PYTHON CRASH COURSE

A HANDS-ON, PROJECT-BASED INTRODUCTION TO PROGRAMMING

DANIEL BEGUM

PYTHON
CRASH COURSE

Table of contents

Introduction

Welcome to Your New Programming Language

So, you've decided to learn Python Programming? Well, congratulations and welcome to your new Programming Language! You're going to love it!

In Eprogramy we believe that the foundation of a good education is to provide the tools able to open the doors of the future. It is indisputable that the world has entered an era in which we all have everything at the fingertips. Social needs have changed. Thus, today communication has changed. Communication opens doors, so it is our challenge to learn how to communicate. In Eprogramy we want you to learn how to communicate with a new language: The programming language. This language allows us to use our PC or notebook to create.

Maybe the road is difficult and hard, but we will give the reader the tools to make learning enjoyable and fruitful. We'll show all aspects necessary to learn how to program. From the ABC ´s to the solution of the most common programming problems and much, much more. Always with the help of numerous examples that will contribute to a better understanding

We know what we do and believe that you are fully capable of incorporating our teachings.

The doors of the future are here. Let's go step by step together.

Let´s get started!

Chapter 1

Python Programming Language

History of Python

Python is known as the go-to language for beginners as it is recommended by computer programmers around the globe as a good language for beginners to learn. This should not be misinterpreted for its powerful nature.

The Python language was created in 1990 by Guido Von Rossum at Stichting Mathematisch Centrium in the Netherlands. The language itself has been actually developed by a large team of volunteers and is available to modify. In the development stage that it was in, it had classes with inheritance, exception handling, functions, and the core datatypes of list, dict, str and more.

In May 2000, Guido and the Python development team moved to BeOpen.com to form the BeOpen PythonLabs team. In the same year, the PythonLabs team moved to Digital Creations which is now known as Zope Corporation.

Python 2.0 was released on the 16th of October in the year 2000 with features including a garbage collector which helps maintain memory handling related issues in programming. The great thing about Python was that it is backed by a community and it has a transparency behind the users that utilize the Python language.

Soon after, Python 3.0 which was a major backwards-incompatible release was released on December 3rd 2008 after a long period of testing. The major features of Python 3.0 also have been backported to backwards-compatible Python 2.6 and 2.7. In this guide, we'll be going over Python 3.4.

Python Version Release Dates:

- **Python 1.0 - January 1994**
    - Python 1.5 - December 31, 1997
    - Python 1.6 - September 5, 2000

- **Python 2.0 - October 16, 2000**
    - Python 2.1 - April 17, 2001
    - Python 2.2 - December 21, 2001
    - Python 2.3 - July 29, 2003
    - Python 2.4 - November 30, 2004
    - Python 2.5 - September 19, 2006
    - Python 2.6 - October 1, 2008
    - Python 2.7 - July 3, 2010

- **Python 3.0 - December 3, 2008**
    - Python 3.1 - June 27, 2009
    - Python 3.2 - February 20, 2011
    - Python 3.3 - September 29, 2012
    - Python 3.4 - March 16, 2014

What is Python?

Python is a freely available object-oriented, high-level programming language with dynamic semantics. Many programmers say great things about Python because of the increased productivity that it provides since the edit-test-debug cycle is incredibly fast compared to other programming languages.

The Python language has a simple, easy to learn syntax which is empowered with many English words for easier readability and helps for increased productivity and efficiency. When coding in Python, it feels more like you are writing out the solution to a problem in your own thoughts rather than trying to refer to ambiguous symbols that are required in the language to commit to certain functionalities.

Python could be used to automate measurements and process data interactively. The language is able to handle large databases and compute large numbers strain-free compared to many other programming languages. It can be used as an internal scripting language so that it is executed for certain functions by other programs. By learning Python, you will be able to write complex software like intricate web crawlers. It is truly an all-purpose language.

The great thing about Python is that it has a giant library for web crawling and is used a lot for its capability in scraping the web. A web crawler is simply a program that can navigate the web depending on the parameters set out for it and scrapes content that you would like for it to scrape.

All in all, Python can be easy to pick up whether you’re a beginner in programming or an experienced one for other languages. It’s a fast, friendly and easy to learn language but don’t mistake that for its powerful nature.

Chapter 2
Installation of Python

In order to install Python on to a machine, you must download the following:

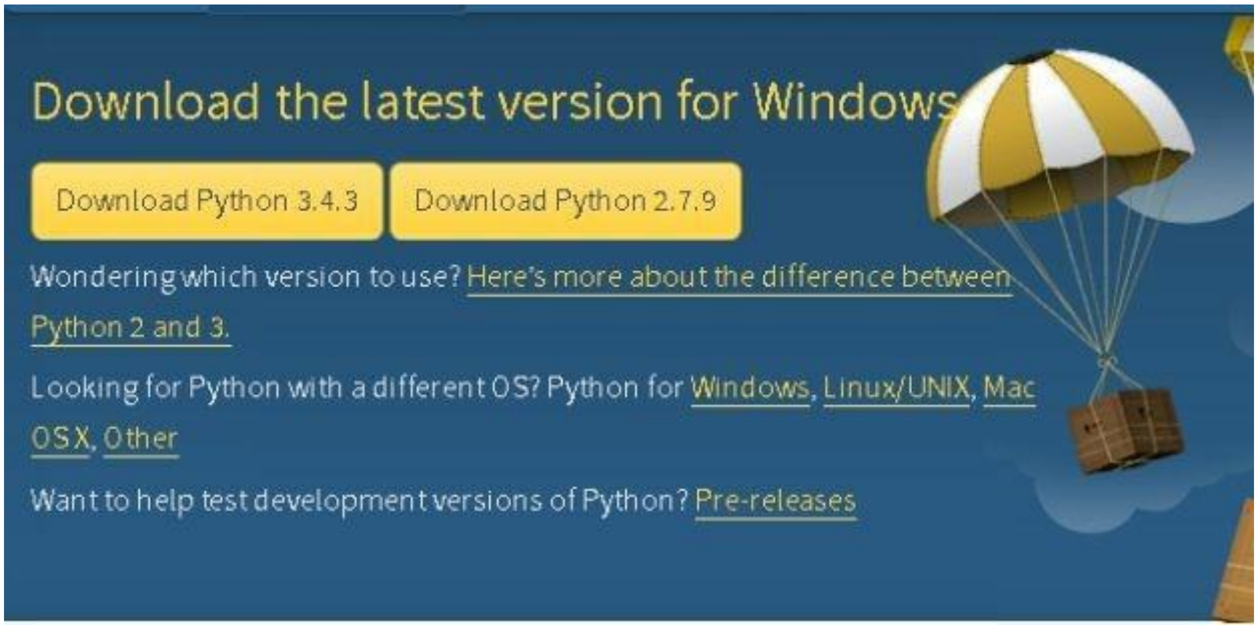
1. Python Interpreter
2. Python IDE

The download of these two tools will put you on your way to becoming a Python programmer. An IDE (integrated development environment) is a packaged application program used by programmers because it contains necessary tools in order to process and execute code. An IDE contains a code editor, a compiler, a debugger, and a graphical user interface (GUI). There are many different type of IDE's but the most commonly used one is PyCharm for Python. Before attaining the PyCharm IDE from JetBrains, you must first install the Python Interpreter (IDLE version 3.4) on to your system so PyCharm is able to detect Python on the computer.

In this guide, it would be recommended to use PyCharm because it will be used by this guide. If you use the same IDE, you can follow me easier. In order to download the Python Interpreter, please use the following link:

https://www.python.org/downloads/

Once you have reached this link, you will have to find this:



Download the latest version for Windows

Download Python 3.4.3 | Download Python 2.7.9

Wondering which version to use? Here's more about the difference between Python 2 and 3.

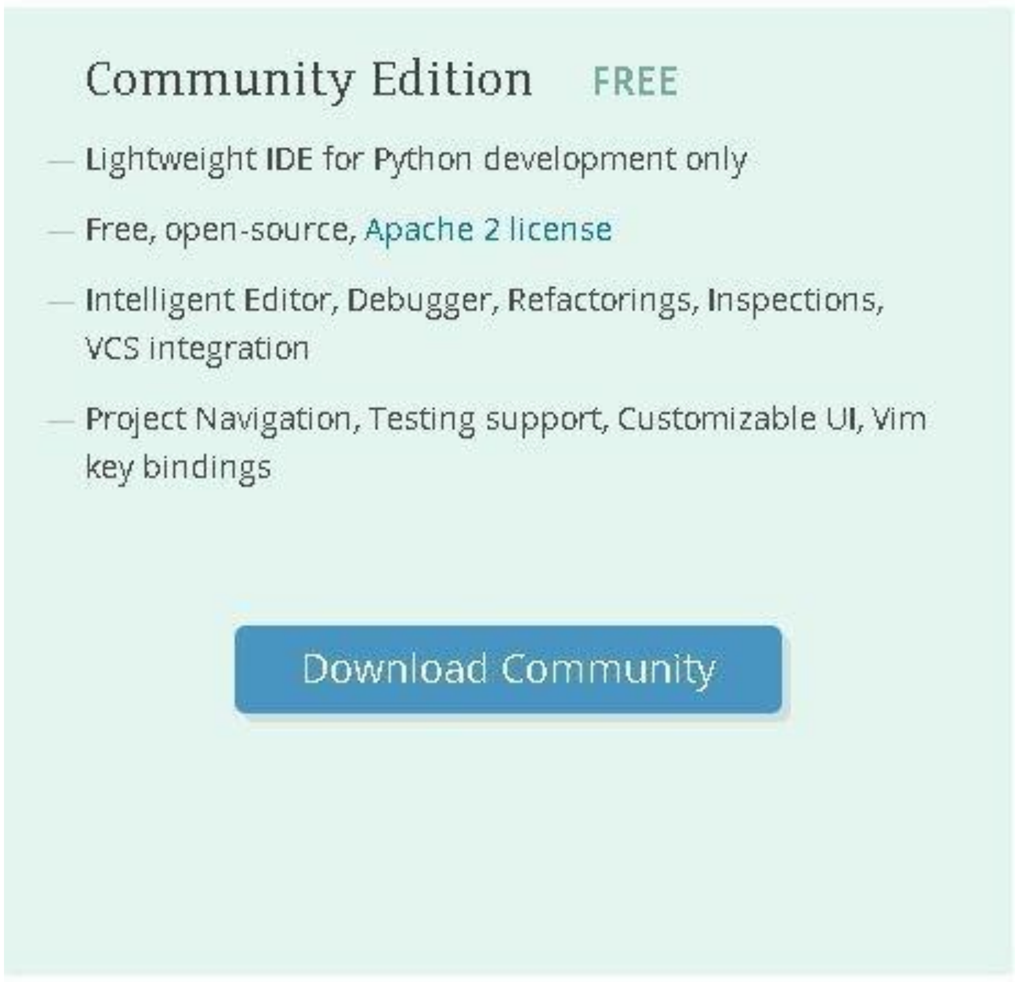
Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OSX, Other

Want to help test development versions of Python? Pre-releases

Then, click download and a file will be downloaded accordingly. Once you have this file, execute it and follow the instructions on the wizard to install the Python Interpreter.

Once the Python Interpreter has been installed, we can move on to downloading PyCharm. In order to install PyCharm, please visit the following link:

https://www.jetbrains.com/pycharm/download/


Community Edition FREE

— Lightweight IDE for Python development only

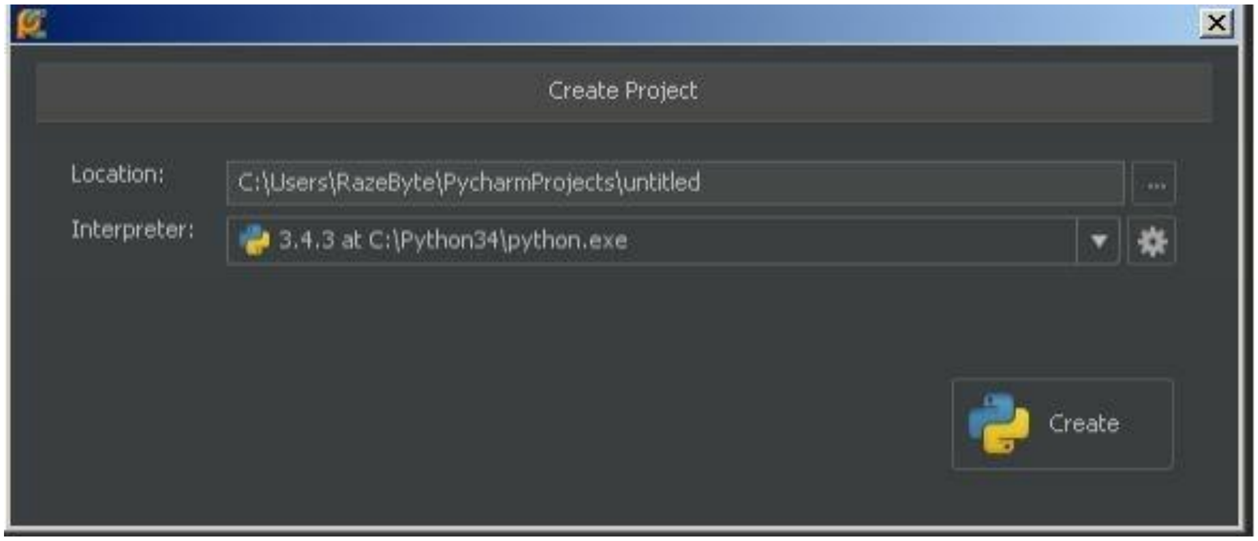
— Free, open-source, Apache 2 license

— Intelligent Editor, Debugger, Refactorings, Inspections, VCS integration

— Project Navigation, Testing support, Customizable UI, Vim key bindings

You should now be on this screen (or something similar) – click the “Download Community” button and continue to download PyCharm. During this process, it will automatically detect the Python 3.4 IDLE but in rare cases you might need to specify the directory that you installed Python in.

Once you have completed the installation of PyCharmand the Python 3.4 IDLE, go to File → New Project. You will reach this screen:



Location: C:\Users\RazeByte\PycharmProjects\untitled

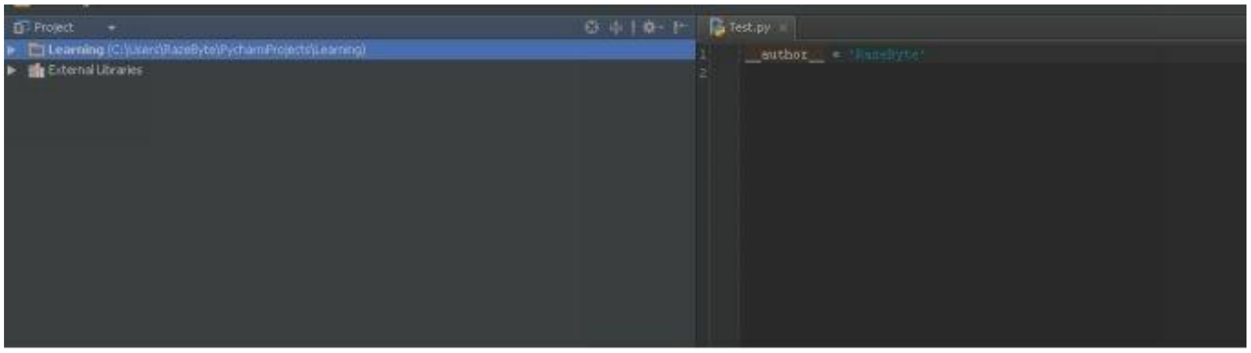
Interpreter: 3.4.3 at C:\Python34\python.exe

You can rename the “untitled” portion of the Location field to a specific project name. Then click the “Create” button and you will end up being in an empty project. The next thing we must do is create a new Python file which you can do by going to:

File → New → Python File

Set the name of your python file accordingly and it will show under your project name in the Project Explorer. Now double click on the file and you will be met with a page where you can start coding Python.

