From The Amazon #1
Best-Selling Author

INTRO TO PYTHON PROGRAMMING BEGINNERS GUIDE SERIES

JOHN ELDER

ALSO BY JOHN ELDER

Intro To Python Programming Beginners Guide Series

John Elder

Codemy.com
Las Vegas

Intro To Python Programming Beginners Guide Series

By: John Elder

Published By Codemy.com
Las Vegas, NV USA

ISBN
979-8-9859654-0-7

First Edition

FOR APRIL

TABLE OF CONTENTS

INTRO TO PYTHON PROGRAMMING

BEGINNERS GUIDE SERIES

ABOUT THE AUTHOR

John Elder is a Web Developer, Entrepreneur, and Author living in Las Vegas, NV. He created one of the earliest online advertising networks in the late nineties and sold it to publicly traded WebQuest International Inc. at the height of the first dot-com boom.

He went on to develop one of the Internet's first Search Engine Optimization tools, the Submission-Spider that was used by over three million individuals, small businesses, and governments in over forty-two countries.

These days John writes about Coding and wealth creation; produces a popular coding Youtube Channel, and runs **Codemy.com** the online learning platform that teaches Coding to hundreds of thousands of students.

John graduated with honors from Washington University in St. Louis with a degree in Economics. He can be reached at john@codemy.com

INTRODUCTION

"Which programming language should I learn?"

Over at Codemy.com I've taught hundreds of thousands of people how to code and if I've heard that question once, I've heard it a thousand times.

It's confusing – there's no doubt about it!

Most people would agree that Python is one of the most popular programming languages out there.

But is it a good first language to learn?

Absolutely!

For one, it's incredibly easy to learn. Python is more "conversational" in nature than many other programming languages. Meaning, it feels more like writing words than it does technical jargon.

Second, it's incredibly powerful! There's not much you can't do with Python.

Want to build websites? Python works for that.

Want to build GUI apps? Yep, Python works great for that too.

Want to do Data Science? Oh yeah, Python shines there too.

There are very few languages that tick all those boxes at once yet remain so easy to learn and fun to work with.

Maybe that's why Python is "eating the world" when it comes to computer programming.

Either way, I think Python is certainly the best programming language to learn if you're just starting out.

What if you aren't just starting out?

Python is great for you too! If you have experience coding in another language, then Python should be very easy for you to pick up.

In fact, the ease of Python may even frustrate you a little bit! You'll be angry at yourself for all the hassle you went through learning those other languages!

I'm not kidding!

You might find yourself trying to slap semi-colons to the end of your Python lines of code, and other unnecessary things like that.

Yeah...Python is drop dead simple and it makes other programming languages seem old and stuffy by comparison.

Wherever you're coming from; complete newbie all the way to seasoned coder, Python is almost certainly right for you.

This book is focused mainly on newer programmers. Though seasoned coders should still find it useful.

If you're new, I suggest starting at the beginning and taking it one page at a time. If you're experienced, feel free to jump around and grab what you need whenever you need it.

If you get confused or stuck along the way you can always head over to Codemy.com and use the contact form to shoot me a message. You bought this book, so I'm happy to help explain anything in here that you don't understand.

FOLLOW ALONG – VIDEO COURSE!

Everyone learns differently. Some people learn by reading, some by watching and listening, others learn by doing. I want you to be able to learn this stuff in the best way *for you.*

So I've created a companion video course that goes along with this book. Basically, it covers everything that the book covers, it's just in video format.

Normally I charge $49 for the course on its own, but I want you to have it for just $9 today as my 'thank you' for giving this book a try.

You can sign up and access the course at Codemy.com/pythonfollow

It's only $9, you don't even need a coupon code. I suggest you go sign up right now, because even if you don't watch the videos, by signing up you'll get a membership account at Codemy.com and you can post questions directly to me under each video.

If you don't understand something in the book, maybe the video will clear things up for you. If not, you can post a question right there and I'll answer personally.

It's not a bad deal for just $9 bucks ;-)

FREE PDF VERSION OF THE BOOK

Programming books are tricky. I like reading my programming books in the paperback form. But you can't copy and paste code from a paperback book.

Then there's Kindle...which is super convenient and sometimes does allow you to copy and paste (sometimes – programming books are sometimes different). But the problem with Kindle is that the formatting sometimes messes things up and makes it hard to read...especially code examples etc.

PDF is the best of both worlds. You can read it on your Kindle or iPad or Desktop, you can read it online, you can print it out if you want. You can copy and paste code examples, etc. etc. etc.

Everyone who signs up for the $49 Python course at **Codemy.com/pythonfollow** will also get a free copy of this book in pdf form.

Like I said, I want to give you all the tools you need to learn this stuff, so giving you a free pdf copy makes sense to me.

Grab it now at:

http://Codemy.com/pythonfollow

CONVENTIONS USED IN THIS BOOK

This is a book on Python programming, so there will be lots of code examples and snippets.

Usually when I'm just talking about something, things will look like this; regular text in paragraph form.

Whenever there's code to be discussed, I'll use a `fixed width font like this` with a bold header like this:

CODE EXAMPLE 2.1

```

if weather == "sunny":
  print("Go for a run")
elif weather == "rainy":
  print("Go to the gym!")
else
  print("Watch TV.")


```

It should be pretty obvious :-p

I'll also use images from time to time. Hopefully the images will be clear and readable, but if not then get the pdf copy over at Codemy.com/python or watch the videos.

EXERCISES

At the end of each chapter or major section of the book, I'll try to give you some practice exercises covering whatever we just discussed. By no means do you need to do all the exercises, but I do recommend that you give them a try.

Even if you think you totally understand the thing I just discussed, there's something about trying it yourself. Don't just go back to what you just read and copy and paste my example code and then modify it to fit the answer.

Instead, I suggest you do the exercises from scratch. Write out all the code by hand, don't copy and paste anything. And maybe wait a little while after you read before trying the exercises.

Python is pretty easy. The exercises will likely seem pretty easy as well, especially after you've just read about them. Instead, wait a few hours, or a day or two and then try the exercises. Give yourself time to forget what you just learned so that you reinforce things in your mind by learning it again.

I think you'll get more out of this book that way, and isn't that the whole point? Sure, you can speed through this book in a couple of hours, but it won't stick with you the same way.

Ok, enough with this jibber jabber...lets dive in and learn some freaking Python!

CHAPTER ONE

SETTING UP A PYTHON DEVELOPMENT ENVIRONMENT

First things first! Before we do any Python coding, we need to set up a development environment so that you can actually run Python code.

It's a small but important step!

Windows / Mac / Linux

Running Python is a little different depending on which operating system you're running. As with most things code related, Mac is going to be a little harder out of the box than Windows.

But in this case, it's not a big deal.

Historically, all Mac's come with Python already installed. But the problem is, the *version* of Python that comes on a Mac is horribly out of date. You'll need to upgrade to the latest version to continue with this book. Yes, really. The old version of Python 2.7 is deprecated. It's no longer supported or recommended for use. Newer Mac's may have an updated Python version and we'll need to determine that now.

MAC

It's quick and easy to determine which version of Python is installed on your Mac computer. All we have to do is open up a Terminal (that comes installed on your computer already) and type:

CODE EXAMPLE 1.1

```

python -V

```

Punch in that command and hit enter and the terminal should respond with a version number. If you have version 2.7 then you'll need to upgrade to the latest version at python.org.

You may be able to continue without installing the latest version of Python by running all your Python code with this command: **python3**

Apart from Python, you'll also need some sort of text editor so that you can actually write Python code. Any text editor will do. I like Sublime Text and a lot of coders use it. It's freeware at **SublimeText.com**

LINUX

Like Macs, Python is likely already installed on Linux already. You can open a terminal and check for sure:

CODE EXAMPLE 1.2

```

python -V

```

If the terminal spits out a version number higher than 2.7, then you're good to go. If your particular flavor of Linux doesn't have Python installed already then you'll need to download and install it.

I'm not going to explain how to do that – let's face it, you're running Linux so you're awesome already and likely don't need my help installing some trifling software!

But in case you can't figure it out, check out the Python website: **https://www.python.org**

You'll also need a text editor, I like Sublime Text but use whatever you like. You're running Linux, so I know you'll do exactly that!

It seems like everyone and their mother has a personal preference when it comes to text editors and the reality is, it doesn't really matter what you use.

WINDOWS

Windows is never particularly friendly to the coder. It always does things it's own way – which usually annoys me because I like to do things MY own way and this can lead to friction.

But in this case, running Python on Windows isn't particularly troublesome. In fact, it's downright easy.

Head over to the official Python website at:
https://python.org

There you can download the free self-contained Python installer for windows. It's one file, download it and double click it to install Python. Be sure to click the "Add To Path" option during installation.

Like I've mentioned for Mac and Linux, you'll also need some sort of text editor to write code in. Any text editor will work (even Notepad), but I suggest Sublime Text – which is free and many many many coders use it.

But as I mentioned earlier, you're free to use any coding text editor that you like. I recommend against using IDE's like Pycharm or Visual Studio if you're a beginner. Really, Sublime Text is all you need and it's drop dead easy to use.

GIT BASH TERMINAL

In order to run your Python code, you'll need a terminal. Most computers come with some sort of terminal.

If you're on a Mac, you can just use the terminal that comes with your computer. Go to the little search icon and type in 'terminal' and it should pop right up.

The same goes for Linux.

Windows users should download and install a better terminal. It's true, Windows comes with a couple of terminals (the cmd terminal and the powershell terminal) and you can certainly use them…but I don't recommend it!

Your life will be easier if you download and install a better, more professional terminal…and that's what I recommend that you do.

The terminal I recommend is the Git Bash terminal. It's similar to the terminal that comes with Mac and Linux.

Newer Windows computers may come with a version of the Git Bash terminal, but I recommend you download the actual Git Bash terminal at:

https://codemy.com/gitbash

You can download the latest version at that web site.

Installing the Git Bash terminal can be confusing. There are about a zillion installation screens. Don't pay them any attention. Just click the "Next" button for all of them.

One of the installation screens will ask you what Text Editor you'll be using (we're using Sublime Text).

You DON'T have to make a selection!

The Git Bash terminal doesn't really need to know which text editor you're using.

So just continue to click "Next" and speed through all the installation screens. You don't even need to read them.

CODING TEXT EDITOR

You'll also need some sort of text editor to write code on. As I've mentioned, I recommend Sublime Text which you can download for free as freeware at SublimeText.com

They'll ask you to pay if you continue to use it, but it will keep working if you don't.

Any coding text editor will work, but you can't go wrong with Sublime Text.

SUMMING UP

Now that you have Python installed on your computer, a text editor to write your code on, and a terminal to run your code…you're ready to go!

There's nothing else you need in order to start writing and running some awesome Python code!

Let's get to it!

CHAPTER TWO

WRITING OUR FIRST PYTHON PROGRAM

Alright! Let's write some Python code! I recommend that you create a directory on your computer on your C:\ drive where you can save all the code we'll be writing throughout this book.

You can open up a file explorer and create a directory or do it when you save your first file in Sublime.

Let's name it C:\code

Name it anything you like but keep it simple and one word because that will make it easier to navigate to in the terminal later when we want to run the code.

Open Sublime Text

It's time to create an actual Python file! Python files generally end in .py so let's create one right now called hello.py

To do this, fire up your Sublime Text and click "New File" in the top right corner of the screen under the "File" menu.

A new file will appear, rename it hello.py by clicking "Save As" in the "File" menu in the top right corner of the screen.

If you've already created your C:\code directory, navigate there and save your file. Otherwise navigate to your C:\ directory and right click, and choose "New>Directory" to create the directory. Then save the file.

There's nothing there now, but let's type in our very first line of Python code, which will be the first line you write in ANY programming language, the infamous "Hello World!" line...

CODE EXAMPLE 2.1

```

print("Hello World!")

```

Type that in and then save the file. You can save it two ways; either by clicking the CTRL+S button on your keyboard (Command S on mac – I believe), or by clicking the "File" link at the very top of the screen, and then choosing "Save" from the menu that pops up.

With me so far? Before we talk about the code, let's run this thing and see what happens!

Open up your Git Bash Terminal. Just click your Windows Start menu and type git and the Git Bash Terminal icon should pop up.

Once open, type the command: **cd /c/code**

(Space after cd). That will navigate you to the code directory where you saved your Python file. To run the file just type: python hello.py

CODE EXAMPLE 2.2

```
$.
$.  python hello.py
$.
```

This should output the line Hello World! right into the terminal.

The command "python" that we typed tells the terminal to execute the file hello.py

Whenever you want to run a Python program, just type in *python file_name.py* and the file will get executed and output right there on the terminal screen. (python3 file_name.py on a mac)

THE CODE

You've created your first Python program! Congratulations! Sure it was pretty easy, but it was still a pretty impressive thing. Most people don't know how to do even that much in Python – and now you do.

