Zip Code Wilmington's Programming in Python

Kristofer Younger

Version 1.1.5, 2023-03-16

Table of Contents

Colophon

Zip Code Wilmington's Programming in Python by Kristofer Younger

Cover Design: Janelle Bowman

Published in the U.S.A.

May 2021: First Edition

Preface

This book is a cousin of the same text for JavaScript. Many of the Java programmers we've trained at ZipCodeWilmington look at Python as a possible second 'main' language, one that'd be good to pick up and have on the resume. I'd agree. Python has been good to me, as I first learned it when I got back into coding about 2007, I've written a **lot** of python, and consider it an old friend. I especially wish to thank Mikaila Akeredolu: without his brilliant javascript prep session slides as the starting point for the javascript version of this book, I would never have thought a small book on the basic fundamentals of programming would be possible or even useful.

Zip Code Wilmington is a non-profit coding boot-camp in Delaware for people who wish to change their lives by becoming proficient at coding. Find out more about us at https://zipcodewilmington.com

About this book

This book's aim is to provide you with the most basic of fundamentals regarding Python, one of the most popular programming languages available. It springs from the Javascript preparation sessions we often give prospective Zip Code applicants on how to do well on the Zip Code application coding assessment. To someone who has spent some time with programming languages, this might be just a breezy intro to Python. If you have almost any serious coding experience, this book is probably too elementary for you. You might, however, find the ideas in the Appendices interesting. There I've added some material that have a few advanced ideas in them, plus there is a full code listing of a Mars Lander simulation.

You may be aware that Zip Code has a *data engineering* course that teaches Python, and while the first book was JavaScript, this one is, well…

Python: Easy to Understand

Python is an easy programming language to learn, and we're going to use it in this book to describe the basic fundamentals of coding. Its form is modern and it is widely used. It can be used for many purposes, from web applications, machine learning, data science to back-end server processing. We're going to use it for learning simple, programming-in-the-small, method-functions and concepts; but make no mistake - Python is world-class language capable of amazing things.

Simple to Use

Python also has the advantage that one can learn to use it without a lot of setup on a computer. In fact, almost every computer you have access to can run Python. You can run almost every code snippet in the book and *see* what the code does.

Focuses on coding

Finally, because in this book all we aim to teach you is "programming in the small", Python is great for that. Many of the examples here are significantly less that 20 lines in length. We want you to get better at looking at small blocks of code to see how they work. These smaller examples and concepts are a core building block as you become proficient in coding.

You'll learn it eventually

The truth is, in today's coding world, all of us eventually learn to do things with Python. (I've said ths about JavaScript too.) So, start early, get comfortable with it, and then go on and study other computer languages like Java or JavaScript. Python will always be there, waiting patiently for you to return. And hey, if you've already learned Java, you might find this book helpful - bridging languages is a great skill for a developer to have.

Coding *The Hard Way*.

Zed A. Shaw is a popular author of several books where he describes learning a programming language *The Hard Way*. Zed suggests, and we at Zip Code agree with him whole-heartedly, that the best, most impactful, highest return for your investment when learning to code, is **type the code using your own fingers** [1]

That's right. Whether you are a "visual learner", a "video learner", or someone who can read textbooks like novels (are there any more of these out there?), the best way to learn to code is **to code** and **to code by typing out the code with your own fingers**. This means you DO NOT do a lot of copy and paste of code blocks; you really put in the work, making your brain better wired to code by **coding with your own typing of the code.**

You're here, reading this, because you're thinking (or maybe you

know) that you want to become a coder. It's pretty straightforward.

You may have heard a friend wistfully dream of making a career at writing. "Oh," they say, "I wish I had time to write a great novel, I want to be a writer someday".

So you can ask them: Did you write *today*? *How many words?* And the excuses flow: "Oh, I have to pick up the kids" "Ran out of time, I'm so busy at work." "I had to cut the grass" and so on. Well, I'm here to tell you that all the excuses in the world don't stop a real writer from writing. They just sit down and do it. As often as they can, sometimes even when they can't (or shouldn't).

Coding, like writing, isn't something you can do when all your other chores, obligations, and entertainments are done. If you're serious about learning coding, you must make time for coding.

Watching hours of YouTube videos *will not* make you a coder.

Reading dozens of blog posts, Medium articles, and books *will not* make you a coder.

Following along with endless step-by-step tutorials *will not* make you a coder.

The only way you're going to learn to code is by doing it. Trying to solve a problem. Making mistakes, fixing them, learning from what worked and what didn't at the keyboard.

Many have heard my often-repeated admonition: **If you coded today, you're a coder. If not, you're not a coder.** It really is as simple as that.

Dedication to the mission

I happen to be among those who feel anyone can learn to code. It's a 21st century superpower. When you code, you can change

the world. Being proficient at coding can be a life-changing skill that impacts your life, your family's life and your future forever. Time and time again, I've seen that the ability to learn to code is evenly distributed across the population, but the *opportunity to learn to code is not*. So, we run Zip Code to give people a shot at learning a 21st century superpower, no matter where you come from.

And fortune favors the prepared. Some day, you may be working at a great company, making a decent living, working with professionals in a great technical job. Your friends may say "You are so lucky!"

And you will think: **Nope. It wasn't luck.** You'll know that truly. You got there by preparing yourself to get there, and by working to get there, working *very hard*. Ain't no luck involved, just hard work. You make your own luck by working hard.

As many know, getting a spot in a Zip Code cohort is a hard thing to do. Many try but only a few manage it. I often get asked "what can I do to prepare to get into Zip Code?"

The best way is to start solving coding problems on sites like https://hackerrank.com - HackerRank (among others) has many programming assignments, from extremely simple to very advanced. You login, and just do exercise after exercise, relieving you of one of the hardest of coding frustrations, that of trying to figure out **what** to code. Solving programming assignments is a good way to start to cultivate a coding mindset. Such a mindset is based on your ability to pay very close attention to detail, a desire to continually learn, and being able to stay focused on problem solving even if it takes a lot of grit and dedication.

Spending even 20 minutes a day, making progress on a programming task can make all the difference. Day after day your skills will grow, and before long you'll look back on the early things you did and be astonished as to how simple the

assignments were. You may even experience embarrassment at remembering how hard these simple exercises seemed at the time you did them. (It's okay, we've all felt it. It's part of the gig.)

Working on code every day makes you a coder. And coding everyday will help with your ability to eventually score high enough on the Zip Code admissions assessment that you get asked to group and potentially final interviews. And then, well, then you get to learn Java or Python and work yourself to exhaustion doing so. Lots and lots more hours.

Why?

You do that hard work, you put in those hours, you create lots of great code, you'll make your own luck, and someone will be impressed and they will offer you a job. And that is the point, right? A job, doing what you love, coding. Right? RIGHT?

You're Welcome,

-Kristofer

Ready?

Okay, let's go.

[1] check out his terrific work: https://learncodethehardway.org

Chapter 1. Output print()

Let's start with a really simple program. Perhaps the simplest Python program is:

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

This program just prints "Hello, World!". [1]

Printing goes to "standard output", a place Python uses to communicate with a user (in this case, us, the programmers.)

(And if you haven't done it yet, go to https://code.zipcode.rocks and make a browser bookmark for yourself. Once that's done, you can use that REPL [2] as a place where you can type in the code from this book and run it to see what each code snippet does. We run `python3` in this book. Don't use python version 2. `python3` is what the pros use.)

We will use `print` to do a lot in the coming pages.

Here's your second Python program:

```
milesPerHour = 55.0
print("Car's Speed: ", milesPerHour)
```

If you typed into your REPL and clicked the "Run" button, you should have seen this as your output:

```
Car's Speed: 55
```

as the program's output.

Cool, huh? The ability to communicate with you is one of Python's

most fundamental capabilities. And you've run two Python programs. Congratulations, you're a coder. (Well, at least for today you are.)

[1] And while you might *not yet* understand this *technical description*, it is a program of one *line* of code, which says "call the 'print' function using the string "Hello, World!" as the argument to be sent to output."

[2] a REPL is short for "read-evaluate-print loop", a special kind of computer program that lets you run code of a given language.

Chapter 2. Comments

While you're not thinking about the long term, or about large Python programs, there is a powerful thing in Python that helps with tracking comments and notes about the code.

In your program, you can write stuff that Python will ignore, it's just there for you (or readers of your code). We use a 'hash character' to start a comment, and the comment goes to the end of the line. Python will ignore anything on a line after a hash character. "#"

```
# this is a comment. it might describe something in the code.
print('Hello')

print('World') # this is also a comment.
```

Often, you'll see something like this in this book.

```
flourAmount = 3.5;

print(flourAmount); # -> 3.5
```

That comment at the end of the print line is showing what you can expect to see in the output. Here it would be "3.5" printed by itself. Try it in your bookmarked Repl.

We can also add useful stuff to the .log call.

```
flourAmount = 3.5;

print("We need", flourAmount, "cups of flour."); # -> We need 3.5
cups of flour.
```

See how Python types it all out as a useful phrase? That proves to

be very handy in a million-plus (or more) cases.

Comments can be used to explain tricky bits of code, or describe what you should see in output. Comments are your friend.

Chapter 3. Statements and Expressions

In Python, there are parts of a program and different parts have different names. Two of the most basic (and fundamental) are **statements** and **expressions**.

3.1. Expressions

An **expression** is something that needs to be computed to find out the answer. Here are a few simple ones.

```
2 + 2 * 65536

speed > 55.0

regularPrice * (1.0 - salePercentOff)
```

Each of these lines is something we'd like Python to **compute** for us. That computation is often referred to as "evaluation" or "evaluate the expression" to get to the answer. There are two kinds of expressions in Python, *arithmetic expressions* and *boolean expressions*.

