**Python read csv file**

**CSV File**

A **csv** stands for "comma separated values", which is defined as a simple file format that uses specific structuring to arrange tabular data. It stores tabular data such as spreadsheet or database in plain text and has a common format for data interchange. A **csv** file opens into the excel sheet, and the rows and columns data define the standard format.

**Python CSV Module Functions**

The CSV module work is used to handle the CSV files to read/write and get data from specified columns. There are different types of CSV functions, which are as follows:

* **csv.field\_size\_limit -** It returns the current maximum field size allowed by the parser.
* **csv.get\_dialect -** It returns the dialect associated with a name.
* **csv.list\_dialects -** It returns the names of all registered dialects.
* **csv.reader -** It read the data from a csv file
* **csv.register\_dialect -** It associates dialect with a name. The name must be a string or a Unicode object.
* **csv.writer -** It writes the data to a csv file
* **o csv.unregister\_dialect -** It deletes the dialect which is associated with the name from the dialect registry. If a name is not a registered dialect name, then an error is being raised.
* **csv.QUOTE\_ALL -** It instructs the writer objects to quote all fields. csv.QUOTE\_MINIMAL - It instructs the writer objects to quote only those fields which contain special characters such as quotechar, delimiter, etc.
* **csv.QUOTE\_NONNUMERIC -** It instructs the writer objects to quote all the non-numeric fields.
* **csv.QUOTE\_NONE -** It instructs the writer object never to quote the fields.

**Reading CSV files**

Python provides various functions to read csv file. We are describing few method of reading function.

* **Using csv.reader() function**

In Python, the **csv.reader()** module is used to read the csv file. It takes each row of the file and makes a list of all the columns.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

We have taken a txt file named as python.txt that have default delimiter **comma(,)** with the following data:

1. name,department,birthday month
2. Parker,Accounting,November
3. Smith,IT,October

**Example**

1. import csv
2. with open('python.csv') as csv\_file:
3. csv\_reader = csv.reader(csv\_file, delimiter=',')
4. line\_count = 0
5. for row in csv\_reader:
6. if line\_count == 0:
7. print(f'Column names are {", ".join(row)}')
8. line\_count += 1

**Output:**

Column names are name, department, birthday month

Parker works in the Accounting department, and was born in November.

Smith works in the IT department, and was born in October.

Processed 3 lines.

In the above code, we have opened 'python.csv' using the **open()** function. We used **csv.reader()** function to read the file, that returns an iterable reader object. The **reader** object have consisted the data and we iterated using **for** loop to print the content of each row

**Read a CSV into a Dictionar**

We can also use **DictReader()** function to read the csv file directly into a dictionary rather than deal with a list of individual string elements.

Again, our input file, python.txt is as follows:

1. name,department,birthday month
2. Parker,Accounting,November
3. Smith,IT,October

**Example**

1. import csv
2. with open('python.txt', mode='r') as csv\_file:
3. csv\_reader = csv.DictReader(csv\_file)
4. line\_count = 0
5. for row in csv\_reader:
6. if line\_count == 0:
7. print(f'The Column names are as follows {", ".join(row)}')
8. line\_count += 1
9. print(f'\t{row["name"]} works in the {row["department"]} department, and was born in {row["birthday month"]}.')
10. line\_count += 1
11. print(f'Processed {line\_count} lines.')

**Output:**

The Column names are as follows name, department, birthday month

Parker works in the Accounting department, and was born in November.

Smith works in the IT department, and was born in October.

Processed 3 lines.

**Reading csv files with Pandas**

The Pandas is defined as an open-source library which is built on the top of the NumPy library. It provides fast analysis, data cleaning, and preparation of the data for the user.

Reading the csv file into a pandas **DataFrame** is quick and straight forward. We don't need to write enough lines of code to open, analyze, and read the csv file in pandas and it stores the data in **DataFrame**.

Here, we are taking a slightly more complicated file to read, called hrdata.csv, which contains data of company employees.

1. Name,Hire Date,Salary,Leaves Remaining
2. John Idle,08/15/14,50000.00,10
3. Smith Gilliam,04/07/15,65000.00,8
4. Parker Chapman,02/21/14,45000.00,10
5. Jones Palin,10/14/13,70000.00,3
6. Terry Gilliam,07/22/14,48000.00,7
7. Michael Palin,06/28/13,66000.00,8

**Example**

1. import pandas
2. df = pandas.read\_csv('hrdata.csv')
3. print(df)

In the above code, the three lines are enough to read the file, and only one of them is doing the actual work, i.e., pandas.read\_csv()

**Output:**

Name Hire Date Salary Leaves Remaining

0 John Idle 03/15/14 50000.0 10

1 Smith Gilliam 06/01/15 65000.0 8

2 Parker Chapman 05/12/14 45000.0 10

3 Jones Palin 11/01/13 70000.0 3

4 Terry Gilliam 08/12/14 48000.0 7

5 Michael Palin 05/23/13 66000.0 8

It is defined as a construct that allows you to create, store, and re-use various formatting parameters. It supports several attributes; the most frequently used are:

* **Dialect.delimiter:** This attribute is used as the separating character between the fields. The default value is a comma (,).
* **Dialect.quotechar:** This attribute is used to quote fields that contain special characters.
* **Dialect.lineterminator:** It is used to create new lines, and the default value is '\r\n'.

Let's write the following data to a CSV File.

1. data = [{'Rank': 'B', 'first\_name': 'Parker', 'last\_name': 'Brian'},
2. {'Rank': 'A', 'first\_name': 'Smith', 'last\_name': 'Rodriguez'},
3. {'Rank': 'C', 'first\_name': 'Tom', 'last\_name': 'smith'},
4. {'Rank': 'B', 'first\_name': 'Jane', 'last\_name': 'Oscar'},
5. {'Rank': 'A', 'first\_name': 'Alex', 'last\_name': 'Tim'}]

### Example -

1. import csv
3. with open('Python.csv', 'w') as csvfile:
4. fieldnames = ['first\_name', 'last\_name', 'Rank']
5. writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
7. writer.writeheader()
8. writer.writerow({'Rank': 'B', 'first\_name': 'Parker', 'last\_name': 'Brian'})
9. writer.writerow({'Rank': 'A', 'first\_name': 'Smith',
10. 'last\_name': 'Rodriguez'})
11. writer.writerow({'Rank': 'B', 'first\_name': 'Jane', 'last\_name': 'Oscar'})
12. writer.writerow({'Rank': 'B', 'first\_name': 'Jane', 'last\_name': 'Loive'})
14. print("Writing complete")

**Output:**

Writing complete

It returns the file named as 'Python.csv' that contains the following data:

1. first\_name,last\_name,Rank
2. Parker,Brian,B
3. Smith,Rodriguez,A
4. Jane,Oscar,B
5. Jane,Loive,B

## Write a CSV into a Dictionary

We can also use the class **DictWriter** to write the CSV file directly into a dictionary.

