# Python Arrays

# Python Magic Methods

# Python Stack and Queue

# PySpark MLlib

# Python Decorator

# Python Generators

# Web Scraping Using Python

# Python JSON

# Python Itertools

# Python Multiprocessing

# Calculate Distance b/w Two Points using GEOPY

# Grid Search in Python

# Python High Order Function

# nsetools in Python

# Python OpenCV object detection

# Python SimpleImputer module

# Python OOPs Concepts

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# Python Arrays

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 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.

The Array can be created in Python by importing the array module to python program.

1. from array **import** \*
2. arrayName = array(typecode, [initializers])

**Accessing array elements**

We can access the array elements using the respective indices of those elements.

1. **import** array as arr
2. a = arr.array('i', [2, 4, 6, 8])
3. **print**("First element:", a[0]) //2
4. **print**("Second element:", a[1]) //4
5. **print**("Second last element:", a[-1]) //8

## How to change or add elements

Arrays are mutable, and their elements can be changed in a similar way like lists.

1. **import** array as arr
2. numbers = arr.array('i', [1, 2, 3, 5, 7, 10])
3. numbers[0] = 0
4. print(numbers) # Output: array('i', [0, 2, 3, 5, 7, 10])
5. numbers[2:5] = arr.array('i', [4, 6, 8])
6. print(numbers) # Output: array('i', [0, 2, 4, 6, 8, 10])

## How to delete elements from an array?

elements can be deleted from an array using Python's **del** statement.If we want to delete any value from the array, we can do that by using the indices of a particular element.

1. **import** array as arr
2. number = arr.array('i', [1, 2, 3, 3, 4])
3. del number[2] # removing third element
4. print(number) # Output: array('i', [1, 2, 3, 4])

## Array Concatenation

We can easily concatenate any two arrays using the + symbol.

1. a=arr.array('d',[1.1 , 2.1 ,3.1,2.6,7.8])
2. b=arr.array('d',[3.7,8.6])
3. c=arr.array('d')
4. c=a+b
5. print("Array c = ",c) //1.1, 2.1, 3.1, 2.6, 7.8, 3.7, 8.6]

# Python Magic Methods

To add "magic" to the class we create, we can define special methods called "magic methods." For example, the magic methods \_\_init\_\_ and \_\_str\_\_are always wrapped by double underscores from both sides. By granting us accessibility to Python's built-in

syntax tools, magic methods can improve the structure of our classes.

## \_\_init\_\_ Method

After we have constructed an instance of the class, but before that instance is returned to the caller of the class, the \_init\_ method is executed. When we create an instance of the class, it is called automatically,

1. **class** methods(): # Creating a class
2. **def** \_\_init\_\_(self, \*args):
3. **print** ("Now called \_\_init\_\_ magic method, after tha initialised parameters")
4. self.name = args[0]
5. self.std = args[1]
6. self.marks = args[2]
7. st = methods("Itika", 11, 98)
8. **print**(st)
9. **print**(f"Name, standard & marks is: \n", st.name, "\n", st.std, "\n", st.marks)

## \_\_new\_\_() Method

The magic method \_\_new\_\_() is called implicitly by the \_\_init\_\_() method. The new instance returned by the \_\_new\_\_() method is initialised. To modify the creation of objects in a user-defined class, we must supply a modified implementation of the \_\_new\_\_() magic method. We need to provide the first argument as the reference to the class whose object is to be created for this static function.

1. **class** Method(object):
2. **def** \_\_new\_\_( cls ):
3. **print**( "Creating an instance by \_\_new\_\_ method")
4. **return** super(Method, cls).\_\_new\_\_(cls)
5. **def** \_\_init\_\_( self ): # Calling the init method
6. **print**( "Init method is called here" )
7. Method()

## \_\_add\_\_ Method

We use the magic method \_\_add\_\_to add the class instance's attributes.

Consider the scenario where object1 belongs to class Method and object2 belongs to class Method 1, both of which have the same attribute called "attribute" that stores any value passed to the class while creating the instance.

If specified to add the attributes, the \_\_add\_\_ function implicitly adds the instances' same attributes, such as object1.attribute + object2.attribute, when the action object1 + object2 is completed.

1. **class** Method: # Creating a class
2. **def** \_\_init\_\_(self, argument):
3. self.attribute = argument
4. **class** Method\_2: # Creating a second class
5. **def** \_\_init\_\_(self, argument):
6. self.attribute = argument
7. instance\_1 = Method(" Attribute") # creating the instances
8. **print**(instance\_1.attribute)
9. instance\_2 = Method\_2(" 27")
10. **print**(instance\_2.attribute)
11. **print**(instance\_2.attribute + instance\_1.attribute) # Adding two attr of instances

O/P:- Attribute

27

27 Attribute

By using \_\_add\_\_ magic method the code changes to this.

## \_\_repr\_\_ Method

The class instance is represented as a string using the magic method \_\_repr\_\_. The \_\_repr\_\_ method, which produces a string in the output, is automatically called whenever we attempt to print an object of that class.

1. **class** Method:
2. **def** \_\_init\_\_(self, x, y, z):
3. self.x = x
4. self.y = y
5. self.z = z
6. **def** \_\_repr\_\_(self):
7. **return** f"values of the attr are Method:\nx = {self.x}\ny = {self.y}\nz = {self.z}"
8. instance = Method(4, 6, 2)
9. **print**(instance) O/P:- x = 4,y = 6,z = 2

## \_\_contains\_\_ Method

The 'in' membership operator of Python implicitly calls the \_\_contains\_\_ method. We can use the \_\_contains\_\_ method to determine if an element is contained in an object's attributes. We can use this method for attributes that are containers (lists, tuples, etc.).

1. **class** Method:
2. **def** \_\_init\_\_(self, attribute):
3. self.attribute = attribute
4. **def** \_\_contains\_\_(self, attribute):
5. **return** attribute **in** self.attribute
6. instance = Method([4, 6, 8, 9, 1, 6])
7. **print**("4 is contained in ""attribute"": ", 4 **in** instance) //True
8. **print**("5 is contained in ""attribute"": ", 5 **in** instance) //False

## \_\_call\_\_ Method

When a class instance is called, the Python interpreter calls the magic method \_\_call\_\_. We can utilise the \_\_call\_\_ method to explicitly call an operation using the instance name rather than creating an additional method to carry out specific activities.

1. **class** Method:
2. **def** \_\_init\_\_(self, a): # Calling the \_\_init\_\_ method and initializing the attributes
3. self.a = a
4. **def** \_\_call\_\_(self, number): # Calling the \_\_call\_\_ method to \* a num to attr val
5. **return** self.a \* number
6. instance = Method(7) # Creating an instance and proving the value to attribute a
7. **print**(instance.a) //7 # Printing the value of the attribute a
8. # Calling the instance while passing a value which will call the \_\_call\_\_ method
9. **print**(instance(5)) //35

## \_\_iter\_\_ Method

For the given instance, a generator object is supplied using the \_\_iter\_\_ method. To benefit from the \_\_iter\_\_ method, we can leverage the iter() and next() methods.

1. **class** Method:
2. **def** \_\_init\_\_(self, start\_value, stop\_value):
3. self.start = start\_value
4. self.stop = stop\_value
5. **def** \_\_iter\_\_(self):
6. **for** num **in** range(self.start, self.stop + 1):
7. **yield** num \*\* 2
8. instance = iter(Method(3, 8))
9. **print**( next(instance) )
10. **print**( next(instance) )

# Python Stack and Queue

## Stack

A Stack is a data structure that follows the LIFO(Last In First Out) principle. To implement a stack, we need two simple operations:

* **push -** It adds an element to the top of the stack.
* **pop -** It removes an element from the top of the stack.

### Methods of Stack

* **empty() -** It returns true, it 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).

**Operations:**

* **Adding -** It adds the items in the stack and increases the stack size. The addition takes place at the top of the stack.
* **Deletion -** It consists of two conditions, first, if no element is present in the stack, then underflow occurs in the stack, and second, if a stack contains some elements, then the topmost element gets removed. It reduces the stack size.
* **Traversing -** It involves visiting each element of the stack.

**Characteristics:**

* Insertion order of the stack is preserved.
* Useful for parsing the operations.
* Duplicacy is allowed.

1. x = ["Python", "C", "Android"] # stack using list
2. x.push("Java")
3. x.push("C++")
4. **print**(x) //['Python', 'C', 'Android', 'Java', 'C++']
5. **print**(x.pop()) //C++
6. **print**(x) //['Python', 'C', 'Android', 'Java']
7. **print**(x.pop()) //Java
8. **print**(x) //['Python', 'C', 'Android']

## Queue

A Queue follows the First-in-First-Out (FIFO) principle. It is opened from both the ends hence we can easily add elements to the back and can remove elements from the front.

To implement a queue, we need two simple operations:

* **enqueue -** It adds an element to the end of the queue.
* **dequeue -** It removes the element from the beginning of the queue.

**Operations on Queue**

* **Addition -** It adds the element in a queue and takes place at the rear end, i.e., at the back of the queue.
* **Deletion -** It consists of two conditions - If no element is present in the queue, Underflow occurs in the queue, or if a stack contains some elements then element present at the front gets deleted.
* **Traversing -** It involves to visit each element of 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 **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 **Queue Underflow**. Time complexity of dequeue is **O(1)**.
* **Front -** An element is inserted in front end. The time complexity of front is **O(1)**.
* **Rear -** An element is removed from the rear end.. Time complexity of rear is **O(1)**.

