**COMPLETE PYTHON BOOTCAMP**

1. **Install Python and setting up development environment:**

Using Anaconda and Jupyter Notebook as Python development environment. There are several ways to run Python code.

There are 3 main types of environments:

* Text Editors (Like Atom, Sublime Text, etc..)
* Full IDEs (PyCharm, Spyder, etc..)
* Notebook Environments (Jupyter Notebook)

**2. Python Object and Data structure basics:**

1. **Basic Data Types:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| Integers | int | Whole numbers: 3, 20 , 400.. |
| Floating Point | float | Numbers with a decimal point: 2.3, 4.6, 100.0 |
| Strings | str | Ordered sequence of characters: “hello”, “Deepak”.. |
| Lists | list | Ordered sequence of objects: [10, “hello”, 20.78] |
| Dictionaries | dict | Unordered Key: Value pairs: {“firstName”: “Deepak”, “lastName”: “Laxkar”} |
| Tuples | tup | Ordered immutable sequence of objects: (10, “hello”, 200.89) |
| Sets | set | Unordered collection of unique objects: {“a”, “b”} |
| Booleans | bool | Logical value: True or False |

* 1. **Check type of data:**

a = 10

type(a) //int

* 1. **Strings:**

greeting = “Hello”

Character: H e l l o

Index: 0 1 2 3 4

Reverse Index: 0 -4 -3 -2 -1

#length

len(greeting) //5

#Indexing

greeting[0] //H; First character

greeting[-1] //o; Last character

#Slicing: [start:end:step]

greeting[2:] //llo; Start from given index

greeting[:2] //He; Go upto the given index, not including the index

greeting[1:4] //ell; Sub section of string

greeting[::2] //Hlo; Step jump size of 2

greeting[::-1] //olleH; Reverse string

#String properties: Strings are Immutable

name = “Sam”

name[0] = “P” //Not allowed. Since strings are immutable

last\_letters = name[1:] //am

“P” + last\_letters //Pam; Concatenation of strings

letter = ‘z’

letter \* 10; //zzzzzzzzzz

name.upper() //SAM

name.lower() //sam

greeting = ‘Hello World’

greeting.split(‘’) //[‘Hello’, ‘World’]

#String Interpolation: Print Formatting with strings

name = “Sam”

print(“My name is ”+name) //Basic way

print(‘This is a string {}’.format(‘INSERTED’)); //This is a string INSERTED

print(‘The {} {} {}’.format(‘fox’, ‘brown’, ‘quick’)) //The fox brown quick

#Change default order

print(‘The {2} {1} {0}’.format(‘fox’, ‘brown’, ‘quick’)) //The quick brown fox

#Formatting with variables names

print(‘The {q} {b} {f}’.format(f=‘fox’, b=‘brown’, q=‘quick’)) //The quick brown fox

result = 100/777 //0.128700128700

#Float formatting follows “{value:width.precision f}”

print(“The result was {r:1.3f}”.format(r=result)) //The result was 0.129

#Using formatted strings (f string) literals introduced in python 3.6. Just another way of formatting the strings

name = “Deepak”

print(f’Hello, my name is {name}’)

* 1. **Lists:**

Lists are ordered sequences that can hold variety of object types. They support indexing and slicing and can be nested.

my\_list = [‘one’, 2, ‘three’]

len(my\_list) //3

#Indexing

my\_list[0] //’one’

#Concatenation

myList = [1,2,3]

anotherList = myList + my\_list //[1,2,3,‘one’,2,‘three’]

#List can be mutated, unlike strings

myList[0] = ‘one’ //[‘one’, 2, 3]

#Add item to the end of list

myList.append(4) //[‘one’, 2, 3, 4]

#Remove item from the end of the list

myList.pop() //4

myList //[‘one’, 2, 3]

#Remove at specific index

myList.pop(0) //’one’

myList //[2, 3]

