27/02/2023

Python:- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together

Interpreted Language:- Interpreted languages are those programming languages that use software called interpreters to convert the high-level source code into machine language.

Machine Language:- Machine language is one of the low-level programming languages which is the first generation language developed for communicating with a Computer. It is written in machine code which represents 0 and 1 binary digits inside the Computer string which makes it easy to understand and perform the operations.

Assembly Language is the second generation programming language that has almost similar structure and set of commands as Machine language. Instead of using numbers like in Machine languages here we use words or names in English forms and also symbols. The programs that have been written using words, names and symbols in assembly language are converted to machine language using an Assembler. Because a Computer only understands machine code languages that's why we need an Assembler that can convert the Assembly level language to Machine language so the Computer gets the instruction and responds quickly.

High Level Language:-The high level languages are the most used and also more considered programming languages that helps a programmer to read, write and maintain. It is also the third generation language that is used and also running till now by many programmers. They are less independent to a particular type of Computer and also require a translator that can convert the high level language to machine language. The translator may be an interpreter and Compiler that helps to convert into binary code for a Computer to understand.

**Compiled and Interpreted Languages are two of the most important terms to know in computer programming. They are both commonly used, but they serve different purposes.

Python Syntax:-
print("Hello, World!")

Python Indentation:-Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

Comments:- Comments can be used to explain Python code.

- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.

1. Single Line comment:- Comments starts with a #, and Python will ignore them
   - Comments can be placed at the end of a line, and Python will ignore the rest of the line.
2. Multiline comments:- To add a multiline comment you could insert a # for each line:
   OR YOU CAN USE
   """
   This is a comment
   written in
   more than just one line
   """

Variables:- variables are containers for storing data values.

Creating Variables:- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.

Get the Type:- You can get the data type of a variable with the type() function.

Single or Double Quotes:- String variables can be declared either by using single or double quotes.

Case-Sensitive:- variable names are case-sensitive.

Identifier Naming

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

- The first character of the variable must be an alphabet or underscore (_).
- All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).
- Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, *).
- Identifier name must not be similar to any keyword defined in the language.
- Identifier names are case sensitive; for example, my name, and MyName is not the same.
- Examples of valid identifiers: a123, _n, n_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.

List:- Lists are used to store multiple items in a single variable.

- Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
- Lists are created using square brackets [].

List Items:- List items are ordered, changeable, and allow duplicate values.

- List items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered:- When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

- If you add new items to a list, the new items will be placed at the end of the list.

Changeable:- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates:- Since lists are indexed, lists can have items with the same value.

List Length:- To determine how many items a list has, use the len() function.

The list() Constructor:- It is also possible to use the list() constructor when creating a new list.

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

Tuple:- Tuples are used to store multiple items in a single variable.

- Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
- A tuple is a collection which is ordered and unchangeable.
- Tuples are written with round brackets.

Tuple Items:- Tuple items are ordered, unchangeable, and allow duplicate values.

- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered:- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable:- The tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates:- Since tuples are indexed, they can have items with the same value.

The tuple() Constructor:- It is also possible to use the tuple() constructor to make a tuple.

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

Set:- Sets are used to store multiple items in a single variable.

- Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
- A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.
- Set *items* are unchangeable, but you can remove items and add new items.

CASE STUDY

```
#To print statement in python

print("Hello world!")
# Print example
print("welcome")
a = 10
print("a=",a)
b=a
print("a =",a, '=b')

#sep keyword
a=10
print("a =",a,sep='dddd',end='\n\n\n')
print("a =",a,sep= '0',end='$$$$')
#example by mate
a = 10
b = 20
print(f'value of number a: {a}.\n value of number b: {b}.\n their sum: {a + b}')


# Input keyword
a= int(input("Enter the number"))
print(a)
b= int(input("Enter the second number"))
print(a+b)

# List example
L1 = ["Aneri", 100, "Germany"]
L2 = [2,3,4,5,6,7]
print(L1)
print(L2)


# to find data type

print(type(L1))
print(type(L2))
print(type(a))

#create list with 100 zeros

list1 = [0] * 100
print(len(list1))

#To find the length of the string with LEN keyword

L3 = ["appy", "cola", "orangejuice", "lime"]
print(L3[2])

# tuple
tup=("appy", "cola", "orangejuice", "lime")
print(tup)
# create tuple() with constructor

thistuple = tuple(("apple", "banana", "cherry"))
print(thistuple)

# we have to put comma if there is one value then only it consider it as tuple

thistuple = ("apple",)
print(type(thistuple))
```

```
#NOT a tuple

thistuple = ("apple")
print(type(thistuple))

# How to use quotes in python

str1="Hi Python"
print(str1)
str2='Hi Python'
print(str2)
str3=''' Hi Python '''
print(str3)
str4=' Hi Friend\'s '
print(str4)
```

28/02/2023

Range:- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

Syntax:
range(start, stop, step)

- start:-Optional. An integer number specifying at which position to start. Default is 0
- Stop:-Required. An integer number specifying at which position to stop (not included).
- Step:- Optional. An integer number specifying the incrementation. Default is 1

Loops:- We can run a single statement or set of statements repeatedly using a loop command.

Sr.No.	Name of the loop	Loop Type & Description
1	**While loop**	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	**For loop**	This type of loop executes a code block multiple times and abbreviates the code that manages the loop variable.
3	**Nested loops**	We can iterate a loop inside another loop.

Loop Control Statement:- Statements used to control loops and change the course of iteration are called control statements. All the objects produced within the local scope of the loop are deleted when execution is completed.

Sr.No.	Name of the control statement	Description
1	**Break statement**	This command terminates the loop's execution and transfers the program's control to the statement next to the loop.
2	**Continue statement**	This command skips the current iteration of the loop. The statements following the continue statement are not executed once the Python interpreter reaches the continue statement.
3	**Pass statement**	The pass statement is used when a statement is syntactically necessary, but no code is to be executed.

1. The for loop:-
   Syntax:-
   for value in sequence:
   { code block }

2. While Loop:-While loops are used in Python to iterate until a specified condition is met. However, the statement in the program that follows the while loop is executed once the condition changes to false.

   Syntax:-
   while <condition>:
   { code block }

Continue:-continue keyword return control of the iteration to the beginning of the Python for loop or Python while loop. All remaining lines in the prevailing iteration of the loop are skipped by the continue keyword, which returns execution to the beginning of the next iteration of the loop.

Pass:- The pass keyword is used when a phrase is necessary syntactically to be placed but not to be executed.

3. Nested Loop:- A nested loop is a loop inside a loop.

   Example:-

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
  for y in fruits:
    print(x, y)
```

Strings indexing and splitting:- string starts from 0

str = "HELLO"

H	E	L	L	O
0	1	2	3	4

str[0] = 'H'

str[1] = 'E'

str[2] = 'L'

str[3] = 'L'

str[4] = 'O'

List Indexing and Splitting:-The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].

Syntax:-
list_var(start:stop:step)

- The start denotes the starting index position of the list.
- The stop denotes the last index position of the list.
- The step is used to skip the nth element within a start:stop

Python List Built-in Functions:-

1. len() - It is used to calculate the length of the list. - **len(list1)**
2. max() - It returns the maximum element of the list - **print(max(list1))**
3. min() - It returns the minimum element of the list- **print(min(list1))**

CASE STUDY

```
# To get the RANGE
r = range(10)
print(r)

# To print range in reverse using reversed keyword
for i in reversed(range(5)):
    print(i)

# incremented by 4 using reversed
for i in reversed(range(0, 30, 4)):
    print(i, end=" ")
print()

# Odd number using tuple
```

```
tuple = (2,5,6,7,8,10,11,13,14,15)
for value in tuple:
        if value%2!=0:
            print(value)
print("Printed all odd number")

# Even number using tuple
tuple = (2,5,6,7,8,10,11,13,14,15)
for value in tuple:
        if value%2==0:
            print(value)
print("Printed all even number")

