

INTRODUCTION TO COMPUTING

David Joyner

1st Edition

**Mc
Graw
Hill**
Education



Introduction to Computing

1st Edition

David Joyner





Copyright © 2016 by McGraw-Hill Education LLC. All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base retrieval system, without prior written permission of the publisher.

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 1: Computing

- 1.1** Computing 3
- 1.2** Programming 17
- 1.3** Debugging 29

Unit 2: Procedural Programming

- 2.1** Procedural Programming 41
- 2.2** Variables 51
- 2.3** Logical Operators 67
- 2.4** Mathematical Operators 81

Unit 3: Control Structures

- 3.1** Control Structures 97
- 3.2** Conditionals 105
- 3.3** Loops 123
- 3.4** Functions 139
- 3.5** Error Handling 155

Unit 4: Data Structures

- 4.1** Data Structures 175
- 4.2** Strings 189
- 4.3** Lists 207
- 4.4** File Input and Output 225
- 4.5** Dictionaries 239

Unit 5: Object-Oriented Programming

5.1 Objects 255

5.2 Algorithms 273

Appendix of Functions and Methods A-1

Glossary G-1

Index I-1

UNIT 1

INTRODUCTION TO COMPUTING

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 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.

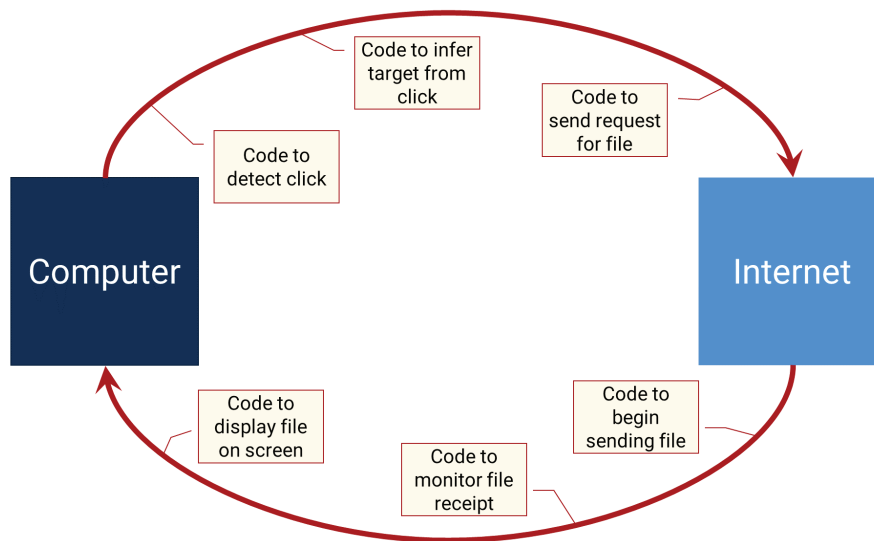


Figure 1.1.1

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.

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.

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

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.

Line of code

A single instruction for the computer to perform.

Program

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

Input

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

Output

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

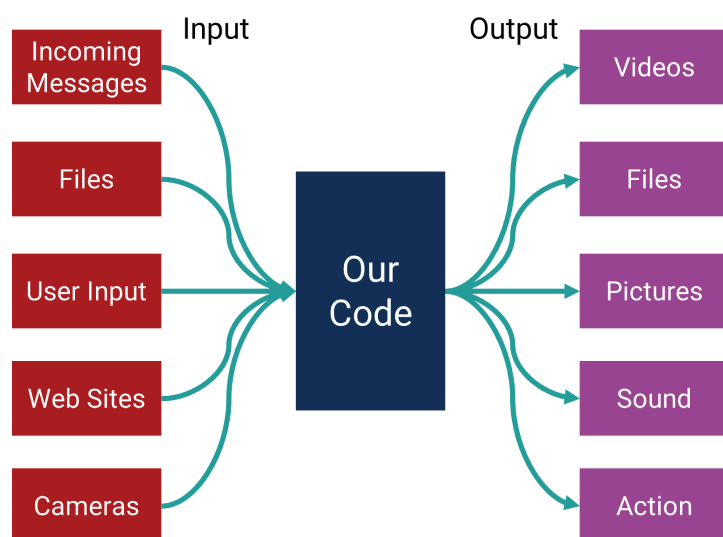


Figure 1.1.2

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

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

Console

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

```

C:\Windows\system32\cmd.exe
C:\Users\David\PycharmProjects\CS1301>py -3 ConsoleSample.py
Hello, world!
I'm a console program. That means I can only print text.
I can take input, too, but only in the form of text.

Want to see?
Enter a number, and I'll tell you the square of that number: 5
5 squared is 25

That's not all I can do!
Enter a list of numbers and I'll give you their average.
I'll ignore any non-numbers you put in, though.
When you're done, type 'end'.
Enter another number, or 'end' to end: 91
Enter another number, or 'end' to end: 94
Enter another number, or 'end' to end: 96
Enter another number, or 'end' to end: 97
Enter another number, or 'end' to end: 98
Enter another number, or 'end' to end: end

Oh, you're done? Okay, the average of those numbers is 95.2.

Even though I only work in text, there are still a lot of things I can do.
C:\Users\David\PycharmProjects\CS1301>

```

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

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 Interface

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

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.