A file named as python.csv contains the following data:

Parker, Accounting, November

Smith, IT, October

### Example -

1. import csv
2. with open('python.csv', mode='w') as csv\_file:
3. fieldnames = ['emp\_name', 'dept', 'birth\_month']
4. writer = csv.DictWriter(csv\_file, fieldnames=fieldnames)
5. writer.writeheader()
6. writer.writerow({'emp\_name': 'Parker', 'dept': 'Accounting', 'birth\_month': 'November'})
7. writer.writerow({'emp\_name': 'Smith', 'dept': 'IT', 'birth\_month': 'October'})

**Output:**

emp\_name,dept,birth\_month

Parker,Accounting,November

Smith,IT,October

## Writing CSV Files Using Pandas

Pandas is defined as an open source library which is built on the top of Numpy library. It provides fast analysis, data cleaning and preparation of the data for the user.

It is as easy as reading the CSV file using pandas. You need to create the DataFrame, which is a two-dimensional, heterogeneous tabular data structure and consists of three main components- data, columns, and rows. Here, we take a slightly more complicated file to read, called hrdata.csv, which contains data of company employees.

1. Name,Hire Date,Salary,Leaves Remaining
2. John Idle,08/15/14,50000.00,10
3. Smith Gilliam,04/07/15,65000.00,8
4. Parker Chapman,02/21/14,45000.00,10
5. Jones Palin,10/14/13,70000.00,3
6. Terry Gilliam,07/22/14,48000.00,7
7. Michael Palin,06/28/13,66000.00,8

### Example -

1. import pandas
2. df = pandas.read\_csv('hrdata.csv',
3. index\_col='Employee',
4. parse\_dates=['Hired'],
5. header=0,
6. names=['Employee', 'Hired', 'Salary', 'Sick Days'])
7. df.to\_csv('hrdata\_modified.csv')

**Output:**

Employee, Hired, Salary, Sick Days

John Idle, 2014-03-15, 50000.0,10

Smith Gilliam, 2015-06-01, 65000.0,8

Parker Chapman, 2014-05-12, 45000.0,10

Jones Palin, 2013-11-01, 70000.0,3

Terry Gilliam, 2014-08-12 , 48000.0,7

Michael Palin, 2013-05-23, 66000.0,8

**Python read excel file**

Excel is a spreadsheet application which is developed by Microsoft. It is an easily accessible tool to organize, analyze, and store the data in tables. It is widely used in many different applications all over the world. From Analysts to CEOs, various professionals use Excel for both quick stats and serious data crunching.

**Excel Documents**

An Excel spreadsheet document is called a workbook which is saved in a file with **.xlsx** extension. The first row of the spreadsheet is mainly reserved for the header, while the first column identifies the sampling unit. Each workbook can contain multiple sheets that are also called a worksheets. A box at a particular column and row is called a cell, and each cell can include a number or text value. The grid of cells with data forms a sheet.

The active sheet is defined as a sheet in which the user is currently viewing or last viewed before closing Excel.

**Reading from an Excel file**

First, you need to write a command to install the **xlrd** module.

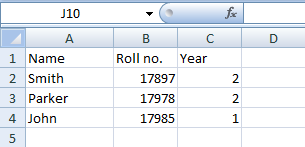
1. pip install xlrd

**Creating a Workbook**

A workbook contains all the data in the excel file. You can create a new workbook from scratch, or you can easily create a workbook from the excel file that already exists.

**Input File**

We have taken the snapshot of the workbook.



**Code**

1. # Import the xlrd module
2. import xlrd
4. # Define the location of the file
5. loc = ("path of file")
7. # To open the Workbook
8. wb = xlrd.open\_workbook(loc)
9. sheet = wb.sheet\_by\_index(0)
11. # For row 0 and column 0
12. sheet.cell\_value(0, 0)

**Explanation:** In the above example, firstly, we have imported the xlrd module and defined the location of the file. Then we have opened the workbook from the excel file that already exists.

**Reading from the Pandas**

Pandas is defined as an open-source library which is built on the top of the NumPy library. It provides fast analysis, data cleaning, and preparation of the data for the user and supports both xls and xlsx extensions from the URL.

It is a python package which provides a beneficial data structure called a data frame.

**Example**

1. Example -
2. import pandas as pd
4. # Read the file
5. data = pd.read\_csv(".csv", low\_memory=False)
7. # Output the number of rows
8. print("Total rows: {0}".format(len(data)))
10. # See which headers are available
11. print(list(data))

**Reading from the openpyxl**

First, we need to install an openpyxl module using pip from the command line.

1. pip install openpyxl

After that, we need to import the module.

We can also read data from the existing spreadsheet using openpyxl. It also allows the user to perform calculations and add content that was not part of the original dataset.

**Example**

1. import openpyxl
2. my\_wb = openpyxl.Workbook()
3. my\_sheet = my\_wb.active
4. my\_sheet\_title = my\_sheet.title
5. print("My sheet title: " + my\_sheet\_title)

**Output:**

My sheet title: Sheet

To learn more about openpyxl, visit our complete tutorial [Click Here](https://www.javatpoint.com/python-openpyxl). We have discussed essential detail in this tutorial.

# Python Write Excel File

The Python write excel file is used to perform the multiple operations on a spreadsheet using the **xlwt** module. It is an ideal way to write data and format information to files with .xls extension.

If you want to write data to any file and don't want to go through the trouble of doing everything by yourself, then you can use a for loop to automate the whole process a little bit.

## Write Excel File Using xlsxwriter Module

We can also write the excel file using the **xlsxwriter** module. It is defined as a Python module for writing the files in the XLSX file format. It can also be used to write text, numbers, and formulas to multiple worksheets. Also, it supports features such as charts, formatting, images, page setup, auto filters, conditional formatting, and many others.

We need to use the following command to install xlsxwriter module:

1. pip install xlsxwriter

#### Note- Throughout XlsxWriter, rows, and columns are zero-indexed. The first cell in a worksheet is listed as, A1 is (0,0), B1 is (0,1), A2 is (1,0), B2 is (1,1)......,and so on.

