**Python\_Bootcamp Notes**

# Background Information

## VS Code

* Visual Studio Code, also commonly referred to as VS Code, is an integrated development environment developed by Microsoft for Windows, Linux, macOS and web browsers
* Basically it is an interface that allows you to write and edit code in a variety of languages
* Can open Jupyter notebook and use python through it

## GitBash

* At its core, Git is a set of command line utility programs that are designed to execute on a Unix style command-line environment
* Modern operating systems like Linux and macOS both include built-in Unix command line terminals – Windows instead uses the windows command prompt
* Git Bash is an application for Microsoft Windows environments which provides an emulation layer for a Git command line experience

## Python

* Python is an **interpreted**, **high-level**, **general-purpose** programming language.

## Anaconda

* Anaconda is a Python and R distribution software. It aims to provide everything you need for Python “*out of the box*.”
* Helps to ensure that previous formats of python which are incompatible, can be opened
* Create an environment – then can download software into it and operate within that environment
  + Pip is an online repository to download programs in anaconda

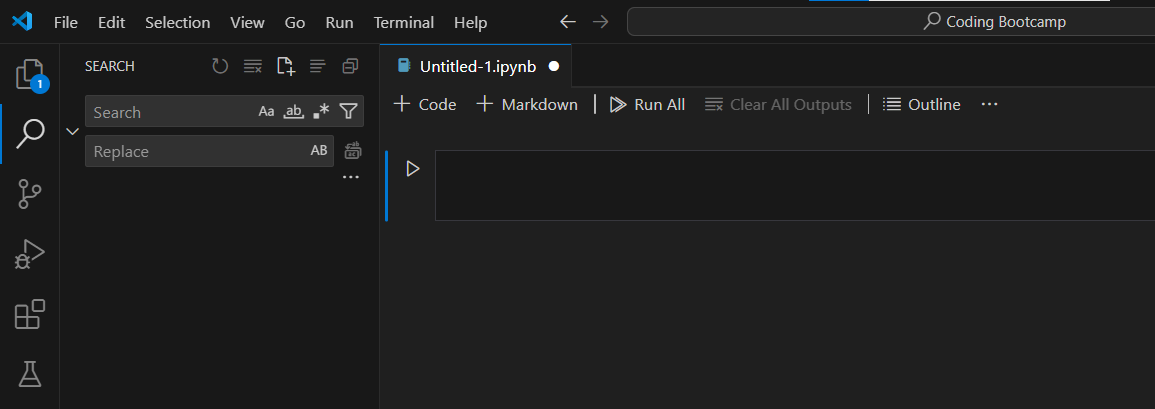
## Jupyter Notebook

* It is an open-source integrated development environment (IDE) that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. For us, it’s essentially our notebook, where we will code along together

# Basics

## Opening a new jupyter notebook

* In VS Code, you can go File > New file
* A prompt will come up to select your file type – choose jupyter notebook

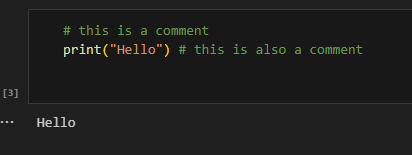


* Code cell: Cells that are executable
* Markdown cell: Text cell – use # to change font size
* Ctrl enter: Executes cell

# Week 2 – Python Basics:

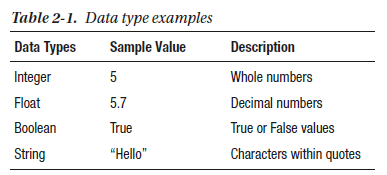
## Comments:

* Comments are like notes that you leave behind, either for yourself or someone else to read.
* In Python, we can write comments using the hash (#) symbol. Any text that follows this symbol will be commented out.



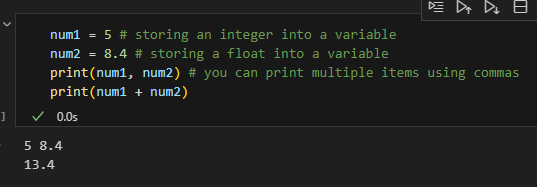
## Data Types:

* Data types are how we define values, likes words or numbers.



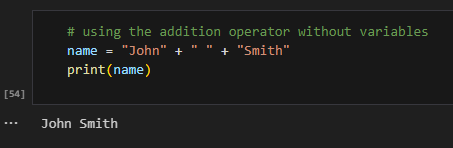
## Variables:

* They allow us to save values into memory using a name that we assign. This lets us use those values later in the program
* We declare a name on the left side of the equals operator (“=”), and on the right side, we assign the value that we want to save to use later.
* Note: Variable names can contain only letters, underscores, and numbers; however, they cannot start with a number.



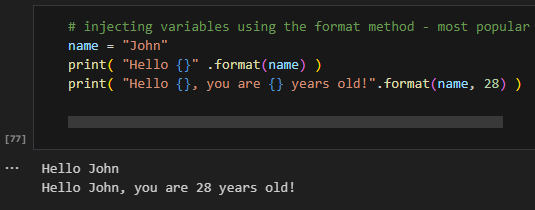
## String concatenation:

* Add one string to the end of another.



