- Account created!
- Balance changed to 20.0
- Balance checked at 20.0

The constructor, getter, and setter all write to Log.txt.

Encapsulating Other Functions

Note that constructors, destructors, getters, and setters are four common paradigms for designing methods, but that certainly is not an exhaustive list of every type of method. We can create methods to do whatever we want. For example, in the `BankAccount` code, we likely don't want a `setBalance()` method because we rarely say, "Regardless of the prior value of this account, set its value to this new number." Instead, we say "deposit $20" or "withdraw $10". So, these are the methods we would likely want to create.

In Figure 5.1.13, instead of just changing the balance manually as in Figure 5.1.12, we add or subtract to or from it with the `deposit()` and `withdraw()` methods. Of course, we could have done this anyway with a call like

EncapsulatingOtherFunctions.py

```
1  class BankAccount:
2      def __init__(self, name, balance = 0.0):
3          self.log("Account created!")
4          self.name = name
5          self.balance = balance
6
7      def getBalance(self):
8          self.log("Balance checked at " + str(self.balance))
9          return self.balance
10
11     def deposit(self, amount):
12         self.balance += amount
13         self.log("+" + str(amount) + ": " + str(self.balance))
14
15     def withdraw(self, amount):
16         self.balance -= amount
17         self.log("-" + str(amount) + ": " + str(self.balance))
18
19     def log(self, message): ...
...
24 myBankAccount = BankAccount("David Joyner")
25 myBankAccount.deposit(20.0)
26 print(myBankAccount.getBalance())
27 myBankAccount.withdraw(10.0)
28 print(myBankAccount.getBalance())
```

Output

```
20.0
10.0
```

Log.txt

```
Account created!
+20.0: 20.0
Balance checked at 20.0
-10.0: 10.0
Balance checked at 10.0
```

Figure 5.1.13

`myBankAccount.setBalance(myBankAccount.getBalance() + 20.0)`, but we want to write methods that are as easy to call as possible. If we know we'll regularly be withdrawing from and depositing to the account, it's better to have methods to take care of that.

5. Advanced Topics in Classes in Python

We've talked a good bit so far about references and mutability in Python. How do these concepts play along with classes? The answer: they pretty much follow the same conventions. Immutable types are still immutable, but most types are still mutable. This can get a little tricky when we start dealing with combinations of the two, though.

Combining Classes

Let's explore this just by trying out some different combinations of things. For this running example, let's use our two classes from before, `Person` and `Name`. Let's also keep things simple by accessing the variables directly instead of using getters and setters. First, let's see how we can combine them in interesting ways.

In Figure 5.1.14, notice that `Person` has a `name`, `eyecolor`, and `age`. There's nothing in `Person` that dictates that the name must be of type `Name`, but that's the value we're supplying it on line 16. Had we supplied the string "David Joyner" directly, `myPerson` would have been created just fine, but the code would crash on line 17 when we tried to access the variable `firstname`, which exists in `Name` but not in string.

#	CombiningClasses.py	Output
1	`#Defines the class Person`	David
2	`class Person:`	Joyner
3	`    def __init__(self, name, eyecolor, age):`	brown
4	`        self.name = name`	30
5	`        self.eyecolor = eyecolor`	
6	`        self.age = age`	
7		
8	`#Defines the class Name`	
9	`class Name:`	
10	`    def __init__(self, firstname, lastname):`	
11	`        self.firstname = firstname`	
12	`        self.lastname = lastname`	
13		
14	`#Creates a person with eyecolor "brown", age 30, and`	
15	`#a name with firstname "David", lastname "Joyner",`	
16	`myPerson = Person(Name("David", "Joyner"), "brown", 30)`	
17	`print(myPerson.name.firstname)`	
18	`print(myPerson.name.lastname)`	
19	`print(myPerson.eyecolor)`	
20	`print(myPerson.age)`	
21		

Figure 5.1.14

Second, notice that we're initializing the argument for name *while* we're initializing `myPerson`, all on line 16. Calling `Name("David", "Joyner")` returns an instance of `Name` with `firstname` "David" and `lastname` "Joyner", which is the argument we want to pass into the constructor for `Person`. So, we can initialize `Name` right there within the constructor for `Person`. We also could have separately called `myName = Name("David", "Joyner")` and used `myName` as the argument, but since we never need `myName` on its own, we might as well create it inside `Person`'s constructor on line 16.

The result is an instance of `Person` called `myPerson`, which has a string for `eyecolor` ("brown"), an integer for `age` (30), and an instance of `Name` for `name`. That instance of `Name` has strings for its `firstname` ("David") and `lastname` ("Joyner").

Instance Assignments

So, using the instance of `Person` from Figure 5.1.16, let's see what happens if we assign it to another instance. What do you think happens with the code in Figure 5.1.15?

InstanceAssignments.py

```
1  class Person:
2      def __init__(self, name, eyecolor, age):
3          self.name = name
4          self.eyecolor = eyecolor
5          self.age = age
6
7  class Name:
8      def __init__(self, firstname, lastname):
9          self.firstname = firstname
10         self.lastname = lastname
11
12 myPerson1 = Person(Name("David", "Joyner"), "brown", 30)
13 myPerson2 = myPerson1
14 myPerson2.eyecolor = "blue"
15 print("myPerson1's eyecolor: " + myPerson1.eyecolor)
16 print("myPerson2's eyecolor: " + myPerson2.eyecolor)
17
```

Output

```
myPerson1's
eyecolor: blue
myPerson2's
eyecolor: blue
```

Figure 5.1.15

We create an instance of `Person` just like before on line 12. We then create a second `Person` instance, `myPerson2`, and set it equal to `myPerson1` on line 13. We then modify `myPerson2` on line 14. Does `myPerson1` also change? The output of line 15 shows it does! An instance of the `Person` class is mutable, so when we say `myPerson2 = myPerson1`, we're really just telling them to look at the same data in memory. So, if `myPerson2` changes something (like `eyecolor`), it's changing it in the same place in memory that `myPerson1` refers to. So, it changes for both.

Instances as Arguments

What happens if we pass an instance into a function? Let's imagine we have a function called `capitalizeName()` on lines 12 through 14, which converts an instance of `Name` to all caps, as shown in Figure 5.1.16.

`capitalizeName(myPerson.name)` passes in the instance of the `Name` object into `capitalizeName`. `Name` is mutable, meaning that `capitalizeName()` is working off the same copy of `myPerson.name`. So, when the name's

InstancesasArguments-1.py

```
1  class Person:
2      def __init__(self, name, eyecolor, age):
3          self.name = name
4          self.eyecolor = eyecolor
5          self.age = age
6
7  class Name:
8      def __init__(self, firstname, lastname):
9          self.firstname = firstname
10         self.lastname = lastname
11
12 def capitalizeName(name):
13     name.firstname = name.firstname.upper()
14     name.lastname = name.lastname.upper()
15
16 myPerson = Person(Name("David", "Joyner"), "brown", 30)
17 capitalizeName(myPerson.name)
18 print(myPerson.name.firstname)
19 print(myPerson.name.lastname)
20
```

Output

```
DAVID
JOYNER
```

Figure 5.1.16

capitalization changes, it *does* change it for the original copy, as seen by the `print()` statements in lines 18 and 19.

What if, though, instead of `capitalizeName()`, we had `capitalizeString()`, and called it separately on `firstname` and `lastname`? This is shown in Figure 5.1.17, and the new method is on lines 12 and 13.

#	InstancesasArguments-2.py	Output
1	`class Person:`	David
2	`    def __init__(self, name, eyecolor, age):`	Joyner
3	`        self.name = name`	
4	`        self.eyecolor = eyecolor`	
5	`        self.age = age`	
6		
7	`class Name:`	
8	`    def __init__(self, firstname, lastname):`	
9	`        self.firstname = firstname`	
10	`        self.lastname = lastname`	
11		
12	`def capitalizeString(instring):`	
13	`    instring = instring.upper()`	
14		
15	`myPerson = Person(Name("David", "Joyner"), "brown", 30)`	
16	`capitalizeString(myPerson.name.firstname)`	
17	`capitalizeString(myPerson.name.lastname)`	
18	`print(myPerson.name.firstname)`	
19	`print(myPerson.name.lastname)`	
20		

Figure 5.1.17

Here, the `firstname` and `lastname` are *not* capitalized when printed on lines 18 and 19. Why? Because strings themselves are immutable, and the assignment operation within `capitalizeString()` operates on a string. So, `capitalizeString()` changes what its local copy of `instring` points at, but that local copy isn't the same as `firstname` or `lastname` in this instance of `Name`.

So, any operations we make on mutable data types propagate out of the function; any operations we make on immutable types do not. If we pass an instance of a class into a function or method, the variables of the class could be changed; if we pass only the immutable variables themselves, then the variables of the class cannot be changed.

Making Actual Copies

So what do you do if you want to make an *actual* copy of an instance, such that you can modify it separately? Although there are more efficient ways, the basic principle is that you must copy at the level of the immutable data types. In other words, setting `myPerson2` equal to `myPerson1` didn't work because it just made the two variables point at the same values. To actually copy the values, we need to do exactly that: copy the values.