## Write Excel File Using openpyxl Module

It is defined as a package which is generally recommended if you want to read and write .xlsx, xlsm, xltx, and xltm files. You can check it by running **type(wb)**.

The load\_workbook() function takes an argument and returns a workbook object, which represents the file. Make sure that you are in the same directory where your spreadsheet is located. Otherwise, you will get an error while importing.

You can easily use a for loop with the help of the range() function to help you to print out the values of the rows that have values in column 2. If those particular cells are empty, you will get None.

## Writing data to Excel files with xlwt

You can use the xlwt package, apart from the XlsxWriter package to create the spreadsheets that contain your data. It is an alternative package for writing data, formatting information, etc. and ideal for writing the data and format information to files with .xls extension. It can perform multiple operations on the spreadsheet.

It supports features such as formatting, images, charts, page setup, auto filters, conditional formatting, and many others.

Pandas have excellent methods for reading all kinds of data from excel files. We can also import the results back to pandas.

## Writing Files with pyexcel

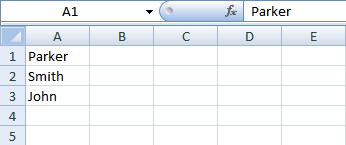
You can easily export your arrays back to a spreadsheet by using the save\_as() function and pass the array and name of the destination file to the dest\_file\_name argument.

It allows us to specify the delimiter and add dest\_delimiter argument. You can pass the symbol that you want to use as a delimiter in-between " ".

**Code**

1. # import xlsxwriter module
2. import xlsxwriter
4. book = xlsxwriter.Book('Example2.xlsx')
5. sheet = book.add\_sheet()
7. # Rows and columns are zero indexed.
8. row = 0
9. column = 0
11. content = ["Parker", "Smith", "John"]
13. # iterating through the content list
14. for item in content :
16. # write operation perform
17. sheet.write(row, column, item)
19. # incrementing the value of row by one with each iterations.
20. row += 1
22. book.close()

**Output:**



# Python Assert Keyword

Python assert keyword is defined as a debugging tool that tests a condition. The Assertions are mainly the assumption that asserts or state a fact confidently in the program. For example, while writing a division function, the divisor should not be zero, and you assert that the divisor is not equal to zero.

It is merely a Boolean expression that has a condition or expression checks if the condition returns true or false. If it is true, the program does not do anything, and it moves to the next line of code. But if it is false, it raises an **AssertionError** exception with an optional error message.

The main task of assertions is to inform the developers about unrecoverable errors in the program like "file not found", and it is right to say that assertions are internal self-checks for the program. It is the most essential for the testing or quality assurance in any application development area. The syntax of the assert keyword is given below.

**Syntax**

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. assert condition, error\_message(optional)

## Why Assertion is used

It is a debugging tool, and its primary task is to check the condition. If it finds that the condition is true, it moves to the next line of code, and If not, then stops all its operations and throws an error. It points out the error in the code.

## Where Assertion in Python used

* Checking the outputs of the functions.
* Used for testing the code.
* In checking the values of arguments.Checking the valid input.

### Example1

This example shows the working of assert with the error message.

1. def avg(scores):
2. assert len(scores) != 0,"The List is empty."
3. return sum(scores)/len(scores)
5. scores2 = [67,59,86,75,92]
6. print("The Average of scores2:",avg(scores2))
8. scores1 = []
9. print("The Average of scores1:",avg(scores1))

**Output:**

The Average of scores2: 75.8

AssertionError: The List is empty.

**Explanation:** In the above example, we have passed a non-empty list **scores2** and an empty list **scores1** to the **avg()** function. We received an output for **scores2** list successfully, but after that, we got an error **AssertionError: List is empty**. The assert condition is satisfied by the **scores2** list and lets the program continue to run. However, **scores1** doesn't satisfy the condition and gives an AssertionError.

### Example2:

This example shows the "Divide by 0 error" in the console.

1. # initializing number
2. x = 7
3. y = 0
4. # It uses assert to check for 0
5. print ("x / y value is : ")
6. assert y != 0, "Divide by 0 error"
7. print (x / y)

**Output:**

x / y value is :

### Runtime Exception :

Traceback (most recent call last):

File "main.py", line 6, in <module>

assert y != 0, "Divide by 0 error"

AssertionError: Divide by 0 error

**Explanation:**

In the above example, we have initialized an integer variable, i.e., x=7, y=0, and try to print the value of x/y as an output. The Python interpreter generated a Runtime Exception because of the assert keyword found the divisor as zero then displayed **"Divide by 0 error"** in the console.

**Python List Comprehension**

Python is known for helping us produce code that is elegant, simple to write, and reads almost as well as plain English. List comprehension is one of the language's most distinguishing features, allowing us to develop sophisticated functionality with just one line of code. On the other hand, many Python writers struggle to fully utilize the more complex aspects of list comprehension. Sometimes programmers may overuse them, resulting in much less efficient and difficult-to-read code.

**Using List Comprehension**

1. newlist = [expression for item in iterable if condition == True]

Here we are showing basic use of list comprehension.

**Code**

1. #using for loop to iterate through items in list
2. numbers = [3, 5, 1, 7, 3, 9]
3. num = []
5. for n in numbers:
6. num.append(n\*\*2)
8. print(num)

**Output:**

[9, 25, 1, 49, 9, 81]

All of this can be accomplished with only single line of code using list comprehension.

**Code**

1. #using list comprehension to iterate through list items
2. numbers = [3, 5, 1, 7, 3, 9]
4. num = [n\*\*2 for n in numbers]
6. print(num)

**Output:**

[9, 25, 1, 49, 9, 81]

**Benefits of Using List Comprehensions**

Loops and maps are typically regarded as more Pythonic than list comprehensions. But, rather than taking that judgment at face value, it's worth considering the advantages of utilizing a list comprehension in Python over the alternatives. We'll learn about a couple of cases when the alternatives are preferable options later on.

One of the most important advantages of utilizing a list comprehension in Python is that it is a single tool that can be used in various circumstances. We don't need to adopt a new strategy for each situation. List comprehensions may be leveraged for mapping or filtering and basic list generation.

List comprehensions are regarded as Pythonic, as Python emphasizes simple, effective tools that can be used in many scenarios. As a bonus, we won't have to remember the appropriate order of parameters when using a list comprehension in Python, as we would when calling map().