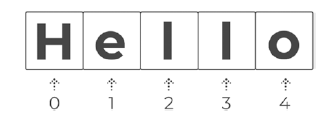
## Formatting Strings (.format()):

* Earlier we created a full name by adding multiple strings together to create a larger string. While this is perfectly fine to use, for larger strings it becomes tough to read
* The format method works by putting a period directly after the ending string quotation, followed by the keyword “format”. Within the parenthesis after the keyword are the variables that will be injected into the string.
* The order of the curly brackets is the same order for the variables within the format parenthesis.
* To include multiple variables in one format string, you simply separate each by a comma



## String Index:

* When a computer saves a string into memory, each character within the string is assigned what we call an “**index.**” An index is essentially a location in memory.
* Note: Indexing in most languages, including python, starts at 0 not 1.
* In order to index a specific element, you use square brackets to the right of the variable name. Within those square brackets, you put the index location you wish to access.



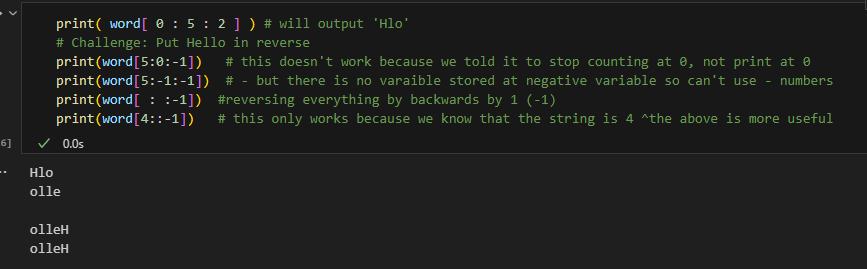
word = "Hello"

print( word[ 0 ] ) # will output 'H'

## String Slicing:

* When only want a piece of the variable

Formatted as *variable\_name[ start : stop : step ]*



## String Manipulation:

.title()

* Often, you’ll run into words that aren’t capitalized that should be usually names. The title method capitalizes all first letters in each word of a string.

.upper() and .lower()

* Converts whole word to capital or lowercase

.replace(“replace this” , “with this”)

* Works like the find and replace tool

.find(“the string we’re searching for”)

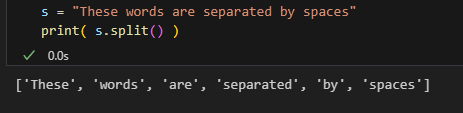
* Output is the starting index of the term

.strip( )

* Default removes spaces (can also .lstrip() and .rstrip()

.split()

* Returns a group of words in a sentence stored as a list



# Week 3 pt 1 - GitBash Crash Course:

Bash is programming language – script based language – file browser and file manipulator

But also allows us to execute and run programs – basically everything windows does for us - but condensed into a single command line

**Terminology:**

My computer is the client, the computer I am trying to login into is the host. They both have an ‘address’, ie. An IP address

* ssh - Way to interact with every type of computer and can be used to monitor and manage remote computers.
* Follows the command: ssh username@hostaddress (example below where the username is ai\_crew and the host is the number
* Will usually then be prompted to enter a password

**pwd** – print working directory: (shows us where we are at)

**~** stands for directory we are in

**ls :** lists the files in that directory (ie. The folders in that folder)

**ls –rtlah** : prints everything in the location including hidden files and who owns them and what types they are

**drwxrwx-x** : from left to right – d= directory, first rwx = user can read, write and execute, next rwx = the group can read, write and execute, last x = anyone using the compute can execute

**mkdir NAME\_HERE**: make directory and name it

**touch name.filetype**: ie. touch test.txt – makes a file in git bash

**cat name.filetype**: displays whats in it (dumps it all)

**less test.txt** :shows you it line by line as opposed to one big thing – q exits

**command >> name.filetype** :outputs the command into the file as opposed to displaying it on the screen

**^** : means ctrl

**nano filename.type** : takes you to edit the file – once you’ve finished editing it Crtl O (write out), enter, Ctrl X (exit)

**cd .** :directory I am currently in

**cd ..** : takes you back one – can use multiple ../../ if you want to get way back

**cd -** :takes you back to where you just got out of if you used the ../

**exit:** gets out of ssh session

**mv oldfilename.filetype newfilename.filetype**: renames a files

**history** : shows you all commands you have used

# Week 3 pt 2 - User input and Type converting:

## Accepting User input:

Like the print function, input will print the string inside of the parenthesis, but it will also create a box for the user to enter information

* Note: Information entered is taken into the program as a string.

## Storing User input:

In order to work with the data that they enter, we need to store it into a variable

Var = input(“Your question here”)

## Type checking:

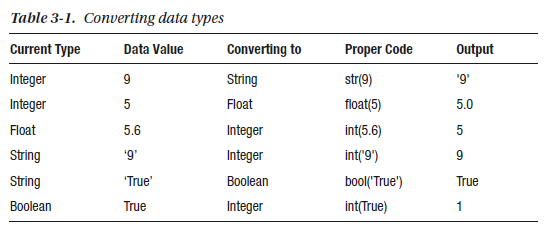
Type conversionfunctions to directly convert one data type to another which is useful in day-to-day and competitive programming

Important because If you are expecting a number to be input, you’ll need to convert the input to an integer data type, so that you’re able to work with it (bc input will automatically be outputted as a string

Print(type(variable)) – will print the variable type

## Type converting:

You can convert the input by wrapping the type you want it to be. Ie. Str(var) will convert the variable to a string



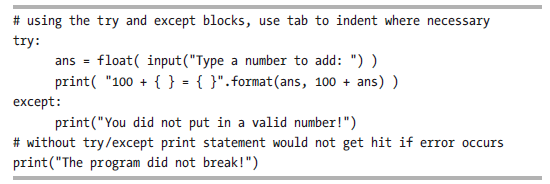
Also important for getting user input:

Ie. If you ask a question which you expect to be a interger you need to define the variable as int(input(QUESTION))

## Handling errors:

As a developer, we must assume that the user won’t put the proper information that we expect them to.