#Sort list

new\_list = [‘a’, ‘e’, ‘b’, ‘x’, ‘d’]

new\_list.sort()

new\_list // [‘a’, ‘b’, ‘d’, ‘e’, ‘x’]

#Reverse list

myList.reverse()

myList //[3,2]

* 1. **Dictionaries:**

Dictionaries are unordered mappings for storing objects. They use key-value pairing. This allows quick access to the information without needing the index location. They cannot be indexed, sorted or sliced.

my\_dict = {‘firstName’: ‘Sam’, ‘lastName’: ‘Ford’}

#Retrieving value from a key in dictionary

my\_dict[‘firstName’] // ‘Sam’

#Add new key-value pair

my\_dict[‘pincode’] = 313001

#Dictionaries can also be mutated

my\_dict[‘firstName’] = ‘Pam’

my\_dict //{‘firstName’: ‘Pam’, ‘lastName’: ‘Ford’, ‘pincode’: 313001}

#Get all the keys

my\_dict.keys() //dict\_keys([‘firstName’, ‘lastName’, ‘pincode’])

#Get all the values

my\_dict.values() //dict\_values([‘Pam’, ‘Ford’, 313001])

#Get all pairs

my\_dict.items() //dict\_items([(‘firstName’, ‘Pam’), (‘lastName’, ‘Ford’), (‘pincode’, 313001)])

* 1. **Tuples:**

Tuples are very similar to lists. However, they have one key difference – immutability. Once an element is inside a tuple, it cannot be reassigned. Tuples use parenthesis: (1,2,3)

my\_tup = (1,2,3)

len(my\_tup) //3

#Indexing

new\_tup = (‘one’, 2, 3.4)

new\_tup[0] //’one’

#Count the appearance of an element

another\_tup = (‘a’, ‘a’, ‘b’)

another\_tup.count() //2

#Get index of first appearance of element

another\_tup.index(‘a’) // 0

* 1. **Sets:**

Sets are unordered collections of unique elements.

my\_set = set()

#Add element to the set

myset.add(1)

myset //{1}

myset.add(2)

myset //{1, 2}

#Cannot add same element again in a set

myset.add(2)

myset //{1, 2}

#Get unique values

set(‘Mississippi’) //{‘M’, ‘i’, ‘s’, ‘p’}

* 1. **Booleans:**

Sets are unordered collections of unique elements.

1 > 2 //False

**3. I/O with Basic Files in Python:**

**3.1 Writing a file**:

This function is specific to Jupyter notebooks. Alternatively, you can create a file with any text editor.

%%writefile test.txt

First Line

Second Line

**3.2 Opening a file**:

For now, using files in same location as the notebook.

myFile = open(‘test.txt’)

**3.3 Reading a file**:

Brings whole text file in memory. Use caution with large files, since everything will be held in memory.

myFile.read()

‘First Line\nSecond Line’ //Output

**3.4 Resetting the cursor**:

myFile.read()

‘’ //Output; This happens because you can imagine the reading "cursor" is at the end of the file after having read it. So, there is nothing left to read.

We can reset the "cursor" like this:

myFile.seek(0)

myFile.read()

‘First Line\nSecond Line’ //Output; Correct this time

#Read a file line by line using the readlines method. Readlines returns a list of the lines in the file

myFile.seek(0)

myFile.readlines()

[‘First Line', ‘Second Line’]

**3.5 Closing the file:**

myFile.close()

**3.6 Use ‘with’ to avoid manually closing the file:**

with open(‘test.txt’) as my\_new\_file:

contents = my\_new\_file.read()

contents

‘First Line\nSecond Line’

**3.7 File Modes:**

#Modes

r: read only

w: write only (will overwrite files or create new)

a: append only (will add on to file)

r+: read and write

w+: write and read (Overwrites existing file or creates new)

#Append mode

with open(‘test.txt’, mode = ‘a’) as f:

f.write(‘\nThird Line’)

First Line

Second Line

Third Line

#Write mode, overwrite

with open(‘test.txt’, mode = ‘w’) as f:

f.write(‘Fourth Line’)

