**Python Introduction & Python Basics**

**Programming Language**

* Simply a way to give instructions to computers/machines
* Without code, computers and programs would not know what to do
* Computers do not understand English or any other language
* Computers speak in 1s and 0s
* Programming languages bridge the gap…they convert something from English into 1s and 0s so computers can understand
* Source code is written in a programming language
  + After source code is written it still needs to be translated for a computer to understand.
  + Enter INTERPRETERS and COMPILERS
    - Interpreters = goes through code line by line and executes said code
    - Compiler = takes code all at once and translates it to machine code

**Python Interpreter**

* Official language of Python is cPython
* Usually what people are talking about when they say Python is cPython
* When you download Python (from their website) you’re downloading an interpreter
  + cPython creates a byte/bite code (closer to machine code)
  + Then takes bite/byte code and uses cPython VM (virtual machine) to run code

**How To Run Python Code**

* Terminals
* Code editors
* IDEs = Integrated Developer Environments
* Notebooks
* REPL = lets you code and run Python without downloading Python (there’s also glot.io)

**First Python Program**

* print (‘....’)
  + REPL already has the interpreter inbuilt into the website/system
* Input = prompts a user, also a way to capture information

**Python 2 VS Python 3**

* Programming languages are always evolving to become better and better
* Python 3 is the upgraded version of Python 2
* Some companies didn’t upgade/update to the latest version of Python
* From 2020 onwards, no more security with Python 2
* Python 3 is the way to go
  + Be cognizant/alert when searching for answers. Make sure it’s Python 3 and not a later version
* Python 2 is legacy code
* Python 3 is more secure

**Why So Many Programming Languages**

* There are too many languages to learn them all
* Different languages are good for different things
* It’s impossible to have just one supreme language when there are so many different applications
  + You can’t build a house with a hammer as your only tool, you need different tools. The same concept applies to programming. Every language is a different tool, more efficient for one task and not so much for another. It all depends on what you’re trying to do.
* Python as a programming language is actually slower than C++ etc
  + Python however is really good at developer productivity
  + It’s closer to English
  + It can do much more with less lines of code (more complex functions)
* Conclusion = every languages has its pros and cons (there’s always a tradeoff)

**Learning Python**

* 4 things you have to learn when learning a programming languages
  + 1. Terms. The different words and definitions said words. Need to know what they mean so you can communicate with other programmers
  + 2. Data type. What sort of data can the program pull. This can also be thought of as values. Each programming language has its data type.
  + 3. Actions. All programming is essentially telling a computer/program to store this data, retrieve this data, and the perform an action. Need to know how to perform these actions.
  + 4. Best practices. There are good and bad ways of writing code (Airbnb has industry standard of writing/programming languages.

**Data Types**

* Fundamental data types in Python
  + int = integer
  + float = floating point number (number with a decimal point)
  + bool = boolean
  + str = string
  + list
  + tuple
  + set
  + dict
* Classes = custom data types
* Specialized data types = these aren’t built into Python. They are special packages (from libraries). Sort of a boost in some way. Whenever you don’t have a data type that you want in the standard Python package, and you don’t want to create your own type, there are specialized data types that can be used from modules. They are extensions to the language.
* None = means nothing, the absence of value

**Numbers**

* int = integer. float = floating point number (number with decimal point\_
* int
  + Use int to do mathematical operations in a programming language
  + You can’t just put 2+4 into a program and have something happen, there has to be an actions [like print (2+4)]
* There needs to be a distinction between int and float because float takes up a lot more space & memory than an integer (takes a lot more 0s and 1s). int gets stored in one place. float gets stored in two different locations hence more space used.
* Double multiplication sign = to the power of (exponent)
  + 2 \*\* 2 = 22 = 4
  + 2 \*\* 3 = 23 = 8
* Double division sign = round down to integer
  + 2 // 4 = 0
  + 3 // 4 = 0
  + 5 // 4 = 1
* Percent sign = what is the remainder of the division
  + 5 % 4 = 1
  + 6 % 4 = 2

**Math Functions**

* round() = rounding number inside brackets
* abs() = absolute value of what’s inside brackets (absolute = non negative number)
* Math functions for Python can be looked up online for whatever it is that you’re trying to do. You just have to know how to implement it properly.
  + Like learning to speak a language, no point in knowing words if you don’t know how to properly use them.

**DEVELOPER FUNDAMENTALS 1**

* Don’t read the dictionary, you can’t memorize everything
* Using the language is more efficient than reading it, just like speaking an actual language
* Focus on the 20% that gets used 80% of the time. Not 100% and 80% barely gets used.
  + The correlation in spoken language is this…you learn all the words in a language, including the big words but how often do you actually use them? As opposed to smaller words (like, the, please, yes, no, thank you, etc) that get used all the time.

**Operator Precedence**

* Similar to PEMDAS or BODMAS
  + The order in which actions get executed
* () then \*\* then \* then / then + then -

**Additional Data Types**

* Probably won’t use, but good to know that these are out there
  + complex = complex number
    - * Used usually when doing some sort of complet problem
      * Complex numbers are generally combinations of real and imaginary numbers
      * If interested do more research into complex numbers
  + bin = binary
    - * ints and floats get stored in memory as binary numbers (0s and 1s)
      * bin returns the binary representation of a number