#While loop
counter = 0
while counter <10:
    counter +=3 # counter = counter + 3
    print("Python while loop")

# While using else
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")

# Just to understand difference between Pass and Continue
# Python program to demonstrate
# difference between pass and
# continue statements

s = "wankhade"

# Pass statement
for i in s:
    if i == 'k':
        print('Pass executed')
        pass
    print(i)

print()


# Continue statement
for i in s:
    if i == 'k':
        print('Continue executed')
        continue
    print(i)

# to get square value
numbers = [3,4,20,10,5]

sum = 0
for num in range(len(numbers)):
    sum= sum + numbers[num] ** 2
print(sum)

# nested loop
cars = [ "Lexus" , "BMW" , "Audi" ]
colors =[ "black" , "blue" , "white" ]

for car in cars:
   for color in colors:
      print(color+" "+car)
```

```
# nested loop
sudent_firstname = 'anna'
student_lastname = 'adva'

record= {'anna': 85, 'abcd': 50, 'defg' :69 }
for (name,mark)in record.items():
    if mark>80:
        print(name,mark)
    else:
        continue
#it gives the name&mark for marks>80
```

```
# import random and use of append

import random
numbers=[]
for i in range(0,11):
    numbers.append(random.randint(0,11))

for num in range(0,11):
    for i in numbers:
        if num == i:
            print(num, end="")
```

```
# nested loop
for i in range(0,10):
    for j in range(1,10):
        print(i,j)
        # print(i+1,0)

# multiplication table using while loop

num = 24
counter = 1
print(f"The multiplication table of {num}")
while counter <=10:
    ans=num*counter
    print(num,'x', counter, '=', ans)
    counter= counter + 1
```

```
# pop() function
list = [2,5,6,4,8]
term = []
while list:
    term.append(list.pop()**2)
print(term)
```

```
#when we multiply data structure with list we get duplicate values not the multiplication

list1=[1,2,3,4,5,6,7,8]
l= list1 * 2
print(l)
```

So the output for this is :-

[1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8]

```
# Addition between two list
list1=[1,2,3,4,5,6,7,8]
list2=[11,12,33,14,15,16,17]
l= list1 + list2
```

```
print(l)

#example of list using for loop
list= [2,3,4,5,6,7,8,9]
pf= [0]* len(list)
pf[0] = list[0]
for i in range(1, len(list)):
    pf[i] = list[i-1] + list[i]
print(pf)

# Example using index, range and list
list=[21,23,213,413,123,412]
pf=[0]*len(list)
pf[0]=list[0]
for i in range(1,len(list)):
    pf[i]=pf[i-1]+list[i]
print(pf)
def sum(startindex,endIndex):
    if startindex ==0:
        return pf[endIndex]
    return pf[endIndex]-pf[startindex-1]
print(sum(3,5))
```

02/03/2023

Time Complexity

→ Theorem to sum N Natural number

N = 100

$S = 1+2+3+4+\ldots+100$

$S = 100+99+98+97\ldots+1$

$2S = 101+101+101+101+101+\ldots+101$

$2S = 100 * 101$

$S = \frac{100 * 101}{2}$

$S = N * (N+1) / 2$

→ Logarithm

$b^c = a$

$\log b^c = \log a$

$c * \log b = \log a$

$c = \log_b^a$

$\log_2^{64} = \log_2^{8^2}$

$= 2 * \log_2^{2^3}$

$= 2 * 3 * \log_2^2$

$= 2 * 3 * 1$

$= 6$

* Given N find number of steps to reach 1 by dividing N/2

Ex: N = 16

$16/2 = 8/2 = 4/2 = 2/2 = 1$

For 16 we need → 4 steps.

N = 63

$63/2 = 31/2 = 15/2 = 7/2 = 3/2 = 1$

For 63 we need → 5 steps.

⇒ $N/2^0 \rightarrow N/2^1 \rightarrow N/2^2 \rightarrow N/2^3 \ldots N/2^k$

$N/2^n = 1$

$N = 2^k$

$\log N = k * \log_2^2$

$k = \log_2^N$

Example

```
for i in range (1, 100)
    {
        S = S+i
    }
```

→ Time Complexity

⇓

How much time it would take to execute the code.

O(N)

→ It will execute N time
→ N will be time require.

O(N)

↗

Big Oh. ← Pronounce

Ex:

```
for (i=1 to N)          → N
{
   if i%2 == 0          → 1      } Execution time.
   print i              → 1
}
```

$O(N+1+1)$

$\boxed{T = O(N)}$

⇒ we don't write any constant values with N.

Ex

```
for 1 to √N
{
}
```

$O(\sqrt{N})$

Ex for i 1 to N

$1 \rightarrow 2^0$
$2 \rightarrow 2^1$
$4 \rightarrow 2^2$
$8 \rightarrow 2^3$
$16 \rightarrow 2^4$

$N = 2^k$

$k = \log N$

$\boxed{T = O(\log N)}$

Ex
```
for i 1 to 10
{
   for j 1 to N
   {
      print i*j
   }
}
```

i	j	Iteration
1	[1 to N]	N
4	[1 to N]	N
3	[1 to N]	N
⋮	⋮	⋮
10	[1 to N]	N

$N+N+N+\ldots+N$

$N = 10N$

$\boxed{T = O(N)}$

Ex

```
for (i=1; i<=N; i++)
{
   for(j=1; j<=N; j = j*2)   } j/2 OR j*2 → log N
   {
      ⋮
   }
```

i	j	Iteration
1	[1..N]	$\log_2 N$
2	[1..N]	$\log_2 N$
3	[1..N]	$\log_2 N$
⋮	⋮	⋮
N	[1..N]	$\log_2 N$

$$T = O(N * \log_2 N)$$

Ex