Fourth Line

**3.8 Iterating through a file:**

%%writefile test.txt

First Line

Second Line

for line in open(‘test.txt’):

print(line)

First Line

Second Line

**4. Operators in Python:**

**4.1 Comparison Operators:**

#Equality

3 == 3

True

#Non equality

4 != 5

True

1 > 2

False

1 < 2

True

1 >= 2

False

1 <= 2

True

**4.2 Logical Operators: Combine comparisons**

#and

(1 < 2) and (2 > 3)

False

#or

(100 == 1) or (2 == 2)

False

#not

not(1 == 1)

False

**5. Statements in Python:**

**5.1 Control Flow Statements:**

To control flow of logic we use some keywords:

if

elif

else

os = ‘iOS’

if os == ‘iOS’:

print(“You are using Apple device”)

elif os == ‘Android’:

print(“You are using Android device”)

else:

print(“You are on some other OS”)

**5.2 For Loops:**

#Iterating lists

myList = [1,2,3,4,5,6]

for num in myList:

print(num)

1

2

3

4

5

6

#Iterating Tuples

myTup = (1,2,3)

for item in myTup:

print(item)

1

2

3

#Tuple unpacking

myList = [(1,2),(3,4),(5,6),(7,8)]

for item in myList:

print(item)

(1,2)

(3,4)

(5,6)

(7,8)

for a,b in myList:

print(a)

print(b)

1

2

3

4

5

6

7

8

#Iterating Dictionaries

myDict = {‘K1’: 1, ‘K2’: 2, ‘K3’: 3}

for item in myDict:

print(item)

K1

K2

K3

for key, value in d.items():

print(value)

1

2

3

**5.3 while Loop:**

x = 0

while x < 5:

print(x)

x = x + 1

0

1

2

3

4

while x < 5:

print(x)

x = x + 1

else:

print(“x is not less than 5”)

0

1

2

3

4

x is not less than 5

**5.4 break, continue, pass:**

break: Breaks out of the current closest enclosing loop

continue: Goes to the top of the closest enclosing loop

pass: Does nothing at all

**5.5 Useful Operators:**

#range operator: (start, end, step)

for num in range(10):

print(num)

0

1

2

3

4

5

6

7

8

9

for num in range(3,6):

print(num)

3

4

5

for num in range(0,10,2):

print(num)

0

2

4

6

8

#enumerate function

word = ‘abcde’

for letter in enumerate(word):

print(letter)

(0, ‘a’)

(1, ‘b’)

(2, ‘c’)

(3, ‘d’)

(4, ‘e’)

#zip function

myList1 = [1,2,3]

myList2 = [‘a’, ‘b’, ‘c’]

for item in zip(myList1, myList2):

print(item)

(1, ‘a’)

(2, ‘b’)

(3, ‘c’)

#in operator

‘x’ in [1,2,3]

False

#min function

myList = [10,20,30]

min(myList) //10

#max function

myList = [10,20,30]

max(myList) //30

#random library

from random import randint: (LowerLimit, UpperLimit)

randint(0, 100) // 78

#Taking user input

result = input(‘Enter a number here: ’)

type(result) //str; input always excepts inputs as strings

**5.6 List Comprehension:**

Unique way of quickly creating a list

#Old way

mystring = ‘hello’

myList = []

for letter in mystring:

myList.append(letter)

myList //[‘h’,’e’,’l’,’l’,’o’]

#New way

myList = [letter for letter in mystring]

myList //[‘h’,’e’,’l’,’l’,’o’]

myList = [num\*2 for num in range(0,3)]

myList //[0, 2, 4]