So what's going on in our code? Well, not much. We have just one line and the only thing that looks unfamiliar is the word "print", which tells the Python interpreter to "print" something on the screen; in our case we're "printing" the phrase "Hello World!" onto the screen. That phrase is a Python string (more on strings later).

Notice anything after the last parenthesis in ("Hello World!")? Nope, there's nothing else there. In some programming languages you have to end each line with a semi-colon ; or maybe a colon : or something else.

Not so in Python, which makes code much easier to read.

TERMINAL NOTE

I should mention that you can clear the screen in the terminal by typing in: clear and then hitting enter. You're going to want to do that from time to time because as we continue to run code and output the contents to the terminal screen, things can get a little jumbled up and cluttered feeling.

I usually clear the screen before running any code.

ERRORS

So far, our code has been incredibly simple, so you probably didn't run into any errors. That won't stay the same for very long. If there's one thing that's as certain as death and taxes, it's that you're going to screw something up in your code and get an error.

In fact, let's create an error right now just to see what happens. Let's leave off the parenthesis…

CODE EXAMPLE 2.3
```

print "Hello World!"

```

Save the file and run it in the terminal.

What happened? Did the code execute? Nope, you got an error code that looked something like this:

SyntaxError: Missing parentheses in call to 'print'. Did you mean print("Hello World!")?

Sometimes the errors will be helpful like that, sometimes you won't be able to understand what they mean at all.

But this is the common situation you'll see whenever you screw something up. The code generally won't execute in the terminal and instead you'll get some sort of error message.

Get used to it - it's going to happen a lot.

I mean a lot.

So many times.

Welcome to being a coder! Right now, it was incredibly easy for us to figure out what went wrong. The error itself suggested a fix, and if it didn’t, a quick glance at the code itself should have done the trick.

But what happens when our code gets more complicated and we have hundreds, or even thousands of lines of code? How do we figure out how to fix it?

Google.

I'm convinced that something like 78% of a good coder's job is simply Googling error messages to debug mistakes.

Just copy and paste the error code into Google.

Most of the time you'll quickly find the solution.

If you take the time to actually Google this error, you'll see that a lot of the results are from a site called StackOverflow.com

Get used to StackOverflow.com because as a developer, you're going to spend a HUGE amount of time there searching for answers to problems with your code. It's a great resource for coders to ask questions and get help with their code.

Chances are, someone's already had the problem you're having and had their questions answered. So you can read about their solutions right there. If you run up against an error that no one has ever had before, you can post a question, along with your actual code, and people will help you out. All for free.

It's a real community and I highly recommend that you take advantage of it whenever needed.

COMMENTS

Before we move on and start talking about some real coding, I want to mention one more minor thing; comments.

Comments are ubiquitous in any and every programming language. They're everywhere and they're pretty important.

Comments are...well, just that. They allow you to describe your code so that someone else later on can look at it and understand what you've done. They're also for YOU later on when you look back over your code and go "What the heck was that bit of code supposed to do?!"

It's not always obvious what any specific bit of code is supposed to do. That becomes even more true the larger and more complicated your program becomes.

Believe me, if you've been working on a project for three months and have written several thousand lines of code for the thing, you just aren't going to remember what every line is supposed to do.

Sometimes several people might hack around on a thing and the code is changed and re-changed many times. Comments are your friend.

Get into the habit of using comments throughout your code.

You don't have to comment on every single line of code. There are best practices that you'll pick up as you gain experience.

But every time you add a major section to your code, get in the habit of writing a little comment to explain what's going on.

Python handles comments with a number sign, or pound sign, or hashtag or whatever you want to call it. # That thing.

Comments get ignored by the Python interpreter and don't execute. They're just for our human eyes. You can use them on their own line, or at the end of any line of code.

CODE EXAMPLE 2.4

```
1.
2.  # Output Hello World! to the screen
3.  print("Hello World!")
4.
```

You see how I put the comment right above the code that I wanted to explain? That's how you do it, but you could also put it at the end of line 3 as well. Like this:

CODE EXAMPLE 2.5

```
1.
2.  print("Hello World!") # Output Hello World! to the screen
3.
```

Python will ignore anything behind the # sign for the rest of the line. Comments can be more than one line too, just use triple single quotation marks ''' at the beginning and end of whatever you'd like to comment, like so:

CODE EXAMPLE 2.6

```

'''
This is a comment and it's long
So I'm going to use two lines for it
'''
print("Hello World!")

```

Don't get carried away with commenting, you should definitely comment, but keep it short, keep it succinct, keep it direct and easy to read. Don't write a paragraph talking about how you're feeling or what you ate for breakfast.

Here's one more example, take a look at this next bit of code and try to figure out what will happen:

CODE EXAMPLE 2.7

```

print("Hello World! # I learned about comments!")

```

Can you guess what will be output to the screen? If you guessed:

```
Hello World! # I learned about comments!
```

Then you guessed correctly. Why wasn't that last bit commented out? Because the # sign was typed INSIDE the quotation marks on line 2.

Yeah, I tricked you there…

CHAPTER TWO EXERCISES

1. Write some code that outputs more than one line of text to the screen.

2. Write a comment on the same line as another line of code.

3. Write a multi-line comment

4. Write some code that creates an error, then google that error.

5. Clear the screen of your Git Bash terminal window

CHAPTER THREE

FUN WITH MATH, VARIABLES, AND INPUTS

Alright, so You've got your development environment all set up and you've familiarized yourself with writing and running some simple code. You've created your first Python program and ran it from the terminal.

Sure, it was a pretty simple little program, but you've got to start somewhere (and every programmer has written that Hello World program when they started learning to code).

It's time to start talking about slightly more complicated things, namely math (and later variables).

Sure, no one really likes math – well I like math – but most normal people don't like math. But math is one of those building block things that you need to master quickly to write code in just about any programming language.

Luckily we're not talking about advanced calculus or linear algebra here; we're just talking about basic arithmetic operators like adding, subtracting, multiplying, and dividing.

In this chapter I'll give a quick overview of Python's basic math operators.

BASIC MATH OPERATORS

Like I said, we're not going to get into any heavy math right now. We're just going to focus on the basics like addition, subtraction, multiplication, and division.

Math with Python couldn't be easier. You might not think you'll use this stuff much because...well...math. But you really will, as we'll see as we go throughout the rest of the book.

Python Handles math just like you would expect, with these operators:

Addition	+
Subtraction	–
Multiplication	*
Division	/
Modulus	%
Exponents	**

Those all probably look familiar to you, except maybe the modulus operator (think remainder), but we'll talk about it in a bit.

For now, let's create a new file called math.py in your Sublime Text editor.

Let's play around with our math operators.

CODE EXAMPLE 3.1

```

print(3 + 2)

```

Go ahead and save the file, then run it and you should see the number 5 output to the terminal screen. Wow, right?!

There's a couple of things to note here. The first is that we didn't use quotation marks around the 3 + 2. In the last chapter, whenever we used "print", we slapped a couple of quotation marks around the thing we wanted to output.

Not so with math. But just for fun, try adding the quotes:

CODE EXAMPLE 3.2

```

print("3 + 2")

```

Save the file and run it. What happened? Instead of outputting 5, the program outputted the text "3 + 2". So, if you want to do math, leave the quotes off. If you want to output text, use quotes.

The other thing to not is the spacing. Notice how we used a space between the 3 and the + and the 2?

Those spaces are just for us and our *stupid human eyes*™ so that it's easier to read. You could have left them out like this:

CODE EXAMPLE 3.3

```

print(3+2)

```

...and that would get you the exact same output of 5. It's really just a personal preference whether or not to use spaces. Personally, I find it easier to read and therefore easier to keep track of what's going on in my code...so I use spaces.

Play around with the other Math operators. In fact, let's make a little program that uses them all:

CODE EXAMPLE 3.4

```

print(3 + 2)
print(5 – 1)
print(3 * 4)
print(10 / 5)
print(35 % 3)
print(2 ** 4)
```

That should output:

```
5
4
12
2
2
16
```

Pretty straight forward, except maybe the modulus operator "%".

Modulus basically returns the remainder when dividing the number on the left by the number on the right.

In our example, three goes into thirty-five 11 times (3 x 11 = 33). What's left over? 2: (35 – 33 = 2).

What is 35 % 7? The answer is zero. Why? Because 7 goes into 35 exactly 5 times (7 x 5 = 35) with zero left over.

This might seem a little weird, but you'll actually use the modulus from time to time because it's helpful to determine programmatically whether a number is fully divisible by another number for a bunch of different reasons that we'll get into later.

INTEGERS AND FLOATS

So we've talked about the basic math operators, now let's talk about numbers in general. This is kind of important.

A number can be an integer or a float (and also a 'complex', but we don't care so much about that right now).

An integer is a whole number. 5 is an integer. So is 55378008. 98 is also an integer. Whole numbers.

A float is a number with decimals in it. Like 5.27, that's a float. 98.1 is also a float. So is 1,902,488,278,211.923

This is important to understand because Python handles floats and integers very differently. Let's take an example:

CODE EXAMPLE 3.5

```

print(3.0 - 1)

```

Go ahead and run that code. What did you get? You got the answer "2.0"!

2.0?

Yep, 2.0 This might seem a little strange because we know that 3 – 1 is 2, but Python returned 2.0.

So why did Python do that? Because we mixed data types; we used a float (decimal) and an integer (whole number) when we wrote the code. So Python returned a float.

We could easily have used all integers in our code, like so:

CODE EXAMPLE 3.6
```

print(3 - 1)

```

If we save that and run it, Python will return 2.

What's the point of all this? Just that integers and floats are different and using them has different consequences.

GETTING A LITTLE CRAZY

So we've talked about some basic math stuff, and we've talked about strings (remember our first program had a string that said "Hello World!" in it).

So let's put the two together and see what we can do. Let's create a math program that outputs all the math operators we've seen so far, with strings as well:

CODE EXAMPLE 3.7

```

print(f"3 + 2 = {3 + 2}")
print(f"5 - 1 = {5 - 1}")
print(f"3 * 4 = {3 * 4}")
print(f"10 / 5 = {10 / 5}")
print(f"35 % 3 = {35 % 3}")
print(f"2 ** 4 = {2 ** 4}")

```

Go ahead and save that and then run it and you should see this output:

- 3 + 2 = 5
- 5 - 1 = 4
- 3 * 4 = 12
- 10 / 5 = 2.0
- 35 % 3 = 2
- 2 ** 4 = 16

So what's going on here? Well, we wrapped everything after our print in quotation marks. Normally when we use quotation marks like that, no math will get executed. But we used something called an **f-string** that allows us to execute code inside of a string…right there between the brackets:

{....}

Whenever you put that inside an f-string, Python escapes it out of the string and executes the code inside the curly brackets.

Just be sure to slap the f before the quotation marks:
f"string stuff here {code stuff here}"

Why did we do this? Just for fun! Plus I think it's more readable when you run the program to be able to get the questions as well as the answers. A bunch of numbers outputted to the screen doesn't help me at all. 5...what does 5 mean? But if the program outputs "3 + 2 = 5" then it makes sense to me.

So that's fun. Hey, we're getting a little more complicated here! It's exciting! Well, maybe just a *little* exciting. Let's move on to some slightly more complicated things.

...but we're still going to stick with the math theme for a bit longer because there's a few more math concepts that you need to learn, and they're actually really important. You'll use them a lot.

COMPARISON OPERATORS

Comparison operators are something you'll use forever no matter what programming language you code in. Apart from If/Else statements and loops, they're possibly the most used things in all programming.

What are they?

Well, comparison operators are exactly what they sound like...they compare things; two or more things.

Is 5 bigger than 2?

Is 10 equal to 27?

All these things and more can be determined by comparison operators.

So here's a list of the biggies:

Table Of Comparison Operators

Equal to	==
Not Equal To	!=
Greater Than	>
Greater Than or Equal To	>=
Less Than	<
Less Than or Equal To	<=

When you use a comparison operator, Python will return either "True" or "False", which might seem a little weird at first but totally makes sense. True and False are "Booleans", but more on that later.