Try and except are used to catch errors. It works by trying to run what is inside the try block; if it doesn’t produce an error, then it continues without hitting the except block; however, if an error occurs, then the code in the except block runs. This is to make sure your program doesn’t stop running if an error pops up



Note: indent needs to be consistent

## If statements:

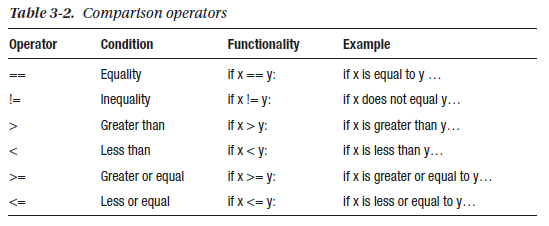
All branching statements begin the same way, with the keyword “if”. Following the keyword is what is known as a condition. Lastly, there will always be an ending colon at the end of the statement. The if statement checks to see if the given condition is True or False. If the condition is True, then the code block runs. If it is False, then the program continues without running any of the code indented directly after the if statement:

x = 10

y = 5

if x > y:

print(“x is bigger than y)



## Logical operator – and:

Checks both sides of the condition are true and you can have as many conditions in one line as you’d like:

if \_\_ and \_\_:

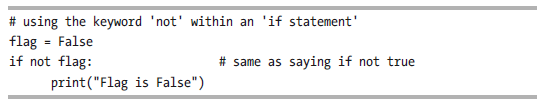
## Logical operator – or:

Checks one or both – ie. Code will run if at least one of them is true

If \_\_ or \_\_:

## Logical operator – not:

Essentially returns the opposite of whatever the current value is.

If not \_\_\_:

Get the same answer as if flag==false

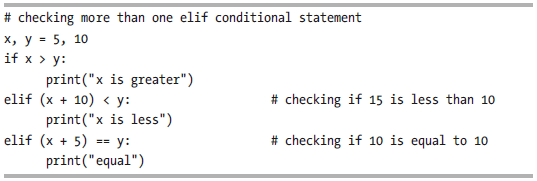
## Membership operators:

Checks if a sequence appears in an object – “in” and “not in”

## Elif Statements:

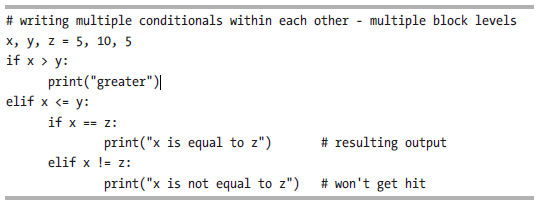
Gives us the ability to run separate blocks of code depending on the condition. They are also known as “else if statements.”

Must be associated with an if statement - as python works from top to bottom, so checks the first if, then the elif and if false as well it keeps reading through.



Note: Within the conditional, we perform addition, but we wrap it within parenthesis so that it executes the math operation first.

You can use multiple lines of conditionals and put if statements within elif statements to further differentiate code:

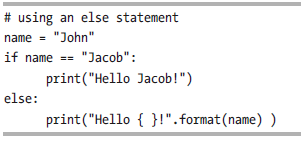


As they are all related to a single if statement, as soon as a line is correct, the block will stop being read

## Else Statements:

Else conditional statements are the end all be all of the if statement. Else statement will cover all other possibilities not covered and will always run the code if the program gets to it.

you don’t need to write a condition at all; you just need to provide the keyword “else” followed by an ending colon. Remember that an else clause will run the code inside of it if the program reaches the statement.



# Week 4 pt 1: Lists and Loops:

## Lists:

A list is a data structure in Python that is a mutable, ordered sequence of elements. Mutable means that you can change the items inside, while ordered sequence is in reference to index location.

* Listname = [list item 1, list item 2]

Accessing items in list based on index (index always begins at 0)

* Print(Listname[index])

You can access a list within a list

* By using multiple square brackets
* Print(listname[index in list][index within indexed item])

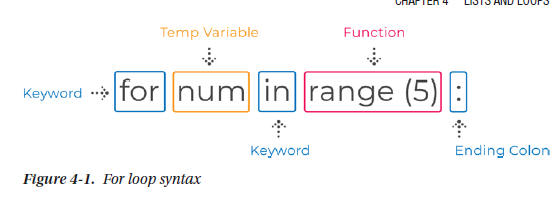
Changing values in a list:

* Re-declare a variable with the index
* List[0]=new item

Copying a list into a new variable using colon:

Data\_copy = data[:]

## Loops pt 1 – For loops:



Loops are how programmers rerun the same lines of code several times – will always run until a condition is met

Temp variable – can be anything we want it to be

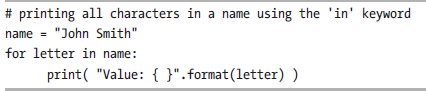
Function – can be a variety of things as well (range() is a built in function)

* for num in range(5):
* print( "Value: {}".format(num) )
* Value: 0
* Value: 1
* Value: 2
* Value: 3
* Value: 4

This loop is essentially counting to five and printing out each number. When the for loop is created, the range function begins at zero by default and assigns the value of zero into our temporary variable num. Each time through the loop is what we call an iteration. For each iteration, once all the code within the block runs, the current iteration is finished, and the loop starts over again at the top. Except this time, it increments the value of num, which by default is 1. Our temporary variable is assigned the value of 1 and continues to run the lines of code inside the for loop, which is simply printing out the value of num. It will continue to do this until we reach the number 5.