**6. Methods and Functions:**

**6.1 Introduction to Functions:**

#Function to add two numbers

def add\_num(num1, num2):

return num1+num2

add\_num(10, 20) //30; Function call

#Function that returns True if there is any even number in the list

def check\_even\_list(num\_list):

for number in num\_list:

if number % 2 == 0:

return True

else:

pass

return False

check\_even\_list([1,2,3]) // True

**6.2 Functions and Tuple Unpacking:**

stock\_prices = [(‘APPL’, 200), (‘GOOG’, 400), (‘MSFT’, 100)]

for ticker, price in stock\_prices:

print(ticker)

APPL  
GOOG  
MSFT

work\_hours = [(‘Abby’, 100), (‘Billy’, 400), (‘Cassie’, 800)]

def employee\_check(work\_hours):

current\_max = 0

employee\_of\_month = ‘’

for employee, hours in work\_hours:

if hours > current\_max:

current\_max = hours

employee\_of\_month = employee

else:

pass

#Returning tuple

return (employee\_of\_month, current\_max)

result = employee\_check(work\_hours)

result // (‘Cassie’, 800)

**6.3 Interaction between functions:**

from random import shuffle

#Shuffled the input list

def shuffle\_list(myList):

shuffle(myList)

return myList

#Get player guess

def player\_guess():

guess = ‘’

while guess not in [‘0’, ‘1’, ‘2’]:

guess = input(“Pick a number: 0, 1 or 2”)

return int(guess)

#Check if player guess is correct or incorrect

def check\_guess(myList, guess):

if myList[guess] == ‘O’

print(“Correct Guess!”)

else:

print(“Wrong Guess!”)

print(myList)

myList = [‘ ’, ‘O’, ‘ ‘]

shuffledList = shuffle\_list(myList)

guess = player\_guess()

check\_guess(shuffledList, guess)

**6.4 \*args and \*\*kwargs:**

#Function accepting 2 arguments

def myFunc(a, b):

return sum((a,b)) \* 0.05

#Function accepting many arguments using args, it creates tuples of arguments

def myFunc(\*args):

print(args)

myFunc(10,20) // (10, 20)

#Function accepting many arguments using \*\*kwargs, it creates dictionary of arguments

def myFunc(\*\*kwargs):

print(kwargs)

myFunc(fruit=’apple’, veggie=’lettuce’) //{‘fruit’: ‘apple’, ‘veggie’: ‘lettuce’}

**6.5 Lambda Expressions: Map and Filter**

#Map function

def square(num):

return num\*\*2

my\_nums = [1,2,3,4,5]

for item in map(square, my\_nums):

print(item)

1

4

9

16

25

list(map(square, my\_nums)) //[1,4,9,16,25]

#Filter function

def check\_even(num):

return num % 2 == 0

my\_nums = [1,2,3,4,5]

list(filter(check\_even, my\_nums)) // [2,4]

#Lambda Functions: Are generally used to create anonymous functions. Best practice is to use them only if it does not create complex to read functions. Otherwise use normal functions

lambda num: num \*\* 2 //square function from above after converting into lambda

list(map(lambda num: num \*\* 2, my\_nums)) //[1,4,9,16,25]

**6.6 Nested statements and scope:**

**LEGB Rule:**

**L: Local** - Names assigned in any way within a function (def or lambda), and not declared global in that function.

**E: Enclosing function locals** - Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

**G: Global (module)** – Names assigned at the top-level of a module file, or declared global in a def within the file.

**B: Built-in (Python)** - Names preassigned in the built-in names module: open, range, SyntaxError, ...

#Example1

#Global

name = ‘THIS IS A GLOBAL STRING’

def greet():

def hello():

print(‘Hello’ +name)

hello()

greet() //Hello THIS IS A GLOBAL STRING

#Example2

#Global

name = ‘THIS IS A GLOBAL STRING’

def greet():

#Enclosing

name = ‘Sammy’

def hello():

print(‘Hello’ +name)

hello()

greet() //Hello Sammy

#Example3

#Global

name = ‘THIS IS A GLOBAL STRING’

def greet():

#Enclosing

name = ‘Sammy’

def hello():