**Characteristics**

* Insertion order of the queue is preserved.
* Duplicacy is allowed.
* Useful for parsing CPU task operations.

1. **import** queue
2. L = queue.Queue(maxsize=10) # Queue is created as an object 'L'
3. L.put(9) # Data is inserted in 'L' at the end using put()
4. L.put(6)
5. **print**(L.get()) //9 # get() takes data from the head of the Queue
6. **print**(L.get()) //6

## Methods Available in Queue

* **put(item) -** This function is used to insert element 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 queue is empty.
* **qsize -** This function returns the length of the queue.
* **full() -** If the queue is full returns true; otherwise false.

## Working With collection.deque Class

The **collection.deque** class is used to implement a double-ended queue that supports adding and removing element from both ends. It takes O(1) time to complete process.

The **deque** class can be used in both Queue and as stacks because it removes and adds elements effectively.

The **collection.deque** can be a good choice for queue data structure in Python's standard library.

# PySpark MLlib

Machine Learning is a technique of data analysis that combines data with statistical tools to predict the output. This prediction is used by the various corporate industries to make a favorable decision.

PySpark provides an API to work with the Machine learning called as **mllib**. PySpark's mllib supports various machine learning algorithms like classification, regression clustering, collaborative filtering, and dimensionality reduction as well as underlying optimization primitives. Various machine learning concepts are given below:

* **classification**

The **pyspark.mllib** library supports several classification methods such as binary classification, multiclass classification, and regression analysis. The object may belong to a different class. The objective of classification is to differentiate the data based on the information. **Random Forest, Naive Bayes, Decision Tree** are the most useful algorithms in classification.

* **clustering**

Clustering is an unsupervised machine learning problem. It is used when you do not know how to classify the data; we require the algorithm to find patterns and classify the data accordingly. The popular clustering algorithms are the **K-means clustering, Gaussian mixture model, Hierarchical clustering.**

* **fpm**

The fpm means frequent pattern matching, which is used for mining various items, itemsets, subsequences, or other substructure. It is mostly used in large-scale datasets.

* **linalg**

The **mllib.linalg** utilities are used for linear algebra.

* **recommendation**

It is used to define the relevant data for making a recommendation. It is capable of predicting future preference and recommending the top items. For example, Online entertainment platform **Netflix** has a huge collection of movies, and sometimes people face difficulty in selecting the favorite items. This is the field where the recommendation plays an important role.

* **mllib regression**

The regression is used to find the relationship and dependencies between variables. It finds the correlation between each feature of data and predicts the future values.

The mllib package supports many other algorithms, classes, and functions. Here we will understand the basic concept of **pyspak.mllib**.

## MLlib Features

The **PySpark mllib** is useful for iterative algorithms. The features are the following:

* **Extraction:** It extracts features from "row" data.
* **Transformation:** It is used for scaling, converting, or modifying features.
* **Selection:** Selecting a useful subset from a larger set of features.
* **Locality Sensitive Hashing:** It combines aspects of feature transformation with other algorithms.

Let's have a look at the essential libraries of PySpark MLlib.

### MLlib Linear Regression

Linear regression is used to find the relationship and dependencies between variables. Consider the following code:

1. frompyspark.sql **import** SparkSession
2. spark = SparkSession.builder.appName('Customer').getOrCreate()
3. frompyspark.ml.regression **import** LinearRegression
4. dataset = spark.read.csv(r'C:\Users\DEVANSH SHARMA\Ecommerce-Customers.csv')
5. dataset.show(10)

In the following code, we are importing the **VectorAssembler** library to create a new column Independent feature:

1. frompyspark.ml.linalg **import** Vectors
2. frompyspark.ml.feature **import** VectorAssembler
3. featureassembler = VectorAssembler(inputCols = ["Avg Session Length","Time on App","Time on Website"],outputCol = "Independent Features")
4. output = featureassembler.transform(dataset)
5. output.show()

## MLlib K- Mean Cluster

The K- Mean cluster algorithm is one of the most popular and commonly used algorithms. It is used to cluster the data points into a predefined number of clusters.

### Parameters of PySpark MLlib

The few important parameters of **PySpark MLlib** are given below:

* **Ratings**

It is RDD of Ratings or (userID, productID, rating) tuple.

* **Rank**

It represents Rank of the computed feature matrices (number of features).

* **Iterations**

It represents the number of iterations of ALS. (default: 5)

* **Lambda**

It is the Regularization parameter. (default : 0.01)

* **Blocks**

It is used to parallelize the computation of some number of blocks.

### Collaborative Filtering (mllib.recommendation)

Collaborative filtering is a technique that is generally used for a recommender system. This technique is focused on filling the missing entries of a user-item. Association matrix **spark.ml** currently supports model-based collaborative filtering. In collaborative filtering, users and products are described by a small set of hidden factors that can be used to predict missing entries.

### Scaling of the regularization parameter

The regularization parameter **regParam** is scaled to solve least-squares problem. The least-square problem occurs when the number of ratings are user-generated in updating user factors, or the number of ratings the product received in updating product factors.

### Cold-start strategy

The **ALS Model (Alternative Least Square Model)** is used for prediction while making a common prediction problem. The problem encountered when user or items in the test dataset occurred that may not be present during training the model. It can occur in the two scenarios which are given below:

* In the prediction, the model is not trained for users and items that have no rating history (it is called a cold-start strategy).
* The data is splitted between training and evaluation sets during cross-validation. It is widespread to encounter users and items in the evaluation set that are not in the training set.

# Python Decorator

Decorators are one of the most helpful and powerful tools of Python. **These are used to modify the behavior of the function.** Decorators provide the flexibility to wrap another function to expand the working of wrapped function, without permanently modifying it.

**In Decorators, functions are passed as an argument into another function and then called inside the wrapper function.**

It is also called **meta programming** where a part of the program attempts to change another part of program at compile time.

## What are the functions in Python?

Python has the most interesting feature that everything is treated as an object even classes or any variable we define in Python is also assumed as an object. Functions are **first-class** objects in the Python because they can reference to, passed to a variable and returned from other functions as well. The example is given below:

1. def func1(msg):
2. print(msg)
3. func1("Hii")
4. func2 = func1
5. func2("Hii")

* The function can be referenced and passed to a variable and returned from other functions as well.
* The functions can be declared inside another function and passed as an argument to another function.

## Inner Function

Python provides the facility to define the function inside another function. These types of functions are called inner functions.

1. def func():
2. print("We are in first function")
3. def func1():
4. print("This is first child function")
5. def func2():
6. print(" This is second child function")
7. func1() //This is first child function
8. func2() //This is second child function
9. func() //We are in first function

In the above program, it doesn't matter how the child functions are declared. The execution of the child function makes effect on the output. These child functions are locally bounded with the **func()** so they cannot be called separately.

Function that accepts other function as a argument is also called **higher order function**.

1. def add(x):
2. **return** x+1
3. def sub(x):
4. **return** x-1
5. def operator(func, x):
6. temp = func(x)
7. **return** temp
8. print(operator(sub,10)) //9
9. print(operator(add,20)) //21

we have passed **sub()** function and **add()** function as argument in **operator()** function.

A function can return another function.

1. def hello():
2. def hi():
3. print("Hello")
4. **return** hi
5. **new** = hello()
6. **new**() //Hello

the **hi()** function is nested inside the **hello()** function. It will return each time we call **hi()**.

### Decorating functions with parameters

1. def divide(x,y):
2. print(x/y)
3. def outer\_div(func):
4. def inner(x,y):
5. **if**(x<y):
6. x,y = y,x
7. **return** func(x,y)
8. **return** inner
9. divide1 = outer\_div(divide)
10. divide1(2,4) //2.0

## Fancy Decorators

### Class Decorators

Python provides two ways to decorate a class. Firstly, we can decorate the method inside a class; there are built-in decorators like **@classmethod, @staticmethod** and **@property** in Python.

The **@classmethod** and **@staticmethod** define methods inside class that is not connected to any other instance of a class. The @property is generally used to modify the getters and setters of a class attributes.

Ex1- **@property decorator** - By using it, we can use the class function as an attribute.

1. **class** Student:
2. def \_\_init\_\_(self,name,grade):
3. self.name = name
4. self.grade = grade
5. @property
6. def display(self):
7. **return** self.name + " got grade " + self.grade
8. stu = Student("John","B")
9. print("Name:", stu.name) //Name: John
10. print("Grade:", stu.grade) //Grade: B
11. print(stu.display) //John got grade B

Ex:2 - **@staticmethod decorator**- The @staticmethod is used to define a static method in the class. It is called by using the class name as well as instance of the class.

1. **class** Person:
2. @staticmethod
3. def hello():
4. print("Hello Peter")
5. per = Person()
6. per.hello() //Hello Peter
7. Person.hello() //Hello Peter

### Singleton Class

A singleton class only has one instance. There are many singletons in Python including True, None, etc.

### Nesting Decorators

We can use multiple decorators by using them on top of each other.

1. @function1
2. @function2
3. def function(name):
4. print(f "{name}")

### **Decorator with Arguments**

It is always useful to pass arguments in a decorator. The decorator can be executed several times according to the given value of the argument.

