CORY ALTHOFF

the self-taught PROGRAMMER

the definitive guide
to programming
professionally

The Self-taught Programmer

Cory Althoff

ISBN 978-1-940733-01-2
Library of Congress Number
www.theselftaughtprogrammer.io

This book is dedicated to my parents Abby and James Althoff for always supporting me.

Part I Introduction to Programming

Chapter 1. Introduction

"Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program."
— Linus Torvalds

I majored in Political Science at Clemson University. Before I chose this path I considered Computer Science. I even enrolled in an Introduction to Programming class my Freshman year, but quickly dropped it. It was too difficult. While living in Silicon Valley after graduation I decided I needed to learn to program. A year later, I was working as a software engineer II at eBay (above an entry level software engineer, but below a senior software engineer). I don't want to give the impression that this was easy. It was incredibly challenging. In between throwing things at the wall, it was a lot of fun too.

I started my journey learning to program in Python, a popular programming language. This book, however, is not about teaching you how to program in a specific language (although it does). There are plenty of amazing books, classes and resources that do that already. The focus is everything else those standard resources do not teach you. It's about the things I had to learn on my own in order to become a software engineer. This book is not meant for someone looking for a casual introduction to programming so they can write code as a hobby. This book is written specifically for those looking to program professionally. Whether your goal is to become a software engineer, an entrepreneur or to use your new programming skills in another profession, this book was written for you.

Learning a programming language is only part of the battle. There are other skills you need in order to speak the language of computer scientists. I will teach you everything I learned on my journey from programming novice to professional software engineer. I wrote this book to give aspiring programmers an outline of what they need to know. As a self-taught programmer, I didn't know what I needed to learn. The introduction to programming books are all the same. They teach you the basics of how to program in either Python or Ruby and send you on your way. The feedback I've heard from people finishing these books is, "What do I do now? I am not a programmer yet, and I don't know what to learn next." This book is my answer to that question.

How This Book Is Structured

This book is divided into six parts, based on moving through the following stages: learning to program, learning object-oriented programming, learning to use programs (like your operating system) that will make you a better programmer, learning Computer Science, learning to program for production and getting a job and working on a team.

Many of the subjects covered in a single chapter of this book could be—and are—covered by entire books. My goal is not to cover every detail of every subject you need to know. My goal is to give you a map—an outline of all of the skills you need to develop in order to program professionally.

Part I: Introduction to Programming. You will write your first program as quickly as possible, hopefully today. If you already know how to program you can use this section to start learning Python. If you already know Python, skip it.

Part II: Introduction to Object-oriented Programming. I cover the different programming paradigms—focussing on object-oriented programming—and build a game that will show you the power of programming. After this section, you'll be hooked on programming.

Part III: Introduction to Programming Tools. You learn to use different tools to take your programming productivity to the next level. By this point you are hooked on programming and want to get even better. You will learn more about your operating system, how to collaborate with other engineers using version control, how to use your Interactive Development Environment to boost your productivity and how to install and manage other people's programs.

Part IV: Introduction to Computer Science. Now that you can program, you will have all kinds of questions about how everything works. This section is where a lot of those questions get answered. I cover algorithms and data structures, network programming and computer architecture.

Part V: Programming for Production. You will learn to program for production (create code that is actually used by other people). I cover the software development process, testing and best programming practices.

Part VI: Land a Job. The final section is about getting a job as a software engineer, working on a team and improving as a programmer. I provide tips on how to pass a technical interview, work on a team as well as advice on how to further improve your skills.

If you don't have any programming experience, you should try to practice programming on your own as much as possible between each section. There are additional resources to explore provided at the end of each section. Don't try to read this book too quickly. Instead, use it as a guide and practice for as long as you need in between sections .

Endgame First

The way I learned to program is the opposite of how Computer Science is usually taught, and I structured the book to follow this approach . Traditionally, you spend a lot of time learning theory ; s o much so, that many Computer Science graduates come out of school not knowing how to program. In his blog, *Why Can't Programmers.. Program?* , Jeff Atwood writes: "Like me, the author is having trouble with the fact that 199 out of 200 applicants for every programming job can't write code at all. I repeat: *they can't write any code whatsoever* ." This led Atwood to create the *FizzBuzz* coding challenge, a programming

test used in programming interviews to weed out candidates. Most people fail the challenge, and that's why we spend so much of this book learning the skills you will use in practice. Don't worry, we also learn how to pass the FizzBuzz test.

In *The Art of Learning* , Josh Waitzkin of *Searching for Bobby Fischer* fame, describes how he learned how to play chess in reverse. Instead of studying opening moves, he started learning the endgame (where there are only a few pieces left on the board) first. This gave him a better understanding of the game , and he went on to win many championships. Similarly, I think it is more effective to learn to program first , then learn theory later, once you are dying to know how everything works under the hood. That is why I wait until the fourth section of the book to introduce Computer Science theory. While theory is important, it will be even more valuable once you already have programming experience.

The Self-taught Advantage

Learning how to program outside of school is increasingly common. A 2015 Stack Overflow (an online community of programmers) survey found 48 percent of respondents did not have a degree in computer science [10] . When I was hired at eBay, I was on a team that included programmers with CS degrees from Stanford, Cal and Duke, as well as two Physics PhD's. At 25, it was intimidating to realize that my 21-year-old teammates knew 10 times more about programming and computer science than I did.

As intimidating as it might be to work with people who have bachelor's, master's and PhD's in Computer Science, never forget you have what I like to call the "self-taught advantage ." You are not reading this book because a teacher assigned it to you, you are reading it because you have a desire to learn, and wanting to learn is the biggest advantage you can have.

Why You Should Program

Programming can help your career regardless of your profession. If you are reading this book I assume you have already decided you want to learn to program. But I'm still going to cover why you should to give you extra motivation. Learning to program is empowering. I love coming up with new ideas, and I always have a new project I want to work on. Once I learned how to program, I could sit down and build my ideas without needing to find someone to do it for me.

Programming will also make you be better at everything. Seriously. There aren't many subjects that don't benefit from finely tuned problem-solving skills. Recently, I had to do the very tedious task of searching for housing on Craigslist, and I was able to write a program

to do the work for me and email me the results. Learning to program will free you from repetitive tasks forever.

If you do want to become a software engineer, there is an increasing demand for good engineers and not enough qualified people to fill the available positions. By 2020, an estimated one million programming jobs will go unfilled [45]. Even if your goal isn't to become a software engineer, jobs in fields like science and finance are beginning to favor candidates with programming experience.

Sticking With It

If you don't have any programming experience and are nervous about making this journey, I want you to know you are absolutely capable of doing it. There are some common misconceptions about programmers like they all are really good at math. This isn't true, but it is hard work. Luckily, a lot of the material covered in this book is easier to learn than you think.

In order to improve your programming skills you should practice programming every day. The only thing that will hold you back is not sticking with it, so let's go over some ways to make sure you do. I used a checklist to make sure I practiced every day and it really helped me stay focused.

If you need extra help, Tim Ferriss, a productivity expert, recommends the following technique to stay motivated. Give money to family or friends with the instructions that it is either to be returned to you upon completing your goal within a timeframe of your choosing, or donated to an organization you dislike.

How This Book is Formatted

The chapters in this book build on one another. If you see something you don't understand, it might have been covered in a previous chapter. I try not to re-explain concepts, so keep this in mind. Important words are italicized when they are defined. There is a vocabulary section at the end of each chapter where each italicized word is defined.

Technologies Used In This Book

This book teaches certain technologies in order to give you as much practical programming experience as possible. In some cases I had to choose between many different popular technologies. In Chapter 20: " Version Control " (for those readers who are

unfamiliar with version control, I will explain later), we go over the basics of using Git, a popular version control system. I chose Git because I consider it the industry standard for version control. I use Python for the majority of programming examples because it is a popular first programming language to learn, and it is a very easy language to read, even if you have never used it. There is also a huge demand for Python developers in just about every field. But, I try to be as technology agnostic as possible: focusing on concepts instead of technologies.

In order to follow the examples in this book you will need a computer. You computer has an *operating system* —a program that is the middleman between the physical components of the computer and you. What you see when you look at your screen is called a graphical user interface or GUI, which is part of by your operating system.

There are three popular operating systems for desktop and laptop computers: *Windows* , *Unix* and *Linux* . Windows is Microsoft's operating system. Unix is an operating system created in the 1970s. Currently, the most popular Unix operating system is Apple's OS X. Linux is an open-source operating system used by the majority of the world's *servers* . A server is a computer or computer program that performs tasks like hosting a website. *Open-source* means the software is not owned by a company or individual, instead it is maintained by a group of volunteers. Linux and Unix are both Unix-like operating systems, which means they are very similar. This book assumes you are using a computer running Windows, OS X or Ubuntu (a popular version of Linux) as your operating system.

Vocabulary

FizzBuzz : a programming test used in programming interviews to weed out candidates.
operating system : A program that is the middleman between the physical components of the computer and you.
open-source : Software that is not owned by a company or individual, but is instead maintained by a group of volunteers.
Windows : Microsoft's operating system.
Unix : An operating system created in the 1970s. Apple's OS X is a version of Unix.
Linux : An open-source operating system used by the majority of the world's servers.
server : A computer or computer program that performs tasks like hosting a website.

Challenge

Create a daily checklist that includes practicing programming.

Chapter 2. Getting Started

“A good programmer is someone who always looks both ways before crossing a one-way street.”
—Doug Linder

What is Programming

Programming is writing instructions for a computer to execute. The instructions might tell the computer to print Hello, World! , scrape data from the internet or read the contents of a file and save them to a database. These instructions are called *code* . Programmers write code in many different programming languages. In the past, programming was much harder, as programmers were forced to use cryptic, *low-level programming languages* like *assembly language* . When a programming language is low-level it is closer to being written in binary (0s and 1s) than a *high-level programming language* (a programming language that reads more like English), and thus is harder to understand. Here is an example of a simple program written in an assembly language:

```
global _start

    section .text

_start:

    mov    rax , 1

    mov    rdi , 1

    mov    rsi , message

    mov    rdx , 13

    syscall

    ; exit(0)

    mov    eax , 60

    xor    rdi , rdi
```

```
        syscall

    message:

        db      "Hello", 10

    43
```

Here is the same program written in a modern programming language:

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

As you can see, programmers today have it much easier. You won't need to spend time learning cryptic , low-level programming languages in order to program. Instead, you will learn to use an easy-to-read programming language called Python.

What is Python

Python is an open-source programming language created by Dutch programmer Guido van Rossum and named after the British sketch comedy group, *Monty Python's Flying Circus* . One of van Rossum's key insights was that programmers spend more time reading code than writing it, so he created an easy-to-read language. Python is one of the most popular and easiest to learn programming languages in the world. It runs on all the major operating systems and computers and is used in everything from building web servers to creating desktop applications. Because of its popularity, there is a large demand for Python programmers.

Installing Python

In order to follow the examples in this book, you need to have Python 3 (version 3 of Python) installed. You can download Python for Windows and OS X at http://python.org/downloads . If you are on Ubuntu, Python 3 comes installed by default. **Make sure you download Python 3, not Python 2. Some of the examples in this book will not work if you are using Python 2.**

Python is available for both 32-bit and 64-bit computers. If you have a new computer, purchased after 2007, it is most likely a 64-bit computer. If you aren't sure if your computer is 32-bit or 64-bit, an internet search should help you figure it out.

If you are on Windows or a Mac, download the 64- or 32-bit version of Python, open the file and follow the instructions. You can also visit

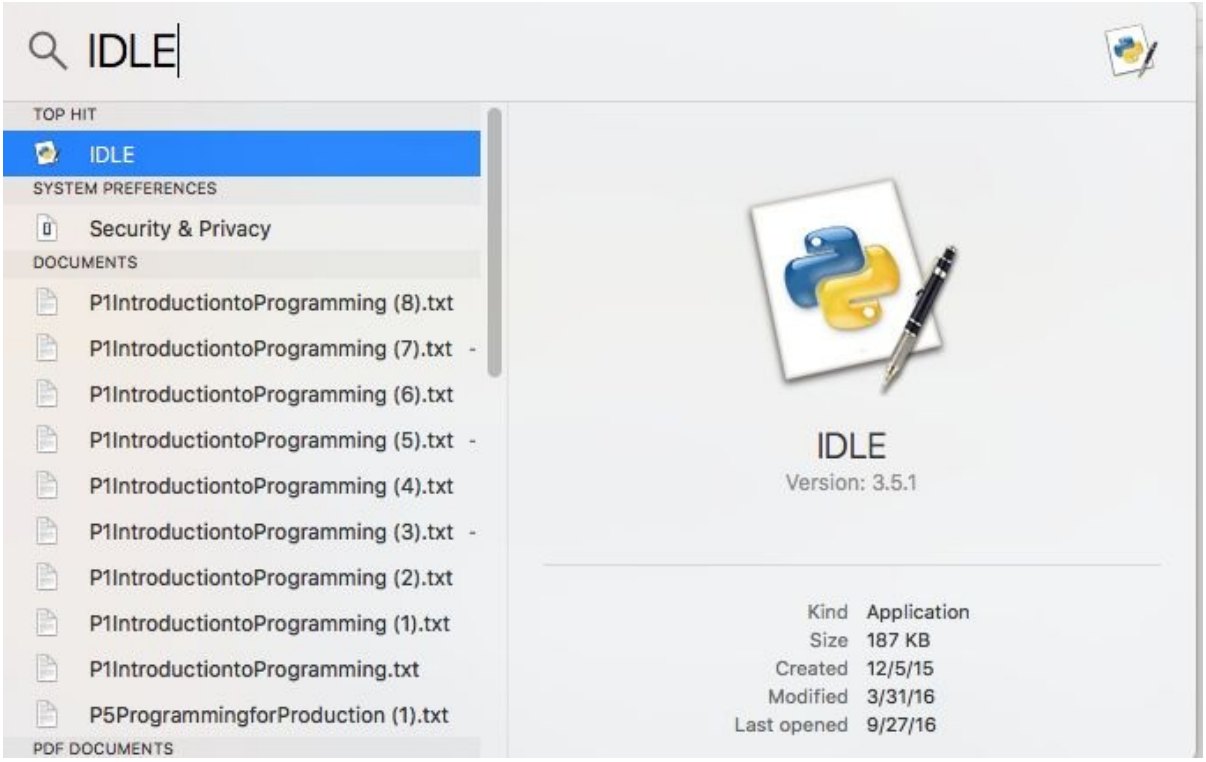
http://theselftaughtprogrammer.io/installpython for videos explaining how to install Python on each operating system.

Troubleshooting

From this point forward, you need to have Python installed. If you having problems installing Python, please skip ahead to chapter 11 to the section titled “Getting Help”.

The Interactive Shell

Python comes with a program called IDLE (short for interactive development environment); it is also the last name of Eric Idle, one of the members of *Monty Python’s Flying Circus* . IDLE is where you will be typing your Python code. Once you’ve downloaded Python, search for IDLE in Explorer (PC) , Finder (Mac) or Nautilus (Ubuntu) . I recommend creating a desktop shortcut to make it easy to find.



C lick on the IDLE icon, and a program with the following lines will open up (although this could change so don’t worry if the message is absent or different) :

Python 3.5.1 (v3.5.1:37a07cee5969, Dec 5 2015, 21:12:44)

```
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
```

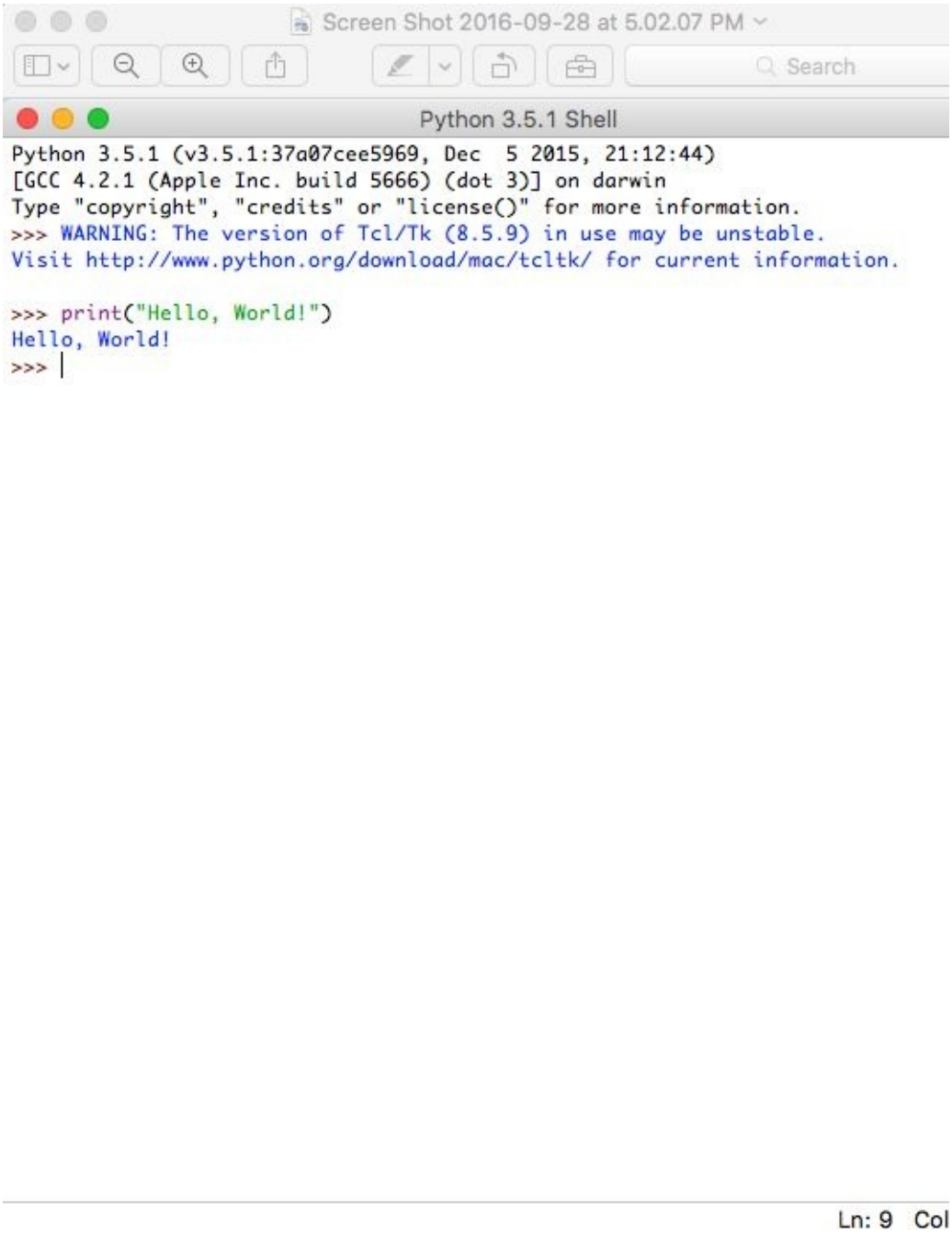
This program is called the interactive shell. You can type Python code directly into the interactive shell and the shell prints the results. At the prompt (>>>) t ype:

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

and press enter.

IDLE might reject code that is copied from Kindle , other eBooks or word processors like Microsoft Word. If you copy and paste code and get an unexplainable error message, try typing the code directly into the window . You must type the code **exactly** as it is written in the example, including quotation marks, parentheses and any other punctuation.

The interactive shell will respond by printing Hello, World!



In the programming world, when you teach someone a new programming language, it is tradition that the first program you teach them is how to print Hello, World! So, congratulations! Y ou just wrote your first program.

Saving Programs

The interactive shell is useful for quick computations, testing small bits of code and writing short programs you don't plan on using again. You can also use IDLE to save a program for reuse. Start the IDLE application, click "File" (in the menu bar on the top left of the IDLE editor) then select "New File." This will open up a t ext editor , which usually has a blank white background . You can write your code in this text editor, then save it to run later.

When you run your code, the output of the code you wrote will appear in the interactive shell. You need to save your changes while editing code before running it again. Type the Hello, World! program into the text editor:

Go to “File” again and select “Save As.” Name your file “hello_world.py” and save it. Python files have to end with .py. Once you’ve saved your file, click “Run” (again, in the menu bar in the top left corner of the IDLE editor) and select “ Run Module .” Alternatively, you can press the F5 key command, the equivalent of selecting “Run Module” from the menu bar. Hello World! will print in the interactive shell, as if you had typed th is line of code. However, in this case, since you saved your program, you can run it as many times as you like.

The program you created is simply a file with a .py extension, located on your computer wherever you saved it. The name I chose for the file—“hello_world.py”—is completely arbitrary, you can name the file anything. Like this example, writing programs in

Python simply involves typing text into files and running them using the interactive shell. Easy, right?

Running Example Programs

Throughout the book, I will give examples of code and the results that print out when you run it. Whenever I do this, you should enter the code and run it yourself.

Sometimes you should type the code in the interactive shell, and sometimes you should type the code into a new .py file. I will explain how you will know where to run your code in the next chapter.

The reason for this is short examples are best run using the shell, and the text editor is better for longer programs you want to save and edit. In the interactive shell, if you make a mistake in your code—a typo for example—and the code doesn't work, you have to type everything again. Using the text editor lets you save your work, so if you make a mistake, you simply edit the mistake and rerun the program.

Another reason the distinction is important is the output of a program running from a file versus the shell can be slightly different. If you type 100 into the interactive shell and press enter, the interactive shell will output 100 . If you type 100 into a .py file and run it, there will be no output. This difference can cause confusion, so be mindful of where you are running a program from if you do not get the same output as the example.

Vocabulary

programming : Writing instructions for a computer to execute.
code : The instructions programmers write for a computer to execute.
low-level programming language : A programming language closer to being written in binary (0s and 1s) than a high-level programming language.
assembly language : A type of difficult to read programming language.
high-level programming language : A programming language that reads more like English than a low-level programming language.
Python : The easy to read, open-source programming language you will learn to use in this book. Created by Guido van Rossum and named after the British sketch comedy group, *Monty Python's Flying Circus*.

Challenge

Try to print something other than Hello, World! .

Chapter 3. Introduction to Programming

“It’s the only job I can think of where I get to be both an engineer and an artist. There’s an incredible, rigorous, technical element to it, which I like because you have to do very precise thinking. On the other hand, it has a wildly creative side where the boundaries of imagination are the only real limitation.”
—Andy Hertzfeld

Our first program printed Hello, World ! . Let’s print it a hundred times. Type the following code into the interactive shell (print needs to be indented by exactly four spaces):

```
for i in range ( 100 ):
    print ( "Hello, World!" )
```

Your shell should print Hello, World! a hundred times. Even though you will probably never need to print Hello, World! a hundred times, this example quickly shows you how powerful programming is. Can you think of anything else you can do a hundred times so easily? I can’t. That is the power of programming.

Examples

Throughout the book, you will see “>>” after each programming example. This represents the output of the program (printed in the interactive shell). Ellipses (...) mean “and so on.” T he previous program will be written like this if it is meant to be typed into the interactive shell:

```
for i in range ( 100 ):
    print ( "Hello World" )
```

```
>> Hello World
>> Hello World
>> Hello World
...
```

And like this if it should be run from a .py file:

```
# https://goo.gl/rKQedq

for i in range ( 100 ):
    print ( "Hello World" )
```

```
>> Hello World
```

>> Hello World
>> Hello World
…

The difference between the two examples is the addition of # https://goo.gl/8eou3Z at the top of the program . This is called a *comment* (explained in the next section). The URL (https://goo.gl/8eou3Z) will take you to a web page that contains the code from the example so you can easily copy and paste it into the IDLE text editor if you are having problems getting the code to run. Whenever you see a comment like this, you should run the code in the example f rom a .py file.

In both examples, the color-coded text represents code, and all the text that comes after ">>" represents the output of the interactive shell. A nything written in Courier New font represents some form of code or code output. For example, if I refer to the word for in the previous program it will be written in the Courier New font .

Courier New is a type of fix-width (non-proportional) font often used to display programming text because each character has the same width, so indentation and other display characteristics of code alignment are easier to observe.

Comments

A *comment* is a line (or part of a line) of code written in English (or another language) preceded by a special symbol that tells the programming language you are using to ignore that line (or part of a line) of code. In Python, the pound symbol is used to create comments.

A comment explains what a line of code does. Programmers use comments to clarify why they did something to make the line of code easier to understand for whoever reads it. You can write whatever you want in a comment, as long as it is only one line long. Here is an example of a comment taking up an entire line of code, followed by an example of a comment taking up part of a line of code:

```
# This is a comment
print ( "The comment does not affect this code" )
```

```
>> The comment does not affect this code
```

```
print ( "The comment does not affect this code" ) # This is a comment
```

```
>> The comment does not affect this code
```

You should only write a comment if you are doing something unusual in your code , or to explain something that is not obvious in the code itself. Use comments sparingly —do not comment on every line of code you write—save them for special situations. Here is an example of an unnecessary comment:

```
print ( "Hello, World!" )  # this line of code prints Hello, World!
```

It is unnecessary because it is very clear what this line of code does. Here is an example of a good comment:

```
d = math.sqrt(l**2 + w**2)  # length of the diagonal
```

Even if you understood exactly how this code works (and you will be the end of Part I), you still might not know how to calculate the length of a diagonal of a rectangle, which is why this comment is useful.

Printing

We are not limited to printing “Hello, World!” in our programs. We can print whatever we’d like as long as we surround it with quotes:

```
print ( "Python" )
>> Python
print ( "Hola!" )
>> Hola!
```

Lines

Python p rograms are divided into lines of code. Take a look at this program:

```
# line1
# line2
# line3
```

There are three lines of code. It is useful to refer to each piece of code by the line it is on. In IDLE you can go to “Edit” and select “Go to line” to jump to a specific line in your program. Sometimes a piece of code is long and takes up more than one line. When this happens, it is ok for the code to extend to the next line (when extended according to Python's rules) :

```
print ( """This is a really really really really really really
        really really long line of code.""" )
```

I will explain the rules for extending lines of Python code in Part V.

Keywords

The Python language has a list of words with special meaning. These are called *keywords* . for is a keyword we've already seen that is used to execute code multiple times. We will learn more keywords throughout this chapter.

Spacing

Let's take another look at our program that prints Hello, World! a hundred times:

```
# https://goo.gl/rKQedq

for i in range ( 100 ):
    print ( "Hello, World!" )
```

As I noted earlier, print is indented by four spaces. We will cover why shortly, but it has to do with letting Python know when blocks of code begin and end. In the meantime, please be aware that whenever you see an indent in an example it is an indent of four spaces. Without proper spacing, your program will not work.

Spacing is not used like this in other programming languages. Other approaches include using keywords or brackets instead of spacing. Python proponents believe the required use of proper spacing makes Python less tedious to read and write than other languages, but the best approach to this problem is frequently debated amongst programmers.

Data Type s

Different kinds of data in Python are grouped into different categories, or *data types* . In Python, each data value, like 2 or "Hello, World!" , is called an *object* . We will learn more about objects in Part II, but for now think of an object as a data value in Python with three properties: an identity, a data type and a value. The identity of an object is where it is stored in memory, which never changes (and which we will be ignoring for now). The data type of an object is the category of data the object belongs to, and determines the properties the object has. This also never changes. The value of an object is the data it represents—the number 2, for example, has a value of 2.

"Hello, World!" is an object with a data type called *str* , short for string, and the value "Hello, World!" . When we refer to an object with a str data type, we call it a string. A string is

a sequence of one or more characters surrounded by quotes. You can use single quotes or double quotes, but the quotes at the beginning and end of a given string must match:

```
“Hello, World!”
>> Hello, World!

‘Hello, World!’
>> Hello, World!
```

A character can be any symbol found in a unicode character table like the interactive one found at http://unicode-table.com/en (I explain what unicode is in Part IV). Strings are used to represent text, and they have unique properties.

The numbers we used to do math in the previous section are also objects—but they are not strings, they have a different data type. Whole numbers have the data type *int* , short for integer. Numbers like 2 , 3 , 4 and 10 all have the data type int. Another way of saying this is they are all integers. Like strings, integers have their own properties. For example, you can divide two integers, but you cannot divide two strings.

Although whole numbers are integers, fractional numbers (numbers with a decimal point) have a different data type called *float* . 2.1 , 8.2 and 9.9999 are all examples of objects with the float data type. They are called floats. Like all data types, floats have their own properties and behave in a certain way. Floats behave similarly to integers:

```
2.2 + 2.2
>> 4.4
```

But , there are some important differences between floats and integers you will learn about later.

Objects with a *bool* data type have a value of either True or False and are called booleans:

```
True
>> True

False
>> False
```

Objects with a data type *NoneType* always have the value None . Objects with a NoneType data type are used to represent the absence of a value:

```
None
>>None
```

I will explain how the different data types are used in programming throughout this chapter.

Constants and Variables

You can use Python to do math just like you would use a calculator. You can add, subtract, divide, multiply, raise a number to a power and much more. Remember to type all of the examples in this section into the shell.

```
 2 + 2
>> 4

2 - 2
>> 0

4 / 2
>> 2

2 * 2
 >> 4
```

A *constant* is a value that never changes. Each of the numbers in the previous example is a constant: the number two will always represent the value 2. A *variable* , on the other hand, refers to a value; but that value can change. A variable consists of a name made up of one or more characters . That name gets assigned to a value using the *assignment operator* (the = sign). Unlike a constant, the value of a variable value can change.

Some programming languages require the programmer to include variable "declarations" that tell the programming language what data type the variable will be. Python makes it simpler: you create a variable simply by assigning a value to it using the assignment operator. Here is an example of creating a variable:

```
 b = 100
b

>> 100
```

Here is how we can change the value of a variable:

```
 x = 100
x

x = 200
x


>> 100
>> 200
```

We can also use two variables to perform arithmetic operations:

```
 x = 10
y = 10
z = x + y
z
a = x - y
a

>> 20
>> 0
```

Often when programming, you want to *increment* (increase) the value of a variable, or *decrement* (decrease) the value of a variable. Because this is such a common operation, Python has a special syntax—a shortcut—for incrementing and decrementing variables. To increment a variable, you assign the variable to itself, and on the other side of the equals sign you add the variable to the number you want to increment by:

```
 x = 10
x = x + 1
x


>> 11
```

To decrement a variable, you do the same thing, but instead subtract the number you want to decrement by:

```
 x = 10
x = 10 - 1
x

>> 9
```

These examples are perfectly valid, but there is a shorter method for incrementing and decrementing variables you should use instead:

```
 x = 10
x += 1
x


>> 11

x = 10
x -= 1
x

>> 9
```

Variables are not limited to storing integer values—they can refer to any data type:

```
 my_string = “Hello, World!”
>>

my_float = 2.2
>>

my_boolean = True
>>
```

You can name variables whatever you’d like, as long as you follow four rules:

0. Variables can’t have spaces. If you want to use two words in a variable, put an underscore between them: i.e., my_variable = “A string!”
0. Variable names can only contain letters, numbers and the underscore symbol.
0. You cannot start a variable name with a number. Although you can start a variable with an underscore, it has a special meaning that we will cover later , so avoid using it until then.
0. You cannot use Python keywords for variable names. You can find a list of keywords here: http://zetcode.com/lang/python/keywords

Syntax

Syntax is the set of rules, principles, and processes that govern the structure of sentences in a given language, specifically word order . [70] The English language has a syntax, and so does Python. For example, in English you must end a sentence with a period. Python also has a set of rules that must be followe d, so it also has syntax.

In Python, strings are always surrounded by quotes. This is an example of Python’s syntax. The following is a valid Python program:

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

It is a valid program because we followed Python’s syntax by using quotes around our text when we defined a string. If we only used quotes on one side of our text we would violate Python’s syntax, and our code would not work. Python does not like when you violate its syntax, and in the next section you find out what happens when you do.

Errors and Exceptions

If you write a Python program and disregard Python's syntax you will get one or more errors when you run your program. When this happens, the Python shell will inform you your code did not work, and gives you information about the error that occurred. Look at what happens if you try to define a string in Python with a quote on only one side:

```
my_string = “Hello World.

>> File "/Users/coryalthoff/PycharmProjects/se.py", line 1
my_string = 'd
          ^
SyntaxError: EOL while scanning string literal
```

This is called a *syntax error* . Syntax e rrors are fatal. A program cannot run with a syntax error . When you run a program with a syntax error, Python lets you know about it in the shell. The message will tell you what file the error was in, what line it occurred on and what kind of error it was. Although this error may look intimidating, errors like this happen all the time.

When there is an error in your code, you should go to the line number the problem occurred on, and try to figure out what you did wrong. In this example, you would go to line 1 of your code. After staring at it for a while, you would eventually notice there is only one quote. To fix it, add a quote at the end of the string and rerun the program. From this point forward, I will represent the output of an error like this:

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

For easier reading, I will only show the last line of the error.

Python has two kinds of errors: syntax errors and exceptions. Any error that is not a syntax error is an *exception* . A ZeroDivisionError is an example of an exception that occurs if you try dividing by zero.

The difference between a syntax error and an exception is exceptions are not necessarily fatal (there is a way to make a program run even if there is an exception which you will learn about in the next chapter). When an exception occurs, Python programmers say “ Python (or your program) raised an exception”. Here is an example of an exception:

```
10 / 0

>> ZeroDivisionError: integer division or modulo by zero
```

If you incorrectly indent your code, you get an IndentationError :

```
y = 2
      x = 1

>> IndentationError: unexpected indent
```

As you are learning to program, you will frequently get syntax errors and exceptions (including ones we did not cover), but they will decrease over time. Remember, when you run into a syntax error or exception, go to the line where the problem occurred and stare at it until you figure out the solution (or search the internet for the error or exception if you are stumped).

Arithmetic Operators

Earlier we used Python to do simple arithmetic calculations like 4 / 2 . The symbols we used in those examples are called *operators* . Operators in Python are divided into several categories, and the ones we've seen so far are called *arithmetic operators* . Here is a list of some of the most common arithmetic operators in Python:

Operator	Meaning	Example	Evaluates to
**	Exponent	2 ** 2	4
%	Modulo/remainder	14 % 4	2
//	Integer division/floored quotient	13 // 8	1
/	Division	13 / 8	1.625
*	Multiplication	8 * 2	16
-	Subtraction	7 - 1	6
+	Addition	2 + 2	4

When two numbers are divided there is a quotient and a remainder. The quotient is the result of the division and the remainder is what is left over. The modulo operator returns the remainder when two numbers are divided:

```
 2 % 2
>> 0

 10 % 3
>> 1
```

There are two different operators for division. The first is //, which returns the quotient:

```
 14 // 3
>  4
```

The second is /, which returns the result of the first number divided by the second as a floating point number:

```
14 / 3
> 4.666666666666667
```

You can raise a number by an exponent with the exponent operator:

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

The values (in this case numbers) on either side of an operator are called *operands* . Together, two operands and an operator form an *expression* . When your program runs, Python evaluates each expression, and returns a single value. When you type an expression like 2+2 into the Python shell, the expression is evaluated to 4 .

The *order of operations* is a set of rules used in mathematical calculations to evaluate an expression. Remember Please Excuse My Dear Aunt Sally ? It is an acronym used to help you remember the order of operations in math equations: parentheses, exponents, multiplication, division, addition and subtraction. Parentheses outrank exponents, which outrank multiplication and division, which outrank addition and subtraction. If there is a tie among operators, like in the case of 15 / 3 × 2 , you evaluate from left to right:. In this case the answer is the result of 15 divided by 3 times 2. Python follows t he same order of operations when it evaluates mathematical expressions:

```
2 + 2 * 2
>> 6

(2+2) * 2
>> 8
```

In the first example, 2 * 2 is evaluated first, followed by 2 + 2 , because multiplication takes precedence over addition. In the second example, (2+2) is evaluated first, because expressions in parentheses are always evaluated first .

Comparison Operators

Comparison operators are another category of operators in Python. Comparison operators are similar to arithmetic operators; they are operators used in expressions with operands on either side. Unlike expressions with arithmetic operators, expressions with comparison operators evaluate to either True or False .

Operator	Meaning	Example	Evaluates to
>	Greater than	100 > 10	True

<	Less than	100 < 10	False
>=	Greater than or equal	2 <= 2	True
<=	Less than or equal	22 / 8	2.75
==	Equal	3 * 5	15
!=	Not equal	5 - 2	3

An expression with the > operator returns the value True if the number on the right is bigger than the number on the left, and False if it is not:

```
100 > 10
>> True
```

An expression with the < operator returns the value True if the number on the left is smaller than the number on the right, and False if it is not:

```
100 < 10
>> False
```

An expression with the >= operator returns the value True if the number on the right is bigger than or equal to the number on the left. Otherwise the expression returns False :

```
2 >= 2
>> True
```

An expression with the <= operator returns the value True if the number on the right is smaller than or equal to the number on the left. Otherwise the expression returns False :

```
2 <= 2
>> True
```

An expression with the == operator returns the value True if the two operands are equal, and False if not:

```
2 == 2
>> True
```

```
1 == 2
>> False
```

Whereas an expression with the != operator check returns True if the two operands are not equal, and False otherwise:

```
1 != 2
>> True
```

```
2 != 2
>> False
```

Earlier, we assigned variables to numbers, like x = 100 using = . It may be tempting to read this in your head as “x equals 100,” but don’t. As we saw earlier, = is used to assign a value to a variable, not to check for equality. When you see x = 100 , you should think “x gets one hundred.” The comparison operator == is used to check for equality, so if you see x == 100 , then you should think “x equals 100”.

Logical Operators

The last category of operators we will cover are called *logical operators* . Expressions with logical operators are similar to expressions with comparison operators in that they both evaluate to True or False .