**Variables**

* Store information that can be used in our programs
  + They hold user inputs like values
* Just another way for us to store information in our computers
  + Eg… iq = 200, print(iq) will give us 200
* Variables can be called names
* BINDING = assigning to a value to a variable
  + Eg…binding the value “200” to “iq”
* Best practices for variables
  + snake\_case = all variables are written in snake case meaning all lower case and all spaces are replaced with underscores
  + Variables must start with lowercase or underscore
  + Variables can be anything with letters, numbers, and underscores (cannot start with a number though)
  + Variables are case sensitive (which letters are capitalized matters)
    - iq = 200 is not the same as iQ = 200.
  + Variables can’t overwrite key words
    - You can’t assign a value to “print”. print is already a keyword in Python
    - Keywords in Python can be looked up online/googled
* When coding, try and make variables are descriptive as possible
  + Another developer should be able to come and look at your code and understand what it is that you coded
* Variables can be reassigned
  + Example
    - iq = 200
    - user\_age = iq / 4
    - print(user\_age) = 50
* Constants
  + Some variables are constants
  + These are things that never change in the program
  + All CAPS signifies a value/variable that remains constant
* Double Underscore = dunder
  + Python comes with some dunder variables
  + These are not meant to be messed with (should be left alone)
  + It isn’t good practice to create dunder variables
* One of the most important skills as programmer is that when you code, you want it to read like English
* Cheat sheet
  + a,b,c = 1,2,3
  + This is shorthand code writing

**Expressions Vs Statements**

* Expressions = A piece of code that produces a value
  + Example
    - iq = 100
    - user\_age = iq / 2
    - {iq / 2} is an expression
* Statement = An entire line of code that performs an action
  + Example
    - {iq = 100} is a statement
    - {user\_age = iq / 2} is a statement
    - Both statements are assigning values to variables

**Augmented Assignment Operator**

* some\_value = 5
  + some\_value = some\_value + 2
  + print(some\_value) ⇒ 7
  + some\_value += 2 ⇒ print(some\_value) ⇒ 7
* Short hand
  + Helps you change the value of a variable without ha ving to type it all out
* Works with +, -, \*, /
* Operator comes on the left side of the equal [=] sign

**Strings**

* Simply a piece of text
* Can be written with quotation marks (single or double quotes)
* You can put anything in quotation marks
* 3 single quotes [‘’’]
  + Used for long or multiple line strings
  + Used if/when you need to type out a paragraph or something with long sentences
* Can used + sign to add strings together

**String Concatenation**

* Simply means adding strings together
  + ‘hello’ + ‘ ‘ + ‘Suleiman’ ⇒ concatenation of strings
  + print(‘hello’ + ‘ ‘ + ‘Suleiman’) ⇒ hello Suleiman
* Can’t add strings to integers
  + print(‘hello’ + 5) ⇒ error

**Type Conversion**

* Data can be converted to different types
* str(100) = converts integer 100 to a string
* int(str(100)) = converts integer 100 to a string and then to integer
* Another way to do this…
  + a = str(100)
  + b = int(a)
  + c = type(b)
  + print(c) ⇒ <class “int”>

**Escape Sequences**

* A sequence of characters that when used inside a character or string does not represent itself, but rather is converted into another character or sequence of characters that are difficult or impossible to express.
  + An example of this is a string with quotes in it, Python may think it’s two different things (two different strings)
    - weather = ‘it’s sunny’
      * print(weather) ⇒ error
    - weather = ‘it\’s sunny’
      * print(weather) ⇒ It’s sunny
    - weather = “it’s \”kind of\” sunny”
      * print(weather) ⇒ It’s “kind” of sunny
  + \t = tab = add tab before whatever comes after
  + \n = new line = whatever comes after goes on a new line

**Formatted Strings**

* Very useful in dynamic coding/programming
  + Example
    - name = Suleiman
    - age = 34
      * print(‘Hi ’ + name + ‘ ! You are ‘ + str(age) + ‘ years old.’)
      * ⇒ Hi Suleiman! You are 34 years old.
  + That works but there’s a better way
    - Use f in the beginning = lets Python know it is a formatted string
    - After using f, use squiggly brackets { } for variable
      * Example
        + print(f ‘Hi {name}! You are {age} years old.)

⇒ Hi Suleiman! You are 34 years old.

* + - f before tells Python it is going to be a formatted string and I want you to make the variables available as strings
  + :format = another way of doing to the same thing
    - has to be used on the same line of code that you want formatted
    - has to be used within the parenthesis of what you want formatted
    - Has to be the order in which you want the variables formatted
    - Since programming is in 0s and 1s, you have to format accordingly

**String Indexes**

* Helps with string manipulation
  + Python has a feature allowing you to grab different pieces of text from string
* [ ] = square brackets
  + Lets you access different parts of a string
  + The first in a square bracket is the start (basically where we want the program to look)
  + One Colon “:” inside the square brackets helps indicate where to start and stop
    - Example
      * selfish = “01234567”
      * print(selfish[0]) ⇒ 0
      * print(selfish[0:1]) ⇒ 01
      * print(selfish[0:7]) ⇒ 0123456
  + Two Colons “: :” inside the square brackets helps indicate where to start, where to stop, and what to stepover
    - Example
      * selfish = “01234567”
      * print(selfish[0:8:1]) ⇒ 01234567
        + Stepover one is also the default of Python
      * print(selfish[0:8:2]) ⇒ 0246
  + [start: ] = indicates where to start and go to the end
  + [ :stop] = indicated start at the beginning and stop where I tell you
  + [ : :1] = start at the beginning, stop at the end, stepover one
  + [-1] = start at the end of the string (or give me the end of the string)
  + [ : : -1] = start at the beginning, stop at the end, in reverse order

**Immutability**

* Strings in Python are immutable, meaning that they cannot be changed
  + selfish = “01234567” can be changed to selfish = 100
  + However cannot do selfish[0] = “8”
    - The value of a string cannot be changed once it’s created
    - Only way to do that change is to reassign it
  + A part of a string cannot be reassigned
    - The only way to change it is to create something new
      * Example
        + selfish = “01234567”
        + selfish = selfish + “8” OR selfish += “8”
        + print(selfish) ⇒ 012345678