List comprehensions are easier to read and grasp than loops since they are more declarative. We must concentrate on how exactly the list is constructed while using loops. We must manually build an empty list, then loop over the list's entries, and add each one to the list's end. Instead, using a list comprehension in Python, we can concentrate on what we want to put in the list and allow Python to handle the list generation.

**Code**

1. # Import module to keep track of time
2. import time
4. # defining function to execute for loop
5. def for\_loop(num):
6. l = []
7. for i in range(num):
8. l.append(i + 10)
9. return l
11. # defining function to execute list comprehension
12. def list\_comprehension(num):
13. return [i + 10 for i in range(num)]
15. # Giving values to the functions
17. # Calculating time taken by for loop
18. start = time.time()
19. for\_loop(10000000)
20. end = time.time()
22. print('Time taken by for loop:', (end - start))
24. # Calculating time taken by list comprehension
25. start = time.time()
26. list\_comprehension(10000000)
27. end = time.time()
29. print('Time taken by list comprehension:', (end - start))

**Output:**

Time taken by for loop: 7.005999803543091

Time taken by list comprehension: 2.822999954223633

**Using List Comprehension to Iterate through String**

List comprehension can be used in case of strings also as they are iterables too.

**Code**

1. letters = [ alpha for alpha in 'javatpoint' ]
2. print( letters)

**Output:**

['j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't']

**Using Conditions in List Comprehension**

Conditional statements can be used by list comprehensions to change existing lists (or other tuples). We'll make a list with mathematical operators, numbers, and a range of values.

**Code**

1. number\_list = [ num for num in range(30) if num % 2 != 0]
2. print(number\_list)

**Output:**

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

**Nested List Comprehensions**

Nested List Comprehensions are similar to nested for loops in that they are a list comprehension inside another list comprehension. The programme that implements nested loop is as follows:

**Code**

1. nested\_list = []
3. for \_ in range(3):
5. # Append an empty sublist inside the list
6. nested\_list.append([])
8. for \_\_ in range(5):
9. nested\_list[\_].append(\_\_ + \_)
11. print(nested\_list)

**Output:**

[[0, 1, 2, 3, 4], [1, 2, 3, 4, 5], [2, 3, 4, 5, 6]]

The same result may now be created in less lines of code by utilizing layered list comprehensions.

**Code**

1. # Nested list comprehension
2. nested\_list = [[\_ + \_\_ for \_ in range(5)] for \_\_ in range(3)]
4. print(nested\_list)

**Output:**

[[0, 1, 2, 3, 4], [1, 2, 3, 4, 5], [2, 3, 4, 5, 6]]

List comprehension is a powerful tool for describing and creating new lists based on existing ones. In general, list comprehension is lighter and easier to use than traditional list construction functions and loops. To provide user-friendly code, we should avoid writing large codes for list comprehensions. Every interpretation of the list or other iterables can be recast in a for loop, but not all the for loops can be rebuilt in the framework of list comprehension.

# Convert Roman Number to Decimal (Integer) | Write Python Program to Convert Roman to Integer

In this tutorial, we will write the Python program to convert the Roman numbers into the integer. It is a popular problem was asked by the tech giant Amazon, Facebook in the interview. Let's see the problem statement and implementation of the solution.

### Problem Statement

A roman number is given as a string; the task is to convert the corresponding integer value. The symbols are given below for reference.

|  |  |
| --- | --- |
| **Symbols** | **Values** |
| I | 1 |
| IV | 4 |
| V | 5 |
| IX | 9 |
| X | 10 |
| XL | 40 |
| L | 50 |
| XC | 90 |
| C | 100 |
| CD | 400 |
| D | 500 |
| CM | 900 |
| M | 1000 |

**Example 1:**

**Input:**s = VI

**Output:** 6

**Example 2:**

**Input:** X**Output:** 10XL is a Roman symbol which represents 40

### Solution Approach

**Algorithm**

1. First, split the Roman numeral string into roman symbols.
2. We can the separate symbol now convert each symbol of Roman Numerals into the value it represents.
3. Picking up a value starting from index 0.

* If the current value of the symbol is greater than or equal to the next symbol, then add this value to the returning total.
* Else subtract this value by adding the value of the next symbol to the running total.

Let's implement the algorithm to the Python program.

## Program

1. def rom\_value(r):
2. if (r == 'I'):
3. return 1
4. if (r == 'V'):
5. return 5
6. if (r == 'X'):
7. return 10
8. if (r == 'L'):
9. return 50
10. if (r == 'C'):
11. return 100
12. if (r == 'D'):
13. return 500
14. if (r == 'M'):
15. return 1000
16. return -1
17. def romanToDecimal(str):
18. res = 0
19. i = 0
21. while (i < len(str)):

24. n1 = rom\_value(str[i])
26. if (i + 1 < len(str)):

29. n2 = rom\_value(str[i + 1])
31. # Comparing both rom\_values
32. if (n1 >= n2):
34. res = res + str1
35. i = i + 1
36. else:
38. # rom\_value of current symbol is greater
39. # or equal to the next symbol
40. res = res + str2 - str1
41. i = i + 2
42. else:
43. res = res + str1
44. i = i + 1
46. return res
48. print(romanToDecimal("VII"))

**Output:**

7

**Explanation**

In the above code, we define a **rom\_value()** function which returns corresponding to the symbol. Next we define the **romanTointeger()** method which converts the value roman value to integer. In the **romanToInteger()** method,

* We have defined the res and i variable with 0.
* The while iterated till i is smaller than the length of the string.
* We converted the first character into an integer and stored it into **n1.** Then, use the condition to check the i+1th element smaller than the length of the string.
* If it returns true, it converts into an integer and is stored in **n2.**
* Compare n1 and n2; if n1 is greater than the n2, add it to the res and increase the ith value by one.
* If it returns false, then subtract the n2 to n1 and make the increment of 2 in i.
* If first if condition returns false add into the res and make increment to i

## Complexity Analysis

**Time Complexity:** O(n), where n is the length of the string. Only one traversal of the string is required.

**Space Complexity:** O(1). As no extra space is required

**Python Collection Module**

The Python collection module is defined as a container that is used to store collections of data, for example - list, dict, set, and tuple, etc. It was introduced to improve the functionalities of the built-in collection containers.

Python collection module was first introduced in its 2.4 release.

There are different types of collection modules which are as follows:

**namedtuple()**

The Python **namedtuple()** function returns a tuple-like object with names for each position in the tuple. It was used to eliminate the problem of remembering the index of each field of a tuple object in ordinary tuples.