Give it a try and see:

CODE EXAMPLE 3.8

```

print(5 > 1)

```

If you run that code, the output is "True". Why? Because 5 is greater than 1. Or to put it another way, it is T**rue** that 5 is greater than 1. Let's make another big batch of code to test out all these operators:

CODE EXAMPLE 3.9

```

print(f"5 == 1: {5 == 1}")
print(f"5 != 1: {5 != 1}")
print(f"5 > 1: {5 > 1}")
print(f"5 >= 1: {5 >= 1}")
print(f"5 < 1: {5 < 1}")
print(f"5 <= 1: {5 <= 1}")

```

Save and run that code and you should see an output that looks like this:

```
5 == 1: false  (because 5 is not equal to 1)
5 != 1: true  (because it's true, 5 is not equal to 1)
5 > 1: true  (because it's true, 5 is greater than 1)
5 >= 1: true (because it's true, 5 is greater than or equal to 1)
5 < 1: false (because it's false that 5 is less than 1)
5 <= 1: false (because it's false that 5 is less than or equal to 1)
```

Play around with these. Enter different numbers. Enter the same numbers (5 == 5 or 5 != 5) see what happens.

This might not seem terribly important right now, but you really will use this stuff a lot in the future. A lot of coding revolves around comparing different things.

We'll use this stuff soon when we create if/else statements, and also when we use loops.

VARIABLES

So far we've just been outputting strings manually to the screen, like "Hello World!" and doing some math and outputting that to the screen. Now it's time to talk about variables.

Variables are like buckets. You can put stuff in them and dump stuff out of them.

In Python, you create a variable just by naming it and sticking something in it. Like this:

CODE EXAMPLE 3.10

```

my_name = "John Elder"

```

In that bit of code we created a local variable called "my_name" and we put 'John Elder' in it. We used an operator to do that, an assignment operator. We "assigned" the string "John Elder" to our variable. We'll talk more about assignment operators in just a minute.

Notice how I named the variable my_name with an underscore. Variable names should be descriptive without going crazy. You wouldn't name a variable this_variable_is_my_name – though I guess technically you probably could.

Keep them short but descriptive, and try to use underscores to separate words in the variable name. You could just as easily have named the variable myname, or MyName though in Python we tend to use underscores to separate words in variable names.

Also keep your variable names lowercase.

Variables are great because from now on within your program, you can use that variable however you want. Let's add another line of code to our program to output my name to the screen:

CODE EXAMPLE 3.11

```

my_name = "John Elder"
print(my_name)

```

Run that program and what do you see? You should see my name, John Elder.

We don't have to put text in a variable, we can put numbers in them too:

CODE EXAMPLE 3.12

```

my_name = "John Elder"
my_age = 38

```

Notice how I didn't put quotation marks around the number 38? There's a reason. Without quotes, Python knows that this is a number. That's important because if we wanted to do math with that variable later, we could!

If you put quotes around the 38, then Python thinks that you're not using a number, but instead using a string. If you try to do math with that variable later, you would run into trouble.

In fact, let's try that right now. Create a new program.

MATH WITH VARIABLES

CODE EXAMPLE 3.13

```

number_1 = 5
number_2 = 10
print(number_1 + number_2)

```

Run that code and see what you get. You should see 15 output to the screen. There are a couple of things to see here. First, you'll notice that we can do math with variables just like we did math with numbers earlier.

Instead of typing 10 + 5 into your program, we assigned 10 and 5 to variables and then did the math on the variables. That's a much better way to do math for a lot of different reasons.

The second thing to see is that we didn't use quotation marks around either of those two numbers. Just for fun, wrap one of the numbers in quotation marks and then run the program and see what happens:

CODE EXAMPLE 3.14

```

number_1 = 5
number_2 = "10"
print(number_1 + number_2)

```

What happened? The terminal threw up a big nasty ERROR and wouldn't add your stuff! The error says something like: "TypeError: unsupported operand type(s) for +: 'int' and 'str' ".

What's going on? Well, our number_2 variable isn't holding a number, it's holding a string. And Python won't let you add a number and a string. That would be like telling it to add 27 + apples. What would the answer be? Who knows! You can't add numbers and words.

If you forget, there are ways to convert a string to an integer, that's a little beyond the scope of this intro section, but look up string to integer if you want to see how (actually, we'll discuss it a little later in the book).

APPLES TO APPLES

We've discovered that you can't add numbers and words, but can you add two words with Python? Let's see!

CODE EXAMPLE 3.15

```

fruit_1 = "apples"
fruit_2 = "oranges"
print(fruit_1 + fruit_2)

```

Care to make a wager what will happen? Will it throw up an error? Actually, it should output this:

```
applesoranges
```

As it turns out, a plus sign (+) for strings is used to concatenate the two variables. Whenever we're working with strings and you want to add something in, or concatenate it in, use the plus sign.

CODE EXAMPLE 3.16

```

fruit_1 = "apples"
fruit_2 = "oranges"
print("I Like " + fruit_1 + " and I like " + fruit_2)

```

This bit of code will output "I Like apples and I like oranges" to the screen. Remember earlier when we used f-strings? We can also use variables inside those curl brackets like this:

CODE EXAMPLE 3.17

```

fruit_1 = "apples"
fruit_2 = "oranges"
print(f"I Like {fruit_1} and I like {fruit_2}")

```

In fact, that way seems a little cleaner to me so I would probably use that instead of a bunch of plus signs. But there are certainly times when either way would be acceptable.

MORE FUN WITH MATH AND STRINGS

Oh, we’re not done yet! What happens when you try to do other types of math on variables? Let's see!

CODE EXAMPLE 3.18
```

fruit_1 = "apples"
print(fruit_1 * 5)

```

Care to wager what will happen? You might be surprised to discover that Python will print out apples 5 times to the screen, like this:

```
applesapplesapplesapplesapples
```

I really don't know why you'd ever want to do that...but hey, go crazy!

What else works? Well division doesn't work, you'll get an error if you try to print fruit_1 / 5. You'll also get an error if you try to subtract from apples.

Not surprisingly, if you use a modulus to print fruit_1 % 5, Python will throw an error as well.

ASSIGNMENT OPERATORS

We've got one last thing to talk about in this chapter and I left it till last because it deals with numbers, variables, and operators.

Assignment Operators are used to assign things. We've seen one already when we assigned a value to our variable using an equal sign (=).

The other main Assignment Operators deal with numbers.

Table Of Assignment Operators

=	Assigns something from the right to the left
+=	Adds and Assigns
-=	Subtracts and Assigns
*=	Multiplies and Assigns
/=	Divides and Assigns
%=	Modulus and Assigns
**=	Exponents and Assigns

What's going on here? The first one is easy; we already understand that one. But what's going on with these other operators?

I like to call these my lazy operators. Coders are some of the laziest no-good layabouts on this planet! We want to work as little as possible. That's why we write code to do stuff for us. These operators allow us to write even less code. Here's how.

Let's say I have a variable (number_1) and I want to add 27 to it. I could do it like this:

CODE EXAMPLE 3.19

```
1.
2.  number_1 = 14
3.  number_1 = number_1 + 27
4.  print(number_1)
5.
```

That's perfectly acceptable. But...well...I had to type out number_1 twice there on line 3. I mean two whole times...I'm too lazy for that! Instead, I could have used the += assignment operator like this:

CODE EXAMPLE 3.20

```

number_1 = 14
number_1 += 27
print(number_1)

```

What will that print out? 14? 27? Nope…it will print out 41.

Now that's much better! Our assignment operator took the value of number_1, added 27 to it, and then assigned that value back to number_1 (erasing the original value that was in there).

All the other assignment operators work the same way. -= subtracts and assigns; *= multiplies and assigns; /= divides and assigns; %= does that weird modulus thing and then assigns; and **= calculates an exponent and then assigns.

You don't need to use manually coded numbers either. You can use two or more variables like this:

CODE EXAMPLE 3.21

```

number_1 = 14
number_2 = 27
number_1 += number_2
print(number_1)

```

That will add number_2 to number_1 and assign the output to number_1. Remember, whenever you use an assignment operator, the original value of your variable gets erased. It's gone forever, well unless you run the program again :-p

So when we started the program, number_1 was equal to 14. During the program that 14 gets erased and number_1 becomes 41 (14 + 27). Just keep that in mind.

GETTING USER INPUT

Ok, so I know I said that the assignment operators would be the last thing we talked about in this chapter but I just flat out lied. I do that sometimes.

Before we finish this chapter we're going to learn something that's actually fun, and really easy.

So far, when we run our programs they just do something and output something to the screen. We haven't really been able to interact with them at all.

Now we're going to learn how to add input into the program after it starts running. We do this by using something called the input() function.

CODE EXAMPLE 3.22

```

name = input("What's Your Name: ")
print(f'Hello {name}')

```

Give that a run and see what happens. You probably already figured it out; it asks for your name, and then after you type in your name, it tells you hello.

The input() function takes user input and assigns it to our name variable. Once assigned, we can use that variable in any way we would normally use a variable.

USING INPUT WITH NUMBERS

This is pretty cool! But what if we want to input numbers instead of words?

By default, input() thinks you're entering a string (text) when you enter things. You have to tell it that you're looking for a number (integer).

You can do that using the int() function…(int as in *integer).*

Here's how we do it:

CODE EXAMPLE 3.23

```

num1 = input("Enter A Number: ")
num2 = input("Enter Another Number: ")
print(f'{num1} + {num2} = {int(num1)+int(num2)}')

```

Notice how we wrapped our variables in the int() function? That converts the strings into integers.

Without int(), the program would simply smoosh together the two numbers you entered, it wouldn't add them.

So if you entered a 1 and a 5, the program would output "15" just like it outputted "applesoranges" earlier when we tried to add those two words (it concatenates them).

So I think we're done with this chapter. You learned about math, math operators, comparison operators, assignment operators, variables, and user input!

We're slowly getting into more complicated and interesting things!

CHAPTER THREE EXERCISES

1. Write some code that outputs your name to the screen

2. Write some code that outputs your name to the screen 10 times using math

3. Write some code that asks how many Apples you'd like to purchase and then outputs the response

4. Write some code that asks for a person's first name on one line, then after they type it in, asks for their last name. Then tells them "Hello first last name"

5. Write some code that asks (one line at a time) what your name is, where you live, what your phone number is, and what your favorite color is; then outputs all that info to the screen one item per line.

CHAPTER FOUR

IF/ELSE STATEMENTS

We've played with some basic math, and tinkered with variables, now it's time to start with the first of our actual real programming topics. Well, I say real...everything's been real so far; but it's all been really easy so far.

I guess that's not going to change, because IF/ELSE statements are really easy too...they just feel a little more *programmy* then things we've done so far.

The great thing about IF/ELSE statements is that they give us a lot of power to do all kinds of other things. Logic things.

I've been writing code since I was seven years old...that's a loooong time. I think IF/ELSE statements were one of the first things I ever learned to do. I used them to make "choose your own adventure" games on my Commodore 64 computer.

Choose your own adventure games were actually books we had as kids. "You're standing in front of two doors, one leads left, the other right...to open the door on the left, turn to page 27, to choose the door on the right, turn to page 12".

...That sort of thing. They were all the rage in the 80's but we didn't have the Internet back then, or cell phones; so I guess that's why.

IF/ELSE statements allow you to make choices and decisions in your code and do different things based on different things.

"Do different things based on different things"...yeah that makes sense I guess.

Enough jibber jabber, let's just start coding.

BASIC IF STATEMENTS

IF statements in Python are really easy.

CODE EXAMPLE 4.1

```
1.
2.  x = 41
3.  if x == 41:
4.    print("X Does in fact equal 41!")
5.
```

So let's look through that code. In line 2 we're just creating a variable (x) and setting it equal to 41.

In line 3 we start our IF statement. They follow this format:

if conditional:
  do something here

So in line 3 we look to see if x equals 41 (remember to use double == when checking for equality, not a single = which would assign something).

x == 41 is a conditional statement, but you could use any condition you want (comparison operators are great here). If x > 2 that's a conditional. If x != 17...you get the idea.

If line 3 is TRUE (ie if x does equal 41, which it does in our case) then execute line 4. If line 3 is not true (false), then skip line 4 and go right to line 5...notice that line 4 is indented. Python is tab specific, you need to use the tab button on your keyboard to tab it over.

Pretty simple, right! Let's change line 3 to make it false, just to see what happens.

Can you hazard a guess?

```
CODE EXAMPLE 4.2
1.
2.  x = 41
3.  if x == 42:
4.    print("X Does in fact equal 42!")
5.
```

All we did was change line 3 to 42 (instead of 41). When you run that program, nothing happens. Why? Because line 3 gets executed, determines that x does not equal 42, and then skips to line 5 which is nothing.

IF/ELSE STATEMENTS

IF Statements are nice, but what if you want to make more than one decision? If x == 41 do this, otherwise do something else? Enter the **IF/ELSE** Statement!

```
if conditional:
  do something here
else:
  do something else
```

Now we're getting somewhere! With IF/ELSE Statements we can use Logic to do just about anything we want. Let's see this in action:

CODE EXAMPLE 4.3

```
1.
2.  x = 41
3.  if x == 41:
4.    print("X Does in fact equal 41!")
5.  else:
6.    print("X Does NOT equal 41!")
7.
```

When we run this code, we get "X Does in fact equal 41!" because x does equal 41 in this example. If we changed line 2 to another number, say 42, and ran the code again, we would get "X Does NOT equal 41!".