This is how it should look like:



```
__author__ = 'RazeByte'
```

You have successfully completed the installation of Python!

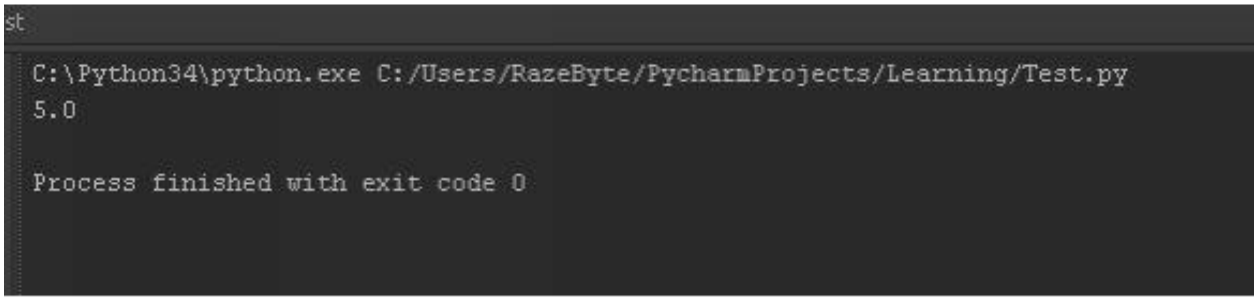
Chapter 3

Python Language Structure

Using the sample code as an example, we will learn how Python as a programming language is structured through the following sample code:

```
import math

class Program:
    def Execute(self):
        x = math.sqrt(25)
        print(x)

run = Program()
run.Execute()
```

Output:



```
st
C:\Python34\python.exe C:/Users/RazeByte/PycharmProjects/Learning/Test.py
5.0

Process finished with exit code 0
```

In order to actually execute the code, you must use the shortcut alt + shift + f10 or execute it using the Run menu item in the main navigation bar in PyCharm.

The first line import math is using a special keyword **import** which allows the programmer to import tools from the Python library which aren't included by default when coding. After the keyword **import** - the programmer specifies a specific directory that is within the Python library. In this case, we specified the math directory, if we wanted to specifically specify something within the System directory, we would add a dot separator (period) and then added a sub directory name. If the library's folders are more intricate, you must use notation like the following:

```
import bs4 from BeautifulSoup
```

The BeautifulSoup library is an example of this where we want to import the bs4 directory from the BeautifulSoup library of directories, which is why the "**from** " keyword is used in the beginning of the program.

In our case, we imported the math module because it contains functions (you will learn more about what these are later on in the guide – they can be defined as functions or methods). For the sake of a brief explanation, a function is simply a block of code that has a series of steps to complete a certain function and can be called to compute numbers for us. Many libraries contain functions that are able to be called so we don't have to code our own functions from scratch. In this case, we used the square root function from the math module. In this case, we used math.sqrt() and then printed out the value it returned relative to the input it received.

We first typed in math to indicate the module we are used and then a dot separator to reference the function within the module which is the sqrt function. The sqrt function takes in a value in its parameter or parenthesis. While the sqrt function computes the number 25 and returns the value of 5, this value is stored in the x variable which is why it has an equal sign to show that it is equal to the value that is computed from the sqrt function. Again, this might not make a lot of sense to you but will be covered in depth later on in this guide.

Now as we run through each statement, one thing to note is that in order to run the next line, we must have a separate line for each statement. The compiler (process to convert code in to readable code that the computer can understand; binary) will then know when statements end and when they start through the use of line breaks.

Now let's back up and talk about the class Program: line.

class - A class can be thought of as a "section" of code. For example:

Section: Everything in here is the contents of the section. Again, you will gain a better understanding how classes are useful in Inheritance and Polymorphism.

Program - This second element of this important line is "Program" which simply is a custom name that the user can redefine. You can call this anything, as all it is doing is giving the "Section" or "Class" a name. You can think of it this way:

Section Name:

    Essay/Contents

In code, it would be presented:

```
class Program:

    Code
```

Another thing you must be scratching your head looking at is the colon: ":" and "}" - all that the colon does is simply tells the compiler when a specific section starts. This could be thought of as when someone is writing an essay and they have to start their sentences with the appropriate words or indent their paragraphs.

One thing to note is that the spacing in code does matter. If you are creating a class, the content of that class will be indented once using the tab key in order for it to interpret that the code that the class has authority over is the tabbed code underneath it. Any code that is not tabbed underneath class is not part of that class. This goes the same for any function you create as well. As shown in the above example, a function is defined with a colon and tabbed content underneath it to indicate that it is part of that function. The way Python works is that it has classes, which are sections and functions which are subsections of the class. These classes can be declared as they are declared in the main program after the class is declared and then called with the functions that they carry within them. In this case, it is Execute. Again, this might not make sense right now but it will later on in the guide.

Chapter 4

Python Variables

What is a Variable?

Variables are created to reserve memory locations and to store values. When you create a variable, you are reserving space in memory. Depending on the type of variable, the interpreter will allocate some memory and decide what type of value can be stored within the reserved memory. Depending on the type of variable you create, you can store integers, characters, or decimals within it.

Assigning Variables

Unlike other programming languages, Python does not need specific declaration of the data type to reserve memory space. Values are assigned to variables with an equals sign (“=”). The operand to the left of the equals sign is the name of the variable, while the operand to the right of the equals sign is the value assigned for that variable to store.

Example:

```
age = 3  # we assign the variable age a integer value of 3
grade = 92.5  # we give the variable grade a floating point value of 92.5
mood = "happy"  # we give the variable mood an string value of "happy"

print(age)
print(grade)
print(mood)
```

In this example, we get three variables, “age”, “grade” and “mood”, and assign them values of 3, 92.5, and “happy”, respectively. Notice that each variable contains a different data type, however, we never had to explicitly declare each of their data types.

You can also assign multiple variables a value at the same time.

Example:

```
a = b = c = 5

print(a)
print(b)
print(c)
```

The following example gives a, b, and c the value of 5, all at the same time.
You can also assign multiple variables on the same line with different values and even different data types:

Example:

```
a, b, c = 1, 2, "example"

print(a)
print(b)
print(c)
```

In the example above, we can declare 3 variables each separated with a comma, and then declare their values on the same line, separated as well by commas, in the same order. The result is a much more compact way to declare your variables. How you choose to declare variables is up to you, I’d recommend doing whatever you feel is the most simple to understand.

Variable Types

In python, each variable has a specific type, which would determine what size it is, and the layout of its memory. The python language defines 2 types of variables:

Instance Variables

Instance variables are declared within a python file, but outside of any method, constructor or block. Instance variables are created with the keyword “new” and is destroyed when the object is destroyed. An instance variable is visible to all functions, constructors and blocks, within the file that it is declared in.

Example:

```
num = 3

def dog():
                print(num)

dog()
```

In this example, we declare the integer “num” outside of any function, and can easily access it within any function that is inside of the same file as “num”.

Local Variables

Local variables are only declared within functions, blocks or constructors. They are only created when the function, block or constructor is created, and then destroyed as soon as the function ends. You can only access local variables within the function, block, or constructor it is called, and they are not visible outside of where it is called. Local variables do not have a default value.

Example:

```
def dog (): # function named dog

      x = 3 # integer variable with value of 3
      print(x) # prints the variable "x"
```

In this example, A variable named “x” is called within the method “dog”. That variable only exists within the context of “dog”, and cannot be called or accessed directly outside of that method.

Example 2:

```
def dog (): # function named dog

      x = 3 # integer variable with value of 3
print(x) # prints the variable "x"
```

In this example, we print the value of “x” outside of the function called “dog”. This code will return an error, because the variable x does not exist outside of the context of “dog”.

Data types

When you create a variable, the data stored within it can be of many different data types. For example, you can store a car's speed with a number value, and then store the name of his car, using a string value. Python has multiple different data types that you can use to store different types of values, each of which will have its own set of unique operations that you can use to manipulate and work with the variables.

Python has 5 standard Variable Types:

Numbers

Number data types store numbers. They are created when you create a variable that holds a numerical value.

Example:

```
cat = 3
dog = 5
```

Here, we assign the variable "cat" a number value of 3, and "dog" a number value of 5. Python automatically allocated the memory as a number value.

Python supports 4 types of number values:

int: the int data type is a 32-bit signed two's complement integer. Its maximum value is 2,147,483,647 (2^{31} -1), and its minimum value is - 2,147,483,648 (-2^{31}). Int is the most common used data type for integral numbers, unless memory is a concern.

Example:

```
i = 1 # has a value of one
```

Long: the long data type is a 64-bit signed two's complement integer. Its maximum value is 9,223,372,036,854,775,807 (2^63 -1), and its minimum value is -9,223,372,036,854,775,808 (-2^63). This data type is used when a larger number is required to work with than would be possible with an int.

Example:

```
longSample = 12351235L   # has a value of 12351235
```

Float: The floating point data type is a double-precision 64-bit IEEE 754 floating point. The min and max range is too large to discuss here. A float is never to be used when precision is necessary, such as currency.

Example:

```
f = 1200.5 # value of one thousand five hundred, and a half
```

Complex: Complex numbers consist of ordered pairs of real floating-point numbers represented by x + yj , where x is the real floating number, while yj is the imaginary part of it.

Example:

```
complexSample = 3.14j
```

Strings

A string is identified as a continuous set of characters within a set of quotation marks. Unlike java, python allows strings to be declared with in single (‘ ’) and double (“ ”) quotes. You can also “splice” strings using square brackets ([] / [:]) and also add and multiple strings.

Example:

```
cat = "cat"  # declares value with double quotation marks
dog = 'dog'  # declares value with single quotation marks
addingAnimals = dog + "mouse"  # adds value of dog with string "mouse"
multiplyAnimals = dog * 3    # sets the value to dog multiplied by three, returning
"dogdogdog"

spliceDog = dog[1]  # returns the value of dog, at the index of 1
spliceCat = cat[0:2] # returns the value of cat, from the index of 0, to 2.
```

When splicing strings, you simply add square brackets followed by the index you want from the string. If you have only one number, it will return only the value of that index (see example of spliceDog). If you have two numbers, if will return the value from the index of the first number, to the index right before the second number. For example, in spliceCat, the value will be “ca”, because it starts at index 0, which is “c”, and goes in till the index of two, but it does not include the index of two, therefor it will end at “a”, giving a value of “ca” for spliceCat. When splicing you can also do the following:

```
first = "I am a set of strings"
second = "I am the second set of strings"

print(first[:5])  # print everything before index 5

print(second[5:])  # print everything after index 5, including 5
print(second[:])  # prints everything
```

By having no number before or after the “:” symbol, it will print all values before or after the other value. In the example above, “print(first[:5]) “ will print all values of first from the first index up in till the index of 5. In the line “print(second[5:]) “ it will print all values of second from the index of 5, all the way to the end. The final line “print(second[:]) “, will print all of second, as no numbers where included, so were basically saying print from beginning to the end.

Lists

A list contains multiple items separated by commas surrounded by square brackets. Lists are very similar to Array in other programming languages such as java of c#, but with one major different; you can have a list hold multiple different types of data, unlike arrays which can contain only one data type per array. Values in Lists can be accessed the same way that we spliced strings up above. You use square brackets and numbers separated by a “:” symbol to choose which values you want to work with.

Example:

```
firstList = ["first", 2, "third", 4, 5]
secondList = ["hello", 3, 4, "bye"]

print(firstList) # prints all of firstList
print(secondList) # prints all of secondList
print(firstList[1:]) # prints all of firstList from index 1 to end
print(secondList[1:3]) # prints all of secondList from index 1 to index 3
print(firstList * 2) # rpints firstList twice
print(firstList + secondList) # prints firstList and secondList
```

You can interact with lists in the same way that you can with strings. You can add lists together, multiply a list and even get substrings of a list.

Tuples

A tuple is a data type similar to lists. Tuples can also contain different types of data separated by commas, however, instead of being enclosed by square brackets, they are enclosed by parentheses ("()") and they cannot be updated. Tuples can be thought of as lists that can only be read, unlike lists which you can read and write too.

Example:

```
firstTuple = ("first", 2, "third", 4, 5)
secondTuple = ("hello", 3, 4, "bye")

print(firstList)
print(secondList)
print(firstList[1:])
print(secondList[1:3])
print(firstList * 2)
print(firstList + secondList)
```

Notice how the above example is near identical to the list example, the only difference being that the square brackets were changed to parenthesis. This code will write to console identically to the list code, however, if we were to try to change the value of one of these lists, we would get an error.

Example:

```
firstList = ["first", 2, "third", 4, 5]
firstTuple = ("hello", 3, 4, "bye")

firstList[2] = 1

firstTuple[2] = 2
```

In The code above we create a list and a tuple. The list we change the value of index 2 to 1 successful, but when we try to change the value of index 2 of the tuple we will get an error, because tuples do not support that action like lists do.

Dictionary

Dictionaries consist of key-value pairs. A Dictionary consists of a name, followed by a key, which then hold a certain amount of values. A key is surrounded by square brackets ([]) while values are surrounded by curly braces ({}). A dictionary works in a similar way to a real dictionary. You can sort of think of a key as if it were a word, and then the value as the definition of the word, or the value that the word holds. A key can be almost any python data type, but are often words or numbers, while the values held by a dictionary can be anything.

Example:

```
dict = {} # declares an empty dictionary
dict["first"] = {3}
dict["second"] = {"I am the value of key second"}

secondDict = {"firstName": "Bob", "lastName": "Smith", "age": 25}

print(dict["first"]) # prints the value of key "first"
print(dict.keys()) # prints all keys

print(secondDict.keys()) # prints all keys
print(secondDict.values()) # prints all values in secondDict
print(secondDict["firstName"]) # prints the value of the key "firstName"
```

In this example, we show multiple ways we can work with dictionaries. With the variable "dict" we declare each key and value separately. As you can see its very similar to a real dictionary, where we essentially create a word and attach a value to it. Note that the key can be a number too, this example only uses string keys however. The second dictionary "secondDict" declares multiple keys and values on one line. When you do this, within the curly brackers, the first value represents the key, then separated by a ":", you give the keys values. You then separate each key and value by a comma, and can add as many keys as you wish.

Converting Data Types

When writing code you may have to work with multiple variable types that you'll need to combine or use together. In situation like this you will need to convert data types to the same type in order to be able to combine them. Luckily, there are multiple built in methods that allow us to convert variables very easily. Here are just a few:

int(x, [,base]) : X is the value converted to int, while base represents the base if x is a string

float(x): Converts x to a float.

str(x): Converts X into a string.

tuple(x): Converts X into a Tuple.

list(x): Converts X into a List.

dict(x): Converts X into a dictionary.

Long(x, [,base]): Converts X into a Long. Base represents the base if x is a string.

Chr(x): Converts X into a char.

Assignment

Using what we learned about variables, we can now create a simple calculator to add numbers together for us. The first thing we will want to do is create 3 variables: one to store the value, one to represent the first number we want to add, and one to represent the second number we want to add. We will declare these variables as Doubles so that we can add decimal numbers:

```
a = 3 # creates first variable to add
b = 4 # creates second variable to add
c = 0 # create variable to store values
```

Next, we will simply set the value of c, to equal the value of a and b combined, and then print out the value of c.

```
c = a + b # sets value of c to value of a + b

print("the sum of a plus b is ", c) # prints the sum
```

If you run the program now, it should print out 7. You now have just created a very simple calculator. I highly encourage you to play around with this, and test things for yourself. Change the data type of the variables, add more numbers together, and experiment to understand how things work.

Chapter 5

Python Operators

Just like in Math class, you learned about addition, subtraction, multiplication, division, etc. These are all arithmetic operators that are used within programming to intake numbers, process them, and calculate them accordingly. Let’s go over these operators in programming, as they are one of the most important things to understand and also one of the easiest to grasp.

The Arithmetic Operators

Addition

```
5+5 = 10
x = 5;
y = 5;
sum = 0;
sum = x + y;
```

In the example above, a variable of sum is taking the addition of two variables (x and y) and adding them together to produce a value of 10.