Operator	Meaning	Example	Evaluates to
and	and	True and True	True
or	or	True and False	True
not	not	not True	False

The Python keyword and takes two expressions and returns True if all the expressions evaluate to True . If any of the expressions are False , it returns False .

```
1 == 1 and 2 == 2
>> True
```

```
1 == 2 and 2 == 2
>> False
```

```
1 == 1 and 2 == 1
>> False
```

```
2 == 1 and 1 == 1
>> False
```

You can use the and keyword multiple times in one statement:

```
1 == 1 and 10 != 2 and 1 % 1 != 3 and 2 < 10
>> True
```

When Python evaluates a statement with the and keyword it stops evaluating the statement as soon as an expression evaluates to False . Python does not evaluate the remaining expressions because it is going to return False regardless of how the rest of the expressions evaluate (only one expression has to evaluate to False for False to get returned for the entire statement) . Understanding this behavior can help you write more efficient code. For example, if one of your expressions takes a long time to evaluate, you should put it last, because it will be less likely to be evaluated there.

The keyword or takes two or more expressions and evaluates to True if **any** of the expressions evaluate to True :

```
1==1 or 1==2
>> True

1==1 or 2==2
>> True

1==2 or 2==1
>> False

2==1 or 1==2
>> False
```

Like and , you can use multiple or keywords in one statement:

```
1==1 or 1==2 or 1==3 or 1==4 or 1==5
>> True
```

The result of the example above is True because 1==1 is True even though all the rest of the expressions evaluate to False . Like and , when evaluating statements with or in them, as soon as one expression evaluates to True , Python stops evaluating the rest of the expressions because it is going to return True regardless of how they evaluate.

You can place the keyword not in front of an expression, and it will change the result of the expressions evaluation to the opposite of what it would have otherwise evaluated to . If the expression would have evaluated to True , it will evaluate to False when preceded by not , and if it would have evaluated to False , it will evaluate to True when preceded by not .

```
not 1 == 1
>> False

not 1 == 2
>> True
```

Conditional Statements

Th e keywords if , elif and else are used in *conditional statements* . Conditional statements are a type of *control structure* : a block of code that can make decisions by analyzing the values of variables. A conditional statement is code that is able to execute additional code circumstantially. Here is an example in pseudocode (fake code used to illustrate an example) to help clarify how this works:

```
If (expression) Then
        (code_area1)
Else
        (code_area2)
```

This pseudocode explains that you can define two conditional statements that work together. If the expression defined in the first conditional statement is True , all of the code defined in code_area1 gets executed. If the expression defined in the first conditional statement is False , all of the code defined in code_area2 gets executed. The first part of the example is called an *if statement* , and the latter is called an *else statement* . Together, they form an if else statement: a way for programmers to say “if this happens do this, otherwise do that”. Here is an example of an if else statement in Python:

```
# https://goo.gl/uYp6ha

country = "America"
if country == "America" :
    print ( "Hello, America!" )
else :
    print ( "Hello, Canada!" )
```

```
>> Hello, America!
```

Lines 2 and 3 form an if statement. An if statement is made up of a line of code starting with the if keyword followed by an expression, a colon, an indentation and one or more lines of code to be executed if the expression in the first line evaluates to True . Lines 3 and 4 form an else statement. An else statement starts with the else keyword, followed by a colon, an indentation and one or more lines of code to execute if the expression in the if statement evaluates to False .

Together they form an if else statement. This example prints Hello, America! because the expression in the if statement evaluates to True . If we change the variable country to Canada , the expression in the if statement will evaluate to False , the else statement’s code will execute and our program will print Hello Canada! instead.

```
# https://goo.gl/bd4LVW

country = "Canada"
```

```
if country == "America" :
    print ( "Hello, America!" )
else :
    print ( "Hello, Canada!" )
```

>> Hello Canada!

An if statement can be used on its own:

```
# https://goo.gl/jOlzVl

country = "America"
if country == "America" :
    print ( "Hello, America!" )
```

>> Hello, America!

You can have multiple if statements in a row:

```
# https://goo.gl/WBoKWA

x = 2
if x == 2 :
    print ( "The number is 2." )
if x % 2 == 0 :
    print ( "The number is even." )
if x % 2 != 0:
    print ( "The number is not odd." )
```

>> The number is 2.
>> The number is even.

Each if statement will execute its code only if its expression evaluates to True . In this case, the first two expressions evaluate to True , so their code is executed, but the third expression evaluates to False , so its code is not executed.

If you really want to get crazy, you can even put an if statement inside of another if statement. This is called nesting:

```
# https://goo.gl/S6Z9rp

x = 10
y = 11

if x == 10 :
    if y == 11 :
        print (x + y)
```

>> 21

In this case x + y will only print if the expressions in both if statements evaluate to True . An else statement cannot be used on its own; it can only be used at the end of an if else statement.

We use the elif keyword to create *elif statements* . elif stands for else if, and elif statements can be indefinitely added to an if else statement to allow it to make additional decisions.

If an if else statement has elif statements in it, the if statement expression gets evaluated first. If the expression in that statement evaluates to True , its code is executed and no other code is executed. However, if it evaluates to False , each consecutive elif statement is evaluated. As soon as an expression in an elif statement evaluates to True , its code is executed and no more code executes. If none of the elif statements evaluate to True , the code in the else statement is executed. Here is an example of an if else statement with elif statements in it:

```
# https://goo.gl/L0OorN

country = "Thailand"
if country == "Japan" :
    print ( "Hello, Japan!" )
elif country == "Thailand" :
    print ( "Hello, Thailand!" )
elif country == "India" :
    print ( "Hello, India!" )
elif country == "China" :
    print ( "Hello, China!" )
else :
    print ( "Hello, world!" )

>> Hello, Thailand!
```

Here is an example where none of the expressions in the elif statements evaluate to True , and the code in the else statement is executed:

```
# https://goo.gl/Qwb5OD

country = "Mars"
if country == "America" :
    print ( "Hello, America!" )
elif country == "Canada" :
    print ( "Hello, Canada!" )
elif country == "Thailand" :
    print ( "Hello, Thailand!" )
elif country == "Mexico" :
    print ( "Hello, Mexico!" )
else :
```

```
    print ( "Hello, world!" )
```

```
>> Hello, World!
```

Finally, you can have multiple if statements and elif statements in a row:

```
# https://goo.gl/tgcmXN

x = 100
if x == 10 :
    print ( "10!" )
elif x == 20 :
    print ( "20!" )
else :
    print ( "I don't know what x is!" )

if x == 100 :
    print ( "x is 100!" )

if x % 2 == 0:
    print ( "x is even!" )
else :
    print ( "x is odd!" )
```

```
>>  I don't know what x is!
>>  x is 100!
>>  x is even!
```

Statements

A *statement* is a technical term that describes various parts of the Python language. You can think of a Python statement as a command or a calculation. This will become more clear as we take a look at examples of different kinds of Python statements . In this section we will also take a detailed look at the syntax of statements. Don't worry if some of this seems confusing at first. Spend as much time as necessary rereading this section and go back to the earlier examples and compare them with what you are learning here. It might take a while to understand the syntax of statements, but it will start to make more sense the more time you spend practicing Python and will help you understand several programming concepts.

Python has two kinds of statements: *simple statements* and *compound statements* . Simple statements can be expressed in one line of code, whereas compound statements generally span multiple lines (but can be written in one line in some circumstances). Here are some examples of simple statements:

```
print ( "Hello, World!" )
```
>> ‘Hello, World!”

```
2 + 2
```
>> 4

Compound statement *s* generally span more than one line of code. You’ve already seen multiple examples of compound statements: if statements, if else statements and the first program we wrote in this chapter that printed “Hello, World!” one hundred times are all examples of compound statements.

Compound statements are made up of one or more *clause* . A clause consists of two or more lines of code: a *header* followed by a *suite* (s). A header is a line of code in a clause that contains a keyword followed by a colon and a sequence of one or more lines of indented code. After the indent there are one or more suites. A suite is just a line of code in a clause. The suites are controlled by the header. Our program that prints “Hello, World!” a hundred times is made up of a single compound statement:

```
# https://goo.gl/rKQedq
for i in range ( 100 ):
    print ( "Hello, World!" )
```

>> Hello World
>> Hello World
>> Hello World
…

The first line of the program is the header. It’s made up of a keyword— for —followed by a colon and an indent ed line of code. After the indentation is a suite — print(“Hello, World!”) . In this case, the header uses the suite to print Hello, World! a hundred times. This is called a loop, which you learn more about in Chapter 7. This code only has one clause.

A compound statement can also be made up of multiple clauses. You already saw this with if else statements. Anytime an if statement is followed by an else statement, the result is a compound statement with multiple clauses. When a compound statement has multiple clauses, the header clauses work together. In the case of an if else compound statement, when the if statement evaluates to True , the if statement’s suites execute and the else statement’s suites do not execute. When the if statement evaluates to False , the if statement’s suites do not execute and the else statement’s suites execute instead. The last example from the previous section has three compound statements:

```
# https://goo.gl/tgcmXN

x = 100
if x == 10 :
    print ( "10!" )
elif x == 20 :
```

```
    print ( "20!" )
else :
    print ( "I don't know what x is!" )

if x == 100 :
    print ( "x is 100!" )

if x % 2 == 0:
    print ( "x is even!" )
else :
    print ( "x is odd!" )
```

```
>>  I don't know what x is!
>>  x is 100!
>>  x is even!
```

The first compound statement has three clauses, the second compound statement has one clause and the last compound statement has two clauses.

One last thing about statements, statements can have spaces between them. Space between statements does not affect the code. Sometimes spaces are used between statements to make code more readable:

```
print ( "Michael" )
print ( "Jordan" )
```

```
>> Michael
>> Jordan
```

Vocabulary

comment : A line (or part of a line) of code written in English (or another language) preceded by a special symbol that lets the programming language you are using know it should ignore that line (or part of a line) of code.
keyword : A word with a special meaning in Python. You can see all of Python's keywords at http://zetcode.com/lang/python/keywords
constant : A value that never changes.
variable : A name assigned to a value using the assignment operator
assignment operator : The = sign.
increment : Increasing the value of a variable.
decrement : Decreasing the value of a variable.
data type : A category of data.
object : A data value in Python with three properties: an identity, a data type and a value.

string : An object with a data type str. Its value is a sequence of one or more characters surrounded by quotes.
integer : An object with a data type int. Its value is a whole number.
float : An object with a data type float. Its value is a fractional number.
boolean : An object with a data type bool. Its value is either True or False .
nonetype : An object with a data type NoneType. Its value is always None .
syntax : T he set of rules, principles, and processes that govern the structure of sentences in a given language, specifically word order . 70
syntax error : A fatal programming error caused by violating a programming language's syntax.
exception : A nonfatal programming error.
operator : Symbols used with operands in an expression.
arithmetic operator : A category of operators used in arithmetic expressions.
operand : A value on either side of an operator.
expression : Code with an operator surrounded by two operands.
order of operations : A set of rules used in mathematical calculations to evaluate an expression.
comparison operator : A category of operators used in expression that evaluate to either True or False .
logical operator : A category of operators that evaluate two expressions and return either True or False
control structure : A block of code that makes decisions by analyzing the values of variables
conditional statement : Code that is able to execute additional code circumstantially.
if else statement : A way for programmers to say "if this happens do this, otherwise do that."
if statement : The first part of an if else statement.
else statement : The second part of an if else statement.
elif statement : S tatements that can be indefinitely added to an if else statement to allow it to make additional decisions.
statement : A command or a calculation.
simple statement : A statement that can be expressed in one line of code
compound statement : A statements that generally spans multiple lines (but can be written in one line in some circumstances).
clause : The building blocks of compound statements. A clause is made up of two or more lines of code: a *header* followed by a *suite* (s) .
header : A header is a line of code in a clause containing a keyword followed by a colon and a sequence of one or more lines of indented code.
suite : A line of code in a clause controlled by a header.

Challeng e

Write a program with a variable called age assigned to an integer that prints different strings depending on what integer age is.

Chapter 4. Functions

“Functions should do one thing. They should do it well. They should do it only.”
— Robert C. Martin

A function is a compound statement that can take input, execute instructions, and optionally return an output. Calling a function means giving the function the input it needs toso it can execute its instructions and optionally return an output. Functions in Python are similar to functions in math. If you don’t remember functions from algebra, here is an example:

```
# the following is algebra, not Python code
 f(x) = x * 2
```

The left half of the equation defines a function f —that takes one parameter— x . A parameter is data passedyou pass into a function when its called. A function can have one parameter, multiple parameters or no parameters. A function’s parameters are its input.

The right half of the equation is the definition of the function (the instructions it executes when called). The right hand side of the equation usesis able to use the parameter that was passed in (x) to make a calculation and return the result (the output). In this case, our function takes a parameter (x); , multiplies it by 2; and returns the result.

In both Python and algebra, you call a function with the following syntax: [function_name]([parameters_seperated_by_commas]) . You call a function by putting parentheses after the function name. TYou put the parameters go inside the parenthesis with each parameter separated by a comma. If we call the function we just defined above (?) and pass in the parameter 4 , we get the following result:

```
# the following is algebra, not Python code
 f(4)
>> 8
```

If we pass in 10 as a parameter we get 20:

```
f(10)
>> 20
```

In our first example, we called our function with 4 as a parameter (x). Our function executed its code, x * 2 , and returned the result — 8 . In our second example, we called our function with 10 as a parameter (x). Our function executed its code, x * 2 , and returned the result — 20 .

Representing Concepts

This is abrupt, maybe something like “As you may have figured out by now…”We are not limited to printing “Hello, World!” in our programs. We can print whatever we’d like:

```
print(“Python!”)

>> Python!
```

From here on out, I will use the following convention to explain concepts like the idea that you can print anything in Python: print(“[what_you_want_to_print]”) . The brackets and the text inside of them represent that you need to be replaced by a piece of codesubstitute a piece of code in place of them.
When you are trying to follow an example written in this formatlike this, do not type the brackets. The words inside of the brackets are a hint for the code you need to replace the brackets with. Everything outside of thenot in brackets represents actual code that you should type. This format is a way of expressing that you can type print(“[what_you_want_to_print]”) into Python, replace [what_you_want_to_print] with whatever you want to print, and Python will print it:

```
print(“I do not like green eggs and ham.”)
print(“The Cat in the Hat”)

>> I do not like green eggs and ham
>> The Cat in the Hat
```

Programming is full of conventions: agreed upon ways of doing things. This format is an example of a convention that is used in the programming world and will be used throughout the book.

Defining Functions

To create a function in Python we choose a function name, define its parameters, define what the function will do, and we can choose to optionally return a value (if we don’t return a value the function will return None). We use the following syntax to define a function:

```
def [function_name]([parameters_seperated_by_commas]):
    [function_definition]
```

Our mathematical function f(x) = x * 2 looks like this in Python:

```
# https://github.com/calthoff/tstp/blob/master/part_I/functions/df_ex1.py
```

```
def f(x):
    return x * 2

>>
```

The keyword def tells Python you are about to define a function. is a keyword used to define a function. When you use it, Python knows you are about to define one. After def , you can name your function anythingwhatever you'd like. By convention, you should never use capital letters in ayour function name,s and if there are two words in your function name you should separate them with an underscore—like_this. Once you've named your function, put parentheses after it. Inside the parentheseis, you put your parameter(s). In this (the previous?) example, our function only has one parameter (x), but if you want your function to accept more than one parameter, you must separate each parameter inside the parentheseis with a comma. After the parentheseis you put a colon and indent by four spaces (like any other compound statement). Any code indented four spaces after the colon is the function's definition. In this case, our function's definition is only one line— return x * 2 . return is another keyword. It is used to define the value a function outputs when you call it, referred to as the value the function returns.

To call a function in Python, we use the syntax we learned earlier: [function_name]() . Here is an example of calling our function with 2 as a parameter:

```
f(2)
>>
```

You will notice the console didn't print anything. Our function worked, but it just didn't print the result because we didn't tell Python to. If we want to print the value our function returned, we can save our function's output in a variable:

```
# https://github.com/calthoff/tstp/blob/master/part_I/functions/df_ex3.py
result = f(2)
print(result)

>> 4
```

You can save the result your function returns in a variable whenever you need to use the value later in your program.

Functions are not onlyjust used to return values. Returning a value is optional, as is including a return statement in your function. Aside from returning values, functions also encapsulate functionality you want to reuse. For example:

```
#add github
def even_odd(x):
  if x % 2 == 0:
     print("even")
  else:
```

```
        print("odd")

    even_odd(2)
    even_odd(3)
```

```
>> even
>> odd
```

We didn't define a value for our function to return, but our function is still useful. ItOur function tests if x % 2 == 0 and prints whether x is even or odd depending on the result. Remember, modulo returns the remainder when you divide two numbers. If there is no remainder when you divide a number by 2 (modulo returns 0) the number is by definition even. If there is a remainder, the number is odd. You may need to use this functionality in several different places in your program. It would be poor programming to type the code we used in our function's definition every time you want to use this functionality. That's what functions are for. You put functionality in a function, and it lets you easily reuse that functionality throughout your program, without having to do any extra work. This is a little confusing - are you going to explain later on? You could mention this is something you'll cover how to do in Chapter __.

Parameters

So far, we've only defined functions that accept one parameter. A function doesn't have to take any parameters. Here is an example of a function that does not take any parameters:

```
# https://github.com/calthoff/tstp/blob/master/part_I/functions/df_ex5.py

def f():
  return 1 + 1

result = f()
print(result)
```

```
>> 2
```

Here is an example of a function that accepts multiple parameters:

```
# https://github.com/calthoff/tstp/blob/master/part_I/functions/df_ex4.py

def f(x, y, z):
   return x + y + z

result = f(1, 2, 3)
print(result)
```

>> 6

There are two types of parameters a function can accept. The parameters we've seen so far are called required parameters. When a function is called, all of the required parameters must be passed into the function, or you will get an error in your program. There is another kind of parameter—optional parameters—that let the caller of the function pass in a parameter if they want to, but they do not have to. If they do not pass in a parameter, a default value defined by the function will be used instead. OYou define optional parameters are defined with the following syntax: [function_name]([parameter_name]=[paramater_value]) . Like required parameters, optional parametersthey must be separated by commas. Here is an example of a function that takes an optional parameter:

```
# https://github.com/calthoff/tstp/blob/master/part_I/functions/op_ex1.py

def f(x=10):
    if x == 10:
        print("x is ten")
    else:
        print("x is not ten")

default = f()
pass_in = f(2)

print(default)
print(pass_in)
```

>> 'x is ten' I'm confused why there are single quotes or any quotes at all.
>> 'x is not ten'I'm confused why there are single quotes or any quotes at all.

First, we call our function without passing in a parameter. Because the parameter is optional we don't have to pass it inthis is allowed, and x is assigned the value we defined in our optional parameter— 10 . When our function is called, x is equal to 10 , and so 'x is ten'(if there should be single quotes above, leave it, if not take them out) prints.

Next—we call our function again—but this time we pass in 2 as a parameter. The default value 10 is ignored because we provided a value this time, and so x gets 2 and "x is not ten" (same thing about the quotes, even if I'm just missing something I think they should be single quotes) prints. You can define a function with both required and optional parameters, but there is one rule: all of your required parameters must be defined before your optional parameters:

```
def required_optional(x, y=10)
  return x + y

>>
```

Passing Parameters

When you define a function with parameters, sometimes those parameters have to be a specific data type in order for the function to work. How do you communicate this to whoever calls your function? When you write a function, it is good practice to leave a comment at the top of the function explaining what data type each parameter needs to be. We will discuss this further in Ppart V. When you give a function parameters when you call it, it is referred to as "passing" the function parameters. This sentence makes sense, but is fairly confusing. Could it be: When you give a function parameters THEN you call it, it is referred to as "passing" the function parameters.

pass

We can use the keyword pass to create a function that does nothing:

```
""" https://github.com/calthoff/tstp/blob/master/part_I/functions/pass_ex1.py
"""

def f():
    pass

f()
>>
```

pass The pass keyword is useful whenever you want to create a function, but finish the definition later.

Nested Functions

You can define a function inside of a function. This is called nesting. The first function is called the outer function, and the second function is called the inner (or nested) function, or nested function. Here is an example:

```
def f ():
    print ( "Inner Function!" )

    def x ():
        print ( "Nested Function!" )
```

```
    x()

f()
```

```
>> Inner Function!
>> Nested Function!
```

We will not be covering why this is important in this book because you do not needit is not important to use nested functions when you are learning to program., but I wanted to included this example so you know that it is possible.

Scope

Variables have an important property called scope we didn't discuss when we first covered them. When you define a variable, the variable's scope refers to what part of your program has access to it. This is determined by where the variable is defined in your program. If you define a variable outside of a function (or a class, which we learn about in Part II) the variable has a global scope: the variable can be accessed anywhere in your program. If you define a variable inside of a function (or class) it has local scope: the variable cannot be accessed anywhere in your program—; it can only be accessed in the function (or class) it was defined in (or any nested functions or classes). Here are some examples of variables with global scope:

```
x = 1
y = 2
z = 3

>>
```

These variables were not defined inside of a function (or class) and therefore have a global scope. This means we can access them anywhere—including inside of a function:

```
x = 1
y = 2
z = 3

def f():
    print(x)
    print(y)
    print(z)

f()
```

```
>> 1
```

```
>> 2
>> 3
```

If we define these same variables inside of a function, we can only access them inside of thatthe function we defined them in (or a function nested inside the function we defined them in). If we try to access them outside of the function they were defined in, Python raises an exception:

```
def f ():
    x = 1
    y = 2
    z = 3

print (x)
print (y)
print (z)
```

```
>> NameError: name 'x' is not defined
```

If we stick to using these variables inside our function, there is no problem:

```
def f ():
    x = 1
    y = 2
    z = 3
    print (x)
    print (y)
    print (z)

f()
```

```
>> 1
>> 2
>> 3
```

If you want to change the value of a global variable inside a local scope, you need to use the global keyword followed by the variable you want to change:

```
# https://github.com/calthoff/tstp/blob/master/part_I/functions/scope_ex3.py

x = 100


def f():
    global x
    x += 1
```

```
    print(x)

f()

>> 101
```

The reason programming languages have scope is because having no scope (every variable can be accessed anywhere in a program) causes problems. If you have a large program, and you write a function that uses the variable x , you might accidently change the value of a variable called x that was previously defined in your program; which will change the behavior of your program and may cause an error or unexpected results. The larger your program gets, and the more variables it has, the more likely this becomes. However, with scope, if you define a variable x inside of a function, there is a guarantee you will not accidentally change the value of any previously defined variables outside of your function because in order to change the value of a variable outside of your function, you must explicitly use the global keyword.

Built-in Functions

Python comes with a library of built-in functions. They perform all sorts of different functionality and are ready to use without any work on your part. We've already seen one example of a built-in function—the first program we wrote used the print function to print "Hello, World!" . len is another built in function. It returns the length of an object—like a string. The length of a string is the number of characters in it.

```
len("Monty")
>> 5

len("Python")
>> 6
```

type is another built-in function. It returns what data type an object is:

```
type("Hello World")
>> <type 'str'>

type(100)
>> <type 'int'>

type(1.0)
>> <type 'float'>
```

The built-in str function takes an object and returns a new object with a string data type. For example, we can use str to convert an integer to a string.

```
str(100)
>> '100'
```

int takes an object and returns a new object with an integer data type:

```
int("1")
>> 1
```

And float takes an object and returns a new object with an integer data type:

```
float(100)
>> 100.0
```

input is a built-in function that collects information from the person using our program.

```
"""
https://github.com/calthoff/tstp/blob/master/part_I/introduction_to_programming/input_
"""

age = input("How old are you?")
age = int(age)
if age < 21:
    print("You are young!")
else:
    print("Wow you are old!")

>> How old are you?
```

The input function takes a string as a parameter and displays the string to the person using the program in the shell. They can then type a response into the shell, and we can save their response in a variable—in this case we save the response in the variable age .

Next we use the int function to change age from a string to an integer because input collects data from the user as a string, and we want our variable to be an integer so we can compare it to other integers. Once we have an integer, our if else statement determines which message gets printed to the user, depending on what they typed into the shell. If the user types a number smaller than 21 , "You are young!" prints. If the user types a number greater than 21 , "Wow you are old!" prints.

Exception Handling

When you rely on user input from the input function, you do not control the input to your program—the user does, and that input could cause an error. For example, say we write a program to collect two numbers from a user, and print out the result of the first number divided by the second number:

```
“““
https://github.com/calthoff/tstp/blob/master/part_I/introduction_to_programming/except
”””

a = input(“type a number”)
b = input(“type another number”)
a = int(a)
b = int(b)
print(a / b)

>> type a number
 >> 10
 >> type another number
 >> 5

 >> 2
```

Our program appears to work. However, we will run into a problem if the user inputs 0 as the second number:

```
 a = input(“type a number”)
b = input(“type another number”)
  a = int(a)
     b = int(b)
print(a / b)

>> type a number
 >> 10
 >> type another number
 >> 0

 >> ZeroDivisionError: integer division or modulo by zero
```

Our program works—until the user decides to enter 0 as the second number, in which case our program raises an exception. We cannot allow people to use this program and hope they will not enter 0 as the second number. One way to solve this is to use exception handling, which allows you to “catch” exceptions if they occur and decide what to do.

The keywords try and except are used for exception handling. We can change our program to use exception handling so if a user enters 0 as the second number, our program prints a message telling them not to enter 0 instead of raising an exception.

In Python exceptions are objects—which allows us to use to them in our programs. Each exception in Python is an object. You can see the full list of built-in exceptions here: https://docs.python.org/3/library/exceptions.html. Whenever you are in situation where you think your code may raise an exception, you can use a compound statement with the keywords try and except to catch the exception.

The try clause defines the error that could occur. The except clause defines code that will only execute if the exception defined in your try clause occurs. Here is an example of how we can use exception handling in our program so if a user enters 0 as the second number our program doesn't break:

```
“““
https://github.com/calthoff/tstp/blob/master/part_I/introduction_to_programming/except
”””

a = input(“type a number”)
b = input(“type another number”)
try:
    print(a / b)
except ZeroDivisionError:
    print(“b cannot be zero. Try again.”)
```

```
>> type a number
>> 10
>> type another number
>> 0
>> “b cannot be zero. Try again.”
```

If the user enters anything other than 0 , the code in our try block is executed and our except block doesn't do anything. But if the user enters 0 , instead of raising an exception, the code in our except block is executed and our program prints “b cannot be zero. Try again.” .

Docstrings

Docstrings are comments at the top of a function or method that explain what the function or method does, and documents what types of the parameters should be passed to it. Here is an example:

```
def add (x , y):
    """
   Returns x + y.
    :param  x: int first integer to be added.
    :param  y: int second integer to be added.
```

```
    :return : int sum of x and y.
    """
    return x + y
```

The first line of the docstring clearly explains what our function does. When other developers reuse your function or method, they do not want to have to read through all of your code to figure out what it does. The rest of the lines of the docstring lists the function's parameters, its return value, and some additional information, including the type for all of the parameters and the return value. Docstrings will help you program faster, because if you forget what a piece of code does, you can quickly figure it out by reading the docstring instead of all of the code in a function, class or method. It will also make it much easier for other programmers to use your code. In some cases I've omitted docstrings I normally would have included them to make my code as concise as possible for easy reading—but whenever I am writing code for production (code that is actually used by other people)— I use docstrings.

Challeng e

Write a function that does something interesting, and use it several times in a program.

Chapter 5. Containers

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."
— Linus Torvalds

In chapter 3, we learned how to store objects in variables. In this chapter we learn to store objects in containers—special objects that can store and retrieve other objects (like strings). In this chapter, we will go over three commonly used containers: lists, tuples and dictionaries.

Lists

A list is a mutable container that stores objects in a specific order. When a container is mutable it means the objects in the the container can change—objects can be added and removed from the container.

[image]

Lists are represented in Python with brackets. There are two syntaxes to create a list. We can create an empty list with the list function:

```
new_list = list()
new_list
>> []
```

Or we can create an empty list with brackets:

```
new_list = []
new_list
>> []
```

Both syntaxes create a new empty list. When you create a new list with the list function you can also pass in objects you want to add to your list as parameters:

```
my_list = list (“Apple”, “Orange”, “Pear”)
my_list

>> ['Apple', 'Orange', 'Pear']
```

Or like this using the second syntax:

```
my_list = [“Apple”, “Orange”, “Pear”]
```

```
my_list

>> ['Apple', 'Orange', 'Pear']
```

Each object in a list is called an item in the list. In this example there are three items in our list: ‘Apple’ , ‘Orange’ and ‘Pear’ . Lists keep their items in order—the order the items entered the list. Unless we change the order of our list, ‘Apple’ will always be the first item, ‘Orange’ the second item and ‘Pear’ the third item. ‘ Apple’ is at the beginning of the list, and ‘Pear’ is at the end. We can add a new item to the end of our list using the append function:

```
my_list.append(“Banana”)
my_list.append(“Peach”)
my_list

>> ['Apple', 'Orange', 'Pear', ‘Banana’,’Peach’]
```

Lists are not limited to storing strings—they can store any data type:

```
new_list = []
new_list.append(True)
new_list.append(100)
new_list.append(1.1)
new_list.append(‘Hello’)

>> [True, 100, 1.1, ‘Hello’]
```

E very item in a list has a position in the list—called its index. You can figure out the index of any item in a list by starting at the beginning of the list and counting. The only tricky part is you have to start counting at zero, because the first item in a list has an index of zero. So the first item in a list is at index zero, the second item in a list is index one, and so on. Counting starting at zero takes some getting used to, so don’t worry if it frustrates you at first. You can access each item in a list with its index using the syntax [list_name][[index]] . I put index in two brackets to represent that [index] should be replaced, but should be inside brackets.

```
my_list = [“Apple”, “Orange”, “Pear”]
my_list[0]
my_list[1]
my_list[2]

>> Apple
>> Orange
>> Pear
```

You can change an item in a list by setting the index of the item to a new object:

```
color_list = [“blue”, “green”, “yellow”]
```

```
color_list
color_list[2] = “red”
color_list

>> [“blue”, “green”,  “yellow”]
>> [“blue”, “green”, “red”]
```

If you try to access an index that doesn’t exist, Python will raise an exception:

```
color_list = [“blue”, “green”,  “yellow”]
color_list[4]

>> IndexError: list index out of range
```

You can remove the last item from a list with pop :

```
color_list = [“blue”, “green”, “yellow”]
color_list
item = color_list.pop()
item
color_list

>> [“blue”, “green”,  “yellow”]
>> “yellow”
>>  [“blue”, “green”]
```

You cannot pop from an empty list, if you do Python will raise an exception.
You can check if an item is in a list using the keyword in :

```
color_list = [“blue”, “green”,  “yellow”]
“green” in color_list

>> True
```

Add the keyword not to check if an item is not in a list:

```
color_list = [“blue”, “green”,  “yellow”]
“black” not in color_list

>> True
```

You can get the size of a list (the number of items in it) with the len function:

```
len(color_list)
>> 3
```

Finally, you can get a range of items in a list with slicing. We slice a list with a start index and an end index separated by a colon in brackets outside of our list. Slicing returns a

new list (a “slice” of the old one) made up of everything between the start and end index. The syntax for slicing is [list_name][[start_index:end_index]]. Here is an example of slicing a list:

```
 new_list = ['Apple', 'Orange', 'Pear', ‘Banana’, ‘Peach’]
new_list[0:3]

>> ['Apple', 'Orange', 'Pear']
```

A “gotcha” with slicing is the start index includes the item at that index, but the end index doesn’t include the item at the end index, it only includes the item before the end index. This means if you want to slice from “Apple” to “Pear”, you need to slice from index 0, to index 3 (instead of index 2), because the item at the end index is not included in the slice.

Tuples

A tuple is an immutable container that stores objects in a specific order. When a container is immutable it means the contents of the container cannot change. That means unlike a list, once you put an object into a tuple, you can no longer change it. Once you create a tuple you cannot change the value of any of the items in it, you cannot add new items to it, and you cannot remove items from it. Tuples are represented with parenthesis. There are two syntaxes to create a tuple:

```
 my _tuple = tuple()
 my_tuple

 >> ()
```

And

```
 my_tuple = ()
my_tuple
>> ()
```

If you want your tuple to contain objects, you must add them to your tuple when you create it. Here is how you add items to a tuple using the first syntax:

```
 my_tuple = tuple(“brown”, “orange”, “yellow”)
my_tuple

>> (“brown”, “orange”, “yellow”)
```

And the second:

```
my_tuple = (“brown”, “orange”, “yellow”)
```

```
my_tuple

>> (“brown”, “orange”, “yellow”)
```

A tuple with one item in it still needs a comma after the item:

```
(‘self_taught’,)

>> (‘self_taught’,)
```

Once you’ve created your tuple, if you try to add an object to it, Python will raise an exception:

```
my_tuple = (“brown”, “orange”, “yellow”)
my_tuple[1] = “red”

>> TypeError: 'tuple' object does not support item assignment
```

You can, however, access data from a tuple like a list—you can reference an index and slice a tuple:

```
my_tuple = (“brown”, “orange”, “yellow”)
my_tuple[0]
my_tuple[1:2]

>> yellow
>> ('yellow', 'orange')
```

You can check if an item is in a tuple using the keyword in:

```
my_tuple = (“brown”, “orange”, “yellow”)
“brown” in my_tuple

>> True
```

Add the keyword not to check if an item is not in a tuple:

```
my_tuple = (“brown”, “orange”, “yellow”)
“black” not in my_tuple

>> True
```

You may be wondering why you would want to use a data structure that appears to be like a list, but less helpful. Tuples are useful when you are dealing with values you know will never change, and you don’t want other parts of your program to have the ability to change those values. A good example is if you are working with geographic coordinates. You may want to store the longitude and latitude of New York in a tuple because you know the

longitude and latitude of New York is never going to change, and you want to make sure other parts of your program don't have the ability to accidentally change them.

Dictionaries

Dictionaries are another built-in container for storing objects. They are mutable—but unlike lists and tuples—they do not store objects in a specific order. Instead, dictionaries are used to map one object (called the key) to another object (called the value). Dictionaries are represented with curly brackets. There are two syntaxes for creating dictionaries:

```
my_dict = dict()
my_dict

>>  {}
```

And:

```
my_dict = {}
my_dict

>> {}
```

You add objects to a dictionary by mapping a key to a value. Each key mapped to a value in a dictionary is called a key value pair. Here is how you create key value pairs when you create a dictionary with the first syntax:

```
my_dict = dict({“Apple”: “Red”, “Banana”: “Yellow”})
my_dict

>> {'Apple': 'Red', 'Banana': 'Yellow'}
```

And the second:

```
my_dict = {“Apple”: “Red”, “Banana”: “Yellow”}
my_dict

>> {'Apple': 'Red', 'Banana': 'Yellow'}
```

Both syntaxes have a key separated from a value by a colon. Each key value pair must be separated by a comma. Unlike a tuple, if you have just one key value pair, you do not need a comma after it. Once you've added key value pairs to a dictionary, you can use a key to lookup a value. You can only use a key to lookup a value. You cannot use a value to lookup a key:

```
my_dict[‘Apple’]

>> Red
```

Dictionaries are mutable, so once you’ve created one you can add more key value pairs with the syntax [my_dictionary][[key]]=[value] :

```
my_dictionary = dict()

my_dictionary[“programming”] = “awesome”
my_dictionary[“programming”]

my_dictionary[“Bill Gates”] = “rich”
my_dictionary[“Bill Gates”]

my_dictionary[“america_founded”] = 1776
my_dictionary[“america_founded”]

>> awesome
>> rich
>> 1776
```

You can use the in keyword to check if a key is in a dictionary. You cannot use the in keyword to check if a value is in a dictionary.

```
“Bill Gates” in my_dictionary

>> True
```

Add the keyword not to check if a key is not in a dictionary:

```
“Bill Plates” not in my_dictionary

>> True
```

Finally, you can delete a key value pair from a dictionary with the keyword del

```
my_dictionary
del my_dictionary[ 'Bill Gates' ]
my_dictionary

>> {'america_founded': 1776, 'programming': 'awesome', 'Bill
      Gates': 'Rich'}
>> {'america_founded': 1776, 'programming': 'awesome'}
```

Challenge

Lists, tuples and dictionaries are just a few of the containers built-in to Python. Take some time to look up and read about Python sets. What is a situation would you use a set in?

Chapter 6. String Manipulation

“ In theory, there is no difference between theory and practice. But, in practice, there is. ”
— Jan L. A. van de Snepscheut

Python has built-in functionality for manipulating strings, such as changing a string’s case or splitting a string into two parts at a given character. This frequently comes in handy. Say for example, you have a string IN ALL CAPS and you want to change it to lowercase. Luckily, with Python, we can easily fix this problem. In this chapter we will learn more about strings and go over some of Python’s most useful tools for manipulating strings.

Triple Strings

If a string is more than one line, you need to put it in triple quotes:

“““line one
line two
line three
”””

If you try to define a string that spans more than one line with single or double quotes, you will get a syntax error.