Figure 5.1.18 is a good demonstration of copying the values. Instead of just setting `myPerson2` equal to `myPerson1`, we create a *new* instance of `Person`, using the *values* of `myPerson1` as the arguments on line 13. So, we tell `myPerson2` to point to a new instance of `Person` because we call `Person`'s constructor, and we populate `name`, `eyecolor`, and `age` with the same values as `myPerson1`. This creates a new instance in memory, as opposed to just another variable that points to the same instance in memory.

Since `myPerson2` has its *own* variables for `eyecolor`, then when we reassign it on line 14, it doesn't affect the `eyecolor` for `myPerson1`. `eyecolor` is a string, which is immutable, so when we reassign it, it doesn't change the value in memory: it instead just points the variable `myPerson2.eyecolor` at a new value. This doesn't change the value that `myPerson1.eyecolor` points at.

```
# MakingActualCopies-1.py
1  class Person:
2      def __init__(self, name, eyecolor, age):
3          self.name = name
4          self.eyecolor = eyecolor
5          self.age = age
6
7  class Name:
8      def __init__(self, firstname, lastname):
9          self.firstname = firstname
10         self.lastname = lastname
11
12 myPerson1 = Person(Name("David", "Joyner"), "brown", 30)
13 myPerson2 = Person(myPerson1.name, myPerson1.eyecolor, myPerson1.age)
14 myPerson2.eyecolor = "blue"
15 print(myPerson1.eyecolor)
16 print(myPerson2.eyecolor)
17 myPerson2.name.firstname = "Vrushali"
18 print(myPerson1.name.firstname)
19 print(myPerson2.name.firstname)
Output
brown
blue
Vrushali
Vrushali
```

Figure 5.1.18

Notice, though, that the same doesn't apply to `name` as written here. We passed `myPerson1.name` as the argument when constructing `myPerson2`, but `myPerson1.name` is an instance of the `Name` class. An instance of the `Name` class is mutable. That means that even though `myPerson1` and `myPerson2` have their own variables called `name`, they're pointing at the same value in memory. So, when we modify what `myPerson2.name.firstname` points at on line 17, it *also* modifies what `myPerson1.name.firstname` points at, as shown by the output of lines 18 and 19. To make a true copy, we need to actually construct a new instance of `Name` as well, as shown in Figure 5.1.19.

```
# MakingActualCopies-2.py
1  class Person:
2      def __init__(self, name, eyecolor, age):
3          self.name = name
4          self.eyecolor = eyecolor
5          self.age = age
6
7  class Name:
8      def __init__(self, firstname, lastname):
9          self.firstname = firstname
10         self.lastname = lastname
11
12 myPerson1 = Person(Name("David", "Joyner"), "brown", 30)
13 myPerson2 = Person(Name(myPerson1.name.firstname, myPerson1.name.lastname),
14                      myPerson1.eyecolor, myPerson1.age)
15 myPerson2.eyecolor = "blue"
16 print(myPerson1.eyecolor)
17 print(myPerson2.eyecolor)
18 myPerson2.name.firstname = "Vrushali"
19 print(myPerson1.name.firstname)
20 print(myPerson2.name.firstname)
Output
brown
blue
David
Vrushali
```

Figure 5.1.19

Instead of just passing `myPerson1.name` as the argument to the constructor creating `myPerson2` on line 13, we instead are calling the constructor of `Name` as well to create a new instance of `Name`, too. Into that constructor, we pass `myPerson1.name.firstname` and `myPerson1.name.lastname`, which are strings and thus immutable. Then, when we modify `myPerson2`'s `name`, it does not affect `myPerson1`'s `name`, as shown by the output of lines 19 and 20.

6. Polymorphism and Inheritance and Abstraction, Oh My!

We've barely scratched the surface of what objects can do. There are many advanced concepts in designing classes, like inheritance and abstraction. We're not going to get into programming these concepts in our material, but we want to quickly preview it so that when you see it later in your computing education, they'll sound familiar. Typically, there are entire classes on object-oriented programming, and these concepts are a big part of that additional depth.

Abstraction

Let's take a quick example. What is this a picture of? Most people would probably say: a chair, and of course, they'd be right. Some people, though, might say it's a dining room chair. They'd also be right. Some people might say it's a Harvest-style dining room chair. They'd also be right. On the other end of the spectrum, some people might say it's a piece of furniture. They'd also be right. Some people might say it's a home good. They'd also be right. And some people might say, well, it's a thing. They'd also be right.

Abstraction
A principle of object-oriented programming that states that only essential information should be made visible to the outside program.

What this object "is" exists at different levels of **abstraction**, and we need different levels for different purposes. For example, it wouldn't make much sense to ask: how many legs does a home good have? But it does make sense to ask: how many legs does a dining room chair have? Certain variables only make sense at a certain level of abstraction.

Or, to take another example, imagine you want to buy a Harvest-style dining room chair. Do you go to the Harvest-style dining room chair store? Probably not. You don't even go to the chair store; you go to the furniture store. You know that's the level of abstraction at which chairs are sold: they're sold as part of general furniture sales. Yet, you rarely go shopping for furniture: you don't go to the store thinking, "I don't know if I want a chair or a table." You probably know what you want. This is an example of different levels of abstraction at work.

Polymorphism

Polymorphism
The principle that a method call can behave differently depending on the arguments and object with which it is called.

Polymorphism is a characteristic of this idea of abstraction that was implicit in that example. Polymorphism is the ability to ask the same questions—or in software, run the same code—on a wide variety of different types of concepts. In the above example, we would likely say that chairs, tables, and beds are three different types of objects, and yet we can ask certain questions across them, like their price and their material. We can imagine a method that returns a material, and we can imagine that method being applicable to any kind of furniture. Then, we can imagine iterating over a catalog of furniture and asking, for each item, "What material is this?" Even though the data are different types of furniture, that one single question makes sense.

One of the most common places this happens in software is with what's often called a `toString()` method. When we're designing classes, one most common desires is to be able to print the object. What it means to "print" the object, however, is very different from class to class: To print a `Person` object, we might want to print the name, but to print a `BankAccount` object, we might want to print an account number. The general idea—printing an object—is the same, though. Polymorphism describes to the ability to write a method in each class that would allow drastically dissimilar objects to be accessed the same way.

Inheritance

Inheritance
A principle of object-oriented programming where classes can be created that are **"subclasses"** of other classes, inheriting all the variables and methods from the other class while supplying new variables, methods, or behaviors of these own.

Inheritance in abstraction is the idea that an object can "inherit" certain variables and methods from its "parent." We echoed this idea with the furniture example above. There are certain questions that make sense to ask about any piece of furniture: what color is it? What material is it made out of? How much does it weigh? These variables would exist for a `Furniture` class.

However, there are questions that only make sense to ask about certain *types* of furniture. For example: mattress size. What size of mattress does a piece of furniture take? That question only makes sense if the piece of furniture is a bed. So, we would say that "mattress size" is a variable of beds, not of all pieces of furniture. However, we would say that color, material, and weight are variables of all furniture. Or, for another example, think of chairs. Color, material, and weight make sense for any piece of furniture, chairs included. A variable representing whether there are arm rests *only* makes sense for chairs, though. A variable representing how many wheels only makes sense for office chairs. So, office chairs would have a "number of wheels" variable, and would inherit "has armrests" from chair, which itself would inherit "material", "color", and "weight" from furniture.

These three concepts, combined with best practices for defining classes, visualizations of class relationships, and constructing complex programs using objects together are effectively what you would cover next in a class on object-oriented programming.

CHAPTER

5.2

Algorithms

Lesson Learning Objectives

By the end of this chapter, students will be able to:

- Describe the purpose of algorithms and use Big O notation for measuring complexity of an algorithm;
- Examine the working of recursion as well as sorting and search algorithms.

1. What Are Algorithms?

One of the biggest strengths of computing is its ability to automatically run extremely complex mathematical operations very quickly. Machine learning, for example, is effectively the math of statistics, equipped with ultra-powerful computers that can perform trillions of operations. This is the "algorithms" side of computer science.

An **algorithm** is effectively a set of rules or calculations to be followed… although that could describe anything we've done so far. Colloquially, algorithms are especially complex sequences of code that transform input into output according to some mathematically-defined requirements. They refer to segments of code where the computer's benefit is its speed more than its distributability, repeatability, or the other benefits of programming. Or, at least, that's how I differentiate algorithms from other coding.

Algorithm
Technically, a collection of steps that transforms input into output; commonly, a complex set of lots of steps that is only feasible to perform with the efficiency of a computer.