Arithmetic expressions are, as their name implies, something that require arithmetic to get the answer. An expression like "5 + 8 - 3" gets *evaluated* to 10.

Boolean expressions result in either a True or a False value. Example: "maxSpeed > 500.0" - this is either true or false depending on the value of maxSpeed.

3.2. Statements

A **statement** is just a line of Python.

```
# at the Grocery

salesTaxRate = 0.06

totalGroceries = 38.99
salesTax = totalGroceries * salesTaxRate

chargeToCard = totalGroceries + salesTax
```

And this is what a Python program looks like. It's just a list of statements, one after the other, that get computed from the top down.

Some of the statements have expressions in them (like totalGroceries * salesTaxRate), while some are just simple **assignment** statements (like totalGroceries = 38.99, where we assign the variable 'totalGroceries' the value 38.99). Don't panic. These are just some simple examples of Python to give you a feel for it. We'll go thru each of these kinds of things slowly in sections ahead.

3.3. Multi-line Statements

In this book, you may see that the code used in examples is longer than can fit on one line in the code boxes. In Python, we have to add a line "continuation" character. So to be clear, a statement with long variable names is the same as one with a short name, but you may have to add "\" (backslashes) to make Python happy.

```
k = h * kph - (rest / 60)

kilometersCycled = numberOfHoursPedalled \
    * kilometersPerHour \
    - (totalMinutesOfRest / 60)
```

When you come across code that goes onto multiple lines, do like

Python does, look for the backslash, when you find it, think of the line being spread out.

3.4. Block Statement & Indentation

Very often in Python, we will see a **block** of statements. It is a list of statements *indented* the same amount. (Yes, unlike many languages, in Python, *white space is important*.) It acts like a container to make clear what statements are included in the block.

It is *so important* that indentation, and making sure you have everything aligned properly, is one of the most important thing to learn about Python when you are a beginner. Many a beginner discovers that the braces *{ }* that other languages use to manage the way code is grouped together are missing from Python. Java, JavaScript, and many other languages use braces to handle how code gets grouped together.

Not Python, that's why is *SO* very important to get it right.

```
if (magePower > 120.0):
    maxMagic = 500.0
    lifeSpan = 800.0
    maxWeapons = magePower / maxPowerPerWeapon
    if (maxWeapons < 150):
        print('You have too many weapons!')
    else:
        backpack.load()

# some more code
print(magePower, "is your Mage's Power rating.")
```

See those SPACES (and confusingly, they might be TABs but you cannot tell that by looking at the line). All the indented lines below the IF statement are part of what gets run when the IF is True. Then you also see a nested IF below that, one that has an *else* statement. But the key thing is to look at the lines of code and

notice how they are *indented*, because that tells what you need to know about how the program works, and what gets done depending on the states of the variables. (Don't worry, we will be going into all this a lot more carefully later.)

Editing tools that let you work on Python do their best make the indentations are correct, but it is a common problem for beginner Python programmers to think the indentation isn't important and spend far too much time tracking down silly indentation errors.

So, the wise Python beginner spends a fair amount of time making sure the details are exact on indentation. Making sure code is indented correctly is part of being a Pythonista.

Indentation is rejected as inconsistent if the code mixes tabs and spaces it gets flagged as an error in the code; a TabError is raised in that case.

Chapter 4. Variables and Data Types

4.1. Variables

In Python, variables are containers used to store data while the program is running. Variables can be named just about anything, and often are named things that make you think about what you store there.

Think of them as little boxes you can put data into, to keep it there and not lose it.

There are some rules about variables.

- All Variables must be named.
- Names can contain letters, digits, and underscores
- Names must begin with a letter
- Names are case sensitive (y and Y are different variables)
- Reserved words (like Python keywords) cannot be used as names
- All variable names must be unique (no two alike)

So this means the following are fine for variable names in Python:

```
x
AREA
height
Width
currentScore
playerOne
playerTwo
sumOfCredits
_lastPlay
```

```
isPlayerAlive
```

And uppercase and lowercase letters are different. So each of these are DIFFERENT variables even though they are based on the same word(s).

```
Current_Speed
current_speed
CURRENT_SPEED
```

So be careful with UPPERCASE and lowercase letters in variable names.

4.1.1. Declaring a Variable

We can declare a variable named 'cats' and assign it the value 3:

```
cats = 3
print(cats)  # -> 3

dogs = 1
print(cats)  # -> 1

player1alive = true
carspeed = 55.6
```

4.1.2. Assign a value to a variable

Variables are meant to be used and re-used throughout a program. You can *set* a variable by *assigning* it a new value. See how we use the *age* and *name* variables twice, for both James and Gina.

```
age = 19
name = "James"
```

```
print(name, "is", age, "years old") # -> James is 19 years old
age = 21
name = "Gina"
print(name, "is", age, "years old") # -> Gina is 21 years old
```

4.1.3. Reassigning a value to a variable

```
x = "five"
print(x) # -> five
x = "nineteen"
print(x) # -> nineteen
```

Notice how we assign "nineteen" to *x*. We can assign (re-assign) to a variable as many times as we might need to.

```
age = 3
# have a birthday
age = age + 1
# have another birthday
age = age + 1
print(age); # -> 5
```

Notice here how *age's* current value is used, added one to it, and re-assigned **back into the variable** *age*.

Now, one of the weird (to me anyway) things Python can do is change the type of a variable while the program is running. A lot of languages won't you do this. But it can be handy in Python. In Python, variables are dynamic (can contain any data) which means a variable can be a string and later be a number.

```
height = 62.0 # inches maybe?
print(height) # -> 62

height = "very tall"
print(height) # -> very tall
# yep, first height is a number
```

```
# and then it's a string.
```

You can't see it, but I am slowly shaking my head in disbelief. Some day, maybe I'll explain why.

4.2. Constants

Constants are like variables but they contain values that SHOULD NOT change such as a person's date of birth. Convention is to capitalize constant variable names.

```
DATE_OF_BIRTH = "04-02-2005"
PI = 3.14
GRAVITY = 9.8
```

They are *usually* declared in a file named 'constants.py'.

Note: In reality, we don't use constants in Python. (The language *doesn't* have them.) Naming them in all capital letters is a convention to separate them from variables, however, it does not actually prevent reassignment. So be careful.

4.3. Data Types

Python can keep track of a number of different kinds of data, and these are known as "data types". Here are a few of them.

- **Numeric** - there are two kinds of numbers: integers, floats and complex
  - **Integers** are like 0, -4, 5, 6, 1234
  - **Floats** are numbers where the decimals matter like 0.005, 1.7, 3.14159, -1600300.4329
  - **Complex** are, well, if you don't already know, let's ignore them.

- **str** - a *string* of characters -
  - like 'text' or "Hello, World!"
- **Boolean** - is either **True** or **False**
  - often used to decide things like isPlayer(1).alive() [True or False?]
- **None** - no value at all (nil or null in other languages)

It is common for a computer language to want to know if data is a bunch numbers or text. Tracking what *type* a piece of data is is very important. And it is the programmer's job to make sure all the data get handled in the right ways.

So Python has a few fundamental **data types** that it can handle. And we will cover each one in turn.

Create variables for each primitive data type:

- boolean,
- float,
- integer,
- str

Store a value in each.

```
# Here are some samples.

# integer
x = 0

# boolean
playerOneAlive = True

# float
currentSpeed = 55.0

# string
```

```
playerOneName = 'Rocco'
```

Now, you try it. Write down a variable name and assign a normal value to it.

4.4. Data Structures

Python has a series of different *data structures* built into the language. Its built-in data structures include lists, tuples, sets, and dictionaries.

Chapter 5. Arithmetic Operators

Python can do math. And many early programming problems you will come across deal with doing fairly easy math. There are ways to do lots of useful things with numbers.

5.1. Basics

Operator	Name	Description
+	Addition	Add two values
-	Subtraction	Subtract one from another
*	Multiplication	Multiply 2 values
/	Division	Divide 2 numbers
%	Modulus	returns the remainder

Say we needed to multiply two numbers. Maybe 2 times 3. Now we could easily write a program that printed that result.

```
print(2 * 3)
```

And that will print 6 on the console. But maybe we'd like to make it a little more complete.

```
a = 2
b = 3
# Multiply a times b
answer = a * b
print(answer) # -> 6
```

Using this as a model, how would you write programs to solve

these problems?

- Lab 1: Subtract A from B and print the result
- Lab 2: Divide A by B and print the result
- Lab 3: Use an operator to increase A by 1 and print result

```
a = 9
b = 3

L1 = b - a
L2 = a / b
L3 = a + 1
print(L1) # -> -6
print(L2) # -> 3
print(L3) # -> 10
```

5.2. Division and Remainder

We know that we can do regular division. If have a simple program like this, we know what to expect:

```
a = 6 / 3 # -> 2
a = 12 / 3 # -> 4
a = 15 / 3 # -> 5
a = 10 / 4 # -> 2.5
```

But sometimes, we have a need to know what the remainder of a division is. The remainder operator %, despite its appearance, is not related to percents.

The result of a % b is the remainder of the integer division of a by b.

```
print( 5 % 2 ) # 1, a remainder of 5 divided by 2
print( 8 % 3 ) # 2, a remainder of 8 divided by 3
```

Now what's this about '%' (the remainder) operator?

```
a = 3
b = 2
# Modulus (Remainder)
answer = a % b
print(answer) # -> 1
```

```
a = 19
b = 4
# Remainder
answer = a % b
print(answer) # -> 3
```

5.3. Order is Important

A strange thing about these operators is the order in which they are evaluated. Let's take a look at this expression.

```
6 × 2 + 30
```

We can do this one of two ways:

- Say we like to do multiplication *(I know, who is that?)*
  - we would then do the "6 times 2" part first, giving us 12.
  - then we'd add the 30 to 12 giving us 42 [1]
- But say we don't like multiplication, and want to save it for later...
  - we would add 2+30 first, giving us 32

- and then we multiply it by 6, and, whoa, we get 192!