Index es

Strings are iterable. You can access each character in a string with its index, just like you can access each item in a tuple. Like tuples, the first character in a string starts with index 0 and each subsequent index is incremented by 1.

```
my_string = “LAX”
my_string[0]
my_string[1]
my_string[2]

>> ‘L’
>> ‘A’
>> ‘X’
```

In this example w e used the indexes 0 , 1 , and 2 to access each of the characters in the string “LAX” . If we try to access an element past the last element, Python raises an exception:

```
my_string = “LAX”
my_string[3]

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

Strings are Immutable

Strings, like tuples, are immutable. You cannot change characters in a string. If you want to change the characters in a string, you need to create a new string.

Methods

In chapter 4, we learned about functions. In this chapter, will be using a concept similar to functions—methods—to manipulate strings. We learn more about methods in Part II of this book— but for now you can think of methods as functions that objects “come” with.

You can pass parameters to a method, which can execute code and return a result just like a function. Unlike a function, a method gets called on an object. For example if we have a string “Hello” , we could call “Hello”.[method_name]() on our string. Other than being called on an object, you can think of a method as the same thing as a function(for now).

Change Case

You can change a string so every letter is uppercase by calling the upper method on it:

```
“““ If computer programming were a country, it would be the third most diverse for
languages spoken. ””” .upper()

>> “““ IF COMPUTER PROGRAMMING WERE A COUNTRY, IT WOULD BE THE
THIRD MOST DIVERSE FOR LANGUAGES SPOKEN ”””
```


Similarly, you can change every letter in a string to lowercase by calling the lower method on it:

“““ Ada Lovelace, the daughter of the English poet Lord Byron, is considered to be the first computer programmer. ””” .lower()

>> “ada lovelace, the daughter of the english poet lord byron, is considered to be the first computer programmer”

42

You can also capitalize the first letter of every word in a sentence by calling the capitalize method on a string:

””” you can build a computer using anything that can implement a NAND-gate and the concept of zero (i.e., something and nothing). all Turing-complete programming languages are equally powerful (ignoring practicalities). lisp appeared in 1958 and is still regarded as being among the more powerful programming languages today.”””.capitalize()

>> ””” You can build a computer using anything that can implement a NAND-gate and the concept of zero (i.e., something and nothing). All Turing-complete programming languages are equally powerful (ignoring practicalities). Lisp appeared in 1958 and is still regarded as being among the more powerful programming languages today.””” 40

format

Sometimes you will want to create a string using variables. This is done with the format method:

year_started = “1989”
“Python was created in {}.”.format(year_started)

>> ‘Python was created in 1989.’

The format function looks for any occurrences of {} in the string and replaces them with the values you pass into format.
You are not limited to using {} once, you can put as many of them in your string as you’d like:

https://github.com/calthoff/tstp/blob/master/part_I/string_manipulation/format.py

year_started = “1989”

```
creator = “Guido van Rossum”
country = “the Netherlands”
my_string = “Python was created in {} by {} in {}.”.format(year_started, creator,
country)
print(my_string)

>> ‘Python was created in 1989 by Guido van Rossum in the
Netherlands.’
```

split

Strings have a method called split used to separate one string into two strings. You pass the split method the character or characters you want to use to separate the string—for example, we can pass in a period to separate this quote by Daniel Coyle into two different strings:

```
”””” Practice doesn’t make perfect. Practice makes myelin, and myelin makes
perfect.””””.split(“.”)

>> ["Practice doesn't make perfect", ' Practice makes myelin, and
myelin makes perfect', '']
```

The result is a list with two different strings split at the period in the original string.

join

The join method lets you add new characters in between every character in a string:

```
my_string = ‘abc’
join_result = ‘+’.join(my_string)
join_result

>> ‘a+b+c’
```

You can turn a list of strings into a single string by calling the join method on an empty string (“”) and passing in a list:

```
the_Fox = [“The”, “fox”, “jumped”, “over”, “the”, “fence”, “.”]
```

```
one_string = “”.join(the_Fox)
one_string
```

>> The fox jumped over the fence.

replace

The replace method lets you replace every occurrence of a character(s) with another character(s). The first parameter is the character(s) to replace and the second parameter is the character(s) to replace it with.

```
my_string = “The cat jumped over the hat.”
my_string = my_string.replace(“a”, “@”)
my_string
```

>> “The c@t jumped over the h@t.”

index

We can get the index of the first occurrence of a character in a string with the index method. We pass in the character we are looking for, and we get the index of the first occurrence of the character in the string:

```
‘cat’.index(‘a’)
>> 1
```

in

The in keyword checks if one string is in another string and returns True or False :

```
“Playboy” in  ””” A picture from Playboy magazine is the
most widely used for all sorts of image processing
algorithms” ””
```

>> True
42

Add the keyword not in front of in to check if one string is not in another string:

```
" hello " not in """ The computer virus was initially
  designed without any harmful intentions """

>> True
42
```

Escaping Strings

What if you want to use quotes inside a string? If we use quotes inside a string we get a syntax error:

```
""" Sun Tzu said "The Supreme art of war is to subdue the enemy without fighting." "
""""

>> SyntaxError: invalid syntax
```

We can solve this with escaping, which means putting a special symbol in front of a character that has special meaning in Python (in this case the special character is a quote), to let Python know that this particular character is meant to be a character, and not the special Python symbol it usually represents. The special symbol we use to escape our quote is a backslash.

```
""""Sun Tzu said \"The Supreme art of war is to subdue the
      enemy without fighting.\" """

>> ' Sun Tzu said "The Supreme art of war is to subdue the enemy
  without fighting." '
```

Newline

We can use "\n" inside a string to represent a newline:

```
print("line1 \nline2")
>> 'line1'
>> 'line2'
```

Concatenation

We can add two strings together using the addition operator. The result will be one string with the characters from the first string followed by the characters from the next strings. This is called concatenation:

“cat” + “in” + “the” + “hat”
> ‘catinthehat’

“cat ” + “in ” + “the ” + “hat ”
>> ‘cat in the hat’

String Multiplication

We can also multiply a string by a number with the multiplication operator:

“cat” * 3
>> ‘catcatcat’

Chapter 7. Loops

The second program we learned how to write printed Hello, World! a hundred times. We did this by using a loop. Loops are compound statements that let us execute code over and over again a certain number of times, or as long as a condition is True . In this chapter we will go over two kinds of loops—for loops and while loops.

For Loops

For loops execute a set of instructions a certain number of times. You give a for loop a number to start at (we will call this number a), a number to stop at (we will call this number z), a set of instructions, and a variable that keeps track of the number of times the instructions have been executed (we will call this variable i). You can think of a for loop as a circle. It goes round and round executing instructions. Every time the loop executes its instructions, i (which starts at the value of a) gets incremented by 1 . When i becomes equal to z , the the for loop stops.

You define a for loop with the syntax for [variable_name] in range(a, z): [code_to_execute] where a is the number to start at, z is the number to stop at, and [variable_name] is a variable name of your choosing that gets assigned to whatever number the loop is on (we've been referring to it as i) and code_to_execute is the code you define that is executed each time around the loop. This all is easier to understand with an example:

```
# https://github.com/calthoff/tstp/blob/master/part_I/loops/for_loops_ex1.py

for i in range(0, 10):
    print(i)
```

```
>> 0
>> 1
…
>> 9
```

In this example we chose i as our variable_name . This is the variable name people in the Python community usually use when they write a for loop. Your code_to_execute has access to the variable i .

When we run this program, our program enters our for loop. a starts at 0 , z starts at 10 and i starts at the same value as a — 0 . The first time around the loop i is 0 , and our for loop executes its instructions— print(i) — which prints 0 (because i is equal to 0). The next time around the loop, i is incremented by 1 , so i is now 1 , and so 1 gets printed. The next

time around the loop, i is incremented by 1 again, and so now i is 2 , and so 2 gets printed. Eventually, i will equal 10 . When this happens, nothing will print, because i is equal to z and so the loop ends.

When a loop ends, its instructions stops executing, and Python moves on to the next line of code after the loop. That is why 9 is the last number printed. If there were more code after our loop, Python would execute it, but in this example there is not, and so the program ends.

There is another for loop syntax we can use for iteration. Iteration means going one by one through an iterable. An iterable is an object that has indexes. Some examples of iterables are strings, lists, tuples and dictionaries. Iteration is done with the syntax for variable_name in iterable: [code_to_execute] . Just like the first syntax, we chose a variable name and define the code to be executed each time around the loop. In this syntax, instead of our loop executing code until i is equal to z , our loop starts at the first index in the iterable, and stops after the last index. Also variable_name gets assigned to the item at the index we are at (in the iterable). Here is an example:

```
# https://github.com/calthoff/tstp/blob/master/part_I/loops/for_loops_ex2.py

my_string = “Python”
for character in my_string:
    print(character)

>> ‘P’
>> ‘y’
>> ‘t’
>> ‘h’
>> ‘o’
>> ‘n’

# https://github.com/calthoff/tstp/blob/master/part_I/loops/for_loops_ex3.py

my_list = [“a”, “b”, “c”]
for item in my_list:
    print(character)

>> ‘a’
>> ‘b’
>> ‘c’

# add github

my_tuple = (“a”, “b”, “c”)
for item in my_tuple:
    print(character)

>> ‘a’
```

```
>> ‘b’
>> ‘c’

 # add github

 my_dict = {“self”: “taught”, “programming”: “wizard”}
for key in my_dict:
    print(key)

>> “self”
>> “wizard”
```

Each of these examples loops through an iterable, and prints each item in it. You will notice we used several different variable names for variable_name . In this syntax, instead of using i , you want to use a descriptive variable name. In the first example, we used character , because each item at an index in a string is called a character. In the second and third examples, we used item , because each object in a list or tuple is called an item. In the last example, we used key , because when you iterate through a dictionary like this, you can only access each key in the dictionary, not the value— so we chose a variable name to make this clear.

Being able to loop through an iterable is very useful. Iterables are used to store data, and you can use for loops to loop through your data to easily make changes to it, or move the data from one iterable to another.

While Loops

While for loops execute code a certain number of times, while loops execute code as long as the expression in its header evaluates to True . The syntax for creating a while loop is while [expression]: [code_to_execute] . Like a for loop, a while loop goes around like a circle executing code. The difference is instead of executing code a set amount of times, a while loop executes code as long as the expression we define in its header evaluates to True .

If we use an expression that always evaluates to True , our loop will run forever. This is called an infinite loop. Writing an infinite loop is easy (be prepared to press control-c on your keyboard in the Python shell. It is the only way to stop an infinite loop from running).

```
 # https://github.com/calthoff/tstp/blob/master/part_I/loops/while_loops_ex1.py

 while True:
    print(“Hello, World!”)

>> Hello World
...
```

Because a while loop runs as long as its expression evaluates to True —and True always evaluates to True —this loop will run forever, continuously executing the code we defined. In other words, our program will never stop printing “Hello, World!” .
Now let’s look at a while loop with an expression that will eventually evaluate to False :

```
# https://github.com/calthoff/tstp/blob/master/part_I/loops/while_loops_ex2.py

 x = 10
while x > 0:
      print(‘{}’.format(x))
     x -= 1
 print(“Happy New Year!”)
```

```
 >> 10
>> 9
>> 8
>> 7
>> 6
>> 5
>> 4
>> 3
>> 2
>> 1
>> ‘Happy New Year!’
```

Our while loop will execute its code as long as the expression we defined is True . In this case, that means it will execute its code as long as x > 0 . x starts at 10 (we defined x before we created our while loop). The first time through our loop, x is 10 , so x > 0 evaluates to True . Our while loop’s code prints x and then decrements x by 1 — x is now 9 . The next time around the loop x gets printed and x becomes 8 . This continues until x becomes 0 , at which point x > 0 evaluates to False and our loop ends. Python then executes the next line of code after our loop— print(“Happy New Year!”) — which prints “Happy New Year!” .

Break

You can prematurely end a for or while loop with the keyword break . For example, the following loop will run one hundred times:

```
# https://github.com/calthoff/tstp/blob/master/part_I/loops/break.py

 for i in range(0, 100):
```

```
    print(i)
```

```
>> 0
>> 1
...
```

But if we add a break statement to the code the loop executes, the loop only runs once:

```
# https://github.com/calthoff/tstp/blob/master/part_I/loops/break_ex2.py

for i in range(0, 100):
    print(i)
    break
```

```
>> 0
```

The loop goes around once and prints 0 . When the the break keyword is executed, the loop ends. This is useful in many situations. For example, we can write a program that asks the user for input until they type “q” to quit:

```
# https://github.com/calthoff/tstp/blob/master/part_I/loops/break_ex3.py

"""If you are unfamiliar the reference in this example, go watch Monty Python and the Holy
Grail!"""

questions = ["What is your name?", "What is your favorite color?", "What is your quest?"]
n = 0
while True:
    print("Type q to quit")
    answer = input(questions[n])
    if answer == "q":
        break
    n += 1
    if n > 2:
        n = 0
```

Each time through our infinite loop, our program will ask the user a question from our list of questions. We use the variable n to keep track of a number which we use as an index to get a question from our questions list. When n becomes greater than 2 , we’ve run out of questions and we set n back to 0 which will ask the first question in the list. This will go on indefinitely, unless the user types in “q” , in which case our program hits the break keyword and the loop ends, which ends our program.

Continue

You can use the keyword continue to stop executing a for or while loop's code, and jump to the top of a loop. Say for instance, we want to print the numbers from 1 to 5 , except for the number 3 . We can do this by using a for loop with the continue keyword:

```
# https://github.com/calthoff/tstp/blob/master/part_I/loops/continue.py

 for i in range(1, 6) :
    if i == 3:
        continue
    print(i)
```

```
>> 1
>> 2
>> 4
>> 5
```

In our loop, when i equals 3 , our program hits the continue keyword. Instead of causing our loop to exit completely—like the break keyword—the loop persists but w e get jumped to the top of our loop, which means any of the loop's code that would have executed that time around the loop gets skipped. In this case when i is equal to 3 , everything after continue is skipped (in this case print(i)). The result is 3 is not printed. Here is the same example with a while loop:

```
 # add github
i = 1
while i <= 5:
    if i == 3:
        continue
    print(i)
```

Nested Loops

You can combine loops in various ways. For example, you can have one loop inside of another loop. You can also have a loop inside a loop inside a loop. There is no limit to the amount of times you can do this, although in practice you want to limit the number of times you do.

When a loop is inside another loop, the second loop is said to be nested in the first loop. The first loop is called the outer loop, and the nested loop is called the inner loop. When you nest two loops, the inner loop runs its full course each time around the outer loop. Here is an example:

https://github.com/calthoff/tstp/blob/master/part_I/loops/nested_loops.py

```
for i in range(1, 3):
  print(i)
  for letter in [‘a’, ‘b’, ‘c’]:
    print(letter)
```

```
>> 1
>> ‘a’
>> ‘b’
>> ‘c’
>> 2
>> ‘a’
>> ‘b’
>> ‘c’
```

The nested for loop will iterate through the list “[a’, ‘b’, ‘c’]” however many times the outside loop runs—in this case twice. If we changed our outer loop to run three times, the inner loop would iterate through the list three times.

If you have two lists of numbers and want to create a new list with all of the numbers from each list added together you can use two for loops:

https://github.com/calthoff/tstp/blob/master/part_I/loops/nested_loops_ex2.py

```
list1 = [ 1 , 2 , 3 , 4 ]
list2 = [ 5 , 6 , 7 , 8 ]
added_up = []
for i in list1:
  for j in list2:
    added_up.append(i + j)

print (added _up)
```

```
>> [6, 7, 8, 9, 7, 8, 9, 10, 8, 9, 10, 11, 9, 10, 11, 12]
```

In the second for loop we used the variable j because i is already in use by the first loop. You can also nest a for loop inside a while loop and vice versa:

https://github.com/calthoff/tstp/blob/master/part_I/loops/nested_loops_ex3.py

```
while input ( 'Continue y or n?' ) != 'n' :
  for i in range (1, 5 ):
```

```
        print (i)

>>
Continue y or n?y
1
2
3
4
5
Continue y or n?y
1
2
3
4
5
Continue y or n?n
>>>
```

This program will print the numbers 0-4 until the user enters n .

Challenge

Write a program that has an infinite loop (with q to quit), and each time through the loop, it asks the user to guess a number and tells them whether their guess was right or wrong.

Chapter 8. Modules

Imagine we wanted to build a large project with ten thousand lines of code. If we kept all of that code in one file, our project would be difficult to navigate. Every time there was an error or exception in our code, we would have to scroll through a file with ten thousand lines to find the line of the code with a problem. Programmers solve this problem by dividing large programs up into multiple pieces. Each piece is stored in a module— which is another name for a Python file. Python has a special syntax that lets you use code from one Python module in another Python module. Python also comes with built-in modules you can use that give you access to extra functionality.

Importing Built-in Modules

In order to use a module, you must import it first, which means use a special syntax to make the code in the module available to use in your program. This is done with the syntax import [module_name] . import is a keyword for importing modules and must be followed by the module name. For now, we are only going to learn how to import built-in modules and modules located in the same folder as the module you are importing it from. In Part III, we learn how import modules from different locations .
We can import Python's built-in math module with the following syntax:

```
import math
>>
```

The math module is a module that comes with Python when you install it. It is a regular Python file with a bunch of math related functionality—it contains Python functions that are useful when you are doing math. Once you've imported a module, you can use any of the code from it. You can access a function in the module with the syntax [path_to_module]. [function_name]() . With this syntax, you can use any of the code (such as a function) from the math module in your program:

```
import math
math.fabs(-3)
>> 3
```

The fabs function returns the absolute value of the parameter you pass in. You may be wondering how you are supposed to know there is a math module with a function called fabs

in it. A list of Python's built-in modules can be found at https://docs.python.org/3/py-modindex.html. If you search for the math module, there is a link that takes you to a page that lists every function in the math module, what each function does, and what parameters it takes.

Another built-in module is the random module. Here is an example of importing the random module, and using a function from it called randint that takes two numbers as parameters and returns a random number between them.

```
# The output of this program might not be 52 when you run it—it's random!

import random

random.randint(0,100)

>> 52
```

There are other syntaxes for importing modules, but in general import [module_name] is the syntax you should use, and the one we will be using throughout the book. Finally, you should do all of the imports for your program at the top of your file.

Importing Modules

In this section, we are going to create a module, and use the code from it in another module. First create a new folder on your computer called tstp , then create a file called hello.py in it. Inside hello.py add the following code:

```
# https://github.com/calthoff/tstp/blob/master/part_I/modules/hello.py

def print_hello():
  print("Hello")
```

Save the file. Inside the tstp folder, create another Python file called project.py . Inside project.py add the following code:

```
# https://github.com/calthoff/tstp/blob/master/part_I/modules/project.py

import hello

hello.print_hello()

>> 'Hello'
```

Using the import keyword we can easily use code from our first module in our second module.

Challenge

I challenge you to write three functions in a module, and use them in another Python program.

Chapter 9. Files

Python makes it easy to work with files. You can easily read and write data from a file. Reading data from a file means accessing the files content. Writing data to a file means adding new content to the file. This is useful whenever you want to use data from a file in a program, or output data from a program to a file. In this chapter, we will learn the basics of working with files—including how to read and write to files.

Working With Files

The open function takes a string representing the path to a file and a string representing the mode to open the file in as parameters. The path to a file—also called a file path—represents the location on your computer a file resides, for example:
/Users/calthoff/my_file.txt is the file path to a file called my_file.txt . Each word separated by the slash that precedes it is the name of a folder. Together they represent the location of that file. If a file path only has the name of the file, Python will look for the file in whatever folder you are running your program in.

The second parameter represents the mode to open the file in, which determines the actions you will be able to perform on the file. Here are a few of the modes you can open a file with:

"r" Opens a file for reading only.

"w" Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

"w+" Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

Once you've passed the open function a file path and mode as parameters, open creates a file object (We learn about objects in Part II) we can use to read or write to our file (or both depending on the mode you chose). Here is an example writing to a file:

```
my_file = open("my_file.txt", "w")
my_file.write("Hello from Python!")
my_file.close()

>>
```

open creates a new file called my_file.txt (because we passed in “w” as our mode) in whatever directory you ran our program in. We save our file object returned by the open function in in the variable my_file .

Now we can call the write method on our file object, which accepts a string as a parameter and writes it to the new file we created. Finally, we closed our file by calling the close method on the file object. This is an important step. Whenever you open a file using the open method, you need to close it with the close method. If you have a program where you use the open method on multiple files, and you forget to close them, it can cause problems in your program.

Using with

Because forgetting to close files you opened can cause problems, there is a second, prefered syntax for opening files. The preferred way to open a file in Python is to use a compound statement using the with keyword and the syntax with open(‘my_file’,[mode]) as [variable_name]: [your_code] . When you use a compound statement using with the file automatically closes after the last suite executes in the statement. The file object gets saved in a variable of your choosing ([variable_name]). Here is the example from the previous section written using the preferred syntax for opening a file:

```
# https://github.com/calthoff/tstp/blob/master/part_I/files/using_with.py
 with open(‘my_file.txt’, ‘w’) as my_file:
      my_file.write(‘Hello from Python!’)

>>
```

As long as you are inside of the with statement, you can work with the file you opened using the variable you created—in this case my_file . As soon as Python leaves the with statement, it closes the file for you .

Reading Files

If you want to access the contents of a file (read the file) you pass in “r” as the second parameter to open . Then you can call the read method on your file object which returns an iterable you can iterate through to get each line of the file.

```
# https://github.com/calthoff/tstp/blob/master/part_I/files/reading_files.py
```

```
with open(“my_file.txt”, “r”) as my_file:
    for line in my_file.read():
        print(line)
```

> > Hello from Python!

You can only call read on a file once (each time you run your program) to gets its contents, so you should save the file contents in a variable or container if you need to use the file contents again later in your program. For example, we could change the previous example to save the file contents in a list:

```
#

my_list = list()

with open(“my_file.txt”, “r”) as my_file:
    for line in my_file.read():
        my_list.append(line)

print(my_list)
```

> > Hello from Python!

With the file contents saved in a list, we can easily access it later in our program whenever we need it.

CSV Files

Python comes with a built-in module for working with CSV files. CSV stands for comma separated value. It is a file format commonly used in Microsoft Excel: a program for creating spreadsheets. A comma in a CSV file is called a delimiter. Every piece of data separated by a comma represents a cell in Excel. Each line of the CSV file represents a row in Excel. Here is an example of the contents of a csv file:

```
# my_file.csv
one, two, three
four, five, six
```

You could load this file into excel and one , two and three would each get their own cells in the first row of the spreadsheet; and four , five and six would each get their own cells in the second row of the spreadsheet.

We can use a with statement to open a CSV file like the example from the previous section, but inside the with statement we need to use the csv module to convert our file object

into a csv object. The csv module has a method called writer that accepts a file object and a delimiter and returns a csv object with a method called writerow we can use to write to our CSV file:

```
# https://github.com/calthoff/tstp/blob/master/part_I/files/csv_files.py

import csv

with open(' my_file.csv' , 'w' ) as csvfile:
   spamwriter = csv . writer(csvfile, delimiter = ',')
    spamwriter . writerow([ 'one', 'two', 'three' ])
    spamwriter . writerow([' four', 'five', 'six' ])

>>
```

The writerow method accepts a list as a parameter. Every item in the list gets written to the CSV file and each item is separated by the delimiter you passed to the writer method (in this case a comma). writerow only writes one row, so we have to call it twice to create two rows. When you run this program, it will create a new file called my_file.csv and when you open the file with a text editor, it will look like this:

```
# my_file.csv
on e, two, th ree
four, five, six
```

If you load this file into Excel (or Google Sheets a free Excel alternative), the commas disappear, but one , two and three will each have their own cell in row one; and four , five and six will each have their own cell in row two.

We can also use the csv module to read the contents of a file. To read from a CSV file, first we pass in 'r' to open as as the second parameter. This opens the file for reading only. Instead of using the writer method like the previous example, we use the reader method, but still pass in the file path and a comma as the delineator.

reader returns an iterable we can iterate through to print each row. We can call the join method on a comma to add a comma in between each value to print the contents of the file like it appears in the original file:

```
# https://github.com/calthoff/tstp/blob/master/part_I/files/csv_files_ex2.py

import csv

with open('my_file.csv', 'r') as csvfile:
      spamreader = csv . reader(csvfile, delimiter = ',' )
      for row in spamreader:
        print ( ',' . join(row))

>> one,two,thre e
```

```
>> four, five, six
```

Challenge

Data persists when it outlives the program that created it. We can use files to persist data by writing the output of our programs to a file. Write a program that collects data from a user—and saves it to a file—so the data persists.

Chapter 10. Bringing It All Together

In this chapter, we are going to combine the concepts we've learned so far to build a text-based game: the classic game Hangman. If you've never played Hangman, here's how it works. Player One picks a secret word and draws lines representing each letter in the word (we will use an underscore to represent each line). Player Two tries to guess the word one letter at a time. If Player Two guess a letter correctly, the corresponding underscore is replaced with the correct letter. If Player Two guesses incorrectly, Player One draws a piece of a picture of a hangman (a person hanging). If Player Two completes the word before the picture of the hangman is drawn, they win, if not they lose. In our program the computer will be Player One, and the person playing the game will be Player Two. Are you ready to build Hangman?

Hangman

Here is the beginning of our hangman code:

#
https://github.com/calthoff/tstp/blob/master/part_I/bringing_it_all_together/hangman.py

```
def hangman ():
  word = "caat"
  wrong_guesses = 0
   stages = [ "" , "_________      " , "|      |      " , "|      0
" , "|     /|\     " , "|     / \     " , "|           " ]
   letters_left = list (word)
  score_board = [ '__' ] * len (word)
  win = False
   print ( 'Welcome to Hang Man' )
```

First we create a function called hangman we can call to start the game. The word to guess is stored in the variable word (you can change the word the player has to guess by changing the variable). We use another variable wrong_guesses to keep track of how many incorrect letters the player has guessed.

stages is a list filled with strings we will use to draw our Hangman. When each string in the stages list is printed on a new line, a picture of a hangman forms. letters_left is a list made

up of each character in word . We will use it to keep track of which letters are left to guess in our word.

scoreboard is a list of strings used to keep track of the hints we display to the user e.g., “c __ t” if the word is “cat” . The initial value of score_board is calculated with ['__'] * len(word) which returns a list made up of the number of underscores to start with. For example if the word is “cat” score_board starts as [“__”, “__”, “__”] .

Finally, we have a win variable that starts as False to keep track of whether the player has won the game yet or not. To start the game off we print Welcome to Hangman .

When you build a game, you normally use a loop that continues until the game is over. Here is the loop we will use in our game:

```
while wrong_guesses < len (stages) - 1 :
    print ( ' \n ' )
    guess = input ( "Guess a letter" )
    if guess in letters_left:
        character_index = letters_left.index(guess)
        score_board[character_index] = guess
        letters_left[character_index] = '$'
    else :
        wrong_guesses += 1
    print ( '' .join(score_board))
    print ( ' \n ' .join(stages[ 0 : wrong_guesses + 1 ]))
    if '__' not in score_board:
        print ( 'You win! The word was:' )
        print ( ' ' .join(score_board))
        win = True
        break
```

Our loop continues as long as the variable wrong_guesses is less than the len(wrong_guesses) - 1 . wrong_guesses keeps track of the number of wrong letters the user has guessed, so as soon as the user has guessed more wrong letters than the number of strings that make up the hangman, the game is over. The reason we subtract 1 is because the length function in Python does not count from 0 , it counts from 1 . In order to compensate for this, we have to subtract one, to compensate for the fact that length counts starting from 1 instead of 0 . The reason we want to count from 0 is because stages is a list, and we are going to be using wrong_guesses as an index to get the strings from stages and indexes start at 0 .

Inside our loop, we print a blank space to make our game look nice when it’s printed in the shell. Next we collect the player’s guess with the built-in input function and store the value in the variable guess .

If guess is in letters_left (a variable from the beginning of our program that keeps track of the letters that haven’t been guessed yet), the player guessed correctly. If the player guessed correctly we need to update our score_board list, which we use later in the game to display the score. If the user guessed “c” , we want to change our score_board to look like

this: [“c”, “__”, “__”] . We use the index method on our letters_left list to get the first index of the letter that was guessed. We use that index to replace the underscore in score_board at the index with the correctly guessed letter. We have a problem though. Because index only returns the first index of the character we are looking for, our code will not work if word (the word the player is guessing) has more than one of the same character. To compensate for this, we modify letters_left by replacing the character that was just correctly guessed with a dollar sign so that the next time around the loop, if there is more than one letter in the word index will find second occurrence of the letter since the first occurrence of the letter was replaced by a dollar sign. If on the other hand the player guesses an incorrect letter, we simply increment wrong_guesses by 1 .

Next we print the scoreboard and print our hangman using our score_board and stages lists. To print the scoreboard, all we have to do is print " .join(score_board) .

Printing the hangman is trickier. When each of the strings in our stages list is printed on a new line, a complete picture of a hangman is printed. We can easily create the entire hangman by printing ' \n ' .join(stages) . This connects each string in the stages list with a blank space (\n) . But we want to print our hangman at the stage we are currently at, which we accomplish by slicing our stages list. We start at stage 0 and slice up until whatever stage we are at (represented by the variable wrong_guesses) plus one. The reason we add one, is because when you are slicing, the end slice does not get included in the results. This gives us only the strings we need to print the stage of the hangman we are currently on which we then print.

The last thing we do in our loop is check if the user has won the game If there are no more underscores in the score_board list, we know the user has guessed all the letters and won the game. If the user has won, we print that they won and we print the word they correctly guessed. We also set the variable win to True which is used when we break out of our loop.

Once we break out of our loop, if the user won, we do nothing, and the program is over. If the user did not win, the variable win will be False . If that is the case we know the user lost the game and we print the full hangman, print “You lose!” followed by the word they incorrectly guessed:

```
    if not win:
        print ( ' \n ' .join(wrong_guesses[ 0 : stage]))
        print ( 'You lose! The words was {}' .format(word))

    hangman()
```

Here is our complete code:

```
    #
https://github.com/calthoff/tstp/blob/master/part_I/bringing_it_all_together/hangman.py

    def hangman ():
        stage = 0
```

```
    wrong_guesses = [ "" , "_________      " , "|     |      " , "|     0      " , "|    /|\     " , "|    / \
" , "|              " ]
    word = "cat"
    score_board = [ '__' ] * len (word)
    win = False
    print ( 'Welcome to Hang Man' )
    while stage < len (wrong_guesses) - 1 :
        print ( ' \n ' )
        guess = input ( "Guess a letter" )
        if guess in word:
            score_board[word.index(guess)] = guess
        else :
            stage += 1
        print (( ' ' .join(score_board)))
        print ( ' \n ' .join(wrong_guesses[ 0 : stage + 1 ]))
        if '__' not in score_board:
            print ( 'You win! The word was:' )
            print ( ' ' .join(score_board))
            win = True
            break
    if not win:
        print ( ' \n ' .join(wrong_guesses[ 0 : stage]))
        print ( 'You lose!' )

hangman()
```

Challenge

Building text-based games is a great way to improve your programming ability. Build another text-based game that interests you.

Chapter 11. Practice

“ The fool wonders, the wise man asks. ”
— Benjamin Disraeli

If this is your first programming book, I recommend you spend time practicing before moving on to the next section. In this chapter, I provide exercises to help you get additional practice before moving on to the next section, resources to check out, and we cover how to get help if you get stuck.

Exercises

0. Create a text based game of your favorite sport.
0. Make up your own text based game. When I was starting out I built a fantasy based game based on Heroes of Might and Magic III
0. Build a text based magic 8 ball program where the user can shake a magic eight ball and get predictions about their future
0. Create a program that asks the user what kind of mood they are in and recommends a song.
0. Build a program that prints out ten brands and lets the user type them in. When they do, the program should print the brand’s trademark i.e., the user could type Nike and the program would print “Just do it.”

Read

0. http://programmers.stackexchange.com/questions/44177/what-is-the-single-most-effective-thing-you-did-to-improve-your-programming-skil
0. https://www.codementor.io/ama/0926528143/stackoverflow-python-moderator-martijn-pieters-zopatista

Getting Help

If you get stuck, I have a few suggestions. The first is posting a question on http://forum.theselftaughtprogrammer.io , it is a forum I set up for people reading this book

to get answers to any questions they have.

I also recommend checking out Stack Exchange—an amazing resource for programmers. There are two websites on Stack Exchange you should explore. The first is Stack Overflow. If you haven't used it yet, you will soon be familiar with it. Google almost any programming question and an answer will pop up on Stackoverflow, which makes it a game changer for learning to program.

Learning to rely on other people's help was an important lesson for me. Struggling to figure things out is a major part of the learning process, but at some point it becomes counter-productive . Working on projects in the past, I used to continue to struggle way past the point where it was productive. Today if that happens, I will post a question online, if the answer is not already on there. Every time I've posted a question online, someone has been able to answer it. I can't say enough about how helpful and friendly the programming community is.

Other Resources

link to website for other python resources

Part II Introduction to Object-oriented Programming

Chapter 12. Programming Paradigms

“There are only two kinds of languages: the ones people complain about and the ones nobody uses”
—Bjarne Stroustrup

A programming paradigm is a certain way of programming. There are many different programming paradigms. You won’t need to learn all of them at the beginning of your career—but it’s important to know what some of the most popular paradigms are. The programming paradigms we will go over in this chapter are imperative programming, functional programming and object-oriented programming—with a focus on object-oriented programming—the subject of the remaining chapters of this section.

State

One of the fundamental differences between different programming paradigms is the handling of state. State is the data your program has access to. Programs store data in variables—so state is the value of a program’s variables at a given time the program is running.

Imperative Programming

In Part I, we learned to program imperatively. Imperative programming can be described as “do this, then that”. An imperative program is a sequence of steps moving toward a solution—with each step changing the program’s state. An example of imperative programming would be:

```
x = 2
y = 4
z = 8
xyz = x + y + z

>> 14
```

Each step of the program changes the program's state. We get to `xyz` by first defining `x`, followed by `y`, followed by `z` and finally defining `xyz`.

Functional Programming

Functional programming is another popular programming paradigm. It originates from lambda calculus. Functional programming involves writing functions that—given the same input— always return the same output. In functional programming, you only program with functions, you do not use classes—a feature of object-oriented programming we will learn about shortly. There is a lot of jargon in functional programming and Mary Rose Cook does a great job cutting through it with her definition, "Functional code is characterised by one thing: the absence of side effects. It doesn't rely on data outside the current function, and it doesn't change data that exists outside the current function." [61] She follows her definition with an example which I will also share with you. Here is an unfunctional function:

```
a = 0

def increment ():
    global a
    a += 1
```

Here is a functional function :

```
def increment (a):
    return a + 1
```

The first function is unfunctional because it relies on data outside of itself, and changes data outside of the current function by incrementing a global variable. The second function is functional because it does not rely on any data outside of itself, and it does not change any data outside of itself either. Functional programmers write functions this way to eliminate side effects—the unintended consequences that happen when you are constantly changing the state of your program.

Object-oriented Programming

The object-oriented programming paradigm involves writing programs where you define and create objects that interact with each other. We've been programming with objects this whole time—strings, integers and floats are all examples of objects. But you can also define your own objects using classes. Classes are the blueprint used to create objects. You can think of a class as the idea of an object. Think of an orange. An orange is an object. A fruit weighing between 2 to 10 ounces is the idea of an orange—a class.

We can model oranges in Python by defining a class we can use to create orange objects. We define a class using the class keyword followed by the name we want to give our class. A class is a compound statement with a header followed by suites. You write suites after

the header, which can be simple statements, as well as compound statements called methods. Methods are like functions, but they are defined inside of a class, and can only be called on the object the class can create. We saw different examples of this in Chapter 5 when we called various methods on strings. Here is an example how we can represent an orange using a class in Python:

#
https://github.com/calthoff/tstp/blob/master/part_II/object_oriented_programming/orange_ex1

```
class Orange :
    print ( "Orange created!" )
```

We started with the class keyword followed by the name of our class—in this case Orange because we are modeling oranges. By convention, classes in Python always start with a capital letter and are written in camelCase—which means if a class name is made up of more than one word, the words should not be separated by an underscore (like a function name), instead each word should be capitalized LikeThis.

After our class definition we have a simple statement— print(“Orange created!”) . This code will execute when we create an orange object. With this class definition, we can create as many Orange objects as we’d like:

```
orange = Orange()
print ( type (orange) )
print (orange )
```

```
>> Orange created!
>> <class '__main__.Orange'>
>> <__main__.Orange object at 0x101a787b8>
```

We created a new Orange object using the same syntax we use to call a function—[classname]() . This is called instantiating an object, which means creating a new object. “Orange created!” prints as soon as we instantiate our Orange object. When we print type(orange) , the type function tells us our Orange object is an instance of the Orange class we just created. When we print our Orange object, Python lets us know it is an Orange object, and then gives us its location in memory. When you print an object like this, the location in memory printed on your computer will not be the same as the example, because the object’s location in memory changes each time the program runs.

Now we are going to add a method to our Orange class. You define a method with the same syntax as a function. There are only two differences: a method must be defined as a suite in a class, and a method has to accept at least one parameter (except in special cases I won’t go into). You can name the first parameter of a method whatever you’d like, but by convention the first parameter in a method is always named self , and I’ve never seen this convention broken.

The reason every method must have a first parameter is because whenever a method is called on an object, Python automatically passes the method the object that called it. This concept exists in most programming languages that support object-oriented programming, however, Python makes the passing of self as a parameter explicit whereas many other languages make it implicit. What I mean is that Python makes you explicitly define self in every method you create whereas in other languages self is just implied to have been passed to the object. If we define a method and print self , we will see it is the Orange object we called our method on:

```
class Orange :
    print ( "Orange created!" )

    def print_orange ( self ):
        print ( self )

Orange().print_orange()
```

```
>> Orange created!
>> <__main__.Orange object at 0x101a787b8>
```

self is useful because we can use it to define and access variables that belong to our Orange object. We do this by defining a special method called __init__ , which stands for initialize. When you instantiate an object, if you've defined a method called __init__ , Python automatically calls the __init__ method for you when the object is created. Inside __init__ we can give our Orange object variables with the syntax self.[variable_name] = [variable_value] . Here is an example:

```
class Orange :
    print ( "Orange created!" )

    def __init__ ( self ):
        self .color = "orange"
        self .weight = 10

    def print_orange ( self ):
        print ( self )
        print ( self .color)
        print ( self .weight)

orange = Orange()
orange.print_orange()
```

```
>> Orange created!
>> <__main__.Orange object at 0x10564dba8>
>> orange
>> 10
```

Using __init__ we can now create oranges that get a color and weight when initialized and we can use and change these variables in any of our methods just like regular variables. In Python, any method surrounded on both sides by underscores is called a magic method which means it does something special. The print_orange method was used to illustrate an example, it will not be a method in our orange as we continue to model it.

We can change our class definition so the person creating the object can pass in their own variables when they create a new orange, instead of the weight and color starting with default values . Here is our new class definition:

#
https://github.com/calthoff/tstp/blob/master/part_II/object_oriented_programming/orange_ex3

```
class Orange :
    def __init__ ( self , weight , color , mold):
        """all weights are in oz"""
        self .weight = weight
        self .color = color
```

Now we can create a wider variety of oranges objects:

```
orange = Orange( 10 , 'orange' , )
```

We just created a 10 oz (per the comment “all weight are in oz”), orange colored orange. We can access the oranges variables using dot notation:

```
print (orange.weight)
```

>> 10

```
print (orange.color)
```

>> “orange”

We can also change any of the values of our orange object:

```
orange.weight = 100
print (orange.weight)
```

>> 100

Moving forward with the modeling of our orange, there is more to an orange than just its physical properties like color and weight. Oranges can also do things, and we need to be able to model that as well. What can an orange do? Well, for one thing, oranges can go bad from mold. We can model a molding orange by adding a mold variable to our Orange class, and creating a method that increments the mold variable when it’s called:

#
https://github.com/calthoff/tstp/blob/master/part_II/object_oriented_programming/orange_ex2

```
class Orange ():
    def __init__ ( self ):
      """all weights are in oz"""
        self .weight = 6
        self .color = 'orange'
        self .mold = 0

    def rot ( self , days , temperature):
        self .mold = days * (temperature * .1 )

orange = Orange()
print (orange.mold)
orange.rot( 10 , 98 )
print (orange.mold)
```

```
>> 0
>> 98.0
```

Now our orange objects will be able to rot. We defined a method that accepts the number of days it's been since the orange was picked, and the average temperature during that time as parameters. With our made-up formula, we can increase the amount of mold the orange has every time we call the rot method; and our orange now has the ability to rot.

That was a lot to take in , so let's go over it all one more time. In Object-oriented programming, we use classes to model objects. These objects group variables (state) and methods (for altering state) together in a single unit—the object. Classes have a special method called __init__ that Python calls when an object is created using the syntax [class_name]() . This is an example of one of Python's magic methods. The first argument to any method is called self by convention and exists because Python automatically passes the object that called the method into the method.

Challenge

Pick a programming paradigm and research what problems it solves.

Chapter 13. The Four Pillars of Object-oriented Programming

"Good design adds value faster than it adds cost."
—Thomas C. Gale

There are four main concepts in object-oriented programming—often called the four pillars of object-oriented programming: inheritance, polymorphism, abstraction and encapsulation. These concepts must all be present in a programming language in order for it to be considered an object-oriented programming language. Python, Java and Ruby are all examples of object-oriented languages. Not all programming languages support object-oriented programming—for example Haskell is a functional programming language that does not support object-oriented programming. In this chapter, we take a deeper look at object-oriented programming by exploring each of the four pillars of object-oriented programming.

I nheritance

Inheritance in programming is similar to genetic inheritance. In genetic inheritance you can inherit attributes from your parents, like your eye color. Similarly w hen you create a class, it can inherit from another class (which is then called its parent class)—giving the new class you created the parent class's variables and methods. In this section we will model a kid and adult using inheritance. First, we define a class to represent an adult:

“““
https://github.com/calthoff/tstp/blob/master/part_II/more_object_oriented_programming/inhe
”””

```
class Adult ():
    def __init__ ( self , name , height , weight , eye_color):
        """height is in feet, weight in lbs."""
        self .name = name
        self .height = height
        self .weight = weight
        self .eye_color = eye_color

    def print_name ( self ):
        print ( self .name)

tom = Adult( "Tom" , 6 , 150 , "brown" )
```

```
print (tom.name)
print (tom.height)
print (tom.weight)
print (tom.eye_color)
tom.print_name()
```

```
>> Tom
>> 6
>> 150
>> brown
>> Tom
```

Using this class we can create Adult objects with a name, height, weight and eye color. In addition, our Adult objects have a method called print_name that prints the parent's name.

We can model a human child that also has a name, height, weight, eye color and can print its name; with an extra method we don't want our Adult objects to have called print_cartoon ; using inheritance. You inherit from a parent class by adding parenthesis to the class name you are defining, and passing in the class name you want to inherit from as a parameter. Here is an example:

```
""""
https://github.com/calthoff/tstp/blob/master/part_II/more_object_oriented_programming
"""""

class Adult ():
    def __init__ ( self , name , height , weight , eye_color):
        # height is in feet, weight in lbs.
        self .name = name
        self .height = height
        self .weight = weight
        self .eye_color = eye_color

    def print_name ( self ):
        print ( self .name)


class Kid (Adult):
    def print_cartoon ( self , favorite_cartoon):
        print ( "{}'s favorite cartoon is {}" .format( self .name ,
            favorite_cartoon))


child = Kid( "Lauren" , 3 , 50 , "blue" )
print (child.name)
print (child.height)
```

```
print (child.weight)
print (child.eye_color)
child.print_name()
child.print_cartoon( 'DuckTales' )
```

```
>> brown
>> Ricky
>> DuckTale s
```

By passing in Adult to our Kid class, our Kid class inherits the variables and methods of our Adult class: when we create a Kid object we pass it a name, height, weight and eye color; and our Kid object is able to use the method print_name ; all of which was inherited from its parent class (without having to define any of it in our Kid class). This is important because not having to repeat code makes our program smaller and therefore more manageable.

After inheriting from our Adult class, all we had to do was define a new method called print_cartoon in our Kid class to create a Kid class with all of the functionality of our Adult class, plus additional functionality; all without affecting our Adult class.

Polymorphism

The best definition I've found of polymorphism is "polymorphism is the ability (in programming) to present the same interface for differing underlying forms (data types) " [69] An interface refers to one or more functions or methods. Let's take a look at a situation where this is the case:

```
print( 'Hello World' )
print( 200 )
print( 200.1 )
```

```
>> "Hello World"
>> 200
>> 200.1
```

In this example, we were able to present the same interface (the print function) for three different data types: a string, an int and a float. We didn't need to call three separate functions — print_string , print_int , or print_float in order to print these three different data types— instead Python has just one interface for all of them.

Let's take a look at another example. Say we want to write a program that can draw different shapes: triangles, squares and circles. Each of these shapes is drawn in a different way, so the methods to draw them would all have different implementations. In Python, we can create different draw methods for each shape so that Triangle.draw() will draw a triangle, Square.draw() will draw a square, and Circle.draw() will draw a circle. Each of these shape

objects has it's own draw interface that knows how to draw itself. When we have a shape object, we know we can call the draw function to draw the shape. The same interface is presented for all the different shape data types.

If Python did not support polymorphism— we would need a function that creates a triangle, and another function called draw_triangle to draw it; a function to create a circle, and a function called draw_circle to draw it; etc. Because Python has polymorphism, every shape can simply be drawn with its draw method. This makes our shape objects much easier to use and explain. Instead of explaining—we have three functions representing three different shapes, and another three functions that draws each of them; we can simply tell whoever is using the code: we have three shapes—if you want to draw one—call its draw method.

Abstraction

We use abstraction in object-oriented programming when we create a class and define its methods. Say we create a class to represent a person. When we define our person class—and the methods that go with it— we are creating an abstraction. Our definition of a person could include eye color, hair color, height and ethnicity as well as the ability to read, write and draw. We could have a five foot three person with blue eyes, blonde hair unable to read, write or draw. Or we could have a six foot five person with brown eyes, brown hair that can read, write and draw. Both of these fall into the category of the person abstraction we've created.

When we design object-oriented programs, we create abstractions of different concepts that all work together to form our program. For example, we may create an abstraction of a person, and an abstraction of a government, and model how many people live under each government in the world. Abstraction allows us to model objects with clear boundaries, and have them interact with each other. In Part 4 we learn more about abstraction and how Python itself is built on multiple layers of it.

Encapsulation

In object-oriented programming, encapsulation hides our codes internal data. When code is encapsulated, it means when it is called, the caller cannot access the code's internal data. Take a look at the method get_data :

```
class Data :
  def get_data (self, index , n):
    data = [ 1 , 2 , 3 , 4 , 5 ]
    data.append(n)
```

The method has a variable called data . When we call get_data , there is no way for us to access this variable because of encapsulation. If there was no encapsulation, we might be able to access the variable data —and append n to it—like this:

```
# warning this code does not work
Data.get_data.data.append(6)
```

If this was allowed, anyone could access the data variable in our get_data method. Instead of relying on our implementation of the get_data method—they could append n to data themselves. This is not a problem—until we change the implementation of get_data . What if we decide want the variable data to be a tuple instead of a list? If we make this change, it will break anyone's code calling append on the variable data , because tuples do not have an append method. But because of encapsulation, this scenario is not possible (which is why the code does not work), and we can be assured changes to the internal implementation of our code won't break our client's code (client is a term for the person using a piece of code).

Composition

While composition is not one of the four pillars of object-oriented programming, it is an important concept related to the rest. Composition is used to represent "has a" relationships—it occurs when one object stores another object as a variable.For example, say we want to represent the relationship between a dog and its owner—this is a "has a" relationship—a dog has an owner. First we define our dog and people classes:

```
class Dog ():
    def __init__ ( self , name , breed , owner):
        self .name = name
        self .breed = breed
        self .owner = owner


class Person ():
    def __init__ ( self , name):
        self .name = name
```

When we create our dog object, we pass in a person object as the owner parameter:

```
mick = Person( "Mick Jagger" )
dog = Dog( "Stanley" , "French Bulldog" , mick)
print (dog.owner)

>> Mick Jagger
```

Now our dog Stanley has an owner—a Person object named Mick Jagger—we can easily reference.

Challenge

Create an Orange class and object as many times as you have to until you can do it without referencing this chapter.

Chapter 14. More Object-oriented Programming

“Commenting your code is like cleaning your bathroom - you never want to do it, but it really does create a more pleasant experience for you and your guests.”
—Ryan Campbell

In this chapter we cover additional concepts related to object-oriented programming. While some of these examples are specific to Python, the majority of these concepts are present in other languages that support object-oriented programming.

How Variables Work

In this section, we are going to learn more about variables. Variable “point” to an object.

```
number = 100
```

number points to an integer object with the value 100.

[illustration of point to an object]

```
number = 101
```

When we assign a new value to number , it points to a new integer object with the value 101 , and the old integer object with a value of 100 is discarded because it is no longer being used. Two variables can point to the same object:

```
x = 100
y = x
```

x points to an integer object with a value of 100 . When we assign y to x , y now points to the same integer object x points to: they both point to an integer object with a value of 100 .

[insert illustration from http://cdn.oreillystatic.com/en/assets/1/event/95/Python%20103_%20Memory%20Model%20_]

What do you think the following program will print?

```
x = 10
y = x
x += 1
print(x)
```

```
print(y)
```

The answer is 11 and 10 . x points to an integer object with a value of 10, and when we create y , it points to the same integer object. When we increment x , x points to a new integer object —with a value of 11, but the integer object with a value of 10 is not discarded, because it is being used: y still points to the integer object with a value of 10 . So when we print x , 11 prints because we assigned x to a new integer object—11—but when we print y — 10 prints because changing the value of x , which points to the integer object 11 , does not affect the value of y . Here is another example to illustrate this point. What do you think the output of this code will be?

```
x = [1, 2, 3]
y = x
y[2] = 100
print(x)
print(y)
```

The output will be [1, 2, 100] twice . The reason is that both x and y point to the same list object. In the third line, we make a change to that single list object, and when we print both variables, it prints the list object they both point to, with the changes made in line 3.

i s

The keyword is returns True if two objects are the same object (they are stored in the same location in memory) and False if not.

“““
https://github.com/calthoff/tstp/blob/master/part_II/more_object_oriented_programming/is_ex
”””

```
class Person :
    def __init__ ( self ):
        self .name = 'Bob'

bob = Person()
the_same_bob = bob
print (bob is the_same_bob)

another_bob = Person()
print (bob is another_bob)
```

```
>> True
>> False
```

When we use the keyword is with bob and the_same_bob , the result is True because both variables point to the same Person object. When we create a new Person object and compare it to the original bob the result is False because the variables point to different Person objects.

None

The built-in constant None is used to represent the absence of a value:

```
x= None
x
>> None
```

We can test if a variable is None using conditional statements.

```
“““
https://github.com/calthoff/tstp/blob/master/part_I/introduction_to_programming/none_
”””

x = 10
if x:
    print ( "x is not None" )
else :
    print ( "x is None :( " )

>> x is not None


x = None
if x:
    print ( "x is not None" )
else :
    print ( "x is None :( " )

>> x is None :(
```

While this may not seem useful now, it will be later.

Classes Are Objects

In Python, classes are objects. This idea comes from the influential programming language SmallTalk. This means that when run a program in which you define a class—

Python turns it into an object—which you can then use in your program:

```
class Pterodactyl :
    pass

print (Pterodactyl)
```

```
>> <class '__main__.Pterodactyl'>
```

Without any work on our part, Python turns our class definition into an object which we can then use in our program, by printing it for instance.

Class Variables vs. Instance Variables

Classes can have two types of variables—class variables and instance variables. All of the variables we've seen so far have been instance variables defined with the syntax self.[variable_name] = [variable_value] . Instance variables belong to the object that created them. In other words we can do this:

```
class Liger :
    def __init__ ( self , name):
        self .name = name

connor = Liger( "Connor" )
print (connor.name)
```

```
>> Connor
```

Class variables belong to both the class that created them and the object. That means we can access them with the class object Python creates for each class:

```
class Liger :
    interests = [ "swimming" , "eating" , "sleeping" ]

    def __init__ ( self , name):
        self .name = name

print (Liger.interests)
```

```
>> ['swimming', 'eating', 'sleeping']
```

In this example we never created a Liger object, yet we were able to access the interests class variable. This is because class variables can be accessed by the class that created them. Class variables can also be accessed by the object:

```
class Liger :
```

```
    interests = [ "swimming" , "eating" , "sleeping" ]

    def __init__ ( self , name):
        self .name = name

larry = Liger( "Larry" )
print (larry.interests)
```

```
>> ['swimming', 'eating', 'sleeping']
```

Class variables are useful when you want every object in a class to have access to a variable. In this case the name of each Liger can be unique, but all of our Ligers to have access to the same list of interests.

Private variables

Most programming languages have the concept of private variables and methods: special variables and methods the designer of a class can create that the object has access to, but the programmer using the object does not have access to. One situation private variables and methods are useful in is if you have method or variable in your class the class uses internally, but you plan to change the implementation of your code later on (or you want to preserve the flexibility to have the option to), and therefore don't want whoever is using the class to rely on those variables and methods used internally because they might change (and would then break their code).

Unlike other languages Python does not have private variables. Variables that are not private are called public variables, and all of Python's variables are public. Python solves the problem private variables resolve another way—by using convention. In Python, if you have a variable or method the caller should not access, you precede its name with an underscore. Python programmers know if a method or variable has an underscore, they shouldn't use it, although they are still able to at their own risk. Here is an example of a class that uses this convention:

```
“““
https://github.com/calthoff/tstp/blob/master/part_II/more_object_oriented_programming/priva
”””

class PublicPrivateExample:
    def __init__ ( self ):
        self .public_variable = "callers know they can access this"
        self ._dontusethisvariable = "callers know they shouldn't access this"

    def public_method( self ):
        # callers know they can use this method
```

```
        pass

    def _dont_use_this_method( self ):
        # callers know they shouldn't use this method
        pass
```

Python programmers reading this code will know self.public_variable is safe to use, but they shouldn't use self._dontusethisvariable , and if they do, they do so at their own risk because the person maintaining this code has no obligation to keep self.dontusethisvariable around because callers are a not supposed to be accessing it. The same goes for the two methods: Python programmers know public_method is safe to use, whereas _dont_use_this_method is not.

Overriding Methods

When a class inherits a method from a parent, we have the ability to override it. Take a look at the following example:

“““
https://github.com/calthoff/tstp/blob/master/part_II/more_object_oriented_programming/over
”””

```
class Mammal :
    def __init__ ( self , name):
        self .hunger = 100
        self .tired = 100
        self .name = name

    def print_result ( self , amount , action):
        print ( "{} {} decreased by {}." .format( self .name , action ,
                amount))

    def eat ( self , decrease):
        self .hunger -= decrease
        self .print_result(decrease , 'hunger' )

    def sleep ( self , decrease):
        self .tired -= decrease
        self .print_result(decrease , 'tiredness' )


class Dolphin (Mammal):
    pass
```

```
class Tiger (Mammal):
    def sleep ( self , decrease):
        self .tired -= decrease
        print ( "The tiger is really tired!" )

dolphin = Dolphin( 'dolphin' )
dolphin.eat( 10 )
dolphin.sleep( 10 )

tiger = Tiger( 'tiger' )
tiger.eat( 10 )
tiger.sleep( 10 )
```

```
> > dolphin hunger decreased by 10.
>> dolphin tiredness decreased by 10.
>> tiger hunger decreased by 10.
>> The tiger is really tired!
```

We created two classes that inherit from Mammal . The first class, Dolphin , inherits all of its functionality from the Mammal parent class without making any changes. The second class Tiger defines a method called sleep , with different functionality than the sleep method it inherited from its parent class. When we call tiger.sleep , the new method we defined gets called instead of the inherited method. Other than this, Tiger and Dolphin have all the same functionality inherited from the parent class Mammal .

Super

The built-in super function, lets us call a method a class inherited from its parent. The super function is used with the syntax super().[parent_method]([parameters]) where you replace parent_method with the parent method you want to call and pass it any parameters it needs. The parent method is then called and executed, and the rest of the code in the method super was called from then finishes executing. Here is an example of how we can call the Mammal parent class's sleep method from our Tiger classes' sleep method, in order to use the code from the Mammal class's sleep method followed by additional functionality:

```
class Tiger (Mammal):
    def sleep ( self , decrease):
        super ().sleep(decrease)
        print ( "The tiger is really tired!" )

tiger = Tiger( 'tiger' )
```

```
tiger.eat( 10 )
tiger.sleep( 10 )
```

```
>>  tiger tiredness decreased by 10.
>> The tiger is really tired!
```

First we use the super keyword to call the Mammal parent class's sleep method and pass in the decrease variable as parameter. Mammal's sleep method is executed and prints "tiger tiredness decreased by 10" . The Tiger classes sleep method then executes the new functionality we added to the Tiger classes sleep method and "The tiger is really tired!" prints. By using the super keyword we were able to give a child class the functionality from a parent class's method without having to retype the functionality in the child class. This is important because you should always avoid repeating code with the same functionality in different places in your program when you can.

Overriding Built-in Methods

Every class in Python automatically inherits from a parent class called Object . All classes in Python inherit methods from this parent class. Python's built-in functions use these methods (which we learned are called magic methods)—in different situations—like when we print an object:

```
class Lion :
    def __init__ ( self , name):
        self .name = name

lion = Lion( "Dilbert" )
print (lion)
```

```
>> <__main__.Lion object at 0x101178828 >
```

When we print our Lion object, Python calls the __repr__ method on our object, which it inherited from the Object parent class. It prints whatever the __repr__ method returns. We can override this built-in method to change what happens when the print function prints our object.:

“““
https://github.com/calthoff/tstp/blob/master/part_II/more_object_oriented_programming/over
”””

```
class Lion:
    def __init__ ( self , name):
        self .name = name

    def __repr__ ( self ):
```

```
        return self .name

lion = Lion( "Dilbert" )
print (lion)
```

>> Dilbert

Because we overrode the __repr__ method, when we print our Lion object, the Lion object's name— Dilbert — gets printed instead of something like <__main__.Lion object at 0x101178828 > which the inherited __repr__ method would have returned.

Not all magic methods are inherited. Python expressions like 2 + 2 expect the operands to have a method the operator can use to evaluate the expression. In the example 2 + 2 , integer objects have a method called __add__ which is called when the expression is evaluated, but __add__ is not inherited when you create a class . We can create objects that can be used as operands in an expression with the addition operator by defining an __add__ method in our class:

```
class AlwaysPositive :
    def __init__ ( self , number):
        self .number = number

    def __add__ ( self , other):
        return abs ( self .number + other.number)

x = AlwaysPositive(- 20 )
y = AlwaysPositive( 10 )

print (x + y)
```

>> 10

Our AlwaysPositive objects can be used as operands in an expression with the addition operator because we defined a method called __add__ . The method must accept a second parameter, because when an expression with an addition operator is evaluated, __add__ is called on the first operand object, and the second operand object gets passed into __add__ as a parameter. The expression then returns the result of __add__ .

In this example, we added a twist. We used the function abs to return the absolute value of the two numbers being added together. Because we defined __add__ this way, two AlwaysPositive objects added together will always return the absolute value of the sum of the two objects.

Challenge

Write classes to model four species in the animal kingdom.

Chapter 15. Bringing It All Together

“It's all talk until the code runs.”
—Ward Cunningham

In this chapter, we are going to use what we’ve learned about classes and objects to create the card game War. If you’ve never played, War is a game where each player draws a card from the deck, and the player with the highest card wins. We will build War by defining a class representing a card, a deck, a player, and finally, a class to represent the game itself. These classes will work together to create the game War.

Cards

First we define our card class:

```
# https://github.com/calthoff/tstp/blob/master/part_II/war/war.py

class Card :
  suits = [ "spades" , "hearts" , "diamonds" , "clubs" ]
  values = [ None , None , "2" , "3" , "4" , "5" , "6" , "7" , "8" , "9" ,
           "10" , "Jack" , "Queen" , "King" , "Ace" ]

  def __init__ ( self , value , suit):
    """suit and value should be integers"""
    self .value = value
    self .suit = suit

  def __lt__ ( self , other):
    if self .value < other.value:
      return True
    if self .value == other.value:
      if self .suit < other.suit:
        return True
      else :
        return False
    return False

  def __gt__ ( self , other):
    if self .value > other.value:
      return True
    if self .value == other.value:
```

```
            if self .suit > other.suit:
                return True
            else :
                return False
        return False

    def __repr__ ( self ):
        return self .values[ self .value] + " of " + self .suits[ self .suit]
```

Our Card class has two class variables: suits and values . suits is a list of strings representing all of the different suits a card could be. values is a list of strings representing the different numeric values a card could be. The first two indexes of values are None so that the strings in the list match up with the index they represent—in other words so that the string "2" in values is positioned at index 2 .

Our card class also has two instance variables: suit , and number —each represented by an integer; and represent what kind of card the Card object is. For example, we can create a 2 of hearts by creating a Card object and passing in the parameters 2 and 1 (1 represents hearts because hearts is at index one in our suits list). Later our Card object will use these variables to print what kind of card it is, by using them as indexes in the suits and values lists.

illustration

In order to play War, we need to compare two cards to see which card is bigger. We added the ability to compare two cards in an expression with the comparison operators greater than(>) and less than(<) to our Card class by defining the two magic methods used by these comparison operators: __lt__ and __gt__ . By defining these methods we can determine what happens when our Card objects are compared—just like we did with our AlwaysPositive example in the previous chapter except with comparison operators instead of the addition operator. The code in these methods looks to see if the card is greater than or less than the other card passed in as a parameter. However, we must take one more thing into consideration—what happens if the cards have the same value—for example if both cards are 10's. If this situation occurs, we compare the value of the suits and return the result. This works because the integers we use to represent the suits, are in order—with the most powerful suit getting the highest integer and the least powerful suit getting the lowest integer. With these methods defined, we can compare to cards using the greater than and less than operators :

```
card1 = Card( 10 , 2 )
card2 = Card( 11 , 3 )
print (card1 < card2)
```

```
>> True
```

```
card1 = Card( 10 , 2 )
card2 = Card( 11 , 3 )
print (card1 > card2)
```

```
>> False
```

The last thing we do is override the magic method __repr__ , which we learned in the previous chapter is used by the print function to print an object. Overriding the __repr__ method lets us print the card a Card object represents:

```
card = Card( 3 , 2 ):
print card
```

```
>> 3 of diamonds
```

Deck

Next we need to define a class to represent a deck of cards. Our deck will create a list of fifty two cards—four of each suit—when it is initialized and it will have a method to remove a card from the cards list. Here is our deck:

```
from random import shuffle

class Deck :
    def __init__ ( self ):
        self .cards = []
        for i in range ( 2 , 15 ):
            for j in range ( 4 ):
                self .cards.append(Card(i , j))
        shuffle( self .cards)

    def remove_card ( self ):
        if len ( self .cards) == 0 :
            return
        return self .cards.pop()
```

When a Deck object is initialized, the two for loops in __init__ create all of the cards in a fifty two card deck and append them to our cards list. The first loop is from 2 to 15 because cards start with the value 2 and end with the value 14 (the ace). Each time around the inner loop, a new card will be created using the integer from the outer loop as the value (i.e., 14 for an ace) and the integer from the inner loop as the suit (i.e. a 2 for hearts). This results in the creation of fifty two cards—one for every suit.

Finally, we use the shuffle method from the random module to randomly rearrange the items in our cards list to mimic shuffling a deck of cards. Our deck has just one method, remove_card , which returns None if our list of cards is empty, and otherwise removes and returns a card from the cards list. Using our Deck class, we can create a new deck of cards and print out all the cards in the deck:

```
deck = Deck()
for card in deck.cards:
    print (card)
```

>> 4 of spades
>> 8 of hearts
...

Player

We need a class representing each player in the game so we can keep track of their cards and how many rounds they've won. Here is our player class:

```
class Player :
    def __init__ ( self , name):
        self .wins = 0
        self .card = None
        self .name = name
```

Our Player class has three instance variables: wins to keep track of how many rounds they've won, card to represent the current card the player is holding, andname to keep track of the player's name.

Game

Finally, we can create the class representing the game itself:

```
class Game :
    def __init__ ( self ):
        name1 = input ( "player1 name " )
        name2 = input ( "player2 name " )
         self .deck = Deck()
         self .player1 = Player(name1)
         self .player2 = Player(name2)

    def play_game ( self ):
        cards = self .deck.cards
         print ( "beginning War!" )
        response = None
        while len (cards) >= 2 and response != 'q' :
```

```
            response =input( 'q to quit. Any other key to play.' )
            player1_card =self .deck.remove_card()
            player2_card =self .deck.remove_card()
             print ( "{} drew {} {} drew {}".format( selfplayer1.name ,
                        player1_card , self .player2.name player2_card))
             ifplayer1_card > player2_card:
               self .player1.wins += 1
               print ( "{} wins this round" .format(self .player1.name))
             else :
               self .player2.wins += 1
               print ( "{} wins this round" .format(self .player2.name))
        print ( "The War is over.{} wins" .format( selfwinner( selfplayer1 ,
            self .player2)))

    def winner ( self , player1 , player2):
        ifplayer1.wins > player2.wins:
            returnplayer1.name
        ifplayer1.wins < player2.wins:
            returnplayer2.name
        return "It was a tie!"
```

When our game object is initialized, the __init__ method is called and we use the input function to collect the names of the two players in the game which we save in the variables name1 and name2 . We create a newDeck object and store it in the instance variable deck, and create two Player objects using the names we collected from the user and store them in the instance variables player1 and player2.

Our Game class has a method called play_game used to start the game. The heart of the method is the loop that keeps the game going as long as there are two or more cards left in the deck, and as long as the variable response is not equal to“q” . Before our loop we define response as None, and later we set it to the input of the user, which is the only situation it can become “q” and end the game. Otherwise, the game ends when there are less than two cards left in the deck.

Each time through the loop, two cards are drawn. The first card is assigned to player1 and the second card is assigned to player2. Next we print the name of each player and what card they drew. We compare the two cards to see which card is greater, increment the wins instance variable for the player that won, and print a message that says which player won.

Our Game class also has a method called winner which takes two player objects, looks at the number of rounds they won, and returns the player that won the most round. When our deck runs out of cards, we print a message that the war is over, call the winner method (passing in both player1 and player2) and print the name of the result—the player that won.

War

When we put it all together, we have the card game War:

```
from random import shuffle


class Card:
    suits = ["spades", "hearts", "diamonds", "clubs"]
    values = [None, None, "2", "3", "4", "5", "6", "7", "8", "9",
              "10", "Jack", "Queen", "King", "Ace"]

    def __init__(self, value, suit):
        """suit and value should be integers"""
        self.value = value
        self.suit = suit

    def __lt__(self, other):
        if self.value < other.value:
            return True
        if self.value == other.value:
            if self.suit < other.suit:
                return True
            else:
                return False
        return False

    def __gt__(self, other):
        if self.value > other.value:
            return True
        if self.value == other.value:
            if self.suit > other.suit:
                return True
            else:
                return False
        return False

    def __repr__(self):
        return self.values[self.value] + " of " + self.suits[self.suit]


class Deck:
    def __init__(self):
        self.cards = []
        for i in range(2, 15):
            for j in range(4):
                self.cards.append(Card(i, j))
```

```
        shuffle( self .cards)

    def remove_card( self ):
        if len( self .cards) ==0   :
            return
        return self .cards.pop()


class Player :
    def __init__( self , name):
        self .wins = 0
        self .card = None
        self .name = name


class Game:
    def __init__( self ):
        name1 = input( "player1 name ")
        name2 = input( "player2 name ")
        self .deck = Deck()
        self .player1 = Player(name1)
        self .player2 = Player(name2)

    def play_game( self ):
        cards = self .deck.cards
        print   ( "beginning War!" )
        response = None
        while len(cards) >= 2 and response != 'q':
            response =input   ( 'q to quit. Any other key to play.')
            player1_card =self .deck.remove_card()
            player2_card =self .deck.remove_card()
            print ( "{} drew {} {} drew {}"  .format( self .player1.name , player1_card, self .player2.name ,
player2_card))
            if player1_card > player2_card:
                self player1.wins +=  1
                print ( "{} wins this round".format(   self player1.name))
            else :
                self player2.wins +=  1
                print ( "{} wins this round".format(   self player2.name))
        print   ( "The War is over.{} wins" .format( self .winner( self .player1, self .player2)))

    def winner ( self, player1 , player2):
        if player1.wins > player2.wins:
            return player1.name
        if player1.wins < player2.wins:
```

```
            return player2.name
        return 'It was a tie!"

game = Game()
game.play_game()
```

>> "player1 name "

…

Chapter 16. Practice

Exercises

0. Build an object-oriented text-based version of Blackjack.
0. Build a web scraper to collect data from another website.
0. Find a Python project you are interested in hosted on pip,(hint look on GitHub), download it, and use it in a program.

Read

0. https://julien.danjou.info/blog/2013/guide-python-static-class-abstract-methods
0. http://stackoverflow.com/questions/2573135/python-progression-path-from-apprentice-to-guru

Part III Introduction to Programming Tools

Chapter 17. Bash

In this section of the book we learn to use tools that will make us more effective programmers. In this chapter we learn to use the command line shell Bash that comes with many versions of Linux and OS X (which are Unix-like operating systems). A command line shell is a program that lets you type instructions into it that your computer executes when you press enter.

Once you are able to use Bash, you will be able to use Unix-like operating systems more effectively. This is important because Unix-like operating systems are so widely used in the programming world: Linux runs on the majority of the world's web servers, and many programming jobs will require you to be familiar with Unix-like operating systems.

The examples in this chapter assume you are using either Ubuntu or OSX . If you are using Windows, you can follow along with the examples by going to theselftaughtprogrammer.io/bash where you will find a link to a website that emulates Bash—you just type the commands into the green box in the website.

Finding Bash

You can find Bash by searching for terminal from the icon titled Search your computer and online resources if you are using Ubuntu or from Spotlight search if you are using a Mac.

Using Bash

Bash is similar to the Python Shell. The Bash command line shell takes its own programming language called Bash as input. The programming language Bash has commands—which are like functions in Python. We start with a keyword; type a space; type the parameter we want to give the command (if any); hit the enter key; and Bash returns the result. One of Bash's commands is echo, which is similar to the print function in Python. Here is an example:

```
$ echo Hello, World!
>> Hello, World!
```

We typed the commandecho into Bash, followed by a space andHello, World! as a parameter. When we press enter, Hello, World! prints in the Bash command line shell.

You can also use programs you've installed—like Python—from the Bash command line shell. Enter the command python3 to use Python from the shell:

```
$python3
>>
```

Now you can execute Python code:

```
print(“Hello, World!”)
>> Hello, World!
```

Enter `exit()` to exit Python.

Relative vs Absolute Paths

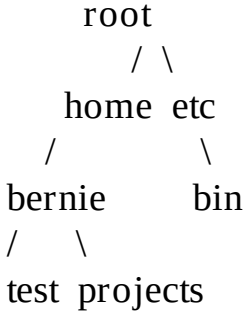
An operating system is made up of directories and files. A directory is another word for a folder on your computer. All directories and files have a path. A path is as an address where a directory or file exists in your operating system. When you are using the Bash command line shell, you are always in a directory, located at a specific path. You can see the path you are at by entering the command `pwd` in the Bash command line shell:

```
pwd
>> /Users/bernie
```

`pwd` stands for print working directory (a working directory is the directory you are currently in), and the path that prints on your computer will be the name of whatever directory you are in.

The folders in your operating system are a tree. A tree is an important concept in Computer Science called a data structure (covered in Part IV). In a tree, there is a root at the top. The root can have branches, and each one of the branches can have more branches, and those branches can have branches. This goes on indefinitely.

[add illustration of a tree data structure]



Every branch on the tree is a directory, including the root. The tree shows how they are connected to each other. When you are using Bash, at any given time you are at one of the locations on the tree, and a path is a way of expressing that location. There are two ways of expressing a location on Unix-like operating systems: absolute paths and relative paths. An absolute path is a way of expressing a location starting from the root directory. An absolute path is made up of the name of folders in the tree, in order, separated by backslashes. The absolute path to the bernie directory (for the operating system illustrated above) is

/home/bernie . The first slash represents the root directory; followed by the home directory; followed by another slash and the bernie directory.

The other way of specifying a location on your computer is called using a relative path. Instead of starting at the root directory, a relative path is relative to the directory you are in. Your operating system knows a path is relative when it doesn't begin with a backslash. If we were located in the home directory in the example above, the relative path to the projects directory would be bernie/projects . If we were in the home directory, the relative path to bernie is simplybernie .

Navigating

You can use the cd command, which stands for change directory, to navigate to a directory using an absolute or relative path. Enter the cd command followed by the absolute path / to navigate to the root directory:

```
$ cd /
```

You are now in your root directory, which you can verify with the command pwd :

```
$ pwd
>> /
```

The command ls, which stands for list directories, prints the folders in the directory you are in:

```
$ ls
>> bin dev initrd.img lost+found
...
```

You can create a new directory with the commandmkdir . Directory names cannot have spaces in them. Use themkdir command to make a new directory called tstp:

```
$ mkdir tstp
>>
```

Verify the new directory exists with the command ls. Now use the cd command to enter the tstp directory by passing in the relative path to tstp as a parameter:

```
$ cd tstp
```

Now use the cd command followed by two periods to return to the folder you were in before you entered the tstp directory:

```
$ cd ..
```

Delete the new directory using the command rmdir , which stands for remove directory:

```
$ rmdir tstp
```

Use the command lsto verify the directory was deleted.

Flags

Commands have a concept called flags that allow the issuer of the command to change the commands behavior. If you use a command without any flags, all of the commands flags are set to false. But if you add a flag to a command, the flag is set to true and the behavior of the command changes. The - and -- symbols are used to attach flags to a command. --author is an example of a flag you can add to the ls command to print the author of all of the directories and files that get listed. This example is for Linux, on OS X you can use the same flag but you need to use one dash instead of two. Here is an example of using the --author flag with the ls command:

```
$ ls --author
>> drwx------+ 13 coryalthoff   442B Sep 16 17:25 Pictures
...
```

When you add the --author flag to your ls command, the name of each directory and folder in your current directory will print—as well as some additional information, including the name of the person that created them.

vim

Vim is a command line text editor. It's like Microsoft Word, except you use it from the command line. If you are using Ubuntu, first install vim with the command apt-get install vim . Make sure to enter Y when prompted. If you are using a Mac,it should come with vim If you are using the online bash shell, it already has vim installed.

You can create a new file with vim by going to the command line and using the command vim [name of the file to create] . Use the command vim self_taught.txt to create a new text file called self_taught.txt . Press the i or insert key and type a paragraph of text into the file. Any paragraph of text you find on the internet will do. The reason you have to hit the i or insert key when you first enter a file is because vim has different modes optimized for different activities. vim starts in, Normal Mode, which is not meant for adding text to the file (it is meant for easy navigation)—you can delete text in normal mode, but you cannot insert new text. Once you press the i or insert key and enter normal mode, you can use vim like a word processor—try typing into vim.

Since you can't use your mouse to move the cursor around, it's important to learn a few shortcuts to jump to different locations in your document, so that you don't end up using the arrow keys on your keyboard (because that's slow). To practice moving around, first make sure you are in Normal Mode (control-c). You can move to the beginning of a word by pressing `b` and the end of a word with `e` . 0 will move you to the beginning of the line you are on, while the dollar sign `$` will move you to the end of the line. `H` will move you to the first line of the page and `L` will move you to the last. You can delete entire lines of text in normal mode by pressing the d key twice. Spend some time using these keys to get familiar with with navigating through a file using vim.

To exit vim you need to first switch to Normal Mode by pressing control-c . Next press the `shift` key and then hit the `colon` key (while still holding the shift key). From here you can type q! if you want to quit without saving your changes to the file, or type x if you want save your changes and quit. Once you've typed one of these options, press the enter key to exit. vim is useful in a few situations: servers are usually only accessed with a command line shell, so if you want to make changes to a file on a server, you need to use a command line text editor, and once you get good at using vim, it is often faster to use it to make changes than using a conventional word processor. Try typing `vimtutor` in the Bash command line shell and see what happens.

Touch

You can use the command `touch` followed by a filepath to quickly create a new file:

```
$ touch purple_carrots.txt
>>
```

View A File With less

The `less` command enables you to view files from the command line. Pick a file on your computer and use the command `less [filepath]` to view the file from the command line in the `less` program. Make sure you are in the same folder you created the file selftaught.txt in with vim and pass the filename as a parameter:

```
$ less self_taught.txt
>> whatever text you put in your file
```

Press `q` to exit.

Users

Operating systems have different users. A user represents a person in the world—someone using the operating system—and each user has different activities they are allowed to do—called their permissions—as well as their own directories and files that other users can't access. Users can also be placed into groups, with an entire group given. You can see the name of the user you are on your operating system with the command `whoami`:

```
$ whoami
>> cory
```

Normally you start as the user you created when you installed your operating system. This user is not the most powerful user the operating system has. The root user is the highest level user, which means it has the highest level of permissions. Every system has a root user. The root user can do an anything: for example the root user can create or delete other users.

For security reasons, you normally do not log in as the root user. Instead you use a command called sudo in front of another command to temporarily use the power of the root user to issue the command.`sudo` allows you to do most things as the root user, but not everything. Make sure to be careful using`sudo`, because using commands with sudo can harm your operating system if you don't know what you are doing. We are not going to cover using sudo in this book, but I've set up a tutorial on using `sudo` at theselftaughtprogrammer.io/sudo you can read through once you feel comfortable using the Bash command line shell.

Permissions

Every directory and file on your computer has a set of permissions. These permissions define what actions a user can perform on them. There are three types of permissions: `r`, `rw` and `x`, which stand for read, write and execute. Reading a file means viewing it, writing a file means changing it, and executing a file means running it as a program. You can view a file or directory's permissions with the command`ls -lah [name_of_file]`. Use the command `touch` to create a new file called`tstp`:

```
$ touch tstp
>>
```

Now we can view the files permissions:

```
$ ls -lah tstp
-rw-r--r--  1 coryalthoff  staff     5B Feb 21 11:55 test.py
```

Our file has three permissions that apply to three different groups of users—represented by -rw-r--r-- . The first set applies to the owner of the file, the second set applies to the group assigned the file and the third set applies to everyone. So in this example, the owner of the file has permission to read and write the file, whereas everyone else can only read the file.

You can add and subtract permissions with the command chmod (short for change mode). You can add permission to read with the command chmod +r , permission to read and write with the command chmod +rw , and permission to execute withchmod +x . You can subtract the same permissions with chmod -r , chmod -w and chmod -x, respectively.

Bash Programs

If we want to run a bash script from the command line, we need to give the file owner permission to execute. Create a new file calledhello_world.sh and type echo Hello, World! in it. The reason we use the .sh extension is to let anyone who sees this file know it is a Bash script. The syntax ./[file_name] is used to execute a file in the directory you are in . You execute a file with/[file_name] and the period means look for the file in the current directory You could also replace the period with the path to the file. Try executing the file (make sure you are in the same directory as the file):

```
$ ./hello_world.sh
>> Permission denied
```

The reason the console printed permission denied is because we do not have permission to execute the file. We can change that with the chmod command.