```
for i = 1 to N
    for j = 1 to i
    {
    }
```

i	j	Iteration
1	[1..1]	1
2	[1..2]	2
3	[1..3]	3
⋮	⋮	⋮
N	[1..N]	N

$= 1 + 2 + 3 + 4 + \ldots + N$

$= \frac{N(N+1)}{2}$

$= \frac{N^2 + N}{2}$ ⇐ We have to take bigger value.

$\boxed{T = O(N^2)}$

Ex
```
for i=1 to N
    for j=1 to 2^i
    {
    ....
    }
```

i	j	Iteration
1	[1,2]	$2 = 2^1$
2	[1,4]	$4 = 2^2$
3	[1,8]	$8 = 2^3$
⋮	⋮	⋮
N		2^N

Geometric Progression :- Ratio is same between consecutive terms.

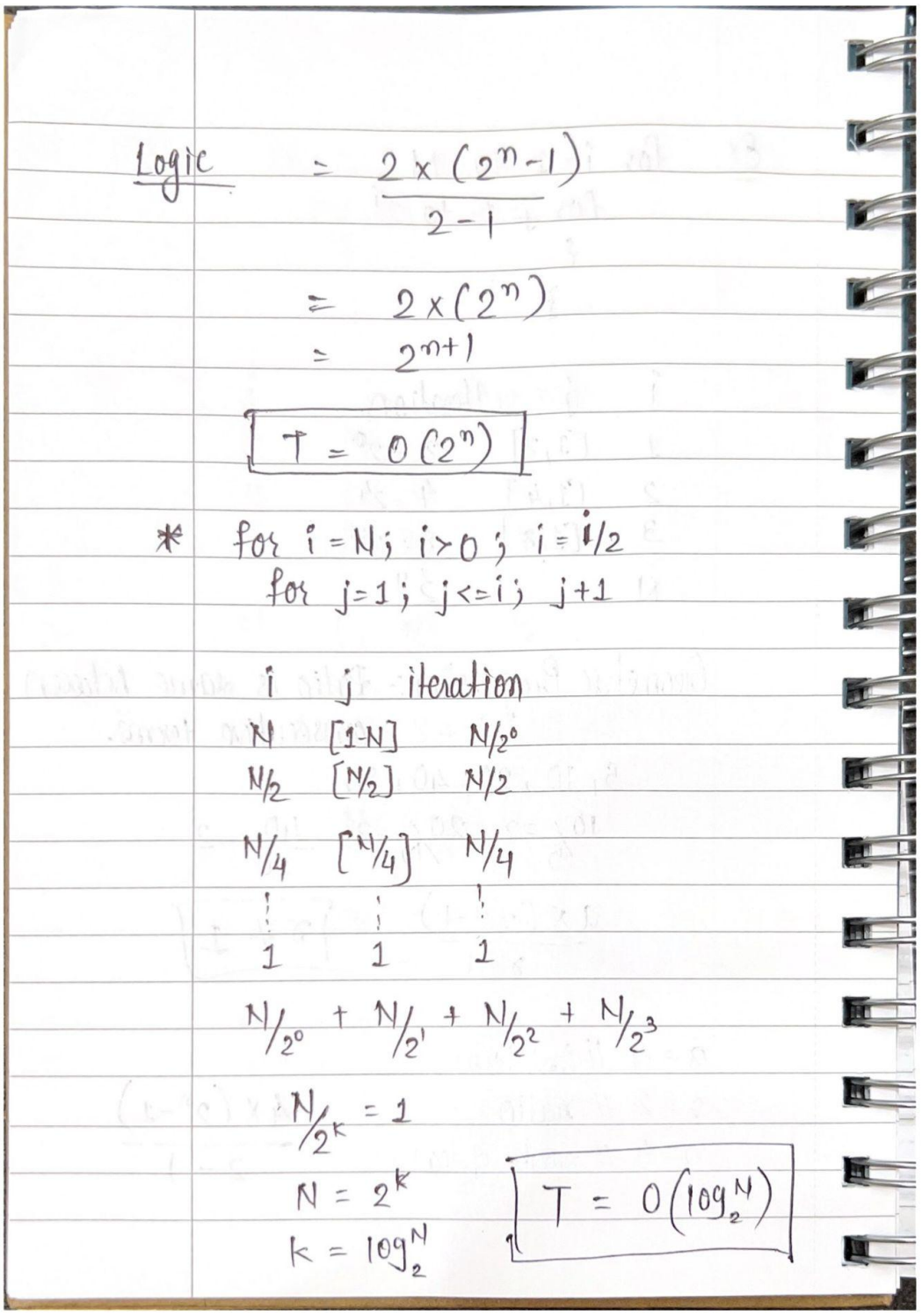
5, 10, 20, 40, 80

$\frac{10}{5} = 2 \quad \frac{20}{10} = 2 \quad \frac{40}{20} = 2$

$\frac{a \times (r^n - 1)}{r - 1}$ $\boxed{r \neq 1}$

$a = 4$ // first term
$r = 2$ // ratio
$n = 4$ // number of term

$\frac{4 \times (2^n - 1)}{2 - 1}$



CASE STUDY

```
# To count the factorial of describe number
a = 25
count = 0
for i in range(1, a+1):
  if a%i==0:
    count+=1
print(count)


# To count the factorial of describe number example by mate
def countOfFactors(num):
   count = 1
   listOfNumbers = list(range(1, num))
   for item in listOfNumbers:
      if num%item == 0:
         count += 1

   return count


# To count the common factorial of describe number
a = 25
count = 0
for i in range(1, int(a ** 0.5)+1):
  if a%i==0:
```

```
        if a == i*i:
            count+=1
        else:
            count+=2

print(count)
```

```
# Sum of n natural number
n = 500
sum = 0
for num in range(0, n+1, 1):
    sum = sum+num
print(sum )

# append example
results = []
x = range (50,100,10)
for i in x:
    results.append(i)

print(results)
```

02/03/2023

Array:- An array is defined as a collection of items that are stored at contiguous memory locations. It is a container which can hold a fixed number of items, and these items should be of the same type.

The array can be handled in Python by a module named array. It is useful when we have to manipulate only specific data values. Following are the terms to understand the concept of an array:

- Element - Each item stored in an array is called an element.
- Index - The location of an element in an array has a numerical index, which is used to identify the position of the element.

Array Representation
An array can be declared in various ways and different languages. The important points that should be considered are as follows:

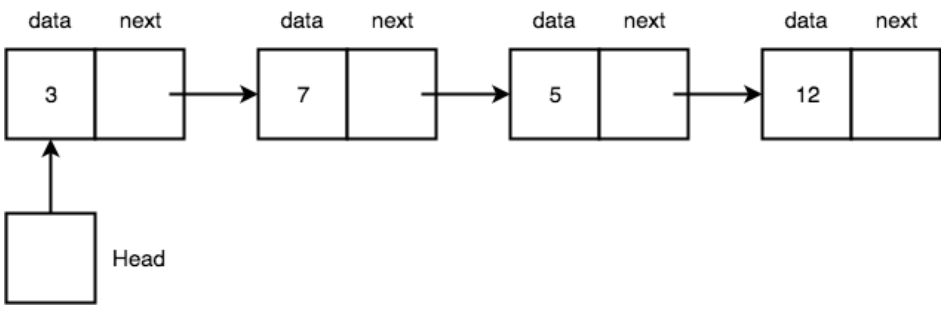
- Index starts with 0.
- We can access each element via its index.
- The length of the array defines the capacity to store the elements.

Array operations
Some of the basic operations supported by an array are as follows:

Traverse - It prints all the elements one by one.
Insertion - It adds an element at the given index.
Deletion - It deletes an element at the given index.
Search - It searches an element using the given index or by the value.
Update - It updates an element at the given index.

#%% :- To debug the particular cell in python file

Linked List:- The linked list is a linear data structure that stores the address of the next element. Each element is known as a node, and a node consists of data and the memory address of the next element. We can access the linked list's element through the pointer, and the first node is known as the head.



Stack:- A stack is a linear data structure where data is arranged objects on over another. It stores the data in LIFO (Last in First Out) manner. The data is stored in a similar order as plates are arranged one above another in the kitchen. The simple example of a stack is the Undo feature in the editor. The Undo feature works on the last event that we have done.

- We can perform the two operations in the stack - PUSH and POP. The PUSH operation is when we add an element and the POP operation is when we remove an element from the stack.

Methods of Stack
Python provides the following methods that are commonly used with the stack.

- empty() - It returns true, if the stack is empty. The time complexity is O(1).
- size() - It returns the length of the stack. The time complexity is O(1).
- top() - This method returns an address of the last element of the stack. The time complexity is O(1).
- push(g) - This method adds the element 'g' at the end of the stack - The time complexity is O(1).
- pop() - This method removes the topmost element of the stack. The time complexity is O(1).

Queue:- A queue is a linear type of data structure used to store the data sequentially. The concept of queue is based on the FIFO, which means "First in First Out". It is also known as "first come first served". The queue has two ends, front and rear. The next element is inserted from the rear end and removed from the front end.

Operations in Python

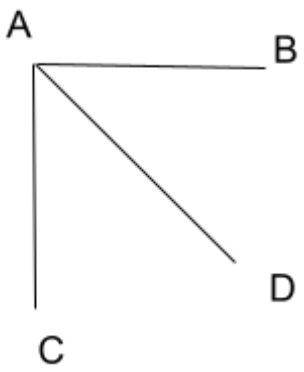
We can perform the following operations in the Queue.

- Enqueue - The enqueue is an operation where we add items to the queue. If the queue is full, it is a condition of the Queue The time complexity of enqueue is O(1).
- Dequeue - The dequeue is an operation where we remove an element from the queue. An element is removed in the same order as it is inserted. If the queue is empty, it is a condition of the Queue Underflow. The time complexity of dequeue is O(1).
- Front - An element is inserted in the front end. The time complexity of the front is O(1).
- Rear - An element is removed from the rear end.. The time complexity of the rear is O(1).

Methods Available in Queue
Python provides the following methods, which are commonly used to perform the operation in Queue.

- put(item) - This function is used to insert elements to the queue.
- get() - This function is used to extract the element from the queue.
- empty() - This function is used to check whether a queue is empty or not. It returns true if the queue is empty.
- qsize - This function returns the length of the queue.
- full() - If the queue is full returns true; otherwise false.

Graph:- A graph is a data structure you can use to model hierarchy and relationships between objects. It consists of a set of nodes and a set of edges. Nodes represent individual objects, while edges illustrate relationships between those objects.



Vertex	Edge
A	B;C;D
B	A
C	A
D	A

CASE STUDY