Play around with it. You'll use IF/ELSE statements a lot...I mean a LOT.

IF/ELIF

To throw just a little more complexity into the mix (and give you one more tool that's pretty useful) we can use the IF/ELIF statement (Else If). This allows you to test against a second condition.

Sometimes people get a little confused with this one, but it's not too bad, and actually really helpful. Before we talk about it, let's just take a look.

CODE EXAMPLE 4.4

```
1.
2.  name = "John"
3.  if name == "Bob":
4.    print("Hi there Bob!")
5.  elif name == "John":
6.    print("What up John!!")
6.  else:
7.    print("I don't know who you are!")
8.
```

So we set our variable to "John", line 3 checks to see if our name is Bob, it's not so it moves down to line 5, where we give it another condition to check for. That's our elif. In this case our variable is in fact "John" so this program will print out "What up John!!" and then end.

If we changed the variable to "Ralph", then the program would print out "I don't know who you are!"

We can slap in as many elif's as we want, so you can check against as many different things as you can dream up.

MULTIPLE CONDITIONALS

So far we've only been testing one condition in our IF statements, but you can test more than one, using things line AND and OR.

CODE EXAMPLE 4.5

```
1.
2.  name = "John"
3.  if name == "John" or name == "Bob":
4.    print("Hi there John or Bob!")
5.
```

Notice the simple difference in line 3? We slapped an "or" in there and added another condition to test against. In this case, if our variable is either John *OR* Bob, the program will return "Hi there John or Bob!"

Instead of or, we could have used "and" as well. For and to be true, our variable would have to be both John and Bob...which I don't really think that's possible. So our program would return nothing in that case.

AND OR

For "And" to evaluate to True, all conditions must be True, otherwise it will return False. For "Or" to evaluate to True, any of the conditions must be True; if none of the conditions are True, the whole thing evaluates to False.

In some programming languages, you see symbols used for "And" and "Or" such as && for and, || for or. Python doesn't allow that.

Using And/Or with Logic gives you lots of control. It might look something like this:

CODE EXAMPLE 4.7

```

name = input("Enter Your Name: ")
if name == "John" or name == "Bob":
  print(f"Hi there {name}!")
else:
  print("Hi there Stranger!")

```

And/Or is very useful for many different things!

CHOOSE YOUR OWN ADVENTURE

Alright, let's take what we've just used and build our own simple little choose your own adventure game.

Before we start, I want to share one little trick that's going to be pretty important for our game (and really any time you input user data).

Let me ask you this; if we build a choose your own adventure game that asks someone to pick a direction – left or right – and our user types in LEFT (all capitals) or Left (first word capitalized)...what happens?

We'll be running an IF/ELSE statement to check if their input says "left" or "right" but that's not the same as "LEFT" or even "Left". Computers seem smart, but they really aren't, they're Case Sensitive. You have to be explicit.

So we need to make sure that the thing a user types in will be matchable by our program. That's easy enough if we standardize our input.

STRING MANIPULTION

Python makes this very easy. We'll just take the user input and change it to lowercase. But how? Python has many function for manipulating strings. Here's a few fun ones.

Table 4.1 String Manipulation

lower()	Convert to all lowercase
upper()	Convert to all uppercase
capitalize()	Capitalize just first letter
title()	Capitalizes first letter of each word
swapcase()	Switches capitalized to lowercase & vice versa
len()	*Returns character length
rstrip()	Removes trailing spaces

So how do we use these? For the most part, you'll use them on the ends of your variable. So if you wanted to convert your variable "name" to lowercase, you would call name.lower()

CODE EXAMPLE 4.8

```

name = input("Enter Your Name: ")
print(f"Hi there {name.lower()}")

```

Notice how we just slapped the .lower() on the end of the variable in real time? You could also assign it there permanently.

CODE EXAMPLE 4.9

```

name = input("Enter Your Name: ")
name = name.lower()
print(name)

```

In this example, name will forever be lowercase because we converted it and assigned it back.

Not all String Functions work by just slapping them on the end of the variable. Take the len() function for instance. You don't slap that on the end of your variable, you put the variable inside the parenthesis, like so:

CODE EXAMPLE 4.10

```

name = input("Enter Your Name: ")
print(f"The name {name} has {len(name)} characters.")

```

If I ran that code and entered my name, John, the program would return:

The name John has 4 characters.

How do you know which go on the end of variables and which go inside the function? I guess you just have to memorize them. But to be honest, I don't do that. I just slap them on the end of the variable and run the code to see if it returns an error.

If it does, I know that the variable goes inside the function. Sloppy? Maybe! But I'm a coder, we're all lazy and a little sloppy.

These are just a few String Manipulation functions. There are many more built into Python.

Refer back to table 4.1 and play around with the other string manipulation methods I showed you.

The rstrip one might be confusing. It removes spaces at the end of your string.

Did you notice the difference between capitalize() and title()? Capitalize just capitalizes the very first word while title() capitalizes the first letters in all the words of your string (if there are more than one word).

So with capitalize(), john elder becomes "John elder" and with title(), it becomes "John Elder".

And of course, upper() just blasts every letter to uppercase, like "JOHN ELDER". That hurts to look at!

CHOOSE YOUR OWN ADVENTURE

Ok let's get back to the fun stuff. Let's choose our own adventure. This thing could easily spiral out of control so let's confine it to one question and have that question ask for one of two answers. Sound good?

The goal of our game is to find the "Python Princess". Isn't that the goal of most games? Yep.

So let's start by asking the users name and converting it to lowercase. And then let's ask our question; be sure to convert it to lowercase right away.

CODE EXAMPLE 4.11

```

from os import system
system("clear")
print("Welcome To Choose Your Own Adventure!")
print("The goal is to find the Python Princess...")
name = input("Enter Your Name: ")
name = name.lower()
system("clear")
print("You're standing in front of two doors...")
print("Do you want the door on the left or right?")
question = input().lower()
if question == "left":
  system("clear")
  print("You fell into a pit and died! GAME OVER")
elif question == "right":
  system("clear")
  print(f"Congratulations {name.capitalize()} you found")
  print("the Python Princess!  YOU WIN!")
else:
  system("clear")
  print("Sorry, I don't recognize your response GAME OVER")

```

Ok, so there's a couple new things going on here. First off right on line two and three we see something new (and also line 8, 13, 16, and 20) system "clear". That just clears the screen to make things easier to read.

The other thing you'll notice is how we sort of stacked things onto the input statement in line 11 (*question = input().lower()*). Yep, you can do that! Python is object oriented, and you can keep stacking things on like that as much as you want.

You'll also notice that we saved the name variable using lower() so that we could test it against our lowercase if/else statement, but in line 17 we capitalized it again when we output the name onto the screen.

That capitalize statement only capitalizes our name there, it doesn't change the variable.

Finally, you'll notice that we put an else statement there at the end as a catchall in case someone entered something other than "left" or "right" to answer our question. It's always a good idea to think of all the weird ways a user might use your program and try to write code to catch all that weirdness!

You can easily expand this game to add more questions and make it more fun to play. But for the purposes of this book I just want to keep it simple and use it as an example.

Now we're cooking! We're starting to do stuff that actually looks like coding!

And it's still pretty easy!

Yeah…that's Python for you!

CHAPTER FOUR EXERCISES

1. Create a simple math flashcard game that asks a user what two numbers added together equal and tell them whether or not they got the answer correct.

2. Write some code that asks for a person's full name (first and last name) and then output their name with both first and last name capitalized

3. Write some code that asks for a person's name and then tell that person how many characters are in their name (added them up manually yourself to see if the answer is correct!)

CHAPTER FIVE

LISTS

So we're moving right along! So far so good, right? Nothing too terribly difficult? Good.

In this chapter I'm going to teach you all about Lists (what other languages call Arrays).

People sometimes have a hard time wrapping their head around Lists. It's ok, they can be a little tricky to understand...but not really. They're actually pretty simple and they're something you're going to use forever, no matter what programming language you happen to be using.

So what is a List?

Basically a List is just a variable...but instead of holding one thing, Lists can hold many many things…like, well, any list you've used in real life.

For instance, imagine we have a variable called "name" and we set it equal to "John".

We already know how to do that, it's drop dead simple. But what if we wanted it to also hold "Tim", "Mary", "Beatrice", and "Bluto"?

A variable can't really do that, but a List can! And this is very useful.

CREATING A LIST

We create Lists almost the same way we create variables:

CODE EXAMPLE 5.1

```

names = ["John", "Tim", "Mary", "Beatrice", "Bluto"]

```

Pretty easy! Just stick whatever information you want to put in your List into the brackets [ ] and be sure to separate each item by a comma. How do we get the information out of our List? Just like this:

CODE EXAMPLE 5.2
```
1.
2.  names = ["John", "Tim", "Mary", "Beatrice", "Bluto"]
3.  print(names[2])
4.
```

Can you guess what line three will print out onto the screen?

Trick question...it's not "Tim" (since it looks like Tim is the 2nd item in our List). The answer is actually "Mary".

Why? Because Lists start at zero. So "John" is the 0th item in our List, Tim is the 1st, Mary is the 2nd, Beatrice is the 3rd, and Bluto is the 4th.

Burn that into your memory! LISTS START WITH ZERO!

What happens if we replaced line three with just print(names) (instead of print(names[2]))? Our program would simply print out every name in our List. Give it a try.

PUTTING DIFFERENT THINGS INTO LISTS

In the example above we put strings into our List. We could also put numbers and variables as well (and other things too)!

CODE EXAMPLE 5.3

```python

variable_1 = "Tim"
names = ["John", variable_1, "Mary", 41, "Bluto"]
print(names[1])

```

What did we do here? We put a **variable** and a **number** into our List. If we ran this code, it would print out "Tim" because "Tim" is what's inside variable_1.

Notice how we didn't put quotation marks around variable_1 or the number 41. That's important. If you put quotation marks around them, they become strings, not variables or numbers.

You can put other things in Lists too ,things like functions and objects; but we haven't talked about functions or objects yet so we'll go over that later. You can also put other lists in your list…mind blown!

ANOTHER WAY TO CREATE A LIST

There are other ways to create a list, for instance you could create one explicitly:

CODE EXAMPLE 5.4

```python

names = []

```

With that code we've created a new list (and called it names) and told Python that we're going to just leave it blank, at least for now...

In this case, our List doesn't actually have anything in it at the moment, but it has space for as many things as you want, whenever you get around to putting things in there.

In fact, you can see how many items are in any given list like this:

```
CODE EXAMPLE 5.5
1.
2.  names = ["John", "April"]
3.  print(len(names))
4.
```

Line three should output 2 in this particular case.

ADDING THINGS TO A LIST LATER ON

We know how to put stuff in a List with our code, but what if we want to add stuff later on...or change stuff, or remove stuff? Easy.

Let's add a name to the end of our List:

```
CODE EXAMPLE 5.6
1.
2.  names = ["John", "April"]
3.  names.append("Bob")
4.  print(names)
5.
```

Append() does what it sounds like…adds an item to the end of your list.

That's fun, but what if you want to add something to your list at a specific position in the list? Say first, or in the middle…or anywhere else?

You can use insert() for that. Insert() takes two arguments; position and value.

CODE EXAMPLE 5.7

```

names = ["John", "Tim"]
names.insert(0, "Bob")

```

That code would insert "Bob" into the first position of your list. (remember, lists start at zero!!)

ADDING MULTIPLE ITEMS TO A LIST

To add multiple items to the end of your list, use extend().

CODE EXAMPLE 5.8

```

names = ["John", "April"]
names.extend(["Tim", "Bob"])

```

Notice that we passed in another list: ["Tim", "Bob"] but that list isn't added to our original list, the *items* in the list are added individually. So our list becomes:

["John", "April", "Tim", "Bob"] and not *["John", "April", ["Tim", "Bob"]]*

REMOVING ITEMS FROM A LIST

There are several ways to remove an item from a List. Imagine we want to remove the last item in our List:

CODE EXAMPLE 5.10

```

names = ["John", "Tim", "Mary", "Beatrice", "Bluto"]
names.remove("Bluto")

```

This will remove poor Bluto from our List. It's bad enough they're named Bluto...now we've gone and deleted them!

REMOVE A SPECIFIC INDEX NUMBER

To remove a specific positioned/numbered item of a List, use **pop()**:

CODE EXAMPLE 5.11

```

names = ["John", "Tim", "Mary", "Beatrice", "Bluto"]
names.pop(0)

```

Alas, we removed my name from our List! Oh the humanity!