```
$ chmod u+x hello_world.sh
$./hello_world.sh .
>> Hello, World!
```

In the example above, we addedu to+x to formu+x because u stands for user, and we only wanted to give ourselves the ability to execute the file. If you are repeating an argument from the previous command line, in this case the filename, you can use !$ to represent the repeated argument.

Processes

Hidden Files

Your operating system, and different programs on your computer, use hidden files to store data. Hidden files in Unix-like systems are files that by default are not shown to users because changing them can affect the program(s) that depends on them. Hidden files start with a period—for example—.hidden . You can see hidden files using the command lswith the flag -a , which stands for all. Create a hidden file named .self_taughtwith touch .hidden and test if you can see it with the commands ls and ls -a .

Environmental Variables

Your operating system can store and use variables called environmental variables. You can create a new environmental variable from the command line with the syntax export variable_name=[variable_value] . In order to use an environmental variable, you must put a dollar sign in front of the environmental variable's name. For example:

```
$ export x=100
$ echo $x
>> 100
```

Creating an environmental variable from the command line is not permanent. If you quit the command line, reopen it, and type echo $x, the terminal will no longer print 100 .
We can make the environmental variable x persistent by adding it to a hidden file used by Unix-like operating systems called .profile located in your home directory. Go to your home directory with the command cd ~ (~ is a shortcut for representing your home directory on Unix-like operating systems) and open your .profile file using vim with the command vim .profile . Make sure to enter Normal Mode, type export x=100 into the first line of the file, and exit with :x . Close and reopen your command line, and you should still be able to print the environmental variable you defined:

```
$ echo $x
>> 100
```

The environmental variable x gets 100 as long as it's defined in your .profile file.

$PATH

When you type a command into the Bash command shell, it looks for the command in all of the directories stored in an environmental variable named $PATH . $PATH is a string of directory paths separated by colons. The Bash command shell looks in each of these directories for a file matching the name of the command you typed. If the file is found, the

command is executed, if not the command shell prints “command not found”. Use echo to print your $PATH environmental variable (your $PATH may look different than mine):

```
echo $PATH
>> /usr/local/sbin:/user/local/bin:/usr/bin:
...
```

In this example there are three directory paths in $PATH : /usr/local/sbin, /user/local/bin and /usr/bin . The Bash command line shell will be able to execute any command found in these three directories. You can see where a command like grep is installed by passing it as a parameter to the command which:

```
$ which grep
>> /usr/bin/grep
```

The grep command is located in /usr/bin , one of the locations in my operating system’s $PATH environmental variable.

pipes

In Unix-like systems, the character | is known as a pipe. You can use a pipe to use the output of one command as the input for another command. For example, we can pass the output of the ls command as the input of the less command to open less with all of the files in the current directory:

```
$ ls | less
>> Applications
...
```

The result is a text file with the output of ls opened up in the program less (press q to quit less). You are not limited to piping two commands—you can chain multiple commands together using pipes.

c at

You can use the versatile catcommand to display the contents of a file and to catenate, which means “to connect in a series.” [11] . Create a new file calledhello.txt and addHello, World! as the first line. Now use the cat command to print the contents of the file:

```
$ cat hello.txt
>> echo Hello, World!
```

Touse the cat command to catenate two files, first create a file called c1.txt and add the text Boy . Then create another file calledc2.txt and add the text Meets World . Now we can catenate them by passing both files as parameters to the cat command , followed by the greater than symbol (>), and the name of the new file to create:

```
$    cat c1.txt c2.txt > combined.txt
>>
```

Open up the newly createdcombined.txt which should say Boy Meets World .

Recent Commands

You can scroll through your recent commands by pressing the up and down arrows in the command line shell. To see a list of all of your recent commands use the commandhistory :

```
$ history
>> 1. echo Hello, World!
>> 2. pwd
>> 3. ls
…
```

Jump Around

When you are typing a command in the terminal, there will be times where you want to edit the command you've already typed. Your first instinct will be to use the arrow right or left key to move the cursor to the part you want to change. But this is slow. Instead you should use shortcuts that will get you there faster.

Type echo hello, world! (without pressing Enter) in the terminal and useesc b to move the cursor back one word, andesc f to move the cursor forward one word. You can also move the cursor to the beginning of the line withcontrol a or the end of the line with control e .

Tab Complete

Tab complete is a feature that will help improve the speed you get things done from the command line shell. If you are in the middle of typing a command you and press thetab button on your keyboard, the command line shell will try to autocomplete the command for you.

Try it for yourself by typing ech in the command line followed by tab ; ech will automatically get turned into echo. You can also use tab to complete file or directory paths. Start typing the path of the directory you are in and finish it off by pressing t ab. If you press tab and nothing happens, it is because two commands or paths are named similarly, and the shell doesn't know which to choose. For example if you have a directory named car , and another directory named candy, and you type ca and try to tab complete, nothing will happen because the shell won't know whether to choose car or candy. If you add an n so you've type can, and press tab complete, the shell will autocomplete candy because the shell knows car is not correct.

Wildcard

A wildcard is a character used to match a pattern. Two examples of wildcards are an asterisk and a question mark. The asterisk wildcard matches everything either starting or ending with a pattern, depending if you put it before or after the pattern. Asterisk wildcards are commonly used with the command ls . The command ls *.txt will show any files in the current directory that end with .txt , whereas ls .txt* will show any files that with .txt .

A question mark will match any single character. Sols t?o will show any files or directories that match t followed by any character followed by o. If you had directories named two and too , they both would get printed.

Other Tools

If your terminal gets cluttered, you can clear it with the command clear . If a process is taking too long, you can kill it withcontrol+c . Another powerful command isgrep, used to search files for patterns and covered in the next chapter.

The One Week Challenge

I challenge you to only use the command line for one week. That means no graphical user interface for anything other than using the internet! You should complete this challenge during a week you are actively working on a programming project.

Chapter 18. Regular Expressions

quote

A **regular expression** is a sequence of characters used to look for a pattern in a string. In this chapter we are going to practice using regular expressions from the command line using grep , a command for using regular expressions to search files. We will also learn to use regular expressions in Python.

Setup

To get started practicing using regular expressions, create a file calledzen.txt . From the command line (make sure you are inside the directory where you created zen.txt) enter the command python3 -c “import this”. This will print out The Zen of Python, a poem written by Tim Peters:

> The Zen of Python
> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one-- and preferably only one --obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than *right* now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea -- let's do more of those!

The - c flag in Python executes Python passed in as a string, andmport this prints the Zen of Python when executed. Copy and paste the Zen of Python into the file zen.txt . I also

recommend reading it as it contains some great wisdom. Use the function exit()to exit Python. If you are using the online bash shell.

If you are using a Mac, set the following environmental variables in the terminal:

```
$ export GREP_OPTIONS='--color=always'
$ export GREP_COLOR='1;35;40'
```

This will make grep highlight the words matched in the terminal, which happens by default on Ubuntu but not on OSX. Remember, setting an environmental variable from the terminal is not permanent, so if you exit the terminal and come back you will have to set the environmental variables again. Add the environmental variables to your .profilefile if you want to make the change permanent.

Simple Match

The command grep accepts a regular expression and the path to a file to look for the regular expression in as parameters. From the command line, in the directory where you created the file zen.txt , enter the following command:

```
$ grep Beautiful zen.txt
>> Beautiful is better than ugly.
```

Beautiful is the regular expression and zen.txt is the path to the file to look for the regular expression in. Beautiful is a sequence of characters that match the word Beautiful . This is the simplest kind of regular expression. Your console printed the line Beautiful is better than ugly . with Beautiful highlighted because it is the word the regular expression matched.

Ignore Case

If we change our regular expression from Beautiful to beautiful, it will no longer match anything in the Zen of Python. Enter grep beautiful zen.py to see for yourself. If we want our regular expression to match the word beautiful regardless of case (whether or not characters are capitalized), we can use the flag -i :

```
$ grep -i beautiful zen.txt
>> Beautiful is better than ugly.
```

Because we added the flag -i to our command, grep ignores case and highlights Beautiful again.

Only Return Matched

grep returns the entire line of the file a match was found on. We can return the exact word matched by using the flag -o:

```
$ grep -o Beautiful zen.txt
>>Beautiful
```

Normallygrep would have returned Beautiful is better than ugly.but because we added the flag-o , only the exact match,Beautiful , gets returned.

Match Beginning and End

Regular expressions have certain characters that don't match themselves, but instead do something special. For example the^ character is used to look for matches only if they occur at the beginning of a line:

```
$ grep ^If zen.txt
>> If the implementation is hard to explain, it's a bad idea.
>> If the implementation is easy to explain, it may be a good
   idea.
```

Similarly, we can use the dollar sign to only match the lines that end with a pattern:

```
$ grep idea.$ zen.txt
>> If the implementation is hard to explain, it's a bad idea.
>> If the implementation is easy to explain, it may be a good idea.
```

The lines matched both end with idea. The lineNamespaces are one honking great idea -- let's do more of those! was ignored because although it includes the word idea, it does not end with the word idea. You can also combine the two anchor matches we've covered into the regular expression ^$ to search for empty lines in a file.

Match Multiple Characters

You can use brackets in a regular expression to match any of the characters inside the brackets. In this example, instead of matching text from our zen.txt file, we are going to use a pipe to pass in a string to grep:

```
$ echo Two is a number and too is not. | grep  -i t[ow]o
>> Two  is a number and too  is not.
```

Remember, the pipe symbol passes the output of one command as the input of the next. In this case the output of echo is passed as the input of grep. The command echo Two is a number and too is not. | grep -i t[ow]o will match both two and too because the regex is looking for at followed by either an o or a w followed by ano.

Repetition

We can add repetition in our patterns with the asterisk symbol. The asterisk symbol means " The preceding item will be matched zero or more times." [56] We might want to say, matchtw followed by any amount ofo's . The regular expression grep two* accomplishes this:

```
$ echo two twooo twoo not too. | grep -o two*
>> two
>> twooo
>> twoo
```

Adding a *after two means the regular expression should match anything with two followed by any number of o's .

Range

You can match a range of letters or numbers by putting the range inside brackets. For example, we might want to only match the numbers in122233 hello 334443 goodbye 939333 . The regex [[:digit:]] * will match all the numbers in the string because it includes a range of numbers (0-9) , followed by* which tells the regex to match as many numbers in a row as it can.

```
$  echo 122233 hello 334443 goodbye 939333 | grep -o [0-9]*
>>  122233
>> 33443
>> 939333
```

Si milarly, you can match a range of characters (all characters in this case) with the regex [[:alpha:]]:

```
$ echo 122233 hello 334443 goodbye 939333 | grep [ [:alpha:]]*
```

Escaping

What if we want to match one of the special characters we've been discussing, like the dollar sign? We can do this by escaping the character. We covered escaping in the chapter Manipulating Strings: escaping means prefixing a character with a special character to let the program evaluating the syntax know you want to use the actual character and not the special meaning the character normally has. Escaping in regular expressions is done with a backward slash:

```
$ echo I love $ | grep  \\$
>> I love $
```

Normally, the dollar sign has the special meaning of only matching something at the end of a line, however because we escaped it, our regex looks for the dollar sign.

Regular Expressions in Python

Python has a library calledre that lets you use regular expressions in Python programs. The symbols in Python regular expressions are not always the same asgrep , but the concept of regular expressions is the same.

```
#
```
https://github.com/calthoff/tstp/blob/master/part_III/regular_expressions/reg_expr_ex1.py

```
import re

line = "Match this."

matchObj = re.search( 'this' , line)

if matchObj:
   print (matchObj.group())
else:
   print ( "No match!" )
```

```
>> this
```

re comes with different methods like search, which returns the first occurrence of the pattern you are looking for. If a match is found, an SRE_Match object is returned, and we can get the match by calling the function group() . If no match is found, re.search() returns None

We can use the function findall() in the re module to find every occurrence of a pattern, instead of just the first match.

#
https://github.com/calthoff/tstp/blob/master/part_III/regular_expressions/reg_expr_ex2.py

```
import re

line = """The numbers 172 can be found on the back of the U.S. $5
          dollar bill in the bushes at the base
       of the Lincoln Memorial."""

matchObj = re.findall( '\d+' , line)

if matchObj:
   print matchObj
else :
   print ( "No match!" )
```

>> [‘172’, ‘5’]

re.findall() returns a list of all the strings matching the pattern, and an empty list if there is no match . The+ symbol matches the preceding character one or more times.
If we want to match everything in between two underscores, we can do it with:

#
https://github.com/calthoff/tstp/blob/master/part_III/regular_expressions/reg_expr_ex3.py

```
import re

line = """__yellow__ __red__ and __blue__ are colors"""

matchObj = re.findall( '__.*?__', line)

if matchObj:
   print matchObj
else :
   print ( "No match!" )
```

The two underscores match two underscores, the period matches any character, and the question mark followed by an asterisk means keep matching any character until there is another double underscore.

Here is a fun example of using regular expressions in Python to create the game Mad Libs:
“““https://github.com/calthoff/tstp/blob/master/part_III/lets_read_some_code/lets_read_some_

```
import re

text = """
```

```
    Giraffes have aroused the curiosity of __PLURAL_NOUN__ since
        earliest times. The giraffe is the
    tallest of all living __PLURAL_NOUN__, but scientists are unable
        to explain how it got its long
    __PART_OF_THE_BODY__. The giraffe's tremendous height, which
        might reach __NUMBER__ __PLURAL_NOUN__,comes from its legs and _
        _BODYPART__.
    """


def mad_libs (mls):
    """
    :param mls: String with parts the user should fill out surrounded
  by double underscores. Underscores cannot
  be inside hint e.g., no __hint_hint__ only __hint__.
  """
    hints = re.findall( "__.*?__" ,mls)
    if hints:
        for word in hints:
         new_word = input( "enter a {}" .format(word))
         mls = mls.replace(word , new_word ,1 )
        print ( \n ' )
        print (mls)
    else :
        print ('invalid mls" )

mad_libs(text)
```

Zen Challenge

Write a regular expression to find every word in the Zen of Python ending with the letter y.

Chapter 19. Package Managers

“Every programmer is an author.”
— Sercan Leylek

Package managers are programs that install and manage other programs on your operating system. An example of managing a program is keeping the program up to date when a new version comes out. In this chapter we are going to learn to use several package managers. Package managers are useful because programmers use programs to create new programs. For example, most programs use one or more databases (we learn to use a database in the last chapter of this section), and which are programs themselves. Programmers use package managers to download and keep their databases up to date, as well as to install and manage the wide variety of other programs they use in their craft.

In this chapter we will learn to use three package managers: apt-get, Homebrew, and pip. apt-get is a package manager for Ubuntu, Homebrew is a package manager for OS X, OneGet is a package manager for Windows and pip is a package manager that comes with Python and is used to download and manage Python programs.

Packages

A package is software “packaged” for distribution—it includes the files that make up the actual program, as well as files with metadata (data about data); such as the software’s name, version number, and dependencies (programs that need to be downloaded in order for it to run properly). Package managers download packages, install them—which means downloading any dependencies the package has.

Apt-get

Apt-get is a package manager that comes with Ubuntu. You cannot use apt-get from the online Bash command line shell emulator. You need to have Ubuntu installed so you can use the commandsudo . If you do not have Ubuntu installed on your computer, you can skip this section.

You install a package with sudo apt-get install [package_name]. Here is an example of installing a package named aptitude:

```
$ sudo apt-get install aptitude

>> Do you want to continue? [Y/N] Y
```

…
>> Processing triggers for libc-bin (2.19-0ubuntu6.7) …

Make sure to type Y when prompted with “Do you want to Continue [Y/N]”. The aptitude package should now be installed. You can use it by typing aptitude into the Bash command line shell:

$ aptitude
>>

This will open up Aptitude, a helpful program that shows information about packages. You should see a list of all the packages installed on your operating system under the option Installed Packages, as well as a list of packages you’ve yet to install, under Not Installed Packages.

You can also list the packages that have been installed with apt-get with the command apt list --installed :

$ apt list --installed
>> python3-requests...

You can remove packages using aptitude with the syntaxapt-get uninstall [package_name] . If you want to removeaptitude from your computer, you can uninstall it with sudo apt-get uninstall aptitude . That’s all there is to it—installing and removing programs is as simple as using two commands with apt-get .

Homebrew

Homebrew is a popular package manager for OS X. If you have a Mac, you can install Homebrew from the command line. Go to http://brew.sh/ and copy and paste the provided script in the terminal . Once installed, you should see a list of commands when you typebrew into your terminal:

$ brew
>> Example usage:
>> brew [info | home | options] [FORMULA...]
...

We can install packages with Homebrew using the syntax install [package_name]. Use the following command to install a package called calc :

$ brew install calc
...
>> ==> Pouring calc-2.12.5.0.el_capitan.bottle.tar.gz

```
>>  /usr/local/Cellar/calc/2.12.5.0: 518 files, 4.4M
```

A calculator program called calc is now installed. You can use it by typing calc from the command line (type quit to exit);

```
 $ calc
>> 2 + 2
>> 4
```

The command brew list prints the software you've installed using Homebrew:

```
$ brew list
>> calc
```

You can remove packages using Homebrew with the syntax brew uninstall [package_name] .If you want to remove calc from your computer, you can uninstall it with the command brew uninstall calc. That is all you need to know—you are ready to use Homebrew to manage your packages.

OneGet

OneGet is the first package manager to come with Windows: it was first released with Windows Ten. If you do not have Windows Ten, you can download OneGet by following the instructions on its GitHub page:https://github.com/OneGet/oneget.

We can install packages on OneGet using the command

pip

Pip is used to download Python packages. Pip will install a Python package for you, and once installed, you can import the program you downloaded as a module in your Python programs. First, check Pip is installed by going to either the Bash command line shell, or the Command Prompt if you are using Windows (which you can find by searching Command Prompt from the Run Window) and typing pip .

```
$ pip
 >> Usage:
      pip <command> [options]

      Commands:
       install                 Install packages.
       download                Download packages.
```

....

A list of commands you can use with pip should print. Pip comes with Python when you download it, but in earlier versions it didn't so if nothing happens when you enter pip, google "installing pip with easy_install".

We can use pip install [package_name] to install a new package. You can find all of the Python packages available for download at https://pypi.python.org/pypi . There are two ways to specify a package: you can just use the package name, or you can give the package name followed by == and the version number. If you use the package name, the most recent version will get downloaded. Sometimes you don't want the most recent version, you want a specific version, which is why the second option exists. Here is an example of how we can install a package called Flask, a popular Python package that lets you easily create websites, using a version number:

```
pip install Flask==0.10.1
>>
```

On Unix-like systems you need to use sudo:

```
sudo pip install Flask==0.10.1
>>
```

When the installation is finished, the flask module will be installed in a special folder on your computer called site-packages. Site-packages is automatically included in your Python path, so when Python imports a module, it looks in site-packages to see if it's there.

Now, if you write a program, you will be able to import and use the Flask module. Create a new Python file and add the following code:

```
from flask import Flask

app = Flask(__name__)


@ app.route ( '/' )
def index ():
   return "Hello, World!"

app.run( port = '8000' )
```

If you go to http://127.0.0.1:8000/ in your web browser, you will see a website that says "Hello, World!". This example was taken from Flask's tutorial, which is available at theselftaughtprogrammer.io/flask if you are interested in learning about how Flask works. You can view the packages you've installed with pip with the command pip freeze :

```
pip freeze
>> Flask==0.10.1
```

…

You can use the syntax `pip freeze > [filename]` to save names of all the packages you've installed with pip to a file. Create a requirements file with:

```
pip freeze > requirements.txt
>>
```

Open `requirements.txt` with a text editor to see the new file. This file is useful because you can use the syntax `pip install [requirements_file]` to install all the packages listed in a requirements file. This command is useful for quickly downloading the dependencies of a Python program that has not been listed on pypi, and thus is not available on pip.

Finally, you can uninstall programs you've downloaded with `pip uninstall [package_name]` . To uninstall Flask, use the following command:

```
pip uninstall flask .
…
>> Proceed (y/n)? y
...
```

Challenge

Find three programs that interest you on Ubuntu using `aptitude` and install them using `apt-get` .

Chapter 20. Version Control

"I object to doing things that computers can do."
— Olin Shivers

Writing software is a team sport. When you are working on a project with more than one person, you will both be making changes to the **codebase** —the folders and files that make up your software—and you need to keep those changes in sync. You could both periodically email each other with your changes, and combine the two different versions yourself, but that would quickly become tedious. Also what would happen if you both made changes to the same part of the project? Whose changes should be used? These are the kinds of problems a **version control system** solves. A version control system is software designed to let you easily collaborate on projects with other programmers.

There are many different version control systems.**Git** and **SVN** are both popular choices. Version control systems are programs, usually used in conjunction with a service that stores your software on the cloud, like **GitHub** . In this chapter we are going to use Git, a version control system, to put software on Github, a website that stores your code on the cloud.

Repositories

A **repository** is a data structure created by the program Git to manage a software project. A data structure is a way of organizing and storing information: lists and dictionaries are examples of data structures (you will learn more about data structures in Part IV). Repositories are used to keep track of all the changes in a software project.

When you are working on a project managed by Git, there are multiple repositories (usually one for each person working on the project). A typical situation looks like this: everybody working on the project also has their own repository on their computer called a **local repository** which keeps track of all the changes they make on their own computer; there is also a **central repository** hosted on a website like GitHub which all of the local repositories communicate with to stay in sync with each other. A programmer working on the project can update the central repository with the changes they've made in their local repository or they can update their local repository with the newest changes other programmers have made to the central repository.

image of repositories communicating

This is done from the command line using the program Git.

You can create a new repository using the Git program from the command line or on GitHub's website. Once you create a repository, you can use the Git program to manage it and communicate with a central repository.

Getting Started

To get started, you need to create a Github account: go to Github.com/join to create one.
Create a new repository on Github. Login to your GitHub account at github.com and click on the + button at the top right corner of the screen. Click Create repositoryfrom the dropdown menu. Give the repository the name hangman . Make itpublic , and checkinitialize the repository with a readme . Now click Create a repository. If at any time you run into trouble and do something wrong in this section, go to the settings page of your repository, delete it, and start over. You also need to install Git. You can install Git using your package manager of choice.
On GitHub, hit the button in the top right corner and select Your Profile.

image

You will see the name of your repository: hangman . Click on it. You will see a button that says Clone Or Download. When you click on it, you will see HTTPS: followed by a link. You can use this link to download your repository to your computer using the command git clone [repository_url] . The repository will download in whatever director you issue the command from. Copy the link, or press the copy link to clipboard button and use it with the git clone command:

```
$ git clone
>>
```

Use ls to verify the repository downloaded:

```
$ ls
>> my_project
```

Pushing and Pulling

There are two main things you will be doing with Git. The first is updating the central repository with changes from your local repository. This is called **pushing** because you are pushing new data to your central repository.

The second thing you will be doing is called **pulling**. Pulling data means updating your local repository with all of the new changes from the central repository.

The commandgit remote -v shows you what central repository your local repository i pushing and pulling from. The -v flag stands for verbose, which means the command will usually print out extra information . Use this command inside to see the central repository your local repository is pushing and pulling from:

```
$ git remote -v
```

```
>> origin https://github.com/[username]/my_git_project.git
  (fetch)
>> origin https://github.com/[username]/my_git_project.git (push)
```

The first line shows the central repository your project will pull data from and the second line shows the central repository your project will push data to. Generally, you will push and pull from the same central repository.

Pushing Example

In this section, you will make a change to the local repository for your hangman project and push that change to your central repository.

Move the Python file with the code we used to create Hangman into the hangman directory we created in Getting Started. Our local repository now has a file that does not exist in our central repository—our local repository is out of sync with our central repository. We can fix this by pushing the changes we made in our local repository to our central repository.

Pushing changes from your local repository to your central repository happens in three steps. First you stage your files which is where you tell Git which files have changes that you want to push to your central repository. Once you've reviewed everything, you commit the files.

, then you commit them and finally you push them. In the first step you tell Git what files you want to push to our central repository. This is called staging a file. When a file is staged, we have the chance to change our mind and unstage it. The syntax git add [file_path] is used to stage a file. Use the command git add hangman.py to stage our file:

```
$ git add hangman.py
>>
```

The command git status shows you the current state of your project in relation to your repository. New files that exist in your local project, but do not exist in your repository are shown in green, as are deleted files. Files that have been modified are in red. A file is modified if it is in your repository, but local version and the one in your repository differ. Enter the command git status :

```
$ git status
>> On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   hangman.py
```

You should see the file hangman.py in a green font. You have to stage each file you want to push to your repository with the commandgit add [file] . If you stage a file and change your mind, you can unstage it without making changes to your central repository. You can unstage a file with the syntaxgit reset [file_path] . Unstage hangman.py with the command git reset hangman.py. Use git status to see the result, and add it again withgit add hangman.py :

```
$ git reset hangman.py
$ git status

>>

$ git add hangman.py

>>
```

Once we've staged our files, and everything looks the way we want, we are ready to move to the next step—committing our files to our central repository. When you commit files, you want to use the-m flag so you can pass along a message. The message will be saved along with your commit in your repository to help you remember what changes you made in that commit.

```
$ git commit -m "my first commit"

>>  1 file changed, 1 insertion(+)
create mode 100644 hello_world.py
```

The final step is to actually push your changes to GitHub. This is done with the command git push origin master :

```
$ git push origin master
>> Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 309 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/[your_username]/my_project.git
   0eb3a47..48acc38  master -> master
```

Once you enter your GitHub username and password from the command line, your changes will be pushed to GitHub. If you look at your repository on GitHub's website, you will see hangman.py is now in your project.

Pulling Example

In this section, we learn how to update your repository with changes from the central repository. First, we have to make a change in our central repository. Use the command git pull origin master to update your local repository with the change we made:

```
$ git pull origin master
>>From https://github.com/calthoff/my_project
>> * branch            master     -> FETCH_HEAD
```

Git applied the changes from our central repository to our local repository, and they are now in sync. You can view the README.md file on your computer to see the change.

Reverting Versions

When you use version control, the entire history of your project is saved and available for you to use. If you decide you want to revert to a version of your project from 10 days ago, you are able to do so. You can view your project's history of commits with git log , which should output something like this:

```
$ git log

commit aeb4ef3cf3aabdb9205ea9e96e8cab5c0f5ca7ea
  Author: Cory Althoff <coryalthoff@Corys-MacBook-Pro.local>
  Date:   Thu Jan 21 13:52:02 2016 -0800
```

The string of numbers and letters after commit is the commit number. We can use this number to revert our project to exactly how it was at that time. We can travel back in time with the commandgit checkout [old commit] . In this case the command would be git checkout aeb4ef3cf3aabdb9205ea9e96e8cab5c0f5ca7ea .

diff

We can use the command git diff to see the difference between the version of a file in our local project, and the version in our repository. Add x=100 to the second line of our hello_world.py file. Enter the command:

```
$  git diff hello_world.py
>>diff --git a/hello_world.py b/hello_world.py
index b376c99..83f9007 100644
--- a/hello_world.py
+++ b/hello_world.py
@@ -1 +1,2 @@
```

```
print('hello')
+x = 100
```

Git highlights +x=100 in green because the line changed, the + is to signify the line x=100 was added.

The Other Pull Request

Confusingly, there are two concepts named pull in version control. We previously talked about pulling data from your repository. There is also an unrelated concept called a pull request. A pull request takes place on GitHub. If you are working on a branch of a project, and you want to merge it with the master repository, you would issue a pull request on GitHub to merge the two. This gives your teammates the chance to review and comment on your changes. If everything looks good, someone on your team can approve the pull request and merge the two branches.

Learning More

You typically create a new branch when you want to develop a new feature for your program or fix a bug or problem. When you are finished with whatever you are doing on your branch, you normally merge your changes with the master branch, which is the process of combining the two branches. Covering merging is outside the scope of this book, but git-scm has a great tutorial on it you can check out at: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging.

Challenge

Chapter 21. SQLite

“Data! Data! Data! I can’t make bricks without clay!”
—Sir Arthur Conan Doyle

Databases are programs used to persist data. Data persists if it outlives the process that created it 46 . Most of the programs we’ve built so far work fine without persisting any data, with one notable exception—our web scraping program we built in part II to collect headlines from Google News. All of the headlines we collect are lost after the program stops running. But what if we want to analyze the headlines from the last year? This is where persistence comes in, and why databases are important. Databases perform two main operations—read and write. When you write to a database, you are giving it information to store. When you read from a database, you are retrieving information from it. In an increasingly data-centric world, databases are becoming exceedingly important. In this chapter, we go over the basics of databases.

NoSQL vs. SQL

Relational databases were first proposed by Edgar F. Codd in 1970. Relational databases store data like an Excel spreadsheet—data is stored in rows and columns. The data is stored and retrieved using the query language SQL, which stands for Structured Query Language. PostGreSQL and MySQL are examples of popular relational databases.

Recently a new breed of databases, called NoSQL have gained popularity. NoSQL literally means “no SQL.” In other words, the thing these new breed of databases have in common is they are not relational databases using SQL. Redis is an example of a popular NoSQL databases. Redis is a “key value store” which means you can store data like a dictionary in Python, i.e., store a key and a value. So for example you could store “Monty” as a key and “Python” as a value; query Redis for “Monty” and it will return “Python”. While NoSQL databases are useful, , in this chapter we are learning about relational databases, because they are very important in the software industry. Once you are familiar with relational databases, I encourage you to learn more about NoSQL.

Getting Started

We will get started with SQL by using SQLite. SQLite is a lightweight database that comes built in to your operating system, so we don’t have to install anything. Go to the command line and type the command “ sqlite self_taught.db” to create a new database. If you

are coming back to You can open SQLite anytime with the command “sqlite3”. If you already created the database, open it with “.open self_taught.db”. Exit SQLite with the command “.exit”.

#explain relational databases what is a column, what is a row

Data Types

Create a Table

```
CREATE TABLE customers  (
   id int
   first_name   text
   last_name text  ,
   date_created date  ,
   PRIMARY KEY (  column1)
);
```

Every table in a relational database has to have something called a primary key —a unique identifier for each row. In this case our primary key is our first column—called id. We can set our primary key to auto increment, which means the primary key will be a number that the database automatically increments for you whenever you add new data. Our customers table also has a row for the customer’s first name and last name.

Relational databases are made up of tables that store data like an Excel spreadsheet, in columns and rows. Let’s create our first tabl e. Type the following inside sqlite. Make sure you are inside SQLite with the command “sqlite3” and your command line says sqlite.

```
CREATE TABLE self_taught.bdfl(
name string
project string
age int
birthday date
);
```

Our new in our database self_taught is called “bdfl”, which stands for benevolent dictator for life. BDFL is a title given to the creators of open source programming projects like Linus Torvalds, creator of Linux, Guido van Rossum creator of Python, David Heinemeier Hansson creator of Ruby on Rails or Matt Mullenweg creator of Wordpress . We will use our table to store data about these bdfls, such as their name, project, and birthdays.

Constraints

find somewhere to put # Creating a relationship with another table is done with SQL, which makes sure the database is aware of the relationship. You can't just enter any integer under the customer column, your database will only let you enter a valid primary key for a customer in the customer table.
A constraint is a rule you can apply to a column that is enforced by the database. Examples of constraints are: not null, unique, primary key and foreign key.

If you put the constraint"not null" on a column, that column cannot be "null" which is like "None" in Python. This means the column must have data in it. Say you are collecting the first, middle and last name of subscribers for a newsletter on your website. The table in your database to collect this information might look like this:

subscribers
first | middle | last

You probably would want to put "not null" constraints on the first column, while allowing "null" in the middle and last columns. The reason being that everyone has a first name, but not everyone has a middle or last name. What if Prince signed up for your newsletter? If you put a not null constraint on last, Prince wouldn't be able to sign up.

Constraints are important because they make guarantees about data. In this case, if we might want to create a program that analyzes the first names of all of our subscribers, our program would probably perform some sort of operation on each string. If the "first" column gave our program a null value instead of a string, and our program treated the null value like a string, it would cause an error in our program. By adding a not null constraint to our first column, our program can now be assured that every first name it gets from the database is going to be a string and our program can treat it as such.

If we are collecting data for a newsletter, we also need to collect our users' email addresses. We can do this by adding an email column to our table:

```
ALTER TABLE customer
  ADD email string UNIQUE
```

We added a "unique" constraint to this new column, so that the email column must be unique. This means if an email is used in one row of our table, it cannot be used in another row. This makes sense because every email address in the world is unique, and if two subscribers have the same email, there is a problem.

A foreign key lets you enforce that the data entered in the table is a primary key in another table. We already saw an example of using a foreign key in our Amazon example. Here is how we could change a table to add a foreign key P_Id to a made up table called Persons:

```
ALTER TABLE customer
  ADD FOREIGN KEY (P_Id)
  REFERENCES Persons(P_Id)
```

A check constraint is used to make sure data entered in a table meets certain specifications. Here is an example of a check constraint we could add to our email column:

```
ALTER TABLE customer
  ADD CHECK email varchar(255)
```

This enforces that all emails entered into our database have to be less than 255 characters.

Insert Data

Time to insert data about our first BDFL into our table. Enter the following SQL statements one at a time:

```
INSERT INTO bdfl (name, project,age)
VALUES (Guido van Rossum,Python,1-31-1956);

INSERT INTO bdfl(name,project,age)
VALUES(David Heinemeier Hansson,Ruby on Rails,10-15-1979);

INSERT INTO bdfl(name,project,age)
VALUES(Linus Torvalds,Linux,12-28-1969);

INSERT INTO bdfl(name,project,age)
VALUES(Matt Mullenweg,WordPress,1-11-1984);

INSERT INTO bdfl(name,project,age)
VALUES(Dries Buytaert,Drupal,11-19-1978);

    INSERT   INTO bdfl(name,project,age)
      VALUES(Larry Wall,Perl,9-27-1954);
```

Query Data

Querying data means looking up data in your database given specific parameters. Example of the parameters you might give are what table the data is in, and different attributes you want the data to contain. Let's start by querying for everything in our bdfl table by entering SELECT* from bdfl into sqlite. In SQL, SELECT * means select everything.

If we do not want to select everything, and just want to select the name of the bdfl of Linux, we can do this with SELECT name FROM bfdfl WHERE project = "Linux"

or Query

You can add “or” to your query to select from a row if either the condition before the or, or the condition after the or is true. Select everything from our table where the project is Ruby on Rails or Wordpress with the statement SELECT * FROM bfdl WHERE project = “Linux” OR
project=”Ruby on Rails”.

and Query

Adding “and” to your query will only select a row if both conditions are true. First try SELECT* FROM bfdl WHERE bfdl.name LIKE D%. This returns any row where the name starts with a D. It should return the data stored for David Heinemeier Hansson and Dries Buytaert.

Now try “” SELECT* FROM bfdl WHERE name LIKE “D%” AND where A.Date >= Convert(datetime, '2010-04-01') . This will only return David Heinemeier Hansson. While Dries Buytaert starts with “D”, his birthday is not greater than the date we provided.

Count

Count the number of rows in a table with:

```
SELECT COUNT(*) FROM bfdl
>>
```

Communicating with Databases

So far, we’ve learned to communicate with a database by using SQL from the terminal. But what if we want to store data collected in a database fr om a program we’ve written in a database? There is tool we can use to abstract SQL away entirely called an ORM, or object relational map. An object relational map is software that lets you interact with your database using classes in your programming language, instead of using SQL.

Using an ORM library, such as the ORM that comes with the Python web framework Django, we can represent a database table as a class in Python. Instead of using SQL statements to read and write from our table, we can use methods from the table class we

defined. Here is an example of how we could represent and query our Amazon customer table using Django's ORM:

```
from django.db import models

class Customer(models.Model):
    username = models.CharField()
```

This example defines a new database table called "customer." In this case we have two columns, id which is our primary key automatically created by our ORM and not reflected in our code, and username which is a variable set to a CharField object which Django uses to create a database column that stores strings.

Now, when we want to interact with our database, we can do it using our Customer class. For example, we can create a new user, which will be stored in MySQL, with the following code:

```
user1 = Customer.objects.create(username="eddie")
```

To query our database we simply need to use the class we created:

```
eddie = Customer.objects.get(username="eddie")
print eddie.username

>> "eddie"
```

With this code, we were able to successfully read and write from our database without any SQL. Django's ORM translates our code to SQL for us, so we don't have to worry about it.

Challenge

Chapter 22. Bringing It All Together

“The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be”....
— Frederick Brooks

In this chapter, we will see how powerful programming can be by building a web scraper: a program that extracts data from a website. Learning to scrape data from websites is powerful. It gives you the ability to extract any data you want from the largest collection of information that has ever existed. Seeing the power of web scrapers, and how easy they are to build, is one of the reasons I got hooked up on programming, and I hope it has the same effect on you.

HTML

Before we build our web scraper, we need a quick primer on HTML, or hypertext markup language. HTML is one of the fundamental technologies used to build websites, along with CSS and JavaScript. You can think of HTML as its own little language, used to give a website structure. HTML is made up of tags that a web browser uses to layout a web page. In fact, you can build an entire website using only HTML. It won’t be interactive or look very good, because JavaScript is what makes websites interactive, and CSS is what gives them style, but it will be a website. Here is an example of a website that will display the text Hello, World!:

```
<!--This is a comment in HTML. Save this file as index.html-->

<html lang="en" >
<head>
  <meta charset=  "UTF-8">
  <title>My Website </title>
</head>
<body>
    Hello, World!
  <a href= "https://news.ycombinator.com/" >  here </a>
</body>
</html>
```

Take a minute to study this code. Now save this HTML into a file and open the file with your web browser by clicking on the file (you may have to right click and change the default

program to open the file with to a web browser like Chrome). Once you open the file with your web browser, you will see a website that says Hello World!with a link to the Y Combinator website.

Your web browser uses the different tags in our HTML to figure out how to display this website. Tags have a beginning tag and closing tag, often with something like text in between. For example, your browser displays the text in between the <title> </title> tags in the tab of your browser. Anything in between the <body> </body> tags, makes up the actual website. Anything inside <a> tags is a link. There is a lot more to HTML, but this is all you need to know in order to build your first web scraper.

Scrape Google News

Now we can build a scraper that fetches all of the headlines from Google News. We will do this by extracting all of the <a> tags in Google News's HTML. As we saw in our HTML example, each <a> tag has a variable in it called href , e.g., . We are going to extract all of the hrefvariables from all of the<a> tags on Google News's website. In other words, we are going to collect all of the URLs Google News is linking to at the time we run our program. We will use the BeautifulSoup library for parsing our HTML (converting HTML to Python objects), so first install it with:

pip install beautifulsoup4==4.4.1

Once BeautifulSoup is installed, we can get Google News's HTML using Python's built-in urllib2 library for working with URLs. Start by importing urllib2 and BeautifulSoup:

""""https://github.com/calthoff/tstp/blob/master/part_III/lets_read_some_code/lets_read_

```
import urllib2
from bs4 import BeautifulSoup
```

Next we create a scraper class

```
class Scraper :
    def __init__ ( self , site):
        self.site = site

    def scrape ( self ):
        pass
```

Our method takes a website to scrape from, and has a method called scrape which we are going to call whenever we want to scrape data from the website we passed in.

Now we can start defining our scrape method.

