**Python**

**Object-oriented programming language**

**(OOP)**

Both OOP & FP are high level languages

When object interacts with another objects through medium(object) by calling methods or sending messages

(HTTP,JDBC,HTTPS,ODBC)

Example: going from one place to another using a car (object)

Functional programming language: depends on using the functions returned as values according to each program.

OOP:

* accessing an object to create a new object
* You are not in control

FP:

* You are in control when it comes to using applications on your mobile phones.

Scripting language:

* Linux & Unix
* SQL

Low level language:

* Machine language
* ALU: Arithmetic logic units;using binary data to perform AND,OR logic operations

**My Research**

Difference between OOP & FP:

OOP: it is how the data is organised in a specific design called data structure for easy & quick access or update. It doesn’t focus much on the algorithm itself and how it is computed.

it consists of

* Class . EX: Human
* Objects EX: name
* Attributes for this object EX: email, address
* Methods. EX: eat,walk,run,work; verbs

FP: Alternate program to OOP using functions on large amounts of data to get an output without modifying the original data.

**Interpreting & Compiling**

Interpreting: software chipped in the computer as high level language is translated into low level language/machine code line by line (the process here takes more time than compiler ). examples: Python , SQL, Ruby,Java Script.

Compiling: Translating the whole program from high level language to low level language.Examples:Java, C++

Python : is an interpreter language executes line by line and translate each line to the machine language

**Python, Visual studio code,Jupyter Notebook and Code Runner Installation**

**Python command lines using Jupyter Notebook:**

| **print('hello world') hello world print('welcome') welcome a = 10 a= 10 =b a=10 print('a =',a)  b=print('a=',a,'=b') *# a=10=b* a=10 print('a=',a,sep='dddd',end='\n\n\n') *# here it gives 3 new lines* print('a=',a,sep='0',end='$$$') a=10 a=dddd10   a=010$$$  *#lists* *#lists* L1=['John',102,'USA'] L2=[1,2,3,4,5,6] print(L1) print(L2) print(L1+L2) ['John', 102, 'USA'] [1, 2, 3, 4, 5, 6] ['John', 102, 'USA', 1, 2, 3, 4, 5, 6]  *# tuples: it is immutable;means once you declared it,you cannot alter it* tup=('Apple','Mango','Orange','banana') tup[2]='Strawberry' print(tup) ----> 3 tup[2]='Strawberry'  4 print(tup)  TypeError: 'tuple' object does not support item assignment *#once you have more than one element ( , ) it is called tuple otherwise it is string* tup2=tuple('orange') print(type(tup2)) print(tup2)  <class 'tuple'> ('o', 'r', 'a', 'n', 'g', 'e')  #dictionaries: contains keys and values employee={'Name':'Jinesh','salary':5454,'company':'gr'} print(type(employee)) Month={'jan','feb','march'} print(type(Month)) <class 'dict'> <class 'set'>** |
| --- |

**Print (type(x))** # to check the type of data

**tup=tuple(list\_name)** # to turn a list into tuple

Python Application:

Tool used to develop web applications: Django,Flask,Pyramid,Web2py,CherryPy

For Desktop UI Application:

For console

For scientific

For Audio/Video

For 3D

For Image processing

For ERP

**My Research/work**

**Python Command line using Jupyter Notebook**

* Tuples,lists & dictionaries
* Finding the type of the data

| **tu=('orange',) tu2=('strawberry',) print(tu+tu2) ('orange', 'strawberry') tup=('Apple','Mango','Orange','banana') list=[] for word in tup:  list.append(word) print(list) *# converting tuple into list using for loop* list=[] for word in tup:  list.append(word) print(list) ['Apple', 'Mango', 'Orange', 'banana']  *#dictionaries: contains key & value* employee={'Name':'Jinesh','salary':5454,'company':'gr'} print(type(employee)) Month={'jan','feb','march'} print(type(Month)) print(employee.keys()) print(employee.values()) print(employee.items()) print(len(employee)) *# gives the number of items in the tuple* <class 'dict'> <class 'set'> dict\_keys(['Name', 'salary', 'company']) dict\_values(['Jinesh', 5454, 'gr']) dict\_items([('Name', 'Jinesh'), ('salary', 5454), ('company', 'gr')]) 3** |
| --- |

**One of the Immutable data types:**

**frozensets()**

To frozen a list,dictionary or a dataset

| **xample\_list = [1,2,3,4,5] frozen\_set = frozenset(example\_list)  print(example\_list) print(type(example\_list))  print(frozen\_set) *# returns a tuple of dictionary* print(type(frozen\_set)) [1, 2, 3, 4, 5] <class 'list'> frozenset({1, 2, 3, 4, 5}) <class 'frozenset'>** |
| --- |

|  |
| --- |

**For Loop**

it is like a cursor that goes into each value in a list,tuple or a dictionary

| listing=[1,2,3,4,5,6] for value in listing:  print(value) print('end') 1 2 3 4 5 6 end |
| --- |

**My Research/work**

For loop:

**Example**: the code reads integer n, and to get all non-negative integers i<n to print i^2

| n = int(input())   for i in range(0,n):  print(pow(i,2)) For n=5 0 1 4 9 16 |
| --- |

The address where the variable is located when a value is stored in it:

| print(id(a)) print(id(b)) *# both is stored in the same location/address as they have the same value* a=500 *# immutable* print(id(a)) 140704327727432 140704327727432 2579860937136 |
| --- |

Immutable: int,float,tuple,complex,string,Bytes,string frozen.frozensets().

Mutable: lists

NOTE THAT: until 2pow(8) which is 256 , the variables are stored in the same address. Above that it won’t be stored in the same address

| a=256 b=256 print(id(a)) print(id(b)) 140704327734024 1407043277340244  a=257 b=257 print(id(a)) print(id(b)) 2579860935952 2579860935088 |
| --- |

**My Research/work**

Intern Objects and whether two objects have the same location or not:

| **x = "Lorem ipsum" y = "Lorem ipsum" x is y print(id(x)) print(id(y)) *#locations will be different as they are two distinct objects although they have the same value*  *#unless they are intern objects;with small integers*  x = 'Lorem' y= 'Lorem' x is y print(id(x)) print(id(y)) 2579855165616 2579856125488  2579861077744 2579861077744** |
| --- |

When you assign the second variable to the first one,it is like aliasing to the first variable so both will have the same location

| x = 'Lorem' y= x x is y print(id(x)) print(id(y)) 2579861077744 2579861077744 |
| --- |

**Using range**

| for i in reversed(range(5)):  print(i) *# here it works*  for i in range(7):  print(i) 4 3 2 1 0 0 1 2 3 4 5 6 *#putting range values in list instead of manually write the values* print(list(range(0,15))) print(list(range(5,25,4))) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] [5, 9, 13, 17, 21] |
| --- |

**My Research/work**

| **n = int(input()) if 1<=n<=150:  print(\*range(1,n+1),sep='') *#n=5* 12345** |
| --- |

**As range function is not supported in python 3 so we use \*range()**

**Using Remainder to get the odd or even numbers**

| tup=(1,2,3,4,5,6) for number in tup:  if number%2!=0:  print(number) 1 3 5 |
| --- |

**While loop**

| *# while loop* *# here means that until counter =10 print (python loops)* counter=0 while counter<10:  counter=counter+3  print('python loops') python loops python loops python loops python loops |
| --- |

**Pass & Continue uses in loops**

| for string in 'python loops':  pass  print(string) S for string in 'python loops':  pass *# here it means to go back to continue the process of the loop; to go to for loop again* print(string) s |
| --- |

**My Research/work**

**Difference between Pass & Continue in loops**

**For loop**

| for string in 'python loops':  pass *# here it says to do nothing and to go to the next line*  print(string) p y t h o n   l o o p S for string in 'python loops':  continue *# here it means to go back to continue the process of the loop; to go to for loop again*  print(string) *# gives nothing as it keeps go to the for loop only*  record={'itika':90,'df':60,'dsde':70} for (name,mark)in record.items():  if mark>80:  print(name,mark)  else:  continue itika 90 |
| --- |