Wait! Which is right? How can the answers be so different, depending on the order we do the math in? Well, this shows us that the Order of Operations is important. And people have decided upon that order so that this kind of confusion goes away.

Basically, multiply and divide are higher priority than add and subtract. And exponents (powers) are highest priority of all.

There is a simple way to remember this.

5.3.1. P.E.M.D.A.S

Use this phrase to memorize the default order of operations in Python.

Please Excuse My Dear Aunt Sally

- Parenthesis ( )
- Exponents 2^3
- Multiplication * and Division /
- Addition + and Subtraction -

Divide and Multiply rank equally (and go left to right) So, if we have "6 * 3 / 2", we would multiply first and then divide. "6 * 3 / 2" is 9

Add and Subtract rank equally (and go left to right) So if we have "9 - 6 + 5", we subtract first and then add. "9 - 6 + 5" is 8

30 + 6 × 2 How should this be solved?

> Right way to solve 30 + 6 × 2 is first multiply, 6 × 2 = 12, then add 30 + 12 = 42

This is because the multiplication is *higher priority* than the addition, *even though the addition is before the multiplication* in the expression. Let's check it in Python:

```
result = 30 + 6 * 2
print(result)
```

This gives us 42.

Now there is another way to force Python to do things "out of order" with parenthesis.

(30 + 6) × 2

What happens now?

```
result = (30 + 6) * 2
print(result)
```

What's going to happen? Will the answer be 42 or 72?

5.4. Python math Object

There is a useful thing in Python called the `math` object which allows you to perform mathematical tasks on numbers.

To make these work, you need to `import` the `math` module.

```
import math

math.pi # returns 3.141592653589793
math.ceil(4.7)    # returns 5
```

```
math.floor(4.4)    # returns 4
x = 5
y = 3
math.pow(x, y) # the value of x to the power of y - x^y^
math.pow(8, 2)      # returns 64.0
math.sqrt(x) # returns the square root of x - 2.2360...
math.sqrt(64)      # returns 8
```

What does "returns" mean?

When we ask a 'function' like sqrt to do some work for us, we have to code something like:

```
squareRootTwo = math.sqrt(2.0)
print(squareRootTwo)
```

We will get "1.4142135623730951" twice in the output. That number (squareRootTwo) is the square root of 2, and it is the result of the function and *what the function sqrt "returns"'*.

math.pow() Example

Say we need to compute "6^2 + 5"

```
result = math.pow(6,2) + 5
print(result)
```

What will the answer be? 279936 or 41?

How did Python solve it?

Well, 6^2 is the same as 6 * 6. And 6 * 6 = 36, then add 36 + 5 = 41.

```
squareRootTwo = math.pow(2, 0.5)
```

```
print(squareRootTwo)
```

And notice, when we raise a number to the *0.5* power, it's **the same as taking its square root!**. And that can be handy sometimes.

You'll learn a lot more about working with numbers in your career as a coder. This is really just the very basics of the very beginnings.

[1] The answer to life, the universe and Everything.

Chapter 6. Algebraic Equations

Some of the most fundamental of computer programs have been ones that take the drudgery of doing math by a person, and making the computer do the math. These kinds of *computations* rely on the fact that the computer won't do the wrong thing if it's programed carefully.

Given a simple math equation like:

a = b3 - 6 and if b equals 3, then a equals ?

In math class, your teacher would have said "How do we solve for a?" The best way to solve for `a = b3 - 6` is to

- figure out what b times 3 is (well, if b equals 3, then 3 times 3 is 9)
- subtract 6 from b times 3 (and then 9 minus 6 is 3)

```
# And in Python:
# a = b3 - 6

b = 3
a = b * 3 - 6
print(a) # -> 3
```

Now you try it.

Solve the equation with Python...

q = 2j + 20

if j = 5, then q = ?

Take a moment and write down your solution before reading on.

```
q = 0
j = 5
q = 2 * j + 20
print(q) # -> 30
```

Let's try another...

Solve the equation with Python...

$x = 5y + y^3 - 7$

if y=2, x = ?

and print out x.

My solution is pretty simple.

```
import math # if you haven't already!

y = 2
x = 5 * y + math.pow(y, 3) - 7
print(x) # -> 11
```

6.1. *Trigonometry*

The word trigonometry comes from the Greek words, trigonon ("triangle") + metron ("measure"). We use trigonometry to find angles, distances and areas.

For example, if we wanted to find the area of a triangular piece of land, we could use the equation **AreaOfaTriangle** = **height * base / 2**

Therefore we just need to create variables for each and use the operators to calculate the area.

Calculate Area of a Triangle in Python Height is 20 Base is 10 Formula: A = h * b / 2

```
height = 20
base = 10
areaOfaTriangle  =  height * base / 2
print(areaOfaTriangle) # -> 100
```

Calculate the area of a circle whose radius is 7.7 Formula: area = Pi * radius^2

Hint: You'll need to use a constant Math property!

Here is a possible solution to the calculation.

```
radius = 7.7
area  =  math.pi * math.pow(radius, 2)
print(area) # -> 186.26502843133886 (wow)
```

See how we used *math.pi* to handle the equation?

Chapter 7. Simple Calculation Programs

7.1. How far can we go in the car?

Let's create a simple problem to solve with Python.

> Our car's gas tank can hold 12.0 gallons of gas. It gets 22.0 miles per gallon (mpg) when driving at 55.0 miles per hour (mph). If we start with the tank full and carefully drive at 55.0 mph, how many miles can we drive (total miles driven) using the whole tank of gas?
>
> BONUS:
>
> How long will it take us to drive all those miles?

What do we need figure out? We need a variable for our result: totalMilesDriven. So we start our program this way...

```
totalMilesDriven = 0

# print result of computation
print(totalMilesDriven)
```

It's often good to start with a very simple template. If we run that, we will see 0 (zero) as the result, right?

Next step, let's add the variables we know.

```
totalMilesDriven = 0
totalHoursTaken = 0

totalGasGallons = 12.0
```

```
milesPerGallon = 22.0
milesPerHour = 55.0

# print result of computation
print(totalMilesDriven)
```

Okay, good. We've added all the stuff we know about the computation. Well, except the part of the *actual computation*.

You probably know that if you multiply the milesPerGallon by the totalGasGallons, that will give you totalMilesDriven. And if you divide the totalMilesDriven by the milesPerHour, you will get the totalHoursTaken.

So let's add those as Python statements.

```
totalMilesDriven = 0
totalHoursTaken = 0

totalGasGallons = 12.0
milesPerGallon = 22.0
milesPerHour = 55.0

totalMilesDriven = milesPerGallon * totalGasGallons
totalHoursTaken = totalMilesDriven / milesPerHour

# print result of computation
print(totalMilesDriven, totalHoursTaken)
```

We get as a result 264 miles driven in 4.8 hours. And that's how a simple Python program can get written.

Let's do another.

7.2. The Cost of a "Free" Cat

A friend of ours is offering you a "free cat". You're not allergic to cats but before you say yes, you want to know how much it'll cost

to feed the cat for a year (and then, approximately how much much each month).

> We find out that cat food costs $2 for 3 cans. Each can will feed the cat for 1 day. (Half the can in the morning, the rest in the evening.) We know there are 365 days in a year. We also know that there are 12 months in the year. So how much will it cost to feed the cat for a year?

Looking at it, this may be quite simple. If we know each can feeds the cat for a day, we then know that we need 365 cans of food. So we can describe that as

```
totalCost = 0
cansNeeded = 365
costPerCan = 2.0 / 3.0

totalCost = cansNeeded * costPerCan # right?

monthsPerYear = 12
costPerMonth = totalCost / monthsPerYear
# print result of computation
print(totalCost, costPerMonth)
```

What's going to be the answer? [1] Run it in your Repl window to work it all out.

And let's do one more.

7.3. You Used Too Much Data!

A cell phone company charges a monthly rate of $12.95 and $1.00 a gigabyte of data. The bill for this month is $21.20. How many gigabytes of data did we use? Again, let's use a simple template to get started.

```
dataUsed = 0.0

# print result of computation
print("total data used (GB)", dataUsed)
```

Let's add what we know: that the monthly base charge (for calls, and so on) is $12.95 and that data costs 1 dollar per gigabyte. We also know the monthly bill is $21.20. Let's get all that written down.

```
dataUsed = 0.0
costPerGB = 1.0
monthlyRate = 12.95

thisBill = 21.20

# print result of computation
print("total data used (GB)", dataUsed)
```

Now we're ready to do the computation. If we subtract the monthlyRate from thisBill, we get the total cost of data. Then, if we divide the total cost of data by the cost per gigabyte, we will get the dataUsed.

```
dataUsed = 0.0
costPerGB = 1.0
monthlyRate = 12.95

thisBill = 21.20
totalDataCost = thisBill - monthlyRate

dataUsed = totalDataCost / costPerGB

# print result of computation
print("total data used (GB)", dataUsed)
```

How many GBs of data did we use? Turns out to be 8.25 gigabytes.

Now if the bill was $24.00? How many GBs then? (go ahead, I'll wait…) [2]

[1] totalCost will be $243.33 and $20.28 per month.

[2] total data used (GB) 11.05

Chapter 8. Boolean Expressions

When starting out in programming, the idea of a boolean variable, something that is either just true or false, seems like an overly simple thing… something that feels rather useless.