```
def scrape ( self ):
  response = urllib2.urlopen(self .site)
   html = response.read()
```

The urlopen() function makes a request to Google News and returns a response object,which includes Google News's HTML in it as a variable. We save the response in our response variable and assign the variable html to response.read() which returns the HTML from Google News. All of the HTML from Google News is now saved in the variable html . This is all we need in order to extract data from Google News. However, we still need to parse the HTML. Parsing HTML means reading it into our program and giving it structure with our code, such as turning each HTML tag into a Python object, which we can do using the Beautiful Soup library. First, we create a BeautifulSoup object and pass in our html variable and the string 'html.parser'as a parameter to let Beautiful Soup know we are parsing HTML:

```
def scrape ( self ):
  response = urllib2.urlopen(self .site)
  html = response.read()
  soup  = BeautifulSoup(html , 'html.parser' )
```

Moving forward, we can now print out the links from Google News with:

```
def scrape ( self ):
  response = urllib2.urlopen(self .site)
  html = response.read()
  soup = BeautifulSoup(html , 'html.parser' )
   for tag in soup.find_all( 'a' ):
     url = tag.get( 'href')
      if url and'html' in url:
        print ( "\n  " + url)
```

find_all() is a method we can call onBeautifulSoup objects. It takes a string representing an HTML tag as a parameter ('a' representing <a>), and returns aResultSet object containing all the Tag objects found byfind_all(). The ResultSet is similar to a list— you can iterate through it (we never save ResultSet in a variable, it is simply the value returned by soup.find_all('a')), and each time through the loop there is a new variabletag, representing a tag object. We call the method get() on the tag object, passing in the string'href' as a parameter (href is the part of an HTML tag which holds the URL), and it returns a string URL which we store in the variable url.

The last thing we do is to check to make sure URL is notNone with if url, because we don't want to print the url if it is empty. We also make sure'html' is in theurl , because we don't want to print Google's internal links. If the url passes both of these tests, we use '\n' to print a newline and then print the url. Here is our full program:

```
import urllib2
```

```
from bs4 import BeautifulSoup

class Scraper:
    def __init__(self, site):
        self.site = site

    def scrape(self):
        response = urllib2.urlopen(self.site)
        html = response.read()
        soup = BeautifulSoup(html, 'html.parser')
        for tag in soup.find_all('a'):
            url = tag.get('href')
            if url and 'html' in url:
                print("\n" + url)

Scraper().scrape('https://news.google.com/')
```

Run the scraper, and you should see a result similar to this:
https://www.washingtonpost.com/world/national-security/in-foreign-bribery-cases-leniency-offered-to-companies-that-turn-over-employees/2016/04/05/d7a24d94-fb43-11e5-9140-e61d062438bb_story.html

http://www.appeal-democrat.com/news/unit-apartment-complex-proposed-in-marysville/article_bd6ea9f2-fac3-11e5-bfaf-4fbe11089e5a.html

http://www.appeal-democrat.com/news/injuries-from-yuba-city-bar-violence-hospitalize-groom-to-be/article_03e46648-f54b-11e5-96b3-5bf32bfbf2b5.html

...

Now that you have all of Google News's headlines available in your program, the possibilities are limitless. You could write a program to analyze the most used words in the headlines, and build a word cloud to visualize it. You could build a program to analyze the sentiment of the headlines, and see if it has any correlation with the stock market. As you get better at web scraping, all of the information in the world will be open to you, and I hope that excites you as much as it excites me.

Challenge

Modify the Google Scraper to save the headlines in a file.

Chapter 23.Practice

Exercises

0. Build a scraper for another website.
0. Write a program and then revert to an earlier version of it using PyCharm.
0. Download pylint using pip, read the documentation for it and try it out.

Read

0. http://www.tutorialspoint.com/python/python_reg_expressions.htm

Part IV Introduction to Computer Science

Data Structures & Algorithms

“I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”
— Linus Torvalds

This chapter is a light introduction to algorithms and data structures, perhaps the most important subject in Computer Science. The title of the influential book *Algorithms + Data Structures = Programs* summarizes their importance. My goal in this chapter is to introduce you to the subject, and clarify some things I found confusing when I was learning about them (which I still am). In addition to reading this chapter, you definitely need to read more about algorithms and data structures outside of this book, and also spend a lot of time practicing the concepts introduced here. Many of the examples in this chapter come from the amazing book *Python Algorithms and Data Structures* by Brad Miller and David Ranum. It is one of my favorite books, available online for free at: http://interactivepython.org.

What Are Algorithms & Data Structures?

An algorithm is a series of steps that can be followed to solve a problem. The problem could be anything, like sorting or searching a list, or traversinge a tree.

A data structure is a way to store and organize information. Data structures are fundamental to programming, and whatever programming language you use will come with built-in data structures. Common data structures include hash tables, stacks, and lists. Data structures come with different tradeoffs, with certain data structures being better suited for specific tasks than others.

Big O Notation

In Computer Science, we solve problems using algorithms. But what if you come up with two different algorithms to solve the same problem? How do you decide which is best? Big O Notation gives you a framework for deciding if one algorithm is better than another by

looking at the number of steps each algorithm takes, and choosing the one that takes the least amount of steps.

We can use an equation like T(n) = n to describe an algorithm .

The following sections introduce some algorithms you should learn.

Modulo

The modulo operator "%" returns the remainder of two numbers when you divide them. For example "2 % 2" returns 0, whereas "3 % 2" returns 1. Modulo is helpful in solving a variety of problems like Fizzbuzz, a popular first interview question designed to weed out people that cannot program. The question was introduced by Imran Ghory in his blog post *Using FizzBuzz to Find Developers who Grok Coding* . If you know the correct approach to the problem, it is easy to solve, whereas if you don't, it can appear complicated. The problem is usually given as:

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

The key to solving this problem is using modulo. To solve this problem you simply need to iterate from 1 to 100 and check if each number is divisible by 3, 5 or both. Here is the solution:

```
def fizz_buzz ():
    for i  in range(0, 101):
        if  i % 3 == 0 and  i % 5 == 0:
            print 'FizzBuzz'
        elif i % 3 == 0:
            print 'Fizz'
        elif i % 5 == 0:
            print 'Buzz'
        else  :
            print i
```

We start by iterating through the numbers 1 to 100 with a for loop. Then we simply check each of the conditions. We need to check if the number is divisible by 3 or 5 first, because if it is, we can move to the next number. This is not true with being divisible by either 5 or 3, because if either are true, we still have to check if the number is divisible by both. We then can check if the number is divisible by 3 or 5 (in any order). Finally, if none of the conditions are true, we simply print the number.

Here is another problem where using modulo is the key to figuring out the answer:

Rotate an array of *n* elements to the right by *k* steps.

For example, with $n = 7$ and $k = 3$, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4], i.e., you are moving every element in the list k positions.

When you solve this problem, your first instinct might be to simply add n to the index of each number and move the number to the new position. The problem is this does not work. The key to solving this problem is once again modulo. Say we had a list of numbers from 1-12. We can think of this list as a clock. If we start at 12 o'clock, and go around the clock twelve hours, we are back to where we started.

We can achieve this by taking the current index, adding the keynew index and getting the remainder. If you take our list of 12 numbers and you want to calculate where the number 12 would be (starting at index 11) if we added 12 to the index, you can calculate that with 11 +

Bubble Sort

A sorting algorithm is an algorithm that takes a group of numbers and puts them in a certain order. There are many different algorithms such as selection sort, insertion sort, shell sort, merge sort, and quicksort. In this section we will implement bubble sort, a sorting algorithm that is not very efficient—but easy to understand and useful for understanding sorting algorithms. Here is an implementation of bubble sort:

```
def bubble_sort(num_list):
    """
    :param num_list: List of numbers
    :return : Sorted list of numbers
    """
    for i in range ( len(num_list)- 1 , 0 , - 1 ):
        for j in range (i):
            if num_list[j] > num_list[j +1 ]:
                temp = num_list[j]
                num_list[j] = num_list[j + 1 ]
                num_list[j + 1 ] = temp

my_list = [ 4 , 266, 9 , 24 , 44 , 54 , 41 , 89 , 20 ]
bubble_sort(my_list)
print (my_list)

>> [4, 9, 20, 24, 41, 44, 54, 89, 266]
```

In this algorithm

Sequential Search

Search algorithms are used to find a number in a list of numbers. Sequential search is a simple search algorithm that checks each number in the list one by one to see if it matches the number it is looking for. A sequential search is the way you search a deck of cards for a specific card. You go one by one through each card, if the card is the card you are looking for, you stop. If you make it through the entire deck without finding the card, you know the card isn't there. Here is an example of sequential search:

```
def sequential_search (number_list, number):
    """
    :param number_list: List of integers.
    :param number: Integer to look for.
    :return: True if the number is found otherwise false.
    """
    found = False
    for i in number_list:
        if i == number:
            found = True
            break
    return found

print sequential_search(range(0, 100), 2)
>> True
print sequential_search(range(0, 100), 202)
>> False
```

First "found" is set to false. Then we loop through every number in the list and check if it is equal to the number we are looking for. If it equal to the number we are looking for, we set found to True, exit our loop and return True. Otherwise, we continue to the next number in the list. If we get through the entire list and found has never been set to True, w. We return found which will still be set to False.

Sequential search is an O(n) algorithm. In the best case scenario, the number we are looking for could be the first number in the list, in which case our algorithm would take only one step. However, in the worst case scenario, the number is not in the list and we have to check every single number in the list, or n steps. If we have a million items in our list, worst case we have to search through a million items. If we have a billion items, worst case we have to search through a billion items. As our list grows, the worst case scenario for our algorithm grows by the size of our list, making this algorithm O(n).

Binary Search

Binary search is a logarithmic algorithm used to search for numbers in a list, but the numbers have to be ordered. Remember this:, in order for an algorithm to be logarithmic, it needs to either be dividing or multiplying to a solution. Any guesses how binary search works? Binary search works by continually cutting the list in half. The algorithm picks a number in the middle of a list, and looks at whether it's the right number. If it is the right number, the search is complete. If it's not the right number, the algorithm throws away half the list . If the number was too big, it throws away everything above the number it selected. If the number was too small, it throws away everything below the number it selected.

Image we have an ordered list from 1 to 10, and we want to search for the number 3. Our list looks like this:

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

Our algorithm would first pick the number 5 because it's in the middle of the list. Since 5 is not the number we are looking for, and 3 is smaller than five, our algorithm would throw out everything above 5. Now our list looks like this:

[1,2,3,4,5]

Our list would now pick the number three, since it's in the middle of the list. Since 3 is the number we are looking for, our algorithm would stop and return that the number 3 was found in our list. Notice our algorithm only took two steps to figure out three was in our list. If we searched through the list linearly, one by one, looking for the number three, it would takes us three steps. Here is an example of a binary search algorithm searching for the number 3:

```
def binary_search (number_list, number):
    """Logarithmic binary search algorithm.
   :param number_list: List of ordered integers.
   :param number: Integer to search for in the passed in list.
   :return: True if the number is found, otherwise False.
   """
    first = 0
  last = len(number_list)-1
  number_found = False

   while first <= last and not number_found:
      middle = (first + last)/2
       if number_list[middle] == number:
          number_found = True
       else :
           if number < number_list[middle]:
              last = middle - 1
           else :
              first = middle + 1
   return number_found
```

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

We use a while loop that continues as long as the variable first is not greater than or equal to the variable last, and the variable not_true is False.

We calculate the middle index of the list by adding the first index of the list with the last index of the list and dividing them by two. The reason we subtract one from last is because the length of a list is calculated starting from one, whereas indexes start at zero.

We check to see if the middle number is the number we are looking for by looking up the middle number in our list with number_list[middle]. If the middle number is the number we are looking for, not_true is set to True, and we exit the loop. Otherwise, we check to see if the number we are looking for is bigger or smaller than the middle number. If our middle number is smaller than the number we are looking for, we change last to the middle index - 1, which on the next loop will split the list in half with everything smaller than our current middle number, whereas if our middle number is bigger than the number we are looking for, we change first to middle + 1, dividing the list in half so it contains everything bigger than our middle number.

Recursion

Recursion is notorious as one of the toughest concepts for new programmers to grasp. If it is confusing to you at first, don't worry, it's confusing to everyone. Recursion is a method of solving problems by breaking the problem up into smaller and smaller pieces until it can be easily solved. This is achieved with a function that calls itself. Any problem that can be solved recursively can also be solved iteratively, however in certain cases, recursion offers a more elegant solution.

A recursive algorithm must follow the three laws of recursion:
"1. A recursive algorithm must have a **base case**.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself, recursively."

Let's go over an example of a recursive function that has all three, a function to print out the lyrics to the popular children's song "99 Bottles of Beer on the Wall"19 :

```
def bottles_of_beer(bob):
    """ Use recursion to print the bottles of beer song.
    :param bob: Integer number of beers that arestart on the wall.
    """
    if bob < 1:
        print "No more bottles of beer on the wall. No more bottles of beer."
        return
    tmp = bob
```

```
    bob -= 1
    print "{} bottles of beer on the wall. {} bottles of beer. Take one down, pass it around,
{} bottles of beer on the wall.".format(tmp, tmp, bob)
    bottles_of_beer(bob)

bottles_of_beer(99)
```

>> 99 bottles of beer on the wall. 99 bottles of beer. Take one down, pass it around, 98 bottles of beer on the wall.

>> 98 bottles of beer on the wall. 98 bottles of beer. Take one down, pass it around, 97 bottles of beer on the wall.

…

>>No more bottles of beer on the wall. No more bottles of beer.

In this example, the base case is:

```
if bob < 1:
    print "No more bottles of beer on the wall. No more bottles of beer."
    return
```

The base case of a recursive algorithm is what finally stops the algorithm from running. If you have a recursive algorithm without a base case, it will continue to call itself forever, and you will get a runtime error saying the maximum recursion depth has been exceeded. The line:

```
bob -= 1
```

satisfies our second rule that we must move toward our base case. In our example, we passed in the number “99” to our function as the parameter bob. Since our base case is bob being less than 1, and because bob starts at 99, without this line we will never reach our base case, which will also give us a maximum recursion depth runtime error.

Our final rule is satisfied with :

```
bottles_of_beer(bob)
```

With this line, we are calling our function. The function will get called again, but with one important difference. Instead of passing in 99 like the first time the function was called, this time 98 will be passed in, because of rule number two. The third the function is called 97 will be called. This will continue to happen until eventually bob is equal to 0, and we hit our base case. At that point we print “No more bottles of beer on the wall. No more bottles of beer.” and return, signaling the algorithm to stop.

Let’s go over one more recursive algorithm. Say you are given the following problem:

Given a non-negative integer num, repeatedly add all its digits until the result has only one digit.

For example:

Given num = 38, the process is like: 3 + 8 = 11, 1 + 1 = 2. Since 2 has only one digit, return it.

One way we can solve this problem is using recursion:

```
def add_digits (number):
    """
    :param number: Int
    :return : Single digit int
    """
    number = str (number)
    if len(number) == 1 :
        return int (number)
    the_sum = 0
    for c in number:
        the_sum += int (c)
    return add_digits(the_sum)
print add_digits( 99 )
>> 9
```

In this example, our function accepts a number as a parameter, calculates its sum and calls itself until it hits the base case which is a number with only one digit. If we pass in the number ninety nine, it has two digits so it fails the base case. We then calculate that the_sum equals 18, so we call add_digits again and pass in 18. Once again 18 does not pass our base case so we calculate the sum of the digits which this time is 9, and call add_digits with 9 as a parameter. The third time around, our number is 9 and since it's only one digit, it does satisfy our base case and we return the answer.

Abstract Data Types

When learning about data structures, you will come across the term abstract data type. I remember what an abstract data type is by thinking of the relationship between an abstract data type and a data structure as similar (although not exactly the same) as the relationship between a class and an object.

If you want to model an orange in object-oriented programming, a class represents the idea of an orange, whereas the object represents the actual orange. Similarly, an abstract data type represents the idea of a certain type of data structure. An example of an abstract data type is a list. A list is an abstract data type that can be implemented several ways with data structures such as an array, or a linked list.

Nodes

Node is a term used frequently in Computer Science. You can think of a node as a point on a graph. The internet is made up of routers that communicate with each other. Each router is a node in the network. Nodes are used in several data structures, including linked lists, trees and graphs.

Stacks

A stack is a last-in-first-out data structure. This is best envisioned as a stack of dishes. Say you stack five dishes on top of each other. In order to get to the last dish in the stack, you have to remove all of the other dishes. This is how a stack data structure works. You put data into a stack. Every piece of data is like a dish, and you can only access the data by pulling out the data at the top of the stack. Here is an example of a stack implemented in Python:

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

The two most important methods in the definition are push and pop. Push lets you put data on top of the stack, and pop lets you take it off the stack.

So what are stacks used for? Well first of all, stacks are important for understanding recursion. Recursion is a fundamental part of programming we go over in a later section. Most new programmers struggle with recursion, but the key to understanding recursion is to deeply understand how a stack works.

Furthermore, stacks are used to reverse things. Whatever you put on a stack comes out in reverse order when you take it off. For example, let's say you want to reverse a string. We could reverse the string by first putting in on a stack, and then taking it off, like this:

```
from collections import stack as s

my_string = "Hello"
      stack = s()

for c in my_string:
 stack.push(c)

new_string = ""
for c in stack:
       new_string += c
print new_string
>>>olleH
```

In this example we went through each character in the word "Hello", and put it on our stack. Then we we iterated through our stack, and took everything we just put on the stack, off of it, and saved the order in the variable new_string. By the time we get to the last line, our word is reversed and our program prints "olleH".

I'm going to share another example of using a stack. It's a rare example of a question I've been asked to solve in an interview, and actually have used on the job. The problem is, write a program that tests a string for balanced parentheses. So, for example, "(hello)" would pass, but "(hello" would fail. "()()()" would pass, but "()(" would fail. This looks easy at first, until we get something like this "(((()(((()(((((())))))".

How are you supposed to keep track of all the parenthesis? The key to solving this problem is to use a stack. Every time we come across an open paren, we put it on a stack. If we come across a closed paren, we pull an open paren off the stack.

```
def balanced_paren (expression):
  stack = []
 for c in expression:
     if c == '(' :
       stack.append(c)
     elif c == ')':
        if len(stack) < 1 :
          return False
```

```
            stack.pop()
    if len(stack) == 0:
        return True
    return False
```

If the parenthesis are balanced, our stack will be empty after our loop, and we can return "True." One thing we need to watch out for, if the parenthesis are unbalanced, we will try to pop from an empty stack which will cause an error. That is why when we come across a closed paren, we have to first make sure the stack is not empty before we pop it off the stack. If we come across a closed paren, and the stack is empty, we know the parenthesis are not balanced and we can return "False." If at the end of the loop there are still open parenthesis on the stack, we can also return "False."

Linked Lists

A linked list is made up of a series of nodes, with each node pointing to the next node in the list. My friend Steve gave a great metaphor for thinking about linked lists. Imagine you are in the Mafia and want to give orders in such a way that no one knows who you are. You could set up a structure where you give an order anonymously to the next person down the chain of command. You know who they are but they don't know you, they only know the next person they should give the order to. The person they give the next order to doesn't know them, but only knows the next person to receive the order. This chain of information is what a singly linked list is. In a singly linked list, all of the nodes in the list only know about the next node. They don't keep track of the node behind them. You can get to every piece of data in a linked list by starting at the head of the list, and moving one by one to each next node. A doubly linked list is the same thing except each node keeps track of the node behind it, in addition to keeping track of the next node. A linked list can also be ordered or unordered. In this section, we will implement an unordered singly and doubly linked list:

```
class Node:
  """Class representing one node in a linked list."""
  def __init__(self, data):
     self.data = Node(data)
     self.next = None

class LinkedList:
  """Class representing a linked list data structure"""
  def __init__(self, head):
     self.head = head

  def add(self, data):
```

```
    """Add a new node to the linked list."""
    previous_head = self.head
    self.head = Node(data)
    self.head.next = previous_head
```

Our linked list class is simple. It stores the head of the linked list and has a method called “add” to add a new node to the list. Since the list is unordered, we don’t care where we put the next node, so it’s easiest to put it at the head of the list. The method add stores the current head in the previous_head variable, so we don’t lose track of it, creates a new node with the passed in data, and sets the new node as the head of the list. The node that used to be the head is then set as the next node after the new head. Now we can create a linked list and add data to it:

```
linked_list = LinkedList()
linked_list.add(1)
linked_list.add(2)
linked_list.add(3)
```

To get the data from our linked list, we start with the head, and visit every node until we hit a next node that equals none.

```
node = linked_list.head
 while node:
    print node.data
    node = node.next

  >> 3
>> 2
>> 1
```

We can change our singly linked list to a doubly linked list by keeping changing our node class to keep track of the node behind it.

```
class Node :
    """Class representing one node in a linked list."""
    def __init__( self , data):
      self .data = data
      self . next =None
      self .previous = None

class LinkedList
    """Class representing a linked list data structure"""
    def __init__( self , data):
      self .head = Node(data)

    def add ( self, data):
      """Add a new node to the linked list."""
```

```
        previous_head =self  .head
        selfhead = Node(data)
    previous_head.previous =  self  .head
        selfhead. next    = previous_head
```

Our new linked list is the same as our previous one, except now every node knows the node in front and back of it.

Arrays

An array is an implementation of the list abstract data type. Every piece of data in an array has to be the same type. So for example, you can have an array made up of strings or ints but you cannot have an array made up of both. Python's built in list data structure is implemented internally in C as an array of pointers.

Binary Trees

A tree is another data structure that gets its name from looking like, you guessed it, a tree. A great example of a tree data structure is the file system on your computer, which is implemented using a tree. There are many different kinds of trees such as red and black trees, AVL trees and binary trees. In this section, we will build a binary tree.

A binary tree is made up of nodes containing data. Each node can have a left child and a right child. In the same way a linked list points to the next element in the list, a node can point to two other nodes called the left and right child. Like a doubly linked list, the left and right child keep track of their parent. Here is an example of a binary tree implemented in Python:

```
class Node  :
    """Binary tree node."""
    def   __init__(self, value):
        self.value = value
        self.left_child = None
        self.right_child = None

class BinaryTree :
    """This class represents a binary tree data structure."""
    def   __init__(self, root):
        """
        :param root: Binary tree node.
```

```
        """
        self.root = root
```

Our binary tree is simply made up of a root. See next section for a tree with children.

Breadth First & Depth First Search

If we want to visit every node in a binary tree, there are two search algorithms we can use: breadth first search, and depth first search. If you think of a tree of being made up of rows and columns, in a breadth first search we visit each row one by one, whereas in depth first search we visit each column one by one. We will use the binary tree from the previous example to create our tree to traverse:

```
tree = BinaryTree("a")
tree.left = TreeNode("b")
tree.right = TreeNode("c")
tree.left.left = TreeNode("d")
tree.right.right = TreeNode("e")
```

Now we can write a function that takes a tree as a parameter and does a breadth first search of the tree, printing out the value of each node it visits:

```
def breadth_first (tree):
    """ Breadth first search of binary tree print out each node.
    :param root: BinaryTree
    """
    current_level = [root]
    next_level = []
    while current_level:
        for node in current_level:
            print node.val
            if node.left_child:
                next_level.append(node.left_child)
            if node.right_child:
                next_level.append(node.right_child)
        current_level = next_level
        next_level = []
```

We use the list current_level to keep track of all the nodes in the level of the tree we are currently in. When there is no more current level, our algorithm stops. Our tree looks like this:

```
a
```

```
  / \
   b   c
  / \
 d   e
```

“current_level” will start as [a], become [b, c], become [d, e] and finally become an empty list [] at which point our algorithm is finished. We are able to do this by keeping track of the next level of nodes in our list next_level.

Before our while loop, we add the root of our tree to current_level, then tell our while loop to continue, as long as current_level isn’t empty. In our while loop, we iterate through every node in current_level, printing out each node. Then we check if the node has any children. If it does, we add those children to our next_level list. At the end of our for loop, we switch the two lists, so the current_level list is set to the next_level list, and the next_level list becomes empty. This is what allows us to move from one level to the next. Eventually, when we reach the last level, the nodes will not have any children, next_level will be empty, current_level will be set to next_level, and because it’s empty the algorithm will stop.

A depth first search searches the tree vertically instead of traversing across it horizontally. Our tree from the previous example would be searched “a”, “b”, “d”, “e”, “c”. We can implement depth first search using recursion:

Hash Tables

In the chapter Containers, we covered Python’s built in dictionary data type. Dictionaries are helpful because they can store keys and values and are incredibly fast at getting and setting data. To recap, dictionaries map keys to values. For instance you could add the key “super_computer” to a dictionary with the value “Watson” with the following code:

```
my_dictionary = {}
my_dictionary[“super_computer”] = “Watson”
```

Now we can retrieve the key “super_computer” in with:

```
print my_dictionary[“super_computer”]
>>> Watson
```

The amazing thing about dictionaries is they can set and get data in constant time. It doesn’t matter how many rows of data we have in our dictionary. We could have one billion rows, and still add and retrieve the value for “super_computer” to our dictionary in O(1) time.

Internally, Python uses a hash table to implement its dictionary. A hash table is a data structure that uses a list and a hash function to store data in O(1) time. When you add a value to a hash table, it uses a hash function to come up with an index in the list to store the data. When you retrieve data from a hash table, it uses the same hash function to find the index so it

can retrieve it from the list. In this example, our hash table is only going to store numbers. The hash function will return the result of the number modulo eleven. So for example, our hash function for one would return one, so we store the number one at index one in our list. Our hash function for the number five would return five, and so we would store the number five at index five in our list. Here is an example of a hash table:

```
class HashTable:
    """Hash table data structure"""
    def __init__( self ):
        self .list = [ None] *  11

    @ staticmethod
    def hash(n):
        """
        :param n: int
        :return: return index in list to store number.
        """
        return n %11

    def set (self , n , v):
        """
        :param n: int
        :param v: can be any type.
        """
        self .list[self .hash(n)] = v

    def get (self , n):
        """
        :param n: int
        :return: int value from list
        """
        return self .list[hash (n)]

hash_table = HashTable()
hash_table.set( 1 , 'Disrupted')
hash_table.set( 5 , 'HubSpot' )
print (hash_table.get( 1))
print (hash_table.get( 5))
```

This is an oversimplified example that clearly has problems. However, the goal is to illustrate how a hash table works. bc

Challenge

Chapter X. Relational Database Design

When you create use a relational database, you have to design the different tables your database will have, how the tables relate to each other, what columns they will have, and what constraints are put on those columns. Together, this makes up your database schema. In this section, we are going to to design a schema to store data for a website like Amazon.com.

First, we need to think about the data Amazon needs to store. The first thing that comes to mind is products; Amazon clearly must have a database where they store all of their products. Amazon also has to keep track of customers—you don't have to register a new account every time you order something on Amazon, so they must store their customers information as well. A customer might order more than one product, so Amazon must also have a way to store orders. Let's start by designing a table to hold data about products:

product

_ _

id | name | price

1 | The Pragmatic Programmer | 14.99

Our product table has a column called id that serves as the primary key, and columns for the name of the product and the price. Our primary id is an integer that auto increments, the name column accepts strings, and the price column accepts integers . The data shown in the table such as 1, "The Pragmatic Programmer," and 14.99 are not part of the database design, but are an example of how data would look in our table. This convention is used throughout this chapter.

Note that when you design a database schema, you want to pick a naming convention and stick to it. In the following this case we will use lowercase letters and an underscore in between words. Now we need a table to keep track of our customers. Our customer table is very similar to our product table:

Customer

id | first_name | last_name

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _

1 | Steve | Smith

Our customer table has a primary key, and two columns that accept strings. This is all we need to keep track of our customers.

We are going to keep track of our orders using two different tables. The first table will map an order id to a specific customer, and the second table will keep track of the products in each order. Here is our first table shown along with the customer table:

customer
id | first_name | last_name
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
1 | Steve | Smith

order
_ _ _ _ _ _ _
id | customer
1 | 1

Our order table has a primary key called id and a column called customer. Our customer column is different than the rest of the columns we've seen so far because it uses a constraint called a foreign key (covered in the chapter SQL). The customer column of our order table accepts an integer that represents the primary key of a customer in our customer table. In our example, the first entry in the order table customer column is 1. If we look up the row with 1 as its primary key in in our customer table we would get the row " 1 | Steve | Smith". By using a foreign key, we've successfully linked our order table to our customer table. This is called creating a relationship.

Imagine if we decided to put the information from our "customer" table in the "order" table instead:

order
id | username | order
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
1 | Steve | NoSQL Distilled
2 | Cory | Think Python
3 | Steve | The Talent Code

The problem with this design is that data is duplicated in our table. The username Steve is repeated twice. If we needed to change Steve's username to "Steven," we might accidentally only change the name in the first row, and forget to change it in the third. This would corrupt our data:

order
id | username | order
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
1 | Steven | NoSQL Distilled
2 | Cory | Think Python
3 | Steve | The Talent Code

In our original design this is not possible. Take another look at our previous design:

customer
id | username

```
_ _ _ _ _ _ _ _ _
8  | Cory
9  | Steve

orders
 id | username | order

 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 1  | 9        | NoSQL Distilled
 2  | 8        | Think Python
 3  | 9        | The Talent Code
```

When we need to change a username, we only have to change it in one place—the customer table. Once we change the name in our customer table, anyone looking up username with a foreign key of 9 will see the customer's username is Steve. There is no chance of accidentally corrupting the data because it only exists in one location.

Tables can have three types of relationships: one to one, one to many and many to many. This is an example of a one to one relationship. You create both a one to one relationship and a many to one relationship using a foreign key. The difference is, in a one to one relationship, both tables can have foreign keys to each other, although like in this case, they don't have to. In a one to many relationship, only the many side has a foreign key linking it to the one. This is not something your database knows about, but rather a construct invented to help you design databases.

In this example, a customer can have many orders, but an order cannot have many customers. Another example of a many to one relationship is a classroom. A teach can have many classes, but a class cannot have many teachers. In a one to one relationship however, the relationship can go both ways. One person has one passport, and one passport has one person. The final relationship tables can have is called many to many. In order to do that we need to create a junction table, which we need to do in order to complete our Amazon design. Our final table order_item will keep track of products:

```
order_item

_ _ _ _ _ _ _ _ _ _
id | order_id | product_id
1  | 1        | 1
```

This table has id as a primary key, and two foreign keys—order_id and product_id linking the table to our order and product table. Our design is complete, we can store and lookup all of the information we need to fulfill an order. Here are all of our tables together:

```
product

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
id | name | price
1  | The Pragmatic Programmer |  14.99

customer
```

id | first_name | last_name

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _

1 | Steve | Smith

order

_ _ _ _ _ _ _

id | customer

1 | 1

order_item

_ _ _ _ _ _ _ _ _ _

id | order_id | product_id

1 | 1 | 1

2 | 1 | 1

If we are ready to ship an order, we can get all of the information we need by looking at the order_item table and using that information to query other tables. First we would select all the rows from our order_id with an order_id of 1. Then we would look up all the products using the product_id in each row. Finally, we would use the order_id key to lookup the name of the foreign customer key in our order table, and lookup the customer's name in the customer table using that information.

Normalization

One of the challenges you face when working with a database is maintaining data integrity, which means “assuring the accuracy and consistency of data over its entire life-cycle” 49 . Normalization and referential integrity are some of the concepts that help ensure data integrity.

Data normalization is the process of designing a relational database in order to reduce data redundancy, which can lead to inaccurate data. While there are many rules for data normalization, there are three specific rules that every database should follow. Each of these rules is called a “normal “form.” If the first rule is followed, the database is in "first normal form" or 1nf. If all three rules are followed, the database is in "third normal form" or 3nf. 52 In order to reach each successive level of normalization, all of the previous rules must be followed. In other words, if the rule for 2nf is satisfied, but 1nf is not, the database is not considered 2nf.

To reach the first normal form, you need to avoid duplicating data in multiple row, avoid storing more than one piece of information in a row and the table must have a primary key. Here is an example of storing duplicate data:

t-shirt

_ _ _ _ _ _ _ _ _ _ _ _ _
color | color
blue | blue

And an example of storing more than one piece of data in one row:

t-shirt

_ _ _ _ _ _
color
blue, large

In this example we are using a comma to store two pieces of data in one column—"blue" and "large." This is something you should never do. Furthermore, neither of these examples are 1nf because they do not have a primary key.

Here is an example of a table that is 1nf:

t-shirt
primary_key = id

_ _ _ _ _ _ _ _ _ _ _ _ _ _
id | color
1 blue

In order for a table to be 2nf, all non primary key columns must relate to the primary key. Let's look at an example that violates 2nf:

t-shirt
primary_key = item
primary_key = color

_ _ _ _ _ _ _ _ _ _ _
item | color |price| tax
t-shirt red 19.99 .90
t-shirt blue 18.00 .78
polo yellow 32 1.4
polo green 40 1.8
polo orange 43 2

This table is not 2nf because the two columns that are not primary keys, price and tax relate to item, but do not relate to color.

dealership

_ _ _ _ _ _ _ _
id |location | available
1 Portland |Yes

Normalization is an important part of database design. While there even more normalization rules we did not cover, it is important to always normalize your database to 3nf. To help you remember the rules, programmers often use the phrase “The data depends on the key [1NF], the whole key [2NF] and nothing but the key [3NF] so help me Codd ” (Codd, mentioned earlier, is the creator of relational databases) to help them remember the rules of of normalization.

Referential Integrity

Referential integrity is another way of ensuring data integrity. It is a measure of consistency in a database. If for example we have the following tables:

```
  customer
id | username
_ _ _ _ _ _ _ _ _
8  | Cory
9  | Steve

order
  id | username | order
  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
  1  | 9        | NoSQL Distilled
```

and we delete the second row from the customer table:

```
  customer
id | username
_ _ _ _ _ _ _ _ _
8  | Cory

order
  id | username | order
  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
  1  | 9        | NoSQL Distilled
```

Our username column in our order table references a foreign id that no longer exists. This is a violation of referential integrity. Fortunately, your database manages referential integrity for you. If you try to do this in a relational database, it won’t let you, you will get an error.

Indexing

You can index a column in a table in order to make reads faster. Indexes work like a telephone book sorted by last name. If you were looking through such a telephone book, you wouldn't look through every single entry, you would immediately skip to the section of the phonebook that matches the last name of the person you were looking for. This is what an index does. When you index a column in a database, internally the database duplicates the data in the column, but arranges it in a specific order —alphabetically for example, that allows it to lookup data faster. Here is an example of creating an index with SQL:

```
CREATE INDEX my_index
  ON table_name (customers)
```

This will cause our database to internally duplicate all of the data in our customer table and arrange it alphabetically. Now we can lookup customers much faster. The drawback to creating an index is that duplicating data and organizing it has a cost—it increases the time it takes to write to your database.

Challenge

Chapter X. Computer Architecture

“There are 10 kinds of people in the world — those who understand binary and those who don't.”
—Anonymous

Now that w e’ve covered the fundamentals of programming—and some tools to program more effectively—we are going to go over some of the basics of Computer Science. Computer Science is usually taught as abstrusely as possible—so I’ve attempted to make it as friendly as possible—while focusing on the most practical parts. In this chapter we take a look under the hood of what we’ve learned so far—we explore how Python, your operating system and computer work.

How Your Computer Works

Computers can only understand binary, so in order to understand how a computer works, you should have a basic understanding of binary. When you normally count, you count in base ten, which means we represent every number in the world using only ten digits. The “base” of a counting system is the number of numbers used to represent all the number in the world. In the base 10 counting system, once we get over the number nine, we recombine numbers from one to ten to create new numbers:

```
0  1 2 3 4 5 6 7 8 9
  11  1  21  3 14 15 16 17 18 19 20
```

Base two is a system of counting just like base ten. However, instead of combining existing numbers after ten numbers, base two starts doing it after two numbers

```
0 ->   0
  1 ->   1
  10 ->  2
  11 ->  3
```

0 and 1 are the same as base ten. However, once we get to 2, we’ve gone past two numbers, and we need to combine our first two numbers to create a new number. Hence one and zero are“10” is combined to represent 2. Each number starting from the left represents whether or not there is a power of 2. So for example, “ 10” means there are zero 2 ** 0’s and 1 2 **1:

```
  10
  2 + 0 = 2
```

Your computer is made up of hardware— a CPU, memory and input/output devices. All computing is done with these physical pieces of hardware. Hardware only understands binary. That means no matter what programming language you use, no matter what operating system you use, every instruction a computer ever executes is in binary.

[explain representing things in binary]
This chart shows how each character is mapped to a number. For example, 97 is mapped to the letter “a”. A program using ASCII would store an “a” as 1100001, or 97 in binary. When the same program needs to retrieve the letter “a”, it looks up 97 in the ASCII table and sees it represents the letter “a”.

You can use Python to easily get the number a character maps to in ASCII. The function “ord” takes a character and returns the number it maps to:

```
ord(“a”)
>> 97

ord(“z”)
>> 122
```

At a high level a computer is made up of a CPU and memory . The CPU is the part of a computer that executes the instructions provided by a program. A computer can have one or more CPUS: a computer with multiple CPUs is said to have a multi-core processor. A CPU has a clock generator that produces “clock cycles.” In a CISC, or Complex Instruction Set processor, each instruction takes one or more “clock cycles,” whereas RISC, or Reduced Instruction Set Computing processors, are faster, able to execute multiple instructions per clock cycle. Clock speed refers to the speed a microprocessor executes instructions.

python version of java CPU program in Structured Computer Organization

Computers have two types of memory, RAM and ROM. RAM stands for random access memory and is volatile, meaning it is erased when a computer turns off. All arithmetic and logic operations take place in ram. ROM stands for read only memory. It is not volatile; it persists after a computer is turned off and is used for the most fundamental parts of an operating system needed when a computer starts up.

Everything in memory is stored in binary, sequences of zeros and ones. The zeros and ones are stored in chips made up of millions or billions of transistors. Each transistor can be turned on or off using a flow of electricity. When a transistor is turned off it represents a zero in binary, and when a transistor is turned on it represents a one.

Memory is not where everything on a computer is stored, despite the name sounding like it is. Memory is used in combination with the CPU to execute tasks, like arithmetic operations. When you store information in a database for example, it is written to disk storage, not memory.

I/O

The CPU combined with memory make up is the brain of a computer. I/O is the transfer of data to or from the brain by other devices. Reading and writing data from disk storage is an example of I/O.

As we learned earlier, computers only understand binary. So how do computers work with characters? The answer is character encoding schemes like ASCII and Unicode. To a computer, characters do not exist. Characters are represented using encoding schemes like ASCII to map characters to binary numbers.

How Programming Languages Work

Machine code is code made up entirely of binary (or hex which is base 16 but ignore that for now) that can be executed directly by your computer's hardware. You can program every Python program we've written so far in binary.

Computer Scientists use a concept called abstraction to manage complexity. When something is an abstraction, it means you can use it without needing to understand how it works. When you program, you are programming beneath several layers of abstraction: you can program in Python without knowing how Python communicates with your operating system, how your operating system communicates with your computer's hardware, or how your computer's hardware executes binary.

Programming languages are built on abstractions. Machine code is the lowest level programming language—that means it has no abstractions beneath it. A programming language becomes higher-level the further away it gets from machine code (the direct instructions executed by the CPU). The higher-level a programming language is—the more abstractions it has beneath it. The more abstractions a programming language has beneath it the slower it runs. If you can write any Python program in machine code—and the programs will run faster—you might be wondering, "Why don't programmers write all of their programs in machine code?" The reason programmers don't write programs in machine code is because it is insanely tedious. People don't think like computers—they don't think purely in numbers— which led to the idea of writing a program in machine code that could translate a more human readable language into machine code, so humans could write in a language that was more natural to them, and this new program (called an assembler) would automatically translate assembly codeit to machine code (called an assembler)—this programming language is called Assembly, and it is the first abstraction above machine code and the program that translates assembly code to machine code. Here is an example of a program adding two numbers and storing them in a variable in machine code:

add machine code example

Here is the same program written in Assembly:

add assembly code example

As you can see, the Assembly language code represents the way human thinks much than the machine code; however, while assembly code is easier to read, the instructions still are one to one with machine code; that is, every line of assembly code translates to exactly one line of machine code—so programming in assembly is still tedious because it requires the programmer to write so many lines of code to get anything done. This led programmers to develop even higher level programming languages—like C—with instructions that are not one to one with machine language and thus let the programmer write programs with fewer lines of code. C code is first translated into assembly code, and then it is translated into machine code. Here the same program adding two numbers and storing them in a variable from the previous examples in C:

```
int x
x = 2 + 2
```

This is similar to how we would write this program in Python, although there are two steps instead of one. In C, unlike Python, Unlike Python—in C—you have to declare what type a variable is before you use it. This is a concept called static typing, which we cover later. Although this code looks familiar, C is a lower level language than Python, which is one abstraction level above it. C code gets converted to assembly, and then to machine, code by a program called a compiler. The difference between an assembler and a compiler is that a compiler includes extra functionality to optimize performance when it translates a higher level language to machine code (compilers are used in every programming language not just C). But the concept is still the same—a high level language is translated to machine code.

When you program in C, you have to manage the memory your program uses yourself; when you program in Python—your program's memory is managed for you, and you have no access to it. When you program in Python, you can create a list, and append as many items to that list as you'd like. When you program in C, this is not possible. When you define an array in C, you have to allocate a certain amount of memory to it. You can't just define an array—you have to include how many items will be in that array. If you define an array that can hold ten integers—and decide you actually need eleven—you have to create a brand new array. In Python, as we've seen, you simply create a list and append as many items to it as needed.

Python achieves this by adding one more layer of abstraction. Python code does not get compiled to machine code the way C code does. Instead, when you run a Python program, it is executed in two steps. First, the Python compiler (written in C) translates Python code to bytecode—a special kind of code consisting only of numbers—but meant to be executed by a virtual machine (software that emulates hardware). Python's virtual machine program is then executed by the hardware , and it goes line by line through the bytecode and executes each instruction. This design offers two important advantages. Because of this, you can write a program in Python without worrying about managing memory, an d why you ca

There are several different implementations of Python. The version of Python we are using is called CPython.

When you are programming in C, you have to compile your program before it will run The C compiler takes your code and translates it into machine code your computer can understand. Once compiled, you no longer need any sort of program to run your code, you can execute it directly.

Python code, however, must always be run using the “python” program. This is because Python is what is often called an “interpreted” language, a term used to differentiate it from a language like C, which is called a “compiled” language. This is confusing, because while Python has an interpreter, it also has a compiler. When you run a Python program, its compiler translates your code to something called bytecode, a special kind of code that is like binary but meant to be consumed by a virtual machine. At runtime, Python’s virtual machine translates the bytecode into machine code and executes it line by line.

These two approaches both have advantages and disadvantages. One advantage of C’s approach is speed—compiling directly to machine code makes C’s programs run faster than Python programs. C’s approach also allows for variables to be statically typed. This eliminates a class of errors, but also has drawbacks such as giving programmers less flexibility. Python’s approach is advantageous because it allows it to be platform-independent. Python’s use of an interpreter also allows its variables to be dynamically typed, which makes programming in Python much more flexible than C.

Programming languages can be either dynamically or statically typed. Python and Ruby are examples of dynamically typed languages. In a dynamically typed language, you can declare a variable as one type, and later change that same variable to a value of a different type. For example, this is allowed in Python:

```
x = 5
x = “Hello World”
```

In a statically typed language, trying to change a variable from one type to another will cause an error because in a statically typed language; once you declare a variable’s type, you cannot change it. Understanding the difference between statically and dynamically typed languages will payoff with hours of fun arguing with your friends with Computer Science degrees about whether or not the former is better than the latter.

Your computer is made up of physical hardware. This hardware is what runs our Python programs and it only understands one thing—binary. Binary is a counting system. Counting in binary is no different than when you normally count—which is called counting in base ten—except there are only two digits instead of ten digits. When we count in base ten, we start at zero, and when we get to nine we say oh no! We ran out of digits. We solve this by taking a digit we already have—zero— and putting it after the first digit (one) to create the number ten. Binary works the same way. Zero in binary is zero. One in binary is one. In binary, a zero or one is called a bit—short for binary digit. In binary, after the number one, we run out of

digits, and like base ten, we reuse digits we already have. We add a zero to the end of the next number—two—which becomes 10. We do the same thing for three—which becomes 11. Counting this way is called counting in base two.

(binary)
base 2 base 10

0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11

I cover how to convert a number in base 10 to its binary equivalent later in the book. For now, just understanding what binary is will suffice, and allows us to explore a fundamental programming concept—data types, or types for short.

At the beginning of this section we learned computers only understand binary. Python needs to store both integers and strings in your computers memory (the part of your computer that saves data and can only store binary)—so how can Python store integers and strings in memory? We already went over how the number two is represented in binary—so it is easy to understand how Python stores integers like 2 in your computer's memory—it just converts them to binary. But how does your Python represent a string like "z" when it talks to your computer? Python represents "z" just like the number two—in binary (everything is in binary!). To your computer "z" is:

01111010

01111010 is also the number 122 in binary. Since everything must be represented in binary, Python represents strings (like "z" or "a") with binary numbers, and it has a table that maps each binary number to a character in the alphabet. This table is called a unicode table. You can check out a cool example of a unicode table here: http://unicode-table.com/en/#0046 (click on a letter and then click on the link in the popup to see the binary). 01111010 is mapped to "z". So 01111010 can represent either the number 122 or "z". When you put quotes around "z", the quotes let Python know you are representing a string and not another type like an integer. It takes the letter in quotes and looks up in the unicode table which binary number represents "z" and then uses that when it talks to your computer. That is why type is so important. A computer can only understand binary and so programming languages like

Python have to differentiate between different types of data—like strings and integers—so it can know how to represent them to your computer.

How Your Operating System Works

An operating system is the software that manages a computer's hardware, allowing you to use your computer. Your computer has a limited amount of resources such as memory and CPU, and your operating system determines the resources each program receives, along with creating a structure for managing files, managing different users and managing other common operations needed by programmers.

The kernel is the most fundamental part of an operating system, responsible for allocating resources like CPU and memory to different processes. Processes are programs that are executing. The kernel assigns memory and a stack to each new program when it starts running. The state of the current process is saved in a data structure called a process control block.

The kernel cannot be accessed directly, so there is another layer of software built on top of the kernel in order to access it called the shell (because it is a shell around the kernel). We learned how to use the shell in the chapter The Command Line in Part III. Operating systems have other responsibilities other than resource sharing, but in order to limit the size of the kernel's code, other operating system jobs (called daemons) are run alongside user programs.

When the kernel switches from one process to another, it is called a context switch.

InterruptsInterupts

The following explanation on Stack Overflow helped me understand the difference between concurrency and parallelism:

"Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn''t necessarily mean they'll ever both be running at the same instant. E.g., multitasking on a single-core machine.

Parallelism is when tasks literally run at the same time, e.g., on a multicore processor." RichieHindle 4.

How can a single core processor run multiple tasks at once?

Challenge

Chapter X. Network Programming

In this chapter, we look into how computers communicate with each other over networks. A network is a group of computers connected through software and hardware that allows them to exchange messages.[54] . The Internet is an example of a network. In this chapter we will explore the foundation of the internet— the client server model and the TCP/IP protocol. Then we will dive deeper into these subjects by building both a client and a server.

Client-Server Model

The Internet communicates using the client-server model. In the client-server model, there is a server actively listening for requests (like Google), sent by a client (your web browser). Clients send requests to servers asking for the resources they need to render a webpage, and if everything goes well, the server responds by sending the resources to the browser. Requests are made using HTTP, or hypertext transfer protocol, which we cover in the next section. When I say resources, I mean the HTML, JavaScript and CSS files the browser needs to display a website along with any images. When you visit Google's website, you are seeing the client-server model in action. Google waits for you to send a request, and responds with the resources your web browser needs to display Google's website to you.

Try going to Google in your browser and copy and paste the URL into a word processor. You will see there is a slash added to the end of the url. That is because when you go to a website like Google, you are really going to " http://www.google.com/ ". The "/" is referencing the root page of the website, which you will recall from the Command Line chapter is how is how you reference the root of an operating system. So when you go to " http://www.google.com/ ", you are requesting Googe's root page, whereas if you go to " http://www.google.com/news" you are requesting "/news" and Google will respond with different resources.

However, before any of this can happen, your web browser needs to translate " http://www.google.com " into an IP address. This is where something called the DNS, or domain name system comes in. The DNS is a giant table that maps all of the domains in the world to their IP addresses, maintained by different internet authorities such as the Internet Assigned Numbers Authority. An ip address is a unique number that represents each computer on the internet. To communicate with Google, your browser needs to get its IP address ,which it does by looking it up in the DNS.

At a low level, all this communication happens through sockets. Sockets are the functions that give programs access to a computer's network hardware. Sockets are created by your operating system as a data structure, allowing computers to establish connections with each

other. A server opens a passive socket, and a client opens an active socket. A passive socket stays open indefinitely, listening for connections, whereas an active socket requests data from a passive socket and then closes.

To recap, the client-server model works as follows— a user enters a domain name into the browser, and the browser looks up the domain's IP address in the DNS. The browser sends an http request to the IP address it looked up, and the server responds with an http request letting the browser know it received its request and then sends the resources the web browser needs to display the requested webpage to you.

TCP/IP

The communication in the client-server model follows the TCP/IP protocol. A protocol is an agreed upon way of doing things, used to standardize a process. Protocols are not limited to Computer Science. If you were to meet the Queen of England, there would be a protocol in place—a set of rules every person has to follow when meeting her. You wouldn't just walk up to her and say "Hey bro!" You would address her a certain way, speak politely, stick to certain subjects etc. That is a protocol.

Computers communicating over the Internet use a protocol called TCP/IP. Imagine an internet without an agreed upon protocol. With no standard for communicating, every time two computers needed to pass data to one another, they would have to negotiate the terms of their communication. Nothing would ever get done. Luckily we have protocols like TCP/IP that ensure communication happens seamlessly.

TCP/IP is what is called a protocol stack. It is is made up of four layers, with each layer using its own protocol. Each layer is a program responsible for accomplishing a task, and communicating with the layers above and below it. While the Internet could use one protocol (instead of a stack), the benefit of using a protocol stack separated into layers is that you can make changes to one layer without needing to change the others. Think about the post office. Someone at the post office accepts packages, then someone else sorts them and passes them off to someone who delivers them. Each person has their own protocol for accomplishing their task (and they all communicate with each other). If the delivery guy decides to deliver packages using drones instead of a truck, the change in protocol doesn't affect the person who accepts packages or the person who sorts them. This is the same reason why TCP/IP uses a protocol stack, so changes to one protocol won't affect the others.

The four layers of TCP/IP are the Application Layer, the Transport Layer, the Internet Layer and the Network Layer.

[tcp/ip picture]

Let's take a look at an example of data moving through TCP/IP by once again thinking about mail. Think of the Application Layer as a program containing a letter. When you type a url

into your web browser, the Application Layer writes a message on the letter that looks something like this:

[picture of letter with http on it]

The information on the letter is an HTTP request. Again, HTTP is a protocol that servers and clients use to send messages to each other. The HTTP requestIt contains information such as the requested resource, the browser the client is using and a few more pieces of information.

The letter is then passed to the next layer, the Transport Layer. You can think of the Transport Layer putting the letter in an envelope. Outside the envelope, the Transport Layer puts more information:

[picture of envelope with writing on it]

The information includes the domain name to send the request to, the domain the request is coming from, the port number the server is on, and something called a checksum. Data is not sent across the network all at once, it is broken up into packets which are sent one at a time. The Transport Layer keeps uses checksum to make sure all the packets get delivered properly.

Now the Transport Layer passes the envelop to the Internet Layer which takes the envelope and puts it in an even bigger envelop, with more information written on it:

[picture of envelop with writing on it]

The information written on the Internet Layer envelop only contains the information the router needs to deliver the data to the server it is sending the data to. It contains the IP address of the server and the IP address of the computer making the request. It also contains the TTL, which stands for time to live and (explain ttl) At this point, the envelop is considered a packet .

This final envelope is sent to the bottom layer, the Network Layer, which uses hardware and software to physically send the data. The data is received by the Network Layer on the servers computers and the envelop is passed in reverse order up the protocol stack with each layer removing an envelop until the letter is revealed at the Application Level of the server. The server then goes through the same process through the TCP/IP stack, sending an HTTP request back signalling that the request was either valid or invalid. If the request was valid, it starts sending the resources the client needs.

It is important to remember that data does not get sent all at once, it gets broken down into packets. The bottom layer of the stack, the Network Layer may send thousands of packets to a client,

Challenge

Chapter #TK. Bringing It All Together

Create a Server

In this section we are going to use the Python socket library to create a simple web server and client, using Python's built-in library for creating and managing sockets. We are going to create a server that listens for requests and responds to them with the data,e and a client we can use to make those requests.

A web server creates a socket, binds it to a port, and then runs an infinite loop, responding to requests as they come through the socket; whereas a client simply opens up a socket and , connects to a server to get the information it needs. We will start by building a server in Python. The first step is to import the socket and date libraries:

```
import socket
import datetime
```

First we get today's date:

```
today = str(datetime.datetime.today())
```

Now we can create a socket:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

socket.AF_INET is an address family specifying what addresses in your socket can communicate with. AF_INET is used for communicating over the internet. There are other address families like AF_BLUETOOTH that can be passed in for communicating over Bluetooth. "socket.SOCK_STREAM" means we want to use TCP to ensure delivery. #fact check

Next we bind our socket to TCP port8888 :

```
s.bind("", 8888)
```

And set the length of the queue (a queue is used because multiple requests can come in at the same time and a data structure is needed to process them):

```
s.listen(10)
```

Now we can create the server's infinite loop which waits for a connection and sends the date back as a response:

```
while   True:
    connect, address = s.accept()
    resp = (connect.recv(1024)).strip()   # limit request to 1024 bytes
   connect.send(“received http request”)
   #Close the connection when we are finished:
  connect.close()
```

Here is our full server:

```
import   socket
import datetime

today = datetime.datetime.today()
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("", 8888))
s.listen(5)

while True:
      connect, address = s.accept()
      resp = (connect.recv(1024)).strip()
      connect.send(today)
      connect.close()
```

You can test this server by running the program and going to localhost:8888 in your browser. You should see the date when you do.

Create a Client

Now lets create a client to make requests to our server. Just like creating a web server, we start out by creating a socket:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

To connect to our server, we get the name of our local machine and set the variable port to the port our server uses so we can use it later:

```
# get name of our local machine
host = socket.gethostname()
  # set port
port = 8888
```

Now we can connect to our hostname at port 8888 by passing in a tuple with our hostname and port:

```
s.connect((hostname, port))
```

Save the response and close the socket:

```
msg = s.recv()
s.close()
```

Print the message we received:

```
print("{}".format(msg))
```

That's all there is to it. We've built a functioning client. When you run our client, you will get the date from our server.

Challenge

Chapter #TK .Practice

Exercises

Read

0.
0. http://stackoverflow.com/questions/2794016/what-should-every-programmer-know-about-security

Part V Programming for Production

Chapter #TK . Testing

“If debugging is the process of removing bugs, then programming must be the process of putting them in.”
-Edsger Dijkstra

When you start building software other people are going to use, the code in the product people end up using is called production code. When you put software into production, it means you put it live where people are using it. Part V of this book is about how you should program when your goal is to put something into production; in this section of the book we learn about the software development process, with a focus on testing. We also learn some best programming practices.

The Waterfall Software Development Process

A software development process is a way of “splitting of software development work into distinct phases (or stages) containing activities with the intent of better planning and management,” [63]The Waterfall model is a “ sequential (non-iterative) design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of conception, initiation, analysis, design, construction, testing, production/implementation and maintenance.” [64] It is made up of five phases: Planning and Requirements Analysis, Defining Requirements, System Design, Implementation and Deployment, System Testing and System Maintenance.

In the first phase of the Systems Development Cycle you determine what problem you want to solve. You look at how the new system affects your current priorities, you analyze the resources you would need to build the new system, and you think about the system’s requirements. You can do a feasibility study at this stage, looking at whether the system is operationally, economically, and technically feasible, among other considerations.

In the Defining Requirements phase you define and document the system requirements. This is done by working with the project stakeholders— “ an individual, group, or organization, who may affect, be affected by, or perceive itself to be affected by a decision, activity, or outcome of a project”. [61]

In the System Design phase, different design approaches are discussed, and a final approach is agreed upon and outlined in a document called a Design Document Specification which outlines the “ design approach for the product architecture.”[62] A design approach “clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of

the details in DDS."[62] In other words, you decide how you are going to architect the system you are building, and commit it to the DDS.

This is where the magic happens. After three stages of planning, coding begins in the Implementation and Deployment phase. The product is built in this phase, following the Design Document Specification.

The System Testing Phase is where the product is tested for bugs and tested to make sure it meets the requirements outlined in the Defining Requirements stage. The next chapter of this book covers testing in detail.

Once the product has been tested, it is put into production and released to the public. During this phase the product is live and the software team performs any necessary maintenance.

Other Software Development Processes

The waterfall method is one of many software development processes— like the incremental model: "a method of software development where the product is designed, implemented and tested incrementally (a little more is added each time) until the product is finished."[65] There are also various implementations of the popular Agile methodology which you can find out more about by reading the Manifesto for Agile Software Development: http://agilemanifesto.org.

While not a software development process, in the next chapter, we utilize Test Driven Development, a **development** technique where you must first write a **test** that fails before you write new functional code."[66] Test Driven Development helps you write better programs by forcing you to think clearly about what you are designing.

Testing

In this chapter we will focus on testing, part of the software development process. Testing a program means checking that the program "meets the requirements that guided its design and development, responds correctly to all kinds of inputs, performs its functions within an acceptable time, is sufficiently usable, can be installed and run in its intended environments, and achieves the general result its stakeholders desire."[57] In order to achieve this, we write programs to test our programs. In most cases, testing is not optional; you should consider every program you intend to put into production incomplete until you have written tests for it, unless you have a very good reason not to. If you write a quick program to do something like manipulate a file, and you never are going to use it again, testing might be a waste of time. But if you are writing a program that other people are going to be using, you

should write tests. As someone smart once said, “Untested code is broken code.” In this chapter, we are going to go over some of the fundamentals of testing. Fun fact—the word computer “bug” originated from an incident in 1942, where Grace Murray Hopper found and removed a moth stuck in a Mark Aiken Relay Calculator.

Assertions

Assertions are the foundation of tests. An assertion is a statement that a programmer expects to be True (in which case it doesn’t do anything) and raises an exception if it is False Python has a built-in assert keyword for creating assertions. Here is an example:

```
x = 1
assert x == 1
>>

x = 1
assert x == 2
>>Traceback (most recent call last):
  File "/Users/coryalthoff/PycharmProjects/self_taught/st.py", line
  2, in <module>
   assert x == 1
  AssertionError
```

In the first example, the condition following assert is True , so no error is raised. In the second example, the condition following assert is False , and an AssertionError is raised. Assertions are used to check whether or not a test (which we will learn how to write shortly) passed. If an AssertionError is raised, the test failed, otherwise it passed.

Types of Tests

Testing is usually done in four different phases: unit testing, integration testing, system testing, and acceptance testing. In this section, we will briefly explore each of these phases.

The unit testing phase involves writing unit tests that test individual pieces of code such as functions, methods and classes. Each unit test tests one aspect of a piece of code with an assertion. Say for example you have a function that prints whatever string you pass it. One unit test might pass the function the string “Hello” and test that “Hello” was printed; whereas another unit test might pass the function an integer to make sure the function is able to properly handle an input that is not a string. Unit tests should test the general-use case for all of your functions classes and methods; checking what happens when they receive input values you weren’t expecting, and test boundary conditions—when things like a list get big or full.

You can write your unit tests within a unit testing framework— a program for creating and structuring unit tests, like Python's built-in unit testing framework called unittest, which comes with different assertion methods that let you easily test different conditions.

Integration testing is performed after unit testing. While unit testing tests each individual piece of code in a module, integration testing tests to make sure the different modules in a project work with each other. For example, say you are building a banking application and you have two modules: transfer—for transferring money—and balance—for showing the customer's balance. An integration test might check that when the transfer module moves 1,000 dollars from a customer's account, and the balance module correctly reduces the customer's balance by 1000 dollars. [59]

System testing tests the entire system. There are many different types of tests used in systems testing: graphical user interface testing—which tests the part of product you can see; usability testing—which tests that people can figure out how to use the product; and software performance testing—which tests to see how the product performs under a heavy workload such as a large number of users; among others.

Finally the last phase of the testing cycle is acceptance testing—which checks to make sure the software meets the requirements agreed upon by the project stakeholders. Acceptance testing is not done programmatically—it is done by people who make sure the the product requirements laid out in the requirements document are met. The rest of this chapter will focus on unit testing, because as a new software engineer on a team, you will be responsible for writing unit tests—but you most likely will not be responsible for integration, systems and acceptance testing.

TDD

As we learned in the previous chapter, TDD stands for test driven development, and is a software development technique that helps you design better programs. When you follow test driven development, you write your unit tests before you write your program. Following TDD forces you to break out of the pattern of putting writing unit tests off until the end of your development cycle, and then deciding not to write them. It also guarantees you will have unit tests throughout your development cycle. Lastly, TDD helps you to design better software by forcing you to think about the design requirements of your program by writing your tests first.

In this section we are going to learn to write unit tests by creating astack using TDD and Python's unittest testing framework. We will start our development process by writing unit tests that will fail (but would work if our stack was properly designed), and then writing code to make the tests pass.

To better visualize the tests we need to create, wWe will start by defining a stack that doesn't do anything to better visualize the tests we need to create. In this section I assume you

remember how a stack works (If you forget how a stack works, please revisit the chapter Data Structures & Algorithms).

```
import unittest

class Stack :
    def __init__ ( self):
        self .stack = []

    def push ( self, item):
        pass

    def pop( self):
        pass

    def peak( self ):
        pass

    def is_empty ( self ):
        pass
>>>
```

We've defined a Stack class, however, none of the methods in our class actually do anything. Before we define the methods for our Stack class, we are going to write all of our tests .

```
class StackTests(unittest.TestCa se):

    def setUp( self ):
        self.stack = Stack()

    def tearDown( self):
        del self.stack

    def test_is_empty( self ):
        self.assertTrue( self.stack.is_empty())

    def test_push( self):
        self.stack.push( 100 )
        self.assertFalse( self.stack.is_empty())

    def test_peak( self):
        self.stack.push( 'test' )
        self.assertEqual( self .stack.peak(), 'test' )

    def test_pop( self):
        self.stack.push( 10.1 )
```

```
        self.stack.pop()
        self.assertTrue(self.stack.is_empty())

    def test_pop_value(self):
        self.stack.push('test_value')
        value = self.stack.pop()
        self.assertEqual(value, 'test_value')
>>>
```

When you write unit tests with the unittest framework, you start by defining a class that inherits from unittest.TestCase — in this our class is called StackTests . Theunittest framework uses the class that inherits from unittest.TestCase to run tests; with each method you define in your class—as long as it starts with test—runs as an isolated test. A test that passes will not raise an AssertionError , and a test that fails will. If you run our program, you will see the results of each test (they all will fail withAssertionErrors).

Here is a more detailed explanation of the testing code you wrote. The first two methods setUp and tearDowndo not start with test because they are not tests: they are methods inherited from unittest.TestCase used to help set up our tests.

```
def setUp(self):
    self.stack = Stack()

def tearDown(self):
    del self.stack
```

setUp runs before each test, and tearDown runs after each test. In this case we use setUp to create a new Stackobject before each test, and use tearDown to delete it after each test. We do this in order to make sure we have a brand new Stack object in each test, so the tests don't interfere with each other.

In the first test we define, test_is_empty() , using the assertion method assertTrue() which takes a parameter and raises an AssertionError if the parameter evaluates to False because if we don't put anything in our stack, it should be empty.

```
def test_is_empty(self):
    self.assertTrue(self.stack.is_empty())
```

Our next test is test_push() . It callspush() on our stack and passes in100 . We use the assertion method assertFalse and pass in self.stack.is_empty() because after pushing something to our stack, the stack should no longer be empty: self.stack.is_empty() should be False .

```
def test_push(self):
    self.stack.push(100)
    self.assertFalse(self.stack.is_empty())
```

In our next test, test_peak(), we push 'test' onto the Stack and use assertEqual() to check that self.stack.peek() returns the value we pushed to our stack, in this case self.stack.peek() and 'test' should be equal.

```
def test_peak(self ):
    self .stack.push( 'test' )
    self .assertEqual( self.stack.peak() , 'test' )
```

Our next test is test_pop(). We push 10.1 to our stack, call self.stack.pop() and use the assertion methodassertTrue() to check that ourpop() method successfully removed an item and our stack is now empty.

```
def test_pop(self ):
    self .stack.push( 10.1 )
    self .stack.pop()
    self .assertTrue( self.stack.is_empty())
```

Our final test, test_pop_value(), pushes the string 'test_value' to our stack and uses the assertion methodassertEqual() to check that pop() returns'test_value' .

```
def test_pop_value(self):
    self .stack.push( 'test_value')
   value = self .stack.pop()
    self .assertEqual(value, 'test_value' )
```

Run our tests. Y ou will be notified a ll five tests failed by raising five AssertionErrors . This is because we have not defined any of the methods in our Stack class. Our methods don't do anything and so all of our tests fail. One of the advantages of TDD is that writing your tests first helps clarify your thinking. We now know exactly how we need to define each method in our Stack class in order to pass each test. Here is what we need to do:

```
class Stack:
  def __init__( self ):
    self .stack = []

  def push(self, item):
    self .stack.append(item)

  def pop(self ):
    return self.stack.pop()

  def peek( self):
    return self.stack[- 1]

  def is_empty(self ):
    if len ( self .stack) > 0:
```

```
            return False
        return True
>>>
```

Now that we've defined the methods in ourStack , when you run our tests again, the tests will all pass. We've tested our class, our methods and general use cases; and now we need to test unexpected use cases, bad input values, and boundary conditions. An example of an unexpected use case is calling pop on an empty stack, which will cause an error. To fix this, first we should write a new test.

```
class StackTests (unittest.TestCase):

    def setUp( self):
        self.stack = Stack()

    def tearDown ( self ):
        del self.stack

    def test_empty_pop( self ):
        with self.assertRaises( IndexError ):
            self.stack.pop()
```

Our new test test_empty_pop uses the assertion method self.assertRaises() with the with statement to test if an exception is raised. The code inside the with statement is expected to raise the exception passed to assertRaises. If the exception is raised the test passes; if the exception is not raised an AssertionError is raised and the test fails. Now we can fix our stack to handle empty pops:

```
class Stack :
    def __init__ ( self ):
        self.stack = []

    def push( self, item):
        self.stack.append(item)

    def pop ( self):
        if not self.stack:
            raise IndexError ( "Cannot pop from empty stack")
        return self.stack.pop()

    def peek( self ):
        return self.stack[- 1]

    def is_empty ( self ):
        if len ( self.stack) >0 :
            return False
        return True
```

>>>

We need to think about bad input values. In this example, there are no bad input values because any object in Python can be added to a list, and therefore can be added to our stack. However, you always want to take the time to at least think about any input values that could break your program. Finally, we should think about boundary cases: the "behavior of a system when one of its inputs is at or just beyond its maximum or minimum limits." [58] In Python, there is no limit to how many objects we can put in a list, therefore the size of our stack is only limited by the amount of memory the computer that created it has. However, some programming languages limit the number of objects you can put in a list, and if that were the case in Python, we would have to write a test for that condition.

Writing Good Tests

Good tests are repeatable. That means when you run your tests , they should work in any environment—if you write a test on OS X, it should also work on Windows without having to make any changes to to the test. An example of violating this would be including hard coded directory paths in your test. Windows and OS X use different slashes for directory paths—so your test in a Windows environment would not work in an OS X environment. This means the test is not repeatable, and needs to be rewritten. Tests should also run quickly. Tests need to run often; try not to write tests that take a long time to run. Finally your tests should be orthogonal—one test should not affect the other.

Code Coverage

Code coverage is the the total number of lines of code in your project called during your tests divided by the total number of lines of code in your project. Code coverage does not measure the efficiency of your tests, but is useful for finding untested parts of your code. You generally want to have code coverage above 80%. If your code coverage is low, it means you have not tested enough of your code. The professional version of PyCharm (the one that costs money) is integrated with a tool for analyzing code coverage. If you use PyCharm's free Community Edition, you are in luck because the tool the professional version is integrated with is called coverage.py and is free to use. The documentation for coverage.py is available at:https://coverage.readthedocs.io .

Testing Saves Time

It’s easy to get lazy and skip testing, justifying your laziness by saying you don’t have time to write tests. Counterintuitively, taking the time to write tests will save you a substantial amount of time in the long run. The reason is because if you don’t write tests, you will end up testing your software manually— running your program yourself with various different inputs and under different conditions to see if anything breaks as opposed to testing your program programmatically. While you should test your entire program by testing it manually and looking for bugs, you don’t want to solely rely on this. Spending your time manually testing inputs and conditions when you could easily automate the process is a huge waste of time. Finally, if you come back to the project in a month, you won’t be able to remember the different tests you were manually running.

Testing is an important part of programming. Most software teams have at least one person dedicated to testing. Getting into the habit of following TDD will improve your code by making sure you always write tests and therefore decrease the number of errors in your code, and by helping you to think carefully about how you design your programs.

Challenge

Write unit tests for the Hangman program we built in Part I.

Whenever you are following programming instructions,and see $, it means whatever follows is a command that you should type into the command line (no need to type the dollar sign).

Programming languages have conventions as well. Conventions are rules generally followed by the community using the language.

Chapter #TK. Best Programming Practices

“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.”
-Martin Golding

In this chapter we will cover a few general programming principles that will help you write production-ready code. Many of these principles originated in the excellent book “The Pragmatic Programmer” by Andy Hunt and Dave Thomas, a book that dramatically improved the quality of my code.

Write Code As A Last Resort

Your job as a software engineer is to write as little code as possible. When you have a problem you need to solve, your first thought should not be “How can I solve this?” It should be, “Has someone else solved this problem already and can I use their solution?” If you are trying to solve a common problem, chances are someone else has already solved it. Start by looking online for a solution. Only after you’ve determined no one else has already solved the problem should you start solving it yourself.

DRY

DRY is a programming principle that stands for Don’t Repeat Yourself. Following DRY is easy: if you are writing code and find yourself repeating the same code—stop. Do not repeat yourself. If you find that you are copying pieces of code, pasting them somewhere else in your program, and making small changes to it to create new code— stop. Do not repeat yourself. We will illustrate this with a program that makes changes to a list of words.