**Examples**

1. pranshu = ('James', 24, 'M')
2. print(pranshu)

**Output:**

('James', 24, 'M')

**OrderedDict()**

The Python OrderedDict() is similar to a dictionary object where keys maintain the order of insertion. If we try to insert key again, the previous value will be overwritten for that key.

**Example**

1. import collections
2. d1=collections.OrderedDict()
3. d1['A']=10
4. d1['C']=12
5. d1['B']=11
6. d1['D']=13
8. for k,v in d1.items():
9. print (k,v)

**Output:**

A 10

C 12

B 11

D 13

**defaultdict()**

The Python defaultdict() is defined as a dictionary-like object. It is a subclass of the built-in dict class. It provides all methods provided by dictionary but takes the first argument as a default data type.

**Example**

1. from collections import defaultdict
2. number = defaultdict(int)
3. number['one'] = 1
4. number['two'] = 2
5. print(number['three'])

**Output:**

0

**Counter()**

The Python Counter is a subclass of dictionary object which helps to count hashable objects.

**Example**

1. from collections import Counter
2. c = Counter()
3. list = [1,2,3,4,5,7,8,5,9,6,10]
4. Counter(list)
5. Counter({1:5,2:4})
6. list = [1,2,4,7,5,1,6,7,6,9,1]
7. c = Counter(list)
8. print(c[1])

**Output:**

3

**deque()**

The Python **deque()** is a double-ended queue which allows us to add and remove elements from both the ends.

**Example**

1. from collections import deque
2. list = ["x","y","z"]
3. deq = deque(list)
4. print(deq)

**Output:**

deque(['x', 'y', 'z'])

**Chainmap Objects**

A **chainmap** class is used to groups multiple dictionary together to create a single list. The linked dictionary stores in the list and it is public and can be accessed by the map attribute. Consider the following example.

**Example**

1. from collections import ChainMap
2. baseline = {'Name': 'Peter', 'Age': '14'}
3. adjustments = {'Age': '14', 'Roll\_no': '0012'}
4. print(list(ChainMap(adjustments, baseline)))

**Output:**

['Name', 'Age', 'Roll\_no' ]

**UserDict Objects**

The UserDict behaves as a wrapper around the dictionary objects. The dictionary can be accessed as an attribute by using the **UserDict** object. It provides the easiness to work with the dictionary.

It provides the following attribute.

**data** - A real dictionary used to store the contents of the UserDict class.

**UserList Objects**

The UserList behaves as a wrapper class around the list-objects. It is useful when we want to add new functionality to the lists. It provides the easiness to work with the dictionary.

It provides the following attribute.

**data** - A real list is used to store the contents of the User class.

**UserString Objects**

The **UserList** behaves as a wrapper class around the list objects. The dictionary can be accessed as an attribute by using the **UserString** object. It provides the easiness to work with the dictionary.

It provides the following attribute.

**data** - A real **str** object is used to store the contents of the UserString class.

# Python Math Module

Mathematical calculations may occasionally be required when dealing with certain fiscal or rigorous scientific tasks. Python has a math module that can handle these complex calculations. Both simple mathematical calculations like addition (+), and subtraction (-), and advanced mathematical calculations like trigonometric operations, and logarithmic operations can be performed by the functions in the math module.

This tutorial teaches us about applying the math module from fundamentals to more advanced concepts with the support of easy examples to understand the concepts fully. We have included the list of all built-in functions defined in this module for better understanding.

## What is Math Module in Python?

Python has a built-in math module. It is a standard module, so we don't need to install it separately. We only have to import it into the program we want to use. We can import the module, like any other module of Python, using import math to implement the functions to perform mathematical operations.

Since the source code of this module is in the C language, it provides access to the functionalities of the underlying C library. For instance,

**Code**

1. # This program will show the calculation of square root using the math module
2. # importing the math module
3. import math
4. print(math.sqrt( 9 ))

**Output:**

3.0

This Python module does not accept complex data types. The more complicated equivalent is the cmath module.

We can, for example, calculate all trigonometric ratios for any given angle using the built-in functions in the math module. We must provide angles in radians to these trigonometric functions (sin, cos, tan, etc.). However, we are accustomed to measuring angles in terms of degrees. The math module provides two methods to convert angles from radians to degrees and vice versa.

## Constants in Math Module

The value of numerous constants, including pi and tau, is provided in the math module so that we do not have to remember them. Using these constants eliminates the need to precisely and repeatedly write down the value of each constant. The math module includes the following constants:

1. Euler's Number
2. Tau
3. Infinity
4. Pi
5. Not a Number (NaN)

Let's go over each of them one by one.

### Euler's Number

The value 2.71828182845 of Euler's number is returned by the math.e constant.

**Syntax of this is:**

1. math.e

**Code**

1. # importing the required library
2. import math
4. # printing the value of Euler's number using the math module
5. print( "The value of Euler's Number is: ", math.e )

**Output:**

The value of Euler's Number is: 2.718281828459045

### Tau

The ratio of a circle's circumference to its radius is known as tau. The value tau returned by the tau constant is 6.283185307179586.

**Syntax of this is:**

1. math.tau

**Code**

1. # Importing the required library
2. import math
4. # Printing the value of tau using math module
5. print ( "The value of Tau is: ", math.tau )

**Output:**

The value of Tau is: 6.283185307179586

### Infinity

Infinity refers to anything limitless or never-ending in both directions of the actual number line. Numbers cannot adequately represent it. The math.inf returns positive infinity constant. We can use -math.inf to print negative infinity.

**Syntax of this is:**

1. math.inf

**Code**

1. # Importing the required library
2. import math
4. # Printing the value of positive infinity using the math module
5. print( math.inf )
7. # Printing the value of negative infinity using the math module
8. print( -math.inf )

**Output:**

inf

-inf

Further, we are comparing a very large floating-point number with positive and negative infinity values.

**Code**

1. # Importing the required library
2. import math
4. # comparing the value of infinity
5. print( math.inf > 10e109 )
6. print( -math.inf < -10e109 )

**Output:**

True

True

### Pi

Pi is known to everyone. It is mathematically represented as either the fraction 22/7 or the decimal number 3.14. math.pi gives the most accurate value of pi.

**Syntax of this is:**

1. math.pi

**Code**

1. # Importing the required library
2. import math
4. # Printing the value of pi using the math module
5. print( "The value of pi is ", math.pi )

**Output:**

The value of pi is 3.141592653589793

Let us calculate the circumference of a circle.