#Local

name = ‘DEREK’

print(‘Hello’ +name)

hello()

greet() //Hello DEREK

#Example4

x = 50

def func(x):

x = 25

print(f’x is {x}’)

func(x) // x is 25

print(x) //50

#Using global keyword to access global variables

x = 50

def func(x):

global x

x = 25

print(f’x is {x}’)

func(x) // x is 25

print(x) //25

#Much better approach than using global keyword

x = 50

def func(x):

x = 25

print(f’x is {x}’)

return x

x = func(x)

print(x) //25

**7. Milestone Project 1**

def displayGameBoard():

for i in range(len(board)):

if i == 0 or i == 3 or i == 6:

print(' | ' + board[i] + ' | ' + board[i+1] + ' | ' + board[i+2] + ' | ')

else:

continue

print('\n')

def getPlayerNumber():

choice = 'wrong'

acceptable\_range = range(1,3)

within\_range = False

print('List of Symbols: ')

for i in range(len(user\_symbols)):

print("{}: {}".format(i+1, user\_symbols[i]))

while choice.isdigit() == False or within\_range == False:

choice = input("Select your choice (1-2): ")

if choice.isdigit() == False:

print("Sorry that is not a digit!")

elif int(choice) in acceptable\_range:

within\_range = True

else:

print("Sorry, you are out of acceptable range (1-2). Please try a valid choice.")

within\_range = False

return int(choice) – 1

def getUserPosition():

choice = 'wrong'

acceptable\_range = range(1,10)

within\_range = False

while choice.isdigit() == False or within\_range == False:

choice = input("Player {}! Select your position (1-10): ".format(current\_player + 1))

if choice.isdigit() == False:

print("Sorry that is not a digit!")

if choice.isdigit() == True:

if int(choice) in acceptable\_range:

if board[int(choice)- 1] == '-':

within\_range = True

else:

print("Sorry, the position is already occupied. Please try a valid choice.")

within\_range = False

else:

print("Sorry, you are out of acceptable range (1-10). Please try a valid choice.")

within\_range = False

return int(choice) – 1

def markPosition(user\_position):

board[user\_position] = user\_symbols[current\_player]

def isBoardPlayable():

if '-' in board:

return True

else:

return False

def gameOnChoice():

choice = 'wrong'

while choice not in ['Y', 'N']:

choice = input("Keep playing? (Y or N): ")

if choice not in ['Y', 'N']:

print("Invalid choice. Please provide answer Y or N")

clear\_output()

if choice == 'Y':

return True

else:

return False

def computeResult():

user\_symbol = user\_symbols[current\_player]

return checkRows(user\_symbol) or checkColumns(user\_symbol) or checkDiagonals(user\_symbol)

def checkRows(user\_symbol):

for i in range(len(board)):

if i == 0 or i == 3 or i == 6:

if board[i] == board[i+1] == board[i+2] == user\_symbol:

return True

return False

def checkColumns(user\_symbol):

for i in range(len(board)):

if i == 0 or i == 1 or i == 2:

if board[i] == board[i+3] == board[i+6] == user\_symbol:

return True

return False

def checkDiagonals(user\_symbol):

for i in range(len(board)):

if i == 0:

if board[i] == board[i+4] == board[i+8] == user\_symbol:

return True

if i == 2:

if board[i] == board[i+2] == board[i+4] == user\_symbol:

return True

return False

from IPython.display import clear\_output

#Initialize variables

board = ['-', '-', '-', '-', '-', '-', '-', '-', '-']

gameOn = True

user\_symbols = ['X', 'O']

#Valid players are 0 and 1, assigning 2 to initialize the game

current\_player = 2

while gameOn:

displayGameBoard()

if current\_player == 2:

current\_player = getPlayerNumber()

user\_position = getUserPosition()

markPosition(user\_position)

displayGameBoard()

gameResult = computeResult()

if gameResult:

print("Congratulations, player {} win!".format(current\_player + 1))

#Re-initialize variables

board = ['-', '-', '-', '-', '-', '-', '-', '-', '-']

gameOn = True

user\_symbols = ['X', 'O']

#Valid players are 0 and 1, assigning 2 to initialize the game

current\_player = 2

elif not isBoardPlayable():

print("It's a Tie!")