1. Import functools
2. def repeat(num):
3. def decorator\_repeat(func): #Creating and returning a wrapper function
4. @functools.wraps(func)
5. def wrapper(\*args,\*\*kwargs):
6. **for** \_ in range(num):
7. value = func(\*args,\*\*kwargs)
8. **return** value
9. **return** wrapper
10. **return** decorator\_repeat
11. #Here we are passing num as an argument which repeats the print function
12. @repeat(num=5)
13. def function1(name):
14. print(f"{name}")

In the above example, **@repeat** refers to a function object that can be called in another function. The **@repeat(num = 5)** will return a function which acts as a decorator.

The above code may look complex but it is the most commonly used decorator pattern where we have used one additional **def** that handles the arguments to the decorator.

#### Note: Decorator with argument is not frequently used in programming, but it provides flexibility. We can use it with or without argument.

### Stateful Decorators

Stateful decorators are used to keep track of the decorator state. Let us consider ex. where we are creating a decorator that counts how many times function is called.

1. Import functools
2. def count\_function(func):
3. @functools.wraps(func)
4. def wrapper\_count\_calls(\*args, \*\*kwargs):
5. wrapper\_count\_calls.num\_calls += 1
6. print(f"Call{wrapper\_count\_calls.num\_calls} of {func.\_\_name\_\_!r}")
7. **return** func(\*args, \*\*kwargs)
8. wrapper\_count\_calls.num\_calls = 0
9. **return** wrapper\_count\_calls
10. @count\_function
11. def say\_hello():
12. print("Say Hello")
13. say\_hello()
14. say\_hello()

### Classes as Decorators

The classes are the best way to maintain state. In this section, we will learn how to use a class as a decorator. Here we will create a class that contains **\_\_init\_\_()** and take **func** as an argument. The class needs to be callable so that it can stand in for the decorated function.

To making a class callable, we implement the special **\_\_call\_\_()** method.

# Python Generators

Python Generators are the functions that return the traversal object and used to create iterators. It traverses the entire items at once. The generator can also be an expression in which syntax is similar to the list comprehension in Python.

There is a lot of complexity in creating iteration in Python; we need to implement **\_\_iter\_\_()** and **\_\_next\_\_()** method to keep track of internal states.

It is a lengthy process to create iterators. That's why the generator plays an essential role in simplifying this process. If there is no value found in iteration, it raises **StopIteration** exception.

## How to Create Generator function in Python?

It is similar to the normal function defined by the **def** keyword and uses a **yield** keyword instead of return. Or we can say that **if the body of any function contains a yield statement, it automatically becomes a generator function.**

1. def simple():
2. **for** i in range(10):
3. **if**(i%2==0):
4. yield i
5. **for** i in simple(): #Successive Function call using **for** loop
6. print(i)

### yield vs. return

The **yield** statement is responsible for controlling the flow of the generator function. It pauses the function execution by saving all states and yielded to the caller.

Later it resumes execution when a successive function is called. We can use the multiple yield statement in the generator function.

The return statement **returns** a value and terminates the whole function and only one return statement can be used in the function.

**Using multiple yield Statement**

We can use the multiple yield statement in the generator function.

1. def multiple\_yield():
2. str1 = "First String"
3. yield str1
4. str2 = "Second string"
5. yield str2
6. obj = multiple\_yield()
7. print(next(obj)) //First String
8. print(next(obj)) //Second string

### Difference between Generator function and Normal function

* Normal function contains only one **return** statement whereas generator function can contain one or more **yield** statement.
* When the generator functions are called, the normal function is paused immediately and control transferred to the caller.
* Local variable and their states are remembered between successive calls.
* StopIteration exception is raised automatically when the function terminates.

### Generator Expression

We can easily create a generator expression without using user-defined function. It is the same as the lambda function which creates an anonymous function; the generator's expressions create an anonymous generator function.

The representation of generator expression is similar to the Python list comprehension. The only difference is that **square bracket is replaced by round parentheses**. The list comprehension calculates the entire list, whereas the generator expression calculates one item at a time.

1. list = [1,2,3,4,5,6,7]
2. z = [x\*\*3 **for** x in list] # List Comprehension
3. a = (x\*\*3 **for** x in list) # Generator expression
4. print(a) //<generator object <genexpr> at 0x01BA3CD8>
5. print(z) //[1, 8, 27, 64, 125, 216, 343]

**Example:** Write a program to print the table of the given number using the generator.

1. def table(n):
2. **for** i in range(1,11):
3. yield n\*i
4. i = i+1
5. **for** i in table(15):
6. print(i)

## Advantages of Generators

### 1. Easy to implement

Generators are easy to implement as compared to the iterator. In iterator, we have to implement **\_\_iter\_\_()** and **\_\_next\_\_()** function.

### 2. Memory efficient

Generators are memory efficient for a large number of sequences. The normal function returns a sequence of the list which creates an entire sequence in memory before returning the result, but the generator function calculates the value and pause their execution.

It resumes for successive call. An infinite sequence generator is a great example of memory optimization. Let's discuss it in the below example by using **sys.getsizeof()** function.

1. **import** sys
2. nums\_squared\_list = [i \* 2 **for** i in range(1000)] # List comprehension
3. print(sys.getsizeof("Memory in Bytes:"nums\_squared\_list)) //4508
4. nums\_squared\_gc = (i \*\* 2 **for** i in range(1000)) # Generator Expression
5. print(sys.getsizeof("Memory in Bytes:", nums\_squared\_gc)) //56

### 3. Pipelining with Generators

Data Pipeline provides the facility to process large datasets or stream of data without using extra computer memory.

Suppose we have a log file from a famous restaurant. The log file has a column (4th column) that keeps track of the number of burgers sold every hour and we want to sum it to find the total number of burgers sold in 4 years. In that scenario, the generator can generate a pipeline with a series of operations. Below is the code for it:

1. with open('sells.log') as file:
2. burger\_col = (line[3] **for** line in file) per\_hour = (**int**(x) **for** x in burger\_col **if** x != 'N/A')
3. print("Total burgers sold = ",sum(per\_hour))

**4. Generate Infinite Sequence**

The generator can produce infinite items. Infinite sequences cannot be contained within the memory and since generators produce only one item at a time,

1. def infinite\_sequence():
2. num = 0
3. **while** True:
4. yield num
5. num += 1
6. **for** i in infinite\_sequence():
7. print(i)

# 

# Web Scraping Using Python

Web Scraping is a technique to extract a large amount of data from several websites. The term **"scraping"** refers to obtaining the information from another source (webpages) and saving it into a local file.

For example: Suppose you are working on a project called **"Phone comparing website,"** where you require the price of mobile phones, ratings, and model names to make comparisons between the different mobile phones. If you collect these details by checking various sites, it will take much time. In that case, web scrapping plays an important role where by writing a few lines of code you can get the desired results.

Web Scrapping extracts the data from websites in the unstructured format. It helps to collect these unstructured data and convert it in a structured form.

Startups prefer web scrapping because it is a cheap and effective way to get a large amount of data without any partnership with the data selling company.

Web scrapping is illegal if someone tries to scrap nonpublic data. Nonpublic data is not reachable to everyone; if you try to extract such data then it is a violation of legal term.

That raw data can be used in various fields. Let's have a look at usage of web scrapping:

* **Dynamic Price Monitoring**

It is widely used to collect data from several online shopping sites and compare the prices of products and make profitable pricing decisions. Price monitoring using web scrapped data gives the ability to the companies to know the market condition and facilitate dynamic pricing. It ensures the companies they always outrank others.

* **Market Research**

Web Scrapping is perfectly appropriate for market trend analysis. It is gaining insights into a particular market. The large organization requires a great deal of data, and web scrapping provides the data with a guaranteed level of reliability and accuracy.

* **Email Gathering**

Many companies use personals e-mail data for email marketing. They can target the specific audience for their marketing.

* **News and Content Monitoring**

A single news cycle can create an outstanding effect or a genuine threat to your business. If your company depends on the news analysis of an organization, it frequently appears in the news. So web scraping provides the ultimate solution to monitoring and parsing the most critical stories. News articles and social media platform can directly influence the stock market.

* **Social Media Scrapping**

Web Scrapping plays an essential role in extracting data from social media websites such as **Twitter, Facebook,** and **Instagram,** to find the trending topics.

* **Research and Development**

large set of data such as **general information, statistics& temperature** is scrapped from websites, which is analyzed and used to carry out surveys or research & development.

## Why use Python for Web Scrapping?

* **Dynamically Typed**

In Python, we don't need to define data types for variables; we can directly use the variable wherever it requires. It saves time and makes a task faster. Python defines its classes to identify the data type of variable.

* **Vast collection of libraries**

Python comes with an extensive range of libraries such as **NumPy, Matplotlib, Pandas, Scipy, etc**., that provide flexibility to work with various purposes. It is suited for almost every emerging field and also for web scrapping for extracting data & do manipulation.

* **Less Code**

Purpose of web scrapping is to save time. But what if you spend more time in writing the code? That's why we use Python, as it can perform a task in a few lines of code.

* **Open-Source Community**

Python is open-source, which means it is freely available for everyone. It has one of the biggest communities across the world where you can seek help if you get stuck anywhere in Python code.

### The basics of web scraping

The web scrapping consists of two parts: **a web crawler and a web scraper**.

In simple words, the web crawler is a horse, and the scrapper is the chariot. The crawler leads the scrapper and extracts the requested data.