If you wanted to remove the second item of our list (Tim):

CODE EXAMPLE 5.12

```

names = ["John", "Tim", "Mary", "Beatrice", "Bluto"]
names.pop(1)

```

That will remove poor Tim from our List (remember, Lists start at zero – I'll keep reminding you till it's burned into your brain!)

As I mentioned earlier, you can also remove a specific item from the List, for instance a particular name, but there's danger there:

CODE EXAMPLE 5.13

```

names = ["John", "Tim", "Mary", "Beatrice", "Bluto"]
names.remove("Mary")

```

That will obviously remove Mary from our List. BE CAREFUL HERE! If there are more than one Mary in the List, delete will remove just the first one.

So I'd be cautious using that unless you know there's only one particular item in your List (or if you don't care that you're deleting just the first instance of that item in the List).

MULTI-DIMENSIONAL LISTS

Remember when I said that we can put all kinds of things into a List? Well, it turns out that you can put other Lists into a List. Did I just blow your mind?

CODE EXAMPLE 5.13

```

names = ["John", "Mary", "Beatrice", "Bluto", [1,2,3,4]]

```

So let's take a look at that code. You'll see we've got our old normal List with names John, Mary, Beatrice, and Bluto; but then we've added another thing to the end – another List with the numbers 1, 2, 3, and 4 in it.

Notice how we separated that second List with a comma, just like we separate everything in our List with a comma.

So what would happen if we print our List to the screen? It would print out our List as you'd expect:

["John", "Mary", "Beatrice", "Bluto", [1,2,3,4]]

So that's cool…but how do we access the stuff in that second List? It's pretty easy. Let's say we wanted to print out the number 3 in our second List (remember, that's the second numbered item in the second List because Lists start at zero, even embedded Lists).

CODE EXAMPLE 5.14

```

names = ["John", "Mary", "Beatrice", "Bluto", [1,2,3,4]]
print(names[4][2])

```

A little weird, but not too bad…right? Our nested List is the fourth item in our original List, and the number 3 is the second item in that List.

You can nest another List inside that second List, and another one in that List and another in that one…but you might go a little crazy if you do.

Another way to create a multidimensional List is by creating a second List separately and then adding the name of that List to your original list (in the same way that we added Variables to our List earlier):

CODE EXAMPLE 5.15

```

numbers = [1,2,3,4]
names = ["John", "Mary", "Beatrice", "Bluto", numbers]
print(names[4][2])

```

See how we created a separate List called numbers, and instead of manually typing in [1,2,3,4] into our 'names' List, we just typed in the name of our numbers List.

If we run this code, we'll get the same "3" answer that we got last time.

So those are Lists! Lists are super useful, and really not that difficult to understand at all.

Just think of them as variables that hold more than one thing.

And remember you can put just about anything into a list, strings, numbers, variables, functions, other lists…if you can dream it up; you can stick it into a List!

CHAPTER FIVE EXERCISES

1. Create a List with the names of five people you know and output the second name to the screen.

2. Create a List where the first item in the List is a math problem, like 1 + 1 and the rest of the items are names. Output the first item to the screen. (WOAH, MATH can be an item in a List?!)

3. Create a multi-dimensional List with 4 items, and each item is itself a List containing a person's name, their address, and phone number (make up the info). Output the second item in your multidimensional List.

4. Output just the phone number of the third item in your List from the last question.

CHAPTER SIX

LOOPS

Now we're really cooking! We've come a long way, from Math to Variables to IF/ELSE statements, and on through Lists. You've got the foundations of a pretty solid understanding of basic coding so far, but now it's time to dive into something a little more complicated…Loops.

Ok, they aren't *really* much more complicated. If you didn't have any trouble wrapping your brain around IF/ELSE statements, then you shouldn't have any trouble with simple Loops.

So what is a Loop?

Basically a Loop is just what it sounds like…it does something over and over and over and over again…looping through again and again and again, until you tell it to stop.

Let's start with the number three. Check to see if three is equal to ten. If it isn't, add one to it, then repeat. Now three has became four, check to see if four equals ten, if not add one and repeat.

Keep looping through that until our number equals ten. Then stop. Or do something else. Or whatever!

That's basically a Loop.

You're going to use Loops a lot in programming. They're one of those things you'll just always use. Loops and comparisons sort of go together. You'll have noticed in my lame Loop explanation above that I said the word "If" a bunch of times (if three equals ten, if four equals ten, if, if, if).

Loops need conditionals to determine whether or not the criteria for the Loop has been met yet.

Every programming language has Loops, in fact they all have a bunch of different loops that you can use, and Python is no different. Most programming languages handle Loops in roughly the same manner, so if you know Loops in one language, it's easy to learn them in another.

We're going to cover the two main loops to get you started, namely W**hile and For Loops.**

WHILE LOOPS

While loops are pretty simple so they're a good place for us to start. A while loop basically says "While something is true, do something". As soon as the thing stops being true, stop looping. Let's take a look:

CODE EXAMPLE 6.1

```

num = 0
while num < 10:
  print(num)
  num += 1

```

So what's going on here? A While Loop has this general format:

```
while  conditional:
  do something
```

We remember conditionals from back at the beginning of the book, we used them with IF/ELSE statements. This is where our comparison operators become handy.

So let's take a look at our code. On line 2 we just created a variable and set it equal to zero. Line 3 is where the while loop starts, num < 10 is our conditional.

The loop says "Hey, take a look at num, if num is less than 10, do the code that's in line 4 and 5, then check (loop) again".

So the first time we run the program num is zero. Zero is definitely less than ten, so our code executes line 4 and prints num to the screen.

Line 5 takes our variable num and adds one to it. Remember that += operator we talked about a few chapters ago?

You could also write that line of code like this:

```
num = num + 1
```

but num += 1 is more elegant. So when our program first runs, it will check to see that zero is less than ten, then print 0 to the screen and then add one to it.

It then loops around and starts over. Now when it checks num, it sees that num equals one, which is still less than ten, so it executes the code and adds one to num, printing out 2 to the screen.

And on and on… so the output of our code looks like this:

```
0
1
2
3
4
5
6
7
8
9
```

After the program loops through ten times it stops. Why? Because the tenth time the program looped, num became 10. The next time the program started to loop through it asked itself "Hey, is 10 less than ten?"

10 is not less than 10. 10 is equal to 10, so the program stops.

That's the thing, you see…a while loop just keeps looping until the thing stops being true…*while true* keep on looping!

THE INFINITE LOOP PROBLEM

You want to be careful when you create your loops and not build an infinite loop. It's happened to the best of us (in fact, I accidentally did it earlier when I was writing the code for this chapter!).

So what is an infinite loop?

Well, an infinite loop is a loop that just keeps looping forever. It's conditional never gets met and it just keeps looping and looping and looping and looping for all time.

An infinite loop is not a good thing, they'll more than likely cause your program to crash and can even tie up all the resources on your computer or server – causing it to crash.

Yeah, it happens…and it's pretty easy to do. Here's an example, try to spot the problem here:

CODE EXAMPLE 6.2
```
1.
2.  num = 0
3.  while num < 10:
4.    print(num)
5.
```

Did you see it? That's basically the exact same code that we just wrote at the beginning of this chapter, I just left off one little thing.

Line 4…all it's doing is printing out our num variable. The code doesn't increment it by one like it did in the previous example.

So what's going to happen if you run that code? Give it a try! You might as well experience the joy and the terror of an infinite loop…I'll wait while you go run that code.

So what happened? It printed out:

```
0
0
0
0
0
0
0
```

```
0
0
0
0
0
0
```

Over and over and over again and you couldn't get it to stop, right? Yep.

You'll likely have to close your terminal and open a new one to get it to stop. Or you might be able to hit the ctrl button on your keyboard plus to "c" key, to break out (Ctrl+c then hit enter). Sometimes that works, sometimes it doesn't.

Either way, don't worry too much about it…like I said, it happens to the best of us from time to time!

BREAK/CONTINUE

If for some reason you need to break out of your loop, you can use the Break/Continue statement and your old trusty IF/ELSE Statement.

The code looks pretty much the same as a while loop:

```
while  conditional
  do something
 if conditional
       break
```

So let's take a look at a basic code example:

```
CODE EXAMPLE 6.3

num = 0
while num < 10:
  num += 1
  if num == 5:
    break
  print(num)

```

That will output the following:

```
1
2
3
4
```

So what's the deal here? We're just sort of sliding in a conditional that stops the whole loop for any reason you want.

You can do basically the same thing with the CONTINUE statement as well, but instead of stopping, it will skip over something:

```
CODE EXAMPLE 6.4

num = 0
while num < 10:
  num += 1
  if num == 5:
    continue
  print(num)

```

That should print out: 1-10 but not 5. So 1,2,3,4,6,7,8,9,10

FOR LOOPS

A For Loop is a little different, it runs a specific number of times over a range of numbers; and of course, you specify the range. Check it out:

CODE EXAMPLE 6.4
```

for num in range(6):
  print(num)

```

This will print out:

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

You'll notice we used num as our variable but we didn't set it to anything. "For" sets it to iterate over the range that we've specified (in this case from 0 to 5 – ranges, like Lists, start at zero). You can specify a starting point if you don't want to start at 0, like this:

for num in range(1, 6)

In example 6.4, we printed out num, but you don't have to use the variable in the output.

Often you'll use a For Loop to do something that has to be done many times repetitively. For instance:

CODE EXAMPLE 6.5

```

for num in range(6):
  print("I LOVE CHEESE!")

```

…will ouput I love cheese 6 times (from 0 to 5):

```
I LOVE CHEESE!
I LOVE CHEESE!
I LOVE CHEESE!
I LOVE CHEESE!
I LOVE CHEESE!
I LOVE CHEESE!
```

That's fun! But the real power of "For" loops is iteration over something…usually a list.

You can use a For loop to grab each item out of a List (or really anything), and do something with it…

CODE EXAMPLE 6.6

```

names = ["John", "Mary", "Tim"]
for name in names:
  print(name)

```

That will iterate through the list of names, and print out each name to the screen:

```
John
Mary
Tim
```

Notice how I named the variable in the for loop "name"? That's the singular version of the name of the list…which was "names".

You can iterate over most things, even variables:

CODE EXAMPLE 6.7

```

name = "John Elder"
for x in name:
  print(x)

```

That's going to output:

```
J
O
H
N

E
L
D
E
R
```

You'll notice that it even printed out the space in "John Elder". Iteration is fun!

USING ELSE WITH FOR

Remember when we learned about IF/ELSE statements? You can actually use an Else statement on a for loop to execute something after the loop has finished.

CODE EXAMPLE 6.8

```
1.
2.  for x in range(3):
3.    print(x)
4.  else:
5.    print("All Done!")
6.
```

That would output:

```
0
1
2
All Done!
```

You might be asking yourself why you really need the else statement at the end. Couldn't you skip it and just print out line 5?

CODE EXAMPLE 6.8

```

for x in range(3):
  print(x)
print("All Done!")

```

Which would print out:

```
0
1
2
All Done!
```

So, yeah…that works too. But there are instances when you might need that last bit (the "All Done") to execute only under certain circumstances, and the Else statement will come in handy.

PASS STATEMENT

For Loops can't be empty…and usually they aren't. But if for some reason yours needs to be empty, you can use the Pass Statement.

CODE EXAMPLE 6.9

```

for x in range(3):
  pass

```

Run that code and…nothing happens. If you left off the "pass", you'd get an error. With the "pass" on there, the code just returns nothing.

So those are the basic Python loops. Personally I find myself using the For Loops most often because iterating through Lists is something that happens a lot.

But While loops are super useful too, especially when you have a counter that needs incrementing.

FIZZ BUZZ

If you've ever gone on a coding interview of any kind, then you've probably had to do some sort of code test to prove your skills. Companies have lots of different puzzles and tests to give you to see whether or not you've got the chops for the job.

Personally, I think code interviews are a ridiculous waste of time and an incredibly bad way to choose someone to hire but that's just me.

But I digress…a common code puzzle is the Fizz/Buzz challenge, and now that we know how to do loops and IF/ELSE statements, we have all the tools we need to complete this test. So let's take a look!

The problem is simple. Print out every number between 1 and 100, one number per line, but if the number is divisible by three, print out "Fizz"; if the number is divisible by five, print out "Buzz"; and if the number is divisible by both three AND five, print out FIZZ BUZZ.

Take a minute and think about this problem yourself before reading on. You have all the tools you need to do this; and there are many different ways to do it.

The idea is to write the most elegant code possible (ie keep it short George!).

How would you do it?

Let's break it down. First things first, we need to loop through the numbers 1 through 100 printing each to the screen. We just learned how to do that no problem.

CODE EXAMPLE 6.10
```
1.
2.  for x in range(1, 101):
3.    print(x)
4.
```

Now we need to figure out whether any given x is divisible by 3, 5, or both 3 and 5. Sounds like a problem for an IF Statement and a few modulus operators (remember %).