**Looping by element:**

* When data types are iterable, ie. String you can define a variable and then write a for loop

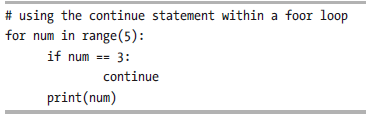


**Range function:**

for num in range(start, stop, step)

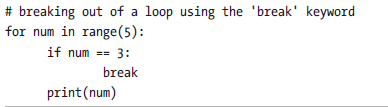
**Continue statement:**

Can use with an if statement and stops the current iteration and starts the next one.



**Break statement:**

Allows us to break out of a loop



**Pass Statement:**

Simply a placeholder so that the program doesn’t break 0 can add a #TODO comment to remind yourself to come back

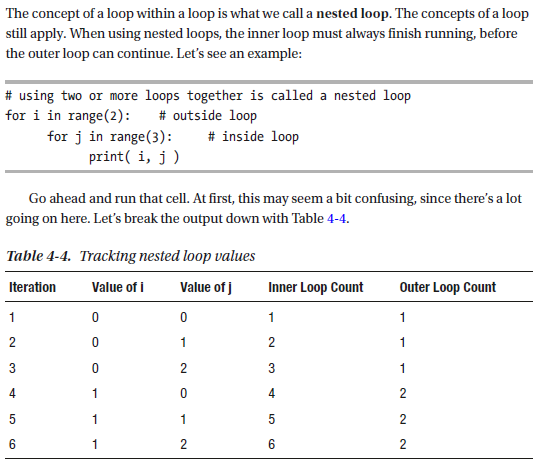
## Loops pt 2 – While Loops:

A **while** loop is generally used when you need to loop based on a condition rather than counting.

The pass, break, and continue statements all work the same way for while loops as well

Infinite loops are bad – will continue until program breaks or computer is shut down

**Nested Loops**:



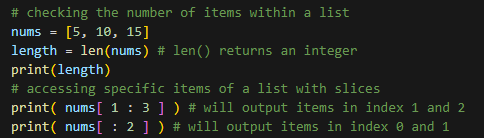
## Working with Lists – Functions to use

1. **Checking length**

len(list)

1. **Slicing lists**

Follows same argument as range function (start, stop step)



1. **Adding iems:**
   1. **List\_name.append(item)**
   2. **List\_name.insert(index, item)** 
      1. Requires an index to insert
2. **Removing Items:**
   1. **List\_name.pop()**
      1. Automatically removes last item
   2. **List\_name.remove(item\_to\_remove)**
      1. Often need to use a try-except function in case the word isn’t in the list

## Working with Numerical Data:

1. **Min, max, sum**
   1. Format – min/max/sum(lis\_name)
2. **Sorting a list:**
   1. **Sorted(list\_name)**
      1. Work with numerical or alphabetical but not a mixed list – returns a copy, doesn’t alter original
   2. **List\_name.sort()**
      1. Changes list directly

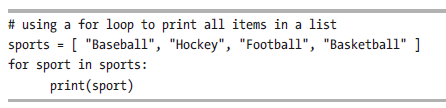
## Conditionals and lists

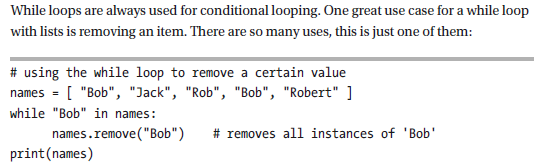
1. **“In” and “not in”:**
   * 1. If “word” in list:

Command

## Loops and lists:

You can use both the for and while loops to iterate over the items within a list





# Week 4 pt 2: SSH into remote host to copy files:

Scp = want to copy from our source to a destination and back again

-v = verbose (prints a lot of info)

. = my home directory

Space = new destination

Scp -v ai\_crew@130.194.216.51:~/bootcamp/muller\_abbey/haiku.txt .

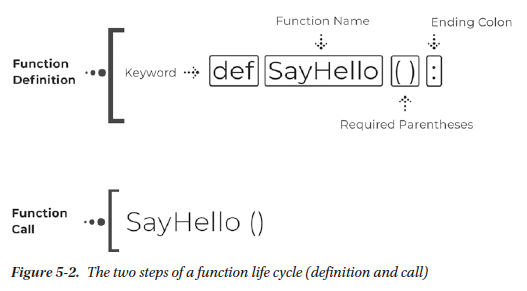
Reverse – can go both directions:

Scp – v ./haiuku.text ai\_crew@130.194.216.51:~/bootcamp/muller\_abbey/haiku.txt

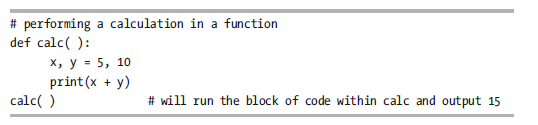
# Week 5 – Functions:

## Functions

A function is a block of code which only runs when it is called – saves you from having to write the same info multiple times

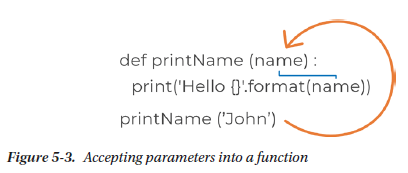


Example:



## Parameters

Temporary variables declared on the function definition



MUST BE IN ORDER-

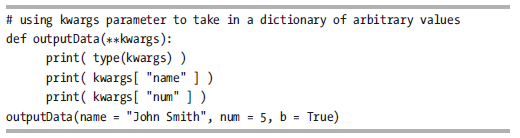
* Can add multiple parameters by adding a comma
* Can also define a list then use that as a parameter
* Can define a parameter using = if you want it to be associated with a default values
  + Ie. def calcarean(r, pi=3.14)
* Can set an optional argument using = “”

## \*Args

\*args allows you to pass a variable number of arguments into a function (the args isn’t the important bit, it’s the \*

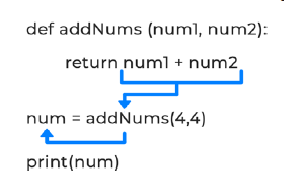
## \*\*kwargs

kwargs allows us to take in an arbitrary number of values in a function; however, it works as a dictionary with keyword arguments instead. Keyword arguments are values passed in with keys, which allow us to access them easily within the function block.

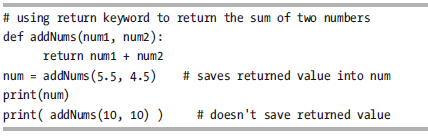


## Return:

Allows you to access information later



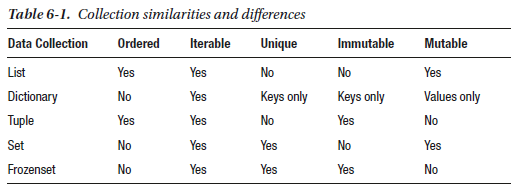
We can return the value and save it into a variable that we can work with later.



## Scope

* Accessibility of variables declared within a program
* In Python, there are three types of scope: global, function, and class.
  + Global: declare a variable to be accessible to an entire file or application
  + Function: variables being declared and accessible only within functions.

# Week 6 – Data collection and Files:



## Dictionary

Collection of unordered data, which is stored in key-value pairs

* Accessed through a key
* the name of the variable goes to the left of the equals operator, and on the right is the dictionary.
* All dictionaries are created by using open and closed curly brackets. In between the curly brackets, we define our key-value pairs.
* Keys can be declared with ONLY strings or numbers.

# accessing dictionary information through keys

person = { "name": “John" }

print( person[ "name" ] ) # access information through the key

* Dictionaries can be defined as lists using [ ] after the :
* Access items in list using [“key”][index]

**Dictionary within a dictionary:**

# storing a dictionary within a dictionary and accessing it

data = {

"team": "Boston Red Sox",

"wins": { "2018": 108, "2017": 93 }

}

print( data["wins"] ) # will output the dictionary within the wins key

print( data["wins"]["2018"] ) # first access the wins key, then the next key

**Adding new info or changing existing to a dictionary:**

Variable\_Name[“newKey/oldKey”] = “NewValue”

**Deleting information:**

Use del function – need to use Try/Except otherwise will crash

car = { "year": 2018 }

try:

del car["year"]

print(car)

except:

print("That key does not exist")

**Looping dictionaries:**

* To iterate through a dictionary while only accessing the keys, you’ll use the .keys() method

person = { "name": "John", "age": 26 }

for key in person.keys( ):

print(key)

print( person[key] ) # will output the value at the current key

* When you don’t need to access the keys, using the .values( ) method is best

person = { "name": "John", "age": 26 }

for value in person.values( ):

print(value)

* If you need the ability to access both the key and value, then you’ll want to use the .items() method

person = { "name": "John", "age": 26 }

for key, value in person.items( ):

print( "{ }: { }".format(key, value) )

## Tuples, Sets and Frozensets

* A **tuple** is identical to a list, except it is immutable – cannot be altered once declared
  + To declare a tuple, you use a comma to separate two or more items within a parenthesis
* A **set** is a collection of information like a list; however, like a key in a dictionary, sets can only contain unique values. They are also an unordered collection. This means that they cannot be accessed by index but rather by the value itself like dictionary keys
  + There are two ways to declare a set
    - first way is by using the keyword “set” followed by parenthesis and enclosing square brackets.
    - second way looks like a dictionary being declared by using a set of curly brackets

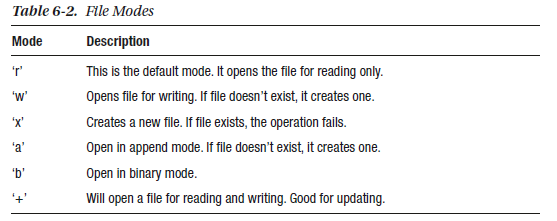
s1 = set( [1, 2, 3, 1] ) # uses the set keyword and square brackets

s2 = {4, 4, 5} # uses curly brackets, like dictionary

* Frozensets are essentially the combination of a set and a tuple. They are immutable, unordered, and unique. These are perfect for sensitive information

## Reading and Writing files:

* Python comes with an open() function that allows us to create or modify files.
* This function accepts two parameters, the file name, and the mode
* If the file name exists, then it will simply open the file for modification; otherwise, it will create the file for you. The mode is in reference to how Python opens and works with the file



1| # opening/creating and writing to a text file

2| f = open("test.txt", "w+") # open file in writing and reading mode

3| f.write("this is a test") # write() method to write our sentence to the file

4| f.close( ) # **Anytime you open a file, you must always close it.**

5| # reading from a text file

6| f = open("test.txt", "r")

7| data = f.read( )

8| f.close( )

9| print(data)

## CSV Files:

* CSV files work with data by separating a comma between each cell
* Python has a default library called “csv.” We’ll need to import that in order to work with them.
* After importing this library, we’ll use the second method of opening files using the “with” keyword. This concept works like a while loop, so that while the file is open, we can work with it, and once the block of code is done running, it closes the file automatically for us.