**Code**

1. # Importing the required library
2. import math
4. # radius of the circle
5. r = 4
7. # value of pi
8. pi\_value = math.pi
10. # circumference of the circle
11. print(2 \* pi\_value \* r)

**Code**

25.132741228718345

### NaN

The math.nan gives us a floating-point nan (Not a Number) value. This amount is not a valid numeric value. Float("nan") and the nan constant are comparable.

**Code**

1. # Importing the required library
2. import math
4. # Printing the value of nan using the math module
5. print( math.nan )

**Output:**

nan

## Mathematical Operations with Math Module

The functions that are required in representation theory and number theory, such as calculating the factorial of an integer, will be covered in this part.

### Calculating the Ceiling and the Floor Value

The terms "ceiling value" and "floor value" refer to the smallest integral value larger than the number and the largest integral value less than the number, respectively. The ceil() and floor() methods simplify calculating this.

**Code**

1. # Python program to show how to use floor() and ceil() functions.
3. # importing the math module
4. import math
6. x = 4.346
8. # returning the ceiling value of 4.346
9. print("The ceiling value of 4.346 is : ", end="")
10. print( math.ceil(x) )
12. # returning the floor value of 4.346
13. print("The floor value of 4.346 is : ", end="")
14. print( math.floor(x) )

**Output:**

The ceiling value of 4.346 is : 5

The floor value of 4.346 is : 4

### Calculating the Factorial of the Number

We may determine the factorial of a given integer in a one-liner code by using the math.factorial() function. The Python interpreter will send a message if the given number is not integral.

**Code**

1. # Python program to show how to use function() functions.
3. # importing the math module
4. import math
6. x = 6
8. # returning the factorial of 6
9. print( "The factorial of 6 is : ", math.factorial(x) )
11. # passing a non integral number
12. try:
13. print( "The factorial of 6.5 in: ", math.factorial(6.5) )
14. except:
15. print( "Cannot calculate factorial of a non-integral number" )

**Output:**

The factorial of 6 is : 720

Cannot calculate factorial of a non-integral number

### Calculating the Absolute Value

The method math.fabs() returns the absolute number of the number given to the function.

**Code**

1. # Python program to show how to use fabs() functions.
3. # importing the math module
4. import math
6. x = -45
8. # returning x's absolute value.
9. print( "The absolute value of -45 is: ", math.fabs(x) )

**Output:**

The absolute value of -45 is: 45.0

### Calculating the Exponential

x to the power of e, often known as the exponential of a number x, is calculated using the exp() function.

**Code**

1. # Python program to show how to use the exp() function.
3. # importing the math module
4. import math
6. # declaring some value
7. num1 = 4
8. num2 = -3
9. num3 = 0.00
11. # passing above values to the exp() function
12. print( f"The exponenetial value of {num1} is: ", math.exp(num1) )
13. print( f"The exponenetial value of {num2} is: ", math.exp(num2) )
14. print( f"The exponenetial value of {num3} is: ", math.exp(num3) )

**Output:**

The exponenetial value of 4 is: 54.598150033144236

The exponenetial value of -3 is: 0.049787068367863944

The exponenetial value of 0.0 is: 1.0

### Calculating the Power of a Number

x\*\*y is computed via the pow() function. This function calculates the value of the power after converting its inputs into floats.

**Code**

1. # Python program to show how to use the pow() function.
3. # importing the math module
4. import math
6. x = 4
7. y = 5
8. # returning x to the power of y.
9. print( f"The value of {x} to the power of {y} is: ", math.pow(x,y) )

**Output:**

The value of 4 to the power of 5 is: 1024.0

### Calculating Sine, Cosine, and Tangent

The values of sine, cosine, and tangent of an angle, which are supplied as an input to the function, are returned by the sin(), cos(), and tan() methods. This function expects a value that is provided in radians.

**Code**

1. # Python program to show how to use the sin(), cos(), tan() function.
3. # importing the math module
4. import math
6. angle = math.pi / 4
8. # returning the sine of pi/4
9. print( "The sine of pi/4 is : ", math.sin( angle ) )
11. # returning the cosine of pi/4
12. print( "The cosine of pi/4 is : ", math.cos( angle ) )
14. # returning the tangent of pi/4
15. print("The tangent of pi/4 is : ", math.tan( angle ))

**Output:**

The sine of pi/4 is : 0.7071067811865475

The cosine of pi/4 is : 0.7071067811865476

The tangent of pi/4 is : 0.9999999999999999

## The dir( ) Function

A sorted list of strings comprising the identifiers of the functions defined by a module is what the built-in method dir() delivers.

The list includes the names of modules, each specified constants, functions, and methods. Here is a straightforward illustration:

**Code**

1. # Importing the math module
2. import math
4. functions = dir(math)
5. print( functions )

**Output:**

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']

## Description of all the Functions in Python Math Module

Here is a list of all the properties and functions specified in the math module, along with a brief description of what each one does.

|  |  |
| --- | --- |
| **Function** | **Description** |
| **ceil(x)** | The lowest integer bigger than or equal to x is returned. |
| **copysign(x, y)** | gives x back with the sign of y. |
| **fabs(x)** | gives x's absolute value back. |
| **factorial(x)** | provides the x factorial back. |
| **floor(x)** | gives back the biggest integer that is less than or equal to x. |
| **fmod(x, y)** | returns the leftover value after dividing x by y. |
| **frexp(x)** | returns the pair of the mantissa and exponent of x. (m, e) |
| **fsum(iterable)** | returns the iterable's correct floating point sum of all values. |
| **isfinite(x)** | If x is neither an infinity nor a NaN, it returns True (Not a Number) |
| **isinf(x)** | If x is a positive or negative infinity, it returns True. |
| **isnan(x)** | If x is a NaN, it returns True. |
| **ldexp(x, i)** | gives back x \* (2\*\*i). |
| **modf(x)** | gives x's fractional and integer components back. |
| **trunc(x)** | x's shortened integer value is returned. |
| **exp(x)** | delivers e\*\*x |
| **expm1(x)** | yields e\*\*x - 1 |
| **log(x[, b])** | gives back the x logarithm in base b. (defaults to e) |
| **log1p(x)** | the natural logarithm of 1 + x is returned. |
| **log2(x)** | gives x's base-2 logarithm back. |
| **log10(x)** | provides x's base-10 logarithm. |
| **pow(x, y)** | gives x raised to the power of y back. |
| **sqrt(x)** | gives x's square root back. |
| **acos(x)** | gives the arc cosine of x back. |
| **asin(x)** | gives the arc sine of x back. |
| **atan(x)** | gives the arc tangent of x back. |
| **atan2(y, x)** | gives back atan(y / x). |
| **cos(x)** | returns the x's cosine. |
| **hypot(x, y)** | returns sqrt(x\*x + y\*y), the Euclidean norm. |
| **sin(x)** | gives the sine of x back. |
| **tan(x)** | gives the tangent of x back. |
| **degrees(x)** | Angle x is transformed from radians to degrees. |
| **radians(x)** | Angle x is transformed from degrees to radians. |
| **acosh(x)** | x's inverse hyperbolic cosine is returned. |
| **asinh(x)** | x's inverse hyperbolic sine is returned. |
| **atanh(x)** | x's inverse hyperbolic tangent is returned. |
| **cosh(x)** | gives x's hyperbolic cosine. |
| **sinh(x)** | gives x's hyperbolic cosine. |
| **tanh(x)** | gives x's hyperbolic tangent back. |
| **erf(x)** | the error function at x is returned. |
| **erfc(x)** | a function that gives the complementary error at x |
| **gamma(x)** | the Gamma function at x is returned. |
| **lgamma(x)** | gives the natural logarithm of the gamma function's absolute value at x. |
| **pi** | The ratio of a circle's circumference to its diameter is a mathematical constant (3.14159...) |
| **e** | e is a constant in mathematics (2.71828...) |