**Using variables in a string by using (f-strings)**

| *# nested loops* student\_fn='itika' student\_1='angola' record={'itika':90,'df':60,'dsde':70} print(f"Marks of the student {student\_fn} are",record[student\_fn]) *# f here is just to get the value of one variable in a string like getting the student\_fn's value* Marks of the student itika are 90 |
| --- |

**Using definitions to make it more flexible or generalised**

| *# using definition* def getMyMarks(student):  return record[student] print(f'marks of student {student\_fn} are',getMyMarks(student\_fn)) marks of student itika are 90 |
| --- |

**My Research/work**

To get the marks which is more than 80 for each student

def getmark(student):

return student,record[student]

for (name,mark)in record.items():

if mark>80:

print(getmark(name))

else:

continue

('itika', 90)

**Using random library(randint() , randrange() )**

import random # it gives random integers/range/bytes

new=list()

for i in range(0,11):

new.append(random.randint(0,11))

print(new)

[11, 4, 4, 9, 3, 2, 10, 1, 7, 8, 3]

for i in range(0,11):

new.append(random.randrange(0,11,2)) # randrange(start,stop,step)

print(new)

[6, 4, 4, 0, 0, 6, 4, 10, 10, 8, 2]

**Using pop() function**

#using pop function as it gives the values in reverse like if there is a bucket and you pick up the last item you put in there

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

new=[]

while list:

new.append(list.pop())

print(new)

print(list) # here the list will become empty as all the values have been poped out of it to ‘new’

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

[]

# to duplicate the values in a list

duplic=new\*2

print(duplic)

len(new) # the no. of values in a list

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

6

**To check if an element/value present in a list,max value and min value**

print(5 in new)

print(max(new))

print(min(new))

True

6

1

**Using Debugging**

* Copy and paste the code in separate python file editor not jupyter notebook
* Start run with debugging then press F11 or step into button which will explain each step executed in the code

**Finding the most optimal algorithm to find the sum of values in a list**

#finding the right/most optimal algorithm to get the summing of some values in a list

def summing(a,b):

if a==0:

abst=listing1[b]

else:

abst=listing1[b]-listing1[a-1]

return abst

listing=[15,21,36,41,54,62]

listing1=[]

i=range(0,len(listing))

sum=0

for value in listing:

sum=sum+value

listing1.append(sum)

print(listing1)

print(summing(3,5))

print(listing[3]+listing[4]+listing[5])

print(summing(0,3))

print(listing[0]+listing[1]+listing[2]+listing[3])

**Indexes by minus**

tuple=('python','tuple','ordered','immutable','collection','object')

print(tuple[ :-4])

print(tuple[ :-5])

print(tuple[ :-3])

('python', 'tuple') # here it counts from backwards starting from -1 and gives the in between

('python',)

('python', 'tuple', 'ordered')

**Getting the remainder for a number**

**option 1:**

| *#getting the remainder of a number* def factor(N):  count=0  for i in range(1,N+1):  if N%i==0:  count=count+1  print('the factor is',i)  return count print(factor(24)) print(factor(16)) the factor is 1 the factor is 2 the factor is 3 the factor is 4 the factor is 6 the factor is 8 the factor is 12 the factor is 24 8 the factor is 1 the factor is 2 the factor is 4 the factor is 8 the factor is 16 5 |
| --- |

**Option 2: the optimised formula**

| def calculatecountfactorial(N):  count=0  for i in range(1,int(math.sqrt(N)+1)):  if N%i==0:  if N==i\*i:  count=count+1  else:  count=count+2  return count N=int(input('enter a number')) *#N=25* print(calculatecountfactorial(N)) 3 |
| --- |

**Sum of the natural numbers**

| def sumnaturalnumbers(N):  s= N\*(N+1)/2  return s print(sumnaturalnumbers(100)) 5050.0 |
| --- |

**Logarithms**

Logarithms allow us to perform calculations using smaller values than the value we get in getting the square root of the same value or the division by 2

**Time Complexity**

How many iterations for the loops are executed to give the output?

N represents the number of iterations depending on the size of the data.

It is a way to measure the performance of the algorithm according to the size of the input and its running time.

Ex1:

for i in range(N):

print(i)

Here the time complexity is O(N)

Ex2:

For i 1 to N:

For j 1 to M:

Here the time complexity for the first loop is O(N)

And the time complexity for the second loop is O(M)

Then the time complexity for the whole code is O(N+M)

Ex3:

For i 1 to N:

For j 1 to N:

Here the time complexity for the first loop is O(N)

And the time complexity for the second loop is O(N)

Then the time complexity for the whole code is O(N+N)=O(2N)

**My Research**

**Time Complexity**

Types of time complexity:

* Constant time O(1): it doesn’t depend on the input data.

if a > b:

return True

else:

return False

* Linear time O(N): it scans the values of the data linearly;value by value.

for value in data:

print(value)

* Logarithmic time O(logN): it reduces the size of the data input or multiplying it by 2.or we don’t need to look at all the values of the data.

for index in range(0, len(data), 3):

print(data[index])

* Quasilinear time O(NlogN): it does both linear and logarithmic time;it scans each value of the data input and reduces the size of this value or multiply it by 2.or do logarithmic operations to it.

for value in data1:

result.append(binary\_search(data2, value))

* Quadratic time O(N^2): to perform a linear operation for each value of the data input

For ex. for value 1 ,we scan all the values in the same data input ;maybe to compare this value 1 with the rest of the values in the same data.

for x in data:

for y in data:

print(x, y)

Or

For x in data:

For y in x:

* Exponential time O(2^N):when the growth doubles with each addition to the input data set

def fibonacci(n):

if n <= 1:

return n

return fibonacci(n-1) + fibonacci(n-2)

**Data structure types**

**(with OOP used for creating objects and methods to store and manipulate the data in data structure)**

Data structure: it is how the data is organised in the memory to be easily accessed as well as manipulating the data;sorting,replacing,deleting,inserting.Using OOP

Types:

Linear data structure:Array, Queue,Linkedlist and Stack

Non linear: Tree and Graph.

1- Array: is a list of data [ , , , , , ] of the same type stored in a block of memory.

Ex1:

| *# classes* *# %%* class Array(object):    def \_\_init\_\_(self,sizeOfArray,arrayType = int):  self.sizeOfArray = len(list(map(arrayType ,range(sizeOfArray))))  self.arrayItems = [arrayType(0)]\*sizeOfArray   def \_\_len\_\_(self):  return len(self.arrayItems)    def insert(self,keyToInsert,position):  if self.sizeOfArray > position:  for i in range(self.sizeOfArray-2,position-1,-1):  *# the method proceeds to shift the existing elements in the array to make room for the new element.*   self.arrayItems[i+1]=self.arrayItems[i]  self.arrayItems[position]=keyToInsert  else:  print("Size of array is ",self.sizeOfArray)   a = Array(10,int) a.insert(2,2) a.\_\_len\_\_() a.arrayItems a.insert(3,4) a.arrayItems a.insert(5,1) a.arrayItems for i in range(9,0,-1):  a.insert(i,i)  print(a.arrayItems) [0, 5, 0, 2, 0, 3, 0, 0, 0, 9] [0, 5, 0, 2, 0, 3, 0, 0, 8, 0] [0, 5, 0, 2, 0, 3, 0, 7, 0, 8] [0, 5, 0, 2, 0, 3, 6, 0, 7, 0] [0, 5, 0, 2, 0, 5, 3, 6, 0, 7] [0, 5, 0, 2, 4, 0, 5, 3, 6, 0] [0, 5, 0, 3, 2, 4, 0, 5, 3, 6] [0, 5, 2, 0, 3, 2, 4, 0, 5, 3] [0, 1, 5, 2, 0, 3, 2, 4, 0, 5] |
| --- |

**My Research/work**

**The difference between the lists and the arrays**

| **Lists** |  | **arrays** |
| --- | --- | --- |
| Stores elements with different data types | Data Type | Stores elements of the same data type |
| The size of the list can be changed (dynamic array) | The size | The size of the array can’t be changed once created |
| Less efficient with large data set | Performance | More efficient with large datasets as it is stored in a block in the memory which allows for more efficient memory and caching |
| **- if you want to insert many elements with different data types**  **- if you want to change the size of data later** |  | **- if you want to store a very large datasets and perform mathematical operations on** |

2- Linkedlist: is a linear data structure consisting of Nodes;head & tails.