1| # opening/creating and writing to a csv file

2| import csv

3| with open("test.csv", mode="w", newline="") as f: #new line blank so no new rows

4| writer = csv.writer(f, delimiter=",")

5| writer.writerow( ["Name", "City"] )

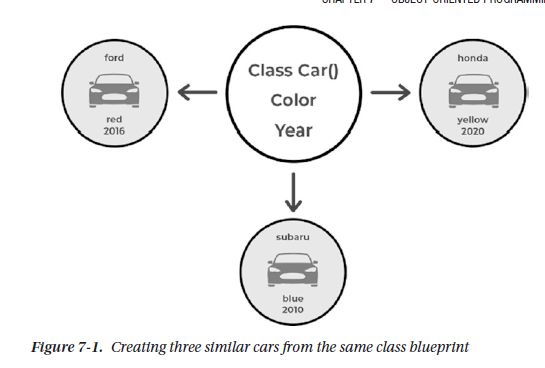
6| writer.writerow( ["Craig Lou", "Taiwan"] )

* In order to read the data from the CSV file we just created, we can simply set the mode to read:

# Week 7 - Object Oriented Programming:

Objects shorten the amount of code – like blueprints – helpful for repetitive tasks

## Creating and Instantiating a class



Two stages when using classes:

1. Class Definition:
   1. this stage is where you write the blueprint to be used when called
   2. Use class keyword instead of def
2. Creating an instance:
   1. the process of creating an object from the class definition – can have multiple instances

Attributes: Personalised features or attributes within a class

# how to define a class attribute

class Car( ):

    sound = "beep" # all car objects will have this sound attribute and its' value

    color = "red" # all car objects will have this color attribute and its' value

ford = Car( )

print(ford.color) # known as 'dot syntax'

#access an object’s attribute, you use dot syntax. You start by writing the name of the instance, followed by a dot and the attribute you want to access.

Change an attribute using the dot syntax as well.

ford = Car( )

print(ford.sound) # will output 'beep'

ford.sound = "honk" # from now on the value of fords sound is honk, this does not affect other instances

print(ford.sound) # will output 'honk'

## \_\_init\_\_( ) Method

To instantiate an object with specific properties, you need to use the initialization (init) method. Whenever an instance is created, the init method is called immediately. You can use this method to instantiate objects with different attribute values upon creation. This allows us to easily create class instances with personalized attributes.

It also includes the “self” keyword inside of the parenthesis as a mandatory parameter.

* The self keyword is a reference to the current instance of the class and is used to access variables and methods associated with that instance.

# defining different values for multiple instances

class Car( ):

    def \_\_init\_\_(self, colour, year):

        self.color = colour # sets the attribute colour to the value passed in

        self.year = year

ford = Car("blue", 2016) # create a car object with the color blue and year 2016

subaru = Car("red", 2018) # create a car object with the color red and year 2018

print(ford.color, ford.year)

print(subaru.color, subaru.year)

## Global vs instance attributes:

Global attributes can be referenced by the class directly and all its instances, whereas instance attributes (which are defined within the init method) can only be accessed by the class instances.

## Methods:

* Actions in classes – ie. Functions within a class
* Defining a method is the same as defining a function; however, you simply put the code within the class indentation block. When declaring a method that you intend to access through instances, you must use the self parameter in the definition

lass Dog( ):

    def makeSound(self):

        print("bark")

sam = Dog( )

sam.makeSound( )

# writing methods that accept parameters

class Dog( ):

    def showAge(self, age):

        print(age) # does not need self, age is referencing the parameter not an attribute

sam = Dog( )

sam.showAge( 6 ) # passing the integer 6 as an argument to the showAge method

## Setters and Getters:

* They are methods that you create to re-declare attribute values and return attribute values.

# using methods to set or return attribute values, proper programming practice

class Dog( ):

    name = ' ' #global attribute is empty -  would normally use init method to declare, this is for testing purposes

    def setName(self, new\_name):        #this is the setter

        self.name = new\_name # declares the new value for the name attribute

    def getName(self):  #this is the getter

        return self.name # returns the value of the name attribute

sam = Dog( )

sam.setName("Sammi")

print( sam.getName( ) ) # prints the returned value of self.name

## Inheritance:

* Inheritance is one of the concepts that allow classes to have code reusability within programming.
* Using inheritance, we can cut down on the repetitive lines that we write between similar classes. Inherited classes are known as superclasses, while those that perform the inheritance are known as subclasses. Also, the ability to override inherited methods is called method overriding and provides class customization for subclasses.