```
def capitalize_item(word, word_list):
    for index, item in enumerate(word_list):
        if item == word:
            word_list[index] = word_list[index].capitalize()

def change_letter(word, word_list, old_letter, new_letter):
    for index, item in enumerate(word_list):
        if item == word:
            word_list[index] = word_list[index].replace(old_letter, new_letter)
```

```
words = ['Programming', 'is' , 'fun' ]
upper('Programming', words)
print(words)
change_letter('fun' , words, 'u' ,'$' )
print(words)
```

```
>> ['PROGRAMMING', 'is', 'fun']
>> ['PROGRAMMING', 'is', 'f$n']
```

Both functions use Python's built-in enumerate function, which takes a list as a parameter, and allows you to use a for loop to easily capture the current index of the list, as well as the current item in the list. "Index" refers to an item's position within a list. Our program works—but if you look closely at the code—you will see both of our functions look for a word in a list to replace it with something new. Instead of having code to search for a word in both functions, we should create one function that returns the index of the word we are looking for (Python has a built-in function index() that finds the index of a string in a list, but for the sake of this example we are not using it). Here is how we should refactor our code to avoid repeating ourselves:

```
def find_index(word , word_list):
    for index , item in enumerate (word_list):
        if item == word:
            return index


def upper(word , word_list):
    index = find_index(word , word_list)
    word_list[index] = word_list[index].upper()


def change_letter(word ,word_list , old_letter, new_letter):
    index = find_index(word , word_list)
    word_list[index] = word_list[index].replace(old_letter , new_letter)

words = ['Programming', 'is', 'fun']
upper('Programming', words)
print(words)
change_letter('fun', words, 'u', '$')
print(words)

>>>
['PROGRAMMING', 'is', 'fun']
['PROGRAMMING', 'is', 'f$n']
>>>
```

By creating a new function that returns the index of a word, we are no longer r epeating code. If we decide to change the way we search for an index, we only need to change our find_index function, instead of changing the code to find an index in multiple functions.

Orthogonality

Orthogonality is another important principle popularized by the book the*Pragmatic Programmer* . The authors Andy Hunt and Dave Thomas explain, “In computing, the term has come to signify a kind of independence or decoupling. Two or more things are orthogonal if changes in one do not affect any of the others. In a well-designed system, the database code will be orthogonal to the user interface: you can change the interface without affecting the database, and swap databases without changing the interface.” [16] Put this in practice by remembering that as much as possible, “a should not affect b”. If you have two modules— module a and module b— module a should not make changes to things in module b, and vice versa. If you design a system where a affects b; which affects c; which affects d; things quickly spiral out of control and the system becomes unmanageable.

Every Piece Of Data Should Have One Representation

This is best explained with an example. Say you are building a project, and in that project you are using Twitter’s API (application programming interface)— a program that gives you access to data from a website like Twitter. Twitter provides a module you can download on pip to query their API for data. In order to use Twitter’s API, you have to register for an API key— a string you send to Twitter when you use their API so they can verify it is you.

Once you’ve obtained your API key from Twitter, you start using the API in two functions. The first function gets data from celebrities, and the second function gets data from non celebrities. Both functions need the API key in order to use Twitter’s API. Here is an example of what this could look like:

```
# WARNING this code does not actually work

import twitter_api
import celebrity_list
import regular_list


def celebrity_data ():
  api_key ='11330000aazzz22'
  return twitter_api.get_data(api_key , celebrity_list)
```

```
def people_data ():
  api_key = '11330000aazzz22'
   return twitter_api.get_data(api_key , regular_list)
```

This is a made up example that won't actually work, but the gist is we are using a made up module called twitter_api to call the function get_data with our api key and a list of people. A few months go by and you end up getting a new API key from Twitter. You go to the celebrity function and change the variable api_key to the new API key. It's been a long time since you wrote this code, and you completely forget the API key is used in the second function. You put your code into production and accidently leaving the old API key in the second function; everything breaks, and you get fired. You could've avoided your tragic fate by following the rule that every piece of data should have one representation. The correct way to handle this situation is to make a Python file with a variable called api_key. This is called a configuration file. Your program should import api_key from this file, and both functions should use it. This way, the piece of data (the API key) is only represented once. That way, no matter how many places the API key is used, if you have to change the API key, you only need to change it in one place, the configuration file, and the previously discussed disaster will be av erted.

Functions Should Do One Thing

Every function you write should do one thing, and one thing only. If you find your functions getting too long, ask yourself if the function you are writing is accomplishing more than one task. Limiting functions to accomplishing one task offers several advantages: your code will be easier to read because the name of your function will describe exactly what it does; and if your code isn't working it will be easier to debug because every function is responsible for a specific task, so you can quickly isolate and diagnose the function that isn't working. As Ryan Singer says, "So much complexity in software comes from trying to make one thing do two things."

Use Dummy Data

While I was at eBay I was given an assignment to fix an error in our code. The program I was debugging processed a large text file and took five minutes to run. I would make a change to the program to try to get some information about what was wrong, run the program, and wait five minutes for the results. I was not making any progress, because I had to wait five minutes every time I made a change, and that quickly added up. I finally took the time to substitute the large text file with dummy data—fake data my program could use but would only take a few seconds to process. This way I could still look for the bug in the

program, but much faster. Taking the time to set up dummy data—even if it takes you twenty minutes—w ill quickly pay off by shortening your debug cycle.

If It's Taking Too Long You Are Probably Making a Mistake

If you are not working on something obviously complex like working with a large amounts of data, and your program is taking a very long time to load, assume you are doing something wron g.

Logging

Logging is the practice of recording data when your software runs. You can use logging to help debug your program, and to gain additional insight into what what happened when your program ran. Python comes with a great logging module that lets you log either to the console or a file.

When something goes wrong in your program, you don't want it to go unnoticed—you should log information about what happened so you can review it later. Logging is also useful for collecting and analyzing data. For example, you might setup a web server set to log data—including the date and time—every time it receives a request. You could store all of your logs in a database, and create another program to analyze that data and create a graph displaying the times of day your website is visited the most.

Good programmers use logging, summarized nicely by Henrike Warne when he said "One of the differences between a great programmer and a bad programmer is that a great programmer adds logging and tools that make it easy to debug the program when things fail." You can learn how to use Python's logging module at https://docs.python.org/3/howto/logging.html.

Do Things The Best Way The First Time

If you are in a situation where you are programming and you think, "I know there is a better way of doing this, but I'm in the middle of coding and don't want to stop and figure out how to do it better." Stop. Do it better.

Follow Conventions

Taking the time to learn the conventions of the new programming language you are trying to learn will help you read code written in the new language faster. Pep 8 is a set of guidelines for writing Python code, and you should read it. It's available at: https://www.python.org/dev/peps/pep-0008.

Use a Powerful IDE

An IDE, or Interactive Development Environment, is a program you use to write your code. Thus far, we've been using IDLE, the IDE that comes with Python. However, IDLE is just one option of many different IDEs available, and I do not recommend using it long term, because it is not very powerful compared to other IDEs. For example, if you open up a Python project in a better IDE, there will be a different tabs for each Python file. In IDLE, you have to open a new window for each file, which in big projects this quickly gets tedious and it's difficult to navigate back and forth between files.

I use an IDE called PyCharm created by JetBrains. They offer a free version as well as a professional version. Either one will work. Sublime is another popular IDE. In this chapter I will be going over some of the features I use in JetBrains IDE that increases my productivity. Because any IDE is liable to change its commands at any time, there are no examples in this chapter. Instead I describe some of the features PyCharm has, to give you an idea of what an IDE is capable of, so you won't waste time doing things manually that you can quickly do with an IDE. I put a tutorial at theselftaughtprogrammer.io/ide so I can keep it up to date.

If you see a variable, function or object being used and you would like to see its definition, PyCharm has a shortcut to jump to the the code that defined it (even if it is in a different file). There is also a shortcut to jump back to the page you started from.

PyCharm has a feature that saves local history which has dramatically improved my productivity. PyCharm will automatically save a new version of your project every time it changes. This means you can use PyCharm as local version control system but without having to push to a repository. You don't have to do anything, it happens automatically. Before I knew about this feature, I would often solve a problem, change the solution, and then decide I wanted to go back to the original solution. If I didn't push the original solution to GitHub, the original solution was long gone, and I would have to rewrite it again. With this feature, you can simply jump back in time 10 minutes, and reload your project exactly how it was. If you change your mind again, you can jump back and forth between the different solutions as many times as you want.

In your workflow you are probably copying and pasting code a lot, moving it from one location on a page to another. In PyCharm, instead of copying and pasting, you can move code up and down on the page you are on.

PyCharm is integrated with popular version control systems like Git and SVN. Instead of having to go to the command line, you can use Git from Pycharm. The fewer trips you have to make back and forth between your IDE and the command line, the more productive you will be.

PyCharm also has a built in command line and Python Shell.

In Pycharms Pro version, there is a built in tool for connecting to relational databases, such as MySQL. If you are using a relational database, this is a huge time saver. Remembering SQL syntax is difficult, having autocomplete for your SQL is a game changer.

PyCharm has a powerful debugger that lets you pause the execution of a program at certain points, check the status of different variables, and walk through a program step by step.

Code Reviews

A code review is when someone reads your code and gives you feedback. You should do as many code reviews as you can—especially as a self-taught programmer getting started. Even if you follow all of the best practices laid out in this chapter, you are still doing things wrong—you need someone more experienced than you to read over your code and tell you the mistakes you are making so you can fix them.

Code Review on Stack Exchange is a website where you can get a code review from a community of programmers. Anyone can go on Code Review and post their code. Other members of the Stack Exchange community review your code and give you feedback about what you did well and offer helpful suggestions on how you can improve.

Security

Security is an easy subject for the self-taught programmer to ignore. You probably won't be asked about security in your interview, and security is not important for the programs you write while you are learning to program. However, once you get your first job programming, you are directly responsible for the security of the code you write. The last thing you want is for your company's database to get hacked because you made a mistake. Better to learn some practical tips to keep your code safe now.

In the chapter The Command Line we learned to use sudo to issue a command as the root user. I mentioned you should never run a program as sudo. You should also disable root logins if you are managing the system. There are several reasons you should do this: every hacker is aware there is a root account and so it is an easy target when attacking a system (hence it should be disabled), if you are logged in as root you can irreversibly damage the system you are running, and if you run a program as root and it gets taken over by a hacker, the compromised program now has root access.

There are several kinds of malicious attacks that rely on exploiting programs that accept user input, so you should also assume all user input is malicious and program accordingly. An SQL injection is a type of attack that occurs when a user submits input to a

program and adds SQL to it, allowing them to execute SQL in your database.For instance, you might have a car website, and on that website you want to return details about a specific car a user enters. You expect the user to provide you with the name of a car, such as “Nissan Leaf”. Once you receive the name of the car from the user, you write more code to query a MySQL table called cars to display details about the car. You use the following query:

```
SELECT* FROM car WHERE name = query_data;
```

The variable query_data represents the data you collected from the user. If the user enters something like “Nissan Leaf” like you expected, this works fine. The problem is the user can enter anything they want. A malicious user could enter something like this into the website:

```
“Ferrari”; DROP TABLE cars;
```

This would cause your program to query MYSQL with the following, valid SQL:

```
SELECT* FROM car WHERE name = “Ferrari”; DROP TABLE cars
```

The query would get all of the data for a car named Ferrari, then it would delete your database table—wiping out all of your data. To combat this, you assume all user input is malicious, and instead of writing raw SQL—you use an ORM like SQLAlchemy or another library that protects against SQL injections. You don’t want to write a solution to this problem yourself because you could too easily make a mistake, which someone could exploit.

A final strategy for keeping your software secure is to minimize your attack surface —the different areas of your program where attackers could extract data or attack your system in some way. By making your attack area as small as possible, you can reduce the likelihood of vulnerabilities in your program. Here are some strategies for minimizing your attack surface: avoid storing confidential data if you don’t have to, give users the lowest level of access you can, use as few third party libraries as possible (the less code the less amount of possible exploits), and get rid of features that are no longer being used (less code less exploits).

Avoiding logging in as the root user on your system, not trusting user input and minimizing your attack surface are important steps to making sure your programs are secure. However, these are just starting points. You should always take time to try to think like a hacker. How would a hacker try to exploit your code? This can help you find vulnerabilities you otherwise would’ve overlooked. There is a lot more to learn about security than we can cover in this book, so make an effort to always be thinking and learning about security. Bruce Schneier said it best—“Security is a state of mind”.

Challenge

Review a program you've written and see if you followed the programming practices outlined in this chapter.

Chapter #TK. Bringing It All Together

“I don't care if it works on your machine! We are not shipping your machine!”
—Vidiu Platon

Congratulations on making it to the final coding exercise of this book. In this chapter we are going to create a program that scrapes a lyrics page on lyrics.wikia.com and creates a word cloud with the lyrics. If you’ve never seen a word cloud, it is an image derived from a piece of text in which the size of each word corresponds to how many times it appeared in the text.
[picture of a word cloud]
We will develop our program using theWaterfall model and test driven development.

Word Cloud Lyrics

We begin our software development process in the Planning and Requirements Analysis phase. First, we define the problem we want to solve: our inability to scrape a song lyrics website and turn the lyrics into a word cloud. We never want to write code unless we have to, so the first thing we do is check if someone has already solved this problem. Some Google searching reveals this problem has been solved, and the code is available on GitHub. Normally we would use the available solution if it meets our requirements, and the software development process would end, but for the sake of learning we can forget we saw the solution, and conclude no one has solved this problem.

Next we enter the Defining Requirements phase, where we go over our project’s requirements, and the resources we will need to complete it. Our project requires two main pieces of functionality: the ability to scrape data from a website, and the ability to create a word cloud. We’ve already scraped data from Google News, so we know we have the tools and expertise to scrape song lyrics from a website. What we don’t know is how to build a word cloud. Building a nice looking word cloud is not easy, so if someone has not already solved this problem for us, it is going to significantly increase the time needed to finish this project. Luckily, a Google search reveals there is a Python module calledword_cloud that lets you easily create beautiful word clouds, so we can conclude we should be able to complete this project in a reasonable amount of time. We also have to define the requirements for our project—we need to be able to pass our program a url that has song lyrics from lyrics.wikia.com, and our program needs to generate a word cloud. We finish this phase by writing our requirements down in a requirements.txt document.

We are now in the third phase of the Waterfall Model—System Design— where we decide how our system will be architected. Our program will consist of two functions—one to get the lyrics from the website, and the other to create the word cloud, which will also be the main function we use to run our program.

After designing our system , we are ready to enter the Implementation and Deployment phase and start to code. Since we are following Test Driven Development, we will start by writing our tests. We know our program needs to do two things: get lyrics from lyrics.wikia.com, and create a wordcloud; so we will start by writing a test for each of these pieces of functionality.

```
import unittest
import os


def get_lyrics():
    pass


def create_wordcloud(wiki_url):
    pass


class TestBIAT(unittest.TestCase):
    def test_lyrics(self):
        """Test that a string gets return from get_lyrics()"""
        self.assertIsInstance(get_lyrics(), str)

    def test_wordcloud_creation(self):
        """Test that a new file is created when create_wordcloud() is
           called"""
        filecount_before = len(os.listdir())
        create_wordcloud('http://lyrics.wikia.com/wiki/The_Beatles:Lucy_In_The_S
                         ky_With_Diamonds')
        self.assertEqual(filecount_before + 1, len(os.listdir()))
```

Our first test checks that `get_lyrics()` function returns a string because we want to scrape the lyrics from the site and return them as a string for our other function to use. Our second test checks that `test_wordcloud_creation()` creates a new file.When we create a wordcloud we are going to save it as a file in the folder where we ran our program, which is why we test to make sure `create_wordcloud()` creates a new file in the current directory when it's called with the url of a lyrics.wikia.com lyrics page. We check that a new file is created using the `os` module—a built-in Python library that has different functionality for interacting with your operating system. `os.list.dir()` returns a list of all the files in the current directory. In our assertion in `test_wordcloud_creation()`, we check the number of files in the current directory before we call `create_wordcloud()` + 1 is the same as the number of files after `create_wordcloud()` was called. This is because after we call`create_wordcloud()`, there should be the same number of files as there were, only one more (the new file that was created). When we run these tests, they both should fail with an`AssertionError`. Now all we have to do is write the code to make them pass. First, we need to install the libraries we are going to use with pip. The first two are requirements for the`wordcloud` library.

```
$ pip3 install numpy
$ pip3 install pillow
$ pip3 install wordcloud
```

You should already have Beautiful Soup installed from when we built our first scraper, but if not:

```
$ pip3 install BeautifulSoup4
```

We start our program by importing the libraries we installed:

```
from wordcloud import WordCloud
from bs4 import BeautifulSoup
import requests
from os import path
```

Now we can write a function to scrape song lyrics:

```
def get_lyrics(wiki_url, tag, tag_name):
    """ Takes a url for a lyrics page of lyrics.wikia.com, a tag and a tag name searches for that tag
    in the urls HTML that has the tag_name passed in. Returns the song lyrics found.
    :param    wiki_url: string lyrics.wikia lyrics url e.g. http://lyrics.wikia.com/wiki/The_Beatles:Girl.
    :param    tag: string HTML tag to look for lyrics in e.g. "div"
    :param    tag_name: string name of HTML tag to look for lyrics in e.g. lyricbox
    :return : string song lyrics
    """
    response = requests.get(wiki_url)
    soup = BeautifulSoup(response.text, 'html.parser')
    if soup.find_all(tag, tag_name):
        return soup[0].text
```

Our get_lyrics() function should look familiar. It is similar to our Google News scraper: we get the HTML from the lyrics website and pass it into a BeautifulSoup object; then we pass the parameter tag and tag_name into soup.find_all() which looks for an HTML tag with the tag and tag_name we passed in. We return the result— the song lyrics (if they were found). We pass tag and tag_name into our function instead of passing it into soup.find_all() directly because we are getting data from a live website, and the tag we are looking for is likely to change, so we don't want to hardcode that data into our function, isntead we pass it in so the caller of the function can decide the correct tag and tag_name to pass in. At the time of this writing, the song lyrics are held in the HTML tag . If the HTML tag ever changes, I will update it on GitHub, so watch out for that if this program stops working. Here is how we would call our current function:

```
get_lyrics('http://lyrics.wikia.com/wiki/The_Beatles:Lucy_In_The_Sky_With_Diamonds', 'div',
        'lyricbox')
```

Now we can code a function to create our word cloud:

```
def create_wordcloud(wiki_url , file_name ,tag , tag_name):
    """ Takes a url for a lyrics page of lyrics.wikia.com and creates a
    wordcloud from the lyrics.
    :param wiki_url: string lyrics.wikia lyrics url e.g.
    http://lyrics.wikia.com/wiki/The_Beatles:Girl.
    """
    lyrics = get_lyrics(wiki_url , tag, tag_name)
    wordcloud = WordCloud().generate(lyrics)
    image = wordcloud.to_image()
    image.show()
    image.save(path.dirname(__file__) + '/wordcloud.jpg')


create_wordcloud(' http://lyrics.wikia.com/wiki/The_Beatles:Lucy_In_The_Sky_With_Diamonds_', 'wordcloud.jpg',
'div', 'lyricbox' )
```

We start by getting the lyrics from the wiki_url we passed in by calling our get_lyrics() function and passing it the wiki_url . From hereonout, the T heWordCloud module does all the hard work for us. We create a WordCloud object and pass it the lyrics. Now we can use the methods the WordCloud object has to create our wordcloud. The trick to knowing what methods exist and figuring out how to use them is to read the Wordcloud module's documentation, available at: https://github.com/amueller/word_cloud.

First we call to_image() on ourWordCloud object, and then call show() on the result to create our word cloud. This creates a word cloud that will pop up on our desktop, but we also want to save the word cloud as a file, so we callsave() —which takes a path to where we should save the file as a parameter, and saves the file there; so if we want to name our file "my_file.txt" and save it to our desktop, we would pass in something like "/users/desktop/my_file.txt". In this case, we want to save our file in the folder where our program is running, so we pass in path.dirname(__file__) (path is a function from theos module which returns the path where our program is running), and concatenate that with the file_name passed to tocreate_wordcloud() as a parameter— resulting in something like "users/program/word_cloud.jpg" (which is where our word cloud will be saved). That's all there is too it, our function now creates a beautiful wordcloud based on the song lyrics from the url we pass in.

When you run the program, it should create a word cloud that pops up on your desktop (if not check the version posted on GitHub because the tags may have changed), save the file to the folder where the program is running, and when you run our tests, they should all pass. Here is our finished code:

```
from wordcloud import WordCloud
from bs4import BeautifulSoup
import requests
from osimport path


def get_lyrics (wiki_url ,tag , tag_name):
```

```
    """ Takes a url for a lyrics page of lyrics.wikia.com, a tag and a
    tag name searches for that tag
    in the urls HTML that has the tag_name passed in. Returns the song
    lyrics found.
    :param wiki_url: string lyrics.wikia lyrics url e.g.
    http://lyrics.wikia.com/wiki/The_Beatles:Girl.
    :param tag: string HTML tag to look for lyrics in e.g. "div"
    :param tag_name: string name of HTML tag to look for lyrics in e.g.
    lyricbox
    :return : string song lyrics
    """
    response = requests.get(wiki_url)
    soup = BeautifulSoup(response.text, 'html.parser')
    if soup.find_all(tag, tag_name):
        return soup[0].text


def create_wordcloud(wiki_url, file_name, tag, tag_name):
    """ Takes a url for a lyrics page of lyrics.wikia.com and creates a
    wordcloud from the lyrics.
    :param wiki_url: string lyrics.wikia lyrics url e.g.
    http://lyrics.wikia.com/wiki/The_Beatles:Girl.
    """
    lyrics = get_lyrics(wiki_url, tag, tag_name)
    wordcloud = WordCloud().generate(lyrics)
    image = wordcloud.to_image()
    image.show()
    image.save(path.dirname(__file__) + '/wordcloud.jpg')


create_wordcloud('http://lyrics.wikia.com/wiki/The_Beatles:Lucy_In_The_Sky_With_Diamonds', 'word_cloud.jpg',
'div',
                 'lyricbox')
```

Now that we've written our program, the next phase is System Testing. We've already written unit tests using Test Driven Development, but we still need to write integration tests and system tests, and have people test the program as users. In this case, we are creating a product for other programmers to use, so we should get some programmers to try out our program, and see if they are able to easily use it. We won't write anymore tests in this example, but try to think about how you would approach this.

The final step in our development process is Implementation and Deployment. Since we are building a product for other programmers, putting our program into production means uploading it to GitHub and PyPi. Once we do that, it is available for other programmers to use, and it is up to us to continuously check in on the code and provide maintenance for the code if anything breaks. If you are interested in learning how to upload your code to PyPi so

it is available on pip, follow the instructions here: http://peterdowns.com/posts/first-time-with-pypi.html.

Practice

Exercises

0. Find a project on GitHub, read through it and think about the quality of the code.

Read

0. http://googletesting.blogspot.com/2016/06/the-inquiry-method-for-test-planning.html

Part VI Land a Job

Chapter X. Your First Programming Job

"Beware of 'the real world.' A speaker's appeal to it is always an invitation not to challenge his tacit assumptions."
— Edsger W. Dijkstra

Welcome to Part VI, the final part of this book, dedicated to helping you with your career as a software engineer. Getting your first programming job requires extra effort, but if you follow the advice in this chapter, you should have no problem. Luckily, once you land your first programming job and get some experience, when it comes time to look for your next job, recruiters will start reaching out to*you* .

Choose a Path

One thing to keep in mind is that programming jobs are lumped into specific domains: each with their own set of technologies and skill sets. If you look at programming job ads, the headline will be something like "Python Backend Programmer Wanted." This means they are looking for someone that programs the backend of a website, and is already familiar with Python. If you go to the job description, there will be a list of technologies the ideal candidate will be familiar with, along with skills they should have. While it's fine to be a generalist (a programmer that dabbles in everything) while you are learning to program, and it is possible to get a job as a generalist programmer, you probably should find a specific area of programming you enjoy and start becoming an expert in it. This will make getting a job significantly easier.

Web and mobile development are two of the most popular programming paths. Application development is often split into two parts—the front end and the back end. The front end of an application is the part that you can see—like the interface of a web app. The back end is what you can't see—the part that provides the front end with data. Some companies will have a team dedicated to the front end, and a team dedicated to the back end. Other companies only hire full stack developers—programmers that can work on both the front and back end. However, this only applies to application development (building websites or apps). There are all kinds of other programming areas you can work in like security, platform engineering, and data science. To learn more about the different areas of programming, go to sites listing programming jobs, and read through the descriptions. The Python Job Board is a good place to start: https://www.python.org/jobs. Look at the requirements for the different jobs, as well as the technologies they use—this will give you an idea what you need to learn to be competitive for the type of job you want.

Getting Initial Experience

In order to get your first programming job you need experience. But how do you get programming experience if no one will hire you without it? There are a few ways to solve this problem. One solution is to focus on open source. You can either start your own open source project, or contribute to the thousands of open source projects on GitHub.

Another option is to do freelance work; you can create a profile on sites like Upwork, and start getting small programming jobs right away. To get started, I recommend finding someone you know that needs some programming work done. Have them sign up for an Upwork account and officially hire you there so they can leave you a great review for your work. Until you have at least one good review on a site like Upwork, it is difficult to land jobs. Once people see that you've successfully completed at least one job, getting hired for jobs by people you've never met becomes easier, because you've established some c redibility .

Getting an Interview

Once you've gained programming experience through either open source or freelance work it's time to start interviewing. I've found the most effective way to get an interview is to focus on LinkedIn. If you don't have a LinkedIn account, create one to get started networking with potential employers. Add a summary about yourself at the top of of your profile, and make sure to highlight your programming skills. For example, a lot of people add something like "Languages: Python, JavaScript" at the top of their profile, which helps recruiters searching for those keywords find them. Make sure to add your open source or freelancing experience as your most recent job.

Once your profile is complete, start connecting with technical recruiters—there are tons of technical recruiters on LinkedIn, so find them and send them a request to connect. They are always looking for new talent, so they will be eager to connect with you. Once they accept your invitation, reach out and ask if they have any open positions available.

The Interview

If a recruiter thinks you look like a good fit for the role they are hiring for, they will send you a message on LinkedIn asking to set up a phone screen. The phone screen will be with the recruiter, so it is usually non-technical (although I've had recruiters ask me some technical questions they've memorized the answer to during the first interview). The

conversation is about the technologies you know, your previous experience and seeing if you would fit into the company's culture.

If you do well, you will advance to the second round—a technical phone screen—where you speak with one or more members of the engineering team and they ask you the same questions about your experience and skills as the first interview. This time however, the questions are followed by a technical test over the phone. The engineers will give you the address of a website where you can edit code; they will post a programming question; and ask you to solve it.

If you make it past the second round—and this nerve-racking process hasn't caused you to abandon this book and quit programmin g—you will have a third interview. The third interview is on site, in person at the company's office. The third round is a lot like the first two. You meet with different engineers on the team, they ask you about your skills and experience, and there are more technical tests. Sometimes you even stay for lunch to see how you interact with the team. The third round is where the famous white board coding tests happen. If the company you are interviewing for does whiteboarding, you will be asked several programming problems and asked to solve them on a whiteboard. I recommend buying a white board and practicing this before hand because solving a programming problem on a whiteboard is much harder than solving it on a computer. Of course not all companies follow this exact formula, you may run into different variations of it, but in general, this is what you should expect.

Hacking The Interview

The majority of programming interviews focus on two subjects—data structures and algorithms. That means to pass your programming interview, you know exactly what you must do— get very good at two specific area s of Computer Science. Fortunately, this will also help you to become a better programmer.

You can narrow down the questions you should focus on even further by thinking about the interview from the interviewer's' perspective. Think about the situation your interviewers are is in; they say software is never finished, and it's true. Your interviewers most likely has a lot of work they need to get done, and don't want to dedicate a lot of his time to interviewing candidates. Coming up with good programming questions is hard. Are they going to spend their valuable time coming up with original programming questions? Probably not. They are probably going to google "programming interview questions" and ask one of the first ones they find. This leads to the same interview questions coming up over and over again—and there are some great resources out there to practice them. Leetcode is one I recommend you check out. Almost every question I've ever been asked in a programming interview is on Leetcode.

Chapter X. Working on a Team

"You can't have great software without a great team, and most software teams behave like dysfunctional families."
- Jim McCarthy

Coming from a self-taught background, you are probably used to programming alone. Once you join a company, you will need to learn how to work on a team. Even if you start your own company, eventually you will need to hire additional programmers, at which point you will also need to learn to work as a team. Programming is a team sport and like any team sport, you need to get along with your teammates. This chapter provides some tips for successfully working in a team environment.

Master the Basics

When you are hired by a company, you are expected to be competent in the skills covered in this book. It is not enough to simply read this book—you need to master the skills in it. Start a side project with a friend and use version control. Write new tests everyday. Schedule programs to run. If your teammates have to constantly help you with the basics, they will become frustrated.

Don't Ask What You Can Google

As a new, self-taught member of a programming team, you will have plenty to learn, and you will need to ask a lot of questions. This is a great way to learn, but you want to make sure you are only asking good questions. A general rule is to only ask a question if you've spent at least five minutes trying to Google the answer yourself. While asking questions is a positive thing, if you ask too many questions you could have easily figured out yourself, you will annoy your teammates. Make sure you are only asking good questions by trying to find the answer for at least five minutes before you ask it.

Changing Code

By reading this book, you've demonstrated you are the type of person who is constantly looking to improve. Unfortunately, not everyone on your team will share your enthusiasm for

becoming a better programmer. Many programmers don't have the desire to keep learning, and they are fine doing things suboptimally. This can be especially prevalent in startups, where shipping code fast is often more important than shipping high quality code. If you find yourself in this situation, you should listen to Walter White and "tread lightly." Programmers can get their egos hurt very easily when you change their code. Even worse, if you spend a lot of time fixing other people's code, you will not have enough time to contribute to new projects, and it may look like you are not working hard enough. The best defense for this is to carefully question any company you join about their engineering culture. If you still find yourself in this situation, it is best to listen to Edward Yourdon, "If you think your management doesn't know what it's doing or that your organisation turns out low-quality software crap that embarrasses you, then leave. "

Even if you work at a great company, there are still times when you will need to fix other people's code, which you should approach delicately. Mike Coutermarsh wrote an article on Medium called "Jr. Developers #5: How to Improve Code Without Anyone Hating You" which I recommend you read for advice on the subject. As a new programmer entering your first programming job, it's important for you to get along with your team—alienating your team members can happen much easier than it may seem—so make sure to always stay cognizant of how your teammates will react if you want to make changes to their code.

Imposter Syndrome

Everyone that programs feels overwhelmed at times and, no matter how hard you work there are going to be things you don't know. As a self-taught programmer, it is especially easy to feel inadequate because someone asked you to do something you've never heard of, or because you feel like there are so many concepts in Computer Science you still do not understand. Remember—this happens to everyone—not just you.

I was surprised when my friend with a Masters degree in Computer Science from Stanford told me he feels this way as well. He said everyone in his program dealt with imposter syndrome, and he noticed they reacted in one of two ways: they either stayed humble and were always willing to admit when they didn't know something—and tried learn it—or they pretended they knew everything (when they didn't) and stifled their learning. Make sure you remember you got to where you are by working hard, and it's ok if you don't know everything, nobody does. Just stay humble, relentlessly study anything you don't understand, and you will be unstoppable.

Further Learning

“The best programmers are not marginally better than merely good ones. They are an order-of-magnitude better, measured by whatever standard: conceptual creativity, speed, ingenuity of design, or problem-solving ability.”
—Randall E. Stross

The article *ABC: Always Be Coding* by David Byttow gives great advice on how to get a job as a software engineer. The article is summarized in the title—always be coding. If you combine ABC with a new acronym I made up—ABL—always be learning—you are sure to have an exceptional career. In this chapter, I am going to go over some programming resources I’ve found helpful.

The Classics

There are a few programming books that are considered must reads: *The Pragmatic Programmer* by Andy Hunt and Dave Thomas; *Design Patterns* by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm (design patterns are an important subject we didn’t get a chance to cover); *Code Complete* by Steve McConnell; *Compilers: Principles, Techniques, and Tools*, by Alfred Aho, Jeffrey Ullman, Monica S. Lam and Ravi Sethi; and *Introduction to Algorithms* by The MIT Press. I also highly reccomend *Problem Solving with Data Structures and Algorithms*, a free, interactive, excellent introduction to algorithms, much easier to understand than *MIT’s Introduction to Algorithms*.

Online Classes

Online coding classes are a popular way to learn to program. Coursera is one of the most popular, but I also love the lesser known Codeschool. I highly recommend you take some of their classes. Not only do they have great classes on different programming languages, but they also have classes on Git, SQL and other subjects discussed in this book.

Hacker News

Hacker News is a platform for user-submitted news on the technology incubator Y Combinators website found at https://news.ycombinator.com. It is popular with programmers and will help you keep up to date with the newest trends and technologies.

Other

This article is great for finding new things to learn: http://matt.might.net/articles/what-cs-majors-should-know.

Next Steps

First of all—thank you for purchasing this book. I hope it's helped you become a better programmer. The programming community has given me so much support, and by writing this book, I hope I've managed to help you in your journey the way so many people helped me.

Now that you've finished, it's time for you to get down to business. Where do you go from here? Data structures and algorithms, algorithms algorithms. Get on LeetCode and practice those algorithms. Then practice them some more! In this chapter I give some final thoughts on how you can continue to improve as a programmer (once you finished practicing your algorithms).

Find a Mentor

Finding a mentor will help you take your programming skills to the next level. One of the hard things about learning to program is that there are so many things you can do suboptimally, without knowing it. I mentioned earlier you can help combat this by doing code reviews: a mentor can do code reviews with you and help you improve your coding process, recommend books, and help you with programming concepts you are having trouble understanding.

Strive to Go Deep

There is a concept in programming called a "black box" that refers to something you use, but do not understand how it works. When you first start programming, everything is a black box. One of the best ways to get better at programming is to try to open up every black box you can find and try to understand how it works. One of my friends told me it was a major "aha" moment for him when he realized the command line itself is a program. Opening up a black box is what I call going deep.

Writing this book helped me go deep. There were certain concepts I thought I understood, only to find out I couldn't explain them. I had to go deep. Don't just stop at one explanation, read all the explanations on the topic you can find. Ask questions and read differing opinions online.

I find one of the most helpful questions to be "What problem does this solve?" For example, we learned about Object-oriented programming in this book. But why was object-

Oriented programming invented? What problem does it solve? Are there other solutions? Pursuing the answers to these types of questions will help you become a better programmer.

Another way to go deep is to build things you want to understand better. Having trouble understanding databases? Build a simple database in your free time. When I was having trouble understanding compilers—I built one. Taking the time to do a project like this is well worth the investment, because it will improve your understanding of whatever you are struggling with.

Other Advice

I once came across a forum discussing ways to become a better programmer. The top voted answer was a somewhat surprising: "Do things other than programming." I've found this to be true—reading books like *The Talent Code* by Daniel Coyle has made me a better programmer; because he lays out exactly what you need to do to master any skill. Another good book about learning a skill is *Mastery* by George Leonard. Keep your eye out for things outside of programming you can bring to your programming game.

The last piece of advice I will leave you with is to spend as much time as you can reading other people's code. Reading other people's code is one of the best ways to improve as a programmer. When you are learning, you need to make sure to strike a balance between writing code, and reading code. Reading other people's code is going to be difficult at first, but it is important to do it because you can learn so much from other programmers.

I hope you enjoyed this book as much as I enjoyed writing it. Please feel free to email me at cory@theselftaughtprogrammer.io for any reason. I also have a programming newsletter you can sign up for at theselftaughprogrammer.io and a forum where you can get in touch with me and a community of people learning to program. If you liked this book, please also consider leaving a review on Amazon , it helps get this book in the hands of more people, and I really appreciate every review I receive. Best of luck on the rest of your journey. And remember—ABL!

Acknowledgements

Parents, Pam, Randee, Anzar, Cover Designer, Lauren, Antoine, Torrey, Jin Chun.

Glossary

Biography

Citations

0. http://stackoverflow.com/questions/466790/assembly-code-vs-machine-code-vs-object-code
0.
0. http://man7.org/linux/man-pages/man1/ls.1.html
0.
0. http://www.webopedia.com/TERM/C/clock_speed.html
0.
0. http://stackoverflow.com/questions/1050222/concurrency-vs-parallelism-what-is-the-difference
0.
0. http://berb.github.io/diploma-thesis/original/022_webapps.html
0.
0. http://stackoverflow.com/questions/11828270/how-to-exit-the-vim-editor
0.
0. http://superuser.com/questions/666942/why-it-is-not-recommend-to-use-root-login-in-linux
0.
0. http://programmers.stackexchange.com/questions/37294/logging-why-and-what
0.
0. http://stackoverflow.com/questions/10925478/how-to-read-api-documentation-for-newbs
0.
0. http://www.infoworld.com/article/2908474/application-development/stack-overflow-survey-finds-nearly-half-have-no-degree-in-computer-science.html
0.
0. http://www.merriam-webster.com/dictionary/catenate
0.
0. Design Patterns kindle location 546
0.
0. Design Patterns kindle location 650
0.
0. Design Patterns kindle location 824
0.
0. https://hungred.com/wp-content/uploads/2009/05/ascii-table-cheat-sheet1.png
0.
0. The Pragmatic Programmer kindle location 830
0.
0. https://automatetheboringstuff.com/chapter0/
0.

0. http://kevinlondon.com/2015/07/26/dangerous-python-functions.html
0.
0.
http://interactivepython.org/runestone/static/pythonds/Recursion/TheThreeLawsofRecur
0.
0. https://automatetheboringstuff.com/chapter1/
0. https://www.quora.com/What-are-some-interesting-facts-about-computer-programming
0. http://www.dkfindout.com/us/explore/eight-cool-facts-about-computer-coding/
0. http://thenextweb.com/insider/2016/02/26/8-facts-every-computer-programmer-should-know/#gref
0. http://cs.lmu.edu/~ray/notes/x86assembly/
0. http://www.tutorialspoint.com/python/python_files_io.htm
0. http://www.wsj.com/articles/computer-programming-is-a-trade-lets-act-like-it-1407109947?mod=e2fb
0. https://en.wikipedia.org/wiki/Persistence_(computer_science)
0. https://en.wikipedia.org/wiki/Column_family
0. http://stackoverflow.com/questions/2570756/what-are-database-constraints
0. https://en.wikipedia.org/wiki/Data_integrity
0. https://www.sitepoint.com/understanding-the-observer-pattern/
0. http://codedx.com/how-to-minimize-your-softwares-attack-surface/
0. https://support.microsoft.com/en-us/kb/283878
0. http://www.slideshare.net/jagaarj/database-design-normalization
0. http://www.tcpipguide.com/free/t_WhatIsNetworking.htm
0. Readwrite.com
0. http://www.gnu.org/software/grep/manual/grep.html
0. https://en.wikipedia.org/wiki/Software_testing
0. https://en.wikipedia.org/wiki/Boundary_case
0. http://www.guru99.com/integration-testing.html
0. https://en.wikipedia.org/wiki/Systems_development_life_cycle#cite_note-1
0. Project Management Institute, 2013
0. http://www.tutorialspoint.com/sdlc/sdlc_overview.htm
0. https://en.wikipedia.org/wiki/Software_development_process
0. https://en.wikipedia.org/wiki/Waterfall_model
0. https://en.wikipedia.org/wiki/Incremental_build_model
0. http://agiledata.org/essays/tdd.html
0. http://www.agilenutshell.com/
0. https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming
0. http://stackoverflow.com/questions/1031273/what-is-polymorphism-what-is-it-for-and-how-is-it-used
0. https://en.wikipedia.org/wiki/Syntax