```
# How is Array working behind the scene

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):
                self.arrayItems[i+1]=self.arrayItems[i]
            self.arrayItems[position]=keyToInsert
        else:
            print("Size of array is ",self.sizeOfArray)
```

```
a = Array(5, int)
a.insert(6,2)
a.insert(7,4)

# Linked List algorithm

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 __str__(self) -> str:
        output = ''

        pointer = self.head
        while pointer is not None:
            output += str(pointer.value) + ' -> '
            pointer = pointer.next
        output += 'None'
        return output

a= LinkedList()
a.add_node(5)
a.add_node(7)
a.add_node(8)
print(a)
# Stack algorithm

class Stack:
    def __init__(self):
        self.items=[]

    def push(self, item):   #add element
        self.items.append(item)

    def pop(self):  # you can add check if loop if stack is empty "no popping" else return
        if self.is_empty():
            return 'Stack is empty'
        else:
            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(9)
my_stack.push(4)
```

```
my_stack.push(3)
print(my_stack.pop())


# Queue algorithm

class Queue():
    def __init__(self) :
        self.items= []

    def enqueue(self,item):
        self.items.append(item)

    def dequeue(self):
        if self.is_empty():
            return "Queue is empty"
        else:
            return self.items.pop(0)

    def is_empty(self):
        return len(self.items) == 0

    def sizeOfQueue(self):
        return len(self.items)
q =Queue()
#q.enqueue(3)
#q.enqueue(4)
#q.enqueue(5)
print(q.dequeue())

#Graph data structure

class Graph:
    def __init__(self):
        self.graph={}

    def add_vertex(self,vertex):
        if vertex not in self.graph:
            self.graph[vertex]=[]    #dictionar 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=[]
        for vertex in self.graph:
            for neighbors in self.graph[vertex]:
                edges.append((vertex,neighbors))
        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())
```

03/03/2023

Tree:- A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search.

- It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.
- The topmost node of the tree is called the root, and the nodes below it are called the child nodes.
- Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

Bubble Sorting.- The bubble sort uses a straightforward logic that works by repeating swapping the adjacent elements if they are not in the right order. It compares one pair at a time and swaps if the first element is greater than the second element; otherwise, move further to the next pair of elements for comparison.

Merge Sorting:- In the merge sort, we will be given an unsorted list of 'n' numbers divided into sub-lists until each sub-list has only one element. So we will divide the list into n sublists having one element, which is sorted by itself. Now we will merge all the sub-lists to get sorted sublists and finally a sorted sub-list.

Logic: (Divide and Conquer) Rule

- We will divide the problem into multiple subproblems.
- We will solve the sub-problems by further dividing the sub-problems into atomic problems where they have an exact solution.
- Finally, we will combine all sub solutions to get the final solution to the given problem.

Quick Sorting:- Quick sort algorithm is an in-place sorting algorithm without the need of extra space or auxiliary array as all operations will be done in the same list, as we divided the given list into three parts as pivot element, elements less than pivot as a one sub-list and elements greater than pivot as another sub-list. This is also called a partition-exchange sort. The quicksort is a better sorting algorithm and, in most programming languages, available as a built-in sorting algorithm.

Insertion Sorting:- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

CASE STUDY

```
# Tree data structure algorithm
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 value == self.data:
```

```
            return str(value) +" is found in BST"
        elif value < self.data:
            if self.leftChild:
                return self.leftChild.search(value)
            else:
                return str(value) +" is not found in BST"
        else:
            if self.rightChild:
                return self.rightChild.search(value)
            else:
                return str(value)+" is not found in BST"

root = Node(27)

root.insert(12)
root.insert(45)
root.insert(5)
root.insert(16)
root.insert(99)
root.printTree()
print(root.search(12))
print(root.search(3))


# 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]

sort = [5,-8,6,7,-2]
#sort = ["mia", "anna", "kristina", "mike"]
#sort = ['c', 'b', 'd', 'a']

bubble_sort(sort)
print(sort)


# Merge Sorting
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:]

#Recursively 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([58,14,45,11,1])


#Quick Sorting
def quicksort(arr):
    if len(arr) < 2:
        return arr

    pivot = arr[0]    #choose the pivot element

    #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([58,14,45,11,1])
```

```
# insertion sorting
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([10, 6, 8, 2, 11])
```

06/03/2023

OOP- Object Oriented Programming

Class:- A Class is a blueprint for an object.

- Class as attribute and behavior
  Human-> Class
  Attribute -> age, gender, height, weight
  Behavior-> run, walk, sit, sleep, sing, dance

Syntax:-
Class Class_Name:

Object:- An Object is an instance of class.

- Object is an entity that has state and behavior.

Syntax:-
obj = Class_Name()

__init__ :- In python __init__ function acts like Constructor.

Constructor:-
Then, we have created the constructor. A constructor is a special method in a class that the python always calls when we create or instantiate an object. Python uses “__init__()” to create a constructor.

The self parameter:-
The __init__() method can take parameters but the first parameter should always be the “self” parameter which refers to the current instance of the class. You can use any name instead of self but it should be the first parameter always.

Instance Attribute:-
Variables that we create inside a constructor are called instance attributes. These attributes are different for different instances.

Methods
Methods are the functions that are defined inside the body of a class and associated with an object. They are used to define the behaviors of an object.

Example 1:-

```
# Class
class Book:

    def __init__(self, title):
        self.title = title

    # if we need to access attribute we need get method
    def getTitle(self):  #getter method
        return self.title

    def setTitle(self,title): #setter method
        self.title = title

# TODO create book instance variable
b1 = Book("Title1")
print(b1.getTitle())
b1.setTitle("New Title")
print(b1.getTitle())
b2 = Book("Title2")
print(b2.getTitle())
```

Example 2:-

```
# Class
class Book:

    def __init__(self, title,pages,author,price,secret): # with secret parameter
        self.title = title
        self.pages = pages
        self.author = author
        self.price = price
```

```
    # self.__secret = "This is the book"  #This the one way to declare without passing parameter
    self.__secret = secret + "has" + author # using secret parameter (It is bad practice to use __secret)

def __str__(self):
    #return f"{self.title} by {self.pages} {self.price}"
    print("the object with name %s has: \ntitle: %s \nprice: %d" %(self.__name__, self.title, self.price))

def __reprr__(self):
    return f"{self.title} by {self.author} has {self.pages} pages and the cost is {self.price}"

def __repr__(self):
    return f"{self.__class__.__name__}({self.title}, {self.pages}, {self.author}, {self.price}, {self.__secret})"

def display(self):
   print("title %s\n pages %s \n author %s \n price %d \n " %(self.title,self.pages,self.author,self.price))

#Optional attribute
def setDiscount(self,amount):
    self._discount = amount # if we don't want an attribute to be defined as default in init method we can use _variablename for that

def getPrice(self):
        if hasattr(self, "_discount"):
            return self.price - (self.price * self._discount)
        else:
            return self.price