Famous Algorithms

Maybe the best way to define algorithms is by example. Here are some algorithms that affect you every day:

- Data Compression. Data compression is about taking very large files and communicating them with a fraction of the data, in a way that loses as little actual meaning as possible. Algorithms responsible for effective compression are used to run video streaming services, online multiplayer games, and more.
- Random Number Generation. Surprisingly, your computer is incapable of doing anything truly random: everything about it is deterministic. Random number generators are algorithms that take some variable input and use it to algorithmically replicate something resembling truly random numbers.
- Search Algorithms. Not search in the way we'll discuss later in this chapter, but rather Internet search engines. Algorithms underlie Google's efforts to predict what you're searching for based on your keywords, relationships between websites, and past users' experiences.

Effectively all of modern computing infrastructure is built on algorithms in some way, from ensuring the accuracy of data that you download to predicting rates of certain diseases based on aggregated medical data. Algorithms are everywhere.

Algorithms and Programming Languages

What is particularly interesting about algorithms is that they generally exist separate and apart from a particular computer implementation of them. That's true for anything we've covered so far in some sense, but in most cases, the underlying concepts were not interesting or complex without the implementation. With algorithms, defining the algorithm separately from the language we might use to implement it is still a productive goal.

For that reason—and because this is a "looking forward" chapter—we're not going to get into the language-specific implementations of these algorithms. We're going to stick to describing algorithms in more general terms. Every algorithm we discuss could be programmed in Python, Java, C++, C#, Visual Basic, Swift, Ruby, or whatever other programming language you choose to use.

Complexity
The rate at which the number of operations requires to run an algorithm grows based on the value of the input on which it operates.

2. Complexity and Big O Notation

In my definition of algorithms, I noted that one of the key features was **complexity**. Computing brings lots of benefits, from easy distribution of software to building dynamic multimedia experience. For algorithms, though, the most important benefit is the ability to perform complex operations automatically. That lets us chain together lots of steps to solve problems and generate results. So, a big part of what makes algorithms interesting is their complexity.

Complexity in Algorithms

Why is dealing with complexity one of the big benefits of computing? Let's take an example. Imagine you have a roster of 500 students in a school. Your goal is to find David Joyner on that roster. How long does it take? Let's measure time in terms of number of comparisons: you're comparing each name to "David Joyner" to see if it matches. How many comparisons does it take?

If the roster is sorted alphabetically, then it likely doesn't take long: you start near the middle, check if the first name you see is before or after "David Joyner", and then start looking in the right direction. If it's perfectly sorted, it takes fewer than 10 comparisons if you use the most efficient algorithm. If it's not sorted, though, it might require up to 500 comparisons because you have to go one name at a time through the entire list. Usually when discussing complexity, we talk in terms of the worst-case scenario, also known as the "upper bound" on how long something could take.

10 vs. 500 is obviously a huge difference. Imagine, however, that instead of finding "David Joyner," we're trying to check the roster as a whole for duplicates. If they're sorted, things become easier: duplicates would be side-by-side. Our best-case and worst-case are the same, 500 comparisons, because we keep going until we've checked every individual item to see if it duplicates the previous item.

However, what if the roster isn't sorted? If the roster isn't sorted, then we start with the first name, and compare it to the next 499 names. Then, we start with the next name, and compare it to the next 498 names. If we do the math, we get almost 125,000 calculations, a nearly impossible number for a person to do: even if they were able to do two calculations a second, it would take over 17 hours.

The reason complexity is so important is that with inefficient algorithms, expanding data set sizes can have massive impacts on execution time. It would take 125,000 operations to perform these comparisons on a roster of 500, but what if the roster grew to 600? That number balloons to 180,000 operations. With 1000 students, it would reach a half-million. Doubling the number of students quadruples the number of operations required.

Big O Notation

Big O Notation
A notation for expressing the worst-case efficiency of an algorithm in terms of the size of the input.

This brings us to a notion called **Big O Notation**. Big O is a measure of algorithmic complexity. In Big O, we ask: given a data set of size n, how many operations are required to complete this algorithm? In the example above, the answer would be $O\left(\frac{n^2}{2}\right)$. If we go through the data set and compare all 500 names each to all 500 other names, our complexity would be n^2 (500×500), but we would also be performing each comparison twice; so, $O\left(\frac{n^2}{2}\right)$ is the total efficiency. However, what we're really interested in with Big O notation are differences of orders of magnitude, so it's common to leave out the coefficients and describe this algorithm simply as having $O(n^2)$ complexity. In fact, the O in Big O Notation stands for the "order" of the function. It's not interested in cutting complexity in half or doubling it; it's interested in squaring it or taking the square root of it.

Common Big O Values

Because of this, there are certain values that are pretty common for Big O notation. The first would be $\boldsymbol{O(1)}$. This stands for a constant order: the same number of operations will be performed no matter the size of the data set. For example, checking whether or not the first student in the roster has a last name that starts with "A" has complexity $O(1)$. It doesn't matter if there are 10 values in the data set or 10,000; it still only takes a constant number of operations to check. In this case, that number is 1, but it doesn't have to be 1; just any constant number regardless of the size of the data set.

Constant Order, $O(1)$
The same number of operations are required regardless of the size of the data set.

Another common complexity is simply $\boldsymbol{O(n)}$, linear order. This means that the number of operations required varies linearly with the number of items in the data set. It doesn't have to be exactly the number of items, but that on average as the data set size increases, the number of operations required grows at the same rate. Checking for duplicates in our sorted roster is an example of this: We always perform one fewer operation than the size of the roster because we compare each name to the next one.

Linear Order, $O(n)$
The number of operations required increases linearly with the size of the data set.

The complexity for checking for duplicates in an unordered set had the complexity $\boldsymbol{O(n^2)}$, quadratic order. This occurs whenever we have to perform a number of operations for each pair of items in the dataset. When checking for duplicates in an unsorted roster, adding one name to the roster means adding n new comparisons because the new name must be checked against every existing name.

Quadratic Order, $O(n^2)$
The number of operations required increases with the size of the data set squared.

Some algorithms can grow in complexity even faster, and require $\boldsymbol{O(n^3)}$ or even higher exponents. This would be called polynomial order. The simplest example of this here would be checking for triple-duplicates in an inefficient manner. Imagine checking every set of three students to see if all three students had the same name; that would have a cubic order because every student must be compared to every pair among the other students. Of course, in practice we would only perform these final comparisons if that initial pairing of students was equal, so the efficiency would be closer to $O(n^2)$, although the worst case would remain $O(n^3)$.

Polynomial Order, $O(n^3)$
The number of operations required increases with the size of the data set raised to a larger exponent.

But that's not even the worst of it! There can exist values such as $\boldsymbol{O(2^n)}$, where the size of the data set *is* the exponent. This is called exponential order. It's tough to even conceptualize of an algorithm with that level of complexity, but brute-force password-hacking is one example. To try every possible numeric password, you need to try 10^n passwords, where n is the length of the password. For a 10-character password of just numbers, that's 10 billion combinations (10^{10}); for just letters, it's 141 trillion (26^{10}).

Exponential Order, $O(2^n)$
The number of operations required increases by a constant raised to the size of the data set.

All of these examples have focused on increasing complexity. The goal in algorithm design is really to decrease complexity, though. The standard example of that is $\boldsymbol{O(\log n)}$, a logarithmic order. With that complexity, the runtime grows slower and slower over time. log(100) is 2, but log(10,000) is just 4; our dataset size increased 100x, but our runtime merely doubled. As we'll see later, a binary search has a complexity of $O(\log n)$: the runtime grows very slowly relative to the growth of the data set.

Logarithmic Order, $O(\log n)$
The number of operations required increases with the square root of the size of the data set.

These are just some of the common complexities we see. Other common ones include loglinear ($O(n \log n)$, equivalent to $O(\log n!)$), which tends to be the optimal efficiency for most sorting algorithms, and factorial ($O(n!)$), which tends to come up in recursive algorithms.

3. Recursion

An old joke in computing is that the definition of **recursion** is, "see recursion". To look for recursion, you must look for recursion. Really, recursion is the repeated application of a certain function. What distinguishes recursion is that the repeated application happens *within* the function. This happens when a function calls itself.

Recursion
A programming method characterized by functions that, during their operation, call additional copies of themselves; see also, recursion. Recursion involves breaking down a problem into smaller instances recursively until each of them can be independently solved. Solutions to these smaller instances combine to form the solution for the original problem

Simple Recursion: Factorial

The easiest way to demonstrate recursion is with a couple of examples. But since we're staying away from actually showing the implementation of these algorithms in this book, we'll keep them in terms of natural language—you can implement them in the language of your choice.

First, just in case you aren't familiar with it, let's define the factorial operator: the factorial of a number is the product of that number multiplied by each number between it and 1. Factorial is represented by an exclamation point. So, 4! = 4 × 3 × 2 × 1 = 24. 5! = 5 × 4 × 3 × 2 × 1 = 120. What makes factorial a good candidate for recursion is that a number's factorial is equal to itself times the factorial of one less than the number. In other words, 5! = 5 × 4!, 4! = 4 × 3!, 3! = 3 × 2!, and 2! = 2 × 1!. 1! = 1, so when we reach 1, we can just work our way back.