**The crawler**

A web crawler is generally called a **"spider."** It is an artificial intelligence technology that browses the internet to index and searches for the content by given links. It searches for the relevant information asked by the programmer.

**The scrapper:**

A web scraper is a dedicated tool that is designed to extract the data from several websites quickly and effectively. Web scrappers vary widely in design and complexity, depending on the projects.

### How does Web Scrapping work?

These are the following steps to perform web scraping.

**Step -1: Find the URL that you want to scrape**

First, you should understand the requirement of data according to your project. A webpage or website contains a large amount of information. That's why scrap only relevant information. In simple words, the developer should be familiar with the data requirement.

**Step - 2: Inspecting the Page**

The data is extracted in raw [HTML](https://www.javatpoint.com/html-tutorial) format, which must be carefully parsed and reduce the noise from the raw data. In some cases, data can be simple as name and address or as complex as high dimensional weather and stock market data.

**Step - 3: Write the code**

Write a code to extract the information, provide relevant information, and run the code.

**Step - 4: Store the data in the file**

Store that information in required csv, [xml](https://www.javatpoint.com/xml-tutorial), [JSON](https://www.javatpoint.com/json-tutorial) file format.

### Getting Started with Web Scrapping

Python has a vast collection of libraries and also provides a very useful library for web scrapping..

**Library used for web scrapping**

* **Selenium-** Selenium is an open-source automated testing library. It is used to check browser activities.
* **Pandas :** Pandas library is used for **data manipulation and analysis**. It is used to extract the data and store it in the desired format.
* **BeautifulSoup :**BeautifulSoup is a Python library that is used to pull data of HTML and XML files.It is mainly designed for web scrapping. It works with parser to provide a natural way of navigating, searching, and modifying the parse tree.

**Installing a parser**

BeautifulSoup supports HTML parser and several third-party Python parsers. You can install any of them according to your dependency. The list of BeautifulSoup's parsers is the following:

| **Parser** | **Typical usage** |
| --- | --- |
| Python's html.parser | BeautifulSoup(markup,"html.parser") |
| lxml's HTML parser | BeautifulSoup(markup,"lxml") |
| lxml's XML parser | BeautifulSoup(markup,"lxml-xml") |
| Html5lib | BeautifulSoup(markup,"html5lib") |

We recommend you to install **html5lib** parser because it is much suitable for the newer version of Python, or you can install **lxml** parser.

BeautifulSoup is used to transform a complex HTML document into a complex tree of Python objects. But there are a few essential types object which are mostly used:

* **Tag**

A **Tag** object corresponds to an XML or HTML original document.

1. soup = bs4.BeautifulSoup("<b class = "boldest">Extremely bold</b>)
2. tag = soup.b
3. type(tag)

Tag contains lot of attributes and methods, but most important features of a tag are name and attribute.

* **Name:** Every tag has a name, accessible as **.name:**

1. tag.name

* **Attribute:** A tag may have any number of attributes. The tag <b id = "boldest"> has an attribute "id" whose value is "boldest". We can access a tag's attributes by treating tag as dictionary.

1. tag[id]

We can add, remove, & modify a tag's attributes. It can be done by using tag as dictionary.

1. tag['id'] = 'verybold' # add the element
2. tag['another-attribute'] = 1
3. tag
4. del tag['id'] # delete the tag

* **Multi-valued Attributes**

In HTML5, there are some attributes that can have multiple values. The class (consists more than one css) is the most common multivalued attributes. Other attributes are **rel, rev, accept-charset, headers,** and **accesskey**.

* **NavigableString:** A string in BeautifulSoup refers text within a tag. BeautifulSoup uses the **NavigableString** class to contain these bits of text.

1. tag.string
2. type(tag.string) # u'Extremely bold'
3. # <**class** 'bs4.element.NavigableString'>

A string is immutable means it can't be edited. But it can be replaced with another string using **replace\_with()**.

1. tag.string.replace\_with("No longer bold")
2. tag

In some cases, if you want to use a **NavigableString** outside the BeautifulSoup, the **unicode()** helps it to turn into normal Python Unicode string.

* **BeautifulSoup object**

The BeautifulSoup object represents the complete parsed document as a whole. In many cases, we can use it as a Tag object. It means it supports most of the methods described in navigating the tree and searching the tree.

1. doc=BeautifulSoup("**<document><content/>**INSERT FOOTER HERE**</document**","xml")
2. footer=BeautifulSoup("**<footer>**Here's the footer**</footer>**","xml")
3. doc.find(text="INSERT FOOTER HERE").replace\_with(footer)
4. print(doc)

### Web Scrapping Example:

Let's take an example to understand the scrapping practically by extracting the data from the webpage and inspecting the whole page.

First, open your favorite page on Wikipedia and inspect the whole page, and before extracting data from the webpage, you should ensure your requirement.

1. Import bs4 #importing the BeautifulSoup Library
2. **import** requests
3. #Creating the requests
4. res = requests.get("https://en.wikipedia.org/wiki/Machine\_learning")
5. print("The object type:",type(res))
6. # Convert the request object to the Beautiful Soup Object
7. soup = bs4.BeautifulSoup(res.text,'html5lib')
8. print("The object type:",type(soup)

In the following lines of code, we are extracting all headings of a webpage by class name. Here front-end knowledge plays an essential role in inspecting the webpage.

1. soup.select('.mw-headline')
2. **for** i in soup.select('.mw-headline'):
3. print(i.text,end = ',')

In the above code, we imported the **bs4** and **requested** the library. In the third line, we created a **res** object to send a request to the webpage. As you can observe that we have extracted all heading from the webpage.

# Python JSON

JSON stands for **JavaScript Object Notation**, which is a widely used data format for data interchange on the web. JSON is the ideal format for organizing data between a client and a server. Its syntax is similar to the JavaScript programming language. The main objective of JSON is to transmit the data between the client and the web server.

* It stores data in the name/value pairs. It is treated as an **object, record, dictionary, hash table, keyed list**.
* The ordered list of values is treated as an array, vector, list, or sequence.

## Working with Python JSON

Python provides a module called **json**. Python supports standard library marshal and pickle module, and JSON API behaves similarly as these library. Python natively supports JSON features.

The encoding of JSON data is called **Serialization**. Serialization is a technique where data transforms in the **series of bytes** and transmitted across the network.

The deserialization is the reverse process of decoding the data that is converted into the JSON format.

## Serializing JSON

Serialization is the technique to convert the Python objects to JSON. We can store JSON data into file using JSON function. The json module provides the **dump()** and **dumps()** method that are used to transform Python object.

| **Sr.** | **Python Objects** | **JSON** |
| --- | --- | --- |
| **1.** | Dict | Object |
| **2.** | list, tuple | Array |
| **3.** | Str | String |
| **4.** | int, float | Number |
| **5.** | True | true |
| **6.** | False | false |
| **7.** | None | null |

* **The dump() function**

**Writing JSON Data into File**

Python provides a **dump()** function to transmit(encode) data in JSON format. It accepts two positional arguments, first is the data object to be serialized and second is the file-like object to which the bytes needs to be written.

1. Import json
2. student = {
3. "Name" : "Peter",
4. "Roll\_no" : "0090014",
5. "Grade" : "A",
6. "Age": 20,
7. "Subject": ["Computer Graphics", "Discrete Mathematics", "Data Structure"] }
8. with open("data.json","w") as write\_file:
9. json.dump(student,write\_file)

* **The dumps () function**

The **dumps()** function is used to store serialized data in the Python file. It accepts only one argument that is Python data for serialization. The file-like argument is not used because we aren't not writing data to disk.

### Deserializing JSON

Deserialization is the process to decode the JSON data into the Python objects. The json module provides two methods **load()** and **loads()**, which are used to convert JSON

data in actual Python object form. The list is given below:

* **The load() function**

The **load()** function is used to deserialize the JSON data to Python object from the file.

1. **import** json
2. with open("data.json", "r") as read\_file:
3. b = json.load(read\_file)
4. print(b)

The json module also provides **loads()** function, which is used to convert JSON data to Python object. It is quite similar to the **load()** function.

1. Import json
2. a = ["Mathew","Peter",(10,32.9,80),{"Name" : "Tokyo"}]
3. b = json.dumps(a) # Python object into JSON
4. c = json.loads(b) # JSON into Python Object
5. print(c)

### json.load() vs json.loads()

**json.load()** func is used to load JSON file, whereas **json.loads()** func used to load string.

### json.dump() vs json.dumps()

**json.dump()** functon is used when we want to serialize the Python objects into JSON file & **json.dumps()** function is used to convert JSON data as a string for parsing & printing.

## Python Pretty Print JSON

Sometimes we need to analyze & debug a large amount of JSON data. It can be done by pasing additional argument indent & sort\_keys in json.dumps() & json.dump() methods.

**Note: Both dump() and dumps() functions accept indent and short\_keys arguments.**

1. **import** json
2. person = '{"Name": "Andrew","City":"English", "Number":90014, "Age": 23,"Subject": ["Data Structure","Computer Graphics", "Discrete mathematics"]}'
3. per\_dict = json.loads(person)
4. print(json.dumps(per\_dict, indent = 5, sort\_keys= True))

In the above code, we have provided the 5 spaces to the indent argument and the keys are sorted in ascending order. The default value of indent is **None** and the default value of **sort\_key** is **False**.