#Re-initialize variables

board = ['-', '-', '-', '-', '-', '-', '-', '-', '-']

gameOn = True

user\_symbols = ['X', 'O']

#Valid players are 0 and 1, assigning 2 to initialize the game

current\_player = 2

else:

current\_player = (current\_player+1) % 2

gameOn = gameOnChoice()

**8. Object Oriented Programming**

**8.1 class keyword and attributes**

class Dog():

def \_\_init\_\_(self, breed, name):

#Attributes

self.breed = breed

self.name = name

my\_dog = Dog(breed=’Lab’, name=’Sammy’)

type(my\_dog) //\_\_main\_\_.Dog

my\_dog.breed //Lab

my\_dog.name //Sammy

**8.2 Class Object attributes and methods**

class Dog():

#Class Object Attribute

#Same for any instance of the class

species = ‘mammal’

def \_\_init\_\_(self, breed, name):

#Attributes

self.breed = breed

self.name = name

#Methods

def bark(self, number):

x = 0

while x < number:

print(“WOOF! My name is {}”.format(self.name))

my\_dog = Dog(breed=’Lab’, name=’Sammy’)

type(my\_dog) //\_\_main\_\_.Dog

my\_dog.breed //Lab

my\_dog.name //Sammy

my\_dog.species //mammal

my\_dog.bark(1) //WOOF! My name is Sammy

class Circle():

#Class Object Attribute

#Same for any instance of the class

pi = 3.14

#Using default arguments

def \_\_init\_\_(self, radius=1):

#Attributes

self.radius = radius

#creating new attribute from other attributes. Notice using class object attribute. Could be used as self.pi or Circle.pi

self.area = self.pi \* self.radius \* self.radius

#Methods

def get\_circumference(self):

return 2 \* self.pi \* self.radius

my\_circle = Circle()

type(my\_circle) //\_\_main\_\_.Circle

my\_circle.get\_circumference //6.28

my\_circle.area //3.14

**8.3 Inheritance**

#Base class

class Animal():

def \_\_init\_\_(self):

print(“Animal Created”)

def who\_am\_i(self):

print(“I’m an Animal”)

def eat(self):

print(“I’m eating”)

#Derived class

class Dog(Animal):

def \_\_init\_\_(self):

Animal.\_\_init\_\_(self)

print(“Dog Created”)

#Method overriding

def who\_am\_i(self):

print(“I’m a Dog”)

#Adding new methods

def bark(self):

print(“WOOF!”)

myDog = Dog() //Animal Created

//Dog Created

myDog.who\_am\_i() //I’m a Dog

myDog.bark() //WOOF!

**8.3 Polymorphism and Abstraction**

class Dog():

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

return self.name + “ says woof!”

class Cat():

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

return self.name + “ says meow!”

niko = Dog(“niko”)

felix = Cat(“felix”)

print(niko.speak()) //Niko says woof!

print(felix.speak()) //Felix says meow!

#Abstract Base class. It is not designed to be instantiated.

class Animal():

def \_\_init\_\_(self):

self.name = name

#Abstract method

def speak(self):

#Raise error

raise NotImplementedError(“Subclass must implement this abstract method”)

class Dog(Animal):

def speak(self):

return self.name + “ says woof!”

class Cat(Animal):

def speak(self):

return self.name + “ says meow!”

fido = Dog(“Fido”)

limo = Cat(“Limo”)

print(fido.speak()) //Fido says woof!

print(limo.speak()) //Limo says meow!