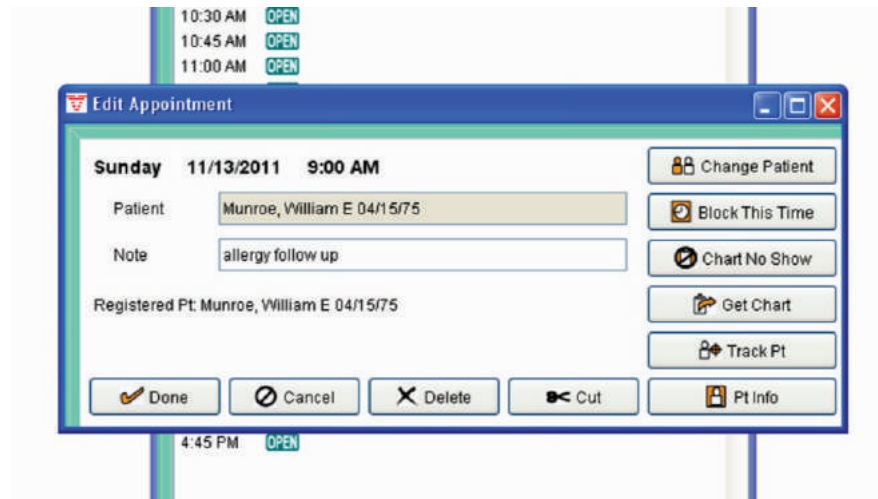


Figure 1.1.4

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	<i>#Gets the library we need to see the date</i>	5
2	from datetime import date	5.1
3	<i>#Prints the integer 5</i>	2016-09-19
4	print (5)	
5	<i>#Prints the number 5.1</i>	
6	print (5.1)	
7	<i>#Prints today's date</i>	
8	print (date.today())	
9		

Figure 1.1.5

#	ForLoopsWithUnknownRanges-1.py	Output
1	<code>#Creates sum with the value 0</code>	Enter number #1: 91
2	<code>sum = 0</code>	Enter number #2: 92
3	<code>#Loop 10 times</code>	Enter number #3: 93
4	<code>for i in range(1, 11):</code>	Enter number #4: 94
5	<code> #Gets the user's number</code>	Enter number #5: 95
6	<code> nextNumber = int(input("Enter number #" + str(i) + ": "))</code>	Enter number #6: 96
7	<code> #Add the inputted number to the sum</code>	Enter number #7: 97
8	<code> sum += nextNumber</code>	Enter number #8: 98
9	<code>#Print the sum over 10</code>	Enter number #9: 99
10	<code>print(sum / 10)</code>	Enter number #10: 100
11		95.5
12		
13		
14		

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 starts 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 *needed* 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.

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.

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.

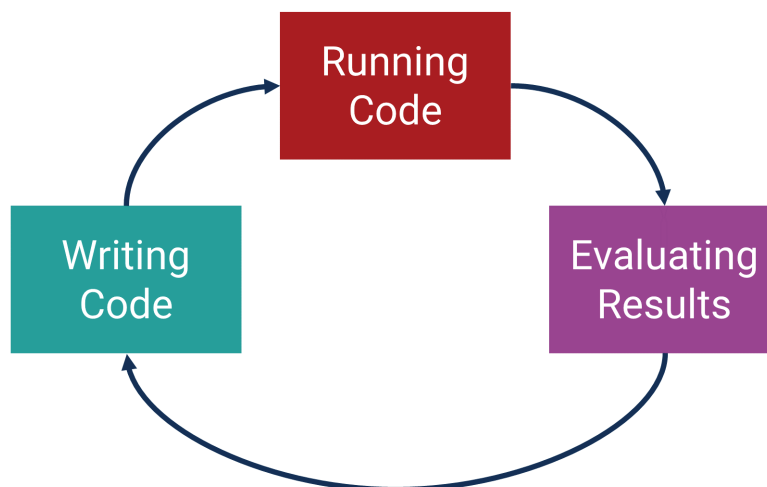


Figure 1.2.1

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.

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.

Print
output some text to the console.

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	<code>print("Hello, world")</code>	Hello, world
2		

Figure 1.2.3

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

Figure 1.2.4

boolean

a simple True or False value.

Printing Other Values

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

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



Figure 1.2.5

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.

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.

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

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.