**Python OS Module**

Python OS module provides the facility to establish the interaction between the user and the operating system. It offers many useful OS functions that are used to perform OS-based tasks and get related information about operating system.

The OS comes under Python's standard utility modules. This module offers a portable way of using operating system dependent functionality.

The Python OS module lets us work with the files and directories.

1. To work with the OS module, we need to import the OS module.
2. import os

There are some functions in the OS module which are given below:

**os.name()**

This function provides the name of the operating system module that it imports.

Currently, it registers 'posix', 'nt', 'os2', 'ce', 'java' and 'riscos'.

**Example**

1. import os
2. print(os.name)

**Output:**

nt

**os.mkdir()**

The **os.mkdir()** function is used to create new directory. Consider the following example.

1. import os
2. os.mkdir("d:\\newdir")

It will create the new directory to the path in the string argument of the function in the D drive named folder newdir.

**os.getcwd()**

It returns the current working directory(CWD) of the file.

**Example**

1. import os
2. print(os.getcwd())

**Output:**

C:\Users\Python\Desktop\ModuleOS

**os.chdir()**

The **os** module provides the **chdir()** function to change the current working directory.

1. import os
2. os.chdir("d:\\")

**Output:**

d:\\

**os.rmdir()**

The **rmdir()** function removes the specified directory with an absolute or related path. First, we have to change the current working directory and remove the folder.

**Example**

1. import os
2. # It will throw a Permission error; that's why we have to change the current working directory.
3. os.rmdir("d:\\newdir")
4. os.chdir("..")
5. os.rmdir("newdir")

**os.error()**

The os.error() function defines the OS level errors. It raises OSError in case of invalid or inaccessible file names and path etc.

**Example**

1. import os
3. try:
4. # If file does not exist,
5. # then it throw an IOError
6. filename = 'Python.txt'
7. f = open(filename, 'rU')
8. text = f.read()
9. f.close()
11. # The Control jumps directly to here if
12. # any lines throws IOError.
13. except IOError:
15. # print(os.error) will <class 'OSError'>
16. print('Problem reading: ' + filename)

**Output:**

Problem reading: Python.txt

**os.popen()**

This function opens a file or from the command specified, and it returns a file object which is connected to a pipe.

**Example**

1. import os
2. fd = "python.txt"
4. # popen() is similar to open()
5. file = open(fd, 'w')
6. file.write("This is awesome")
7. file.close()
8. file = open(fd, 'r')
9. text = file.read()
10. print(text)
12. # popen() provides gateway and accesses the file directly
13. file = os.popen(fd, 'w')
14. file.write("This is awesome")
15. # File not closed, shown in next function.

**Output:**

This is awesome

**os.close()**

This function closes the associated file with descriptor **fr**.

**Example**

1. import os
2. fr = "Python1.txt"
3. file = open(fr, 'r')
4. text = file.read()
5. print(text)
6. os.close(file)

**Output:**

Traceback (most recent call last):

File "main.py", line 3, in

file = open(fr, 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'Python1.txt'

**os.rename()**

A file or directory can be renamed by using the function **os.rename()**. A user can rename the file if it has privilege to change the file.

**Example**

1. import os
2. fd = "python.txt"
3. os.rename(fd,'Python1.txt')
4. os.rename(fd,'Python1.txt')

**Output:**

Traceback (most recent call last):

File "main.py", line 3, in

os.rename(fd,'Python1.txt')

FileNotFoundError: [Errno 2] No such file or directory: 'python.txt' -> 'Python1.txt'

**os.access()**

This function uses real **uid/gid** to test if the invoking user has access to the path.

**Example**

1. import os
2. import sys
4. path1 = os.access("Python.txt", os.F\_OK)
5. print("Exist path:", path1)
7. # Checking access with os.R\_OK
8. path2 = os.access("Python.txt", os.R\_OK)
9. print("It access to read the file:", path2)
11. # Checking access with os.W\_OK
12. path3 = os.access("Python.txt", os.W\_OK)
13. print("It access to write the file:", path3)
15. # Checking access with os.X\_OK
16. path4 = os.access("Python.txt", os.X\_OK)
17. print("Check if path can be executed:", path4)

**Output:**

Exist path: False

It access to read the file: False

It access to write the file: False

Check if path can be executed: False

**Python Random module**

The Python Random module is a built-in module for generating random integers in Python. These are sort of fake random numbers which do not possess true randomness. We can therefore use this module to generate random numbers, display a random item for a list or string, and so on.

**Generate Random Floats**

The random.random() function gives a float number that ranges from 0.0 to 1.0. There are no parameters required for this function.

**random.random():-** Returns The second random floating point value within [0.0 and 1) is returned.

**random.uniform(a, b):-** Generates a random floating point R in which a <= R <= b if a <= b and b <= R <= a if b < a.

**random.expovariate(lambda):-** Returns the random value according to exponential distribution.

**random.gauss(mu, sigma):-** Returns the random value according to gaussian distribution.

There are other distributions also, such as Gamma Distribution, Normal Distribution, etc.

**Code**

1. import random
2. num=random.random()
3. print(num)

**Output:**

0.3232640977876686

**Generate Random Integers**

The random.randint() function generates a random integer from the range of numbers supplied.