Subtraction

```
10-5 = 5
x = 10;
y = 5;
total = 0;
total = x - y;
```

In the example above, a variable of total is taking the subtraction of two variables (x and y) and subtracting them together to produce a value of 5.

Multiplication

```
5*4 = 20
x = 5;
y = 4;
total = 0;
total = x * y;
```

In the example above, a variable of total is taking the multiplication of two variables (x and y) and multiplying them together to produce a value of 20.

Division

```
20/5 = 4
x = 20;
y = 5;
total = 0;
total= x / y;
```

In the example above, a variable of total is taking the division of two variables (x and y) and dividing them together to produce a value of 20. If you were to take the number 35 and divide it by two, as you may know, it will produce a value that is not whole, but rather decimal. If you wanted to produce a value that is whole regardless of the decimal (automatically rounding down regardless of the decimal) – then you must use a double slash like the following:

Integer Division

```
35//2 = 17
x = 30;
y = 2;
total = 0;
total= x // y;
```

```
Output: 17
```

Modules

```
7 % 2 = 1
x = 7;
y = 2;
total = 0;
total = x % y;
```

In the example above, a variable of total is taking the remainder of two variables (x and y) and by finding how many times 2 multiplies in to 7 evenly before it can’t. After that processes, the remainder is the output. For example:

> How many times does 2 go into 7 evenly?
> 3 times.
>
> 2 * 3 = 6
> 7 - 6 = 1

Therefore, 7 modules 2 is equal to 1 because that is the remainder.

The Assignment Operators

The assignment operators are operators that are used when assigning variables values, the most commonly used operator being (=). Here is a list of examples:

Equal Sign: =

```
x = 5;
```

In this example, if you were to print out x, the value would be 5 because you assigned the variable x equal to 5.

Add-Equal Sign: +=

```
x = 5;
    x += 5;
```

In this example, if you were to print out x, the value would be 10 because you are adding the value of 5 onto the value of x. This statement is the same thing as saying x = x + 5 → x += 5.

Subtract-Equal Sign: -=

```
x = 5;
    x -= 5;
```

In this example, if you were to print out x, the value would be 0 because you are subtracting the value of 5 from the value of x. This statement is the same thing as saying x = x - 5 → x -= 5.

Multiplication-Equal Sign: *=

```
x = 5;
    x *= 5;
```

In this example, if you were to print out x, the value would be 25 because you are multiplying the value of 5 onto the value of x. This statement is the same thing as saying x = x * 5 → x *= 5.

Division-Equal Sign: /=

```
x = 5;
    x /= 5;
```

In this example, if you were to print out x, the value would be 1 because you are dividing the value of 5 onto the value of x. This statement is the same thing as saying x = x / 5 → x /= 5.

Modules-Equal Sign: -=

```
x = 5;
    x %= 5;
```

In this example, if you were to print out x, the value would be 0 because you are finding the remainder in (5/5). This statement is the same thing as saying x = x % 5 → x %= 5.

Chapter 6

User Input

When writing code, you might run in the many times where you want the user to give the program a value for the program to work with. So far, what we would do is just use a variable and change it around, but the average user doesn't have access to code, nor would they understand what to do with it. That's why you can have the user of the program input a value through the program. The input can be any character and be used in any way that you wish. This can all be done with the input function. It is used as follows.

```
varName = input("words")
```

In this case, varName is a variable, set to equal the keyword input, followed by the words that you would like displayed in console. The result is simple.

Example:

```
age = input("what is your age?")

print(age)
```

How this works is, when the code runs, it will display in console what you put inside the brackets of input. After that is displayed you must write an input into the console and press enter. Whatever you wrote in console, then becomes the value of age. This can be tested and witnessed with the above code.

Example 2:

```
name = input("what is your name?")

print(name)
```

Input works the same way with strings.

Example 3:

```
names = [input("firstname"), input("secondname")]

print(names)
```

Input even works with lists. Input works with anything where the value you will input is a valid data type for the variable.

Assignment

Create a program where you ask the user how many people they want to input. Then ask them the name of each person and each person's age. Then output all names and ages organized, the average age, the youngest age, and the oldest age.

Solution

```
amount = int(input("how many people")) # sets amount to user input

people = {} # creates an empty dictionary

average = 0 # sets average

# for all numbers in the range of zero to amount
for x in range(0, amount):
        name = input("what is the persons name? ") # set persons name to inpt
        people[name] = int(input("age?")) # creates a dictionary key of person name, with value age, from input

# for all values in all values in people dictionary
for ages in people.values():
        average += ages # add age of each person to average

print("the average age of all the people is ", average / amount) # divides average by amount of people to get real average

print("the oldest person is ", max(people.values())) # min function gets smallest number in values

print("the youngest person is ", min(people.values())) # max function gets largest number in values

print() # print blank line

print("all the people are: ")

print(people) # prints entire dictionary
```

Chapter 7

Strings in Python

Although we have already gone through strings in the variable section, there is still much more to talk about the topic. Strings are quite commonly used in Python programming or any other programming language for that matter. A string is a sequence of characters, or a line of text. A string is like an object, meaning that it has properties that be called from it.

The way to declare a string variable is by doing the following:

```
x = "A string!"
```

As you can see, we have surrounded the text with quotation marks to indicate to the Python interpreter that it is a string and not an integer or any other value other than a string. Now let’s go over the fundamental properties of a String object that can be used to gain extra information over the piece of text or string.

String Length

A way to receive information on the amount of characters in a String is by doing the following:

```
test = "test"

amount = test.__len__()

print(amount)
```

The .__len__() function in a String object returns a value for the amount of characters within the String object and assigns the value to the integer amount. As seen, the dot property is used to access certain functions that are part of the test string object. The output in this sample code would be 4 since “test” is 4 characters long. An alternate method of this could simply to surround the test variable with the len function like so:

```
test = "test"
amount = len(test)
print(amount)
```

Concatenating Strings

Concatenating is the idea of joining two strings together. There are different ways of joining two strings together. Here are a few examples:

Example 1:

```
text = "Hello"
text2 = " World"
print(text + text2)
                    Output: Hello World
```

Example 2:

```
text = "Hello"
text2 = "World"
print(text + " fun " + text2 + " of programming!")
                    Output: Hello fun world of programming!
```

As shown in the examples above, you are able to use the "+" operator to join two or more strings together.

Here are various examples of utilizing the String object:

Example 3 – string[int index]:

```
text = "Hello World"
letter = text[0]
print(letter)
```

The value of "letter" contains the letter "H" because the inputted index of the string is 0 which is the first letter of the string. The way the computer reads the string, is from 0 to the length of the string, not 1. To the computer, a string is a list of characters, so indicating a square bracket with an index is like referring to a specific index of a character in a list of characters when really it´s a string variable.

Example 4 - Equals(String anyText):

```
text = "Hello"
doesItEqual = text is "Hello"
print(doesItEqual)
```

The value of doesItEqual is going to equal true in this case because the is keyword returns a bool value (true or false) and the text does indeed equal to Hello. Later on, you will learn about "if" statements, which can allow you to check whether or not something equals to something else. When comparing two strings, it is recommended to always use the is keyword, but when comparing most other data types you will use "==". You will learn more about this later on.

Example 5 - string[x:y]

```
text = "Hello World"
```

```
justWorld = text[6:]
print(justWorld)
```

Since the position of the letter “W” is 6, if you say :6 - it will essentially divide that text starting from that position. Therefore, the “justWorld” variable ends up containing the value of “World”. The colon can be used to do many cool things. If you set the left side of the colon to a value and the right side as a value, it will splice the string from the 6th index to another index depending on what you specified. For example:

```
text = "Hello World"
justWorld = text[6:8]
print(justWorld)
```

In this case, it will print out “Wo” because the position of W is 6 and the position of O is 8. You can also use negative numbers which will make it go backwards starting from the end of the string instead of the beginning. The technique used in the first example is checking all the values after the 6th index. You can also do [:6] which will print out Hello instead.

Example 6 - strip():

```
text = "  Hello  World  "
trimmedText = text.strip()
print(trimmedText)
```

As seen, the text variable has spaces in the beginning and the front. All the strip() method does is simply delete any spaces on the left or right side of the string of text. Therefore, the trimmedText would hold the value of “Hello World” instead of “   Hello World “.

There are many other String methods that can be used to manipulate a String and can be referenced using the Python API. An API is simply a directory of all the methods/functionalities and objects/classes of the Python platform that can be used to help the user program. It can be referenced here:

https://docs.python.org/3/c-api/index.html

Chapter 8

Boolean Logic

Python provides us a vast set of operators to manipulate variables with. In the following section we are going to go over Relational Operators and Logical Operators. Before we can go to understand how to use these operators, we need to know where we can use them. There are many different **statements** that can be used, in order to perform and test different types of logic. We will go over some more complex statements in later sections.

If Statements

In order to show the following examples, I will use an “If” statement. An if statement, simply put, checks to see **IF** something is true, or false. To create an “if” statement, you simply write the word “if” followed by the logic to check whether **IF** it’s true or not. If statements can also include elif statements, and else statements. An elif statement will run if the “if” statement returns false, however, it also contains its own parameter that must return true. An else statement also is called if the if statement returns false, the difference between an else statement, and a elif statement, is that no parameters need to be true for an else statement to run.

Example:

```
a = 10
b = 20

if a == b:
        print("a is equal to b")
elif a > b:
        print("a is greater than b")
else :
        print("a is not equal to b or greater than b")
```

The operands used in this example (“==” and “>”) are explained below. In the following code, we first check to see if a equals to b. if that is true, we simple print “a is equal to b”, and ignore the rest of the code. If a does not equal b, then it goes down and called else if, with else meaning if the first if returned false, and if is checking its own parameters, which happen to be whether a is greater than b. if a was greater than b, then we would have printed “a is greater than b”. But since a is less than b, we go further down and simple call else. Else runs if all the above parameters return false, and does not require its own parameter to be true. **Note** That each if statement can have an infinite amount of else if statements follow it, but only one else statement. Also **Note** that you cannot have code between an if statement and an else statement, because the code would not be able to find the else. As well, an else statement requires there to be an if statement before it, since for an else statement to be called, an if statement HAS to have returned false immediately before it. It is also important to know that it is possible to have an IF statement INSIDE another if statement. This is called a **NESTED** if statement

Example:

```
a = 10
b = 20

if a == 20:

        if a < b:

            print("a is less than b and equal to 20")
```

In this example, we first check to see if the value of a is equal to the value of b. If it is, we then check to see if the value of a is less than the value of b. If this statement also returns true, we can print to the console “a is less than b and equal to 20”. This can work with as many if statements as you would like. This can be useful if completely necessary to check the parameters separately, or if you want to add additional logic between each check.

You will understand fully the way an if statement works by the end of this section.

The Relational Operators

Python supports 6 different types of relational operators, which can be used to compare the value of variables. They are the following:

== This Operator checks to see if the value of two integers are equal to each other. If they are the same, the operator returns true, if not, it returns false.

Example:

```
a = 10
b = 10

if a == b:
        print("they are the same")
```

This code checks to see **IF** The value of a is equal to the value of b. because the value of a is equal to the value of b, the value within the brackets will run, causing the above code, to print out “they are the same”. Note that the operator can also be written as “is” instead of “==”.

Example:

```
a = 10
b = 10

if a is b:
        print("they are the same")
```

!= This operator checks to see if the value of two things DO NOT equal the same thing. If two things DO NOT equal the same thing, then it will return true. If they DO equal the same thing, it will return false.

Example:

```
a = 10
b = 20

if a != b:
        print("they are not the same")
```

This code will check if A does not equal B. Since A does not equal to B, the program will print “They are not the same” in the console. The operator can also be written as “is not” instead of “!=”.

Example:

```
a = 10
b = 20

if a is not b:
        print("they are not the same")
```

> This operator Checks to see if something is **Greater than** something else. If the value of the variable in front of it is greater than the value of the variable after it, it will return true. If the value of the variable in front of it is less than the value of the variable after it, it will return false.

Example:

```
a = 10
b = 20

if a > b:
        print("a is larger than b")
else:
        print("a is not larger than b")
```

This example checks to see if a is larger than b. since a is not larger than b, this if statement will return false, and instead will print “a is not larger than b”, because the code passes the failed if statement and calls the else statement.

< This operator checks to see if something is **Less Than** something else. If the value of the variable before the operator is less than the value of the operator after the variable, than the code will return true, else, it will return false.

Example:

```
a = 10
b = 20

if a < b:
        print("a is smaller than b")
else:
        print("a is not smaller than b")
```

This Example is just like the one before it, however, note that the operator changed from Greater than, to Less Than. Therefore the if statement will return true this time, because the value of a is less than the value of b, and the program will print to the console “a is smaller than b”.

>= This operator checks to see if something Greater than **Or Equal** to something else. If the value of the variable before it is greater than **Or Equal** to the variable after it, than it will return true. If the variable after it is greater but NOT EQUAL to the variable before it, it will return false.

Example:

```
a = 20
b = 20

if a >= b:
        print("a is larger or equal to b")
else :
        print("a is not larger or equal than b")
```

In this example, A and B both have a value of 20. Although a is not greater than b, it is EQUAL to b, therefore the statement returns true, and the code will print out “a is larger or equal to b”.

<= This operator checks to see if something is less than **Or Equal** to something else. If the value of the variable before it is less than **Or Equal** to the variable after it, than it will return true. If the variable after it is less but NOT EQUAL to the variable before it, it will return false.

Example:

```
a = 20
b = 20

if a <= b:
        print("a is larger or equal to b")
else :
        print("a is not larger or equal than b")
```

This example is identical to the one before it except for the operator was changed from >= to <=. Although the operator has changed, the result is the same. Since a is equal to b, this code will also return true, printing “a is larger than b” to the console.

The Logical Operators

Java supports 3 Logical Operators that can be used in your logic, which will often be used in conjunction with Relational Operators.

& This is known as the logical AND operator. If the operands before and after this operator both return true, then the condition returns true. If both of the operands are false, or only one operands is false, the condition will return false. **BOTH** operands MUST be true for the condition to return true.

Example:

```
a = 20
b = 20

if a == b & a == 20:
        print("a is equal to 20 and b")
else :
        print("a is not equal to 20 and b")
```

In This Example, we check to see if “a” is equal to 20, AND if “a” is equal to “b”. These two conditions will be referred to as operands. Since A is equal to 20, and equal to “b”, the statement returns true, and we print to the console “a is larger than b”. Note that we can also write the operator as “and” instead of “&”.

Example 2:

```
a = 20
b = 20

if a == b and a == 20:
        print("a is equal to 20 and b")
else :
        print("a is not equal to 20 and b")
```

Both operators do the same thing and which you use is up to personal preference.

Example 2:

```
a = 20
b = 30

if a == b and a == 20:
        print("a is equal to 20 and b")
else :
        print("a is not equal to 20 and b")
```

In This next example, we check the exact same thing as the first example. The difference this time however, is that we changed the value of b from 20 to 30. This means that when we run this code, the if statement will return false because although “a” is equal to 20, “a” does not equal to “b”, therefore we print out “a is not larger than b”. Note that the order does not matter. If the first operand was false instead of the second, we would have the same result.

| This Operator is known as the logical OR operator. If the first operand **OR** the second operand is true, the statement will return true. This means that if the first operand returns false and the second is true, the statement will return true, and vice versa. The statement also will return true if both the operands are true. Essentially, when using an OR operator, at least one Operand must return true.

Example:

```
a = 20
b = 30

if a == b | a == 20:
        print("a is equal to 20 or b")
else :
        print("a is not equal to 20 or b")
```

In this example. We check to see if “a” is equal to 20, OR if “a” is equal to b. In this case, a is equal to 20 and is not equal to b, But since only one of these operands need to return true, the statement as a whole will return as true, therefore the program will print “a is equal to 20 or b” in the console.