**Built-In Function & Methods**

* len = length of string
  + print(len(‘Suleiman’)) ⇒ 8
* Python has built-in functions and methods
* Built in functions are basically predefined functions in Python
  + They can be used without having to import anything (modules)
* Built in methods in Python are functions that are associated with specific data types
  + Some examples include string methods, list methods, etc
  + Each built-in data type comes with its own set of methods
* All the necessary functions and methods are in Python documentation (can be looked up online).
* String Methods
  + Methods used specifically for strings
  + They have special syntax
    - .format() = a method
    - Example
      * .upper
      * quote = “to be or not to be”
      * print(quote.upper()) ⇒ TO BE OR NOT TO BE
      * .upper = everything capitalized
    - Example
      * .capitalize
      * print(quote.capitalize()) ⇒ To be or not to be
      * .capitalize = only the first letter of string is capitalized
    - Example
      * .find
      * print(quote.find(‘be’)) ⇒ 3
      * .find = find first occurrence of the string/word ‘be’
    - Example
      * .replace
      * print(quote.replace(‘be’ , ‘me’)) ⇒ to me or not me
      * .replace = replace every occurrence of of ‘be’ with ‘me’

**BOOLEANS**

* bool
* A bool can either be true or false
  + In application it can be used to control the flow of a program
  + Meaning…if something is true, do this
  + If something is false, do something else
* In boolean 1 = true, 0 = false
* Example
  + print(bool(0)) ⇒ false
  + print(bool(1)) ⇒ True
  + print(bool(17)) ⇒ True
  + print(bool(‘True’)) ⇒ True
  + print(bool(‘False’)) ⇒ True

**Exercise: Type Conversion**

* Example
  + name = ‘Magnificent’
  + age = ‘50’
  + relationship\_status = ‘single’
  + relationship\_status = ‘it\’s complicated’
  + print(relationship\_status) ⇒ it’s complicated
* Example
  + birth\_year = input(‘what year were you born?’)
  + age = 2023 - int(birth\_year)
  + print(f’your age is: {age}’) ⇒ What year were you born?...1989…your age is: 34

**DEVELOPER FUNDAMENTALS 2**

* Commenting your code
  + ‘#’ before typing comments your code
  + There are good and bad ways to comment your code
  + Comments have to add value, add valuable information
    - Helps the code read like English
  + Good to comment when something important is happening
    - It might also be a little complex
  + More comments doesn’t necessarily mean better code
  + Try and keep code concise and neat

**Exercise: Password Checker**

* input prompts user to give variable
* Variables are then processed and defined
* Example
  + username = input(‘what is your username?’) …magnificent
  + password = input(‘what is your password?’) …password
  + password\_length = len(password)
  + hidden\_password = ‘\*’ \* password\_length
  + print(f’hey {username}, your password {hidden\_password} is {password\_length} letters long.’) ⇒ hey magnificent, your password \*\*\*\*\*\*\*\* is 8 letters long.

**Lists**

* An ordered sequence of objects that can be of any type
* Lists (li) are denoted with square brackets [ ] …similar to string indexes
* li are often referred to in other languages as arrays (similar at least)
* A data structure
  + A way for us to organize information and data into folders, boxes, cupboards etc, so they can be used with different pros and cons
  + In essence it’s a container with your data

**List Slicing**

* Similar to string slicing
* [start:stop:stepover] = this method applicable to lists just like in strings
* Unlike strings which are immutable, lists are mutable = lists can be changed
  + Example
    - amazon\_cart = [‘notebook’, ‘sunglasses’, ‘toys’, ‘grapes’]
    - amazon\_cart[0] = ‘laptop’
    - print(amazon\_cart) ⇒ [‘laptop’, ‘sunglasses’, ‘toys’, ‘grapes’]
  + Example
    - new\_cart = amazon\_cart
    - new\_cart[0] = ‘gum’
    - print(new\_cart) ⇒ [‘gum’, ‘sunglasses’, ‘toys’, ‘grapes’]
    - print(amazon\_cart) ⇒ [‘gum’, ‘sunglasses’, ‘toys’, ‘grapes’]
    - This is because we made new\_cart = amazon\_cart. So they have to be the same. Another way to do this if you just want to copy the list and then change a part of it…next example
  + Example
    - new\_cart = amazon\_cart [:] ….in this case all you’re doing is copying the list
    - new\_cart[0] = ‘gum’
    - print(new\_cart) ⇒ [‘gum’, ‘sunglasses’, ‘toys’, ‘grapes’]
    - print(amazon\_cart) ⇒ [‘notebook’, ‘sunglasses’, ‘toys’, ‘grapes’]

**Matrix**

* A way to describe 2D or multidimensional lists
* Topics like this come up a lot in machine learning and image processing
* Basically it’s a list inside a list (array inside an array)
* The same concept from list slicing can be used to access different parts of a matrix
  + Example
    - matrix = [

[1,5,1]

[2,7,9]

[4,6,3]

]

* print(matrix[0][1]) ⇒ 5 ||| print(matrix[1][2]) ⇒ 9

**List Methods**

* These are list specific functions
* len = just like in strings can be used to see length of list
  + Example
    - basket = [1,2,3,4,5]
    - print(len(basket)) ⇒ 5
* .append = adds something to end of list. Doesn’t change list or produce a value
  + Example
    - new\_list = basket.append(100)
    - print(basket) ⇒ [1,2,3,4,5,100]
    - print(new\_list) ⇒ None
    - There’s a better way to do it which is
      * Example
        + basket.append(100)
        + new\_list = basket
        + print(new\_list) = [1,2,3,4,5,100]
* .insert = inserts something to specified place(index) of list
  + .insert modifies list in place, doesn’t copy list
  + Example
    - basket.insert(4,100)
    - print(basket) ⇒ [1,2,3,4,100,5]
    - basket,insert(5,100)
    - print(basket) ⇒ [1,2,3,4,5,100]
* .extend = extends the list with whatever is in the parenthesis
  + Example
    - basket.extend([100,101])
    - print(basket) ⇒ [1,2,3,4,5,100,101]
* .pop = by itself, pops off the last item of list
  + Example
    - basket = [1,2,3,4,5]
    - basket.extend([100])
    - basket.pop()
    - print(basket) ⇒ [1,2,3,4,5]
    - basket.pop() || now done twice
    - print(basket) ⇒ [1,2,3,4]
  + .pop with a value in parenthesis removed the specified index value from list
    - Example
      * basket = [1,2,3,4,5]
      * basket.pop(0)
      * print(basket) ⇒ [2,3,4,5]
* .remove = removes a specific value from list
  + Example
    - basket = [1,2,3,4,5]
    - basket.remove(4)
    - print(basket) ⇒ [1,2,3,5]
* .clear = removes whatever is in the list, completely clears is it
  + Example
    - basket = [1,2,3,4,5]
    - basket.clear
    - print(basket) ⇒ [ ] ….basket is empty