This is the perfect example of a recursive function: the factorial for a certain number calls the factorial for another number. So, let's translate this into a high-level algorithm:

- Function `factorial`, given integer `n`:
  - If `n` > 1, return `n` × `Factorial(n - 1)`
  - If `n` = 1, return 1

Part of the definition of the factorial function is a call to the factorial function itself in the first bullet, but with a different number. The last bullet is called an escape condition: every recursive function has a branch where it returns a value instead of calling itself again, breaking the chain of recursive calls. So, let's trace through how this calculates 5!:

1. 5! = 5 × 4!, so it calls `factorial(4)`
2. 4! = 4 × 3!, so it calls `factorial(3)`
3. 3! = 3 × 2!, so it calls `factorial(2)`
4. 2! = 2 × 1!, so it calls `factorial(1)`
5. 1! = 1, so 1 is returned to step #4
6. 2 × 1 = 2, so 2 is returned to step #3
7. 3 × 2 = 6, so 6 is returned to step #2
8. 4 × 6 = 24, so 24 is returned to step #1
9. 5 × 24 = 120, so 120 is the result of 5!

That matches the value we got by calculating 5! manually above. Note especially the symmetry involved: the 1st call was completed by the 9th line, the 2nd call by the 8th line, the 3rd call by the 7th line, the 4th call by the 6th line, and the 5th call was self-fulfilling. This is the general nature of recursive methods: additional copies of the method keep getting added on top, until eventually they start to be removed. This is why we noted that the stack was more common than the queue in computing: it's very common for function calls to work like this, where calls are added to the top until one is ready to return something, then we work back down the stack.

Of course, we could certainly implement this without recursion, such as with a `for` loop. However, there are some functions that are easier to implement recursively than other ways. For example, Fibonacci's series lends itself nicely to recursion.

Intermediate Recursion: The Fibonacci Series

The Fibonacci series is a series of numbers where each number is the sum of the two previous numbers. It starts with 1 and 1 as the first two numbers. Then, the third number is the sum of the first and second numbers: 1 + 1 = 2. The fourth number is the sum of the second and third numbers: 1 + 2 = 3. What's the 10th number? To find the 10th number, we need to know the 8th and 9th numbers; to know the 8th number, we need to know the 6th and 7th numbers; and so on. That's why Fibonacci series lends itself to a recursive implementation.

So what would that algorithm look like?

- Function `Fibonacci`, given integer `n`:
  - If `n` > 2, return `Fibonacci(n - 1)` + `Fibonacci(n - 2)`
  - If `n` < = 2, return 1

This becomes a bit hard to trace because there are two recursive calls on the first bullet per function call. However, the overall pattern follows that of the Factorial function: it keeps adding additional calls to Fibonacci, decreasing the argument by 1 each time, until it calls it for 2 or 1. At that point, it returns 1, and the results percolate all the way back up to the top result.

What's important to note about Fibonacci series as implemented here is that it's highly inefficient. Why? Because every call to Fibonacci is going to be called twice for a single parameter. `Fibonacci(5)` will call `Fibonacci(4)` and `Fibonacci(3)`, but `Fibonacci(4)` will *also* call `Fibonacci(3)`. So, we end up calculating each number twice. We could resolve this by maintaining a dictionary `calculatedNumbers` of previously-calculated Fibonacci numbers.

So is recursion only useful for these super-mathematical examples? Not at all! It has more immediate practical applications, too.

Advanced Recursion: Directory Exploration

Imagine you're trying to list the files on your computer (perhaps to then write a search function that searches for certain file names). How would we write a function that can do that, given all the different folders that need to be opened and browsed? We can do that recursively as well! Folders are made of files and folders, meaning that for every folder we encounter, we need to open every folder and list its files.

Here's what that function would look like in abstract terms:

- Function `ListFiles`, given directory `dir`:
  - List each `file` in `dir`
  - For `each folder` in `dir`, `ListFiles(folder)`

Given a directory name, that function will list all the files in the directory, then call `ListFiles()` on each folder in the directory. As a result, each folder in the directory will then get *its* files listed, followed by the folders in *those* folders, and so on.

This also gives a little glimpse at the notions of head recursion and tail recursion. Here, we list the files in the current directory before making the recursive call; as a result, this folder's files will be first, the first subfolder's files will be second, the first subfolder's first subfolder's files will be third, etc. If instead we did the listing first, we'd find the first files listed would be the deepest subfolder in the first subfolder of `dir`, and `dir`'s files would only be printed once *everything* else had been printed.

4. Sorting Algorithms

Finally, we'll close out our material with two types of general algorithms, **sorting algorithms** and search algorithms. These are often used as the first exposure to algorithms; if you go on to take an algorithms class, these might be the first things you cover, or they might assume you've already learned this.

Sorting Algorithms
Algorithms that take as input a list, and produce as output a sorted version of that list. Examples include bubble sort, insertion sort, selection sort, merge sort, shell sort, quick sort, and heap sort.

First, we'll cover sorting algorithms. A sorting algorithm is an algorithm that takes as input a list, and produces as output a sorted version of that list. What is being sorted can differ; it could be numbers, strings, dates, people (sorted by age, for example), or anything else. The only stipulation is that the algorithm must have some way to judge whether one item is greater than, less than, or equal to another.

There are lots of different sorting algorithms, with different efficiencies. We'll discuss them in order based on the difficulty to implement them, not the computational complexity. In order to demonstrate these, we'll chat about sorting a list of 10 numbers, as shown in Figure 5.2.1.

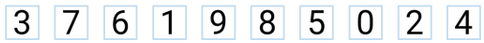


Figure 5.2.1

Bubble Sort

The easiest sort to implement is the bubble sort. In a bubble sort, we iterate through the list one pair at a time; if the pair is in the wrong order, we switch it. We repeat that process so long as a switch was needed on the previous pass; once no switches were needed, we stop.

Put in terms of our algorithm, bubble sort would look like this:

1. For each item in the list:
   a. If this item and the next item are in the wrong order, swap them
2. If any swaps were necessary, repeat step #1

You can see that if we were implementing this in code, we would need a boolean `swapOccurred` that was reset to false every time we started #1, and got set to true whenever a swap occurs.

Let's try this out. Figure 5.2.2 shows the results of bubble sort after each individual passage through the list, starting before the first passage.

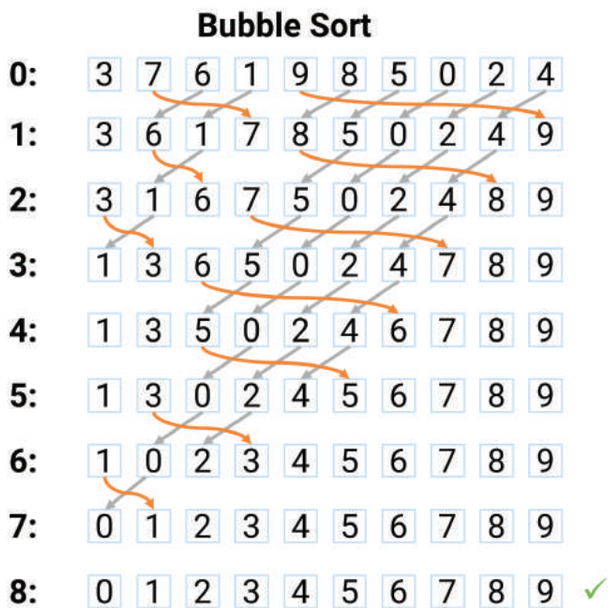


Figure 5.2.2

The first row is the list before we have attempted to sort it at all. Then, we begin the bubble sort. The bubble sort compares 3 and 7; these are in the correct order, so they aren't swapped. Then, it compares 7 and 6; these are *not* in the correct order, so they are swapped, changing the start of the list from 3 7 6 to 3 6 7. Then, it compares 7 to 1; these are in the wrong order, so they are also swapped. Then it compares 7 to 9; these are in the correct order, so they are not swapped. It starts comparing 9 to the next numbers; 9 is the largest number, so it is swapped in each subsequent comparison, moving 9 to the end. Thus, at the end of the first pass, we've moved 7 past 6 and 1, and 9 to the end.

A swap happened on the previous pass, so the algorithm runs again from the top. 3 and 6 are still in the right order, but 6 and 1 are in the wrong order, so they're swapped. 6 and 7 are in the right order, as are 7 and 8, but 8 and 5 are not. So, 8 and 5 are swapped. 8 is the next-largest number in the list, so it continues to be swapped until it stops before 9. So, we've moved the 6 to after the 1, and the 8 to the end. A swap occurred, so the algorithm runs again.

Notice that with every pass, the next-highest number lands at the end of the list; it takes 1 pass to put 9 in the right place, 2 passes to put 8 in the right place, and so on. As a result, smaller numbers tend to "float" to the front—notice also how with every row, the number 0 moves forward by one. Notice also that this took 8 runs to sort, but that's because 4 was already in the right place after run #5, and 2 was already in place after run #6; had they been out of place, we would have needed two more runs. Finally, notice that the last two rows are duplicates: the algorithm repeats until there are no swaps. There was a swap on run #7, so it has to run one more time just to confirm everything is sorted.