In fact, booleans in computer code are **everywhere**. They are simple, but also useful in many ways. You've probably heard about how everything in computers is ones and zeros at the lowest level - and that's true. But on this super simple base of `0 and 1` is built all of the power of the internet, and all the apps you've ever used.

When you are coding, you often have to make a choice about what to do next based on some kind of condition or to do something repetitively (over and over) based on some condition. Something like `is there gas in the car?` or `are we moving faster than 100mph?` In real life, these are considered to be YES or NO kinds of questions. If the gas tank is empty, the question results in a FALSE condition. If there is some gas in the tank, then the question's result is TRUE, "yes, there is some gas in the tank."

And while this may seem super simple to you, and it is, it is also very powerful when used in a program.

This idea of a condition that is either TRUE or FALSE, is known as a **boolean expression**. And in Python, they crop up everywhere. They are in *conditional statements* and they are part of *loops*.

Boolean expressions can be very complex, or very simple:

```
playerOne.isAlive() == true
```

might be a key thing to know inside of a game. But it might be more complicated:

```
player[1].isAlive() == true and player[2].isAlive() == true
and spaceStation.hasAir() == true
```

All three things need to be true to continue the game. Using boolean expressions, we can build very powerful tests to make sure everything is just as we need it to be.

We also need more kinds of boolean expressions when we are programming. Things like **less than** or **greater than or equal to**, and other **comparison operators** so we can compare things to work out the relationships within our data.

Chapter 9. Comparison Operators

```
healthScore = 5
```

We need a way to ask about expressions like "is healthScore less than 7?? (very healthy)" or "is healthScore greater than or equal to 3?? (maybe barely alive?)"

To do that we need a bunch of **comparison operators**.

Operator	Description	Example
==	Equal to	x == 5
!=	Not equal to	x != 55
>	Greater than	x > 1
<	Less than	x < 10
>=	Greater than or equal to	x >= 5
<=	Less than or equal to	x <= 5

Each of these can be used to make it very clear to someone reading your code what you meant. Imagine a flight simulator, where you're flying a big, old fashioned airplane. The code that keeps track of the status of the plane might need to be able to make decisions on boolean expressions like:

```
altitude > 500.0   # high enough to not hit any trees!
airspeed >= 85.0   # fast enough to stay in the air.

fuelAvailable <= 5.0   # need to land to refuel!

totalCargoWeight < 6.0  # more than 6 tons and we can't take off!

pilot.isAlive() and copilot.isAlive()  # everything is fine, keep
```

```
flying.
```

Like the ANDs in the examples above or this last boolean expression with the `and` in it, we have in Python the ability to combine expressions into larger more complex expressions using `and` and `or`.

Chapter 10. Logical Operators

The **logical operators** are AND and OR, except in Python we use **and** for AND and **or** for OR.

Operator	Description	
and	Logical AND	playerOneStatus == 'alive' and spacecraft.hasAir()
or	Logical OR	room.Temp > 70 or room.Temp < 75

The `and` operator is an operator where BOTH sides have to be true for the expression to be true.

```
(5 < 9) and (6-3 == 3)  # true
```

See how both expressions on either side of the `and` are true? That makes the entire line true.

The `or` operator is an operator where if ONE or the OTHER or BOTH boolean expressions are true, the entire expression is true.

```
(5 < 9) or (6-3 == 3) # true

(5 == 4) or (7 > 3)   # true!

(5 == 4) or (6 == 2) # false (both are false)
```

Both sides of a logical operator need to be Boolean expressions. So it's all right to use lots of different comparisons, and combine them with and and or.

```
# deep in a cash machine application...
```

```
(customer.balance() <= 20.00 \
and \
customer.hasOverDraftProtection() == true) \
or \
(customer.savings.balance() > 20.0 \
and \
customer.canTransferFromSavings() )
```

See how these conditions could line up to allow a customer to get cash from the cash machine? Again, this is **why** boolean expressions are important and powerful and why coders need to be able to use them to get the software **just right**.

- Create 2 variables to use for a comparison
- Use at least two comparison operators in Python
- And print them "print(2 > 1)"

Here is an example:

```
houseTemp = 67.0
thermostatSetting = 70.0

print(houseTemp >= 55.0)
print(houseTemp <= thermostatSetting)
print(thermostatSetting != 72.0)
print(houseTemp > 65.0 and thermostatSetting == 68.0)
```

These print statements should produce True, True, True and False.

Chapter 11. Strings

Strings are perhaps the most important data type in Python. Many other computer languages have strings, and they are used in almost ALL modern programs. Knowing how to manage them, create and modify them to do what you need them to do, is a "sub-superpower" within Python.

Pay close attention; this stuff is VERY important.

11.1. What is a String?

Think about the words on this page. The text here is made up of a bunch of letters, and spaces. Now, when we write by hand, we don't really think about the space between the words, do we? If we truly ignored the notion of space between the words, wewouldendupwithtextlikethis. And while it is possible to read, our modern eyes are trained on well-edited texts; having no spaces tires us pretty quickly.

So yes, what we see as text in this book is really a series of letters and spaces strung together in a line - line after line, paragraph after paragraph. In modern computing, that kind of data is often called a **String**. It is one of the most fundamental aspects of coding: the manipulation of strings by programs to transform, present or store text in some fashion.

Many programming languages use some kind of quote or double quote to show where strings start and end. There is really no difference between using single or double quotes in Python. So a string like "the quick brown fox" would be a string from the 't' to the 'x'. And notice the three spaces within the string. If they were not there, the string would be "thequickbrownfox". And that's important, because to the computer, if it keeps these two strings around, it doesn't really understand that 'the' and 'quick' are just two common English words. The spaces are there to retain more

of what the human meant.

No, to the computer, each letter, including the space 'letter', is just a piece of data and very important.

String - a string of letters and numbers and spaces and punctuation, kept altogether for some use. Here are some strings for you to consider.

```
"the quick brown fox"
"The New York Times"
"And lo, like wave was he..."
"oops"
"Hello, World!"
"supercalifraglisticexpealadocious"
"On sale for $123.99!!"
"Pi is approximately 3.14159"
"Merge left at the ramp to the right, the restaurant is on the
right"
'He said, "Wait there is more!"'
```

Think of strings as a tightly packed list. Each item and letter is numbered. The entire string can be "indexed", meaning I can reach in and copy out, say, the fifth letter easily. String indexes are zero-based; therefore, the first character (element) is in index position 0.

```
Index  -> 012345
String -> Hello
```

So here, "H" is at 0, "e" is at 1, 'l' is at 2 & 3, and 'o' is at index position 4. Computers often start numbering things like strings, lists, and arrays at 0, not at one. It's just one of those things: all strings and arrays (which are coming up) start at zero.

11.2. Declaring a string

```
name = "Wacka Flocka"
```

Now we have a string variable named **name** and it's value is "Wacka Flocka".

11.3. String Properties

A common and often used string property is **length**.

We can use `len()` builtin function to find the length of a string

```
motto = "Wakanda Forever!"
answer = len(motto)
print(answer) # -> 16
```

11.4. Accessing Characters in a String

As mentioned before, we can reach into a string and copy out the stuff we find there.

```
word = "Hello"

# Access the the first character by _indexing_
print(word[0]) # H

# the last character
print(word[len(word) - 1]) # o
```

When you see something like **word[0]**, it is pronounced like "word sub zero". If you have **word[5]**, you would say "word sub five". This is just verbal shorthand for the expression.

11.5. String Concatenation (Joining strings)

This simply means joining strings together using the + operator. We have to turn the 20 (which is a number) into a string. We do this by calling the builtin `str()` function.

```
price = 20
dollarSign = "$"
priceTag = dollarSign + str(price) # $20
```

Or perhaps a little more useful example:

```
name = "Mikaila"
hoursWorked = str(12)

workReport = "Today, " + name + " worked a total of " +
hoursWorked + " hours."
print(workReport)
```

The output would be:

```
Today, Mikaila worked a total of 12 hours.
```

11.6. SubStrings

Getting a substring is a common operation. This is how we extract the characters from a string, between two specified indices. (Which is why it's important to remember the indexes start at 0.) Python offers many ways to substring a string. It is often called 'slicing'. It follows this pattern:

`somestring[start: end: step]` and each of the start, end and step indices are *optional*.

`start:` The starting index of the substring. The character at this index is included in the substring. If start is not included, it is assumed to equal to 0.

`end:` The ending index of the substring. The character at this index is NOT included in the substring. If end is not included, or if the value is greater than the string length, it is assumed to be equal to the `len()` of the string by default.

`step:`` Every 'step' character after the current character to be included. If the step value is not there, Python sets it to 1.

A start position is required, where to begin the extraction (or substring). Remember, first character is at position 0. Characters are extracted from a string between "start" and "end", not including "end" itself.

```
firstName = "Christopher"
```

Now let's use the 3 substring method-functions on firstName and extract and print out "Chris"

```
firstName = "Christopher"

print(firstName[0:5]) # "Chris"
```

We took the `firstName` string and extracted the characters 0 to 4. BUT, let's try to extract the string "stop" from the name.

```
firstName = "Christopher"
print(firstName[4:8]) # -> "stop"
```

Let's try a little harder idea...

```
firstName = "Christopher"
```

- Your turn to use the string slicing on firstName
- Extract and print out "STOP" from inside the string above
- And make it uppercase! ("stop" to "STOP") [1]

Well?

```
firstName = "Christopher"
print(firstName[4:8].upper())
```

Want to bet there is also a ".lower()" method-function as well?