## Encoding and Decoding

Encoding is the technique for transforming the text or values into an encrypted form. Encrypted data can only be used by preferred user by decoding it. Encoding is also known as **serialization** & decoding is also called **deserialization**. Encoding & decoding are done for JSON(object) format. Python provides a popular package for such operations.

**Encoding** - The demjson package provides **encode()** function that is used to convert the Python object into a JSON string representation.

**demjson.encode(self,obj,nest\_level = 0)**

1. **import demjson**
2. **a = [{"Name": 'Peter',"Age":20, "Subject":"Electronics"}]**
3. **print(demjson.encode(a))**

**Decoding**-The **demjson** module provides **decode()** function, which is used to convert JSON object into Python format type.

1. Import demjson
2. a = "['Peter', 'Smith', 'Ricky', 'Hayden']"
3. print(demjson.decode(a))

# Python Itertools

According to the official definition of itertools, "**this module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML**."

In simple words, the number of iterators can together create 'iterator algebra' which makes it possible to complete the complex task. The functions in itertools are used to produce more complex iterators.

Let's take an example: [Python built-in zip() function](https://www.javatpoint.com/python-zip-function) accepts any number of arguments as iterable. It iterates over tuples and return their corresponding elements.

1. a = [1,2,3]
2. b= ['a', 'b', 'c']
3. c = zip(a,b)
4. print(c) //[(1, 'a'), (2, 'b'), (3, 'c')]

In the above code, we have passed two lists [1,2,3] and ['a', 'b', 'c'] as iterable in **zip()** function**.** These lists return one element at a time. In [Python](https://www.javatpoint.com/python-tutorial), an element that implement **.\_\_iter\_\_()** or **.\_\_getitem\_\_()** method called iterable.

The [Python zip() function](https://www.javatpoint.com/python-zip-function) calls **iter()** on each of its argument and then calls **next()** by combining the result into tuple.

#### Note: If you are using the zip() function and map() function that means you are already using itertools. You don't need to import it distinctly.

## Types of Iterator

There are various types of iterator in itertools module. The list is given below:

* Infinite iterators
* Combinatoric iterators
* Terminating iterators

### 1)Infinite Iterators

In Python, any object that can implement **for loop** is called iterators. Lists, tuples, set, dictionaries, strings are the example of iterators but iterator can also be infinite and this type of iterator is called **infinite iterator**.

| **Iterator** | **Argument** | **Results** |
| --- | --- | --- |
| count(start,step) | start, [step] | start, start+step, step+2\*step |
| cycle() | P | p0,p1,….plast |
| repeat() | elem [,n] | elem, elem, elem,….endlessly or upto n times |

**A)count(start, stop)**: It prints from the start value to infinite. The step argument is optional, if the value is provided to the **step** then number of steps will be skipped.

* **import** itertools
* **for** i in itertools.count(10,5):
* **if** i == 50:
* **break**
* **else**:
* print(i,end=" ") //10 15 20 25 30 35 40 45

**B)cycle(iterable)**: This iterator prints all value in sequence from the passed argument. It prints the values in a cyclic manner.

1. **import** itertools
2. temp = 0
3. **for** i in itertools.cycle("123"):
4. **if** temp > 7:
5. **break**
6. **else**:
7. print(i,end=' ') //1 2 3 1 2 3 1 2 3 1 2
8. temp = temp+1

**C)repeat(val,num)**: As the name suggests, it repeatedly prints the passed value for infinite time. The **num** argument is optional.

1. **import** itertools
2. print("Printing the number repeadtly:")
3. print(list(itertools.repeat(40,12))) //[40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,]

**2)Combinatoric iterators:** The complex combinatorial constructs are simplified by the recursive generators. The permutations, combinations, and Cartesian products are the example of the combinatoric construct.

In Python, there are four types of combinatoric iterators:

**A)Product() -** It is used to calculate the cartesian product of input iterable. In this function, we use the optional **repeat** keyword argument for computation of the product of an iterable with itself. The **repeat** keyword represents the number of repetitions. It returns output in the form of sorted tuples.

1. from itertools **import** product
2. print("We are computing cartesian product using repeat Keyword Argument:")
3. print(list(product([1, 2], repeat=2))) //[(1, 1), (1, 2), (2, 1), (2, 2)]
4. print("We are computing cartesian product of the containers:")
5. print(list(product(['Java', 'T', 'point'], '5'))) //[('Java', '5'), ('T', '5'), ('point', '5')]
6. print("We are computing product of the containers:")
7. print(list(product('CD', [4, 5]))) //[('C', 4), ('C', 5), ('D', 4), ('D', 5)]

**B)Permutations()**: It is used to generate all possible permutation of an iterable. The uniqueness of each element depends upon their position instead of values. It accepts two argument **iterable** and **group\_size**. If the value of group\_size is **none** or not specified then group\_size turns into length of the iterable.

1. from itertools **import** permutations
2. print("Computing all permutation of the following list")
3. print(list(permutations([3,"Python"],2))) //[(3, 'Python'), ('Python', 3)]
4. print("Permutations of following string")
5. print(list(permutations('AB'))) //[('A', 'B'), ('B', 'A')]
6. print("Permutation of the given container is:")
7. print(list(permutations(range(4),2))) //[(0, 1), (0, 2), (0, 3), (1, 0), (1, 2), (1, 3), (2, 0), (2, 1), (2, 3), (3, 0), (3, 1), (3, 2)]

**C)Combinations()**: It is used to print all the possible combinations (without replacement) of the container which is passed as argument in the specified group size in sorted order.

1. from itertools **import** combinations
2. print("Combination of list in sorted order(without replacement)",list(combinations(['B',3],2))) //[('B', 3)]
3. print("Combination of str in sorted order",list(combinations("ZX",2))) //[('Z', 'X')]
4. print("Combination of list in sorted order",list(combinations(range(20),1)))

//[(0,), (1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,), (9,)]

* **D)Combination\_with\_replacement()**: It accepts two arguments, first argument is a r-length tuple and the second argument is repetition. It returns a subsequence of length n from the elements of the iterable and repeat the same process. Separate elements may repeat itself in **combination\_with\_replacement()**

1. from itertools **import** combinations\_with\_replacement
2. print("Combination of string in sorted order(with replacement) is:")
3. print(list(combinations\_with\_replacement("XY", 3))) //[('X', 'X', 'X'), ('X', 'X', 'Y'), ('X', 'Y', 'Y'), ('Y', 'Y', 'Y')]
4. print("Combination of list in sorted order(with replacement) is:")
5. print(list(combinations\_with\_replacement([4, 2], 3))) //[(4, 4, 4), (4, 4, 2), (4, 2, 2), (2, 2, 2)]
6. print("Combination of container in sorted order(with replacement) is:")
7. print(list(combinations\_with\_replacement(range(3), 2))) //[(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)]

### 3)Terminating Iterator

Terminating iterators are generally used to work on the small input sequence and generate the output based on the functionality of the method used in iterator.

There are different types of terminating iterator:

* **A) accumulate(iter, func)**: It takes two arguments, the first argument is iterable and the second is a function which would be followed at each iteration of value in iterable. If the function is not defined in **accumulate()** iterator, addition takes place by default. The output iterable depends on the input iterable; if input iterable contains no value then the output iterable will also be empty.

1. **import** itertools
2. **import** operator
3. list1 = [1, 4, 5, 7, 9, 11]
4. # using accumulate() that will prints the successive summation of elements
5. print("The sum is : ", end="")
6. print(list(itertools.accumulate(list1))) //[1, 5, 10, 17, 26, 37]
7. # using accumulate() that will prints the successive multiplication of elements
8. print("The product is : ", end="")
9. print(list(itertools.accumulate(list1, operator.mul))) //[1, 4, 20, 140, 1260, 13860]

**B)chain(iter1, iter2)** - It is used to print all the values in iterable passed in the form of chain and declared in arguments.

1. **import** itertools
2. list1 = [1, 2, 3, 4]
3. list2 = [1, 5, 6, 8]
4. list3 = [9, 10, 11, 12]
5. # using chain() function that will to print all elements of lists
6. print(list(itertools.chain(list1, list2, list3))) //[1, 2, 3, 4, 1, 5, 6, 8, 9, 10, 11, 12]

**C)dropwhile(func, seq)** - It starts printing the character only after the **func**.

1. **import** itertools
2. list1 = [2, 4, 5, 7, 8]
3. # using dropwhile() iterator that will print start displaying after condition is **false**
4. print(list(itertools.dropwhile(lambda x: x % 2 == 0, list1))) //[5, 7, 8]

**G)takewhile(func, iterable)** - It is visa-versa of **dropwhile().** It will print values until it returns false condition.

1. **import** itertools
2. list1 = [20, 42, 64, 77, 8, 10, 20]
3. # takewhile() iterator is used to print values till condition **return** **false**.
4. print(list(itertools.takewhile(lambda x: x % 2 == 0, list1))) //[20, 42, 64]

**D)filterfalse(func,seq)** - We can assume it by its name, as this iterator prints only those values that return false for the passed function.

1. **import** itertools
2. list1 = [12, 14, 15, 27, 28]
3. # using filterfalse() iterator that will print **false** values
4. print(list(itertools.filterfalse(lambda x: x % 2 == 0, list1))) //[15, 27]

**E)islice(iterable,start,stop,step)** - It slices the given iterable according to given position. It accepts four arguments respectively and these are iterable, container, starting pos., ending position and step(optional).