Example 2:

```
a = 20
b = 20

if a == b | a == 20:
        print("a is equal to 20 or b")
else :
        print("a is not equal to 20 or b")
```

In this example, I have changed the value of b from the previous example from 30 to 20. This means that “a” is equal to 20 and “a” is equal to “b”. Both operands return true therefore the statement returns true, because an OR operator require 1 or more of the operators to be true. This program will output “a is equal to 20 and b” in to the console. Note that this operand can also be written as “OR” instead of “|”, which is again, a matter of preference.

Example:

```
a = 20
b = 20

if a == b or a == 20:
        print("a is equal to 20 or b")
else :
        print("a is not equal to 20 or b")
```

Not This operator is known as the logical NOT operator. It is used to reverse the logical state of its operand. This operand can be used with any other operand to reverse its output. If an operand was returning true, after applying the logical NOT operator, the operand will return false, and vice versa.

Example:

```
a = 21
b = 20

if not (a is 20) and a > b:
        print("a is not equal to 20, and a is greater than b")
else :
        print("a is equal to 20, and a is not greater than b")
```

In this example, we check to see if the value of “a” DOES NOT equal to 20, and we check if “a” is greater than “b”. Since “a” is both not equal to 20, and greater than b, we can print out to console “a is not equal to 20, and a is greater than b”.

Example 2:

```
a = 21
b = 20

if not (a is 20 and a > b):
        print("a is not equal to 20, and a is greater than b")
else:
        print("a is equal to 20, and a is not greater than b")
```

Combining Operators

In Python, it is possible to use as many operators as you require per statement.

Example:

```
a = 21
b = 20

if (a is 20 and a is b) | (a is not 20 and a > b):
        print("the statement returns true")
else :
        print("the statement returns false")
```

With this example, we can see how we can achieve much more complicated statements. Here, we apply a OR operator on two operands, however, each operand, also incorporates an AND operator. Therefore, for this statement to return true, “a” must be equal to 20 and be greater than “b” OR “a” must not be equal to 20 AND “a” must be less than “b”. Note that we use brackets to make the code more easy to understand, and to prevent the code from misreading our logic, be applying operators in an incorrect order.

Example 2:

```
a = 21
b = 20

if a is 20 and (a > b or a is not 20) & a < b:
        print("the statement returns true")
else :
        print("the statement returns false")
```

In the above example, we have significantly changed the logic of the program, by only switching the position of a bracket. Notice how now the brackets surround the two inner operands, instead of two pairs of brackets surrounds each pair of outer operands. For this statement to return true now, 3 different conditions must return true. “a” must be equal to 20, AND, either a must be greater than “b” OR it must not be equal to 20, AND “a” must be less than “b”. with this new logic, the first operand must be true, on top of EITHER the second or third having to return true, and finally Also on top “a” must be less than “b”. This statement will return false. It will fail the first condition, because a does not equal 20. It will pass the second condition, because it is greater than b, and it will fail the last condition because a is not less than b. Since this would have required three incorrect conditions, and only had one, the statement returns false and prints “wrong”.

Assignment

We are now going to go through another assignment to make sure we understand everything about operators. We have 3 friends who want to know how much money they have compared to the others. We will make a program that will output who is richer than who. We will have three integers, named bob, john, and tom. We will give each one of them a different value, and the program must output who is richer, but it must also tell us if someone has the same amount of money as someone else.

For example:

If bob = 20, tom = 10, and john = 5, the output must be:
“bob is richer than tom, who is richer than john”
But if bob = 20, tom = 20, and john = 5, the output must be:
“bob is just as rich as tom, both are richer than john”

The program needs to work for each possible outcome. I highly encourage you to try this program by yourself and struggle through it as much as you can, before you look at the answer. A big part of programming is problem solving and understanding code, so I recommend you try to figure it out by yourself first.

Answer and Explanation

Note that the program will be explained through comments in the code, to make it easier to understand.

```
# not that i use boolean operators such as "is" and "=="
# interchangeably a lot, but which you use is up to you
# i prefer the word forms since its similar to just
# regular english and easier to read.

# feel free to change these values around
tom = 10  # amount of money that tom has
bob = 20  # amount of money that bob has
john = 10  # amount of money that john has

# we first check if to see if everyone has the same amount of money
# note how i never checked if tom is equal to john because if
# tom is equal to bob, and bob is equal to john, then obviously
# tom is equal to john
if tom is bob and bob is john:
        print("Everyone has the same amount of money")

# if the first statement returns false,
# I will next check the unique case where tom and bob have
# the same amount but john does not

elif tom is bob and bob is not john:

        # I now check if bob is greater or less than john
        # and then output the correct answer
        if bob > john:
            print("tom is just as rich as bob is, both are richer than john")
        elif bob < john:
            print("tom is just as rich as bob is, both are poorer than john")

# I repeat the same process as the previous statement but with different people
```

```
elif tom is john and john is not bob:

            if john > bob:
                print("tom is just as rich as john, both are richer than bob")
            elif john < bob:
                print("tom is just as rich as john, both are poorer than bob")

# now i check the last possible combinations
# where each person has different amounts of money
# the next 6 statements cover each possible outcome

elif tom > bob and bob > john:
            print("tom is richest, followed by bob, and followed by john")

elif tom > john > bob: # you compare 3 variables like this without using and, to same space and
time if you wish
            print("tom is richest, followed by john, followed by bob")

elif bob > tom and tom > john:
            print("bob is richest, followed by tom, followed by john")
elif bob > john > tom:
            print("bob is richest, followed by john, tom is the poorest")
elif john > bob > tom:
            print("john is the richest, bob is the second richest, tom is the least richest")
elif john > tom > bob:
            print("john is the richest, followed by tom, and bob is the least rich")
```

When writing code it is important to be able to comment all of your logic so that someone who has not written it can easily understand it and edit it. I recommend that when you write code you also comment it as much as you feel is required. I also encourage you to try to find a better way to solve this problem. Maybe there is a way you can write this in a quarter of the length, with programming, there is never only one answer.

Chapter 9

Loops, Tuples, and Dictionaries

Loops

When programming, you might run into a situation where you will need to loop through a large amount of numbers. If we used what we learned so far to loop through 100 numbers, we would need 100 if statements, which I think you can agree, would get really messy and frustrating. That is why python supports multiple type of loops. python has three types of loops that you can use, that all loop through things in slightly different ways. Below is a quick explanation of each one of them.

While Loop

A while loop is a control structure that will allow you to repeat a task as many times are you program it to. The syntax for a while loop is as follows:

```
while (expression):
        # insert code
```

This works in a similar way that if statements work, except, WHILE the expression is true, the code within the while loop will run infinitely, intill the expression becomes false (if it ever does).

Example:

```
x = 10

while x > 0:
        print(x)
        x -= 1
```

The Following code will print out the value of x, and then subtract it by one, continuously, intill the value of x is no longer greater that zero. If you were to run this code, you would get an output of every number from 1 to 10. Once the value of x reaches zero, it no longer allows the expression to return true, and there for the while loop finishes. **Note** that you should watch out for infinite loops. This is where an error in your code causes the loop to never end, essentially freezing the program. This would happen if there is an error in your logic, and the statement never become false.

For Loops

For loops allow you an easy way to loop or increment through a specific range of values. The syntax for a for loop is as follows:

```
        for (data in sample){
    //statement
}
```

Initialization: represents the data to be looped in

Sample: is the sample of data that will be looped through

Example:

```
listSample = [1, 2, 3, 4]

for numbers in listSample:
        print(numbers)
```

In this example, we create a list called “listSample” with the values 1 to 4. We then create a for loop to loop through the list. We create a variable called numbers, which will represent each number during the loop, and we tell it to loop in “listSample”. So what were saying is for the numbers represented by the variable “number”, loop through all the values inside listSample. The code will then run through each number inside listSample, and during the loop you can reference these numbers through the variable “numbers”.

Lists and Tuples

Lists

Lists are a data type used to hold a sequence of numbers. Each element inside a list is represented by an index, which tells you where the element is located within a list. Lists are very powerful tools that give you a wide array of functions to use on your data. With lists you have the ability to hold large amounts of data and have those all be represented in an organized manner, by one variable. You can then multiple, splice, subtract, add and index all this information into many meaningful things. One of the most useful things about a list is the fact that you can hold multiple different data types within one list. This is something that would not be possible with arrays in other languages such as Java and C#.

Updating and Accessing Lists

Earlier when we discussed data types, we talked about how to splice up lists a little and access then a little. Now we will go a little bit more In depth.

Example:

```
firstList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(firstList[:5])

firstList[2] = 10
print(firstList[:5])

firstList[:5] = 1, 1, 1, 1, 1
print(firstList[:5])
```

In the example above, I first print the values from index 0 to 5. **Note** that indexing always starts at zero in programming. Next I change the value of index 2 to 10, meaning that the number 3 will now turn to 10. Finally, I decided to change all values to 5 to the number one. Note that when you change multiple values in a list, you also need to give the same number of values. For Example, When I change index zero to five to the number one, I needed to give each index the value separately, which is why I wrote 5 ones separated with commas, each was changed individually.

Deleting List Elements

It is also possible to delete elements within a list. You can use the del statement or the remove function to delete elements within a list.

Example:

```
firstList = [1, 2, 3, 4, 5, 6, 25, 25, 9, 10]

del firstList[2]

print(firstList)

firstList.remove(25)
```

```
        print(firstList)
```

In this example, we use the del statement to delete the number at index 2. When we print, you'll notice that the number 3 is now missing. Del is used when you want to remove whatever is at the index specified, regardless of what it is. The second thing I did was use the remove function to remove the number 25. What remove does, is look through the list and removes the first thing it finds that I told it to remove. Since I told it to remove the number 25, it removes the first index of 25 that it finds. Note that when I print this it only removes one 25, as it only removes the FIRST index, not all. Another cool thing about lists is that unlike arrays, the size can dynamically change. When you remove an index it disappears, unlike in arrays in other languages where the value would have to become zero, or the array reinitialized.

List Operations

Below are a few common list operations that you might see later on in this tutorial.

```
firstList = [1, 2, 3, 4, 5, 6, 25, 25, 9, 10]

len(firstList) # returns the length of the list
firstList.reverse() # reverses the list
firstList[1] + firstList[2] # returns the value of the first two indexs combined
print(firstList[1] * 5) # returns the value multiplied by 5
```

Tuples

A tuple Is a sequence just like lists. The major difference between tuples and lists is that tuples cannot be changes, while lists can. Declaring a Tuple is very similar to declaring a list, the only difference being you use round braces instead of brackets.

Example:

```
firstTuple = ("cats", "dogs", "pig", "wolf")
print(firstTuple)
```

As you can see above, it looks near identical to a list, with only the round braces being the difference.

Accessing and Updating Tuples

The way you access tuples is the same way that you access lists. You simply use square brackets and state the index range of which you would like to access. Updating tuples however, is slightly different. Since you cannot edit tuples, you cannot simply state the index you wish to change like in lists.

Example:

```
firstTuple = ("cats", "dogs", "pig", "wolf")

print(firstTuple)
firstTuple[2] = "wolf" # this does not work

print(firstTuple[2]) # print index 2
print(firstTuple[:]) # prints all indexes
print(firstTuple[2:]) # print from index 2 to the end
```

In this example, we declare a tuple with 4 values. notice that when I try to change the value of index 2, you will get an error, since you cannot edit values of a tuple. What we can do however is view all the content of a tuple the same way that we can view the content of a list; using square brackets.

Example:

```
firstTuple = ("cats", "dogs", "pig", "wolf")

secondTuple = ("bob", "joe")

thirdTuple = firstTuple + secondTuple # makes third tuple the combination of first and
second tuple

print(thirdTuple)

del firstTuple
del secondTuple
```

In this example, we see how we can sort of edit tuples. Since we cannot actually edit tuples, we have to create a new tuple if you would like to add more values to a tuple. Above, if I wanted to add the values of secondTuple to firstTuple, I have to declare a new tuple and make it equal to the first and second tuple variables. This way we have successfully created a new tuple containing the values of both firstTuple and secondTuple, without ever having edited a tuple. I then proceeded to delete the first two tuples since I don't need them. Note that we can also do this without declaring a third variable.

Example:

```
firstTuple = ("cats", "dogs", "pig", "wolf")

secondTuple = ("bob", "joe")

firstTuple = firstTuple + secondTuple

print(firstTuple)

del secondTuple
```

In this example, I never declared a third variable. All I simply did, was declare the first variable and said that its value is equal to the original firstTuple combined with second tuple. This is still valid since we are not actually editing the first Tuple. What we're doing is declaring it, it's as if I'm declaring a new variable that overwrites the old one, since it has the same name.

Dictionaries

Dictionaries consists of keys and values, separated by colons (:) and all within curly braces. Every key within a dictionary must be unique (think words must have distinct meanings), while values do not have to be unique (multiple words can mean the same thing). Key's must be either numbers, strings, or tuples, while values can be any data type.

Accessing and Updating Dictionaries

You can access dictionaries in a very similar way that you access a list or tuple, with the different being instead of having an index inside the square brackets, you have the key name

Example:

```
firstDict = {"name" : "bob"}

print(firstDict["name"])
```

In this example, we can access the value of name by having "name" inside square brackets. You can think of dictionaries like lists, but instead of the index being in numerical order, each index is specially names by you! You can easily update a dictionary by either adding a new key and giving it a value, or directly editing an existing key value

Example:

```
firstDict = {"name" : "bob"}

firstDict["age"] = 25

firstDict["name"] = "jimmy"

print(firstDict)
```

In this example, I first decided to create a new key called "age", and gave it a value of 25. I then decided to change the value of "name" to jimmy. Now when we print "firstDict" you will see it has two keys "age" and "name" with their respective values.

Deleting Elements

With dictionaries it is possible to delete individual elements and even the entire dictionary. Just like previous examples, the del statement can be used to delete parts or all of a dictionary.

Example:

```
firstDict = {"name" : "bob", "age": 25}

del firstDict["age"]

print(firstDict)

del firstDict

print(firstDict) # this will return an error
```

In this example, I first created a simple dictionary with two keys. I then used the del statement to delete the key “age” in firstDict. This is done in the same way as you would print the key or change it, except for I used the del statement instead of something else. The second del statement was used with firstDict in its whole. It deletes all of firstDel, which is why the final print statement will return an error, because firstDect no longer exists when that code is reached.

Combining Loops with Lists, Dictionaries, and Tuples

After reading about of loops and sequential data types you might have been able to notice the potential to combine them to get greater use and production out of your program. Loops allow us to assign or get values very quickly for large ranges, while list, dictionaries and tuples allow us to create really large ranges of numbers really easily. So how convenient would it be to combine the thing that creates large ranges quickly, to the thing that accesses large values easily. Below I'll show a few examples for each type of sequential data type for how to loop through a data quickly:

Example:

```
firstList = []

x = 0

while x < 100: # while x is less than 100
        firstList.append(x) # add the value of x to the end of the list
        x += 1 # increase the value of x

for nums in firstList: # for each number in firstList
        print(nums) # print the number
```

In this example, we first create an empty list called "firstList" and we also create a variable called "x" which we will use as a counter. We then create a while loop and tell it to loop while the value of x is less than 100. We can then use the append function which adds a value to the end of a list, to add the value of X to the end of firstList. Then we simply increase X by one. This will cause 100 values to be added to firstList. We can verify that this works by using a for loop and saying that for each value inside firstList, print the value. When you run this program, you're output will have 100 values, from 0 to 99.

Example 2:

```
firstTuple = ()

x = 0

while x < 100: # while x is less than 100
        tempTuple = (x,)
        firstTuple = firstTuple + tempTuple # add the value of x to the end of the
list
                x += 1 # increase the value of x

for nums in firstTuple: # for each number in firstTuple
        print(nums) # print the number
```

In this next example, we reproduce the first example, except with tuples. One major difference with tuples and lists is that we can't change the value of the tuple "firstTuple" since tuples are read only. So what we do to work around this is, during the while loop, we create a new tuple each loop, which holds the value of x. We then re-declare firstTuple and tell it to equal the concatenation of firstTuple and tempTuple. This will successfully allow us to create a new tuple each loop which will eventually hold the values form 0 to 99. The for loop works identically to the list example, since you can read a tuple identically to lists with a for loop.