    # if we need to access attribute we need get method
    def getTitle(self):  #getter method
        return self.title

    def setTitle(self,title): #setter method
        self.title = title

    def __eq__(self,value):
        if not isinstance(value,Book):
            raise ValueError("Can\'t compare a non book object to a book object")
        else:
            return (self.title == value.title and self.author == value.author)

class Car:
    def __init__(self,name):
        self.name= name

# TODO create book instance variable
b1 = Book("Title1", 156, "Anna", 150, "Book")
#print(b1.getTitle())
#b1.setTitle("New Title")
#print(b1.getTitle())
#print(b1.getPrice())
#b1.setDiscount(0.05)
#print(b1.getPrice())
b2 = Book("Title2", 200, "Daina", 250, "Books")
b3 = Book("Title1", 156, "Anna", 250, "Book")
c1 = Car("BMW")
print("Comparison is ", b1 == b3)
#print(b2.getTitle())
#print(b2.getPrice())
#print(b1._Book__secret)

# deleting the attribute from instance variable
# del b1.price

#deleting the object itself
# del b1

#b2.display()  # To display variable name and value
```

```
#print(str(b2))  # to display values using __str__ function

print(repr(b2))
```

TypeCheck:-

Example:-

```
class Book:
    def __init__(self, title):
        self.title=title

class NewsPaper:
    def __init__(self,name):
        self.name=name

b1= Book("The Catcher in Rye")
b2=Book("The Grapes of Wrath")
n1 =NewsPaper("The Washington Post")
n2= NewsPaper("The New York times")

print(type(b1))
print(type(b2))

print(type(b1) == type(b2))
print(type(n1) == type(n2))

print(isinstance(b1,Book))
print(isinstance(n1,NewsPaper))
print(isinstance(n2,object))
```

07/03/2023

Inheritance:- Inheritance allows us to define a class that inherits all the methods and properties from another class.

- Parent class is the class being inherited from, also called base class.
- Child class is the class that inherits from another class, also called derived class.

Multiple Inheritance:- A class can be derived from more than one superclass in Python. This is called multiple inheritance.

Multilevel Inheritance:- In Python, not only can we derive a class from the superclass but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.

Method Resolution Order (MRO):- If two superclasses have the same method name and the derived class calls that method, Python uses the MRO to search for the right method to call.

Using Super():- Python super() function provides us the facility to refer to the parent class explicitly. It is basically useful where we have to call superclass functions. It returns the proxy object that allows us to refer to the parent class by 'super'.

Example:-

```
class Publication:
    def __init__(self,title,price):
        self.title=title
        self.price=price

class Periodical(Publication):  # Inheriting Publication class
    def __init__(self,title,publisher,period,price):
        super(). __init__(title,price)
        self.period = period
        self.publisher = publisher

class Book(Publication):   # Inheriting Publication class
    def __init__(self,title,price,author,pages):
        super() .__init__(title,price)
        self.author = author
        self.pages = pages

class Magazine(Periodical):  # Inheriting Periodical class
    def __init__(self, title, publisher, period, price):
        super().__init__(title, publisher, period, price)

class NewsPaper(Periodical):  # Inheriting Periodical class
    def __init__(self, title, publisher, period, price):
        super().__init__(title, publisher, period, price)

b1= Book("Dark", 12, "sadhkjs",12)
m1= Magazine("123sa","132sad","Monthly",200)
n1= NewsPaper("123sa","132sad","Monthly",200)
print(issubclass(Magazine,Publication))    #like transitivity property
print(issubclass(Magazine,object))
print(isinstance(m1,object))

print (b1.__dict__)
print (m1.__dict__)
print (n1.__dict__)
```

Example 2:- Multiple Inheritance example.

```
class A:
    def __init__(self):
        super().__init__()
        self.foo="foo"
        self.name= "class A"

class B:
    def __init__(self):
        super().__init__()
```

```
        self.bar="bar"
        self.name="class B"

class C(B,A):    # inheriting B, A classes
    def __init__(self):
        super().__init__()

    def showprops(self):
        print(self.foo)
        print(self.bar)
        print(self.name)

c = C()
c.showprops()
```

Static Method:- A static method is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variables because this static method doesn't take any parameters like self and cls.

- A static method doesn't have access to the class and instance variables because it does not receive an implicit first argument like self and cls. Therefore it cannot modify the state of the object or class.

Example:-

```
class Employee:
    COMPANY_NAME = "Wiley"
    def __init__(self, name,age,id):
        self.name = name
        self.age = age
        self.id = id

    @classmethod
    def setTitle(self,newTitle):
        self.title=newTitle

e1 = Employee("Aneri", 25, 132)
print(e1.COMPANY_NAME)
print(e1.__dict__)  # dictionary of instance variable
print(Employee.__dict__)
```

Example 2:-

```
class Book:
    # define 3 count variable and increment it in instance method ,
    # static method and class method and print it

#TODO properties defined at class level shared by all instance variables
__COUNT=0                #STATIC and constant
BOOK_TYPES=("HARDCOVER","PAPERBACK","EBOOK")      # STATIC constants
BOOK_CLASSLEVEL_COUNT=0     # static constants

#TODO __ properties are hidden from other classes
__booklist = None
def __init__(self,title,bookType):
    self.title=title
    self.instnce_count=0
    if (not bookType in Book.BOOK_TYPES):
        raise ValueError(f"We don't support the feature you are asking {bookType}")
    else:
        self.booktype=bookType

@staticmethod
def incrementCount():    #These is not associated with b1 or b2 its associated with class name Book
    Book.__COUNT = Book.__COUNT+1
    return Book.__COUNT
```

```
# static methods do not receive class or instance argument
@staticmethod
def getBookList(arg1,arg2):    # these method is to set something at class level

  #  print(Book.BOOK_TYPES)
    if Book.__booklist == None:
        Book.__booklist = []
    return Book.__booklist

@classmethod      # same like setter and getter
def getBookTypes(cls):
    return cls.BOOK_TYPES


@classmethod
def returnCount(cls):
    return cls.__COUNT

def setTitle(self,newTitle):
    self.title=newTitle

  def setCount(self):       #  it's associated with that b1 instance variable is not associated Book
    self.count=self.count+1


print("Book Types : ", Book.getBookTypes())
b1 = Book("title1","HARDCOVER")
b1.setCount()
b2= Book("title2","PAPERBACK")
b2.setCount()
theBook=Book.getBookList(1,2)
theBook.append(b1)
theBook.append(b2)
print(theBook)

Book.incrementCount()
print(b1.count)
Book.incrementCount()
print(b2.count)
Book.incrementCount()
Book.incrementCount()
print(b1.returnCount())
print(b1.incrementCount())
```

Abstraction:- Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. Users are familiar with "what function does" but they don't know "how it does."

Abstract Base Classes(abc):-
An abstract base class is the common application program of the interface for a set of subclasses. It can be used by the third-party, which will provide the implementations such as with plugins. It is also beneficial when we work with the large code-base hard to remember all the classes.

Syntax:-

```
from abc import ABC
class ClassName(ABC):
```

Example:-

```
from abc import ABC,abstractmethod
#abc is a Abstract Base Class
class GraphicsShape(ABC):
    def __init__(self):
        super().__init__()

    @abstractmethod
```

```
    def calcArea(self):
        pass

class Circle(GraphicsShape):
    def __init__(self,radius):
        super().__init__()
        self.radius = radius

    def calcArea(self):
        return 3.14 *(self.radius ** 2)

c = Circle(10)
print(c.calcArea)
```

08/03/2023

Composition.-Composition is the way of combining two or more functions in such a way that the output of one function becomes the input of the second function and so on.

Example:-
#Author class to use it in different file

```
class Author:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def __str__(self):
        return f"{self.fname} {self.lname}"

# Importing Author function from the folder and file name author

from author import Author
class Book:
    def __init__(self,title,price,author=None):
        self.title = title
        self.price = price
        self.author = author

        self.chapter=[]
    def add_chapter(self,chapterName,pages):
        self.chapter.append((chapterName,pages))

a1= Author("Ray", "Bay")
b1= Book("Dreams", 200, a1)
b2= Book("Joy", 500, a1)
b1.add_chapter("abc",100)
b1.add_chapter("gbc",100)

print(b1.title)
print(a1.fname, a1.lname)
```

JSONIFY Example:-

```
# JSONify example  using abstract method
from abc import ABC,abstractmethod

#here abc is abstract base class
class GraphicShape(ABC):
    def __init__(self):
        super().__init__()