To determine whether or not a number is divisible by 3, the modulus would return zero (no remainders).

CODE EXAMPLE 6.11
```
1.
2.  for x in range(1, 101):
3.    if x % 3 == 0 and x % 5 == 0:
4.      print(f"{x} FIZZ BUZZ!")
5.    else:
6.      print(x)
7.
```

So far so good…line 3 might look a little confusing but it's not too bad. We briefly talked about "and" earlier (it lets you check two different conditions instead of just one).

So line three says basically if, when you divide x by 3 you get a remainder of zero (meaning x is divisible by 3) AND when you divide x by 5 you also get a remainder of zero (meaning x is also divisible by 5) then print to the screen the words "FIZZ BUZZ!"

Else, just print the number to the screen.

If we run that code, we can spot check that it's working by searching for the first couple of numbers that are divisible by 3 and 5. 15 comes to mind, as well as 30…and I can spot check that our code is printing FIZZ BUZZ! Instead of 15 and 30. So we're good so far.

Now we need to break it down further and check to see if x is divisible by JUST 3 or JUST 5. Sounds like a job for Elif.

CODE EXAMPLE 6.12

```
1.
2.  for x in range(1, 101):
3.    if x % 3 == 0 and x % 5 == 0:
4.      print(f"{x} FIZZ BUZZ!")
5.    elif x % 3 == 0:
6.      print(f"{x} FIZZ")
7.    elif x % 5 == 0:
8.      print(f"{x} BUZZ")
9.    else:
10.     print(x)
11.
```

Notice I put a f-string {x} in there so that it would print out the number as well as the words FIZZ, BUZZ, or FIZZ BUZZ just so we can go through and spot check to make sure the program is doing it's job.

```
2
3 FIZZ
4
5 BUZZ
6 FIZZ
7
8
9 FIZZ
10 BUZZ
11
12 FIZZ
13
14
15 FIZZ BUZZ!
16
17
18 FIZZ
19
20 BUZZ
.
.
```

Yep, looks like it works! And just like that we solved a coding puzzle that has stumped many people in interviews. Don't laugh, I personally know people who have not been able to solve that puzzle during an interview.

So we used modulus's and IF/ELSE and IF/ELIF statements, and our basic For Loop… pretty simple!

I said there are many ways to do this…can you think of another way? A simpler way? A way that uses less code? Give it a try…

CHAPTER SIX EXERCISES

1. Try FIZZ/BUZZ again but do it in less lines of code!

2. Create a multi-dimensional List with 4 items, and each item is itself a List containing a person's name, their address, and phone number (make up the info). Loop through the List and output just each person's phone number.

3. Loop through the List from exercise 2 and print out the full information of even items in the List (ie the 2nd and 4th List in your multidimensional List).

CHAPTER SEVEN

FUNCTIONS

Now it's time to learn about Functions. Functions are like little programs inside of your program. If you've programmed in another programming language, you might have called them "Methods", but Python calls them Functions.

Sometimes I slip and just call them methods…for all intents and purposes they're the same thing.

Functions do specific things, at specific times. In fact, Python will ignore any Function you write until you specifically call it. That's different than what we've seen so far.

Up until now, we've written code and whenever we run our program Python executes everything we've written and gives us any output that the code creates.

Not so with Functions. You'll see what I mean in a minute. First, let's create our first Function. Functions have the form of:

```
def name(arguments):
    do stuff
```

And this basic form should look familiar to you by now…it seems like everything in Python tends to look like this. The name should almost always be in lowercase. Let's create an example:

CODE EXAMPLE 7.1

```

def wisdom():
  print("You have to know when to hold em")


```

Go ahead and run that code and see what happens. Did you run it? Nothing happened! Remember, I told you that Functions don't get executed unless you specifically call them.

To call a Function, just type the name of it with parenthesis. Like this:

CODE EXAMPLE 7.2

```
1.
2.  def wisdom():
3.    print("Life is like a box of chocolates...")
4.
5.  wisdom()
6.
```

We call the Function on line 5, and if we run this program the output will be: "Life is like a box of chocolates..."

This is cool, but we've left off part of it…the arguments part. Running a Function that just returns some text isn't very interesting. We want to be able to interact with our Functions, feed them information, have them do something with that information, and then return results based on what it did to that information.

So let's build something more interesting that lets us pass arguments to the Function and have it actually do something.

Hey I know, let's create a Fizz Buzz Function that will tell us if a number is a Fizz or a Buzz (remember in the last chapter fizz numbers are divisible by 3, buzz numbers are divisible by 5, and fizz buzz numbers are divisible by both 3 and 5).

We want to be able to pass any number into it and get a result. So let's do that:

CODE EXAMPLE 7.3

```
1.
2.  def fizz_buzz(x):
3.    if x % 3 == 0 and x % 5 == 0:
4.      print(f"{x} is FIZZ BUZZ!")
5.    elif x % 3 == 0:
6.      print(f"{x} is FIZZ")
7.    elif x % 5 == 0:
8.      print(f"{x} is BUZZ")
9     else:
10.     print(f"{x} is Boring")
11.
12.  fizz_buzz(97)
13.
```

So basically we're using the same code as in the last chapter when we wrote our Fizz Buzz program, but this time we're building a Function. In line 12 we called the Function and passed the number 97 into it.

Do you see how we pass arguments into the Function? When we define the Function in line 2, we slapped that (x) on there, that's basically a variable. Then when we call the Function in line 12, we designate 97 as the argument we want to pass into the Function. That 97 gets passed into the Function via that (x)…or to put it another way, that x variable becomes 97.

Now anytime within the Function that you see x, our program is putting a 97 in there. Cool!

If you run that code it will return "97 is Boring" because 97 is not divisible by 3, or 5.

If we changed line 12 to fizz_buzz(15) and ran it again, the program would output "15 is FIZZ BUZZ!"

Pretty cool.

Functions are fundamental. They're another one of those things that you're going to use forever, no matter what programming language you end up using. Python makes using them pretty easy, as you've already seen.

What else can we do with Functions?

RETURNING TRUE OR FALSE

Sometimes it's useful to find out if something is True or False. Often you'll want to do a certain thing if a thing is True, and do something else if a thing is False. So let's create a Function that will return True if a number is even (divisible by two), and return False if a number is odd.

CODE EXAMPLE 7.3

```
1.
2.  def is_even(x):
3.    if x % 2 == 0:
4.      return True
5.    else:
6.      return False
7.
8.  print(is_even(99))
9.
```

When we run this code, the terminal will output: False. If we changed line 8 to print(is_even(98)), the terminal would output: True.

True and False are a datatype called Booleans and they're useful for all sorts of things.

Notice also how we wrapped our is_even(99) function call in a print function. What would happen if we left that off?

CODE EXAMPLE 7.4

```

def is_even(x):
  if x % 2 == 0:
    return True
  else:
    return False

is_even(99)

```

If you call that code, the terminal wouldn't return anything at all.

ASSIGNING FUNCTION OUTPUT TO A VARIABLE

Function output can also be assigned to a variable, and many times that's what you will do.

CODE EXAMPLE 7.5

```
1.
2.  def is_even(x):
3.    if x % 2 == 0:
4.      return True
5.    else:
6.      return False
7.
8.  my_variable = is_even(99)
9.  print(my_variable)
10.
```

In that example, we called the function in line 8 and assigned it's output to a variable called my_variable.

Then later on, in line 9, we printed out that variable…which in this example would print: "False" to the terminal.

FUNCTIONS CAN HAVE MORE THAN ONE ARGUMENT

Up until now our functions have passed one argument through. But you can pass more than one just as easily:

CODE EXAMPLE 7.6

```
1.
2.  def namer(first, last):
3.    print(f"Hello {first} {last}, nice to meet you!")
4.
5.  namer("John", "Elder")
6.
```

In this program our Function has two arguments (first and last). We call the function by typing in namer on line 5, but this time instead of passing just one variable, we pass two (in this case John and Elder).

Also notice on line 5, we didn't wrap our function call in a print function. Why? Because the function itself prints to the screen on line 3. So we don't have to wrap the function in a print function.

ERRORS

Try running the code in example 7.6 again, but this time only pass one parameter in line 5: namer("John").

You'll notice that the program throws an error. This is because our function is expecting to be passed two arguments, and if you pass only one, the code won't work.

USING DEFAULT ARGUMENTS

The first function we created back at the beginning of the chapter didn't pass any arguments through. Sometimes you want your Function to use default arguments if none are passed. Let's take a look:

CODE EXAMPLE 7.6

```
1.
2.  def namer(first = "John", last = "Elder"):
3.    print(f"First Name: {first}")
4.    print(f"Last Name: {last}")
5.
6.  namer()
7.
```

If we run that code it will output:

```
First Name: John
Last Name: Elder
```

…notice how we didn't pass any variables when we called the Function in line 6? We just called namer(). Since we didn't pass any arguments when we called it, our function fell back on the default arguments we gave it when we wrote the function (John and Elder) on line 2.

Our Function will only use those defaults because we didn't pass any arguments there on line 6. But if we changed line 6 to: namer("Adam", "Smith"), then our program would output:

```
First Name: Adam
Last Name: Smith
```

…because we no longer want to use the default John and Elder arguments. Neato.

Likewise, if we only passed one argument, the function would accept that argument, and then use the default for the other.

So if we did something like: namer("Adam") on line 6, the terminal would output:

```
First Name: Adam
Last Name: Elder
```

First name would be Adam because that's what we passed the function in line 6, and Last Name would be Elder even though we didn't pass that when we called the function; our code fell back on the default last name that we built in when we created it.

So there you have functions. Much of what you do as a Python programmer will resolve around Functions, and you'll learn to use them in more interesting ways as you grow as a coder.

For now, I just want you to become familiar with them and their basic usage. And I think we've covered that here!

CHAPTER SEVEN EXERCISES

1. Re-write the fizz_buzz function to prompt a user to enter a number and then return the function result.

2. Write some code using the same fizz_buzz function but have it print out all numbers between 1 and 100 by calling the function 100 times.

3. Create a function that has 7 things passed to it.

4. Create a function that asks a persons name, allows them to enter the name, and then prints out a welcome message with that name.

CHAPTER EIGHT

DICTIONARIES

Now it's time for Dictionaries…actually, we probably should have talked about Dictionaries right after we talked about Lists…because they're very similar.

But I didn't want to confuse you and I wanted to give you enough time to let Lists sink in.

Remember our names List?

```
names = ["John", "Tim", "Mary", "Beatrice", "Bluto"]
```

That's a pretty simple List, and we access the items in that List using index numbers, starting with zero. So to access the John item in our List we would call: names[0] and to access Mary in our List we would call: names[2].

To access a specific item in a List, we need to know it's index number.

That's fine, there are a lot of times when that will be perfectly acceptable. But when you get right down to it, sometimes being forced to know an item's index number is kind of a hassle.

Enter Dictionaries.

Dictionaries are an awful lot like Lists, but instead of associating an item with its index number; we can associate it with *any damn thing we want!* ™

I don't know about you, but I like having the option to do *any damn thing I want!*™ but then again, I'm 44 years old, live in Las Vegas, and have never had a real job besides running my own startups. So there you go.

By the way; if you want to learn how I build startups and float through life without ever having a real job, check out my site JohnElder.com or grab my "Smart Startup" book or "Living the Dotcom Lifestyle" book from Amazon.

But I digress…Dictionaries. Let's build one that takes the names from our List and output's everyone's favorite pizza:

CODE EXAMPLE 8.1

```

favorite_pizza = {
  "John":"Pepperoni",
  "Tim":"Mushroom",
  "Mary":"Cheese",
  "Beatrice":"Ham and Onion",
  "Bluto":"Supreme"
  }

print(favorite_pizza["John"])

```

So the first thing to notice is that we wrap our Dictionary in curly brackets { and }.

The next thing to notice is that we separate each value of our Dictionary by a comma.

The items in our Dictionary are called "Key" and "Value" pairs.

With a List, the "Key" is an index number (the 2nd item in the List, the 4th item in the List, etc) and that's how we access the items, by calling the index number key.

With a Dictionary, the "Key" is whatever you want it to be. In this case, we're using names as Keys.

So "John" is the Key, and "Pepperoni" is the Value. The Value is the thing we want to know.

Calling a Value in a Dictionary is very similar to calling an item in a List, but instead of referencing a List index number, we reference the Dictionary Key (like John, or Tim, or whatever).

```
print(favorite_pizza["John"])
```

If this was a List, we'd do something like print(favorite_pizza[0])…but since this is a Dictionary, we put "John" as the key instead of "0". Make sense?