Example 3:

```
firstList = []

x = 0

for x in range(0,100): # for the value of x inside the range 0 to 100
        firstList.append(x) # add x to firstList

for nums in firstList: # for each number in firstList
        print(nums) # print the number
```

This example uses only for loops. Instead of use a while loop, I told the program to simply loop through each value in the range 0 to 100. This way instead of manually having to increment x like in the while loops, we can simply say that for every number in that range, which we will represent with X, append what the value of X is. This method is a lot more compact and readable, and does the exact same thing as the previous examples.

Assignment

Loops are really useful for editing really large amount of values. One thing that would be greatly simplified thanks to loops is editing images. Am image can consists of hundreds, maybe even thousands of pixels each with their own color values. With the power of loops, we can very quickly edit these color values. For this assignment, we are going to invert the colors of an image of your choosing. There are some parts to this program that you would not have learned yet, so I'm going to help you out with that. To edit an image, We are going to need to import a package called Pillow into our project interpreter. To do this in Pycharm, you need to go to settings, project: "yourprojectname", project interpreter. There, you just press the plus button, and add Pillow. This package will allow us to use a vast amount of functions to play with pictures. Once you add it to your project, import it into your program:

```
from PIL import Image
```

Since we are going to only be using the Image object, that is all I will import. After that, we will need am image to edit. Get any image of your choosing and add it to your package. You should drag it into the main package folder of your project. Try to get a small image, the program works fairly slow the method that im using, so a very large image may take too much time to process. I found that a 600 by 600 image works fine. Next you are gonna want to start your code with the following two lines:

```
img = Image.open("cat.jpg")
size = img.size
newImg = img
```

img will be a variable which will hold your image. "cat.jpg" should be replaced with the name of your file. Size will hold a tuple which carries the x and y lengths of your image. And finally newImg will eventually hold the new image we are making.

The only hint I will give for this rest now is that you can use img.getpixel which takes a tuple value which has two values ((x,y)) to get a tuple representing pixel color values at the given coordinate, and img.putpixel to change the pixel value. Putpixel takes two parameters. The first is a tuple representing the x and y coordinates, and the second a tuple which holds 3 values, representing the r, g, and b color values of the pixel. to set RGB values and also edit them. In the end you can also use:

```
    newImg.show()
    newImg.save("new.jpg")
```

The first line will show the new img In your default image program, while save will save the image with the name you give it. Good Luck!

Answer and Explanation

The following code is how I achieve this program, and it is commented through out to explain it. As always, there is more than one way to do this, so if you got it working with a different method, you are still correct, as long as it works.

```
                    from PIL import Image # imports image from PIL

img = Image.open("cat.jpg") # opens the image with the given name
size = img.size # sets size variable to img size
newImg = img # sets new img to img

# these two for loops will loops through each X and Y coordinate in the image,
# allowing us to edit each pixel individually

for x in range(0, size[0]): # for each number in the range of the width

                 for y in range(0, size[1]):  # for each number in the range of the height
                     color = img.getpixel((x, y))  # get the color at the given pixel
                     newColor = (255 - color[0], 255 - color[1], 255 - color[2])  # create a new
color,
                     # which will be the inverse of the given color
                     position = (x,y) # create a new variable to hold the current pixel position
                     newImg.putpixel(position, newColor)  # sets the pixel at the position, to the
new inverted color

newImg.show() # shows the image
newImg.save("new.jpg") # saves the image with the name "new.jpg"

# the new image created be colored significantly different from the original.
# each color will be the opposite of it. IE, black will be white.
```

This assignment is a lot more advanced than previous ones, but it really helps to show the practicality of loops, and the amount of freedom and power they give you.

Chapter 10

Functions/Methods

A function is a collection of code that is grouped together to create a specific function or task. This section will teach you how to create and utilize functions to their fullest extent. When you do something like “print” or “append”, you are using a function, which is a group of code which does a certain task.

Creating a Method

The syntax to create a method is as follows:

```
def functionName(parameters):
        # code to execute
        # return value (optional)
```

def: def is a keyword which identifies a function. All functions start with this keyword.
functionName: Is the function name, can be named any legal name of your choosing
parameters: parameters are variables you can give the function to work with. When you print you give it a parameter of the word you want it to print.

Code: The code in the function is executed when the function is called

Return Statement: The return statement is optional. It causes the function to return the value you choose for it to return.

Example 1:

```
x = 1

def value_increase(num):
        num += 1
        return num

print(value_increase(x))
```

In this example we create a method called “value_increase” and have it take one parameter called num. The function will then take the value of num and add one to it. It then returns the value of num. Since this function has a return type, it returns a value. I call this method directly inside a print statement and the code will return 2 because since it has a return type it will print the value of what it returns.

Example 2:

```
x = 1
y = 5

def average (num1, num2):

        print("average is ", ((num1+num2) / 2))

average(x, y)
```

In the example above I create another method that does not return anything. This is good for if you want the method to perform a specific function that does not actually set any value outside of it. The example above has a function which takes two numbers and prints the average of them within the function. Note that if we tried to print the function, it would return an error, because it does not return a value which can be used outside of itself. Also note that a function always needs to be called below where it was written or else the code will not find it.

Example 3:

```
x = 1
y = 5

def sum ():

        print("average is ", ((x+y) / 2))

sum()
```

In this example, we create a function called sum that takes no parameters at all. When a function has no parameters, you must still have brackets, except you put nothing within them. Although this function has no parameters, we can still access the variables above it to use. A function can access code outside of it, if it is accessible. We will go more in depth into this concept when we talk about classes, for now just know that a function can access code if it is called somewhere in the same file.

Anonymous Functions

Anonymous functions are functions that are not declared in the standard way, using the “def” keyword. Instead of the def keyword, anonymous functions use the “lambda” keyword. These are used to create small one line functions. Unlike functions, anonymous functions cannot access variables outside of their own namespace, or global namespace. These functions can take any number of arguments and return exactly one value in an expression form.

Example

```
sum = lambda num1, num2: num1 + num2

print(sum(1,2))

print(sum(5,10))
```

The variable sum gets a value of an anonymous function. The do this by using the lambda keyword, followed by the parameters you want it to take separated by commas. After that, you use a “:” symbol, to separate the parameters from the expression. The value of sum then becomes the value of the expression, which in this case, is the value of num1, and num2 combined together. Now when you print sum it works exactly like a method, however by using lambda the code becomes a lot more compact. This is again, another case of programmer preference, and whether you would prefer to use an anonymous function over a normal one, since there is no difference in the way they work, other than the structure.

Assignment

Create a more advanced calculator. Have three values. The first tells you which kind of operation to do (for example if a == 1, add the values, if a == 2, subtract them). The next two values will be used to manipulate. Make the calculator support addition, subtraction, division, and multiplication, and have each feature in its own method.

```
action = 5  # variable that states which operation to do

x = 4.5  # first number to manipulate
y = 3  # second number to manipulate

# function to add variables
def addition (num1, num2): # takes two arguments
        return num1 + num2 # returns the sum of two numbers

# function to subtract variables
def subtraction (num1, num2): # takes two arguments
        return num1 - num2 # returns the difference of two numbers

# function to multiply variables
def multiply (num1, num2): # takes two arguments
        return num1 * num2 # returns the product of two arguments

# function to divide variables
def divide (num1, num2): # takes two arguments
```

```
        return num1 / num2 # returns the division of two numbers

if action is 1: # if the value of action is 1
        print("The sum is ", addition(x, y)) # calls the addition function
elif action is 2: # if the above fails, check if value of action is 2
        print("the difference is ", subtraction(x, y)) # calls the subtraction function
elif action is 3: # check for value 3
        print("the product is ", multiply(x, y)) # calls the multiply function
elif action is 4: # check for a value of 4
        print("the answer is ", divide(x, y)) # calls the divide function
else :
        print("invalid action, please use a action number between 1 and 4")
```

At this point the code above should be pretty self-explanatory. You should attempt to spice things up a little. Maybe work with a list to play with a few hundred values instead of two. Add some more functions and see how complex you can make this calculator. You have all the knowledge you need to make the most awesome calculator ever now.

Chapter 11

Classes

A class is an extendible code template used to create objects. All the code that we have written can be written exactly as is, inside a class. The creation of classes however, allows us the treat variables and functions inside the class, as objects that we can use in other classes and even other python scripts. It also allows us to create multiple instances of the same variables and functions. With this tutorial on classes, we will be diving into object oriented programming (OOP), and more advanced things in python.

Terminology

Some terms we might use in this section that are useful to know when it comes to Object oriented programming:

Method : A function that is declared within the body of a class

Object: An instance of a data type that is defined by a class. A object contains both variables and methods.

Instance: A standalone object of a class. A instance of a class is like a clone of it, identical, but also independent of it, in terms of editing within it.

Inheritance: The obtaining of variables and methods from one class, into another.

Class Variable: A variable that all instances of a class have, and share. They are created within a class, but outside of a method.

Instance Variable: A variable that is created within a method and only belongs to one instance of a class.

Creating a Class

A class is created in a similar way as a function, but instead of using the keyword "def", we use the keyword "class"

Example:

```
class classname:
        # code

classobject = classname()
```

You first state the keyword class, followed by the name of the class. No brackets are used like when you create a function. We can reference things within the class by creating an object of type "classname".

Example:

```
class sample :

        x = 3

        def __init__(self):
            print("this runs as soon as a class object is declared")

        def double (self, number):
            print("double is ", number * 2)

sampleObject = sample()
```

In this example, X is called a class variable, meaning it's a variable within the class simply. Within the class it can be used like any normal variable, but outside the class you must access it through an object of the class. The function "__init__" is automatically called whenever you create a new instance of the class. If you run the code above, you will notice it will print something into the console. This is because when you said that sampleObject is a sample object, you created a new instance of the class, and therefore "__init__" is automatically called. Note that whenever you create a function inside of a method, you must have the first parameter be self. Self is a more complicated thing that is not something that should be worried about other than that it exists, and that it essentially just references the class, self, being the class itself.

Creating Instance Objects

Creating a instance of a class is very similar to creating a variable. You simply create a variable name, and make it equal to a class. Since the _init_function is called when a new instance is created, you can make your class take parameters like a function through it.

Example:

```
class sample :

        x = 3

        def changeX (self, number):
            self.x = number

sampleObject = sample()

sampleObject2 = sample()

sampleObject.x = 5

print(sampleObject2.x)
```

In this example, we create two instances of the class "sample". Each instance is independent of the other, as can be seen when I say "sampleObject.x = 5". What is happening there is that im changing the value of x, inside the class "sample", but only for the sampleObject, object. Since im only changing the value of x for "sampleObject" the value of x does not change for sampleObject2. Because of this, when you print the value of sampleObject2.X It remains 3, and not 5, because only sampleObject.X was changed.

Example:

```
class sample :

        x = 3

        def _init_(self, xValue):
            self.x = xValue

        def changeX (self, number):
            self.x = number

sampleObject = sample(5)

sampleObject2 = sample(6)

print(sampleObject.x)
print(sampleObject2.x)
```

In this example, we can add in parameters when we create the class object. Since the __init___ function is called when the class object is created, you can give the __init___ functions parameters, which can then be passed into the brackets after the class name. Notice in _init____I add the keyword "self" before x. this is to let the program know that I'm changing the variable x of class, and not creating a new variable inside of the function called x. When this program prints, each value is different, and has the value given through the __init___function.

Accessing Attributes

Any Attribute in a class can be accessed through the class object. Up above we accessed variables through the class object, but methods can also be accessed.

Example:

```
class sample :

        x = 3

        def changeX (self, number):
            self.x = number

sampleObject = sample()

sampleObject.x = 15
print(sampleObject.x)

sampleObject.changeX(13)
print(sampleObject.x)
```

All variables and functions within a class can be easily references and used by adding a dot after the class object, and then writing the name of the variable/function. In the code above, we changed the variable x directly, and changed it through the changeX function. Both are referenced the same way.

Inheriting Classes

When creating a class, you can derive the class from a already existing class, allowing you to access variables and function within the class you just created, as well as the class that you are inheriting.

Syntax:

```
class classname (parentclass):
# code inside class
```

You declare the class the same way as you would if you were not inheriting. The difference though is that inside brackets you add the name of the class of which you want to inherit from.

Example:

```
class parentclass :
        y = 13

        def parenttest (self):
            print("im from the parent class")

class childclass (parentclass):

        x = 4
        def childtest (self):
            print("im inside the child class")

children = childclass()
children.childtest()
children.parenttest()

children.x = 3
children.y = 3

print(children.x)
print(children.y)
```

In the example above, we create two classes, one called “parentclass”, and the other “childclass”. The child class we have inherit the parent class therefore allowing us to access all things within the parent class as well as the child class, when ever we create a childclass object. We create a childtest object called children. Through this object we are able to call methods from within the parent object, and the child object, as well as access and edit variables in the parent and the child. Note that it is also possible to inherit more than one class. You may inherit as many classes as you want.

Example:

```
class parentclass2 :
        z = 13

class parentclass :
```

```
            y = 13

            def parenttest (self):
                print("im from the parent class")

class childclass (parentclass, parentclass2):

            x = 4
            def childtest (self):
                print("im inside the child class")

children = childclass()
children.childtest()
children.parenttest()

children.x = 3
children.y = 3
children.z = 3

print(children.x)
print(children.y)
```

I’ve added another parent class to childclass, by adding a second class within the brackets of childclass, separated by a bracket. You can access the properties of all parent classes the same way. It is even possible to access classes that the parent inherited.

Example:

```
class grandparentclass :
            z = 13

class parentclass (grandparentclass):
            y = 13

            def parenttest (self):
                print("im from the parent class")

class childclass (parentclass):

            x = 4
            def childtest (self):
                print("im inside the child class")

children = childclass()
children.childtest()
children.parenttest()

children.x = 3
children.y = 3
children.z = 3

print(children.x)
print(children.y)
```

This class is like the one before it, except instead of having childclass inherit two classes, childclass only inherits parentclass. The catch though is that parentclass itself inherited grandparentclass, so childclass indirectly inherits from grandparentclass, allowing childclass to access all 3 classes. This works the exact same way as if it were to inherit both classes directly, there are no disadvantages to inheriting indirectly.

Overriding Methods

When inheriting from a parent, if the child and parent both have a function with the same name then the child function will be called. This is called overriding methods. The child function will always override the parent function in the case of any overlapping names.

Example:

```
class parentclass :

        def earth (self):
            print("im the parents earth")

class childclass (parentclass):

        def earth (self):
            print("im the childs earth")

childObject = childclass()

childObject.earth()
```

In this example, we have two classes “parentclass” and “childclass”. Childclass is inheriting from parentclass. Both classes have a function called “earth”, but when I create a childclass object and call the earth function, only the child function gets called. This is because when they both have the same name, the parent function will ALWAYS be overwritten by the child function.

Assignment

For this assignment, we are going to recreate the calculator we created earlier in this tutorial, except this time we will utilize our knowledge of classes. All your functions must be inside a parent class, which you then must inherit and test your cases inside your childclass. As before, you must have 3 variables, a action variable, and two variables to manipulate. Your program must support addition, subtraction, multiplication, division. You must have at least two classes, and you need to give the value of the action variable through the object of the child (hint: use __init__).