    #a method with an empty body , if you don't know the logic you declare method
    # abstract and the class inherit the abstract method would have to implement it
    @abstractmethod
    def calcArea(self):
        pass

# JSON example
class JSONify(ABC):
    @abstractmethod
    def toJSON(self):
        pass

class Circle(GraphicShape,JSONify):
    def __init__(self,radius):
        super().__init__()
        self.radius =radius

    def calcArea(self):
```

```
        return 3.14 * (self.radius ** 2)

    def toJSON(self):
        return f"{{\"Circle\" :{str(self.calcArea())} }}"

class Square(GraphicShape,JSONify):
    def __init__(self,side):
        super().__init__()
        self.side =side

    def calcArea(self):
        return (self.side ** 2)

    def toJSON(self):
        return f"{{\"Square\" :{str(self.calcArea())} }}"

c =Circle(10)
s =Square(8)
print(c.calcArea())
print(s.calcArea())
print(c.toJSON())
print(s.toJSON())
```

Encapsulation:- Wrapping up data under a single unit.

- Prevent from direct modification.

Example:-

```
#Encapsulation
class Logincredentials:
    def __init__(self):
        self.username = "Aneri"
        self.__password = "abcd123"

    def newpass(self):
        print(format(self.__password))

    def setnewpassword(self, newpassword):
        self.__password = newpassword

lc = Logincredentials()
print(lc.username)
lc.newpass()

lc.__password="abcde123"
lc.newpass()
lc.setnewpassword("aneri123")
lc.newpass()
```

Polymorphism:- Polymorphism means the ability to take various forms.

- Polymorphism allows us to define methods in the child class with the same name as defined in their parent class.

Example:-

```
# Polymorphism
class Bikes:
    def __init__(self, name):
        self.name = name

    def max_speed(self):
        print('Bike maximum speed is 200')

    def seats(self):
        print('Bike has maximum two seats')

class Cars:
    def __init__(self, name):
        self.name = name
```

```
    def max_speed(self):
        print('Car maximum speed is 350')

    def seats(self):
        print('Car has maximum four seats')

def speciality(vehicle):
    vehicle.max_speed()
    vehicle.seats()

bike = Bikes("Ducati")
car = Cars("BMW")
speciality(bike)
speciality(car)
```

09/03/2023

__getattr__() method :- Is an object method that is called if the object's properties are not found. This method should return the property value or throw AttributeError.

__getattribute__() method:-This method is called unconditionally when accessing the properties of an object. This method only works for new classes.

__setattr__() method:- This method is used to assign the object attribute its value.

Example:-

```
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} {self.author} {self.price}"

    # getattribute method
    def __getattribute__(self, name: str):
        if(name == "price"):
            p= super().__getattribute__("price")
            d= super().__getattribute__("discount")
            return p-(p * d)
        return super().__getattribute__(name)

    # setattr method
    def __setattr__(self, name: str, value: str):
        if(name == "price"):
            if type(value) is not float:
                raise ValueError("value must be a float")
        return super().__setattr__(name, value)

    # getattr method
    def __getattr__(self, name):
        return  name + " is not a variable in class book."



b1 = Book("The Humans", "Ray", 200, 250.50)
b2 = Book("The World", "Bay", 400, 400.99)

b1.price = 99.99
print(b1.price)

b2.price = 199.89
print(b2.price)

print(b1.author)
```

__gt__:- Greater than magical method. Use to compare >, >= using this operator.
__lt__:- Less than magical method. Use to compare <, <= using this operator.
sort():- Sort is used to sort the given value in ascending and descending order.

Example:-

```
# greater than and less than function in python
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}"

    #  greater 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

    # less 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

b1 = Book("The Humans", "Ray", 200, 250.50)
b2 = Book("The World", "Bay", 400, 400.99)
b3 = Book("Moral", "Viv", 100, 55.55)
b4 = Book("Pride and Prejudice", "Jane Austen", 250, 66.99)

print(b1 == b3)

#first parameter is self and the second parameter is the value in __gt__ and __lt__
print(b2 >= b1)
print(b2 <= b1)
print(b2 < b1)
print(b4 >= b2)
print(b3 > b4)

books=[b1,b4,b3,b2]
books.sort(reverse=True)  # to print the value in descending order and if we need in ascending no need to mention reverse= true
print([book.title for book in books ])
```

Call():- The __call__ method enables Python programmers to write classes where the instances behave like functions and can be called like a function.

Example:-

```
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} {self.author} {self.price}"

    # call() function
    def __call__(self,title,author,pages,price):
        self.title=title
        self.author=author
        self.price=price
        self.pages=pages

b1 = Book("The Humans", "Ray", 200, 250.50)
b2 = Book("The World", "Bay", 400, 400.99)
print(b1)

b1 = (“ Humans", "Raymond", 600, 259.50)
b2 = ("World", "Baylive", 500, 460.99)
print(b1)
```

Decorator:- A Python decorator is a function that takes in a function and returns it by adding some functionality.

- Any object which implements the special __call__() method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.

Example 1 :- Example using importing datetime()

```
# Decorator
# i am passing a parameter which can be a variable or function but these time its a function
# what is the use :-
# deposit money or withdraw money you can have a function call audit

import datetime
def my_decorator(my_function):
    def inner_decorator():
        print("this happened before",datetime.datetime.utcnow())     # you add the balance
        my_function()    # when you call my function the my_decorated function get executed
        # here you do the audit function call
        print("this happened after",datetime.datetime.utcnow())   # ensure database is updated
        print("This happened at the end!!!!",datetime.datetime.utcnow())  # drop a message on console
    return inner_decorator

@my_decorator
def my_decorated():
    print("welcome to first lecture of class",datetime.datetime.utcnow())

# 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("first line of main",datetime.datetime.utcnow())
    my_decorated()
    print("after decorator",datetime.datetime.utcnow())
```

Example 2:-

```
def deco(func):
    def inner(*args, **kwargs):
        print("Doing", func.__name__)
        result = func(*args, **kwargs)
        print("Doing executed")
        return result
    return inner

@deco
def addition(a,b):
    print(a+b)

@deco
def subtraction(a,b):
    print(b-a)

if __name__ == "__main__":
    addition(2,4)
    subtraction(2,4)
```

Operator using:- Usage of __add__ and __sub__ function

Example:-

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)
    #these function is created by me

    def __add__(self, other):
```

```
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
    def __sub__(self,other):
        x = self.x - other.x
        y = self.y - other.y
        return Point(x, y)

    def multiply(self,other):
        x= self.x * other.x
        y = self.y * other.y
        return Point(x,y)

    def divide(self,other):
        x= self.x / other.x
        y= self.y / other.y
        return Point(x,y)

p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1-p2)
print(p1==p2)
print(p1.multiply(p2))
print(p2.divide(p1))
```

10/03/2023

Try….Except:- The try...except block is used to handle exceptions in Python. Here's the syntax of try...except block:

Syntax:-

```
try:
   # code that may cause exception
except:
   # code to run when exception occurs
```

Try….Finally:- In Python, the finally block is always executed no matter whether there is an exception or not.

- The finally block is optional. And, for each try block, there can be only one finally block.