EX:Creating a linkedlist.

| class Node:  def \_\_init\_\_(self,value):  self.value=value  self.next=None class LinkedList:  def \_\_init\_\_(self):  self.head=None  self.tail=None  def add\_node(self,value):  new\_node=Node(value)  if self.head is None:  self.head=new\_node  self.tail=new\_node  else:  self.tail.next=new\_node  self.tail=new\_node  def print\_list(self):  curr\_node = self.head  while curr\_node is not None:  print(curr\_node.value)  curr\_node = curr\_node.next   a=LinkedList() a.add\_node(5) a.add\_node(6) a.add\_node(7) a.print\_list() *#the output of a.print\_list()* 5 6 7 |
| --- |

3- Stack: is a collection of items that can be accessed or manipulated based on the concept of ‘last input first output’ LIFO .

It consists of many operations:

* Push: to add the item to the top of the stack;using ‘append’
* Peek : to view the last element in the stack without removing it;using index[-1]
* Pop : to remove the items starting from the last one;using .pop()
* isEmpty: to check whether the stack is empty or not; using len()

EX: creating a stack

| class Stack:  def \_\_init\_\_(self):  self.items=[]   def push(self, item): *#add element*  self.items.append(item)     def peek(self):  return self.items[-1] *# peek operation is to view the last element inserted in the stack without removing it*   def pop(self):*# you can addd check if loop if stack is empty "no poping" else return*  return self.items.pop()    def is\_empty(self):  return len(self.items)==0   def size(self):  return len(self.items)   my\_stack=Stack() my\_stack.push(1) my\_stack.push(2) my\_stack.push(3) my\_stack.peek() *# the output of the last item inserted* 3 |
| --- |

4- Queue: is a collection of items that can be accessed and manipulated like stack but based on the First Input First Output ‘FIFO’

Operations made:

* Enqueue :to add elements to the end of the queue/list;using .append()
* Dequeue:to remove the first element of the queue and returning the value of this element; using .pop(0)
* isEmpty: to check whether the queue is empty or not; using len()==0(false or true)
* Size: to check the size of the queue; using len()
* To check the element removed in dequeue or the size of the queue is by using str()

| *#Queue* *# %%* class Queue:  def \_\_init\_\_(self) :  self.items= []    def enqueue(self,items):  self.items.append(items)    def dequeue(self):  if self.is\_empty():  print("Queue is empty")  return None  else:  return self.items.pop(0)  def is\_empty(self): *# a method used for the dequeue method in 'if' condition*  return len(self.items) == 0    def sizeOfQueue(self):  return len(self.items) q =Queue() q.enqueue(3) q.enqueue(4) q.enqueue(5) print(str(q.sizeOfQueue())) *# the value is converted to string to be able to be printed to the console* 3  print(str(q.dequeue())) *# returns the element removed which is the first element in the Queue* 3 print(str(q.sizeOfQueue())) 2 q.is\_empty() *# to check whether the queue is empty or not* False |
| --- |

5- Graph :

| class Graph:  def \_\_init\_\_(self):  self.graph={}    def add\_vertex(self,vertex):  if vertex not in self.graph:  self.graph[vertex]=[] *#dictionary key is list and value as a list B :- []*    def add\_edge(self,vertex1,vertex2):  if vertex1 not in self.graph:  self.add\_vertex(vertex1)    if vertex2 not in self.graph:  self.add\_vertex(vertex2)    self.graph[vertex1].append(vertex2)  self.graph[vertex2].append(vertex1)   def get\_vertice(self):  return list(self.graph.keys())    *# def getEdges(self):*  *# edges=[]*  *## print(self.graph.items())*  *# for vertex in self.graph:*  *# for neighbours in self.graph[vertex]:*  *# edges.append((vertex,neighbours))*  *# return edges*    def getEdges(self):  edges=[]  for key,value in self.graph.items():  for edge in value:  edges.append((key,edge))  return edges  g =Graph() g.add\_vertex("A") g.add\_vertex("B") g.add\_vertex("C")  g.add\_edge("A","B") g.add\_edge("B","C") g.add\_edge("C","A")  print(g.getEdges()) [('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'B'), ('C', 'A')] |
| --- |

6- Tree: a set of nodes connected by branches.

It consists of one parent node and child nodes;each child node can have zero or other child nodes.

When the node doesn’t have child nodes then it is called ‘leaf node’.

The length of the tree is the longest path from the parent/root node to the leaf node.

Applications where Tree structure is used:

* B-tree in partitioning in SQL to speed the look up process for the data retrieval.
* For file systems
* For family trees

Operations used for Class tree:

* Setting the root/parent node,leaf node and right node; using the \_\_init\_\_
* Insert: is to insert the new nodes in a specific position based on the value;if the value is less than the root node then it goes to the left node. If not then it goes to the right node.
* PrintTree: to sort the data in the tree from the smallest to the largest and print the data
* Searching: to search for a node in a tree; using like a pointer to start from the root/parent node

| *# Tree* class Node:  def \_\_init\_\_(self,data):  self.data= data  self.leftChild=None  self.rightChild=None   def printTree(self):  if self.leftChild:  self.leftChild.printTree()  print(self.data)  if self.rightChild:  self.rightChild.printTree()    def insert(self,data):   if data <self.data:  if self.leftChild:  self.leftChild.insert(data)  else:  self.leftChild = Node(data)  return  else:  if self.rightChild:  self.rightChild.insert(data)  else:  self.rightChild = Node(data)  return    def search(self,value):  if self.data==value:  return ('found:',self.data)  elif value < self.data and self.leftChild is not None:  return self.leftChild.search(value)  elif value > self.data and self.rightChild is not None:  return self.rightChild.search(value)  else:  return ('not found')   root = Node(27)  root.insert(14) root.insert(35) root.insert(31) root.insert(10) root.insert(19) root.printTree() |
| --- |

**Sorting data Algorithms**

**Different ways to sort a data set**

**Each Algorithm has different space complexity**

* Bubble sort : stable algorithm
* Merge sort : stable algorithm
* Quick sort : unstable algorithm
* Insertion sort : stable

**Option 1: Bubble\_sort**

| def bubble\_sort(arr):  n=len(arr)   for i in range(n):  for j in range(0,n-i-1):  if arr[j]>arr[j+1]:  arr[j],arr[j+1]=arr[j+1],arr[j]  a=bubble\_sort([14,27,35,10,19]) |
| --- |

**Option 2: Merge sort algorithm**

**def mergesort(arr):**

**if len(arr) ==1:**

**return arr**

| ***#Divide the array in 2 half*  mid=len(arr) // 2  left\_half=arr[:mid]  right\_half=arr[mid:]   *#Recursivel sort each half*  left\_sorted=mergesort(left\_half)  right\_sorted=mergesort(right\_half)   *#merge the halves*  i=j=0  result=[]  while i < len(left\_sorted) and j <len(right\_sorted):  if left\_sorted[i]<right\_sorted[j]:  result.append(left\_sorted[i])  i=i+1  else:  result.append(right\_sorted[j])  j=j+1  result +=left\_sorted[i:]  result+= right\_sorted[j:]  return result mergesort([27,14,35,10,19]) [10, 14, 27, 35, 19, 19, 35, 14, 27, 35, 19, 19, 35, 27, 35, 19, 19, 35]** |
| --- |

**Option 3: Quick sort Algorithm**

| **def quicksort(arr):  if len(arr) < 2:  return arr    *#choose the pivot element*  pivot = arr[0]   *#Partition the array in two sub arrays*  lesser =[i for i in arr[1:] if i<=pivot]  right =[i for i in arr[1:] if i > pivot]   return quicksort(lesser)+[pivot]+quicksort(right)  quicksort([27,14,35,10,19]) [10, 14, 19, 27, 35]** |
| --- |

**Option 4: Insertion sort Algorithm**

| **def insertion\_sort(arr):  for i in range(1,len(arr)):  key=arr[i]  j=i-1  while j>=0 and arr[j]>key:  arr[j+1]=arr[j]  j=j-1  arr[j+1]=key  return arr insertion\_sort([27,14,35,10,19]) [10, 14, 19, 27, 35]** |
| --- |

**Time & Space complexity**

**Time complexity: is the time the algorithm takes to output the data based on the data input.**

**Space complexity: is the amount of memory this algorithm takes based on the data input; measured based on the bytes or the number of variables created.**

**Example:**

**Int takes 4 bytes**

**Long takes 8 bytes**

**Double takes 8 bytes**

**Fun (N:int)** which takes 4 bytes and N here is limited by one value

**{**

**x=N** it will take 4 byte

**y= x\*x** it will take 4 byte

**z= x+y** it will take 8 bytes

**}**

Here it all takes 20 bytes

and these are all integers which is time constant so the time and space complexity is O(1)

But

If N is int and unlimited ;varies from 0 to 3000 for example

So z,y and z will vary depend on the value of N

Then the time complexity here is O(N).