1. **import** itertools
2. list1 = [12, 34, 65, 73, 80, 19, 20]
3. # starts printing from 3nd index till 8th skipping 2
4. print(list(itertools.islice(list1, 2, 8, 2))) // [65, 80, 20]

**F)starmap(func, tuple list) -** It takes two arguments; first argument is function and second argument is list which consists element in the form of tuple.

1. **import** itertools
2. # Declaring list that contain tuple as element
3. list1 = [(10, 20, 15), (18, 40, 19), (53, 42, 90), (16, 12, 27)]
4. # selects max of all tuple values
5. print(list(itertools.starmap(max, list1))) //[20, 40, 90, 27]

**H)tee(iterator, count)** - It divides the container into a number of iterators which is defined in the argument.

1. **import** itertools
2. li = [1, 2, 3, 4, 5, 6, 7]
3. iti = iter(li)
4. it = itertools.tee(iti, 3) # Creating list of 3 iterators having similar values.
5. **for** i in range(0, 2):
6. print(list(it[i])) // [1, 2, 3, 4, 5, 6, 7] [1, 2, 3, 4, 5, 6, 7]

**I)zip\_longest(iterable1, iterable2, fillval)** - It prints the values of iterable alternatively in sequence. If one of the iterable prints all values, remaining values are filled by the values assigned to fill value.

1. **import** itertools
2. print(\*(itertools.zip\_longest('Java', 'Tpoint', fillvalue='\_')))

// ('J', 'T') ('a', 'p') ('v', 'o') ('a', 'i') ('\_', 'n') ('\_', 't')

# Python Multiprocessing

Multiprocessing is the ability of the system to run one or more processes in parallel. In simple words, multiprocessing uses the two or more [CPU](https://www.javatpoint.com/cpu-full-form) within the single computer system. This method is also capable to allocate tasks between more than one process.

Processing units share main memory & peripheral to process programs simultaneously.

* Multiprocessing can be represented as a computer with more than one central processor.
* A Multi-core processor refers to single computing component with two or more independent units.

1. from multiprocessing import Process
2. def disp():
3. print ('Hello !! Welcome to Python Tutorial')
4. if \_\_name\_\_ == '\_\_main\_\_':
5. p = Process(target=disp)
6. p.start()
7. p.join()

In the above code, we have imported the Process class then create the Process object within the **disp()** function. Then we started the process using the **start()** method and completed the process with the **join()** method. We can also pass the arguments in the declared function using the **args** keywords.

1. import multiprocessing
2. def square(n):
3. print("The Square is: {}".format(n \* n)) //# It will print square of given number
4. if \_\_name\_\_ == "\_\_main\_\_":
5. process1 = multiprocessing.Process(target= square, args=(5, ))
6. process1.start()
7. process1.join()

## Python Multiprocessing Classes

Python multiprocessing module provides many classes which are commonly used for building parallel program. We will discuss its main classes - Process, Queue and Lock. We have already discussed the Process class in the previous example. Now we will discuss the Queue and Lock classes.

1. import multiprocessing
2. print("no. of CPU currently working in system : ", multiprocessing.cpu\_count())

Queues are passed as a parameter in the Process' target function to allow the process to consume data. The Queue provides the **put()** function to insert the data and **get()** function to get data from the queues.

1. from multiprocessing import Queue
2. fruits = ['Apple', 'Orange', 'Guava', 'Papaya', 'Banana']
3. count = 1
4. queue = Queue() # creating a queue object
5. for fr in fruits: #print('pushing items to the queue:')
6. print('item no: ', count, ' ', fr)
7. queue.put(fr)
8. count += 1
9. print('\npopping items from the queue:')
10. count = 0
11. while not queue.empty():
12. print('item no: ', count, ' ', queue.get())
13. count += 1

## Python Multiprocessing Lock Class

The multiprocessing Lock class is used to acquire a lock on the process so that we can hold the other process to execute a similar code until the lock has been released. The Lock class performs mainly two tasks. The first is to acquire a lock using the **acquire()** function and the second is to release the lock using the **release()** function.

## Python Multiprocessing Example

Suppose we have multiple tasks. So, we create two queues: the first queue will maintain the tasks, and the other will store the complete task log. The next step is to instantiate the processes to complete the task. As discussed previously, the Queue class is already synchronized, so we don't need to acquire a lock using the Lock class.

## Python Multiprocessing Pool

Python multiprocessing pool is essential for parallel execution of a function across multiple input values. It is also used to distribute the input data across processes **(data parallelism)**.

1. from multiprocessing import Pool
2. def fun(x):
3. return x\*x
4. if \_\_name\_\_ == '\_\_main\_\_':
5. with Pool(5) as p:
6. print(p.map(fun, [1, 2, 3])) //[1, 4, 9]

## Commonly Used Functions of Multiprocessing

| **Method** | **Description** |
| --- | --- |
| pipe() | The pipe() function returns a pair of connection objects. |
| run() | The run() method is used to represent the process activities. |
| start() | The start()method is used to start the process. |
| join([timeout]) | The join() method is used to block the process until the process whose join() method is called terminates. The timeout is optional argument. |
| is\_alive() | It returns if process is alive. |
| terminate() | As the name suggests, it is used to terminate the process. Always remember - the **terminate()** method is used in Linux, for Windows, we use **TerminateProcess()** method. |
| kill() | This method is similar to the **terminate()** but using the SIGKILL signal on Unix. |
| close() | This method is used to close the **Process** object & releases all resources associated with it. |
| qsize() | It returns the approximate size of the queue. |
| empty() | If queue is empty, it returns **True**. |
| full() | It returns **True**, if queue is full. |
| get\_await() | This method is equivalent **get(False)**. |
| get() | It is used to get elements from the queue. It removes and returns an element from queue. |
| put() | This method is used to insert an element into the queue. |
| cpu\_count() | It returns the number of working CPU within the system. |
| current\_process() | It returns the Process object corresponding to the current process. |
| parent\_process() | It returns the parent Process object corresponding to the current process. |
| task\_done() | This function is used indicate that an enqueued task is completed. |
| join\_thread() | This method is used to join the background thread |

# Calculate Distance b/w Two Points using GEOPY

The **geopy** is a Python library which helps to calculate geographical distance.

## Calculate Distance between Two Points

Below are important methods that used to calculate the distance between two points.

### Method 1: By using Geodesic Distance

The geodesic distance is the length of the shortest path between two points on any surface of Earth. In the following example, we will show how the user can calculate the Geodesic Distance from the latitude and longitude data.

1. from geopy.distance **import** geodesic as GD
2. New\_York = (40.7128, 74.0060)
3. Texas = (31.9686, 99.9018)
4. print ("distance b/w New York & Texas is: ", GD(New\_York, Texas).km) //2507km

### Method 2: By using Great Circle Distance

The great circle distance is the shortest path between two points on the sphere. In this case, we will assume the earth is the perfect sphere. following example shows how user can calculate great circle distance by using longitude and latitude data of two points.

1. from geopy.distance **import** great\_circle as GC
2. New\_York = (40.7128, 74.0060)
3. Texas = (31.9686, 99.9018)
4. print (" distance b/w New York & Texas is: ", GC(New\_York, Texas).km) //2503km

# Grid Search in Python

the Grid Search is used for hyperparameter tuning.

**Hyperparameter tuning** is significant for the appropriate working of the models of [Machine Learning (ML)](https://www.javatpoint.com/machine-learning). A method like **Grid Search** appears to be a basic utility for hyperparameter optimization.

The **Grid Search** Method considers some hyperparameter combinations and selects the one returning a lower error score. This method is specifically useful when there are only some hyperparameters in order to optimize. However, it is outperformed by other weighted-random search methods when Machine Learning model grows in complexity.

## Understanding Grid Search

**Grid Search** is an optimization algorithm that allows us to select the best parameters to optimize the issue from a list of parameter choices we are providing, thus automating the 'trial-and-error' method.

Although we can apply it to multiple optimization issues; however, it is most commonly known for its utilization in machine learning in order to obtain the parameters at which the model provides the best accuracy.

Let us consider that the model accepts the below three parameters in the form of input:

1. Number of hidden layers [2, 4]
2. Number of neurons in every layer [5, 10]
3. Number of epochs [10, 50]

If we want to try out two options for every parameter input (as specified in square brackets above), it estimates different combinations. For instance, one possible combination can be [2, 5, 10]. Finding such combinations manually would be a headache.

Now, suppose that we had ten different parameters as input, and we would like to try out five possible values for each and every parameter. It would need manual input from the programmer's end every time we like to alter the value of a parameter, re-execute the code, and keep a record of the outputs for every combination of the parameters.

Grid Search automates that process, as it accepts the possible value for every parameter and executes the code in order to try out each and every possible combination outputs the result for the combinations and outputs the combination having the best accuracy.

## Implementation of Grid Search in Python

1. mydf = pd.read\_csv("Diet\_Dataset.csv") # importing the dataset
2. print(mydf.head()) //# head method() print the first five lines of dataset

# Python High Order Function

A function that is having another function as an argument or a function that returns another function as a return in the output is called the High order function. High order functions operate with other functions given in the program.

### Properties of High order functions in Python

* In high order function, we can store a function inside a variable.
* In high order function, a function can act as an instant of an object type.
* In high order function, we can return a function as a result of another function.
* We can pass a function as a parameter or argument inside another function.
* We can store Python high order functions in data structures format such as lists, hash tables, etc.