Example:-

```
try :
   print(1/0)

except:
   print("Error occurred because you cannot divide a value by zero")

finally:
   print("abcdefghijklmnop")
   print("Hi")
```

Example:- Example using Exception

```
try :
   print(1/0)   # failure condition code can be written here

except Exception as ex:   # only when you get an error
   print("an error occurred",ex)

finally:  # these is always getting executed
   print("hi how are ayoposadjklasndksanknd")
   print("I am god")
```

Example:-

```
def handle_exception(func_name):
   def inner(a,b):
      try:
         return func_name(a,b)
      except Exception as ex:
         print('We have following exception', ex)
   return inner

@handle_exception
def divide(x,y):
   return x/y

divide(8,0)
```

Example:-

```
try :
   print(1/0)   # failure condition code can be written here
   raise RuntimeError("you are wrong")

except ZeroDivisionError as error:   # only when you get an error
   print("an error occurred",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 god")
```

Example:-

```
#Error can be anything insufficentfunderror or uncorrected error
class Customerror(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self) -> str:
        return f"{type(self).__name__}: {self.message} written by Aneri"
print("Welcome to program")
raise Customerror("Welcome to the first python custom error")
```

Example:- Example using “PASS”

```
# Error with pass
class Customerror(Exception):
    pass

print("Welcome to program")
raise Customerror("Welcome to your first python customerror")
```

CustomException:-we can define custom exceptions by creating a new class that is derived from the built-in Exception class.

Syntax:-

```
class CustomError(Exception):
    ...
    pass

try:
    ...

except CustomError:
    ...
```

Example:-

```
class CustomException(Exception):
    pass   #we can try using pass the format of the error message will be change
    """
    def __init__(self, message,code):
        self.message=message
        self.code=code
    def __str__(self) -> str:
        return f"{self.code} : {self.message}"
    """
print("hello")
raise CustomException("This is a custom exception message for class C339", 17362178)
print("heeeello")
```

13/03/2023

```
# MAtrix Example
#def make_matrix(n_rows,n_cols):
   #return [list(range(n_cols *i , n_cols*(i+1))) for i in range(n_rows)]
#test_data=make_matrix(40,5)
#test_data

def my_matrix(no_rows,no_cols):
   return [list(range(no_cols *i, no_cols*(i+2))) for i in range(no_rows)]
test_data=my_matrix(30,5)
Test_data

# To calculate the amount of time to execute the loop
%%timeit
sum([sum(r) for r in test_data])
```

Numpy:- NumPy is a Python library used for working with arrays.

Installation:- pip install numpy
NumPy is usually imported under the np alias.
Import numpy as np

Example:-

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

Example 2D Array:-

```
arr2d=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(arr2d)
```

Example:-

```
# array scalar getting one element from numpy array
print(arr[3])
print(arr2d[0][2])
print(arr2d[0,2])
```

Example:- To get shape and datatype of the array

```
print(arr.shape)    # get the size of row
print(arr2d.shape)   # get the size of row and columns
print(arr.dtype)   # get the data type
```

Example:- to get even number or multiple of 2
these is for multiplying 2 with the array elements

```
arr_squared = arr *2
print(arr_squared)
```

Example:- Sum of the array using sum keyword
these code is for calculating array sum

```
arr_Sum=np.sum(arr)
print(arr_Sum)
```

Transpose:-

Example:- Transpose of the matrix using “transpose” keyword
these code is for doing numpy transpose of matrix

```
arr2d_transpose =np.transpose(arr2d)
print(arr2d_transpose)
```

Example:- example using complex number
#complex number formula:-(x+yi)

```
acomplex=np.array([1,2,3],dtype="complex")
print(acomplex)
```

Linspace:-

Example:- linspace example

linspace here 0= starting element, 12= ending element, 4= number of element
array=np.linspace(0,12,4,dtype=int)
print(array)
Example:-
here endpoint is not included
array=np.arange(0,11,2)
print(array)

setting random set for consistency
np.random.seed(0)
array_random=np.random.rand(5,2)
print(array_random)

Array Math :- Basic mathematical functions operate elementwise on arrays.

x = np.array([[2,6],[5,4]], dtype=np.float64)
y = np.array([[3,9],[1,8]], dtype=np.float64)

1. Add
   Example:- Sum of the given array
   print(x + y)
   print(np.add(x, y))

2. Subtract
   Example:- difference of the given array
   print(x - y)
   print(np.subtract(x, y))

3. Square root
   Example:- square root of the given array
   np.sqrt(x)

4. log
   Example:- Logarithm of the given array
   np.log(x)
   np.log10(x)

5. Sin/Cos
   Example:- To get a sin or cos value
   np.sin(data)

Boolean Array:-Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

Example:-
Boolean Array
#np.ones((3,3), dtype ="bool")
np.full((3,3),True, dtype = "bool")

Example:- Extract all odd numbers
given_arrays =np.array([0,1,2,3,4,5,6,7,8,9,10])
given_arrays[given_arrays % 2 == 1]

Min & Max Keyword:- It is used to get the minimum and maximum value from the given array

Example:- To find the minimum and maximum value
so here min and max keyword are used to get minimum and maximum value from the array
sample= np.array([1,2,3,4,5,6,0,7,8,9])
print("max:", np.max(sample))
print("min:", np.min(sample))
print(sample.argmin())
print(sample.argmax())
print("mean:", np.mean(sample))

Example:-

#most functions have a version to ignore nan:
np.nansum(sample)
np.nanmin(sample)
np.nanmean(sample)

Example:-
Converts the given data into the string datatype we enter the data into the tuple
```
import numpy as np
a = np.array([('abc',12,10),("def",7,79)])
print(a)
```

Reshape:- By reshaping we can add or remove dimensions or change the number of elements in each dimension.

Example:-
convert 2D array into 3D array
```
a = np.array([[1,2,3],[4,5,6]])
b = a.reshape(3,2)
print(b)
```
To convert it into a single array
```
b= a.reshape(-1)
print(b)
```

Example:-
saving and retrieving custom object from npz file
```
import numpy as np
class Myclass1:
   def __init__(self,obj1,obj2):
      self.value=obj1
      self.value2=obj2

obj= Myclass(5,'Aneri')
obj2=Myclass(6, "Anna")
np.savez('aneri.npz',obj=obj,obj2=obj2)
load_object=np.load('aneri.npz',allow_pickle=True)
load_object=load_object['obj2'].item()
print(load_object.value2)
```

PANDAS:- Pandas is a Python library. It is used to analyze data.

- It has functions for analyzing, cleaning, exploring, and manipulating data.
- Pandas allows us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.

Installation:-

- **pip install pandas**
- Then to use pandas we need to import it in our file to do that use command
- Create an alias with the as keyword while importing:
- **import pandas as pd**

Pandas Series:- Series is like a column in a table, it is a one-dimensional array holding data of one type.
Example:-
```
import pandas as pd
s= pd.Series([4.5,7,1.3,9.34])
s
```
Example:-
```
s= pd.Series([5,7,8.6,5.7], index=['a', 'b', 'c', 'd'])
```

to get the specific value from the array
```
s['b']
```

to print the index
```
print(s.index)
```

to get the particular values from the array it will return the value of a,b,c
```
s['a':'c']
```

14/03/2023

Adding two arrays
Adding in pandas
x= pd.Series([1,2,3], index = ['a', 'b', 'c'])
y= pd.Series([2,1,4], index = ['b', 'a', 'd'])
x+y

to fill the values null values with 0
x.add(y,fill_value=0)

values method would give you numpy array
x.values

Subtracting two arrays
#Subtracting in pandas
x= pd.Series([1,2,3], index = ['a', 'b', 'c'])
y= pd.Series([2,1,4], index = ['b', 'a', 'd'])
x-y

won't written none
x.sub(y,fill_value=0)

DataFrames:- Datasets in Pandas are usually multi-dimensional tables, called DataFrames.

- Series is like a column, a DataFrame is the whole table.