**My Research**

**Time and Space Complexity for sorting data algorithms**

| **Sorting data Algorithm type** | **TC** | **SC** |
| --- | --- | --- |
| Bubble sort | O(N^2)  Two ‘for’ loops | O(1)  Variables are Time constant |
| Merge sort | O(NlogN)  Log N for mid=len(arr) // 2  N for while loop in comparing and i+=1 or j+=1 | O(N)  N for the array size  result=[] |
| Quick sort | O(NlogN)  For scanning the values linearly and then divide it  lesser =[i for i in arr[1:] if i<=pivot]  right =[i for i in arr[1:] if i > pivot] | O(logN)  Lesser takes logN  Right takes log N  logN+logN=2logN  lesser =[i for i in arr[1:] if i<=pivot]  right =[i for i in arr[1:] if i > pivot] |
| Insertion sort | O(N^2)  For nested loops:  ‘For’ and ’while’ loops | O(1)  Variables are time constant |

**Stability of Algorithms**

Stable algorithm : it maintains the order of the inputs having the same value in the output.

Unstable algorithm: there is no guarantee it will maintain the order of the inputs having the same value.

**Time limit Exceeded(TLE)**

That occurs when the program takes a long time to give the output which is not the optimal situation so it gives an error of ‘time limit exceeded’.

To avoid this error,is to take the input size into consideration for optimisation by:

Checking the number of iterations used in this program/algorithm as a function of the input size. ( the time complexity)

**My Research**

To check the time it takes for the program to execute:

Import time

start \_time=time.time(); to be written before the input data to be printed

end\_time=time.time(); to be written in the end of the program

print(end\_time-start\_time); to check the time this program took to execute the input data

Here we try different input sizes and check the time executed per each.

**OOP Examples**

Defining the initials for a **class** and its **object**:self , **Attribute**:title, **methods** for accessing/updating the data: getTitle and setTitle and **instances** for storing this data into.

Here for each attribute we may define its methods for accessing & updating the data under this attribute;as it is not a good practice to access the data using the initial definition.

| class book: *#POJO in*  *#here title is an instance variable for that we would define both getTitle for accessing the data/data retrieval and setTitle for updating the data*  def \_\_init\_\_(self,title):  self.title=title   def getTitle(self): *# for accessing the data*  return self.title    def setTitle(self,title): *# for updating the data*  self.title=title  # TODO create book instance variable b1  b1=book('getting things done') print(b1.getTitle()) b1.setTitle('diamond dust') print(b1.getTitle())  getting things done  diamond dust |
| --- |

Setting more attributes and more methods in return

Setting an optional attribute & optional method in return;using self.\_AttrNamesingle underscore

Setting a secret attribute;using self.\_\_AttrName double underscore

Deleting an attribute in an instance;using del instance\_name.Attr

Display all the attributes for an instance/printing an instance to get all the Attr.;using display(),\_\_str\_\_() or \_\_repr\_\_()

Equal comparison to compare between two instances ; using \_\_eq\_\_()

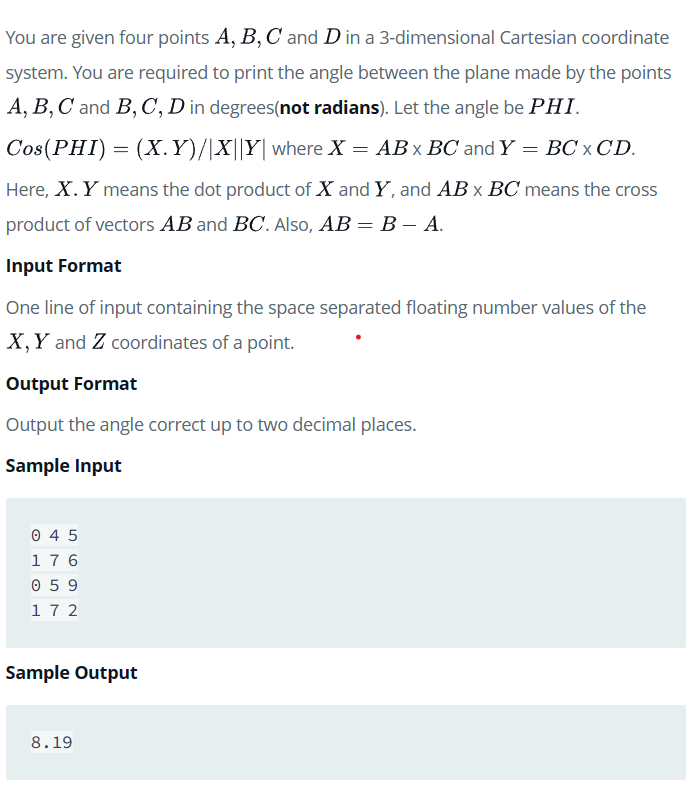
| *# inserting more attributes which will have in turn more methods* *# inserting optional attribute which will have in return optional method (setting discount)* class book: *#POJO in*  *#here title is an instance variable for that we would define both getTitle for accessing the data/data retrieval and setTitle for updating the data*  def \_\_init\_\_(self,title,author,pages,price,secret):  self.title=title  self.author=author  self.pages=pages  self.price=price  self.\_\_secret=secret + 'another title'  def getTitle(self): *# for accessing the data*  return self.title    def setTitle(self,title): *# for updating the data*  self.title=title  # optional attribute and method  def setDiscount(self,discount):  self.\_discount=discount/100 *# .\_discount the under score here means that it is an optional method*  return 'done'   def getPrice(self):  if hasattr(self,'\_discount'): *# here this is an optional attribute the price may have which is the discount*  return self.price - (self.price\*self.\_discount)  else:  return self.price  # Display function  def display(self): *# to display all the attributes for an instance*  print('title is %s\nPages are %s pages\nthe author is %s\nprice is %d euro'%(self.title,self.pages,self.author,self.price))    def \_\_str\_\_(self): *# used if we want simply to print the instance*  return f'title:{self.title}\npages:{self.pages}\nPrice:{self.price}'  def \_\_repr\_\_(self):  return f'title:{self.title}\npages:{self.pages}\nPrice:{self.price}'  # Comparison between two instances  def \_\_eq\_\_(self,value): # value here is the instance being compared to 'self' which is the first instance  if not isinstance(value,book):  raise ValueError('this instance is not found in class book')  return (self.title==value.title and self.author==value.author)    *# TODO create book instance variable b1 and b2* b1=book('getting things done','david allen',300,16,secret='has'  ) print(b1.getTitle()) print(b1.price) *# not a good practice to call the attributes in the initial* print(b1.getPrice()) print(b1.setDiscount(5));print(b1.getPrice()) |
| --- |
| *# Secret Attribute output*  *print(b1.\_book\_\_secret) # to call for a secret attribute*  *#print(b1.\_\_secret) # here it will give error to the user because this is a secret attribute so as an admin we write our secret query*  *print(b1.\_book\_\_secret) # the secret query*  *# another option is not to add the attr ‘secret’ and just define it as self.\_\_secret=secret*  def \_\_init\_\_(self,title,author,pages,price):  self.title=title  self.author=author  self.pages=pages  self.price=price  *self.\_\_secret='another title'*  *b1=book('getting things done','david allen',300,16)*  *print(b1.\_book\_\_secret)*  *another title* |
| *# deleting an attribute in an instance*  *del b1.price*  *print(b1.getPrice()) # here it will give error that b1 doesn't have 'price' attribute* |

| *# Display & Printing* output  b1.display() print(b1) print(str(b1)) print(b1.\_\_str\_\_()) *# same result as str()* print(repr(b1)) *# using the method repr to print the attr. option 1* print(b1.\_\_repr\_\_()) *# option 2: same result as repr()* *#output 1* title is getting things done Pages are 300 pages the author is david allen price is 16 euro *# output 2* title:getting things done pages:300 Price:16  *# Comparison output* print(b1==b2) *# here it will give true after doing the \_\_eq\_\_ method* class car:  def \_\_init\_\_(self,color):  self.color=color  def \_\_eq\_\_(self,value): *# value here is the instance being compared to 'self' which is the first instance*  if not isinstance(value,car):  raise ValueError('this instance is not found in class car')  return (self.color==value.color) c1=car('pink') *# print(b1==c1) # here it will raise the value error* *#print(c1==b1) # here it will raise the value error because b1 is not instance of the class car* print(b1.\_\_eq\_\_(b2)) *# here it does the equal method b1 is the main instance to be compared to and b2 is the value* |
| --- |