**8.4 Special methods (Magic/Dunder)**

myList = [1,2,3]

len(myList) //3

print(myList) // [1,2,3]

class Sample():

#**pass** denotes compiler that it is empty. So, it won’t generate errors

pass

len(Sample) //TypeError: object of type ‘Sample’ has no len()

print(Sample) //prints some garbage

#We use some special methods to use built-in python defined functions with own user defined objects ?

class Book():

#Special method, this gets called automatically when object in instantiated

def \_\_init\_\_(self, title, author, pages):

self.title = title

self.author = author

self.pages = pages

#Special method, this gets called when string representation of class is required

def \_\_str\_\_(self):

return f”{self.title} by {self.author}”

#Special method

def \_\_len\_\_(self):

return self.pages

#Special method

def \_\_del\_\_(self):

print(“Book object deleted!”)

myBook = Book(‘Python programming’, ‘Deepak’, 100)

#This asks the book object, that if you have the string representation of the object, then give it. This is where \_\_str\_\_ special method will get called

print(myBook) //Python Programming by Deepak

len(myBook) //100

#del will delete the object or variables from memory. But sometimes we may want to perform some operations before deleting. That’s where \_\_del\_\_ special method will get called

del myBook //Book object deleted!

**9. Modules and Packages**

**9.1 PyPI and pip install**

**PyPI** is a repository for open-source third-party Python packages. It is similar to **npm** for Node.js.

We can use **pip install** at command line to install these third party packages.

Text, letter

Description automatically generated

**9.2 Writing your Modules and Packages**

**9.2.1 Creating Modules and using them**

Modules are just .py scripts that you can call in another .py script.

Packages are a collection of modules.

#mymodule.py

def my\_func():

print("Hey I'm in mymodule.py")

#myprogram.py

from mymodule import my\_func

my\_func()

Graphical user interface, text

Description automatically generated

**9.2.2 Creating Packages and using them**

Steps to create a package:

1. Create a folder with the package name of your choice.
2. Inside that folder create an empty \_\_init\_\_.py file. This file differentiates that it is a package and not just a folder.
3. You can have other scripts in this folder.
4. Repeat steps 1-3 if you have sub folders (or sub packages) inside the main package folder.

**9.2.3 \_\_name\_\_ and \_\_main\_\_**

Sometimes when you are importing from a module, you would like to know whether a modules function is being used as an import, or if you are using the original .py file of that module. In this case we can use the:

if \_\_name\_\_ == "\_\_main\_\_":

line to determine this.

When your script is run by passing it as a command to the Python interpreter: python myscript.py

All of the code that is at indentation level 0 gets executed. Functions and classes that are defined are, well, defined, but none of their code gets ran. Unlike other languages, there's no main() function that gets run automatically - the main() function is implicitly all the code at the top level.

In this case, the top-level code is an if block. \_\_name\_\_ is a built-in variable which evaluate to the name of the current module. However, if a module is being run directly (as in myscript.py above), then \_\_name\_\_ instead is set to the string "\_\_main\_\_". Thus, you can test whether your script is being run directly or being imported by something else by testing.

if \_\_name\_\_ == "\_\_main\_\_":

If that code is being imported into another module, the various function and

class definitions will be imported, but the main() code won't get run.

#one.py

def func():

print("Hey I'm a function in one.py")

print("Top level in one.py")

if \_\_name\_\_ == "\_\_main\_\_":

print("one.py is being run directly")

else:

print("one.py has been imported")

Output:



#two.py

import one

print("Top level in two.py")

one.func()

if \_\_name\_\_ == "\_\_main\_\_":

print("two.py is being run directly")

else:

print("two.py has been imported")

Output:

A picture containing chart

Description automatically generated

**10. Errors and Exceptions Handling**

Errors are bound to happen in your code. We can use error handling to attempt to plan for possible errors.