**List Methods 2**

* .index() = gives index of specified value
  + Example
    - basket = [1,2,3,4,5]
    - print(basket.index(3)) ⇒ 2
    - Essentially you’re asking the computer which index can I find 3
    - basket = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
    - print(basket(‘d’)) ⇒ 3
* in = keyword in Python
  + Helps you see if something exists within something else
  + In this case it’ll check and see if value exists within list
    - Example
      * basket = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
      * print(‘d’ in basket) ⇒ True
      * print(‘x’ in basket) ⇒ False
      * This works for strings as well
        + print(‘i’ in ‘hi my name is Ian’) ⇒ True
* .count() = counts how many times an item or value occurs
  + Example
    - print(basket.count(‘d’)) ⇒ 1

**List Methods 3**

* .sort = sorts the list in place
  + Example
    - basket = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘d’]
    - basket.sort
    - print(basket) ⇒ [‘a’, ‘b’, ‘c’, ‘d’, ‘d’, ‘e’]
    - It only sorts the list in place, it doesn’t create a new list
    - sorted() function can be used in this case to create a new list without modifying original list
* .copy = copies the list in place and creates a new list
* .reverse = reverses the list in place

**Common List Patterns**

* Slicing creates a new version of the list
* List functions (list methods) usually just modify the list in place
* range() = a good function to generate a list quickly
  + print(list(range(1,101))) ⇒ [1,2,3,4,.....100]
  + print(list(range(101))) ⇒ [0,1,2,3,4,....100]
  + This helps with loops
* .join() = actually a string method, but can be applied here to add a list to a string
  + Example
    - sentence = ‘ ‘
    - new\_sentence = sentence.join([‘hi’, ‘my’, ‘name’, ‘is’, ‘Magnificent’]
    - print(new\_sentence) = hi my name is Magnificent
    - Another way (shorthand)
      * Example
        + new\_sentence = ‘ ‘.join([‘hi’, ‘my’, ‘name’, ‘is’, ‘Magnificent’])
        + print(new\_sentence) = hi my name is Magnificent

**List Unpacking**

* Basically shorthand in list creating or coding
  + Example
    - a,b,c = 1,2,3
    - a,b,c = [1,2,3]
  + Example
    - a,b,c,\*other = [1,2,3,4,5,6,7,8,9]
    - print(other) ⇒ [4,5,6,7,8,9]
  + Example
    - a, b, c, \*other, d = [1,2,3,4,5,6,7,8,9]
    - print(d) ⇒ 9

**None**

* The absence of value.
* There’s nothing there

**Dictionaries**

* dict = a data type in Python. It is also a data structure (like lists etc)
* dict uses curly brackets {}
* dicts have keys and values (key:value pairs)
  + Example
    - dictionary = {
      * + ‘a’: 1,
        + ‘b’: 2
        + }
      * In this case ‘a’ is a key used to grab the value 1
* A dictionary is an unordered key:value pair
* You can put anything in a dictionary (lists, strings, integers, etc) as long as it has a key to access it
* Dictionary keys must be immutable. They can’t be changed.

**DEVELOPER FUNDAMENTALS 3**

* Understanding data structures
* Should be able to know what data structure to use and when
* Lists would be good for something with people in line at your store
* Dictionary would be good for gaming (user, weapons, profile specifics, etc…these would be keys that values could be attached to)
* Dictionaries also hold a lot more information
  + You can have a dictionary inside a list and vice versa
* Learning which data structure to use and when is something that comes with time and practice as a developer/coder

**Dictionary Keys**

* Dictionary keys need to have a special property
  + Dictionary keys must be immutable (they can’t be changed)
  + A key in a dictionary has to be unique (one of one)
    - If you use the same key twice, the last one written will overwrite the previous one

**Dictionary Methods**

* .get = helps access a key, to see if it even exists
  + Example
    - user = {
      * ‘basket’ = [1,2,3]
      * ‘greet’ = ‘what\’s good’
      * ‘b’ = 2
      * }
    - print(user.get(‘age’)) ⇒ None …absence of value, doesn’t exist
    - print(user.get(‘age’, 55)) ⇒ 55
      * This basically means, grab the age key from the dictionary and give me the value, if there’s no age/age doesn’t exist, use 55
* Another way to create a dictionary
  + Example
    - user2 = dict(name = ‘JohnJohn’)
    - print(user2) ⇒ {‘name’: “JohnJohn’}

**Dictionary Methods 2**

* in = similar use like in list methods
  + Example
    - print(‘basket’ in user) ⇒ True
    - print(‘size’ in user) ⇒ False
* .keys() = simply checks keys in dictionary
  + Example
    - print(‘greet’ in user.keys()) ⇒ True
    - print(‘yes’ in user.keys()) ⇒ False
* .values() = simply checks values in dictionary
  + Example
    - print(‘hello’ in user.values()) ⇒ False
* .items = grabs entire item
  + This will be better understood after explanation of Tuples
* .clear() = clears the dictionary (modifies the existing dictionary)
* .copy() = copies the contents of dictionary and creates a new dictionary
* .pop() = removes the key and value selected
  + Example
    - print(user.pop(‘b’)) ⇒ 2
    - print(user) ⇒ returns dictionary without ‘b’ key:value pair
* .popitem() = removes last key;value pair from dictionary
* .update() = updates key:value pair
  + Example
    - print(user.update({‘b’: 45})) ⇒ None
    - print(user) ⇒ returns dictionary with ‘b’ updated value of 45