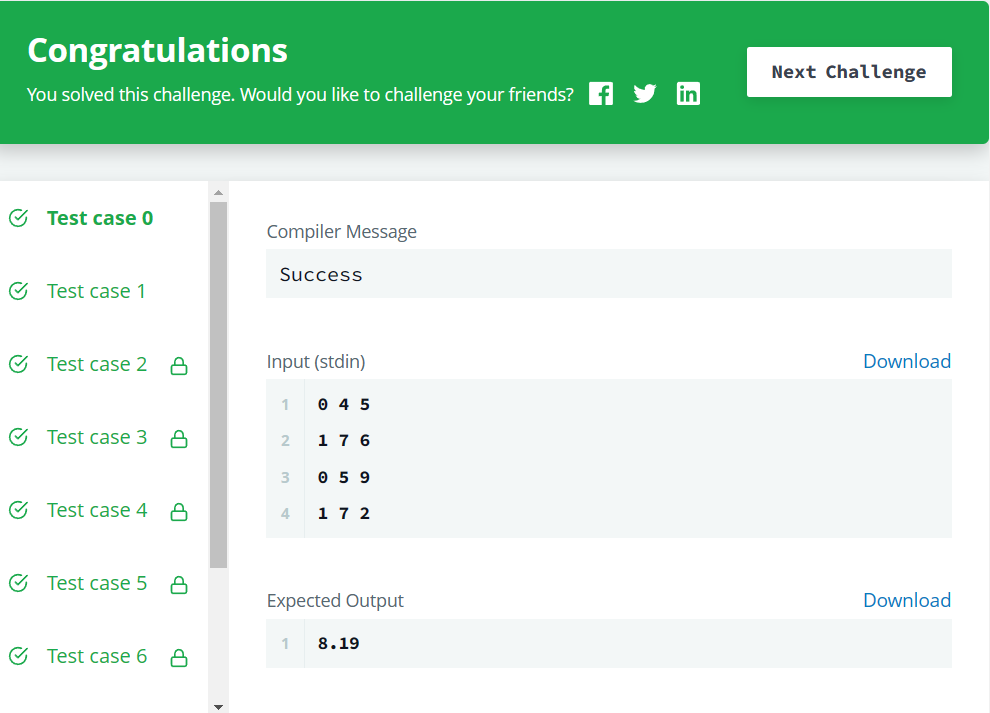
**My Research/Work**

**HackerRank Practice on Classes**

**Task: finding the torsional angle by creating the class and its methods.**

****

| **import math  class Points(object):  def \_\_init\_\_(self, x, y, z):  self.x=x  self.y=y  self.z=z  def \_\_sub\_\_(self, no):  return Points(self.x-no.x,self.y-no.y,self.z-no.z)  def dot(self, no):  return self.x\*no.x+self.y\*no.y+self.z\*no.z  def cross(self, other):  x = self.y \* other.z - self.z \* other.y  y = self.z \* other.x - self.x \* other.z  z = self.x \* other.y - self.y \* other.x  return Points(x, y, z)  def absolute(self):  return pow((self.x \*\* 2 + self.y \*\* 2 + self.z \*\* 2), 0.5)  if \_\_name\_\_ == '\_\_main\_\_':  points = list()  for i in range(4):  a = list(map(float, input().split()))  points.append(a)   a, b, c, d = Points(\*points[0]), Points(\*points[1]), Points(\*points[2]), Points(\*points[3])  x = (b - a).cross(c - b)  y = (c - b).cross(d - c)  angle = math.acos(x.dot(y) / (x.absolute() \* y.absolute()))   print("%.2f" % math.degrees(angle))** |
| --- |

****

**Difference between instance method,static method and classic method**

* Instance method(bond to the instance variables): Bond to the object of the class (self) and access/manipulate the instance variables
* Static method(bond to the class and not the instance): to access and manipulate the private class variables.
* Classic method(bond to the class and not the instance): to access the class-level constants.

EX: setting counter for each of the instance and the class using instance method,static method and classic method

| *# defining a counter for the instances and for the whole class* class book: *#POJO in JAVA*    \_\_BOOK\_COUNT=0  BOOK\_TYPES=('HARDCOVER','PAPERBACK','EBOOK') *# STATIC CONSTANTS*  \_\_BOOK\_LISTS=None *# can be secret variable \_\_BOOKLISTS=NONE*  *#here title is an instance variable for that we would define both getTitle for accessing the data/data retrieval and setTitle for updating the data*  def \_\_init\_\_(self,title,book\_type):  self.title=title  self.bookType=book\_type  self.instnc\_count=0  if not book\_type in book.BOOK\_TYPES:  raise ValueError('this book type is not supported')  else:  self.book\_type=book\_type   def getTitle(self): *# for accessing the data*  return self.title    def setTitle(self,title): *# for updating the data*  self.title=title   def setCount(self): *# its asscoiated with that b1 instance variable is not associated Book( contains 'self' so it is associated with instances)*  self.instnc\_count=self.instnc\_count+1  return self.instnc\_count   @staticmethod #this method is to set something at a specific class level  def getBookList():    if book.\_\_BOOK\_LISTS==None:  book.\_\_BOOK\_LISTS=[]  return book.\_\_BOOK\_LISTS    def incrementCount(): *# related to the whole class not an instance*  book.\_\_BOOK\_COUNT = book.\_\_BOOK\_COUNT +1      @classmethod # this to get a constant from the class level specified;static constants  def getBookTypes(cls):  return cls.BOOK\_TYPES    def returnCount(cls):  return cls.\_\_BOOK\_COUNT   *#print(book.getBookList()) # before adding any instances' titles or attributes, it will return an empty list*  b1 = book("title1","HARDCOVER") print('count for b1',b1.setCount()) book.incrementCount() b2= book("title2","PAPERBACK") book.incrementCount() print('count for b2',b2.setCount()) print(book.returnCount(book)) *#output* count for b1 1 count for b2 1 count for class book 2 |
| --- |

**Class level variables and instance variables**

**What happens when we change the class variable in terms of the instance variables and not the class name?**

| *#07.03.2023* *## what happens when we try to change a class-level variable in an instance variable?*  class Employee:  COMPANY\_NAME='Wiley' *# class variable / static variable*  def \_\_init\_\_(self,name):  self.name=name  print('first')    def \_\_init\_\_(self,name):  self.name=name  print('second') # trying defining two initials and see which initials will be used b1=Employee('rovan') *# it will use the second inital*  # using instance variable to change the class variable  b1.COMPANY\_NAME=5  print(b1.COMPANY\_NAME) *# gives 5; here the change is made to this instance only* print(Employee.COMPANY\_NAME) *# gives wiley; here it is for the whole class which will give the class variable*  b2=Employee('abdou') print(b2.COMPANY\_NAME) *# gives wiley; it will give the class-level variable*  print(b1.COMPANY\_NAME) *# so if you want to change a class variable, use the class name to change it and don't use an instance to change this class-level variable!*  # getting all the attributes to an instance print(b1.\_\_dict\_\_) *# it shows all the attributes to this instance* print(b2.\_\_dict\_\_)  #output  {'name': 'rovan', 'COMPANY\_NAME': 5}  {'name': 'abdou'}  # getting all the methods that can be used to a class  print(dir(book)) #(get all methods of a class)  ['BOOK\_TYPES', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_book\_\_BOOK\_COUNT', '\_book\_\_BOOK\_LISTS', 'getBookList', 'getBookTypes', 'getTitle', 'incrementCount', 'returnCount', 'setCount', 'setTitle'] |
| --- |