Dictionaries are really really cool because you can do most of the things you'd do with a List, but you can do it in a more human readable and human understandable way. Referencing 0 to see what John's favorite pizza is can be weird, but referencing "John" instead to see what Johns favorite pizza is makes more sense.

ADDING AND REMOVING ITEMS FROM A DICTIONARY

Just like a List, we want to be able to remove things and add things to our Dictionary.

Adding items is just as easy as declaring them:

```
CODE EXAMPLE 8.2
1.
2.  favorite_pizza = {
3.    "John":"Pepperoni",
4.    "Tim":"Mushroom",
5.    "Mary":"Cheese",
6.    "Beatrice":"Ham and Onion",
7.    "Bluto":"Supreme"
8.    }
9.
10.  favorite_pizza["Bob"] = "Tuna"
11.
```

The important line is line 10. That adds a new key ("Bob") and assigns the value "Tuna" to it. Can you put Tuna on a pizza? Wouldn't that be terrible? Well, Bob's a weird dude.

Deleting things is just as easy, we just use the pop function:

```
CODE EXAMPLE 8.3
1.
2.  favorite_pizza = {
3.    "John":"Pepperoni",
4.    "Tim":"Mushroom",
5.    "Mary":"Cheese",
6.    "Beatrice":"Ham and Onion",
7.    "Bluto":"Supreme"
8.    }
9.
10.  favorite_pizza.pop("Tim")
11.  print(favorite_pizza)
12.
```

Line 10 delete's the key "Tim" and it's value "Mushroom". We can confirm that by printing out our Dictionary in line 11.

Pretty simple!

DICTIONARY ODDS AND ENDS

So far we've been using strings as Keys and Values for our Dictionary, but you can use any other data types as well. You can use numbers, variables, Boolean, Lists, really just about anything you like!

CODE EXAMPLE 8.4

```
1.
2.  user_name = "John"
3.
4.  favorite_pizza = {
5.    user_name:"Pepperoni",
6.    "Tim":"Mushroom",
7.    "Mary":"Cheese",
8.    "Beatrice":"Ham and Onion",
9.    "Bluto":41
10.   }
11.
12.  print(favorite_pizza)
13.
```

In this example the first key is the variable user_name (which we created in line 2 by assigning "John" to the variable user_name). If we run this program we get this output:

```
{"John":"Pepperoni", "Tim":"Mushroom", "Mary":"Cheese",
"Beatrice":"Ham and Onion", "Bluto":41}
```

So even though we used a variable in our code (line 5), the value of the variable becomes the key.

Interestingly enough, if you want to access that item of the Dictionary, you can do it two different ways, either by calling it by the variable name (user_name), or the actual value that we assigned to that variable (John).

So you could call it like this:

```
print(favorite_pizza[user_name])
```

or you could also call it like this:

```
print(favorite_pizza["John"])
```

Either way will return the value "Pepperoni". I think that's pretty cool. But then, I've been slamming caffeine pretty hard all morning and I'd think just about anything is cool right now…

You'll also notice that Bluto's favorite pizza has been changed to 41, which is a number. You can do numbery things to it now.

```
print(favorite_pizza["Bluto"]+3)
```

…would print out 44 (41+3=44). Neat!

ALTERNATIVE WAYS TO WRITE A DICTIONARY

This is a silly point to make, but up until now we've been writing our Dictionaries on multiple lines. But I only do that to make them easier to read.

We don't really need to do that. I just like to write them like that because they're easier to read that way. You can write the same exact Dictionary all on one line if you prefer!

CODE EXAMPLE 8.5

```

favorite_pizza = {"John":"Pepperoni", "Tim":"Mushroom",
"Mary":"Cheese", "Beatrice":"Ham and Onion",
"Bluto":"Supreme"}

print(favorite_pizza)

```

That does the exact same thing as our earlier code, it's just all jumbled together.

I'm not sure why in the world you'd want to write code that was all jumbled up like that…but hey, go crazy if you feel like it.

FINDING THE KEYS OR VALUES OF A DICTIONARY

Up until now, we've just returned the values of a specific key in our dictionary. But you may want to return just the keys, or just the values. Python makes it easy to do both.

To return the keys of a Dictionary, just use the keys() function.

```
print(favorite_pizza.keys())
```

That would return: dict_keys(['John', 'Tim', 'Mary', 'Beatrice', 'Bluto'])
To return the values of a Dictionary, just use the values() function.

    print(favorite_pizza.values())

That would return:

dict_values(['Pepperoni', 'Mushroom', 'Cheese', 'Ham and Onion', 41])

You could easily loop through either of those using a for loop to get the actual items.

LOOPING DICTIONARIES TO GET KEYS AND VALUES

If you need to reference both the keys and values in a Dictionary, you can loop using the items() function.

CODE EXAMPLE 8.6

```

favorite_pizza = {
  "John":"Pepperoni",
  "Tim":"Mushroom",
  "Mary":"Cheese",
  "Beatrice":"Ham and Onion",
  "Bluto":"Supreme"
  }

for x,y in favorite_pizza.items():
  print(f"Key:{x}  Value:{y}")

```

This will print out the keys and values as x and y variables:

```
Key:John  Value:Pepperoni
Key:Tim  Value:Mushroom
Key:Mary  Value:Cheese
Key:Beatrice  Value:Ham and Onion
Key:Bluto  Value:Supreme
```

WHEN TO USE A DICTIONARY VS. A LIST

Dictionaries or Lists, which should you use and when? It's pretty straight forward…when you need to access items by an index number, use a List. For all other purposes, use a Dictionary.

To tell you the truth, there's probably times when I've used a Dictionary even though all my Key values where numbers. You can use numbers in a Dictionary if you want…so why not? We could just as easily have done this:

CODE EXAMPLE 8.7

```

favorite_pizza = {
  1:"Pepperoni",
  2:"Mushroom",
  3:"Cheese",
  4:"Ham and Onion",
  5:"Supreme"
  }

print(favorite_pizza)

```

I'm not really sure *why* you'd want to do that…but you absolutely could if you wanted to use a Dictionary instead of a List.

Just don't try to access the 0'th item of the Dictionary like you would a List because there isn't one in this code…unless we specifically named the first item 0!

Those are dictionaries! If you haven't figured it out yet…they're really useful!

CHAPTER EIGHT EXERCISES

1. Create your own Dictionary, with five of your friends' names and their phone numbers.

2. Create a program using your answer to exercise 1 that prompts a user to enter a name from your list your five friends, and then returns the phone number for that specific friend.

3. Add a feature to your code from exercise 2 that allows people to add or delete a name from the Dictionary, then return the updated Dictionary to the screen.

4. Create a loop that cycles through your Dictionary from exercise 1 and outputs (one per line) "My friend X's phone number is: xxx-xxx-xxxx" replacing the x's with the persons actual name and phone number.

CHAPTER NINE

PUTTING IT ALL TOGETHER TO MAKE A MATH FLASHCARD GAME

Congratulations! You've made it through the book (well mostly) and now you know Python (well mostly)!

Ok…you know the basics of Python…but the basics will take you pretty far. You've got a solid foundation of basic programming concepts and how they're done with Python.

We talked about Variables, Math, all kinds of Operators like Math Operators, Comparison Operators, Assignment Operators, and even some Logical Operators (And & Or).

You learned all about IF/ELSE and IF/ELIF statements, Loops, Lists, Dictionaries…even Functions…we really covered a lot of good stuff!

I hope you had a good time learning these things and I hope it wasn't too hard…it really wasn't too bad, was it?

Well now it's time to put everything we've learned together and have a little fun.

Right now I'm working on a new course over at **Codemy.com** that teaches you how to build a children's math flashcard web app. It's *pretty* cool. If you're interested in that I highly recommend you go join my Codemy.com site.

Membership is usually $198 for all the courses (or $49 per course if you take any of the over 50 courses individually) but sign up using the coupon code **pythonbook** and I'll give you complete access to all the courses (and all the new courses I create) for just $99 total.

But back to us…I thought it might be fun for us to build our own Python math flashcard game.

It won't be a web app (like the one I'm building at Codemy.com) but we can still have some fun banging out the code for just a basic math flashcard game and more importantly, it'll give us a chance to use nearly everything we've learned so far.

A MATH FLASHCARD GAME

Before you build any sort of program or project, it's a good idea to take a few minutes and map out what the program needs to do. I'm just talking broad sketch here.

So what does our program need to do?

Well…we want to build a simple math flashcard game. It should allow the user to choose to do Addition, Subtraction, Multiplication, or Division.

It should then randomly create a math problem and ask the user to solve it. Then it should allow the user to enter their answer and see whether or not the answer is correct or not.

Nothing too tricky here!

Before we get started though, there's one more thing you need to learn…Randomization.

RANDOMIZATION

We're going to need some way to create random numbers for our game. You'd be surprised how often you need to generate random numbers in everyday coding life. Luckily Python has an easy way to do it:

CODE EXAMPLE 9.1
```
1.
2.  import random
3.  print(random.random())
4.
```

First, notice line two. We've imported the random module that comes with Python. The random module does what you'd think…it generates random numbers in a variety of way.

Line 3 will generate a random number. It will return something that usually looks like 0.6768796878363911, which isn't super useful. Instead, let's define a range for our random number.

CODE EXAMPLE 9.2
```
1.
2.  import random
3.  our_rand = random.randint(1,10)
4.  print(our_rand)
5.
```

First off, you'll notice that I slapped a variable on there, so we can do stuff to that number we generate later if we want.

Second off, you'll notice that we used the randint() function and that it takes two arguments. The first is our starting integer, the second is the stopping integer. Randint() will return a random number between (and including) those two integers.

So this could return a one, or it could return a ten…or it could return any number between those two numbers.

What if you wanted to return a number between zero and ten? Then you'd use randint(0,10). Get it? Good.

So now we know how to generate Random numbers and slap them into a Variable so that we can play around with them.

So let's start building our game!

I highly recommend you try to build this game on your own without looking at my code. I'm going to suggest how to proceed with each section of the game, and I hope you'll take my suggestions and then write your own code.

Then come back and look at my code to see if I did it the same way. Remember, there's no right way to do this. Your code could look drastically different than mine. That ok! As long as it gets the job done, it doesn't really matter how you do it.

So let's get started…

I think we should break the game apart into different functions that we can call at various times throughout the game. Let's start with a start game function.

Basically, we need to greet the user and ask them what kind of flashcards they'd like to play (Addition, Subtraction, Multiplication, or Division).

This seems like an input() sort of thing, and also an IF/ELIF type of situation. Be sure to put in error handling too (in case the stupid user types in Bacon instead of Subtraction!).

CODE EXAMPLE 9.4

```
from os import system
# function To Start The Game And Pick Cards
def start_game():
 system("clear")#clear the screen
 print("Welcome to Math Flashcards!")
 pick = input(
  "Choose your flashcards (add|subtract|multiply|divide): ")

 if pick.lower() == "add":
  print(f"You picked {pick}")
 elif pick.lower() == "subtract":
  print(f"You picked {pick}")
 elif pick.lower() == "multiply":
  print(f"You picked {pick}")
 elif pick.lower() == "divide":
  print(f"You picked {pick}")
 else:
  print(f"Sorry, I don't recognize {pick}")
  input("please hit enter to try again")

  start_game()

 start_game()
```

This is how I chose to start things. Can you think of a simpler more elegant way? Notice in our IF/ELSE statements I just printed out the user selection. As we build out our functions, we'll replace those lines with calls to those functions.

Let's look through this code. I started out by importing os so we can clear the screen, and then adding a simple comment to describe what's going on there. Don't forget that commenting code is important, especially now that we're building more complicated programs.

Next I defined a function to start the game, called start_game(). You'll notice we aren't passing any variables into it. You'll also notice that I've called the function on line 24 (remember Functions don't run until you call them).

Next we've got some code to ask a person what sort of flashcards they want, then a simple IF/ELIF statement to figure out what to do next based on whatever the person typed in.

You'll notice that for each part of the IF statement, I've just printed out the user's selection. Don't worry, we'll add function calls there soon.

The only other thing to really look at is the last ELSE statement. That's our error handling. If the stupid user types in NACHOS instead of "add" then our program throws up a statement telling them that we don't recognize their response and asks them to hit enter to start over.

Notice the input() call on line 20? That allows the user to hit the "Enter" key but doesn't really do anything. It's just a way to pause the program while the user reads the message we printed to the screen telling them that we don't recognize their selection.

Once they hit Enter, the program continues on, and in line 22, starts the game over again.

So far so good!

MATH FUNCTIONS

Now we need to write the four different math Functions; one for addition, subtraction, multiplication, and division.

We need our program to randomly generate two numbers, and then ask the user to either add, subtract, multiply, or divide those two numbers together.