## Matt’s Workshop: Anaconda work

<https://github.com/StanislavPetrovV/Mandelbrot-set-Realtime-Viewer-/blob/main/requirements.txt>

In anaconda

1. Open anaconda prompt
2. conda create --name=mandelbrot python=3.9
3. conda activate Mandelbrot
4. pip install numpy taichi pygame (now we have anaconda environment and the requirements

open gitbash (cloning a repository)

git clone https://github.com/StanislavPetrovV/Mandelbrot-set-Realtime-Viewer-.git

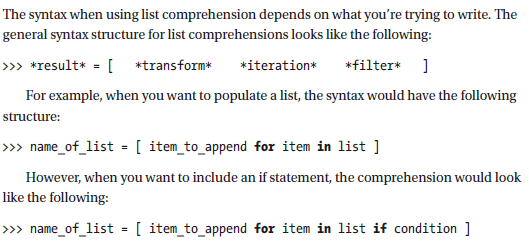
back to anaconda

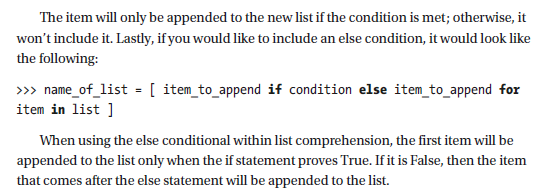
1. dir (will list – and should be able to see Mandelbrot-set-Realtime-Viewer-)
2. cd Mandelbrot-set-Realtime-Viewer- (go into directory)
3. dir (will show you the files within)
4. code main.py (opens it in VS code)
5. python main.py (run the program in python
6. Wowee – looks groovy
7. Close with x

# Week 8 – Efficiency:

## List Comprehension

* Allows us to create a list filled with data in a single line (cleans up code)





nums = [ x for x in range(10) if x % 2 == 0 ]   # generates a list of even numbers up to 10  (%==0 means remainder = 0)

print(nums)

# creating a dictionary of even numbers and square values using comprehension

numbers = [ x for x in range(10) ]

squares = { num : num\*\*2 for num in numbers if num % 2 == 0 }

print(squares)

## Lambda Functions:

* Lambda functions, otherwise known as anonymous functions, are one-line functions within Python.
* Lambdas will always begin with the keyword lambda. Following that you’ll find any arguments that are being passed in. On the right side of the colon, we’ll see the expression to be performed and returned.

lambda arguments : expression

# When using lambdas without storing them into a variable, you need to wrap parenthesis around the function, as well as any arguments being passed in.

(lambda x : x\*\*2 )( 4 ) # takes in the argument x (4) and returns the number squared

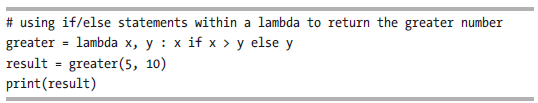
lambda arguments : value\_to\_return if condition else value\_to\_return

# using if/else statements within a lambda to return the greater number

greater = lambda x, y : x if x > y else y

result = greater(5, 10)

* Once a lambda function is used, it can’t be used again unless it is saved into a variable.
* Once you begin adding conditional statements into a lambda function, they act the same way that ternary operators do. The only difference is that you must provide both the if and else statements.



* Returning lambda functions

# returning a lambda function from another function

def my\_func(n):

    return lambda x : x \* n

doubler = my\_func(2) # returns equivalent of lambda x : x \* 2

print( doubler(5) ) # will output 10

## Map, filter and reduce

* The **map** function is used to iterate over a data collection and modify it.
  + The map function is used when you need to alter all items within an iterable data collection
  + Takes into two arguments, the function to be applied on each element and the iterable data
* The **filter** function is used to iterate over a data collection, and filter out data that doesn’t meet a condition.
* Lastly, the **reduce** function takes a data collection and condenses it down to a single result, like the sum function for lists.
  + accepts two arguments, the function to perform the execution and the data collection to iterate over
  + reduce iterates two items at a time instead of one. The result of reduce is to always return a single result

## Recursive Functions and Memoization

Recursion is a concept in programming where a function calls itself one or more times within its block. These types of functions can often run into issues with speed, however, due to the function constantly calling itself. Memoization helps this process by storing values that were already calculated to be used later.

* All recursive functions have what is known as a “base case,” or a stopping point. Like loops, you need a way to break out of a recursive call. Without one you create an infinite loop that will eventually crash.
* Memoization is an optimization technique used primarily to speed up computer programs by storing the results of previously called functions and returning the saved result when trying to calculate the same sequence

## @lru\_cache - Least Recently Used Cache

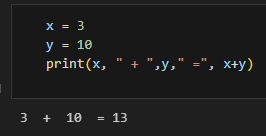
* Python’s built-in method for memoization

# Command appendix

## Print

print("Your text here")

print(variable 1, variable 2) # will print the defined variables with a space in between



print(“\t{}”.format(your text here) # \t = tab, same as \n = new line

## Type checking

Type(data type here) #will output class ‘data type here’

## .format()

Print(“hello my name is {}” .format(Abbey))

## .title()

Capaitalises first letter of everyword

## .replace(“replace this” , “with this”)

Find and replace tool

## .find(“the string we’re searching for”)

Output is the starting index of the term

## .split()

Returns a group of words in a sentence stored as a list

## Pwd

Print working directory: (shows us where we are at)

## ~

Stands for directory we are in

## ls

lists the files in that directory (ie. The folders in that folder)

## ls –rtlah

prints everything in the location including hidden files and who owns them and what types they are

## drwxrwx-x :

from left to right – d= directory, first rwx = user can read, write and execute, next rwx = the group can read, write and execute, last x = anyone using the compute can execute

## mkdir NAME\_HERE

make directory and name it

## touch name.filetype

ie. touch test.txt – makes a file in git bash

## cat name.filetype

displays whats in it (dumps it all)

## less test.txt

shows you it line by line as opposed to one big thing – q exits

## command >> name.filetype

outputs the command into the file as opposed to displaying it on the screen

## ^

means ctrl

## nano filename.type

takes you to edit the file – once you’ve finished editing it Crtl O (write out), enter, Ctrl X (exit)

## cd .

directory I am currently in

## cd ..

Takes you back one – can use multiple ../../ if you want to get way back

## cd –

Takes you back to where you just got out of if you used the ../

## Exit

Gets out of ssh session

## Committing info to Gitbash

Cd directory

Git add .