We use three keywords:

**try:** This is the block of code to be attempted (may lead to an error)

**except:** Block of code will execute when there is any error in the try block.

**finally:** A final block of code to be executed, regardless of an error.

try:

result 10 + ‘10’

except:

#Generic catch, will catch all the errors

print(“There is some problem with adding the numbers”)

**try...except...finally**

try:

f = open(‘testfile’, ‘w’)

f.write(“Writing some text”)

except TypeError:

#Specifically catching TypeError

print(“There was a Type Error”)

except OSError:

print(“You may have an OS Error”)

finally:

print(“I always run”)

**try...except...else**

def ask\_for\_int():

while True:

try:

result = int(input(“Enter a number: ”))

except:

print(“Whoops!, that’s not a number”)

continue

#Block of code which executes if there was no error

else:

print(“Thank you for your input”)

break

**11. Python Decorators**

Python has decorators that allows you to add on extra functionality to an already existing function. They use the @ operator and are then placed on top of the original function.

To add a new feature or functionality to a function, we can:

1. Update the function with some new code. The problem with this is, that if you want to remove the new functionality, you need to make modifications to the function.
2. Create a new function and copy the existing function and add new feature. Benefit of this is, that you can have a switch to toggle on/off the functionality whether you need it or not.

#Some Basic concepts

#Defining a function within another function and returning function from a function

def hello(name = ‘Jose’):

print(‘The hello() function’)

def greet():

return ‘The greet() function’

def welcome():

return ‘The welcome() function’

if name == ‘Jose’:

return greet

else:

return welcome

my\_new\_func = hello(‘Jose’) //The hello() function

print(my\_new\_func()) //The greet() function

#Passing function within another function

def hello():

return ‘Hi Jose!’

def other(some\_other\_func):

print(‘Some other code’)

print(some\_other\_func())

other(hello) //Some other code

//Hi Jose

#Decorating a function: Adding some feature to the original function, by creating copy of original function and decorating it with new feature.

def new\_decorator(original\_func):

def wrap\_func():

print(‘Some extra code before the original function’)

original\_func()

print(‘Some extra code after the original function’)

return wrap\_func

def func\_needs\_decorator():

print(‘I want to be decorated’)

decorated\_func = new\_decorator(func\_needs\_decorator)

decorated\_func() //Some extra code before the original function

//I want to be decorated

//Some extra code after the original function

#Using @ operator

@new\_decorator

def func\_needs\_decorator():

print(‘I want to be decorated’)

func\_need\_decorator() //Some extra code before the original function

//I want to be decorated

//Some extra code after the original function

#You can comment the @ line to remove to new functionality and the original function will behave like the old one.

**12. Python Generators**

Generator function allows us to write a function that can send back a value and then later resume to pick up where it left off. Generators allows us to generate a sequence of values over time. The difference in syntax will be of **yield** keyword

When a generator function is compiled they become an object that supports an iteration protocol. That means when they are called in your code they don’t actually return a value and then exit. Generator function will automatically suspend and resume their execution and state around the last point of value generation. The advantage is that instead of having to compute an entire series of values up front, the generator computes one value waits until the next value is called for.

#Problem is, here we are keeping entire list of numbers in memory all the time. Imagine if there are 1Million numbers then, it is not an efficient way.

def create\_cubes(n):

result = []

for x in range(n):

result.append(x\*\*3)

return result

create\_cubes(5)//[0,1,8,27,64]

#Better way using Generator

def create\_cubes(n):

for x in range(n):

yield(x\*\*3)

#Now we need to just iterate it

for x in create\_cubes(5)

//0

//1

//8

//27

//64

#You can create a list, if you really need it

list(create\_cubes(5)) //[0,1,8,27,64]

def simple\_gen():

for x in range(3):

yield x

for number in simple\_gen():

print(number)

//0

//1

//2

g = simple\_gen()

print(next(g)) //0

print(next(g)) //1

print(next(g)) //2

s = ‘hello’

for letter in s:

print(letter)

//h

//e

//l

//l

//o

print(next(s)) //This won’t work since string is not a generator. We can use iter to convert it to generator and use next on it.

s\_iter = iter(s)

next(s\_iter)// ‘h’

next(s\_iter)// ‘e’