Answer

```
class mathfunctions : # class filled with math functions

        # function to add variables
        def addition (self, num1, num2): # takes two arguments, and self
argument
            return num1 + num2 # returns the sum of two numbers

        # function to subtract variables
        def subtraction (self, num1, num2): # takes two arguments, and self
argument
            return num1 - num2 # returns the difference of two numbers
```

```
    # function to multiply variables
    def multiply (self, num1, num2): # takes two arguments, and self
argument
        return num1 * num2 # returns the product of two arguments

    # function to divide variables
    def divide (self, num1, num2): # takes two arguments, and self argument
        return num1 / num2 # returns the division of two numbers

class actiondeclarer (): # class used to check actions.

    # the none keyword creates an empty variable
    action = None   # declare an empty action variable
    x = None  # declares an empty variable to manipulate
    y = None  # declares an empty variable to manipulate
    functions = mathfunctions() # object for mathfunctions

    # init function runs as soon as the class object is made
    def __init__(self, actionnum, num1, num2):
        self.action = actionnum  # variable that states which operation to do
        self.x = num1  # first number to manipulate
  self.y = num2  # second number to manipulate
        self.actionpicking() # calls the actionpicking function in the class

    # this method checks the action number
    # and runs the appropriate function
    # self references variables inside the class.
    def actionpicking (self):
        if self.action is 1:
            print("The sum is ", self.functions.addition(self.x, self.y)) # calls
addition method
        elif self.action is 2:
            print("the difference is ", self.functions.subtraction(self.x, self.y))
## calls subtract method
        elif self.action is 3:
            print("the product is ", self.functions.multiply(self.x, self.y)) # calls
multiply method
        elif self.action is 4:
            print("the answer is ", self.functions.divide(self.x, self.y)) # calls
divide method
        else :
            print("invalid action, please use a action number between 1 and 4")
# no methods called

# creates an actiondeclarer object
# and gives it the three parameters required
# because of _init___
checker = actiondeclarer(2,5,10)
```

Of course, there is not much purpose for over complicating something like this when we already did it in a much simpler way. But it’s good to see how you can write the same thing in multiple different ways, and it could be a bit easier to grasp classes by practicing on something you already have experience doing without classes.

Chapter 12

Debugging

Sometimes when programming, your code may give wrong outputs. This is called a logic error. Your IDE will not tell you anything is wrong, because technically your code follows the rules of Python. It’s like writing in any language. You may follow the rules of the language, like grammar, structure, etc., but your paragraphs don’t necessarily have to make sense. The most common method of debugging is simply using print() and printing out variables to find where values stop making sense. Then you can find the lines that are causing the error and debug that. Another tool when debugging is breakpoints. These are points that act as “checkpoints”, where the code will stop executing until the programmer lets the code continue.

One thing that may or may not be considered debugging is optimization. Once you’ve written your code, it may be tempting to just leave it and continue on with other tasks, but this may not be what you want to do. If your program gets the desired results eventually, but it takes a long amount of time, then it is not very good. With that being said, there is no single strategy to make code more optimized. Sometimes your logic could simply be a long method of doing something, and sometimes it could be how you implement that logic. Be wary of creating unused variables, as they can take up processing time and memory. This is because of a system called “garbage collection”. It goes through all the variables in your code and removes any ones that are no longer valid. For example, if you make a for loop, when it completes, that variable is no longer valid, so it should be deleted to save on memory. This takes up (small) amounts of processing time.

Here is an optimization example. Say you want to find the distance from one point to another. Math class tells us to use Pythagorean Theorem. But this is rather slow. First the computer must square both sides (which is quite expensive performance-wise), then add them, and finally square root them. This calculates the accurate distance between two points. Another solution though, could be to use “Manhattan distance”. This will not get you accurate distances, but it could be good in situations where the exact value is not important, and speed is more important. To do Manhattan distance, simply absolute (which means to make a value positive, so -7 becomes 7 and 8 becomes 8) both the x and y components and then add them. This gives you an inaccurate distance, but it is much faster than its more accurate counterpart. This is particularly good when you are guessing the fastest route. Rather than constantly calculating the distance accurately, Manhattan distance is a cheap alternative and will get you a near enough distance.

Chapter 13

Exception Handling

When you're coding, you may land in situations where your code returns an exception when attempting to do something. For example, if an object requires an integer in its constructor, but you feed it null instead, the compiler will return an error indicating that the object didn't receive the appropriate parameter.

Another example would be when you're working with file handling which will later be taught, and you're trying to read a file in a directory that doesn't exist. The compiler would then again return an exception such as "FileNotFoundException" - there are different exceptions but this type of exception is regularly found in File Handling.

In order to regulate exceptions in your code, you can use the try-catch statements. What these statements do is check if a block of code returns an exception, and if it does, instead of halting the whole operation of the program, it will catch the exception and try to either fix it or do something alternatively depending on what the programmer codes. Here is a use of a try-catch statement:

The file "text.txt" does not exist.

```
try:
        ReadFile ("text.txt")

     except:
        print("error")
```

The except stands for exception which is the "catch" of the Python's try-catch statement. The ReadFile() function is not a real function but is simply here to give you an example of when it might need to be used.

Chapter 14

Threading

Threading is a way to run two “threads” of code at the same time. Threading does not necessarily make your program run faster, but in select situations it can. Threading shares processing power, so it does not speed up any code. Threading works like two very nice people walking through a corridor divided by tons of doors. The first person lets the second go through the first door, and then the second person lets the first go through the second door and repeat. Essentially, what threading allows programmers to do is run loops that will run for a very long time, while still doing other things in the background. For example, if you want your program to accept a connection from another computer, you could run a loop that constantly “listens” for a connection.

The problem with this is that your program will freeze while it listens. Instead, we use a thread to listen for a connection, and then do other things in our main thread. We would show the user something like a GUI (not available in this guide since it is a beginners crash course), and give them feedback, based on the listening status. So how do we make a thread? First, we specify the library we want to use, which is threading. Then we create a class that will be utilized as the thread proxy, and we must initialize the method which will act as our thread using the target key in the Thread constructor object. Anything within this method will be our thread. In order to call our thread, we must declare a thread object referencing the thread proxy class and then starting it implicitly. How do we do this? Here’s an example:

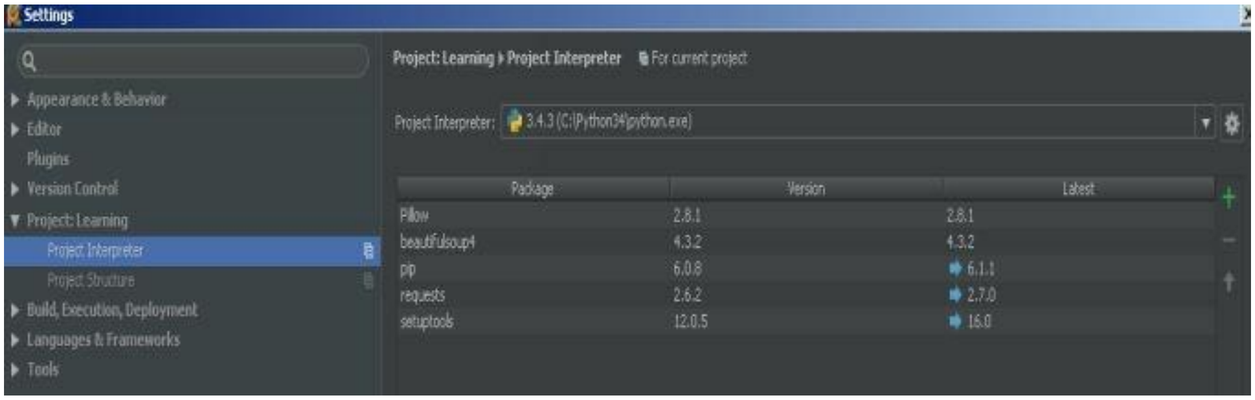
```
import threading
import time

class mySillyThread:
    while True:
        print("Groundhog Day!")
        time.sleep(1)

thread = threading.Thread(target=mySillyThread)
thread.start()
```

The code above will now run two infinite, unrelated loops, and will “randomly” print out “Groundhog Day” every 1 second due to the time library that we imported. Threading is not limited to infinite loops though. Threads are simply for whenever you need two concurrent lines of processing. For example, say you’re making a video game and you’re making a loading menu. You may make one thread load the game data, and then simply put its status into a variable (like how many files have been loaded). Then, you could have another thread modify your window or console or whatever to tell the user the status of loading based on the variable that the original thread is using. This is how (most) loading bars are made.

Chapter 15

Web Crawlers

Web Crawlers are basically programs that navigate around the web and scrape for content depending on the parameters that are set out for them to patrol the web and what to do when finding certain content. Python has a great support for web crawling as they have many libraries built for them in Python. In order to download these libraries in PyCharm, you must use the shortcut: crtl + alt + s. Then you must go to the Project → Project Interpreter. Once you have reached this page, press the + button in order to add a new package.

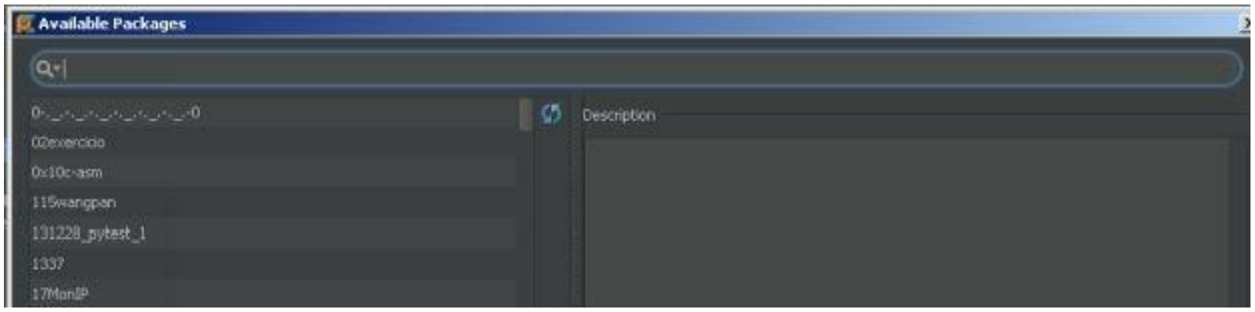


Project: Learning › Project Interpreter   For current project

Project Interpreter: 3.4.3 (C:\Python34\python.exe)

Package	Version	Latest
Pillow	2.8.1	2.8.1
beautifulsoup4	4.3.2	4.3.2
pip	6.0.8	6.1.1
requests	2.6.2	2.7.0
setuptools	12.0.5	16.0

Once you have pressed the green + button, you will reach this page:



```
0-._.-._.-._.-._.-._.-._.-0
02exercicio
0x10c-asm
115wangpan
131228_pytest_1
1337
17MonIP
```

Now search for the following package: beautifulsoup4. Now select this and install the package. Once it has been successfully installed, we can reference it in our script like the following:

from bs4 import BeautifulSoup as bSoup

Since bs4 is a more intricate package, we must access it in the similar fashion as the above line. An extra keyword has been added here "as" – this is not necessary but is simply used to indicate a name for the module we are adding so we can reference it in the code as bSoup and not as the module name itself implicitly which in this case is "BeautifulSoup".

Now before we move on to BeautifulSoup and what it does, we are going to create a simple web crawler that downloads an image if given a link to download it from. The first step is to import urllib.requests which is a package that will allow us to download an image when retrieving the link of it. Here is an example of an image downloader:

```
import urllib.request
import random

def DownloadImage(url):
    fileName = str(random.randrange(1, 100)) + ".jpg"
    urllib.request.urlretrieve(url, fileName)

DownloadImage("http://images.cdn.stuff.tv/sites/stuff.tv/files/styles/big-
image/public/news/telltale-gameofthrones-screen.jpg?itok=BjepCsCy")
```

In the above example, we are importing urllib.request and importing the random module. Next, we define a function named DownloadImage and requesting a variable of name url. Then, we set the filename to a random value that is outputted using the randrange function in the random module and then extending the string (filename) with a .jpg so that when PyCharm does download the image, it is in proper format to be able to view. Next, we use the urlretrieve function to input the url parameter and the filename when it downloads the content from the webpage. Finally, we call our function outside of the function after its definition and give it a random image. Once the program has executed and has downloaded the image, we should then be able to view it in PyCharm.

Now that was a really simple web crawler that simply retrieved a URL and downloaded the image associated with it. Now let’s create a more sophisticated web crawler that can go to a forum post and dissect a certain block of text. Let’s use thenewboston.com website’s forum to do this. Before we get started, let’s understand what BeautifulSoup does. The module BeautifulSoup in the bs4 library is used to parse HTML data (code for webpages) in a way where you don’t have to mess with string functions in order to dissect certain values from the webpage. It is important for us to understand HTML to be able to web crawl so we will go over how HTML works right after this little guide on web crawling.

```
import requests
from bs4 import BeautifulSoup as bSoup

def ScrapeThread(threadNumber):
    url = "https://www.thenewboston.com/forum/topic.php?id=" + str(threadNumber)
    srcCode = requests.get(url)
    code = bSoup(srcCode.text)
    for postedCode in code.find('code'):
        result = str(postedCode).replace('<br>', '\n').replace("\ufffd", ' ').replace('&l_t_;',
'<').replace('</br>', '')
        print(result)


ScrapeThread(1610)
```

In the above script, there are many things going on. Let’s start from the beginning. First, we must import requests, which will allow us to get source code from a URL that we supply it. Next, we’ll import BeautifulSoup and make the alias bSoup so we can reference it easily. Next, let’s define a function named “ScrapeThread” which will take in a thread number. If you ever browsed a forum, you will notice that each thread has a thread id that is associated with it in the URL. That is why we create a url string object that is equal to the standard URL of a forum thread in thenewboston.com website with a thread number at the end so that it can dynamically fetch data from the specified thread number that is supplied to the function.

Next, we create a variable named srcCode that is equal to the requests object's function get which returns the HTML content of a web page when given the URL of the webpage. Next, we create a code variable that converts all the jumbled HTML code of the web page in to an organized template that BeautifulSoup's functions could understand. Now things become interesting. We then create a for loop with a variable of postedCode which then gets filled with the data of the code.find("code"). What exactly is going on here? Well when you learn HTML, you will understand that there are something called tags that are used to indicate where a certain piece of content is. We are calling upon the "code" tag which is a tag used in forums when people want to post their code (this is a programming forum). Once we retrieve the content within the code tag, we then do some string magic in order to make it cleaner to read and then print it out accordingly.

Now, this must've been all confusing because you don't yet understand how HTML works. Here's a quick guide on HTML/CSS:

Basics of HTML and CSS

HTML

In order to understand how we can implement the Python language and work with it, we must understand the fundamentals of how a web page in its most basic form works. HTML stands for HyperText Markup Language. It is the standard markup language used to create static web pages. That is why, with the help of Python, we can make these web pages more dynamic and responsive.

Tags

In HTML, there is a standard convention to starting a statement, and ending one. In English, we must start our sentences with a capital letter and end them with a period to maintain a standard flow of writing which is easy to interpret. Well, HTML uses the idea of "Tags" to be able to understand what exactly you are writing. There are a few vital tags you must remember in HTML, and understand what they are used for.

In any HTML file, there is a standard structure that must be followed. Here is sample code showing an example of it:

```
<html>
        <head>
                        <title> This the webpage title </title>
        </head>
        <body>
                        This is the content of the webpage.
        </body>
</html>
```

When you are making a tag, you must start it with a "<" symbol and then indicate the name of the tag, and then end it off with ">" symbol. Then, the "inner" portion of the tag, or within the tag, goes the rest of the code. To end the tag, you do the same convention as starting the tag, except end it with a "/". This is why you see the top of document as: <html> and the end of the document as </html>.

Visual demonstration:

<tagname> content within tag </tagname>

Now let's analyze the sample code down:

<html>

This is the tag that indicates when the HTML portion of the site will be, so it must be presented in every HTML document. The tag is then closed at the end of the document. Therefore, the <html> tag is the parent of all the tags within it, the children.

<head>

This is the tag that would hold all the header information when loading a web-page. Everything within this tag will not be shown to the user, and is mostly used to initialize scripts to ready them for use. An example would be to set a script tag in the head section to prepare for use for the rest of the HTML document.

<title>

The contents of this tag will hold the title of the webpage.

<body>

This is the whole content part of the website. Just line in any essay, you have your thesis, body and conclusion. This is the same idea in the form of tags.