### Method 1: Using functions as objects in High order function

In Python, we can even assign a given function to a variable also. This assignment of function into variable will not call the actual function, instead of it will create a reference to the function that is created. Thus, it makes this assignment of assigning a function as a variable object will create a high order function in the program.

1. def spell(text):
2. **return** text.upper() # Making text in upper
3. text = input("Enter a text to print in uppercase and double: ") # Taking user input
4. print(spell(text)) //JAVATPOINT
5. scream = spell # Assigning variable with the **default** function
6. # Scream with text variable
7. print(scream(text)) //JAVATPOINT

### Method 2: Functions as a parameter for another function

Basically, Python functions are like Python objects, and therefore we can use Python functions to pass them as an argument inside another function, and that will create a high order function in the program.

1. def scream(word):
2. **return** word.upper()
3. def spell(word):
4. **return** word.lower()
5. def speak(funct): # A third function that work as a high order function
6. speaking = funct("Hello!") # Storing function in variable in high order function
7. print(speaking)
8. speak(scream) # Printing text in uppercase
9. speak(spell) # Printing text in lowercase

### Method 3: Returning function as a result in high order function

We can also return a function as the result of another function as an object, and that makes the function a high order function.

1. def Adding(a):
2. def Addition(b): # Nested function with second number
3. **return** a + b # addition of two numbers
4. **return** Addition # Result
5. a,b=5,10
6. AddVariable = Adding(a) # Assigning nested adding function to a variable
7. Result = AddVariable(b) # Using variable as high order function
8. print("Sum of Two numbers given by you is: ", Result) //15

### Method 4: Decorators as high order function

Decorators in Python allow us to modify the behavior of methods or functions we defined in the program, and it also allows us to wrap a function inside another function to extend the behavior of wrapped or parent function. We can even wrap a function inside another function without even permanently modifying the parent function.

In Python decorators, a function is taken as an argument for the other function, and then these decorators are called inside the wrapped function.

1. def Python\_Decorator():
2. .
3. Python\_Decorator = @JTP\_Decorator(Python\_Decorator)

We have referred @JTP\_Decorator as a callable function inside the default Python\_Decorator() function in the above-given code. We will have to add just some extra code in this structure, and we will get the output as the wrapper function.

1. def Python\_Decorator(funct):
2. def inner(): # Inner nested function
3. print("It will be printed before the execution of high order function") //1
4. funct()
5. print("It will be printed after the execution of high order function") //3
6. **return** inner
7. def JTP\_Decorator(): # A **default** function as decorator
8. print("It will be printed inside the execution of high order function") //2
9. JTP= Python\_Decorator(JTP\_Decorator) #Python decorator as high order functio
10. JTP() # Python decorator calling out as high order function

# nsetools in Python

Python offers a library that allows the programmers to collect real-time data from National Stock Exchange (India). This library is known as **nsetools**.

We can use this library in different projects, which requires fetching live quotes for a provided index or stock or creating large sets of data for further data analytics.

We can also create Command-Line Interface (CLI) Applications that may deliver us the details of the live market at a blazing fast speed, pretty faster than any web browser.

## Main features of the Python nsetools library

1. The **nsetools** library works out of the box, without any setup requirement.
2. This library helps programmers to fetch livestock code and index codes at blazing fast speed.
3. It also offers a set of all stocks & indices traded on the National Stock Exchange.
4. Moreover, it also provides a set of:**Top losers,Top gainers,Most active**
5. It also delivers several helpful APIs in order to validate a stock code & index code.
6. The library optionally returns data in JSON format.
7. It has a hundred per cent Unit test coverage.

## Creating an NSE object

We can create an NSE object using the **Nse()** function offered by the **nsetools** library.

1. from nsetools **import** Nse
2. nse\_obj = Nse()
3. print("NSE Object:", nse\_obj)

# Python OpenCV object detection

OpenCV is the huge and open-source library for image processing, machine learning and computer vision. It is also playing an important role in real-time operation. With the help of the OpenCV library, we can easily process the images as well as videos to identify the objects, faces or even handwriting of a human present in the file.

## Object Detection

Basically, object detection is a modern computer technology that is related to image processing, deep learning and computer vision to detect the objects present in an image file. All the technologies used in the Object detection technique (as we mentioned earlier) deals with detecting instances of the object in the image or video.

## Object Detection using OpenCV

### Haar cascade:

Basically, the Haar cascade technique is an approach based on machine learning where we use a lot of positive and negative images to train the classifier to classify between the images. Haar cascade classifiers are considered as the effective way to do object detection with the OpenCV library. Now, let's understand the concept of positive and negative images that we have discussed earlier:

* **Positive images:** These are the images that contain the objects which we want to be identified from the classifier.
* **Negative Images:** These are the images that do not contain any object that we want to be detected by the classifier, and these can be images of everything else.

### Requirements for object detection with Python OpenCV:

### 1. Installation of OpenCV library:

### 2. Installation of matplotlib library: Matplotlib is very helpful in the opening, closing, reading etc., images in a Python program, and that's why the installation of this library for object detection becomes an important requirement.

1. **import** cv2
2. from matplotlib **import** pyplot as pltd
3. imaging=cv2.imread("C:\\Users\VishalKumar\PycharmProjects\pythonProject\Bishal.jpg") # Opening the image from files
4. # Altering properties of image with cv2
5. img\_gray = cv2.cvtColor(imaging, cv2.COLOR\_BGR2GRAY)
6. imaging\_rgb = cv2.cvtColor(imaging, cv2.COLOR\_BGR2RGB)
7. pltd.subplot(1, 1, 1) # Plotting image with subplot() from plt
8. pltd.imshow(imaging\_rgb) # Displaying image in the output
9. pltd.show()

**Explanation:** we have opened the image file using the imread() function of cv2.

Then, we have defined the properties for the image we opened in the program using the cv2 functions. Then, we subplot the image using the subplot() function of plt and giving parameters in it. In last, we have used the imshow() and show() function of the plt module to show the image in the output.

As we can see in the output, the image is displayed as a result of the program, and its borders have been sub-plotted.

### Recognition or object detection in the image

Now, we will use detectMultiScale() in program to detect the object present in image.

1. found = xml\_data.detectMultiScale(img\_gray, minSize = (30, 30))

We will use a condition statement with this function in the program to check if any object from the image is detected or not and highlight the detected part.

1. **import** cv2
2. from matplotlib **import** pyplot as plt
3. imaging = cv2.imread("opencv-od.png")
4. imaging\_gray = cv2.cvtColor(imaging, cv2.COLOR\_BGR2GRAY)
5. imaging\_rgb = cv2.cvtColor(imaging, cv2.COLOR\_BGR2RGB)
6. # Importing Haar cascade classifier xml data
7. xml\_data = cv2.CascadeClassifier('XML-data.xml')
8. # Detecting object in the image with Haar cascade classifier
9. detecting = xml\_data.detectMultiScale(imaging\_gray, minSize = (30, 30))
10. amountDetecting = len(detecting) # Amount of object detected
11. **if** amountDetecting != 0: # Using **if** condition to highlight the object detected
12. **for** (a, b, width, height) in detecting:
13. cv2.rectangle(imaging\_rgb, (a, b), # Highlighting detected obj with rectangle
14. (a + height, b + width),
15. (0, 275, 0), 9)
16. pltd.subplot(1, 1, 1) # Plotting image with subplot() from plt
17. pltd.imshow(imaging\_rgb) # Displaying image in the output
18. pltd.show()

# Python SimpleImputer module

A scikit-learn class that we can use to handle the missing values in the data from the dataset of a predictive model is called SimpleImputer class. With the help of this class, we can replace NaN (missing values) values in the dataset with a specified placeholder. We can implement and use this module class by using the SimpleImputer() method in the program.

**SimpleImputer(missingValues, strategy)**

**Parameters:** Following are the parameters which has to be defined while using the SimpleImputer() method:

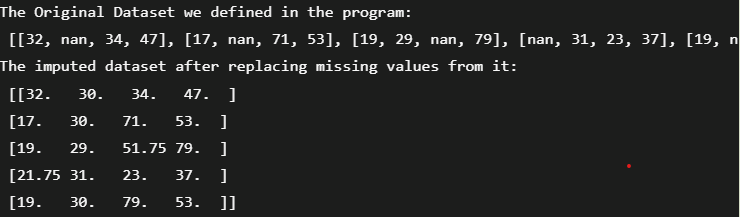
1. **missingValues:** It is the missing values placeholder in the SimpleImputer() method which has to be imputed during the execution, and by default, the value for missing values placeholder is NaN.
2. **strategy**: It is the data that is going to replace the missing values (NaN values) from the dataset, and by default, the value method for this parameter is 'Mean'. The strategy parameter of the SimpleImputer() method can take 'Mean', 'Mode', Median' (Central tendency measuring methods) and 'Constant' value input in it.
3. **fillValue:** This parameter is used only in the strategy parameter if we give 'Constant' as replacing value method. We have to define the constant value for the strategy parameter, which is going to replace the NaN values from the dataset.

### Handling NaN values in the dataset with SimpleImputer class

Now, we will use the SimpleImputer class in a Python program to handle the missing values present in the dataset (that we will use in the program).

We will define a dataset in the example program while giving some missing values in it, and then we use the SimpleImputer class method to handle those values from the dataset by defining its parameters.