**INHERITANCE**

**Between classes**

Inheritance: it is an OOP that a class can borrow the attributes and the methods(for overriding) of another class;using subclass(base class) subclass=derived class & superclass=base class

**Ex:** 3 classes of book, magazine and newspaper. Each has ‘title’ and ‘price’ attributes.

Books attributes: title,price,author,pages

newspaper/magazine: title,price,period,publisher

Solution here is to get a class or two-class that has the common attributes between the classes

| class publication:  def \_\_init\_\_(self,title,price):  self.title=title  self.price=price class periodical(publication):  def \_\_init\_\_(self, title, price,period,publisher):  super().\_\_init\_\_(title, price)  self.period=period  self.publisher=publisher class book(publication):  def \_\_init\_\_(self, title,author,pages,price):  super().\_\_init\_\_(title, price)  self.author=author  self.pages=pages  class magzine(periodical):  def \_\_init\_\_(self, title, price, period, publisher):  super().\_\_init\_\_(title, price, period, publisher)  class Newspaper(periodical):  def \_\_init\_\_(self, title, price, period, publisher):  super().\_\_init\_\_(title, price, period, publisher)  b1=book('getting things done','david allen',300,16) m1=magzine('vogue',13,'monthly','abc') n1=NewsPaper('new york times',5,'weekly','def') print(b1.title) print(b1.\_\_dict\_\_) print(m1.\_\_dict\_\_) print(n1.\_\_dict\_\_) *# functions to check the relationship between classes and objects* print(issubclass(magzine,publication)) *# is the magazine a subclass of publication?*  print(issubclass(magzine,periodical))  print(isinstance(m1,magzine)) *# is the instance m1 belongs to the magazine?* print(isinstance(b1,magzine)) *# output* getting things done {'title': 'getting things done', 'price': 16, 'author': 'david allen', 'pages': 300} {'title': 'vogue', 'price': 13, 'period': 'monthly', 'publisher': 'abc'} {'title': 'new york times', 'price': 'def', 'publisher': 5, 'period': 'weekly'} True True True False |
| --- |

**Multiple inheritance**

When the class uses the attributes and the methods of two or more classes

| class A:  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.foo = 'foo'  self.name ='class A'  class B:  def \_\_init\_\_(self):  self.bar = 'bar'  self.nameb = 'class B'  class D:  def DD(self):  print('DD')  class C(A, B):  def \_\_init\_\_(self,namec):  super().\_\_init\_\_()  self.namec=namec    def showprops(self):  print(self.namec)  print(self.name) *# prints the name from A class instance*  print(self.bar)  print(self.nameb)   c = C('rovan') c.showprops() *#output* rovan class A bar class B |
| --- |

**My Research/work**

Ex:Playing the instrument: guitar,piano and guitar piano instruments

| ***#multiple inheritance* class Guitar:  def \_\_init\_\_(self):  self.type = 'acoustic'  self.strings = 6    def play(self):  print("Playing the guitar")  class Piano:  def \_\_init\_\_(self):  self.type = 'grand'  self.keys = 88    def play(self):  print("Playing the piano")  class AcousticGuitar(Guitar):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.type = 'acoustic'    def play(self):  print("Playing the acoustic guitar")  class ElectricGuitar(Guitar):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.type = 'electric'    def play(self):  print("Playing the electric guitar")  class GrandPiano(Piano):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.type = 'grand'    def play(self):  print("Playing the grand piano")  class GuitarPiano(AcousticGuitar, GrandPiano):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.type = 'hybrid'    def play(self):  print("Playing the guitar and piano at the same time!")  a=ElectricGuitar() a.play() b=AcousticGuitar() b.play() c=GrandPiano() c.play() d=GuitarPiano() d.play() *# output* Playing the electric guitar Playing the acoustic guitar Playing the grand piano Playing the guitar and piano at the same time!** |
| --- |

**NOTE: Here we can use the abstract method for ‘def play()’ in an ‘instrument’ class and to use this abstract method in the custom classes;guitar,piano,etc.**

**Abstract method is imported from ABC(Abstract base classes)**

| ***# in the previous example, we can use the abstract method for the play() in an 'instrument' class* from abc import ABC,abstractmethod class instrument(ABC):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.type=self.\_\_class\_\_.\_\_name\_\_ *# here it will return the type of the instrument customised by each class; guitar,piano,etc.*   @abstractmethod   def play(self):  pass  class guitar(instrument):  def \_\_init\_\_(self):  super().\_\_init\_\_()  *#self.type='guitar'*  def play(self):  print('playing the guitar')  a=guitar() A.\_\_dict\_\_ *#{'type': 'guitar'}* a.play()  *#output* playing the guitar** |
| --- |

**Overriding Methods used in inheritance between the classes**

As the class inherits the attributes and the methods of the superclass, the overriding happens when it comes to the methods inheritance.

Overriding methods happens when you want to change the behaviour of the same method in the subclass which exists in the superclass .

| *#methods overriding with the same instance* *# methods overriding is used when you want to change the behaviour of the method for each class* *# if you don't want to and the behaviour is the same anyway; it is better to user super() method* class Animal:  def speak(self):  print("speaking") class Dog(Animal):  def speak(self):  print("barking")  d =Dog() d.speak() *# here it will print the speak method of Dog class; as it is the last command written*  *# output*  *barking* |
| --- |

However we can avoid the overriding method using super()

Ex: getting the area of the ‘square’ class from the ‘area’ method in the ‘rectangle’ class and adding to the behaviour of the method using super()

| *# to avoid the overriding methods* class rectangle:  def \_\_init\_\_(self,x,y):  self.x=x  self.y=y  def area(self):  print (self.x\*self.y)  class square(rectangle):  def area(self):  super().area()  print('the area of the square')  a=rectangle(5,3) a.area() b=square(6,4)  b.area() *#output* 15 24 the area of the square |
| --- |

**\_\_Post \_initial\_\_ method**

**Using data classes library**

**In version 3.7 & later**

Post initial method is used to do additional initialization after the object is initalized and you don’t have to create def \_\_init\_\_() but just by declaring the attributes.

| ***# using data classes* from dataclasses import dataclass  @dataclass class book:  Title:str # just declaring the attributes without \_\_init\_\_()  author:str  pages:int  price:float  def \_\_post\_init\_\_(self):  if self.price<=0:  raise ValueError('insert a price')  b=book('gf','dsz',300,0) *#print(dir(b))* print(b.\_\_dict\_\_) *#output* ValueError: insert a price** |
| --- |

**My Research/work**

**Post init method deals with the attributes of the class;doing mathematical operations/logical operations/finding the value of another attribute/calling methods**

Uses of \_\_Post\_initial\_\_():

* Data validation: to make sure the data inserted in the attributes is valid

In the previous example: to make sure the price of the book entered is valid.

* Data Cleaning: to clean the data in the instance variables.

Ex:

| from dataclasses import dataclass @dataclass class student:    name: str  email:str  def \_\_post\_init\_\_(self):  self.email=self.email.lower().strip()  s=student(name='rovan',email='ROVAN.ELGENDI@gmail.com') print(s.email) rovan.elgendi@gmail.com |
| --- |

* Computation: to get the result out of two or more instance variables(mathematical operations for example)

Ex: getting the area attribute of a rectangular shape automatically after setting the length and width attributes; computing the third attribute in terms of the other attributes.

| *# 2 using post init method for computation* from dataclasses import dataclass  @dataclass class Rectangle:  length: float  width: float  area: float = 0.0   def \_\_post\_init\_\_(self):  self.area = self.length \* self.width  *# Creating an instance of Rectangle class* *#rect=Rectangle(5,3)*  *# or* rect = Rectangle(length=5.0, width=3.0) print(rect.area) *# Output: 15.0* |
| --- |

* Dependency : if an attribute value is dependent on another attribute so the post init method is used to do the ‘if’ statement for ex.