There are many other tags in HTML that you could look up online. Let’s look at a variety of examples and identify some helpful tags.

```
<html>

        <head>
                <title> Example of "<p>" tag </title>
        </head>

        <body>
                <p> This is a paragraph. </p>
        </body>

</html>
```

The bolded tag in the above example essentially describes a paragraph and formats it in such a way where it has proper padding/spacing without you having to format it yourself through the use of CSS (ability to style HTML elements).

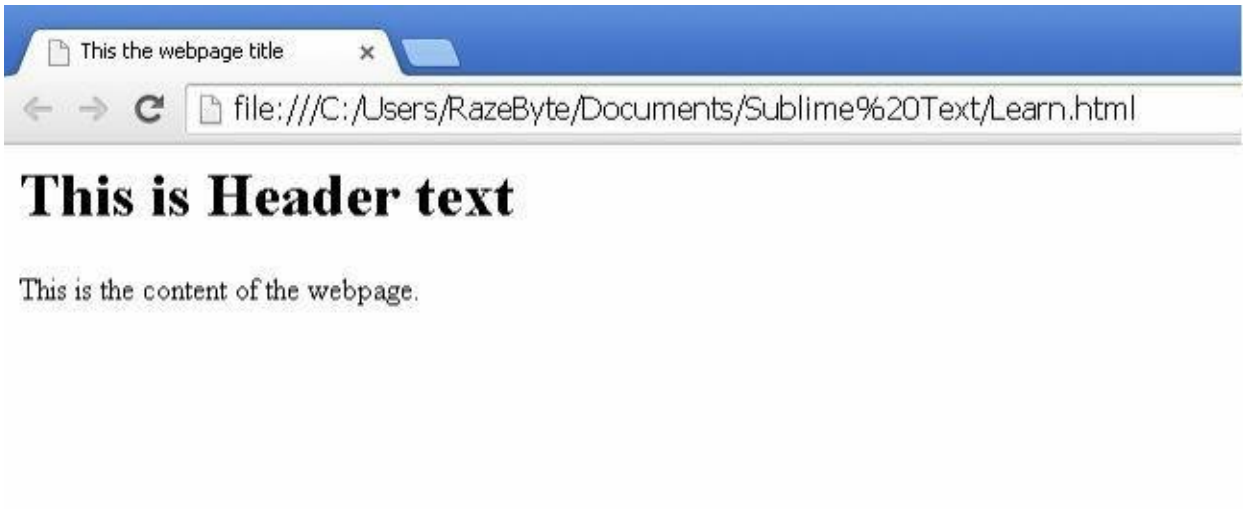
```
<html>

        <head>
                <title> Example of "<p>" tag </title>
        </head>

        <body>
                <h1> This is a heading. </h1>
        </body>

</html>
```

This <h1> tag used in the above example simply bolds the text in between it and enlarges it. All properties of tags that you will be using are editable through CSS which we will be going through soon.

Here is an example using both tags that we have learned and showing the output. Note that once you have created a file with HTML in it, you must save the file with a .html extension. You can then open the file and test it out by opening with a browser of your choice.

```
<html>

        <head>
                <title> Example of "<p>" tag </title>
        </head>

        <body>
                <h1> This is a heading. </h1>
        </body>

</html>
```

Output:


This the webpage title
file:///C:/Users/RazeByte/Documents/Sublime%20Text/Learn.html

```
This is Header text

This is the content of the webpage.
```

Finally, let's go over the idea of commenting in HTML. Sometimes you might want to code something but not exactly print it to the screen, but just want to explain a statement that you are doing or adding a reminder in the code without having it translate visually on the web browser so that it won't be able to see it.

You can comment by starting your comment with <!-- and ending it with →

Therefore, if I wanted to comment a certain line, I would do the following with a functioning paragraph:

```
<p> This is paragraph text </p>
<!-- This line prints out a paragraph that says "This is a paragraph text" -->
```

There are tags, and then there are attributes, which contain values which are then used to determine how the tag's contents are displayed or are interpreted by CSS or Javascript. Two important attributes to note when going in to Javascript and CSS are: id and class.

You can use these on any appropriate tag like the <p> tag within the <body> tag.
The format of how an attribute works is like the following:

```
<p id = "content"> This is the content of the webpage. </p>
```

As shown in the example, in order to declare an attribute, you must make space within the first tag which is <p> and then name the known attribute, so id for example. Then to give it a value, you must have an equal sign and then two quotation marks and the value within the quotation marks. In this case, this paragraph, contains the attribute id which is the value of “content” - this can be used as a reference for languages like CSS and Javascript through the use of selectors which you will be learning about.

There are still many tags that you can learn about in HTML but these are the basic tags that you need to know for now. Also, you should now understand the basic structure of an HTML document.

Let’s review:

The tag **<html>** and **</html>** defines the HTML document, and the tag itself simply defines the content within it as HTML type so the browser is able to interpret it.

The tag **<head>** and **</head>** provide information about the document and initialize any external scripts or styling sheets.

The tag **<title>** and **</title>** simply give information to the browser to display a title of the HTML document.

The tag **<body>** and **</body>** holds all the content of the document which includes the paragraphs, navigation bars, tables, images, etc.

The tag **<h1>** and **</h1>** simply formats text in between it as a header or a title to a paragraph.

The tag **<p>** and **</p>** formats text in between it as a paragraph accordingly.

CSS

CSS stands for “Cascading Style Sheets” and essentially styles how HTML elements are displayed on the web browser.

Now, before we move on, let’s quickly go over how commenting works in CSS. Just like in HTML, commenting is useful, and can also be done in CSS, but it has different syntax for it. In order to comment in CSS, you must use the following notation:

/* This is a comment */

The text in between the slash and symbol is not processed and is simply there for human reading. So to begin a comment you must do /* and to end a comment you must do */.

Let’s scan through the idea of how to write CSS code.

Example:

```
p {
        color: red;
        font-size: 14px;
}
```

In the above example, there are a few critical things to note of the format in the above CSS code.

The element we must specify when writing CSS code is the tag name or the selector.

```
p {
        color: red;
        font-size: 14px;
}
```

The bolded text indicates the selector, which is the <p> tag, which handles the paragraphs of the document. If you place any text in between <p> in the body, you will now see the text being red, and the font size being 14 pixels. Let’s move on to analyzing this code further.

```
p {
        color : red;
        font-size : 14px;
}
```

The bolded text indicates the property in the above example. In every selected tag, you must indicate properties of that tag, like the color of the text, the size of the text, the style of the text, the font-weight and much more. You can always look for any property that you’re wanting on the internet. There are many properties and we won’t be going over them in this guide. There are different properties that are applied to different tags. For the “p” tag in this instance, we can reference the color property and the font-size property. Upon referencing them within curly braces (which simply indicates the block of properties that applies to the selector which is p) and then assigning it to a certain value.

```
p {
        color: **red** ;
        font-size: **14px** ;
}
```

To assign a certain value to the property of color, you must make a colon symbol which simply separates the property and the value, and then type out a value accordingly. Once you have written out the property and the value, use the semi-colon as a way to let the browser know that you have ended that statement so it can read the next statement.

When working with colors, instead of using the standard red, blue, white, yellow, etc - You can reference more specific colors depending on the RGB values (Red, Green, Blue) which can be obtained using an HTML color selector. This is a link of a site that can give you HTML color codes for use in your CSS.

Link: http://html-color-codes.info/

To review CSS syntax let's look at it in the following approach:

1. p
2. {
3. color:
4. red;
5. }

The code above takes the color of the paragraph and changes the color to red. Each line has been numbered, but wouldn't be included in the code. It is numbered so we can scan through each line and go over how it works.

1. Selector/Tag

2. Indicating the start of the selector properties

3. Property/Attribute name with a colon to prepare to indicate the value

4. Value of the Property/Attribute and then a semi-colon to indicate the end of statement

5. Indicating the end of the selector properties for the certain selector which was p

Remember, you can do this as many times, just remember to separate the selector blocks so the browser doesn't get confused. Well, let's first go over how you can actually start coding in your HTML document with CSS. To begin, you must go in between your <head> </head> tags and input the following:

```
<html>

        <head>
                <title> This the webpage title </title>

                <style type = "text/css">
                        p {
                                color: red;
                                font-size: 14px;
                        }
                </style>

        </head>

        <body>
                <h1> This is Header text </h1>
                <p id = "content"> This is the content of the webpage. </p>
        </body>

</html>
```

In order to start coding CSS in an HTML document, you must use the “style” tag with the help of the “type” attribute which references to the standard convention “text/css” which you must use every time you want to write CSS in your HTML document. Within this style tag, is your CSS code which references successfully to all p tags in the HTML document. Well, what if you’d like to only reference a specific paragraph?

Now the reason I like to call p as a selector instead of a tag is because the term selector is a general term and what can be put in the position of p is also general. Instead of putting an actual tag that is being referenced in the HTML document, you are able to reference an id by using a hashtag and then the id name accordingly:

```
#idname {
        color: red;
        font-size: 12px;
}
```

You can also use the selector to reference a tag with a “class” attribute with a certain name. For example:

```
.classname {
        color: red;
        font-size: 12px;
}
```

Now you must be wondering why id or class could ever become helpful. Let me enlighten you with the following sample code:

```
<html>

        <head>
                <title> This the webpage title </title>

                <style type = "text/css">
                        #content {
                                color: red;
                                font-size: 14px;
                        }
                </style>
</head>

        <body>
                <h1> This is Header text </h1>
                **<p id = "content"> This is the content of the webpage. </p>**
        </body>

</html>
```

In the above example, you can see that we listed a specific paragraph to be affected by the change we listed in the CSS within the head tag. This is one way that you can use the id selector. To use the class selector (not really a big difference, just two preferences) is to do the following:

```
<html>

                <head>
                                <title> This the webpage title </title>

                                <style type = "text/css">
                                                .content {
                                                                color: red;
                                                                font-size: 14px;
                                                }
                                </style>

                </head>

                <body>
                                <h1> This is Header text </h1>
                                <p class = "content"> This is the content of the webpage. </p>
                </body>

</html>
```

In the code above, instead of using a hashtag symbol to select a specific attribute with a specified id name, but rather a class name this time. Using this knowledge of selectors, we can reference specific HTML elements using Javascript and use that to manipulate the tags, by adding attributes or changing the content within the tags and making them dynamic to user interaction.

In CSS, there are different ways to actually implement the styling language into the HTML. I have already showed you one way, which is to have it in the head tag with a style tag. Well, there are three main different types of ways to implement CSS, one way is inline, other way is internal, which is the way we did it and the other way is to do it externally. The way we did it directly in the head, internally. To make it external, instead of having the CSS code within the tags, we have the style tag reference a file that contains all the CSS code.

Let’s simply look at examples of how each way works. Remember, if you reference a certain file, like a CSS one in this instance, if you don’t list the direct directory path (e.g. C:\Program Files (x86)\Crazybump) and just the name, then it will resort to looking at the root directory your HTML file which is referencing is in.

External CSS

Filename: *Learn.html*

```
<html>

        <head>
                <title> This the webpage title </title>

                <link rel="stylesheet" type="text/css" href="pageStyle.css">

<!-- This is the conventional line used in the head tags to reference an external CSS file. The href
attribute is used to reference the specific file and is the only one you will change from this line
for your CSS file. -->

        </head>

        <body>
                <h1> This is Header text </h1>
                <p id = "content"> This is the content of the webpage. </p>
        </body>

</html>
```

Filename: pageStyle.css

```
#content {
        color: red;
        font-size: 14px;
}
```

Now, since we've already seen how internal CSS and external CSS referencing works, we can look at the cheapest way to do CSS: Inline. This way is not recommended just because it isn't neat or organized to look at. Although it is useful when testing a few things with a specific tag. In order to do inline, for any tag you want to style, you must add the attribute "style" with an equal sign and then quotation marks for all the properties. Then for each property, you must have the value separated by a colon and when you are done writing down all your properties, you must end it off with a semi colon. Here is an example:

```
<p style = "color: red; font-size: 14px" id = "content"> This is the content of the webpage. <p>
```

This above is an example of inline coding. Here is the full source code:

```
<html>

        <head>
                <title> This the webpage title </title>
        </head>

        <body>
                <h1> This is Header text </h1>
                <p style = "color: red; font-size: 14px" id = "content"> This is
the content of the webpage. </p>
```

```
		</body>
</html>
```

We have finally gone over the fundamentals of HTML and CSS and can finally understand how web crawling can work. Through this guide, we have went over tags, and with BeautifulSoup, we can dissect information between certain tags when asked for the name, which is what we did in the TheNewBoston web crawler.

Chapter 16

Example Programs

Let’s take a look at a few low difficulty problems on ProjectEuler (http://projecteuler.net) that we can solve and let’s go over how we can solve them using Python. I will explain how we can approach each algorithm to help you start thinking in a programmer’s mind and then you can try to solve it. It is recommended that you sign up for this site and practice your new found Python skills on there.

Multiples of 3 and 5

Problem 1

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

Now let's think about this. What exactly is this problem asking from us? This problem is asking us to find all the multiples of 3 and 5 below 1000. Now mathematically, without programming, let's think about how we would approach that. Anything that is divisible by 3 or 5 that is within the sequence from 1 to 999 would be added to a list of numbers.

After finding all the numbers that are divisible by 3 or 5, we would then take a look at our list of numbers and add all these numbers up together and then provide the sum to ProjectEuler to see if we got the answer correct. This seems pretty easy enough but it would take a long time to complete if we were to simply have a pen and paper handy. But now, with our programming knowledge, we are able to figure out how to get the sum of all the multiples of 3 and 5 below 1000.

In order to actually loop through from 1 to 1000, we should use a loop. It doesn't matter what type of loop we would use but we need to use a loop that affects a variable to increment from 1 to 1000, not including 1000 because the question is asking us to find the sum of all the multiples of 3 or 5 below 1000, not below or equal to.

Then, through each iteration of the loop, we must check if the variable being affected by the loop is divisible by 3 or 5. There are multiple ways to do this, which can vary from checking the remainder (using the % symbol) or dividing and checking whether or not it is a whole number. This is pretty trivial to figure out on your own. The next thing we must do if the number is divisible by 3 or 5, is to add it to a list. We can either add it to a list and then loop through the list (or array) or add all the numbers together or we can create a variable before the loop and simply add on to any number it finds is divisible by 3 or 5. An efficient way to do it is by doing the latter.

The reason I pointed out the array way is to essentially let you know that there are multiple ways to solve problems in programming but it is important to find out which way is the most efficient way using your knowledge and a spice of logic. Now try programming this using what you know in Python and the logic I have provided and try to create a block of code that can do the functionality accordingly.

Let's move on to the next problem on Project Euler.

We first check if our solution works with their example:

''' If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.'''

```
        sum = 0;
```

```
limit = 10
for x in range(1, limit):
    if (x % 3) is 0 or (x % 5) is 0:
        #print(x)
        sum += x

print(sum)
```

We then get the output: The sum below 10 is 23

The output is correct, so we are going to input the number they ask to solve the problem with the following code:

'''If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000. '''

```
sum = 0;
limit = 1000
for x in range(1, limit):
    if (x % 3) is 0 or (x % 5) is 0:
        #print(x)
        sum += x

print(sum)
```

This time the limit is 1000. The output is now:

The sum below 1000 is 233168.

Even Fibonacci Numbers

Problem 2

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

In this problem, we are told the pattern the Fibonacci sequence is generated by and then we are told what we must solve. This problem asks us to find the sum of the even valued term values in the Fibonacci sequence. Don’t get confused and assume they are talking about the term indexes, but rather the term values.

There are technically two problems in this problem, one of which we have already completed. You see, in the last problem, we were checking whether or not a number was divisible by x or y in a range from 1 to 1000 and then adding it to an accumulation variable. Well the same idea applies here, where while we generate the Fibonacci sequence, we check whether or not the term value that was generated in the certain iteration is even or odd. If the term is even, then we simply add it in to an accumulation variable, if not, we do nothing about that iteration.

So truly, we’re only really solving the Fibonacci problem here. This is the great thing about practicing algorithms, you start to gain a muscle memory over them so you get better and better at doing them efficiently the more you practice.