Bubble sort is probably the easiest sort to implement, but it's also among the least efficient. It operates in $O(n^2)$.

Selection Sort

In all likelihood, we've never actually sorted things by hand the bubble sort way. Imagine you were sorting the numbers 1 through 10. If you encountered 1 in the middle, you wouldn't just swap it with the adjacent number; you'd move it all the way back to the beginning.

That's essentially a selection sort. A selection sort goes through the list, finds the "lowest" (e.g., first) item, and moves that item to the beginning. It then goes through the list starting with the second item, finds the next-lowest item, and moves that item to the second spot. It repeats this for every item in the list.

Put in terms of our algorithms, selection sort would look like this:

1. For each spot in the list:
   a. Check each spot from this spot to the end.
   b. Move the lowest value found to this spot.

Of course, that simple representation belies a lot of complexity. First, we would need variables to track both the value and the index of the current lowest value. We would then need to compare each value to this current lowest value.

Let's try this one out as well. Figure 5.2.3 shows the results of a selection sort after each passage; the first row (row #0) is before any sorting, row #1 is after one

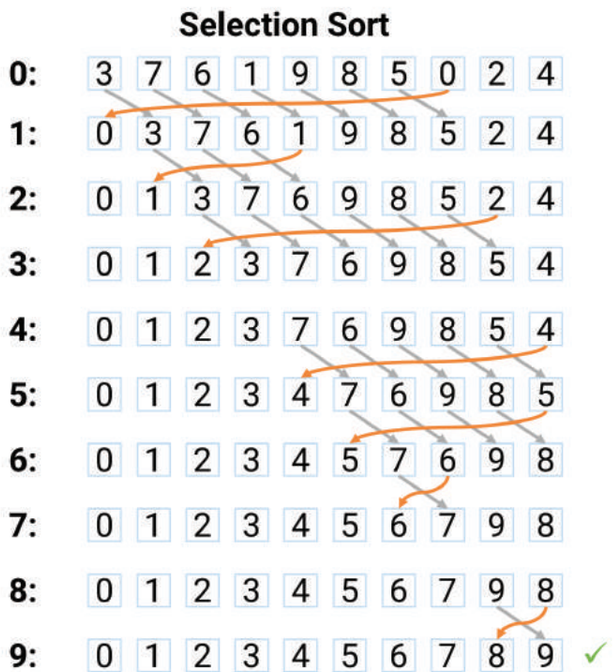


Figure 5.2.3

passage of the algorithm, and so on. Notice that we're finding the smallest numbers and moving them to the start; we could also find the largest numbers and move them to the end. That would still be a selection sort: selecting each value in order and moving it to the right position.

We start with the same unordered list on row #0. Then, starting at position 1, we go look for the lowest number. We find it, move it to the front, and move all the other numbers back one (you might also see this implemented by swapping these two numbers instead of moving the others). We now know the number in position 1 is correct. So, we repeat the algorithm, this time starting at position 2: we find the smallest number from position 2 to the end, and move it to position 2. We repeat for each position in the list.

Notice that there are a couple of duplicate rows here: the list is unchanged in runs #4 and #8. The algorithm has no way of knowing that the next number is in the right place until it checks, unlike bubble sort where it automatically left that number in its place. Thus, with selection sort, we guarantee n executions, and each execution involves up to n comparisons. So, like bubble sort, selection sort runs in $O(n^2)$. It's very similar to our check for duplicates in our roster example; we still repeatedly compare every pair, but we track the lowest and move it to the beginning.

Insertion Sort

Still though, the selection sort probably isn't the way you would sort a list. You'd likely use something like a hybrid of the two methods. You'd check the second value, and if it was lower than the first, you'd move it to the front. Then you'd check the third value, and you'd place it in order with the first two; if it was the lowest or the highest among these, you'd move it to the beginning or leave it where it is as with a selection sort; if it was between the first two values, though, you'd go ahead and move it between these two values.

In other words, for each item in the list, you'd start at the beginning, searching for the first value that was larger than the current item, and move it before that item. In that way, you're not just finding the first value, then the second value, and so on as with a selection sort, but instead you're putting each value in place one by one.

The algorithm for insertion sort would look something like this:

- For each unsorted value in the list:
  - **a.** For each sorted value earlier than the unsorted value in the list:
    - **i.** If the sorted value is greater than the unsorted value, move the unsorted value before the sorted value.

So, let's check this one out as well. Like the others, row #0 in Figure 5.2.4 is before any sorting, row #1 is after one run of the algorithm, and so on.

The best way to think about insertion sort is that it has an unsorted half and a sorted half. With each iteration, it takes the first unsorted value and puts it in the right position relative to the sorted values. So, initially, the only sorted part of this list is the first number: 3. One number is always sorted. The algorithm then puts 7 in the correct place with regard to 3; after it. Now, the sorted portion of the list is 3 7. It then takes the first unsorted value in the list, 6. Relative to the sorted portion, 6 goes between 3 and 7. So, the algorithm moves 6 to the right spot, between 3 and 7, and the sorted portion is now 3 6 7. Then, the next unsorted value is 1. 1 is less than all the sorted values, so it moves it to the beginning; the sorted portion is now 1 3 6 7. This continues until the list is sorted.

Like selection sort, insertion sort doesn't know when some values are already sorted. Notice that run s #1 and #4 don't change anything; in run #4, the sort hadn't yet put 9 in place, so it didn't realize 9 was already in the right spot relative to the sorted portion. Notice also how every number stays in place until it is moved into the right spot; the number 2, in the second-to-last spot, isn't moved until the second-to-last execution. The number 4, in the last spot, isn't moved until the last execution. In an insertion sort, numbers are moved in the order of their original location to the

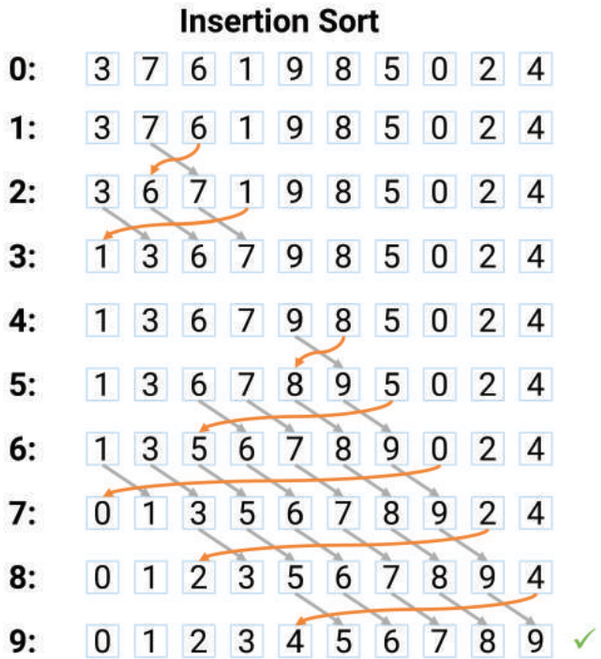


Figure 5.2.4

correct location; in a selection sort, numbers are moved in the order of their *value* to the correct location.

Like selection and bubble sort, insertion sort still runs in $O(n^2)$: it runs once for each item in the list, and within each run it compares to up to every other item in the list. In practice, bubble sort is less efficient on average than selection sort, which is less efficient than insertion sort. However, all three have the same worst-case scenario: $O(n^2)$.

Can we do better? We can!

Merge Sort

There are actually several more efficient sorting algorithms, but we'll only talk about one in detail: merge sort. Merge sort is difficult to explain in text or with static figures, but the heart of merge sort is a recursive method, which we'll call Merge. Merge takes an unsorted list of numbers and returns a sorted list. It does this by first splitting the incoming list into two lists, the left and the right, and sorting them individually using Merge. So, it first recursively breaks down the list until each list has only one item; a one-item list is guaranteed to be sorted.

Then, once it can guarantee that the left and right are both sorted, it repeatedly compares the first item from left to the first item from right. Whichever is lower gets added to the final sorted list. Once both left and right are empty, it returns the newly sorted list. That newly sorted list is then used in a previous call to Merge. So, if Merge is given a list with 16 items, it first sorts 8 2-item lists, then 4 4-item lists, then 2 8-item lists.

This algorithm looks something like this—since we're calling Merge recursively, we need to define it:

- Function `Merge`, given a `List`:
  - If there is only one item in `List`, return `List`.
  - Else, split `List` into `Left` and `Right`, and `Merge()` each.
  - While there are items in `Left` or `Right`, compare the first items of each and move the lowest to the sorted list.
  - Then, add any remaining items to the end of the sorted list.
  - Return the sorted list.

The result is an efficiency of $O(n \log n)$, significantly more efficient than bubble, insertion, or selection sorts. For a 1000 item list, the three earlier sorts would require up to a million operations; merge sort would require only 3000, making it over 3000x more efficient.