1. **import** numpy as nmp
2. from sklearn.impute **import** SimpleImputer
3. imputerFunc = SimpleImputer(missing\_values = nmp.nan, strategy ='mean')
4. dataSet = [[32, nmp.nan, 34, 47], [17, nmp.nan, 71, 53], [19, 29, nmp.nan, 79], [nmp.nan, 31, 23, 37], [19, nmp.nan, 79, 53]] # Defining a dataset
5. print(" Original Dataset we defined in program: ", dataSet) # Print original dataset
6. imputrFun= imputerFunc.fit(dataSet) # Imputing dataset by replacing missing val
7. dataSet2 = imputrFun.transform(dataSet)
8. print("The imputed dataset after replacing missing values from it: \n", dataSet2)



# Python OOPs Concepts

## Class:The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods.

**class** ClassName:

In Python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.\_\_doc\_\_.** A class contains a statement suite including fields, constructor, function, etc. definition.

1. **class** Employee:
2. id = 10
3. name = "Devansh"
4. **def** display (self):
5. **print**(self.id,self.name)

**self** is used as a reference variable, which refers to the current class object. It is always the first argument in the function definition., using **self** is optional in the function call.

## Delete the Object

We can delete the properties of the object or object itself by using the del keyword.

1. **class** Employee:
2. id = 10
3. name = "John"
4. **def** display(self):
5. **print**("ID: %d \nName: %s" % (self.id, self.name))
6. emp = Employee()
7. **del** emp.id
8. **del** emp
9. emp.display() //will throw error

# Python Constructor: A constructor is a special type of method (function) which is used to initialize the instance members of the class.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

## Creating the constructor in python

In Python, the method the **\_\_init\_\_()** simulates the constructor of the class. This method is called when the class is instantiated. It accepts the **self**-keyword as a first argument which allows accessing the attributes or method of the class.

We can pass any number of arguments at the time of creating the class object, depending upon the **\_\_init\_\_()** definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

1. **class** Employee:
2. **def** \_\_init\_\_(self, name, id):
3. self.id = id
4. self.name = name
5. **def** display(self):
6. **print**("ID: %d \nName: %s" % (self.id, self.name))
7. emp1 = Employee("John", 101)
8. emp2 = Employee("David", 102)
9. emp1.display() # accessing display() method to print employee 1 information
10. emp2.display() # accessing display() method to print employee 2 information

## Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument.

## Python Parameterized Constructor

The parameterized constructor has multiple parameters along with the **self**.

1. **class** Student:
2. **def** \_\_init\_\_(self, name): # Constructor - parameterized
3. **print**("This is parametrized constructor")
4. self.name = name
5. **def** show(self):
6. **print**("Hello",self.name)
7. student = Student("John")
8. student.show()

## More than One Constructor in Single class

1. **class** Student:
2. **def** \_\_init\_\_(self):
3. **print**("The First Constructor")
4. **def** \_\_init\_\_(self):
5. **print**("The second contructor")
6. st = Student() //The second contructor

**the object of the class will always call the last constructor if the class has multiple constructors.**

#### Note: The constructor overloading is not allowed in Python.

## Python built-in class functions

| **Function** | **Description** |
| --- | --- |
| getattr(obj,name,default) | It is used to access the attribute of the object. |
| setattr(obj, name,value) | It is used to set a particular value to the specific attribute of an object. |
| delattr(obj, name) | It is used to delete a specific attribute. |
| hasattr(obj, name) | It returns true if the object contains some specific attribute. |

1. **class** Student:
2. **def** \_\_init\_\_(self, name, id, age):
3. self.name = name
4. self.id = id
5. self.age = age
6. s = Student("John", 101, 22) # creates the object of the class Student
7. **print**(getattr(s, 'name')) # prints the attribute name of the object s
8. setattr(s, "age", 23) # reset the value of attribute age to 23
9. **print**(getattr(s, 'age')) # prints the modified value of age
10. **print**(hasattr(s, 'id')) # prints true if student contains the attribute with name id
11. delattr(s, 'age') # deletes the attribute age
12. **print**(s.age) # this will give an error since the attribute age has been deleted

## Built-in class attributes

Along with the other attributes, a Python class also contains some built-in class attributes which provide information about the class.

| **SN** | **Attribute** | **Description** |
| --- | --- | --- |
| 1 | \_\_dict\_\_ | It provides the dictionary containing the information about the class namespace. |
| 2 | \_\_doc\_\_ | It contains a string which has the class documentation |
| 3 | \_\_name\_\_ | It is used to access the class name. |
| 4 | \_\_module\_\_ | It is used to access the module in which, this class is defined. |
| 5 | \_\_bases\_\_ | It contains a tuple including all base classes. |

1. **class** Student:
2. **def** \_\_init\_\_(self,name,id,age):
3. self.name = name;
4. self.id = id;
5. self.age = age
6. **def** display\_details(self):
7. **print**("Name:%s, ID:%d, age:%d"%(self.name,self.id))
8. s = Student("John",101,22)
9. **print**(s.\_\_doc\_\_) //None
10. **print**(s.\_\_dict\_\_) //{'name': 'John', 'id': 101, 'age': 22}
11. **print**(s.\_\_module\_\_) //\_\_main\_\_

## Object:-The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

## Method:The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

## Inheritance:child object acquires all the properties & behaviors of parent object.

A class can inherit multiple classes by mentioning all of them inside the bracket.

**class** derive-**class**(<base **class** 1>, <base **class** 2>, ..... <base **class** n>):

1. **class** Animal:
2. **def** speak(self):
3. **print**("Animal Speaking")
4. **class** Dog(Animal): #child class Dog inherits the base class Animal
5. **def** bark(self):
6. **print**("dog barking")
7. d = Dog()
8. d.bark() //dog barking
9. d.speak() //Animal Speaking

## Python Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

1. **class** class1:
2. <**class**-suite>
3. **class** class2(class1):
4. <**class** suite>
5. **class** class3(class2):
6. <**class** suite>

## Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.

1. **class** Base1:
2. <**class**-suite>
3. **class** Base2:
4. <**class**-suite>
5. **class** BaseN:
6. <**class**-suite>
7. **class** Derived(Base1, Base2, ...... BaseN):
8. <**class**-suite>

**If child & parent class both have same method then priority goes to child class method**.

## The issubclass(sub,sup) method

The issubclass(sub, sup) method is used to check the relationships between specified classes. It returns true if first class is the subclass of second class, and false otherwise.

1. **class** Calculation1:
2. **def** Summation(self,a,b):
3. **return** a+b;
4. **class** Calculation2:
5. **def** Multiplication(self,a,b):
6. **return** a\*b;
7. **class** Derived(Calculation1,Calculation2):
8. **def** Divide(self,a,b):
9. **return** a/b;
10. d = Derived()
11. **print**(issubclass(Derived,Calculation2)) **//True**
12. **print**(issubclass(Calculation1,Calculation2)) //**False**

## The isinstance (obj, class) method

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

## Method Overriding

We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.

## Data abstraction in python

Abstraction is an important aspect of object-oriented programming. In python, we can also perform data hiding by adding the double underscore (\_\_\_) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

1. **class** Employee:
2. \_\_count = 0;
3. **def** \_\_init\_\_(self):
4. Employee.\_\_count = Employee.\_\_count+1
5. **def** display(self):
6. **print**("The number of employees",Employee.\_\_count) //2
7. emp = Employee()
8. emp2 = Employee()
9. **try**:
10. **print**(emp.\_\_count) //will throw error
11. **finally**:
12. emp.display()

## Polymorphism:when an object gives diff-2 behavior in diff-2 circumstances.

## Encapsulation:It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

## Abstraction: Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

A class that consists of one or more abstract method is called the abstract class. Abstract methods do not contain their implementation. Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass.

Python provides the **abc** module to use the abstraction in the Python program.

1. from abc **import** ABC
2. **class** ClassName(ABC):

We use the ***@abstractmethod*** decorator to define an abstract method or if we don't provide the definition to the method, it automatically becomes the abstract method.

1. from abc **import** ABC, abstractmethod
2. **class** Car(ABC):
3. def mileage(self): # **abstract** method
4. pass
5. **class** Tesla(Car):
6. def mileage(self):
7. print("The mileage is 30kmph")
8. **class** Suzuki(Car):
9. def mileage(self):
10. print("The mileage is 25kmph ")
11. t= Tesla ()
12. t.mileage() //The mileage is 30kmph
13. s = Suzuki()
14. s.mileage() //The mileage is 25kmph

Below are the points which we should remember about abstract base class in Python.

* An Abstract class can contain the both method normal and abstract method.
* An Abstract cannot be instantiated; we cannot create objects for abstract class.

| **Object-oriented Programming** | **Procedural Programming** |
| --- | --- |
| Object-oriented programming is the problem-solving approach and used where computation is done by using objects. | Procedural programming uses a list of instructions to do computation step by step. |
| It makes development and maintenance easier. | In procedural programming, It is not easy to maintain the codes when the project becomes lengthy. |
| It simulates the real world entity. So real-world problems can be easily solved through oops. | It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions. |
| It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere. | Procedural language doesn't provide any proper way for data binding, so it is less secure. |
| Ex- object-oriented programming languages is C++, Java, .Net, Python, C#, etc. | Ex: procedural languages are: C, Fortran, Pascal, VB etc. |