So how exactly do you generate the Fibonacci sequence? Well we know that you start with 1 and 2. Then what happens is you add the two terms together to get the third. Then you get the last 2 terms and add those together to get the fourth, and so on.

1, 2, 3, 5, 8, 13…

1 + 2 = 3
3 + 2 = 5
5 + 3 = 8
8 + 5 = 13

Now that we have a better understanding of how the Fibonacci sequence works, let’s see how we can recreate that pattern in Python. First, we must create a loop that has the amount of iterations identical to the term value that we want to get. So if we want to get the 3rd term value, then we have 3 iterations, and if we have 3 iterations, we’re going to end up getting the value of 3 because in the sequence, the third term value is 3.

Although, before creating the loop, what we should do is declare two variables, one that is initialized at one, and another which is initialized at two. Why? Because we should simulate exactly how the Fibonacci sequence works by having these two variables dynamically change as the iterations of the loop go on. Let’s go back to our loop and look at how we would do this. First, let’s create a quick variable that is the sum of the two variables we initialized before the loop. Next, let’s check if the second variable that we declared as 2 is even, and if it is, then we add it to an accumulation variable. Next, we assign our variable that was assigned one, to the second variable that was assigned two and then assign our second variable that was assigned two, to the sum of the two numbers.

Now if you read that carefully, you would understand that we are simply cycling through the Fibonacci sequence in this exact algorithm that I have just specified. Now you can try it out!

Solution for first 10 terms by specifying limit of 89:

'''Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms. '''

```
prev = 1

next = 2

sumTotal = 0

termValue = 89

for n in range(1,termValue):

    tmpSum = (prev + next)

    if (prev < termValue):

        if  (next  %  2)  is  0:

            sumTotal += next

    else :

        break

    prev = next

    next = tmpSum

print(sumTotal)
```

Solution for problem with limit changed to four million:

'''Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms. '''

```
prev = 1
next = 2
sumTotal = 0

termValue = 4000000

for n in range(1,termValue):
    tmpSum = (prev + next)
    if (prev < termValue):
        if (next % 2) is 0:
            sumTotal += next
    else :
        break
    prev = next
    next = tmpSum

print(sumTotal)
```

Let’s check out another Project Euler problem.

Largest Prime Factor

Problem 3

The prime factors of 13195 are 5, 7, 13 and 29.

What is the largest prime factor of the number 600851475143 ?

Now in order to do this problem efficiently and under a minute, we must carefully consider how we want to approach solving this problem. Since it is asking us to find the largest prime factor for a really high digit number, it can cause long processing time for the program to figure it out if our algorithm isn't properly configured. That is why we must think about this carefully.

I know that if I take a number like 100 and wanted to figure out the largest prime factor I would do it by doing the following:

100 / 2 = 50
50 / 2 = 25
25 / 2 = Remainder
25 / 3 = Remainder
25 / 4 = Remainder
25 / 5 = 5
5 / 2 = Remainder
5 / 3 = Remainder
5 / 4 = Remainder
5 / 5 = 1

Therefore, the largest prime factor of 100 is 5.

Once my answer has reached one, I know that I have reached my greatest prime factor. This is an efficient way of doing the algorithm because it means that I am not constantly processing the large number that we are to find the prime factor of, but rather are deducting its size as a number as we divide it until we find the largest prime factor of that number. This makes sense because all we are doing is simply slowing reducing the number as we change its ratio through division until we reach a certain point where we can't divide by anything other than 1 or itself, which is what a prime number's definition is.
So, how would we approach this problem programmatically in Python? Well, the first thing we must do is again, create a loop, but right before it, create a boolean that declares the MaxPrimeFound variable as false. Once we have this complete, we are able to process a loop from 1 to the number's variable itself. While we are processing, we try dividing the variable being affected in the loop's process by a factor of 2 for example, and checking whether or not it is a whole number. If it is a whole number, we continue on. If it is not a whole number, we increment the dividing factor until we reach a point where it divides evenly. If the number that it divides evenly is in to itself, then we know we have indeed found the largest prime factor of the large digit number. We simply repeat this process until it divides into itself and becomes 1. Good luck!

Solution using their example:

```
'''The prime factors of 13195 are 5, 7, 13 and 29. What is the largest prime factor of the number
600851475143 ? '''

    #number = 10
    #number = 25
    #number = 1319555555
    #number = 600851475143
    number = 13195
    largestPrimeFactor = 0

    # 13195 / 29 = 455
    # 13195 / 13 = 1015

    factor = 2
    MaxPrimeFound = False
    found = 0

    while MaxPrimeFound is False :
        for f in range(factor, number+1):
            if (((number / f) % 1).is_integer()):
                if number is f:
                    found = f
                    MaxPrimeFound = True
                    break
                #print("Number being tested:", number, "is divisible by",f)
                factor = f
                largestPrimeFactor = f
                number //= factor
                break
            elif (number // f) is 1:
                found = f
                MaxPrimeFound = True
```

```
if (found > largestPrimeFactor):
    largestPrimeFactor = found
elif (largestPrimeFactor > found):
    largestPrimeFactor = largestPrimeFactor

print("Largest Prime Factor is:", largestPrimeFactor)
```

Output: The largest prime factor is 29.

This is correct in the circumstance of 13915. Let’s figure out the problem’s solution though. Solution for the problem:

''' The prime factors of 13195 are 5, 7, 13 and 29. What is the largest prime factor of the number 600851475143 ? '''

```
#number = 10
#number = 25
#number = 1319555555
number = 600851475143
largestPrimeFactor = 0

# 13195 / 29 = 455
# 13195 / 13 = 1015

factor = 2
MaxPrimeFound = False
found = 0

while MaxPrimeFound is False :
    for f in range(factor, number+1):
        if (((number / f) % 1).is_integer()):
            if number is f:
                found = f
                MaxPrimeFound = True
                break
            #print("Number being tested:", number, "is divisible by",f)
            factor = f
            largestPrimeFactor = f
            number //= factor
            break
        elif (number // f) is 1:
            found = f
            MaxPrimeFound = True

if (found > largestPrimeFactor):
    largestPrimeFactor = found
elif (largestPrimeFactor > found):
    largestPrimeFactor = largestPrimeFactor

print("Largest Prime Factor is:", largestPrimeFactor)
```

Output: The largest prime factor is 6857.

I have now went over how your thought process should be for the following programs and how you should be thinking when approaching these mathematical programming problems on ProjectEuler and programming in general. Here are some of the solutions for the first 12 problems on the site in case you get stuck:

10001 st Prime Number:

```
import time


'''

By listing the first six prime numbers: 2, 3, 5, 7, 11, and 13, we can see that the 6th prime is 13.

What is the 10 001st prime number?

'''

# 10,001 :: 104743

start_time = time.time()

elementTermIndex = 6
```

```
elementTermGoal = 10001
currentPrime = 13

save = [2, 3, 5, 7, 11, 13]

# 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

def FindPrimeAfter (num):
    index = num + 1
    PrimeFound = False
    while (PrimeFound is False ):
        #for x in range(2, (index-1)//2):
        for x in save:
            if (index % x) is 0:
                PrimeFound = False
                index += 1
                break
            else :
                PrimeFound = True
    return index

while (elementTermIndex < elementTermGoal):
    print("Element",elementTermIndex,":",currentPrime)
    currentPrime = FindPrimeAfter(currentPrime)
    save.append(currentPrime)
    elementTermIndex += 1

print("Element",elementTermIndex,":",currentPrime)

print("Time Executed:",time.time() - start_time)
```

Largest Palindromic Number:

'''

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is 9009 = 91 × 99.

Find the largest palindrome made from the product of two 3-digit numbers.

'''

```
digits = 3

start = 1 * (10**(digits-1))

end = 1 * (10**digits-1)

LargestPalindromic = 0

for x in range (start, end+1):

    for y in range (start, end+1):

        product = x * y

        productText = str(product)

        if (productText._len_() % 2) is 0:

            limit = productText._len _ () // 2

            left = 0

            right = 0

            Palindromic = False

            for z in range (0, limit):

                left = (int(productText[z]))

                right = (int(productText[(productText.__len__()-1)-z]))

                if (left == right):

                    Palindromic = True

                else :

                    Palindromic = False

                    break

            if (Palindromic is True ):

                if (LargestPalindromic < (int)(productText)):

                    LargestPalindromic = int(productText)
        else :
            pass

print("The largest palindromic number from the product of two",digits,"digit numbers is:",LargestPalindromic)
```

Largest Product in a Grid:

```
import math

'''

In the 20×20 grid below, four numbers along a diagonal line have been marked in red.

What is the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the 20×20 grid?

'''

string_grid = """08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08

49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00

81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65

52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91

22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80

24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50

32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70

67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21

24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72

21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95

78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92

16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57

86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58

19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40

04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66

88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69

04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
```

```
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48
"""

def ConvertToInt (grd):
    grid = []
    curr = ""
    for n in grd:
        if n is not " " and n is not "\n":
            curr += n
        else :
            grid.append(int(curr))
            curr = ""
    return grid

def DisplayGrid (grd):
    for n in grd:
        print(n)

def FindGreatestProduct (grd):
    product = 0
    for x in range(0, len(grd)):

        left = x >= (20 * math.floor(x / 20))+3
        right = x <= (20 * math.ceil(x / 20))-4
        up = x >= (20*3)
        down = x <= (len(grd) - (20*3))-1

        if up is True and left is True :
            p = grd[x] * grd[x-21] * grd[x-42] * grd[x-63]
            if p > product:
```

```
        product = p
if up is True and right is True :
    p = grd[x] * grd[x-19] * grd[x-38] * grd[x-57]
    if p > product:
        product = p
if down is True and left is True :
    p = grd[x] * grd[x+19] * grd[x+38] * grd[x+57]
    if p > product:
        product = p
if down is True and right is True :
    p = grd[x] * grd[x+21] * grd[x+42] * grd[x+63]
    if p > product:
        product = p
if left is True :
    p = grd[x] * grd[x-1] * grd[x-2] * grd[x-3]
    if p > product:
        product = p
if right is True :
    p = grd[x] * grd[x+1] * grd[x+2] * grd[x+3]
    if p > product:
        product = p
if up is True :
    p = grd[x] * grd[x-20] * grd[x-40] * grd[x-60]
    if p > product:
        product = p
if down is True :
    p = grd[x] * grd[x+20] * grd[x+40] * grd[x+60]
    if p > product:
        product = p
    return product

grid = ConvertToInt(string_grid)
print("The largest product is: " + str(FindGreatestProduct(grid)))
#DisplayGrid(grid)
```

Largest Product in a Series

```
series = """96983520312774506326239578318016984801869478851843
85861560789112949495459501737958331952853208805511
12540698747158523863050715693290963295227443043557
66896648950445244523161731856403098711121722383113
62229893423380308135336276614282806444486645238749
30358907296290491560440772390713810515859307960866
70172427121883998797908792274921901699720888093776
65727333001053367881220235421809751254540594752243
52584907711670556013604839586446706324415722155397
53697817977846174064955149290862569321978468622482
83972241375657056057490261407972968652414535100474
82166370484403199890008895243450658541227588666881
16427171479924442928230863465674813919123162824586
17866458359124566529476545682848912883142607690042
24219022671055626321111109370544217506941658960408
07198403850962455444362981230987879927244284909188
84580156166097919133875499200524063689912560717606
05886116467109405077541002256983155200055935729725
71636269561882670428252483600823257530420752963450"""

adjacent = 13
iIndex = adjacent-1
fIndex = adjacent-1

series = series.replace('\n','')
```

```
product = 1
save_iIndex = 0
save_fIndex = adjacent - 1

for n in range(adjacent - 1, series._len_()):
    iIndex = n - (adjacent - 1)
    fIndex = n
    newProduct = 1
    for x in range(iIndex, fIndex+1):
        newProduct *= int(series[x])
    if (newProduct > product):
        product = newProduct
        save_iIndex = iIndex
        save_fIndex = fIndex

print("The numbers: ", end="")
for x in range(save_iIndex, save_fIndex+1):
    if (x != save_fIndex):
        print(series[x],end=" * ")
    else :
        print(series[x],end=" ")
print("produce the largest product which is equal to:",product)
```

Smallest Multiple

```
import time
'''
2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any
remainder.

What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?
'''

start_time = time.time()
```

```
Found = False
NumberFound = 0
start = 1
end = 20
num = 0

inc = 0
for n in range(2, end+1):
    if (n % 2) is 0:
        #print(n)
        inc += n

#print(inc)

while (Found is False ):
    for x in range(start, end+1):
        if ((num % x) is 0 and num is not 0):
            Found = True
            NumberFound = num
        else :
            Found = False
            break
    num += inc

print("The smallest positive number in the range given is:",NumberFound)

print("\nTime Executed:", time.time() - start_time)
```

Special Pythagorean Triplet

```
import math

# a < b < c

a = 1
```

```
b = 2
c = math.sqrt(a**2 + b**2)

# 1. Find a Triplet
# 2. Check if a + b + c = 1000
# 3. If it is, multiply all terms to get product

for aCheck in range(1, 501):
    for bCheck in range (1, 501):
        c = math.sqrt(aCheck**2 + bCheck**2)
        if (c % 1).is_integer():
            if (aCheck < bCheck and int(c) > bCheck):
                if (aCheck + bCheck + c == 1000):
                    product = (aCheck * bCheck * int(c))
                    print("The three terms are:",aCheck,bCheck,int(c))
                    break

print("The product is:",product)
```

Summation of Primes:

```
import math
import time

limit = 2000000

start_time = time.time()

terms = [2]
sumOfPrimes = terms[0]

for n in range(3, limit, 2):
    FoundPrime = True
    for t in terms:
        if n % t is 0:
            FoundPrime = False
            break
        elif t > math.sqrt(n):
            break
    if FoundPrime is True :
        #print(n)
        terms.append(n)
        sumOfPrimes += n

print("The sum of all the primes below",limit,"is:",sumOfPrimes)
print("Time Executed for completion:",time.time()-start_time)
```

Sum Squares Difference

'''

The sum of the squares of the first ten natural numbers is,

1^2 + 2^2 + ... + 102 = 385

The square of the sum of the first ten natural numbers is,

(1 + 2 + ... + 10)2 = 55^2 = 3025

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is 3025 − 385 = 2640.

Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

'''

```
#Find square of sum
squareSum = 0
#Sum of squared terms
squareTermSum = 0
for n in range(1, 101):
    squareSum += n
    squareTermSum += (n**2)
```

```
sumSquared = squareSum**2

difference = sumSquared - squareTermSum
print("The difference is:",difference)
```

Chapter 17

Final Words

This is the start of your journey as a Python programmer. You have barely scratched the surface with this guide as learning the syntax and conventions of a language is just the beginning. The most important part of programming is the logical aspect of it. Sure, you may know how to loop through an array of variables like a list of shopping items but if someone asks you to process an image using your knowledge of programming, and with the help of an API and some thinking, you can figure out how you are able to invert colors of an image, flip it, rotate it, scale it, etc.

The real programming comes in the logical portion of the mind. It’s similar to when you’re learning any other language, like English for example. You may understand the grammar rules and the conventions like adding periods to the end of sentences, but to be able to write clean and logical thought-out and structured essays is where the true skill lies. The same concept applies to programming where the person writing the code, must know how to apply his knowledge of the rules in the considered language, like Python, and use it to his advantage to come up with neat programs.

The knowledge and understanding of programming is truly great because it’s the closest thing to having a power. You can literally create something out of an empty notepad, from scratch and have it function to do things you want it to do. Whether it be a bot to analyze the stock market and come up with predictions or creating a game. That choice is yours.

In this guide, you have learned the fundamentals of Python. You haven’t learned all the possible methods that can be used in the language, but that isn’t the point. The point of this guide was to set you on a journey to discover objects and methods that you need in order to help you to create programs that you desire. You have been given the optimum knowledge to understand reading an API and be able to understand what it is saying and adding to your code.

Good luck as a new-born Python programmer!