**Tuples**

* Another data structure
* They are basically immutable lists
* Regular parentheses ( ) help denote a tuple
  + Example
    - my\_tuple = (1,2,3,4,5)
    - my\_tuple[1] = ‘f’
    - print(my\_tuple) ⇒ error…because tuples can’t be changed
* Although a tuple can’t be changed, it can be accessed through indexing
* Benefits of tuples (pros and cons)
  + You don’t need to change the list
  + It tells other programmers, other people viewing your code that it shouldn’t be changed
  + It makes things easier and more efficient
  + Makes code safer and more predictable, but less flexible
  + Uber user needs a fixed location, using tuple would be good in that instance
  + Uber driver changes location, would be better to use a list
* A tuple can be in a dictionary
  + It can be a dictionary key since dictionary keys are immutable

**Tuples 2**

* Tuples can be sliced
  + Example
    - my\_tuple = (1,2,3,4,5)
    - new\_tuple = my\_tuple[1:4]
    - print(new\_tuple) ⇒ [2,3,4]
  + Example
    - x, y, z, \*other = (1,2,3,4,5)
    - print(x) = 1
    - print(y) = 2
    - print(z) = 3
    - print(other) = [4,5]

**Tuple Methods**

* There are two main tuple methods
* .count() and .index()
* count() = how many occurrences in tuple
  + Example
    - my\_tuple = (1,2,3,4,5)
    - print(my\_tuple.count(5)) ⇒ 1
  + Example
    - my\_tuple = (1,2,3,4,5,5)
    - print(my\_tuple.count(5)) ⇒ 2
* .index() = when is the first occurrence in tuple
  + Example
    - my\_tuple = (1,2,3,4,5,5)
    - print(my\_tuple.index(5)) ⇒ 4
* len function will also work, gives you the length of a tuple

**Sets**

* Sets are an unordered collection of unique objects
* Uses curly brackets { } like dictionaries but no key:value pairs
* There are no duplicates in sets. Everything has to be unique
  + Example
    - my\_set = {1,2,3,4,5,5}
    - print(my\_set) ⇒ {1,2,3,4,5}
* .add = a method that adds values to set
  + Example
    - my\_set = {1,2,3,4,5,5}
    - my\_set.add(100)
    - print(my\_set) ⇒ {1,2,3,4,5,100}
* Type conversion can be used to change a list into a set in order to remove duplicates
  + Since sets must have unique objects
* Indexing doesn’t work with sets
  + In this way it is more similar to dictionaries
  + You have to grab the actual item/object that’s in the set
* in function also works with sets
  + Example
    - my\_set = {1,2,3,4,5}
    - print(1 in my\_set) ⇒ True
* len functions works with sets as well
  + Example
    - my\_set = {1,2,3,4,5,5}
    - print(len(my\_set)) ⇒ 5
      * Returns 5 and 6 because only unique objects counted
* .clear() = clears set
* .copy() = copies set and creates new set

**Sets 2**

* There are many set methods
* For future examples use:
  + - * my\_set = {1,2,3,4,5}
      * your\_set = {4,5,6,7,8,9,10}
* .difference() = shows the difference between sets (doesn’t modify set)
  + Example
    - print(my\_set.difference(your\_set)) ⇒ {1,2,3}
* .discard() = removes an element from a set if it is a member
  + Example
    - my\_set.discard(5)
    - print(my\_set) ⇒ {1,2,3,4}
* .difference\_update() = removes all elements of another set from this set, modifies the set
  + Example
    - my\_set.difference\_update(your\_set)
    - print(my\_set) ⇒ {1,2,3}
* .intersection() = shows where two sets intersect, shows what they have in common
  + Example
    - print(my\_set.intersection(your\_set)) ⇒ {4,5}
    - Another way to do this
      * print(my\_set & your\_set) ⇒ {4,5}
* .isdisjoint() = shows if sets have anything in common
  + Example
    - print(my\_set.isdisjoint(your\_set)) ⇒ False
      * Means they aren’t disjointed, they have 4, 5 in common
* .union() = combines/unites two sets, makes them a union
  + Example
    - print(my\_set.union(your\_set)) ⇒ {1,2,3,4,5,6,7,8,9,10}
      * Units the two sets and removes duplicates, creates a new set
    - Another way to do this
      * print(my\_set | your\_set) ⇒ {1,2,3,4,5,6,7,8,9,10}
* .issubset() = my\_set.issubset(your\_set)
  + Essentially asking if all of my\_set can be found in your\_set
  + print(my\_set.issubset(your\_set)) ⇒ True
* .issuperset() = my\_set.issuperset(your\_set)
  + Essentially asking is all of your set can be found in my\_set
  + print(my\_set.issuperset(your\_set)) ⇒ False
  + print(your\_set.issuperset(my\_set)) ⇒ True

**Python Basics II**

**Breaking The Flow**

* Conditions and conditional logic
* Looping
* Skipping lines of code
  + Running or evaluating certain codes based on conditions etc

**Conditional Logic**

* if = a keyword in Python = conditional operation
  + Will only run following code when condition is met
    - Example
      * is\_old = True
      * if is\_old:
        + print(‘you are old enough to drive’) ⇒ you are old enough to drive
      * is\_old = False
      * if is\_old:
        + print(‘you are old enough to drive’ ⇒ nothing happens
* else = keyword in Python
  + Will run following code when ‘if’ conditions not met or is False
  + Will not run following code when ‘if’ condition is True, or condition is met
* elif = used in combination with if
  + Used to check additional conditions when if conditions aren’t met
  + When if evaluated to False, elif is triggered
* and = keyword in Python
  + Logical keyword used in boolean expressions
  + Often used to combine multiple conditions in an if statement
    - Example
      * if is\_old and is\_licensed:
        + print(‘you can now drive’)
      * When both is\_old and is\_licensed are True
        + ⇒ you can now drive
* You can have multiple elif conditions but only one if and one else statement in conditional logic

**Indentation in Python**

* Indentation helps with organization of code
* In Python, indentation of a line of code is interpreted as that line of code being part of the previous line or a particular block of code
* If using spaces in Python, the standard is 4 spaces
* The important part is the distinction and setting hierarchies in code
* Code editors usually help with this