11.7. Summary of substring method-functions

Take a look at these various ways to copy out a substring from the source string named 'rapper', which contains the string 'mikaila'.

```
rapper = "mikaila"

print(rapper[0:4])  # mika
print(rapper[:4])  # mika
print(rapper[4:])  # ila

print(rapper[1:4])  # ika
print(rapper[1:3]) # ik
```

How about a few more?

```
print(rapper[:-2]) # mikai
print(rapper[-2:]) # la
print(rapper[5:]) # la

print(rapper[:]) # mikaila
```

We're using a variety of examples to copy out some smaller piece of the 'rapper' string. This is a powerful way to handle strings in Python.

11.8. Reverse a String

Now, using the `firstName` variable again, let's reverse the string "STOP" to say "POTS".

To Reverse a String

string = string[::-1]

Whoa. Using defaults as start and end indicates default to 0 and string length respectively and "-1" denotes starting from end and stop at the start, hence reversing string!

`print(rapper[::-1]) # aliakim`

There are several other means to reverse a string, but this one it most 'pythonic'.

Solution

```
firstName = "Christopher"
res = firstName[4: 8].upper() # -> "STOP"

rev = res[::-1] # -> POTS
print(rev) # -> POTS
```

Strings are perhaps the most important data type in almost any language. Being able to manipulate them easily and do powerful things with them in Python, makes you a better coder.

[1] You could google how to do this, try "python string make upper case"

Chapter 12. Lists

Lists are a very powerful idea in many programming languages. In Python, Lists are one of the most important built-in tools of the language. You need to master lists. Let's start with **why** we need them.

Imagine you have a small number of things you want to track. Let's use our vague computer game we've been using for an example. The game has 5 players, friends that get together over the internet to play a dungeon game.

Now, if you're the coder of this game you could keep track of each player's healthScore by have 5 different variables. (for players Zero to Four)

```
playerZeroHealthScore = 100
playerOneHealthScore = 100
playerTwoHealthScore = 100
playerThreeHealthScore = 100
playerFourHealthScore = 100
```

If we setup these 5 variables, our game can track 5 players! But **we'd have to change the game's code to track SIX players.** Well, that not good. Kind of silly actually.

To get around this kind of problem we use a **list**. We could ask, "how many players are playing?", and then make the list that size. We know we need to track each player's healthScore, so we create an list:

```
playerHealthScores = [100, 100, 100, 100, 100]
```

Now, like a *string*, list indexes start at zero.

```
                    #   0    1    2    3    4
playerHealthScores = [100, 100, 100, 100, 100];
```

This list is a **data structure** - a way for us to keep track of lots of data in a controlled fashion. (We can make lists any size, a list with a million things in it is not unreasonable.) If we need to deduct health points from one of the players, we can do something like this:

```
majorHit = 50

playerHealthScores[2] = 67  # player 2 just took a hit!

playerHealthScores[1] = 105  # player one is getting stronger.

playerHealthScores[3] = playerHealthScores[3] - majorHit
# the list now is [100, 105, 67, 0, 100]
```

The best way to think about a list is like all those postal boxes at the post office. Each box has a number on it, and things get put in the box depending on the box number.

Lists are **indexed** like that. Each list spot has an index number, starting at zero. See how we use the number 2 to *index* into the playerHealthScore list?

Lists:

- Can store multiple values in a single variable
- Start counting from index position zero
- Elements can be primitive data types or/and Objects

So let's think about an list of donuts for the following examples.

12.1. Declaring Lists

Declaring and initializing some lists in Python:

```
donuts = ["chocolate", "glazed", "jelly"

listofLetters = ['c','h','r','i','s']

mixedData = ['one', 2, True]  # a string, a number and a boolean!
```

12.2. Accessing elements of an List

We use square brackets to get elements by their index. We'll use an list of strings to identify our donuts. Sometimes, we say something like "donuts sub 2" to mean *donuts[2]*.

```
donuts = ["chocolate", "glazed", "jelly"]

print(donuts[0])  # "chocolate" (we could say "donuts sub zero")

print(donuts[2])  # "jelly"
```

12.3. Append to an List

We can also add things to the end of the list.

```
donuts = ["chocolate", "glazed", "jelly"]

donuts.append("strawberry")   # notice there is no element 3
before this,

print(donuts)  # but after, there are now 4 things in the list.
```

12.4. Get the size of an List

We can use the **length** property to find the size of an array.

```
donuts = ["chocolate", "glazed", "jelly"]

print(len(donuts))  # it'll print 3
```

Note: A string is an LIST of single characters

12.5. Get the last element of an List

If we use the `len()` function carefully, we can always get the last element in a list. The index of the last element in the list is 'length minus 1'.

```
donuts = ["chocolate", "glazed", "jelly"]

donuts("strawberry")    # -> ["chocolate", "glazed", "jelly",
"strawberry"]

print(donuts[len(donuts) - 1]) # strawberry

donuts.append("powdered")    # -> ["chocolate", "glazed",
"jelly", "strawberry", "powdered"]

print(donuts[len(donuts) - 1]) # powdered
```

Some computer languages have *arrays*. Python's **Lists** are like arrays and even more powerful.

Chapter 13. Changing the Control Flow

In many of these examples so far, we see a very simple **control flow**. The program starts at the first line, and just goes line by line until runs out of statements.

```
q = 0
j = 5
q = j * 4 - 20
print(q) # -> 0
```

When programs start to get more sophisticated, the *control flow* can be changed.

```
peoplePerTable = 5
tables = 8

maximumRoomOccupancy = tables * peoplePerTable
# up to this point we just stepped line by line

if (bridesDesiredPartySize > maximumRoomOccupancy):
    canHandleThisReception = False
else:
    canHandleThisReception = True
```

See how the IF statement can change a variable based on a decision about another variable? We are changing the *control flow* based on the state of the data within the program!

There are various conditional statements, loop statements, and functions that can cause the control flow to move around within the code. Here we see using both a loop and a conditional IF statement to change the flow of control.

```
q = 0
```

```
j = 6
while (j > 0) :
    q = j * 4 - 20
    print(q)
    j -= 1  # Hey! this how you can decrement by 1
    if (q > 0) :
        print("q is still positive")
```

This ability to manipulate the control flow of a program is very important when you start developing logic for your apps and programs. Logic in programs depends heavily on being able to manipulate the control flows through the code. Let's take a look at how each kind of statement allows a programmer to change the flow of control in programs.

Chapter 14. Conditional Statements

We have been seeing programs which consist of a list of statements, one after another, where the "flow of control" goes from one line to the next, top to bottom, and so on to the end of the list of lines. There are more useful ways of breaking up the "control flow" of a program. Python has several conditional statements that the programmer do things based on conditions in the data.

14.1. If statement

The first conditional statement is the **if** statement.

```
if (something-is-true):
    doSomething()
```

Here are a few simple examples.

```
if (speed > speedLimit) :
    driver.getsATicket()

if (x <= -1) :
    print("Cannot have negative numbers!")

if (account.balance >= amountRequested) :
    subtract(account, amountRequested)
    produceCash(amountRequested)
    printReceipt(amountRequested)
```

Python also has an **else** part to the **if** statement. When the **if** condition is False, the else part gets run. Here, if the account doesn't have enough money to fulfill the amountRequested, the else part of the statement gets run, and the customer gets an

insufficient funds receipt.

```
if (account.balance >= amountRequested):
    # customer has the money
else:
    printReceipt("Sorry, you don't have enough money in your
account!")
```

Python can also "nest" if statements, making them very flexible for complicated situations. You can also see here how the `elif` works. There can be zero or more `elif` parts, and the `else` part is optional.

```
timeOfDay = "Afternoon"

if (timeOfDay === "Morning"):
    print("Time to eat breakfast")
    eatCereal()
elif (timeOfDay === "Afternoon"):
    print("Time to eat lunch")
    haveASandwich()
else:
    print("Time to eat dinner")
    makeDinner()
    eatDinner()
    doDishes()
```

Notice how this becomes a 3-way choice, depending on the timeOfDay.

Write code to check if a user is old enough to drink.

- if the user's age is under 18. Print out "Cannot party with us"
- Else if the user's age is 18 or over, Print out "Party over here"

- Else print out, "I do not recognize your age"

You should use an if statement for your solution!

Finally, make sure to change the value of the age variable in the repl, to output out different results and test that all three options can happen. What do you have to do to make the `else` clause happen?

```
userAge = 17
if userAge < 18:
    print("Cannot party with us")
elif userAge >= 18:
    print("Party over here")
else:
    print("I do not recognize your age")
```

If statements are one of the most commonly used statements to express logic in a Python program. It's important to know them well.

Chapter 15. Loops

Loops allow you control over repetitive steps you need to do in your *control flow*. Python has two different kinds of loops we will talk about: **while** loops and **for** loops. Either one can be used interchangeably; but, as you will see there are couple cases where using one over the other makes more sense.

The primary purpose of loops is to avoid having lots of repetitive code.

15.1. While Loop

Loop through a block of code (the body) WHILE a condition is true.

```
while (condition_is_true):
    # execute the code statements
    # in the loop body
    pass
```

See the code below. In this case, we start with a simple counter in x = 1. Then, after the loop starts, it checks to see if x < 6, and 1 is less than 6, so the loop body gets executed. We print out 1 and then increment x. Then we go to the top of the loop and check to see if x (now 2) is less than 6. Since that's true so we print out 2 and increment x again. This continues like this for three more times, printing 3, 4, and 5.