Merge sort is tougher to visualize than the others. The other algorithms ran effectively the same procedure multiple times, each time resulting in a slightly more-sorted list. Merge sort is more of an "all or nothing" sort; there are no distinct runs in the same way. That said, there are separate stages. Let's try to visualize this. In the Figure 5.2.5, numbers listed side-by-side are in the same list; spaces separate lists.

Initially, the list is broken up into ten one-item lists. A one-item list is guaranteed to be sorted. Then, for each pair of lists (e.g., 3 and 7, 6 and 1), merge sort merges the left list and the right list into one sorted list by taking the smallest values first. To compute the first list, it compares 3 and 7, finds 3 is smaller, and grabs 3; then, there are no more items in the left list, so it grabs the remaining item in the right list, 7. This leads to 5 sorted pairs: 37 16 89 05 24.

Then, merge sort again grabs each pair of lists and merges them. To merge 37 and 16, it compares 3 to 1, finds 1 is smaller, and grabs 1. Then it compares 3 to 6 (the next item in the right list), finds 3 is smaller, and grabs 3. Then it compares 7 (the next item in the left list) and 6, finds 6 is smaller, and grabs 6. Then, there are no more items in the right list, so it moves the remaining item (7) from the left list to the sorted list, ending with 1367. It does the same thing for the next two lists, 89 and 05, resulting in a second list of 0589.

Merge sort generally operates on two lists at a time (although implementations vary), so temporarily, 24 stays as a two-item list; then, the next round merges 0589 with 24, yielding 024589. Then, merge sort merges 1367 and 024589: it first grabs 0 from the right, then 1 from the left, then 2 from the right, then 3 from the left, then 4 from the right, then 5 from the right, then 6 from the left, then 7 from the left, and finally 8 and 9 from the right since the left is now empty.

Where does merge sort get this drastic uptick in efficiency? It comes from merge sort's reliance on each smaller list being sorted; because each small list is

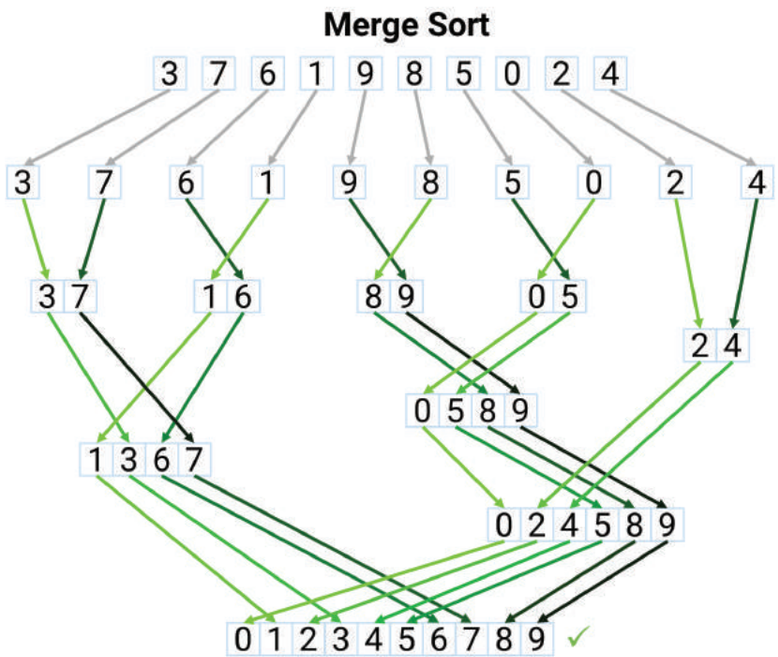


Figure 5.2.5

sorted, it need not compare each item to every other item, but rather just compare the first items on each smaller list. If the first item in the left list is smaller than the first item in the right list, then we know it's also smaller than every other item in the right list, even though we didn't manually check it against every other value. That's why this is so much more efficient. The $O(n^2)$ efficiency of bubble, selection, and insertion sorts came because all n items had to be compared to all other n items, and $n \times n = n^2$. With a merge sort, all n items need only be compared to a subset—log n, in fact—of the other items, and $n \times n = n \log n$.

There are several other sorting algorithms as well, like Shell sort, Heap sort, and Quick sort; these are all relatively close to the Merge sort in terms of efficiency.

5. Search Algorithms

That brings us to our final topic, **search algorithms**. Search algorithms take as input a value to try to find, and produce as output the index in a list where the value can be found.

Search Algorithms
Algorithms that take as input a list and a value for which to search, and produce as output the index or indices where that value was found in the list.

Search algorithms are far more efficient than sorting algorithms; in many ways, sorting algorithms are just repeated search algorithms that react to the results by moving values around. Thus, a sorting algorithm operates like searching for each value in the list one by one.

We'll only cover two search algorithms: the linear search and the binary search.

Linear Search

Linear search is exactly what you would do if you were searching for a name on an unsorted list: you'd check the names one by one, and when you found the right one, you'd stop. If you were looking for more than one instance, you'd search the entire list, writing down all the places the name was found. Figure 5.2.6 visualizes this.

This algorithm would look pretty simple. If we only wanted to find one value, we'd simply say, "for each value in the list, if it matches the search value, return

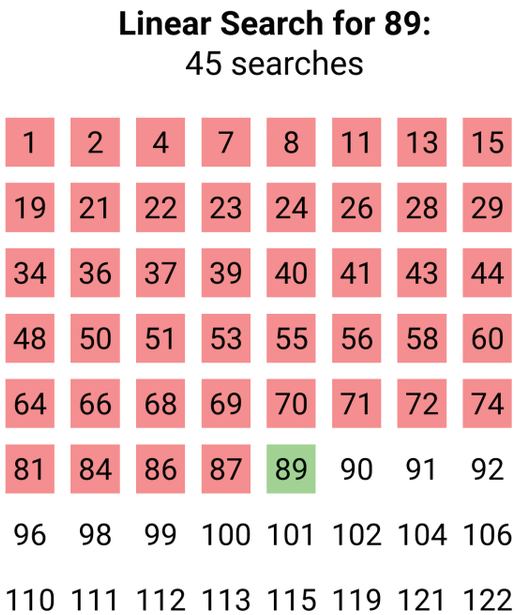


Figure 5.2.6

its index." If we wanted to find all the instances, we'd say, "for each value in the list, if it matches the search value, add its index to a list; then, return the list".

Either way, this runs in $O(n)$ time: we check each value at least once. The benefit is that this works whether the list is sorted or unsorted.

Binary Search

A binary search, on the other hand, works only if the list is sorted. A binary search is a recursive method that looks like this:

- Function `BinarySearch`, given `List` and `value`:
  - If the middle value of `List` is `value`, return the index of the middle value.
  - If the middle value of `List` is greater than `value`, `BinarySearch` the left side of `List` with `value`.
  - If the middle value of `List` is less than `value`, `BinarySearch` the right side of `List` with `value`.

That's not quite as complicated as it sounds. Think of it in terms of a number-guessing game. I say, "I'm thinking of a number between 1 and 100." You guess 50. I say, "Higher." Now, you've cut the possible answers in half. Instead of 1 to 100, it's now 51 to 100. So, you guess 75. I say "Lower." Now the search space is 51 to 74. You guess 63, I say "Higher"; the search space is now 64 to 74. You guess 69, I say "Lower"; now it's 64 to 68. You guess 66, I say "Higher"; now it's 67 or 68. You guess 68, I say "Lower"; now you know it's 67. Figure 5.2.7 visualizes a similar binary search with a sorted but non-sequential list of numbers.

It took you 7 guesses to get the right number. What's more, if I doubled the number of items to 200, it would only take you one extra guess; your first guess would be 100, which would cut the search space to either 1 to 99 or 101 to 200, the same size as the previous example.

Indeed, the efficiency of a binary search is far higher; it runs in $O(\log n)$. Doubling the number of items only requires one additional operation. It took up to 7 operations to find 1 number in 100, but it would take only up to 17 operations if you were finding a number between 1 and 100,000. Between 1 and a billion would take only 30 operations max!

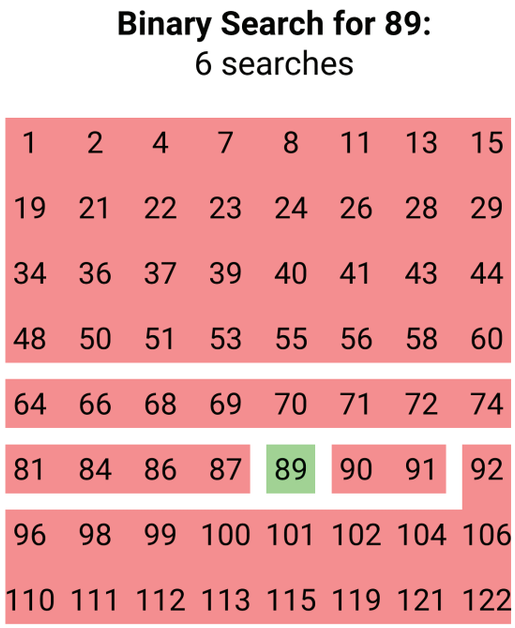


Figure 5.2.7

Linear Search vs. Merge Sort + Binary Search

The drawback of binary search, of course, is that the list must be sorted. Is it worth sorting the list just to allow us to do a binary search? Let's find out.