**Truthy Vs False**

* Truthy value in Python, is a value that when bool type conversion is run on it, it evaluate to True
* Falsey values in Python are those values that when you run a bool type conversion on them, it evaluates to False
* This is really good for conditional logic

**Ternary Operator**

* Another way to do conditional logic
* The same as if, elif, else statements but it is a shortcut
* Can only be used in certain conditional logic
* Ternary operators can also be called conditional expressions
  + Expressions evaluate to a value
* Ternary operator/conditional expression is an operation that evaluates to something based on the condition being True or False
  + Example
    - is\_friend = True
    - can\_message = ‘message allowed’ if is\_friend else ‘not allowed’
    - print(can\_message) ⇒ message allowed
  + Example
    - is\_friend = False
    - print(can\_message) ⇒ not allowed
* Usual format for ternary operator is as follows
  + condition\_if\_true if condition else condition\_if\_false

**Short Circuiting**

* This essentially means that the evaluation of an expression stops as soon as the final result is determined
* The ‘and’ and ‘or’ operators exhibit short circuiting in Python
  + Example
    - is\_friend = True
    - is\_user = True
    - print(is\_friend and is\_user) ⇒ True ⇒ because they’re both true
  + Example
    - if is\_friend or is\_user:
      * print(‘best friends forever’) ⇒ best friends forever
    - Since we only care if one of the variables is true, it skips the second part of the code because the first part is true

**Logical Operators**

* Very useful when it comes to conditional logic
* ‘and’ is a logical operator
* ‘or’ is a logical operator
* Logical operators allow us to perform logic between things
* logical operators ⇒ > , < , ==,
  + In Python == is used for equal sign because one equal sign is used ot assign things)
* print(‘a’ > ‘b’) ⇒ False.
  + This is because when you compare strings in Python it is done in lexicographical (dictionary) order. The > comparison checks the unicode code points of characters in strings. The point for ‘a’ is smaller than the point for ‘b’ as such it evaluates to false.
* More logical operators
  + >= ⇒ greater than or equal to
  + <= ⇒ less than or equal to
  + != ⇒ not equal to (opposite to equal or ==)
  + not ⇒ also a function, does the opposite

**is vs ==**

* ==
  + Checks for equality
  + It compares
  + It’s a logical operator
  + Evaluates as true if two things/items have the same value
  + Example
    - print(True == 1) ⇒ True
    - print(‘ ‘ == 1) ⇒ False
    - print(10 == 10.0) ⇒ True
    - print([ ] == 1) ⇒ False
    - print([ ] == [ ]) ⇒ True
* Is
  + Keyword in Python
  + Checks whether the location of value in memory is the sme
  + Different from ==
  + It is more strict than == which checks for value
  + Location must be the same to evaluate as True
  + Example
    - print(True is 1) ⇒ False
    - print(True is True) ⇒ True
    - print(‘1’ is 1) ⇒ False
    - print(‘1’ is ‘1’) ⇒ True
    - print([ ] is [ ]) ⇒ False …each list/array is stored in a different place in memory
    - print(10 is 10.0) ⇒ False … floats and integers are stored in different places

**For Loops**

* Allows you to run lines of code over and over and over
  + This is where machines excel
  + They can do small tasks, a lot of times, in a small amount of time
  + One of the most powerful features or programming languages
* Creating a loop is as simple as using ‘for’ (called for loops in Python)
  + ‘for’ similar to ‘if’ in conditional logic
    - Example
      * for item in ‘Zero to Mastery’:
        + Item can be anything, any variable in ‘Zero to Mastery’
      * In this case ‘Zero to Mastery’ is an iterable
        + Iterable is something that’s able to be looped over
* For loops allows us to iterate over anything that has a collection of items
* For loops can be nested within each other
  + There is always operator precedence

**Iterable**

* An object or a collection that can be iterated over
* Can be a list, dictionary, tuple, set, string
  + These are iterables because they can be iterated
  + Iterated in this case means you can go one by one to check each item in the collection
* Similar to… dictionaries methods can be used in loops/iterables
  + .items, .values, .keys = allow you to iterate over dictionaries
* Integers aren’t iterable
  + Integers aren’t a collection

**Exercise: Tricky Counter**

* Using loops to find sum of items in list/array
  + Example:
    - my\_list = [1,2,3,4,5,6,7,8,9,10]
    - Easy way
      * sum(my\_list[0:10:1]) ⇒ 55
    - Looping way
      * counter = 0
      * for item in my\_list:
        + counter = counter + item
        + print(counter) ⇒ 55

**range()**

* When looping range() is one of the most common functions
* A function that comes with Python
* You can iterate over a range
* range() has 3 parameters
  + range(start, stop, stepover)
* range() creates a special type of object that can be iterated over
  + If you want iteration in reverse, make sure to add ‘-1’ in stepover slot
  + range will also have to be specified in reverse order
    - Example
      * for x in range(10,0):
        + print(x) ⇒ nothing happens
      * for x in range(10, 0, -1)
        + print(x) ⇒ range in reverse order
* Wrapping range with list is a quick way to create a list
  + list(range(start, stop, stepover)
    - Example
      * print(list(range(10))) ⇒ [0,1,2,3,4,5,6,7,8,9]
  + When done in loops
    - for x in range(2):
      * print(list(range(10))) ⇒ 2 lists [0…9]

**Enumerate**

* Another function, not as popular as range
* enumerate function is a concise way to iterate over both the index and elements of sequence or collection
  + Example
    - for index, char in enumerate(list(range(100))):
      * if index == 50:
        + print(index) ⇒ 50
    - Another way
      * print(f’index of 50 is: {index}’) ⇒ index of 50 is: 50