Then, x is incremented to 6, and the check is made again, 6 < 6 ... well, no that is false. So we don't execute the loop's body and we fall through to the last print line, and print out x.

```
x = 1

while (x < 6):
```

```
    print(x)
    x += 1

print("ending at", x) # ? what will print here ?
```

While loops work well in situations where the condition you are testing at the top of the loop is one that may not be related to a simple number.

```
while (player[1].isAlive() == True):
    player[1].takeTurn()
    game.updateStatus(player[1])
```

This will keep letting player[1] take a turn in the game until the player dies. Another way to do something like this is with an *infinite loop*. (No, infinite loops are not necessarily a bad thing, watch.) We're going to use both **continue** and **break** in this example, and we will describe them better after we're done with loops.

```
player = game.newPlayer()

while (true):  # <- notice right here, an infinite loop

    player.takeTurn()
    game.updateScores()
    game.advanceTime()

    if (player.isAlive() == true):
        continue # start at top of loop again.
    else:
        break # breaks out of loop and ends game.

game.sayToHuman("Game Over!")
```

Here, we are using the continue statement to force the flow of control to the top of the loop if the player is still alive after the 'take turn' code. We are also using the break statement to break

out of the infinite loop when the player dies, letting us do other things after the player has 'died'.

15.2. For Loop

The **for** loop is more complex, but it's also the most commonly used loop.

In Python, the for loop is very powerful. It has a number of different forms, including one which is remarkably simple.

Maybe we have a list of donuts. And we want to print out each one.

```
list_of_donuts = ["chocolate", "glazed", "jelly"]

for donut in list_of_donuts:
    print(donut)

# you'll see as output
chocolate
glazed
jelly
```

This is called a **for-each** loop. It steps thru the list, and **For Each** donut in list_of_donuts, it does whatever the code in its block specifies - here just printing out the donut. It has the ability to iterate over the items of any sequence, such as a list or a string. A *sequence* here is a list, or a string, or a couple of other data structures we haven't yet talked about.

Just like slicing a string, a list (or any **sequence of data**) can be thought of a series of data point where the **index** start at zero and goes to **len(thing)-1**. Just like a string "hello" which goes from 0..4, our donut list goes from 0..2 - in this example you see the **range()** function for the first time. And in python code, you see a lot of range().

```
list_of_donuts = ["chocolate", "glazed", "jelly"]

for index in range(len(list_of_donuts)):
    print(list_of_donuts[index])
```

Here the index was set to 0 and then 1, and then 2, and the index was used to retrieve from the list the donut name at the index. list_of_donuts[1] is "glazed". This example's out is the same as the for-each loop above.

Let's use range() is much more common usage.

Here's one where we go from 0 to 4.

```
for j in range(5):
    # loop body code
    print(j)
```

Notice how in this case, the loop prints out 0..4!

Now say we wanted to print out the sum of all the numbers from 1 to 10.

```
# performing sum of 1 to 10
sum = 0
for i in range(1, 11):
    sum = sum + i
print("Sum of first 10 natural numbers is ", sum)
```

Wait! Notice how we have to ask range to **one beyond** the last number we want added to the sum! That's because the range starts at 0 and includes every whole number up to, but **not including**, the number that you have provided as the stop number.

There are three versions to the range()'s mechanism.

- range(stop) when you provide one number it acts as the stop value.
- range(start, stop) when you provide two numbers, the first is start, the second is stop.
- range(start, stop, step) when you provide three, they are taken to mean start, stop and step.

So if you have:

- range(5) gives 0, 1, 2, 3, 4
- range(2, 5) gives 2, 3, 4
- range(1, 8, 2) gives 1, 3, 5, 7

Let's show you a glimpse of the **break** statement.

```
for p in range(1,6):
    if p == 4:
        break

print("Loop " + str(p) + " times")
```

Jumps out of the loop when p is equal to 4.

What about if we print from 10 to 1 with a for loop and a while loop (hint: we need to decrement).

```
for p in range(10, 0, -1):
    print(p)
```

Look, we had to range(10, 0, -1) to get the one to print. Otherwise if we had had used range(10,1,-1), we would have only printed 10 to 2.

The **range()** function gets used a lot in Python, and is the envy of many programmers in other languages who have use the **for** loop

that was handed down from the C programming language. So use it widely, and don't gloat — too much :-)

15.3. Pass Statement

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
while True:
    pass
```

That was an *infinite loop*!

`pass` is commonly used for creating minimal classes:

```
class MyEmptyClass:
    pass
```

Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

```
def isPlayerAlive(*args):
    pass   # you'll program this later
```

15.4. Break Statement

Normally, a loop exits when its condition becomes false. But we can force the exit at any time using the special **break** statement.

```
def execute(c):
```

```
  pass

while (True):
  cmd = input("Enter a command? ")
  if (cmd == "exit"):
    break
  execute(cmd)

print("Exiting.")
```

Here, you are asking the user to type in a command. If the command is "exit", then quit the loop and output "Exiting", and end the program. Otherwise, execute the command and go around to the top of the loop and ask for another command.

15.5. Continue Statement

The continue statement doesn't stop the whole loop. Instead, it stops the current iteration and forces the loop to start a new one (if the condition allows).

We can use it if we're done with the current iteration and would like to move on to the next one. This loops prints odd number less than 10.

```
for i in range(1,10):
  # if true, skip the remaining part of the body
  # will only be true if the number is even
  if (i % 2 == 0):
    continue
  print(i) # prints 1, 3, 5, 7, 9
```

What's interesting here is the use of the remainder operator (%) to see if a number is odd. The expression (i % 2) is zero if the number is even, if not, the number must be odd. You want to remember this trick of how to find odd or even numbers. It's a common programming problem that you will get asked. The continue statement starts the loop over, not letting the print to

print out the number when it’s even.

But you, being a newbie **pythonista** might say: 'yeah, but why not just use **range(1,10,2)** instead of all the IF and Modulo and **continue**??'

And me, being the crusty old instructor might say, 'Bah, Humbug!'

Chapter 16. Code Patterns

Any experienced coder would say that the ability to see patterns in code, remember them, and learn from them when creating code is another kind of 'superpower'. The following samples are really simple techniques, but they show some common ways of doing things that you should think about and study. In almost all these examples, there may be some missing variable declarations. Just roll with it. If you think about it, I'm sure you can figure out what variables are needed to run the sample in the REPL page.

16.1. Simple Patterns

If you wanted to find the larger of two values, x and y, and assign it to 'max':

```
if (x > y):
    max = x
else:
    max = y
```

Related to it, if we have two variables x and y, and we want the smaller in x, and the larger in y.

```
if (x > y):
    t = x
    x = y
    y = t
```

Do you see the three statements in the block there? That's called a 'swap'. If you need to swap two values in two variables, you just create a quick temporary variable 't' and use it as a place to make a copy of the first variable's value.

But let me show you a **really cool** trick in Python. It's related to a

data type we haven't talked about yet, call a **tuple** (which I pronounce "two-pell").

I can in python do things like this:

```
x, y = 5, 7
# yes that assigns 5 to x and 7 to y!

# I can also, and this is the cool part,

y, x = x, y
# which does the swap of the values *without a temporary 't'
variable*.
```

If I needed to make sure a number is always positive (greater than zero), it's easy - this is called taking the "absolute value" of a number.

```
if (n < 0):
    n = -n
```

16.2. Loop Patterns

The next few are examples of the handy use of loops to do a bunch of math easily and quickly. Imagine a problem where you have to "add all the numbers from 1 to 100 and print the sum." It might also be expressed as "**sum** all the number from x to y" (where x and y are two integers). Turns out there is a very easy pattern to learn here.

```
sum = 0
n = 100
for i in range(0, n+1):
    sum = sum + i

print(sum)  # 5050
```

Now, if you wanted to find the average of a bunch of numbers, that's as easy as taking the sum of the numbers and dividing the sum by the number of numbers (or n).

```
sum = 0
n = 100
for i in range(0, n+1):
    sum = sum + i

average = sum / n
print(average)  # 50.5
```

Pretty easy, yes? And the other common pattern here is doing a **product** of all the numbers from 1 to n. (Let's try 20)

```
product = 1
n = 20
for i in range(1, n+1):
    product = product * i;

print(product) # Whoa! -> 2432902008176640000
```

Perhaps you want to print a table of values of some equation.

```
n = 20
for i in range(1, n):
    print(str(i) + " " + str(i*i/2))
```

16.3. List Patterns

Lists are often something that confuses beginning coders. Let's look at some code patterns with lists that you see how lists and loops can work together to get a lot of work pretty easily.

The list we are going to use in all these cases is pretty simple. It's a list of 7 numbers.

```
a = [ 4, 3, 7, 0, -4, 1, 8]
```

Here how to print out the list, one value per line.

```
for i in a:
    print(i) # each element in the list
```

If we needed to find the **smallest** number in the list, we could do:

```
min = a[0]
for i in a:
    if (i < min):
        min = i

print(min)
```

We should look carefully here. First, notice how I have taken the first element a[0] and made my first 'min' that value. Then we step through the list, looking at each value and if the new value is smaller than the previous one, we update it; otherwise, we just do the next value.

NOW, if you wanted to find the **largest** value in the list, you really only have to change a couple things.

```
max = a[0]
for i in a:
    if (i > max):
        max = i

print(max)
```

Carefully look at the code, comparing to the one above. What's different? Well, for one, we changed the variable from 'min' to 'max'. (But did we need to do that? We could have left it max, but

it's cleaner to make the change so people who read it aren't confused.) We also changed the comparison in the 'if' statement from "less than <" to "greater than >" which lets us decide if the new number is larger than the previous largest we found.

In both of these cases, we start with an initial value, then we step through the list, look at each value comparing it to the smallest (or largest) value we have yet found. If we need to update the 'carrying variable', we do; otherwise, we just ignore the value.

What about finding the average of the values in the list? Well, we do it a lot like the average of the series of numbers.