The efficiency of a linear search is simply $O(n)$. The efficiency of merge sort is $O(n \log n)$, and the efficiency of binary search is $O(\log n)$; thus, the efficiency of merge sort and binary search together is $O(\log n + n \log n)$, or $O((n + 1)\log n)$. Binary search is so efficient that it's barely a blip compared to merge sort. So, when is linear search more efficient, and when is merge sort + binary search more efficient?

We can see pretty clearly that actually, a linear search will always be more efficient. $\log n > 1$, so $n \log n > n$. While the binary search is just a blip in the efficiency of the merge sort + binary search method, the merge sort alone is less efficient than the linear search alone. However, if you're going to be searching more than once, that advantage disappears quickly. For a list with 100 elements, sorting will require around 600 operations with merge sort, but linear searching will require 100 operations per search. A binary search on 100 elements requires only 7 operations, so merge sort + binary search is more efficient than a linear search if more than seven searches will be performed.

Appendix of Functions and Methods

`bool(variable)` Takes as input some variable and attempts to convert it to a boolean, returning the boolean value if successful or raising a `ValueError` if unsuccessful. 61

`close()` A method of a variable with type file, closes the file from further reading or writing. 227

`date.today()` After importing date from datetime, returns a date object representing the current date. 61

`find(text, [start], [end])` A method of the string data type that will find the first instance of the value of text within the string calling the method. Optionally, also takes parameters start and end to mark where to search in the string. 199

`float(variable)` Takes as input some variable and attempts to convert it to a float, returning the float if successful or raising a `ValueError` if unsuccessful. 61

`input(prompt)` Takes as input some string to use as a prompt for user input, and returns as a string the text the user enters. 61

`int(variable)` Takes as input some variable and attempts to convert it to an integer, returning the integer if successful or raising a `ValueError` if unsuccessful. 61

`items()` A method of the dictionary type that returns all the items in the dictionary as (key, value) tuples. 247

`keys()` A method of a dictionary type that returns a list of all the keys in that dictionary. 243

`len()` A function that takes as input a variable with a length, such as a string of characters or a list of items, and returns its length. 82

`open(filename)` A function that takes as input a filename and, optionally, a write mode ("r" for read, "w" for write, "a" for append), and opens the file for access. 227

`print(message)` Takes as input a message as a string of characters and prints it to the console. 19

`random.randint(min, max)` Returns a random integer greater than or equal to min and less than or equal to max. 130

`range()` Takes as input two variables, a first number and a last number, and provides the list of numbers for a for loop to iterate over in a for loop. 125

`readline()` A method of a variable of type file, reads and returns the next line of the file as a string. 232

`split([separator])` A method of the string data type that will split a string up into a list of smaller strings. If a separator string is given, that string will be used to determine where to split; if not, the string will be split by spaces. 201

`str(variable)` Takes as input some variable and returns a string representation of the variable's value. 59

`turtle.forward(distance)` Takes as input distance as a float and moves the turtle forward the given distance. 49, 65

`turtle.penup() and turtle.pendown()` Two methods of the turtle library that toggle off and on, respectively, whether the turtle draws lines as it moves. 205

`turtle.right(angle)` Takes as input an angle as a float and rotates the turtle the given number of degrees. 49, 65

`turtle.write(message, [move], [align], [font])` A method of the turtle library that will write the given message on the canvas. If move is True, it will move the turtle along with the text. Align determines whether the text is left, right, or center aligned, and font is a three-tuple that contains the font face, size, and style. 204

`type(variable)` Takes as input some variable or value directly and returns the type of the variable such as an integer or string of characters. 57

`values()` A method of the dictionary type that returns a list of all the values of the dictionary. 243

`write(text)` A method of a variable with type file, writes the text to the file. 227

Glossary

A

Abstraction A principle of object-oriented programming that states that only essential information should be made visible to the outside program. 270

Algorithm Technically, a collection of steps that transforms input into output; commonly, a complex set of lots of steps that is only feasible to perform with the efficiency of a computer. 273

And An operator that acts on two boolean (`true` or `false`) values and evaluates to "`true`" if and only if *both* are `true`. 72

Arguments Values passed into parameters during a function call. Essentially, these are the values assigned to the function's dedicated variables (i.e., parameters). 141

Assignment Operator An operator that takes the output of an expression and assigns it to a variable. 81

B

Big O Notation A notation for expressing the worst-case efficiency of an algorithm in terms of the size of the input. 274

Boolean A simple True or False value. 20

Boolean Operators Operators like "`and`" and "`or`" that act on pairs of boolean (`true` or `false`) values, or that act on single boolean values, like "`not`". 72

C

Catch A control structure that designates what error it anticipates in a `try` block and provides the code to execute if that error arises. 156

Catching Errors Using error handling to prevent a program from crashing when an error is encountered. 155

Character A single letter, number, symbol, or special character. 189

Class A custom data type comprised of multiple variables and/or methods. Instances or objects are created based on the template provided by the class. 255

Comments Notes from the programmer supplied in-line alongside the code itself, designated in a way that prevents the computer from reading or attempting to execute them as code. 46

Compilation Errors Errors that occur during the computer's read through of the code. 30

Compile To translate human-readable computer code into instructions the computer can execute. In the programming flow, this functions as a check on the code the user has written to make sure it makes sense to the computer. 5, 21

Complexity The rate at which the number of operations requires to run an algorithm grows based on the value of the input on which it operates. 274

Conditional Statements Programming statements that control what code is executed based on certain conditions; usually of the form "`if`", "`else if`", and "`else`". 97, 105

Console An output medium for a program to show exclusively text-based output. 7

Constant Order, $O(1)$ The same number of operations are required regardless of the size of the data set. 275

Constructor A common type of method in writing classes that specifies some code to run whenever a new instance of the class is created. The constructor often has parameters that provide values to initialize the variables defined by the class. 261

Control Structures Statements that control the flow of execution of the program. Or, more simply, lines of code that control when other lines of code run. 97

D

Data Structures Approaches to organizing abstract data types, such that the data can be accessed efficiently. 175

Data Type The type of content a variable holds, like an integer or a string of characters. 56

Debugging Resolving problems in code, whether it be errors thrown in compilation or running or mismatches between the desired and observed output. 29

Destructor A common type of method in writing classes that specifies how the instance of a class is to be destroyed, such as releasing its memory back to the computer. 261

Dictionaries A data structure comprised of key-value pairs, where a key is entered into the dictionary to get out a value. Similar to or synonymous with Maps, Associative Arrays, HashMaps, and Hashtables. 176, 239

Dictionary Key A value then, when passed into a dictionary, returns a corresponding value, like a word and its definition. Similar to a variable. 239

Dictionary Value A value returned in response to a key in a dictionary. Similar to a value of a variable outside a dictionary. 239

Documentation Collected and set-aside descriptions and instructions for a body of code. 46

E

Edge Case A rare situation that requires special processing to handle. 128

Else-If Statement A conditional control structure that runs a block of code if all preceding `if-then` and `else-if` statements have been false *and* some other conditions are met. 106

Else Statement A conditional control structure that runs a block of code if all preceding `if-then` and `else-if` statements have been false. 105

Encapsulation The ability to combine variables and methods into class definitions in object-oriented programming. It helps avoid modification or misuse of data by other functions or programs. 260

Error A problem that prevents code from continuing to run if not handled. 21

Escape Sequence A sequence of characters that, when occurring in a string, is interpreted to have a meaning beyond the characters themselves. The most common example is "\n", which is interpreted by many languages as representing a newline character. 192

Event-Driven Programming A type of programming where the program generally awaits and reacts to events rather than running code linearly. 42

Exception An error that a program might want to anticipate and catch instead of outright avoiding. 99

Exception Handling A control structure that catches certain anticipated errors and reacts to them accordingly. 99

Execution Running some code and having it actually perform its operations. 5, 21

Exponential Order, $O(2^n)$ The number of operations required increases by a constant raised to the size of the data set. 275

F

File Input and Output The complementary processes of saving data to a file and loading data from a file, generally such that the state of the memory of the program is the same after saving and loading have occurred. 176, 225

Finally A control structure that designates some code to run after a `try` and `catch` structure regardless of whether or not an error arose. 157

Floor Division Division that rounds the result down to the nearest integer. 84

For-Each Loop A loop control structure that runs a block of code a predetermined number of times, where the number of times comes from the length of some list and the items in the list are automatically loaded into a variable for usage in the block of code. 126

For Loop A loop control structure that runs a block of code a predetermined number of times. 123

Function A segment of code that performs a specific task, sometimes taking some input and sometimes returning some output. 41, 98, 139

Function Body The code that a function runs when called. 140

Function Call A place where a function is actually used in some code. 139

Function Definition A segment of code that creates a function, including its name, parameters, and code, to be used by other portions of a program. 140

Function Header The name and list of parameters a function expects, provided as reference to the rest of the program to use when calling the function. 140