**While Loops**

* Another way to loop
* while condition: ⇒ means while conditioning is happening do something
  + Example
    - i = 50
    - while i < 50:
      * print(i) ⇒ evaluates to 0 nonestop
    - This is called an infinite loop
* break = helps solve the issue of infinite loops, lets program know to break out of while loop
* To specify how many times you want to loop
  + Example
    - while i < 100:
      * print(i)
      * i += 1
        + ⇒ evaluates to 50…99 then stops
* To get out of a while loop you can either ‘break’, or turn the condition to False
* else blocks can also be used in while loops
  + Won’t see else blocks too often in while loops (just an option)
  + else will not work if there if there is a break
  + else condition only works if the while loop becomes False

**While Loops 2**

* When to use a while loop or a for loop will depend on the problem you’re trying to solve
* while is usually followed by some condition
* while loops often come after a variable is assigned
* while loops more powerful (for loops simpler)
* while loops, you usually have to create a variable,
  + Remember to increment the variable so you don’t create an infinite loop
  + Remember to halt the loop at some point
* Simple loop rules to follow
  + For simple loops and iterating over iterable objects, for loops are great
  + Maybe you’re not sure how many times you want/need to loop over something, you’re not sure how long looping will take…use while loops
  + At the end of the day, use whatever solves your problems
  + As more experience is gained with coding, you’ll know which one to use

**break, continue, pass**

* break
  + Keyword to break out of a loop
  + Works in for loops as well as while loops
* continue
  + Keyword in Python loops used to tell program to skip the rest of the code inside the loop
* pass
  + Essentially does nothing, just passes to the next line of code
  + Useful as a placeholder while coding
    - Use pass in loop code when you haven’t decided what to do
  + Rare in production code, more common in development

**Our First GUI**

* GUI = graphical user interface
* The default end in print() is a new line
  + To print next code on the same line you need to specify
    - end=’’ ⇒ this tells the program next code goes on same line
* Combine looking things up with already gained knowledge

**DEVELOPER FUNDAMENTALS 4**

* What is good code?
  + Good code is clean
    - Follows a style that the Python community endorses
    - Auto format exists in repl to help with this
    - Clean code follows the best practices
    - Python has standard ways of using spaces
    - Try not to have extra stuff that you don’t need
  + Good code has readability
    - A lot this is based on personal preferences
    - This is where the commentary usually comes into play
      * Comments help clarify what it is you’re doing
    - Makes sure you’re not the only one that understands the code
  + Good code has predictability
    - Avoid using new features and obscure tools just for the sake of it
      * It doesn’t make you look smarter
    - The simpler the better
    - Should be able to predict what will happen, or what’s about to happen while reading coe
    - Very good as code gets larger, predictability gets more important
  + Good code is DRY = do not repeat yourself
    - Don’t want to have code where you’re constantly repeating yourself
    - Making code reusable

**Exercise: Find Duplicates**

* count() used to find the number of times something occurs in a list

**Functions**

* print(), input(), bool()
  + Functions allow you to perform actions on data types
* You aren’t limited to the functions provided by Python
  + You can create your own functions
* def = lets Python know we are about to create a function (short for define)
  + After def comes the variable/word about to be made into a function
  + Make sure to add () after to let Python know it’s a function,
    - Lets program know it’s going to perform an action on a data type
    - : follows the ()
      * Example
        + def say\_hello():

print(‘what\’s good’)

* + () helps the interpreter know you want to run function
* Defining functions helps tremendously with developer fundamentals (DRY)
  + Helps with not repeating yourself
  + When you have an extended/long lines of code you want to run multiple times
    - Instead of repeating it over and over, whole action can be defined as a function and just repeat that
    - Defined functions help keep your code DRY
  + Make sure to define functions in the beginning
    - Define functions before using it

**Parameters and Arguments**

* After functions are defines, parameters can be given in parentheses
  + Can put as many parameters as you want in defined functions
  + Each parameter in a defined function is a variable that can be used in the code or definition of the function
* Arguments are the actual values that correspond to the parameters
  + Simply put, arguments are values assigned to corresponding parameters
* Parameters and arguments corresponding to each other are called positional arguments
  + This means that the position we put them matters
    - When defining function and when using said functions

**Default Parameters and Keyword Arguments**

* Positional arguments are required to be in the proper position
  + Example
    - def variable(parameter 1, parameter 2):
    - Then when using variable it must be
      * variable(argument 1, argument 2)
      * In the case where it’s variable(argument 2, argument 1)
        + That’s messed up code
* Keyword arguments = these allows us to not worry about position
  + Tells program specifically what we want arguments to be
    - Example
      * def function(parameter 1, parament 2)
        + Then when using function
        + function(parameter 2=’’, parameter 1=’’)
        + In this form/example the order or position doesn’t matter, program already knows what to do based on keywords
      * Bad practice to have arguments out of order or position even when using keyword arguments
* Default parameter = help defined function work even if coded the wrong way later in code
  + Basically fall back arguments in the case that arguments aren’t specified when using defined functions
    - Example
      * def function(parameter 1=’hey’, parameter 2=’girl’)
      * function() ⇒ hey girl
      * function(‘what’, ‘up’) ⇒ what up

**return**

* Keyword in Python, will see it a lot in functions
* return statement is used in functions to specify the value the function should output
  + It essentially sends a value back to the caller of the fuc
    - Example
      * def sum(num1, num 2):
        + num1 + num 2
        + return
      * print(sum(4,5) ⇒ 9
* Functions either modifies something in our program or returns something
  + Functions should do one thing really well
  + Functions should return something
* Having a function that does multiple things or a function that doesn’t return something isn’t necessarily bad
  + It’s just good practice to make code simple and readable governed by those rules
* return keyword automatically exits the function

**Exercise: Tesla**

* Wrap already created code with defined function
* Toggle between asking for input or just giving an argument
* Giving parameter a default value allows function to work without specified arguments

**Methods Vs Functions**

* Functions are called with (), then we give it data, or leave it empty
* Methods have to be owned by something
  + Methods are owned by functions by functions or data types aka objects
  + There are ways to build your own methods
  + Can learn more about methods in Python documentation
* Both methods and functions allow us to take action on our data types