Git commit

Remove # from files to commit

Crt O, enter, Crt X

Git push

## mv oldfilename.filetype newfilename.filetype

Renames a files

## History

Shows you all commands you have used

## Input()

Generates a box for user to enter information

## print(type(variable))

Will print the variable type

## Try: and Except:

Try and except are used to catch errors

Format:

Try:

Command

Else:

Do this

## If, elif, else:

Format follows:

If \_\_ numerical\_equation/and/or/not/in \_\_\_:

Command (eg. Print(“Something)

Elif \_\_\_\_\_\_\_ (another condition):

Command (eg. Print(something else)

Else (If none of the above):

Command

## Lists:

Listname = [list item 1, list item 2]

Accessing items in list based on index (index always begins at 0)

* Print(Listname[index])

You can access a list within a list

* By using multiple square brackets
* Print(listname[index in list][index within indexed item])

Changing values in a list:

* Re-declare a variable with the index
* List[0]=new item

Copying a list into a new variable using colon:

Data\_copy = data[:]

## For Loops: for variable in function:

Temp variable – can be anything we want it to be

Function – can be a variety of things as well (range() is a built in function)

* for num in range(5):
* print( "Value: {}".format(num) )
* Value: 0
* Value: 1
* Value: 2
* Value: 3
* Value: 4

**Range function:**

for num in range(start, stop, step)

**Continue statement:**

Can use with an if statement and stops the current iteration and starts the next one.

**Break statement:**

Allows us to break out of a loop

**Pass Statement:**

Simply a placeholder so that the program doesn’t break 0 can add a #TODO comment to remind yourself to come back

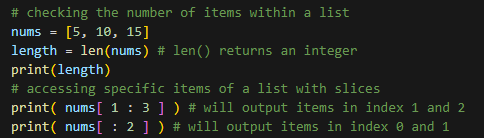
## Working with Lists – Functions to use

1. **Checking length**

len(list)

1. **Slicing lists**

Follows same argument as range function (start, stop step)



1. **Adding items:**
   1. **List\_name.append(item)**
   2. **List\_name.insert(index, item)** 
      1. Requires an index to insert
2. **Removing Items:**
   1. **List\_name.pop()**
      1. Automatically removes last item
   2. **List\_name.remove(item\_to\_remove)**
      1. Often need to use a try-except function in case the word isn’t in the list

## Working with Numerical Data:

1. **Min, max, sum**
   1. Format – min/max/sum(lis\_name)
2. **Sorting a list:**
   1. **Sorted(list\_name)**
      1. Work with numerical or alphabetical but not a mixed list – returns a copy, doesn’t alter original
   2. **List\_name.sort()**
      1. Changes list directly

## Conditionals and lists

1. **“In” and “not in”:**
   * 1. If “word” in list:

Command

## Functions

DEFINE:

def functionName(parameter1, parameter 2, \*args, \*\*kwargs):

function needed (ie. Print(“ “)

CALL IT: functionName(parameter)

## Return

def functionName(parameter1, parameter2):

return num1 + num2

NewVariable = functionName(paramter1, parameter2) # saves returned value into num

print(num)

print( addNums(parameter3, parameter4) ) # doesn't save returned value

## Dictionaries

FunctionName = {

"key1": "value",

“key2”:[1,2,3,4]

"key3": { "key3.a": value, "key3.b": 93 }

}

print( FunctionName["key"] ) # will output the dictionary within the wins key

print( FunctionName["key3"]["key3.a"] ) # first access the key3 key, then the next key

**Adding new info or changing existing to a dictionary:**

Variable\_Name[“newKey/oldKey”] = “NewValue”

**Deleting information:**

Use del function – need to use Try/Except otherwise will crash

FunctionName = { "key": value }

try:

del FunctionName["key"]

print(FunctionName)

except:

print("That key does not exist")

## Creating files

2| f = open("FileName.format", "mode") # open file in writing and reading mode

3| f.write("YourTextHere") # write() method to write our sentence to the file

4| f.close( ) # **Anytime you open a file, you must always close it.**

5| # reading from a text file

6| f = open("test.txt", "r")

## Creating classes

class class\_name( ):

def \_\_init\_\_(self, attribute1, attribute2):

self.attribute1 = attribute1 # sets the attribute to the value passed in

self.attribute2 = attribute 2

def action/method\_name(self):

function (ie. Print(“word”)

class\_object = class("attribute1value", attribute2value) # create a class object with the attributes passed in

## List comprehension

name\_of\_list = [ item\_to\_append if condition else item\_to\_append for item in list ]

eg. # creating a dictionary of even numbers and square values using comprehension

numbers = [ x for x in range(10) ]

squares = { num : num\*\*2 for num in numbers if num % 2 == 0 }

print(squares)

## List Comprehension

\*result\* = [ \*transform\* \*iteration\* \*filter\* ]

name\_of\_list = [ item\_to\_append for item in list ]

name\_of\_list = [ item\_to\_append if condition else item\_to\_append for item in list ]

## Lambda Functions:

lambda arguments : expression

lambda arguments : value\_to\_return if condition else value\_to\_return