Ex: getting the discount on the price of a book depending on the price itself; one attribute depends on another one

| *# 3 using the post init method for dependency; one attribute is dependent on another attribute.* from dataclasses import dataclass  @dataclass class Book:  title: str  author: str  price: float  discount: float = field(default=0.0)   def \_\_post\_init\_\_(self):  if self.price <= 0:  raise ValueError("Price must be positive")  self.discount = self.calculate\_discount()   def calculate\_discount(self):  if self.price >= 100:  return 0.2  elif self.price >= 50:  return 0.1  else:  return 0.0  # Creating an instance of Book class  book=Book('dfg','fde',55)  #book = Book(title="The Great Gatsby", author="F. Scott Fitzgerald", price=120.0)  print(book.discount) # Output: 0.2  print(book.\_\_dict\_\_)  *0.1*  *{'title': 'dfg', 'author': 'fde', 'price': 55, 'discount': 0.1}* |
| --- |

**ABC ( Abstract Base Class)**

| *# using ABC to convert some data to JSON* *# using abstract method* *# ABC means Abstract Base Classes that can be used as a base for custom classes that uses this abstract class* from abc import ABC,abstractmethod class GraphicShape(ABC):  *#def \_\_init\_\_(self):*  *# super().\_\_init\_\_()*  *# a method with an empty body; if you dont know the logic, you declare this method abstract and the class which will inherit the abstract method would have to implement it.*  @abstractmethod  def CalcArea(self):  pass class JSONify(ABC): |
| --- |



| @abstractmethod  def toJSON(self):  pass class HTML(ABC):  @abstractmethod  def toHTML(self):  pass class Circle(GraphicShape,JSONify,HTML):  def \_\_init\_\_(self,radius):  super().\_\_init\_\_()  self.radius=radius   def CalcArea(self):  return 3.14\*(self.radius\*\*2)  def toJSON(self):  return f'{{\n\"Circle\":\n {{\n "radius\":{self.radius},\n "area\":{self.CalcArea()}\n }}\n}}'  def toHTML(self):  return f'<Circle>{self.CalcArea()}</Circle>' class square(Circle,JSONify):  def \_\_init\_\_(self, radius):  super().\_\_init\_\_(radius)  self.side=radius  def CalcArea(self):  return self.side\*\*2  def toJSON(self):  print(f'{{\n\"square\":\n {{\n "each\_side\":{self.side},\n "area\":{self.CalcArea()}\n }}\n}}')    c=Circle(15) c.CalcArea() print(c.toJSON()) c.toHTML() s=square(4) s.CalcArea() s.toJSON() #output { "Circle":  {  "radius":15,  "area":706.5  } } { "square":  {  "each\_side":4,  "area":16  } } |
| --- |

**Composition**

When you use another class attributes in a specific class.

Adv: to decrease the redundancy of data like SQL.

| *#Composition*  class Author:  def \_\_init\_\_(self,fname,lname):  self.fname=fname  self.lname=lname  def \_\_str\_\_(self):  print(self.fname,self.lname)   class Book:  def \_\_init\_\_(self,title,price,author=None):  self.title=title  self.price=price  self.author=author   self.chapters=[]  def addChapters(self,name,pages):  self.chapters.append((name,pages))  a1= Author('leo','egxd') b1=Book('fsgs',39,a1) b2=Book('rds',55,a1) b1.addChapters('chapter 1',124) b1.addChapters('chapter 2',44)  b1.\_\_dict\_\_ |
| --- |

**Magical Methods & \_\_call\_\_method**

* Magical methods also known as dunder methods; starts and ends with double underscore \_\_

ex: \_\_init\_\_ , \_\_str\_\_ ,\_\_getattribute\_\_,\_\_setattr\_\_ and \_\_getattr\_\_,  
\_\_getattribute\_\_,\_\_setattr\_\_ and \_\_getattr\_\_: customises the behaviour of the attributes access in the class

* \_\_call\_\_ method: is a built-in method used to a callable object/instance.whenever you want to treat the object/instance as a function, we use \_\_Call\_\_ method to define the behaviour of this instance.

| class Book:  def \_\_init\_\_(self,title,author,pages,price):  self.title=title  self.author=author  self.price=price  self.pages=pages  self.discount =0.1   def \_\_str\_\_(self):  return f"{self.title} by {self.author}, costs {self.price}"   def \_\_getattribute\_\_(self,name: str) : *# this method is used/get called automatically when there is an attempt to access an attribute whether it is existed or not.you can add any actions/operations or even raise an error if the attr is not existed before setting this attribute*   if (name == "price"):  p =super().\_\_getattribute\_\_("price")  d=super().\_\_getattribute\_\_("discount")  return p -(p \* d)  return super().\_\_getattribute\_\_(name)    def \_\_setattr\_\_(self,name: str, value: str): *# this method is used/get called automatically when there is an attempt of setting an attribute value in the class*  if name =='price':  if type(value) is not float:  raise ValueError("The 'price' attribute must be float")  return super().\_\_setattr\_\_(name,value)     def \_\_getattr\_\_(self,name): *# this method is used/get called automatically when there is an attempt to get an attribute that doesn't exist in the class*  return name +" is not a variable in book class!!!"    def \_\_call\_\_(self,title,author,pages,price): *# here using the call method to update a data in a variable without having to go to each attribute and update it*  self.title=title  self.title=title  self.author=author  self.price=price  self.pages=pages     b1 = Book("ware and peace","leo tolsty",32332,39.65) b2 = Book("the catcher in rye","JD",322,29.65)  b1.price = 38.65 print(b1)  b2.price=float(40) print(b2)  print(b1.randomprop) *#output* ware and peace by leo tolsty, costs 34.785 the catcher in rye by JD, costs 36.0 randomprop is not a variable in book class!!! |
| --- |
| *# \_\_call\_\_ output* b1.\_\_call\_\_('tgrd','ghf',150,55.9) print(b1) tgrd by ghf, costs 50.31 *# after the discount*  b1('efdx','tyyy',456,40.30) print(b1) efdx by tyyy, costs 36.269999999999996  # comparison output  print(b1 == b2) # false  print( b2 >=b1) # it will give error as it is not supported so we define the \_\_gt\_\_ and \_\_lt\_\_ methods |

**Built-in Comparison Methods**

Comparison methods are the comparison between the instances/objects.

Used for:

* Sorting data in ascending/descending order
* Filtering
* Data analysis: take decisions based on the results sorted/filtered.

These methods are : \_\_lt\_\_,\_\_le\_\_,\_\_gt\_\_,\_\_ge\_\_,\_\_eq\_\_,\_\_ne\_\_

# to sort objects

class Book:

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

self.title=title

self.author=author

self.price=price

self.pages=pages

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

return f"{self.title} by {self.author}, costs {self.price}"

# its greatr than function

def \_\_ge\_\_(self,value):

if not isinstance(value,Book):

raise ValueError("Can't compare book with a non book")

return self.price >= value.price

# its lower than function

def \_\_lt\_\_(self,value):

if not isinstance(value,Book):

raise ValueError("Can't compare book with a non book")

return self.price < value.price

def \_\_le\_\_(self,value):

if not isinstance(value,Book):

raise ValueError('this object is not in Book class')

return self.price <= value.price

b1 = Book("ware and peace","leo tolsty",32332,39.65)

b2 = Book("the catcher in rye","JD",322,29.65)

b3 = Book("ware and the peace","JD",322,10.55)

b4 = Book("to kill","JD",322,19.65)

print(b1 == b3)

print(b1 == b2)

print(b4 == b1)

#here first parameter becomes self and and second parameter becomes value in gt and lt function

print( b2 >=b1)

print(b2 < b1)

print(b3 >=b2)

books=[b1,b4,b3,b2]

books.sort()

# or

#books.sort(reverse=True) # to sort in descending order

listing=list()

for book in books:

listing.append(book.\_\_dict\_\_)

print(listing)

# or

#print([book.\_\_dict\_\_ for book in books ])

print([book.title for book in books ])

#output

False

False

False

False

[{'title': 'ware and the peace', 'author': 'JD', 'price': 10.55, 'pages': 322}, {'title': 'to kill', 'author': 'JD', 'price': 19.65, 'pages': 322}, {'title': 'the catcher in rye', 'author': 'JD', 'price': 29.65, 'pages': 322}, {'title': 'ware and peace', 'author': 'leo tolsty', 'price': 39.65, 'pages': 32332}]