**Docstrings**

* Essentially allows you to comment inside of a defined function
* Docstrings are used to describe the purpose, usage, and behavior of the code or function created
* 3 single quotes denote the opening and closing of docstrings ⇒ ‘’’ …docstring…’’’
  + Example
    - def test(a):
      * ‘’’
      * Info: this functions tests and prints parameter a
      * ‘’’
      * print(a)
    - test(yo) ⇒ yo
* help = a function in Python to find out what something does (gives docstring of function)
* .\_\_doc\_\_ = dunder method that gives you the docstring of a function
  + Example
    - print(test.\_\_doc\_\_) ⇒ gives info/docstring of test function

**Clean Code**

* Do not need unnecessary code
* If you have a code checking one thing, and that’s the only thing you care about
  + Don’t necessarily need to check the opposite
  + Just need to tell program what to do if what is checked for is false
* There is no perfect solution
  + You can always clean up code
* Get rid of extra unnecessary lines
* Think like this will lead to development of an advanced programming mindset or how advanced programmers think

**\*args And \*\*kwargs**

* \*args = arguments. \*\*kwargs = keyword arguments
* By adding \* before a parameter in defined fucntion, it allows you tell program that it can accept any number of positional arguments or arguments period
  + Example
    - def super\_func(\*args):
      * return sum(args)
    - print(super\_func(1,2,3,4,5) ⇒ 15
  + \*args as a parameter makes the arguments you give it into a tuple of said arguments
* By adding \*\*kwargs in parameter in defined function, it allows you to tell program that it can accept any number of keyword arguments
  + \*\*kwargs as a parameter makes keyword arguments into a dictionary of said arguments with keys and values
  + Example
    - def super\_func(\*args, \*\*kwargs)
      * Now super\_func() can have as many arguments and keyword arguments as you want
* There are rules to the order you can put parameters in defined functions
  + function(parameter, \*args, default parameters, \*\*kwargs)
  + Has to be this order if using multiple parameters
  + Usually only using one or two of these at a time

**Exercise: Functions**

* Finding the highest even number in a lit
* Introduction of the in built max() function
  + Example
    - highest\_even(li):
      * evens = []
      * for items in li:
        + if items % 2 == 0:

evens.append(items)

* + - * Return max(evens)
    - print(highest\_even([2,10,2,3,4,8,11,15])) ⇒ 10

**Walrus Operator**

* := ⇒ this denotes the walrus operator
* Walrus operator allows you to assign a value to a variable as part of an expression
  + Particularly useful in while loops and list comprehensions
* Example (without walrus operator)
  + a = ‘what it is, what it do playa’
  + if len(a) > 10:
    - print(f’too long {len(a)} elements.)
    - ⇒ too long 28 elements.
* Example (with walrus operator)
  + if (n := len(a)) > 10:
    - print(f’too long {n} elements.’)
    - ⇒ too long 28 elements
* Essentially a way to minimize calculations that are similar
* Look out for new Python features in new Python versions (updates)
  + Language is always evolving

**Scope**

* Simply means what variables do I have access to (this is computer or program asking this)
* Understanding scope is crucial to avoid naming conflicts, manage variable lifetimes, and creating modular & maintainable code
* While coding, when you make variable: total = 100
  + It is now part of a global scope
  + Anyone in file now has access to total variable
  + Can be used inside a conditional block
  + Can be used inside of a function
* If a variable is created inside of a block of code, it isn’t global
  + it only exists in that world (world of the block of code)
  + An example of this is creating a variable while defining a function
  + That variable is only available within that function
  + Nowhere else within that file does the variable exists
  + It can be created again outside of the defined function
    - But that wouldn’t be good practice
    - Code wouldn’t be DRY

**Scope Rules**

* 1 - Starts with local scope
* 2 - Parent local
* 3 - Global (indentation of nothing/no indentation)
* 4 - Built in Python
* These are the rules that the Python interpreter follows when executing functions etc

**Global Keyword**

* Parameters in defined functions are considered to be local variables
  + They can only be used within that function
  + Can’t be used anywhere else in code outside of function
* global = keyword in Python
  + Used to designate a variable within a function as a global scope
  + Tells program that variable created within function can be used outside of function
  + Designating a variable a global allows it to be accessed and modified inside as well as outside of a defined function
    - Example
      * total = 0
      * def count():
        + total += 1
        + Return total
      * print(count()) ⇒ 1
    - Better way to do this is dependency injection
      * Look up for better explanation/understanding
      * Helps avoid the need to create global variables within defined functions

**nonlocal Keyword**

* New keyword in Python 3
* It’s a way to refer to parent local (from scope rules)
* Essentially saying I want to use a variable that isn’t global, but outside of the scope of my function
* Usually will be used in special cases (not very often)
  + Like closures
* After designating variable as nonlocal
  + If that variable is then modified or assigned a new value
  + It also changes/modifies/assigns the new value to the variable
* nonlocal variables are there for a reason (useful in some instances)
  + Same developer fundamentals rules apply
  + Make code predictable, clean, and DRY

**Why Do We Need Scope?**

* Machines don’t have infinite power, cpu, memory, they all have limited resources
  + As such have to be conscious what resources we use
  + Can cost money, and crash computers
* An example of this is making a variable nonlocal
  + Instead of creating a whole new variable, program just grabs already created and modifies it
* Another example of this is in a defined function
  + After defining function, when function is called and output given, memory is erased
  + Avoids clogging computers memory
* Helps program run smoothly and avoid crashes, etc.

**Actions/Functions**

* print() = show result of what’s in parenthesis
* type() = tell me what type of data is in parenthesis
* input() = prompt to collect information
* int() = converts what’s in parenthesis to integer
* float() = convert what’s in parenthesis to number with decimal (floating point number)
* str() = converts what’s in parenthesis to string
* round() = round what’s in parenthesis
* len() = gives the length of string in parenthesis

<https://medium.com/ayuth/iterm2-zsh-oh-my-zsh-the-most-power-full-of-terminal-on-macos-bdb2823fb04c>