```
sum = 0
for i in a:
    sum += i # this is the same as `sum = sum + i`

average = sum / len(a)  # whoa! lookee there?

print(average)
```

Yep, the "len(a)" is very handy, it has exactly the count of the numbers in the list!

Finally, if we wanted to reverse the values in the list, we could write some code:

```
import math

print("before:", a)
n = len(a)
half = math.ceil(n / 2)
for i in range(0,half):
    t = a[i]
    a[i] = a[n-1-i]
    a[n-i-1] = t

print("after: ",a)
```

But perhaps the easier way to reverse an list in Python is to just call the library function:

```
a.reverse()
print(a)
```

It can be useful to look at the "longer" way to continue to get a feel for how to do small, useful things with simple logic.

Chapter 17. Functions

A function is a block of code designed to perform a particular task. A function encloses a set of statements and is a fundamental modular unit of Python. They you reuse code, and provide a way for you to organize your programs, keeping them easier to understand and easier to modify. It's often said that the craft of programming is the creation of a set of functions and data structures which implement a solution to some problem or set of requirements.

Functions are objects, like many things in Python. They can be stored in variables, other objects, or even collected into lists. Functions are very powerful because in Python, they can be passed as arguments to other functions and returned from functions. The most important thing that functions can do is get **invoked**.

You create functions easily.

17.1. Function Definition

An example of how you write a function definition:

```
def add(p1, p2):
    return p1 + p2
```

You end up with a function named 'add' that takes two parameters and adds them, returning the result. (Yes, you could just write (p1 + p2) and everyone would understand. It's just a very simple example.) Here is another example:

17.2. Creating a Function

```
def greetUser(username):
```

```
    print( "Hello " + username)
    return

# calling/Invoking the function
greetUser("Mike Jones"); # "Hello Mike Jones"
```

17.3. Invoking Functions

Functions are meant to be *invoked*. So you can have one function call another function which calls a third function and so on; it's very common.

Imagine we have a program that gets an airplane ready for flight. We can imagine a whole series of functions we'd have to write. Things like 'loadPassengers', 'loadBaggage', 'loadFuel', 'checkTirePressures', and so on. Then we might have a 'higher level' functions which brings all these pieces together:

```
def prepFlight(airplane):
    loadBaggage(airplane)
    loadPassengers(airplane)

    loadFuel(airplane)
    performPreflightChecklist(airplane[copilot])
    askTowerToDepart(airplane[pilot])
    departGate(airplane)

    taxiToRunway(airplane, mainRunway90)
    takeoff(airplane[pilot], airplane)

    return
```

You can see how functions you perform different things in a particular order, and while you might have no idea *how* 'loadBaggage' does what it does, you can see how the program preps the airplane object for flight and makes sure everything important is done. Each of the functions from 'loadBaggage' to 'takeoff' is invoked and returns so the next one can be invoked.

This is the power of functions - the ability to take some code and put it a function so that it much easier to understand.

17.4. Lambda Functions

A common pattern used in Python revolves around an **lambda** function. Something like:

```
double = lambda x: x * 2

print(double(5))
```

This gives us another way to wrap up a little code into a function. We could redo the first example in this chapter with a lambda function.

```
add2 = lambda p1, p2: p1 + p2

print(add2(5,7))
```

17.5. Function Return

Once Python reaches a return statement, the function will stop executing. Functions often compute a return value. The return value is "returned" back to the "caller". You can have many returns in a functions, depending on how the flow of control is changed.

```
def greetUser(username):
    return "Hello " + username

result = greetUser("Welcome back, Mike Jones")
print(result); # will print "Hello Welcome back, Mike Jones"
```

Or like this:

```
def determineWinner(home, visitor):
    if (home[score] > visitor[score]):
        return "Home Team Wins! Let's have a Parade!"
    elif (home[score] < visitor[score]):
        return "Visitors Win! (oh Well)"

    return "It's a Tie!"
```

Notice how in this case, we check to see the scoring results with two conditions (which are, what? yes, *boolean expressions*). If neither condition is true, the third one must be the case. But if either condition is true, then we return right away, and the function is done.

Again, to be clear, we might use this function like this:

```
home = {'name': "Fightin Cats", 'score': 0 }

visitor = {
    'name': "Wild Horses",
    'score': 0
}

playGame(home, visitor) # a lot of work done in this function(!)

# game is done
result = determineWinner(home, visitor)

# and then print the result..
print(result)
```

17.6. Function Parameters

As you just saw, functions can also take parameters to be used within a function.

```
def addThreeNumbers(a, b, c):
    return (a + b + c)

def determineWinner(home, visitor):
    if (home['score'] > visitor['score']):
        return "Home Team Wins! Let's have a Parade!";
    elif (home['score'] < visitor['score']):
        return "Visitors Win! (oh Well)";

    return "It's a Tie!"

def makeNegative(number):
    if (number > 0):
        return -(number)
    # already negative, it's less than 0
    return number
```

Remember how we had the expression ot see if a number was even? (x % 2 === 0) Now, here's a way to decide is number was divisible cleanly by another, it's a standard arithmetic expression:

```
(number % divisor == 0)
```

So to see if a number is even, we could use '(number % 2 == 0)':

```
print((8 % 2 == 0)); # true
print((7 % 2 == 0)); # false
print((4 % 2 == 0)); # true
```

And we can use the same technique to see if a number is evenly divisible by 3 or 5.

Try to write a function that will perform the following requirements:

- Create a function called zipCoder

- Your function takes one parameter of type number
- Your function checks and does the following
- If parameter is divisible by 3 and 5 (15). Print ZipCoder
- If parameter is divisible by 3. Print Zip
- If parameter is divisible by 5. Print Coder Phew...Finally
- Call the method-function and pass in 45 as your parameter

OKAY! Write it yourself!

Do it.

Just write it yourself.

C'mon, write your own version first.

No, really.

Wait.

Do you want to be a ZipCoder, or just a Copy-Paste Stylist?

Well, here's one solution:

```
# Function ZipCoder

def zipCoder(aNumber):
    if (aNumber % 15 == 0):
        print("ZipCoder")
    elif (aNumber % 3 == 0):
        print("Zip")
    elif (aNumber % 5 == 0):
        print("Coder")
```

```
zipCoder(45) # -> ZipCoder
```

Chapter 18. Return statement

The **return** statement is a very simple one. It just finishes the running of code in the current function and "returns" to the function's caller.

As you have seen, *functions* are used to make code more understandable, cleaner and more organized. Say we have a couple of functions in our program:

```
minorHit = 3
majorHit = 7

def adjustHealth(player, hit):
    player[health] = player[health] - hit

    if (isAlive(player) == false):
        return playerDead

    return playerAlive

def isAlive(player, hit):
    if (player[health] >= 20):
        return true
    else: # player has died!
        return false
```

If someplace in our code we were to do something like:

```
# big hit!
continuePlaying = adjustHealth(playerOne, majorHit)

if (continuePlaying == playerDead):
    endGame()
```

You can see how when we call the function "adjustHealth()" it returns either playerAlive or playerDead, and we make a decision to end the game if the player has died.

Notice too, you can have multiple return statements in functions, and each one can return a different value if that's what you need.

Chapter 19. Dictionaries

Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.

It is best to think of a dictionary as a set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

19.1. Creating a Dictionary

Creating a dictionary is easy:

```
phonenumbers = {'Sam': 6554098, 'Jade': 6554139}
```

19.2. Modifying a Dictionary

Modifying a dictionary and getting values from them is very easy.

```
# get Sam's number
currentPhone = phonenumbers["Sam"]

# we can also update a value
phonenumbers['Jade'] = 6551439

# you can delete pairs too
del phonenumbers["Sam"]
```

You can add another dictionary key:value pairs.

```
# we can also add a new key:value pair.
phonenumbers['Sadie'] = 6551004
```

We can print it out.

```
# we can also add a new key:value pair.
print(phonenumbers)

# output is
# {'Sam': 6554098, 'Jade': 6554139, 'Sadie': 6551004}
```

19.3. Testing for a Key

And one of the very cooler aspects is the testing of whether or not a key is in the dictionary. You use *in* and *not in* to determine

```
if 'Sam' in phonenumbers: # -> True
    doSomething()

if 'Guido' not in phonenumbers: # -> False
    figureOutWhyNot()
```

We could use them to hold information about some object we're holding data about.

```
vehicle1 = {
    'Name': "Mars Lander"
    'Altitude': 8000,
    'Speed': 1000,
    'Fuel': 12000,
}
```

Or this dictionary

```
katniss = {
    'firstname': "Katniss",
    'lastname': "Everdene",
    'homedistrict': 12,
    'skills': ["foraging", "wildlife", "hunting", "survival"]
}
```

Dictionaries are a tremendous tool. They are used in many, many places within Python systems. Dictionaries are built-in data types in Python that associate (map) keys to values, forming key-value pairs. You can access, add, modify, and delete key-value pairs.

Chapter 20. Modules

In Python, modules allow for code to loaded into a program only if it is needed. Modules are one of the advanced topics in Python that we won't spend too much time on, but here are the basics.

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable *name*.

Most of your Python programs are fairly small when you are creating solutions to HackerRank type problems.

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

helloworld.py

If your program is much larger, it might be split into different

files to keep it all more organized or readable. All those files might be kept in a folder all together, as a project. But again, this is beyond what you need to know to do HackerRank Python problems.

Modules are also used to import code others have written that you wish to take advantage of. You've seen examples of this when we've used "import math" to get access to function definitions within the math module. There are millions of chunks of Python you can find and use in your code. A lot of it is used by many, many people, and it's important to know where the code you use comes from. It can be dangerous to use someone else's code that isn't trustworthy.

See https://docs.python.org/3/tutorial/modules.html for more on modules! Look for information on "import" to see how modules interact with your code.

Chapter 21. Objects

There are only a few data types in Python. All but one of them are called “primitive” data types, because their values contain only a single thing (be it a string or a number or whatever).

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

So you write "classes", and then when you run the code, Python then creates the objects to be used by the code.

Objects are used to collect and organize data - and that data can be variable values, functions and other things. Objects can also contain other objects(!), in kind of a "nesting" way. This allows for large data structures to be built using a very simple and elegant mechanism.