We need to capture their response, and check to see whether they answered correctly or not. If not, we could give them the chance to try again, but in this case I think we'll just tell them that they're wrong and show them the right answer.

Then we need to give them the option to quit, or get another flashcard. We should probably also give them the chance to switch flashcards completely, ie to switch from Addition flashcards to Subtraction flashcards (or whatever).

So let's start out with the addition flashcard function and then go from there…

Remember to add a call to our new add_flashcards() function to line 11 of our original code, 9.4. Just replace

```
11. print(f"You picked {pick}")
```

with:

```
11. add_flashcards()
```

CODE EXAMPLE 9.5

```
import random
# Start Addition Flashcards Function
def add_flashcards():
 system("clear")
 card_one = random.randint(0,10)
 card_two = random.randint(0,10)
 correct = card_one + card_two
 answer = input(f"{card_one} + {card_two}: ")

 if int(answer) == correct:
  print(f"Correct! {card_one} + {card_two} = {answer}")
 else:
  print(f"Wrong! {card_one} + {card_two} = {correct}")
 play = input("Would you like another card?
  (yes|no|restart): ")

 if play.lower() == "yes":
  add_flashcards()
 elif play.lower() == "no":
  print("Thanks for playing!")
 elif play.lower() == "restart":
  start_game()
 else:
  print(f"Sorry, I don't recognize {play}")
  input("please hit enter to try again. ")
  add_flashcards()
 # End Addition Flashcards Function
```

First things first; where does this code go? If you put this whole chunk of code BELOW our original start_game() function call, you'll get an error.

Why?

The answer has to do with flow control. If the start_game() Function is first, it will try to call the add_flashcards() function. The problem is; Python can't find that Function. Why? Because it hasn't gotten down to it yet.

Python starts at the top of a program and works its way down. Our add_flashcards() Function doesn't necessarily have to be above the start_game() Function, but it DOES have to be above the line of code where we first CALL the start_game() function (line 24 in our 9.4 example code).

I tend to put all my Functions first before we run any actual code to call a Function. That way Python runs over all our Functions, knows they exist, and feels good about the situation. Then later on when we call one of those Functions, everything works the way it's supposed to work.

So let's look at our add_flashcards() Function.

Most of it should be self-explanatory. Line 2 imports random so we can generate our numbers (You should move that up to the very top of your program).

Line 6 and 7 gives us our two random numbers and line 8 adds them together so that we know what the correct answer should be. You don't really need to explicitly create a variable with the correct answer, you can do it in the IF Statement, but I like to keep things explicit.

Line 9 is where our user types in their answer, it gets assigned to the answer variable.

After that we just have a series of IF/ELSE statements to check and see whether the answer given is correct or not and whether the user wants to keep playing or not.

Pretty simple, eh?

Now we can just copy and paste this function to create our other math functions. We'll just need to tweak it a bit so that it subtracts instead of adds; and multiplies or divides or whatever. But I think the basic foundation of the program is solid.

Wasn't that easy?

Could you have done that just a couple of days ago?

I hope you're starting to understand just how easy Python is…and just how much fun it is to use! Sure, this is a pretty simple little program, but all programs are simple when you really break them down.

If you understand these basics, you're well on your way to becoming an awesome Python programmer.

Ok enough blather, let's build our subtraction function:

CODE EXAMPLE 9.6

```

# Start Subtraction Flashcards Function
def subtract_flashcards():
 system("clear")
 card_one = random.randint(0,10)
 card_two = random.randint(0,10)
 correct = card_one - card_two
 answer = input(f"{card_one} - {card_two}: ")

 if int(answer) == correct:
  print(f"Correct! {card_one} - {card_two} = {answer}")
 else:
  print(f"Wrong! {card_one} - {card_two} = {correct}")

 play = input(f"Would you like another card?
  (yes|no|restart): ")

 if play.lower() == "yes":
  subtract_flashcards()
 elif play.lower() == "no":
  print("Thanks for playing!")

 elif play.lower() == "restart":
  start_game()
 else:
  print(f"Sorry, I don't recognize {play}")
  input("please hit enter to try again ")
  subtract_flashcards()

# End Subtraction Flashcards Function

```

All I did was go through the add_flashcards() code and replace the plus signs with subtraction signs. Let's do the same thing for the final two math functions and call this sucker done!

CODE EXAMPLE 9.7

```

# Start Multiplication Flashcards Function
def multiply_flashcards():
 system("clear")
 card_one = random.randint(0,10)
 card_two = random.randint(0,10)
 correct = card_one * card_two
 answer = input(f"{card_one} * {card_two}: ")

 if int(answer) == correct:
  print(f"Correct! {card_one} * {card_two} = {answer}")
 else:
  print(f"Wrong! {card_one} * {card_two} = {correct}")


 play = input(f"Would you like another card?
  (yes|no|restart): ")

 if play.lower() == "yes":
  multiply_flashcards()
 elif play.lower() == "no":
  print("Thanks for playing!")

 elif play.lower() == "restart":
  start_game()
 else:
  print(f"Sorry, I don't recognize {play}")
  input("please hit enter to try again ")

  multiply_flashcards()

 # End Multiplication Flashcards Function

```

Again, I didn't make any real changes to this code, I just changed all the addition signs to multiply. I also changed the name of the Function from add_flashcards to multiply_flashcards in a couple places throughout the code. So let's knock out the division code and call it a day!

```
CODE EXAMPLE 9.8
1.
2. # Start Division Flashcards Function
3. def divide_flashcards():
4.  system("clear")
5.  card_one = random.randint(0,10)
6.  card_two = random.randint(1,10)
7.  correct = card_one / card_two
8.  answer = input(f"{card_one} / {card_two}: ")
9.
10. if int(answer) == correct:
11.  print(f"Correct! {card_one} / {card_two} = {answer}")
12. else:
13.  print(f"Wrong! {card_one} / {card_two} = {correct}")
14.
15. play = input(f"Would you like another card?
16.  (yes|no|restart): ")
17.
18. if play.lower() == "yes":
19.  divide_flashcards()
20. elif play.lower() == "no":
21.  print("Thanks for playing!")
22. elif play.lower() == "restart":
23.  start_game()
24. else:
25.  print(f"Sorry, I don't recognize {play}")
26.  input("please hit enter to try again ")
27.  divide_flashcards()
28.
29. # End Division Flashcards Function
30.
```

I did need to make a minor change in this code. Check out line 6. I changed the range of numbers for our Random number. Why? Because technically a number can't be divided by zero; and if Python tries it will throw a big ugly error. So I just changed the range.

If you run this code, you'll see a few weird things. You'll see things like 5 divided by 3…and the answer is a decimal. Since we didn't program in floats or use the modulo (%), our program doesn't know how to deal with remainders.

You'll see things like 0 divided by 7…which is 0.

So, this isn't all that practical. But you can tinker with it and make it smarter if you like…in fact, I'll add that to the exercises at the end of this chapter.

MAKING IT SIMPLER

So we finished the flashcard game! But, when I add up all the lines of code for this thing, it ends up being like 140+ lines! Surely we can simplify things a bit.

One area I would look at for simplification is the math Functions. We have four math Functions (addition, subtraction, multiplication, division) but each of those functions is essentially the same big block of code (we just changed the name of the Function and the math operators).

Can you think of a way to reduce those four Functions into one?

There are many different ways to do it. You could use loops somehow…or make just one Function and pass a variable into it as the argument; the variable could be "add", "subtract", "multiply", or "divide" and then based on what that variable is, the Function could add, subtract, multiply, or divide.

I'll leave it to you, it's great practice to take some code and try to simplify it!

If you want to get the entire code for this program in an easy to copy and paste format, head over to codemy.com/python and sign up for the video course that goes along with this book.

It's usually $49 but as a thank you reading this book, you can get the course for $9…just use coupon code **python9** at checkout.

Everyone who signs up also gets a pdf copy of this book and you can copy and paste the code right out of there.

I've also listed a pdf of just this program so you can easily see all the code in one place.

Plus, if you have any questions about the code, you can post them there and I'll answer them personally.

CHAPTER NINE EXERCISES

1. Re-Create the flashcards game to keep track of how many questions the user answered right and wrong.

2. Re-Create the flashcards game with questions about fractions.

3. Re-Create the flashcards game a different way using less code (make it more elegant Eugene!)

4. Re-Create the flashcards game using While statements to check for correct answers in each of our Math functions (instead of the If statements we're using now).

5. Re-Create the flashcards game, but modify the subtraction function to make sure card_one is greater than card_two (so the answer can't be negative!)

6. Re-Create the flashcards game but clean up the division function so that it makes more sense. Put in logic that keeps picking random numbers until you get one that is divisible by the other one (otherwise it's going to ask you to divide 3 by 9, which is a decimal or other dumb things. Floats? A larger range of Random Numbers? Go crazy!

CHAPTER TEN

CONCLUSION

You made it! We're done! How 'bout them apples?

What'd you think? Python is pretty freaking easy – right? Yes it is!

I hope you enjoyed the book, I really enjoyed writing it and head over to Codemy.com/python-follow and sign up for my discounted Python course (use coupon code **python9**). You can ask me questions directly if you got stuck anywhere along the way.

PYTHON ON THE WEB?

Throughout this book we've been playing around with Python in the terminal, but we haven't done any web development work with it.

That's because Python alone can't easily do web stuff. You need a framework to go along with it…

That’s where Django or Flask comes in.

If you're interested in Django or Flask, I hope you'll checkout out my website Codemy.com

I've got a bunch of Django courses and a couple Flask courses as well.

Django is a more robust, complete web framework; whereas Flask is a lighter weight framework that gives you a little more hands-on control over everything.

Both are great, and both have their place in your coding arsenal!

Courses at Codemy.com are usually $49 each, but you can also sign up for "Total Membership" that gives you access to all the courses for one low fee.

At the time of this writing, there are over 50 courses on the website, with more being added all the time.

Total Membership usually costs $249 but if you use coupon code **python50** I'll knock the price in half and you pay just $99.

And remember, you get all the current courses, but you also get all future courses at no additional cost. They just show up in your account whenever I release them!

And that's a one time fee…it's not monthly…it's not yearly…you'll pay nothing more for lifetime access to the courses.

CAN I ASK YOU A HUGE FAVOR?

Book reviews at Amazon.com are a HUGE deal for small writers like me. Just a few reviews can mean the difference between a book getting ranked well by Amazon's algorithm, and complete oblivion. Without reviews, the book won't even show up when someone searches for Python at Amazon.

I'd consider it a great favor if you would head back to Amazon.com and leave a quick review of this book. It doesn't have to be long-winded, just a few words will make a big big difference as to whether my book gets out there or not.

So if you enjoyed the book, got a lot out of it...heck even if you didn't like it, please head back to Amazon and leave a review. Then shoot me an email and let me know so I can thank you properly!

Just go to **Codemy.com/pythonreview**

And that URL will re-direct to this book's Amazon page where you can leave a quick review.

I really appreciate it!!

Thanks!
-John Elder
Codemy.com
john@Codemy.com

APPENDIX A

SPECIAL CODEMY.COM OFFER

Learning never stops, especially for coders. There's always something new and cool to learn. I've tried to build a website that makes it super easy to learn how to code, and learn new coding skills…and it's called Codemy.com

Each course at Codemy.com is a series of videos where you watch over my shoulder as I build something.

In one course I build a Django Math Flashcard App. In another course I build a clone of the website Pinterest with Ruby on Rails. In another course I build an affiliate marketing site that makes money from Amazon affiliate products.

I teach Rails courses, PHP courses, HTML and CSS courses, Database courses, GUI app courses with Tkinter, PyQT5, Kivy, and more.

Each course costs $49, or you can sign up for all the courses for $198 (which is a pretty good deal if you ask me!) and that entitles you to all the future courses that I add absolutely free (and I've got some cool courses on the horizon).

AS A SPECIAL THANK YOU FOR READING THIS BOOK…

I'd love to see you over at Codemy.com and I'd like to bribe you to join today; so I'm giving you a special coupon code (**python50)** that will give you half off membership (so you pay just $99 instead of $249)…

It's my gift to you! **https://www.Codemy.com**

So you get access to ALL my best-selling courses for just **$99** instead of the regular $249.

And I offer a month-long 100% money back guarantee. Check out the site, if it isn't for you…just shoot me a message and I'll immediately refund your money, no questions asked, no hoops to jump through.

HANDS ON HELP

Membership doesn't just get you videos…you also get hands on help from me and other members. Any time you get stuck with something from a video, you can post a question to me directly, or post a question directly under the video itself.

It's a great resource and I hope you'll take advantage of it.

Just use coupon code **python50** at checkout for the special $99 price.
http://www.Codemy.com

See you on the inside!

-John Elder
Codemy.com

THE END

NOTES

NOTES

NOTES

NOTES

NOTES