G

Getter A common type of method in writing classes that returns the value of a variable contained within the class. They are commonly used to allow other processing to occur whenever the variable is accessed, like logging. 261

Global Variable A variable whose scope is the entire program; it is visible within any function or method in the program. 236

Graphical User Interface An output medium that uses more than just text, like forms, buttons, tabs, and more. More programs are graphical user interfaces. 8

H

Hexadecimal A short-hand expression of the ones and zeroes that comprise computer data, comprised of 16 characters, 0 through 9 and A through F. 190

Homogeneity A property of lists determining whether they can accept multiple types of variables. A homogenous list can only accept one type of variable; a non-homogenous or heterogenous list can accept multiple types. 207

I

If-Then Statement A conditional control structure that runs a block of code only if a certain condition is `true`. 105

Immutable Variable A variable whose value cannot change after it has been declared. 182

Increment Repeatedly adding a constant, typically one, to a variable. 85

Indentation Spaces at the beginning of a line that are used to group together blocks of code. All consecutive lines of code at the same level of indentation are in a single code block. 99

Index A number used to access a particular element from a list-like data structure. Traditionally, most programming languages assign the first item of a list-like data structure the index 0. 175

Infinite Loop A loop that will never end because the conditions for ending the loop will never be met. 130

Inheritance A principle of object-oriented programming where classes can be created that are **"`subclasses`"** of other classes, inheriting all the variables and methods from the other class while supplying new variables, methods, or behaviors of these own. 270

Input Data that is fed into a program for it to operate upon. 4

Instance A single set of values of a particular class. Classes may be comprised of multiple variables; an instance is a set of values for these variables. The term "instance" is often used interchangeably with the term "object". 256

Iterate To repeat code a number of times. For example, if a loop runs for each item in a list, the loop "iterates" over the list. Each time the code is repeated is a single iteration. 126

Iteration A single execution of a repeated task or block of code. 123

K

Keyword Parameters A special kind of optional parameter to which the program may choose to assign an argument during a function call, or may ignore. Typically, keyword parameters have a default value that is used if it is not overridden by a function call. 150

L

Linear Order, $O(n)$ The number of operations required increases linearly with the size of the data set. 275

Line of code A single instruction for the computer to perform. 4

Linked List A list-like structure where the location of each item in the list is contained in the previous item in the list. 220

List-Like Structures Also referred to as sequences and collections, a data structure that holds multiple individual values gathered together under one variable name, accessed via indices. Includes to lists, arrays, and tuples. Lists are simultaneously a type of data structure and a specific type in some languages. 175, 207

Lists A data structure that holds multiple individual values gathered together under one variable name, accessed via indices. Similar to arrays and tuples. 176

Lists A mutable form of a list-like structure in Python. 208

Logarithmic Order, $O(\log n)$ The number of operations required increases with the square root of the size of the data set. 275

Logical Operators Operators that perform logical operations, such as comparing relative values, checking equality, checking set membership, or evaluating combinations of other logical operators. 67

Loop A programming control structure that executes a segment of code multiple times. 98, 123

Loop Control Variable A variable whose value is the number of times a loop has run. It is used to check if the loop should keep running (e.g. if it has run as many times as it's supposed to). 124

M

Mathematical Operators Operators that perform mathematical functions, like adding numbers together or assigning values to variables. 67

Methods Functions that are contained within data types. 186

Modulus The remainder function, returns the remainder of one number divided by another. 81

Multi-Dimensional Lists A list-like structure where the items in a list are themselves lists, such that the practical effect is a multi-dimensional list. 216

Mutability Whether or not a variable can have its value changed after being declared. 182

Mutable Variable A variable whose value can change after it has been declared. 182

N

Nested Conditional A conditional statement that is itself controlled by another conditional statement. More simply, an `if-then` statement within another `if-then` statement. 116

Newline Character A Unicode character, either LF (line feed) or CR (carriage return), that is rendered as the beginning of a new line of text. 190

Not An operator that acts on one boolean (`true` or `false`) value and evaluates to the opposite value (`false` becomes `true`, `true` becomes `false`). 72

Null The "value" a variable has when it doesn't actually have a value. 54

O

Object An object is a custom data structure that organizes and encapsulates variables and methods into a single data type. It is used near-interchangeably with "instance." 255

Object-Oriented Programming A programming paradigm where programmers define custom data types that have custom methods embedded within them. 41, 255

Operators Specific, simple functions that act on primitive data types, like integers and strings. 67

Or An operator that acts on two boolean (`true` or `false`) values and evaluates to "`true`" if and only if *at least one* is `true`. 72

Output What the computer provides in return after running some lines of code. 4

P

Parameter A variable for which a function expects to receive a value when called, whose scope is the function's own execution. 140

Passing by Reference An approach for passing arguments into a function where the function is able to modify the variable whose value was getting passed, changing it for both the function and the code that called the function. 176

Passing by Value An approach for passing arguments into a function where the function is not able to modify the variable whose value was getting passed, only its local parameter that accepts the argument. 176

Polymorphism The principle that a method call can behave differently depending on the arguments and object with which it is called. 270

Polynomial Order, $O(n^3)$ The number of operations required increases with the size of the data set raised to a larger exponent. 275

Print Debugging A form of debugging where print statements are added throughout the code to check how the program is flowing. 33

Print output some text to the console. 19

Program An independent collection of lines of code that serves one or more overall functions. 4

Programming Writing code through an iterative process of writing lines of code, attempting to execute them, and evaluating the results. 17

Q

Quadratic Order, $O(n^2)$ The number of operations required increases with the size of the data set squared. 275

Queue A list-like structure that follows the "First-In-First-Out" paradigm, where we can only access the least recently-added item on the list and can only access it by removing it from the list. 219

R

Recursion A programming method characterized by functions that, during their operation, call additional copies of themselves; see also, recursion. Recursion involves breaking down a problem into smaller instances recursively until each of them can be independently solved. Solutions to these smaller instances combine to form the solution for the original problem. 275

Reference An alias to a variable that already exists. Either the reference or the variable name can be used to access the value stored in that variable. 177

Relational Operators Operators that check the relationships between multiple variables, such as checking if they are equal or if one is greater than another. 68

Return Statement The line of code that defines what output will be send back at the end of a function. 140

Rubber Duck Debugging A form of debugging where the programmer explains the logic, goals, and operations to an inanimate listener to methodically step through the code. 34

Runtime Errors Errors that arise when trying to actually execute the code. 30

S

Scope Debugging A form of debugging where print statements are added to check the status of the variables in the program at different stages to see how they are changing. 33

Scope The portion of a program's execution during which a variable can be seen and accessed. 101

Search Algorithms Algorithms that take as input a list and a value for which to search, and produce as output the index or indices where that value was found in the list. 283

Self A keyword in Python classes that is used to refer to the instance itself. It defines the scope of variables and methods that methods in the class can see. 257

Self-Assignment A common programming pattern where a variable is assigned to the output of an expression that included the variable itself. 85

Self-Documenting Code Code whose variables and functions are named in a way that makes it clear what their underlying content and operations clear to the reader. 47

Setter A common type of method in writing classes that sets a variable contained within the class to a new value. They are commonly used to allow other processing to occur whenever the variable is changed, like logging. 261

Sorting Algorithms Algorithms that take as input a list, and produce as output a sorted version of that list. Examples include bubble sort, insertion sort, selection sort, merge sort, shell sort, quick sort, and heap sort. 277

Stack A list-like structure that follows the "Last-In-First-Out" paradigm, where we can only access the most recently-added item on the list and can only access it by removing it from the list. 219

String A data structure that holds a list, or a string, of characters. 175, 189

String Concatenation The process of putting two or more strings together in order to form one string made of the individual strings. For example, concatenating "A" with "B" would give "AB". 193

String Slicing The Python term for obtaining substrings from within a string based on character indices. 194

T

Truth Tables Tables that map out the results of a statement in boolean logic (that is, using boolean operators) depending on the values of the individual variables. 77

Try A control structure that sets aside a block of code in which an error might occur so that the computer will look for error handling capabilities. 156

Tuple An immutable form of a list-like structure in Python. 208

U

Uncaught Error An error that is not handled by error handling code, and thus usually forces the program to crash. 155

Unicode A computing industry standard that sets what hexadecimal codes correspond to what characters, so that text appears consistent across platforms. 190

V

Value The content of some variable. The variable `myAge` might hold the value 29. The variable `yourName` might hold the value "Adelene". 51

Variables Alphanumeric (letters and numbers) identifiers that hold values, like integers, strings of characters, and dates. 51

W

While Loop A loop control structure that runs a block of code until a certain logical expression is satisfied. 124

X

Xor An operator that acts on two boolean (`true` or `false`) values and evaluates to "`true`" if and only if *exactly one* is `true`. 72

Z

Zero-Indexing A convention in most programming languages where the first item of a list of items is considered the "0th" item, not the 1st item. 195

Index

F

G

H

I

J

K

L

M

N

O

P

Q

R