```
class Spacecraft:
    name = ""
    type = 'warp starship'
    topspeed = 9.9
```

21.1. Object Creation

We can imagine an object as a container where everything is collected about some thing in the program:

```
enterprise = Spacecraft()
enterprise.name = "Enterprise"

print(enterprise) # -> <__main__.Spacecraft object at
0x103013100>
```

Which isn't very helpful, is it? We need to create a *method* in our class. One that gets called whenever we want to display which spacecraft to a user.

```
class Spacecraft:
    name = ""
    type = 'warp starship'
    topspeed = 9.9

    def __str__(self):
        return "Starship " + self.name

    def speed(self):
        return self.topspeed

enterprise = Spacecraft()
enterprise.name = "Enterprise"

# now when we
print(enterprise)
# we get: Starship Enterprise
```

There a few things to point out in this class. First, notice how the *str* function is indented? It's defined *inside* the class. That makes the function a *method*; objects have methods. You can call a method on a particular object.

```
enterprise = Spacecraft()
enterprise.name = "Enterprise"

constellation = Spacecraft()
constellation.name = "Constellation"
constellation.topspeed = 9.2

ets = enterprise.speed()
cts = constellation.speed()

# ets will be 9.9, cts will be 9.2
```

This example also shows how you use a class to create multiple

objects. It helps with making code simpler, and easier to follow when reading it.

21.2. Follow Ons

There are a number of very powerful things we have left out of this discussion about Python objects. We have not covered the ideas of **inheritance** or the idea of **subclassing**. And there is much more in Python about objects. Master what we've written about here and then forge ahead into more complicated and powerful capabilities.

There is a lot more to learn about Python.

Appendix A: Advanced Ideas

We're going to look at a few "modern" ways of handling a collection of data. Frequently, you have a list, or an array, of data that needs to be gone through to print it out, transform it in some way, or to summarize it (such as a total or an average). As you have seen in the code patterns section, there are common loops used for such things, a simple pattern that you can memorize.

There are other method-functions of doing these things, and we're going to discuss a few of them here. These ideas are based primarily on method-functions made popular by Hadoop and other "big data" applications and tools. And what's good for "big" data is often good for "small" data as well.

Each of these sections is an example of a more "elegant" way of expressing coding logic. By studying each one and comparing it to the ways we've discussed before using loops and conditional statements, we're expanding your understanding, making you see how these techniques can be used to create more extensible and elegant code.

Let's use this array for the following examples.

```
groceries = [
  {
    'name': 'Breakfast Cereal',
    'price': 5.50,
  },
  {
    'name': 'Rice',
    'price': 14.99,
  },
  {
    'name': 'Oranges',
    'price': 6.49,
  },
  {
    'name': 'Crackers',
```

```
    'price': 4.79,
  },
  {
    'name': 'Potatoes',
    'price': 3.99,
  },
]
```

A common grocery list, we have this as a list of dictionaries (what's known as a key/value data structure). The dictionaries hold a grocery item, its name and its price.

A.1. Simplifying Loops

Now, if you wanted to print out each item's name in the grocery list to the console, you could do something like this:

```
for item in groceries:
    print(item['name'])
```

This is a very common code pattern in Python. It's also a pretty simple loop.

Rather, how about this:

```
[print(item['name']) for item in groceries]
```

It is list comprehension, and in Python its remarkably powerful. Say we wanted to print the total cost of our grocery list. If we did it with a loop and sum variable:

```
prices = []
for item in groceries:
    prices.append(item['price'])

sump = 0
```

```
for price in prices:
    sum = sum + price

print(p)
```

Not too bad, but we can put it all on one line, and only one loop:

```
total = sum([item['price'] for item in groceries])

# printing total gets us 35.76
```

This example step through the grocery list, pulling out each item, and the item['price'] pulls out the price and places it in a list. When loop is done, the sum() function sums and returns all the prices in the list.

Appendix B: Mars Lander

This is some code to show you how you might write a simple Mars lander simulation in Python. It's taken from history, way back in the 1970's - this idea was passed around as some of the very first open source.

Meant as an example of a longer program (159 lines) to get you thinking, it's really not very complicated. The general idea is you have a series of "burns" in a list, and the game (or simulator, if you will) steps through the list applying each burn. If you run out of altitude (or height) while you're going too fast, you will crash.

The tricky bit would be for you to figure out what `burnArray` would be used to safely land at a vehicle speed 1 or 2. That could be hard.

```
# Mars Lander Source Code.
import math
import random

GRAVITY = 100
# The rate in which the spaceship descents in free fall (in ten
seconds)

version = "1.2" # The Version of the program

# various end-of-game messages.
dead = "\nThere were no survivors.\n\n"
crashed = "\nThe Spaceship crashed. Good luck getting back
home.\n\n"
success = "\nYou made it! Good job!\n\n"
emptyfuel = "\nThere is no fuel left. You're floating around like
Wheatley.\n\n"

def randomheight():
    # start from a random altitude
    max = 20000
    min = 10000
    r = math.floor(random.random() * (max - min)) + min
    return (r % 15000 + 4000)
```

```
def gameHeader():
    s = ""
    s = s + "\nMars Lander - Version " + version + "\n"
    s = s + "This is a computer simulation of an Apollo Mars
landing capsule.\n"
    s = s + "The on-board computer has failed so you have to land
the capsule manually.\n"
    s = s + "Set burn rate of retro rockets to any value between
0 (free fall) and 200\n"
    s = s + "(maximum burn) kilo per second. Set burn rate every
10 seconds.\n"
    # /* That's why we have to go with 10 second-steps. */
    s = s + "You must land at a speed of 2 or 1. Good Luck!\n\n"
    return s


def getHeader():
    s = ""
    s = s + "\nTime\t"
    s = s + "Speed\t\t"
    s = s + "Fuel\t\t"
    s = s + "Height\t\t"
    s = s + "Burn\n"
    s = s + "----\t"
    s = s + "-----\t\t"
    s = s + "----\t\t"
    s = s + "------\t\t"
    s = s + "----\n"
    return s



def computeDeltaV(vehicle):
    return (vehicle['Speed'] + GRAVITY - vehicle['Burn'])


def checkStatus(vehicle):
    s = ""
    if (vehicle['Height'] <= 0):
        if (vehicle['Speed'] > 10):
            s = dead

        if (vehicle['Speed'] < 10 and vehicle['Speed'] > 3):
            s = crashed
```

```
        if (vehicle['Speed'] < 3):
            s = success
    else:
        if (vehicle['Height'] > 0):
            s = emptyfuel


    return s


def adjustForBurn(vehicle):
    # save previousHeight
    vehicle['PrevHeight'] = vehicle['Height']
    # compute new velocity
    vehicle['Speed'] = computeDeltaV(vehicle)
    # compute new height of vehicle
    vehicle['Height'] = vehicle['Height'] - vehicle['Speed']
    # subtract fuel used from tank
    vehicle['Fuel'] = vehicle['Fuel'] - vehicle['Burn']


def stillFlying():
    return (vehicle['Height'] > 0)


def outOfFuel(vehicle):
    return (vehicle['Fuel'] <= 0)


def getStatus(vehicle):
    # create a string with the vehicle status on it.
    s = ""
    s = str(vehicle['Tensec']) + "0 \t\t" + str(vehicle['Speed'])
\
        + " \t\t" + str(vehicle['Fuel']) + " \t\t" + str(vehicle
['Height'])
    return s


def printString(string):
    # print long strings with new lines the them.
    # a = string.split(/\r?\n/)
    #for (i = 0 i < a.length i++):
    #    print(a[i])
    print(string)
```

```
# this is initial vehicle setup
vehicle = {
    'Height': 8000,
    'Speed': 1000,
    'Fuel': 12000,
    'Tensec': 0,
    'Burn': 0,
    'PrevHeight': 8000,
    'Step': 1,
}


# main game loop
def runGame(burns):
    status = ""

    # Set initial vehicle parameters
    h = randomheight()
    vehicle['Height'] = h
    vehicle['PrevHeight'] = h

    burnIdx = 0

    printString(gameHeader())
    printString(getHeader())

    while (stillFlying() == True):
        status = getStatus(vehicle)
        vehicle['Burn'] = burns[burnIdx]
        printString(status + "\t\t" + str(vehicle['Burn']))
        adjustForBurn(vehicle)
        if (outOfFuel(vehicle) == True):
            break

        vehicle['Tensec'] += 1
        burnIdx += 1



    status = checkStatus(vehicle)
    printString(status)
```

```
# these are the series of burns made each 10 secs by the lander.
# change them to see if you can get the lander to make a soft
landing.
# burns are between 0 and 200. This burn array usually crashes.
burnArray = [100, 100, 200, 200, 100, 100, 0, 0, 200, 100, 100,
0, 0, 0, 0]

runGame(burnArray)
```

Appendix C: Additional Python Resources

Here are a series of other resources to go on from this point.

Some Python sites for you to explore:

- https://docs.python-guide.org (The Hitchhiker's Guide to Python)
- https://docs.python.org/3/tutorial/index.html

If you're looking for more of a professional code tool, use an IDE like vscode: https://code.visualstudio.com (Many people use this these days.) It has all sorts of tools to help you create Python programs.