['ware and the peace', 'to kill', 'the catcher in rye', 'ware and peace']

**Data Encapsulation**

| *# to DEbug* *# data encapsulation is the process of restricting direct access to an object's data or attributes from outside the class.* *# This is achieved by making the data private or protected and providing public methods to access and modify the data.* *#Python implements data encapsulation by using naming conventions for variables and methods.* *# By convention, variables and methods that are intended to be private or protected are prefixed with underscores.* *# Example 2: Data Encapsulation in Python*  class Employee:  def \_\_init\_\_(self, name, salary):  self.name = name  self.\_\_salary = salary   def show(self):  print("Name is ", self.name, "and salary is", self.\_\_salary)  *# Outside class* E = Employee("Bella", 60000) *#E.PrintName()* print(E.name) print(E.\_\_dict\_\_) *#print(E.PrintName())* *#print(E.\_\_salary) # AttributeError: 'Employee' object has no attribute '\_\_salary'* print(E.\_Employee\_\_salary) *# here it will access the 'salary' attribute*  class Computer:   def \_\_init\_\_(self):  self.\_\_maxprice = 900   def sell(self):  print("Selling Price: {}".format(self.\_\_maxprice)) *# it is like f string*   def setMaxPrice(self, price):  self.\_\_maxprice = price  c = Computer() c.sell()  *# change the price* c.\_\_maxprice = 1000 c.sell()  *# using setter function which is one of the data encapsulation uses ' Getter & Setter '*  c.setMaxPrice(1000) c.sell()  *#output 1* Bella {'name': 'Bella', '\_Employee\_\_salary': 60000} 60000  *#output 2* Selling Price: 900 Selling Price: 900 Selling Price: 1000 |
| --- |

**Polyphorism**

| *# to Debug* *#polymorphism, which is one of the fundamental concepts in object-oriented programming (OOP).* *#Polymorphism allows different objects to be treated in a common way, even if they have different implementations for the same methods.* *# In this case, the Parrot and Penguin classes have different implementations for the fly method, but they both have a fly method and can be treated in a common way.* class Parrot:  def fly(self):  print("Parrot can fly")    def swim(self):  print("Parrot can't swim")  class Penguin:  def fly(self): *# overriding method*  print("Penguin can't fly")    def swim(self):  print("Penguin can swim")  *# common interface* def flying\_test(bird): *# polymorphism which acts on the objects that have the same method (have the overriding methods)*  bird.fly()  *#instantiate objects* blu = Parrot() peggy = Penguin()  *# passing the object* flying\_test(blu) flying\_test(peggy) |
| --- |

**Decorators**

*# decorator is a special type of function that is used to modify or extend the behaviour of other functions or classes.*  
*# Decorators are themselves functions that take another function as an argument*  
*# and return a new function that wraps the original function with some additional functionality.*

| import datetime def my\_decorator(my\_function): *# to make an extension of a function*  def inner\_decorator(\*args):  print("this happend before") *# you add the balance*  my\_function(\*args) *# when you call my function the my\_decorated function get executed # here you do the audit function call*  print(datetime.datetime.utcnow()) *# during the process time*  print("this happened after") *# ensure database is updated*  print("This happend at end!!!!") *# drop a message on console*  return inner\_decorator  @my\_decorator def my\_decorated():  print("welcome to first lecture of class",datetime.datetime.utcnow()) @my\_decorator def add (a,b):  print (a+b) *# whenever you write a python file first function which get invoked is main function* if \_\_name\_\_ == "\_\_main\_\_": *# these code check name of the function is it main*  print(datetime.datetime.utcnow()) *# the start time*  add(3,4)  print(datetime.datetime.utcnow()) *# the end time*  my\_decorated()   *# output 1 for add() function* 2023-03-09 10:24:37.080842 this happend before 7 2023-03-09 10:24:37.080842  *# output 2 for my\_decorated() function* this happened after This happend at end!!!! 2023-03-09 10:24:37.081353 this happend before welcome to first lecture of class 2023-03-09 10:24:37.081353 2023-03-09 10:24:37.081353 this happened after This happend at end!!!! |
| --- |

**My work on subtracting/summing using decorators**

| ***#doing summing/subtracting operations* def operations(my\_function):  def inner\_decorator(\*args,\*\*kwrgs):  *# print("this happend before")*  try:  my\_function(\*args,\*\*kwrgs)  except Exception as ex:  print('stop!!',ex)   return inner\_decorator   @operations def add (\*args:int)->int:  print (sum(args)) @operations def subtr(\*args:float)->float:  sub=args[0]  for i in range(1,len(args)):sub=sub-args[i]  print(sub)   if \_\_name\_\_ == "\_\_main\_\_":  add(3,'g')  add(5,76)  subtr(4,2.1) *#output* stop!! unsupported operand type(s) for +: 'int' and 'str' 81 1.9** |
| --- |

| *# using \*\*kwrgs* def operations(my\_function):  def inner\_decorator(\*args,\*\*kwrgs):  *# print("this happend before")*  try:  my\_function(\*args,\*\*kwrgs)  except Exception as ex:  print('stop!!',ex)   return inner\_decorator   @operations def add (\*args:int,\*\*kwrgs)->int:  x=sum(args)  for key,value in kwrgs.items(): x=x+value  print(x) @operations def subtr(\*args:float)->float:  sub=args[0]  for i in range(1,len(args)):sub=sub-args[i]  print(sub)   if \_\_name\_\_ == "\_\_main\_\_":  add(3,7,item=30)  add(item1=4,item2=6)  add(3,7,item='f')  subtr(4,2.1)  *#output* 40 10 stop!! unsupported operand type(s) for +: 'int' and 'str' 1.9 |
| --- |

**Type of Errors and Error Customisations**

**VERY IMPORTANT FOR EXCEPTION/ERROR HANDLING**

| *# Error types* try :   print(1/0) *# failuere codnition code can be written here*  raise RuntimeError("you are wrong") except ZeroDivisionError as error: *# only when you get an error*  print("an error occured",error) except Exception as ex: *# only when you get an error*  print("from exception",ex) finally: *# these is always getting executed*  print("hi how are ayoposadjklasndksanknd")  print("I am good") |
| --- |

| *# customized error* class customerror(Exception):  def \_\_init\_\_(self,message):  self.message=message   def \_\_str\_\_(self):  return f'{self.\_\_class\_\_.\_\_name\_\_}:{self.message} written by Rovy'   print('welcome') raise customerror('sorry here is an error for you!')  #output welcome  --------------------------------------------------------------------------- customerror Traceback (most recent call last) Cell In[991], line 10  7 return f'{type(self).\_\_name\_\_}:{self.message} written by Rovy'  9 print('welcome') ---> 10 raise customerror('sorry here is an error for you!')  customerror: customerror:sorry here is an error for you! written by Rovy |
| --- |

| *# customized error* *# here it can be raised without writting custom message* *# both are the same depends on the user prefering* class customerror(Exception):  pass   print('welcome') raise customerror('sorry here is an error for you!') *#output* welcome  --------------------------------------------------------------------------- customerror Traceback (most recent call last) Cell In[998], line 7  4 pass  6 print('welcome') ----> 7 raise customerror('sorry here is an error for you!')  customerror: sorry here is an error for you! |
| --- |

**Quizes Notes**

**1-**

**T**his question has a mistake as we can’t add to the tuples but just to do sum operations +(‘abc’,)

Which of the following statements will add an item to a tuple named "trees"?

Select one:

a.

trees = ("maple")

b.

trees.append("maple")

c.

trees.add("maple")

d.

trees.insert("maple")

**2- good question as i answered it partially correct(the first two options) but the third options to be included as well**

Which of the following statements are true about dictionaries?

Select all appropriate answers:

a.

A dictionary is a collection of items and each item includes a key and a value.

b.

Each key in a dictionary acts as an index on an item in the dictionary, so each key must be unique within the dictionary.

c.

Each value in a dictionary acts as an index on an item in the dictionary, so each value must be unique within the dictionary.

d.

A dictionary allows the developer to define indexes on the values in the collection, rather than depending on Python to assign indexes.