Example:-
import numpy as np
df1 = pd.DataFrame({'customer': ['A', 'B', 'C','D'],
'balance': [12345, 131234, 12333413,1565663],
'age': [23,45,456,np.NaN]})
df1

Math
Example:-
Different NaNs though:
import math
all these nans work with pandas fine:
s_nan = pd.Series([float('nan'), math.nan, np.nan])
print(s_nan[0] == s_nan[1])
print(s_nan[1] == s_nan[2])
print(s_nan[0] == s_nan[2])

Example:- To Fetch the particular Column
to get a particular field
print(df1['balance'])

to get pandas datatype of the particular field
print(type(df1['age']))

#to get all datatypes
df1.dtypes

Load Files Into a DataFrame:- If your data sets are stored in a file, Pandas can load them into a DataFrame.

Example:- To read a ZIP file
import pandas as pd
df = pd.read_csv('college_scorecard_2017-18.zip',dtype={'OPEID':str})
df.head() #to display the demo or the head of the data sets

to get all the information of the data set

df.info()

Example:- To read a CSV file

```
import pandas as pd
df_excel= pd.read_csv("college_scorecard_2017-18.csv",dtype={'OPEID':str})
df_excel.head()
```

Example:- To read a url

```
url="https://en.wikipedia.org/w/index.php?title=World_population&oldid=948301297"
dfread=pd.read_html(url)
dfread[12].head() # dfread  to access all the detail of the url with the line number
```

Lambda Function

Example:-

```
# add using lambda
x= lambda a,b,c : a+b+c
result = x(2,3,5)
print(result)
```

Example:-

```
#use of sum keyword using lambda for the n number using args
y= (lambda *args: sum(args))(5,2,3,6)
print(y)
```

Example:-

```
# operator
print((lambda x : x if(x>20) else 20)(5))
```

Example**:-**

```
lst=[33,5,65,7,8,55,99]
b= tuple(filter(lambda x: x>10,lst))  # we can print the value as tuple
b= list(filter(lambda x: x>10,lst))  # we can print the value as list as well
```

Example

```
#sorting
sorted(filter(lambda x : x>10, lst))
```

Map

Example:-

```
tpl=tuple(map(lambda x : x*10, xyz))
tpl
```

Example:-

```
import pandas as pd
df = pd.DataFrame({'col1':[1,2,3,4,5],'col2':[0,0,0,0,0]})
print(df)
df['col3']=df['col1'].map(lambda x :x*10)
df
```

```
# to display tables in better way in jupyter
display(df)
```

Apply

```
# using apply keyword instead of map
df['col4'] = df['col1'].apply(lambda x :x*10)
df
```

Example:-

```
df['col5'] = df['col3'].map(lambda x : 30 if x < 30 else x)
df
```

Reduce:-

To use reduce function we need to import it from functools

- **from functools import reduce**

Example:-

```
from functools import reduce
list= [1,2,3,4,5]
b=reduce(lambda x,y : x+y, list)
result = b
print(result)
```

Example:-

```
#to create dictionary
create_dict = lambda **kwargs:kwargs
print(create_dict(a=1,b=2,c=3, d=4))
```

Example:-

```
multiply = lambda x , *args, **kwargs: x * sum(args) * kwargs['multiplier']
print(multiply(2,1,2,3,multiplier =3))
```

Example:-

```
# filter example to get smaller numbers, even numbers & odd numbers
list1 = [8, 2, 6, 4, 3, 1]
xy = filter(lambda x: x<8, list1)
print("smaller than 8:-", tuple(xy))
#even elements
even = filter(lambda x: x%2==0, list1)
print("even:-", tuple(even))
#odd elements
odd = filter(lambda x: x%2, list1)
print("odd:-", tuple(odd))
```

Example:-

```
# Celsius to Fahrenheit Conversion using map
town = [('Munich',8),("Hamburg",10),("Berlin",15), ("Köln",18)]

fahrenheit = tuple(map(lambda c: (c[0],  (float(9/5) * c[1]+ 32)), town))
print(fahrenheit)

# Fahrenheit to Celsius Conversion using map
Celsius = tuple(map(lambda f: (f[0], (float(5)/9) * (f[1]-32)), fahrenheit))
print(Celsius)
```

15/03/2023

Here I am reading a CSV file and from the data set I will use different keywords and functions.

Example:-

```
college= pd.read_csv("college_scorecard_2017-18.csv",dtype={'OPEID':str})
college.loc[:,'CITY']

# selecting all the data which is having CITY name as YUMA
college.loc[college.CITY == 'Yuma']

# selecting all the data which is having CITY name as YUMA and COSTT4_A >13000
college.loc[(college.CITY == 'Yuma') & (college.COSTT4_A > 13000)]

#selecting all the .COSTT4_A null values
college.loc[college.COSTT4_A.isnull()]

# describe all details about CITY
college.CITY.describe()

# unique keyword to get all unique city
college.CITY.unique()

# loc
college.loc[2,'INSTNM']
college.loc[:2,'INSTNM':'ZIP']
college.loc[3:10]
college.loc[[3,10],['CITY', 'LOCALE']]
college.loc[college['CITY']=='Normal']

# iloc
college.iloc[2,2]
college.iloc[:2,2:6]
college.iloc[-5:] #It will give last five records

# to start the index value from 1
new_index=pd.RangeIndex(1,len(college)+1)
college.set_index(new_index).head()

# to set the new index
college_city=college.set_index('CITY')
college_city.head()

#to drop the copy of the index
college = college.drop(['index'], axis=1)

# select all the city named Bloomington
college.set_index(['STABBR','CITY']).loc['MN',:].loc['Bloomington',:]

# rename the column name
college = college.rename(columns={'STABBR':'STATE'})
college

#access multi index row with one loc
college.set_index(["STATE","CITY"]).loc[("MN","Bloomington"),:]

# to check if the column is present or not
'Normal' in college.columns

# to drop the particular number of rows
college_new = college.drop([2,4])
college_new

# to remove the duplicate state, city (first, last)
college_new = college.drop_duplicates(subset=['STATE', 'CITY'], keep='first').head()
```

to read excel file
pip install openpyxl

```
import pandas as pd
df_econ = pd.read_excel("AGI_zipcode_2016.xlsx",dtype={'zipcode':str})
df_econ.head()
#To merge to columns from different data set
df_merged=df.merge(df_econ,left_on='ZIP',right_on='zipcode', how='left')
df_merged[['INSTNM','ZIP','AGI']].head(10)
```

#dropna
```
college.dropna(subset=['INSTURL']).count()
college= college.dropna(thresh=1)
```

Example using thresh:-

```
data = {'A': [1,2,None,4,None],'B':[None,6,None,None,None]}
df= pd.DataFrame(data)
df = df.dropna(thresh=1)
df
```

Datetime:-

Example:-
#Datetime
```
from datetime import datetime
dt = datetime(1995,12,31,11,55,40)
print(dt)
```

-Import parse from dateutil.parser

#with different format
```
from dateutil.parser import parse
dt= parse('Jan 26 , 1956  20:15:13')
print(dt)
```

#print the dates
```
print(parse('Jan 26 , 1956  20:15:13'))
print(parse('26/12/1956  20:15:13'))
print(parse('12/26/1956 20:15:13'))
```

Subtracting in datetime
```
delta_t=(datetime.now() - dt)
print(f'{delta_t} is the subtract timing')
print(type(delta_t))
```

using strftime
```
dt_str=dt.strftime('%m/%d/%y')
print(dt_str)
dt_str=dt.strftime('%b %d, %Y')
print(dt_str)
```

cumsum
```
import numpy as np
a = np.array([1,2,3,4,5])
cumulative_sum=np.cumsum(a)
print(cumulative_sum)
```

Example:-
```
import numpy as np
import pandas as pd
np.random.seed(50)
```

```
def random_walk(n_steps):
    step= 0.5 - np.random.random(n_steps)
    return np.cumsum(step)

date =pd.date_range('1/1/2019','12/31/2019',freq='D')
data = {'a':random_walk(365), 'b':random_walk(365)}
df=pd.DataFrame(data,index=date)
df.head(365)

#to get the type of index
type= (df.index)
print(type)
```