Error

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

#	ExecutingCodeinPython.py	Output
1	<code>print("This is Line 1")</code>	This is Line 1
2	<code>print("This is Line 2")</code>	This is Line 2
3	<code>print("This is Line 3")</code>	This is Line 3
4	<code>print("This is Line 4")</code>	This is Line 4
5	<code>pritrn("This is Line 5")</code>	Traceback (most recent call last):
6		File "ExecutingCodeinPython.py",
7		line 5, in <module>
8		pritrn("This is Line 5")
9		NameError: name 'pritrn' 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.

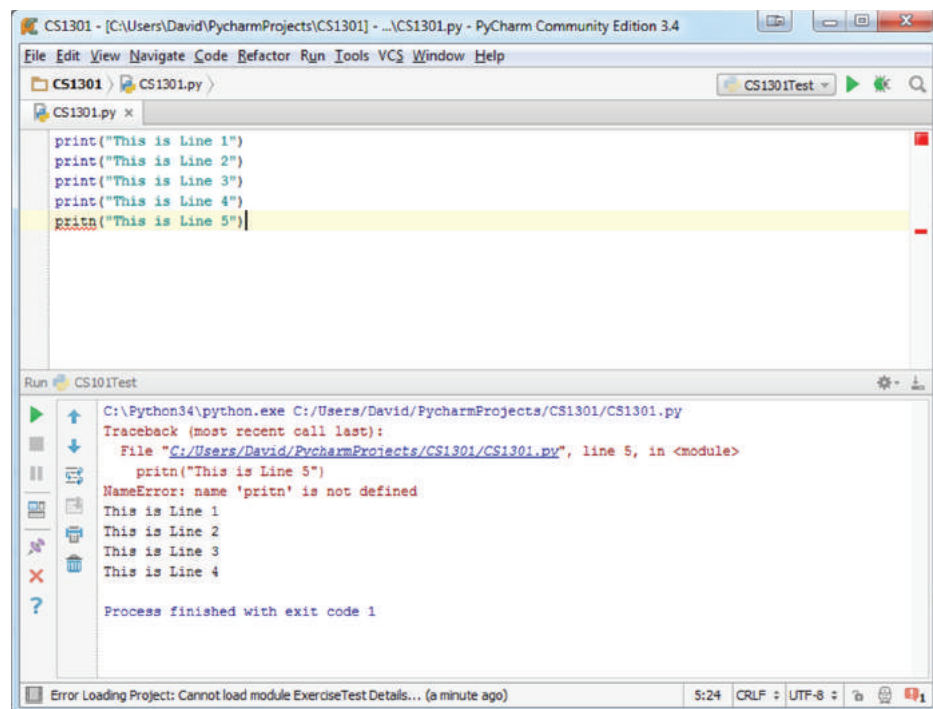
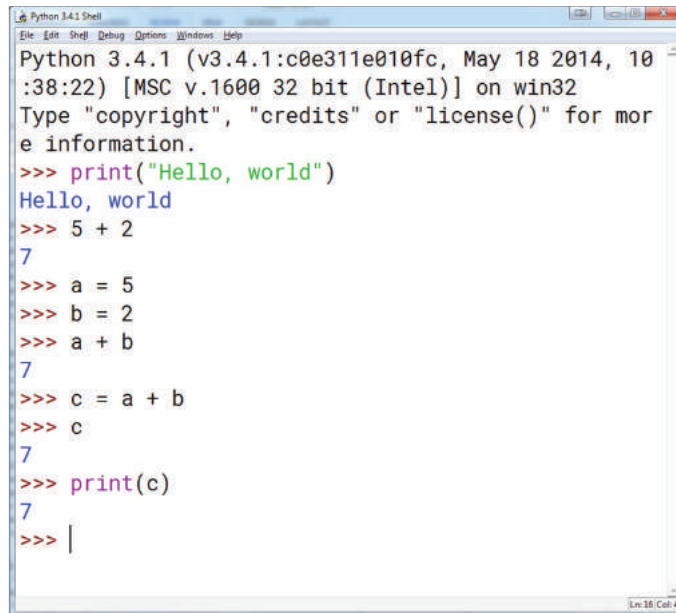


Figure 1.2.7



```

Python 3.4.1 Shell
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello, world")
Hello, world
>>> 5 + 2
7
>>> a = 5
>>> b = 2
>>> a + b
7
>>> c = a + b
>>> c
7
>>> print(c)
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	<code>for i within range(1, 9):</code>	File "ErrorsinPython.py",
2	<code>print(i)</code>	line 1
3		for i within range(1,9):
4		^
5		SyntaxError: invalid syntax
6		

Figure 1.2.9

#	IncorrectOutputinPython-1.py	Output
1	<code>for i in range(1, 9):</code>	1
2	<code>print(i)</code>	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	<code>for i in range(1, 10):</code>	1
2	<code>print(i)</code>	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.