**Code**

1. import random
2. num = random.randint(1, 500)
3. print( num )

**Output:**

215

**Generate Random Numbers within a Defined Range**

The random.randrange() function selects an item randomly from the given range defined by the start, the stop, and the step parameters. By default, the start is set to 0. Likewise, the step is set to 1 by default.

**Code**

1. import random
3. num = random.randrange(1, 10)
4. print( num )
5. num = random.randrange(1, 10, 2)
6. print( num )
7. num = random.randrange(0, 101, 10)
8. print( num )

**Output:**

4

9

20

**Select Random Elements**

The random.choice() function selects an item from a non-empty series at random. An IndexError is thrown when the parameter is an empty series.

**Code**

1. import random
2. random\_s = random.choice('Random Module') #a string
3. print( random\_s )
4. random\_l = random.choice([23, 54, 765, 23, 45, 45]) #a list
5. print( random\_l )
6. random\_s = random.choice((12, 64, 23, 54, 34)) #a set
7. print( random\_s )

**Output:**

M

765

54

**Shuffle Elements Randomly**

A general sequence, like integers or floating-point series, can be a group of things like a List / Set. The random module contains methods that we can use to add randomization to the series.

The random.shuffle() function shuffles the entries in a list at random.

**Code**

1. a\_list = [34, 23, 65, 86, 23, 43]
2. random.shuffle( a\_list )
3. print( a\_list )
4. random.shuffle( a\_list )
5. print( a\_list )

**Output:**

[23, 43, 86, 65, 34, 23]

[65, 23, 86, 23, 34, 43]

**Random Seed**

We normally use the time of the system to ensure that the software delivers a different output each time we execute it because pseudorandom synthesis is dependent on the preceding number. As a result, we employ seeds.

We can specify a seed to have an initial number using Python's random.seed() function. This seed number determines a random number generator's outcome; therefore, if it stays the same, the outcome will continue to be the same.

**Code**

1. import random
2. random.seed(2)
3. print('Generating 5 random numbers: ')
4. print([ random.randint(1, 300) for r in range(6)])
6. # Reseting the seed value to 1
7. random.seed(2)
9. # We will get the same numbers as before
10. print([random.randint(1, 300) for i in range(6)])

**Output:**

Generating 5 random numbers:

[29, 47, 44, 185, 87, 158]

[29, 47, 44, 185, 87, 158]

**Various Functions of Random Module**

Following is the list of functions available in the random module.

|  |  |
| --- | --- |
| **Function** | **Description** |
| **seed(a=None, version=2)** | This function creates a new random number. |
| **getstate()** | This method provides an object reflecting the generator's present state. Provide the argument to setstate() to recover the state. |
| **setstate(state)** | Providing the state object resets the function's state at the time getstate() was invoked. |
| **getrandbits(k)** | This function provides a Python integer having k random bits. This is important for random number production algorithms like randrange(), which can manage arbitrarily huge ranges. |
| **randrange(start, stop[, step])** | From the range, it produces a random integer. |
| **randint(a, b)** | Provides an integer within a and b at random (both inclusive). If a > b, a ValueError is thrown. |
| **choice(seq)** | Produce a non-empty series item at random. |
| **shuffle(seq)** | Change the order. |
| **sample(population, k)** | Display a list of k-size unique entries from the population series. |
| **random()** | This function creates a new random number. |
| **uniform(a, b)** | This method provides an object reflecting the generator's present state. Provide the argument to setstate() to recover the state. |
| **triangular(low, high, mode)** | Providing the state object resets the function's state at the time getstate() was invoked. |
| **betavariate(alpha, beta)** | Beta distribution |
| **expovariate(lambd)** | Exponential distribution |
| **gammavariate(alpha, beta)** | Gamma distribution |
| **gauss(mu, sigma)** | Gaussian distribution |
| **lognormvariate(mu, sigma)** | Log normal distribution |
| **normalvariate(mu, sigma)** | Normal distribution |
| **vonmisesvariate(mu, kappa)** | Vonmises distribution |
| **paretovariate(alpha)** | Pareto distribution |
| **weibullvariate(alpha, beta)** | Weibull distribution |

We learned about various methods that Python's random module provides us with for dealing with Integers, floating-point numbers, and other sequences like Lists, tuples, etc. We also looked at how the seed affects the pseudo - random number pattern.

**Python statistics module**

Python statistics module provides the functions to mathematical statistics of numeric data. There are some popular statistical functions defined in this module.

**mean() function**

The mean() function is used to calculate the arithmetic mean of the numbers in the list.

**Example**

1. import statistics
2. # list of positive integer numbers
3. datasets = [5, 2, 7, 4, 2, 6, 8]
4. x = statistics.mean(datasets)
5. # Printing the mean
6. print("Mean is :", x)

**Output:**

Mean is : 4.857142857142857

**median() function**

The median() function is used to return the middle value of the numeric data in the list.

**Example**

1. import statistics
2. datasets = [4, -5, 6, 6, 9, 4, 5, -2]
3. # Printing median of the
4. # random data-set
5. print("Median of data-set is : % s "
6. % (statistics.median(datasets)))

**Output:**

Median of data-set is : 4.5

**mode() function**

The mode() function returns the most common data that occurs in the list.

**Example**

1. import statistics
2. # declaring a simple data-set consisting of real valued positive integers.
3. dataset =[2, 4, 7, 7, 2, 2, 3, 6, 6, 8]
4. # Printing out the mode of given data-set
5. print("Calculated Mode % s" % (statistics.mode(dataset)))

**Output:**

Calculated Mode 2

**stdev() function**

The stdev() function is used to calculate the standard deviation on a given sample which is available in the form of the list.

**Example**

1. import statistics
2. # creating a simple data - set
3. sample = [7, 8, 9, 10, 11]
4. # Prints standard deviation
5. print("Standard Deviation of sample is % s "
6. % (statistics.stdev(sample)))

**Output:**

Standard Deviation of sample is 1.5811388300841898

**median\_low()**

The median\_low function is used to return the low median of numeric data in the list.

**Example**

1. import statistics
2. # simple list of a set of integers
3. set1 = [4, 6, 2, 5, 7, 7]
4. # Note: low median will always be a member of the data-set.
5. # Print low median of the data-set
6. print("Low median of data-set is % s "
7. % (statistics.median\_low(set1)))

**Output:**

Low median of the data-set is 5

**median\_high()**

The median\_high function is used to return the high median of numeric data in the list.

**Example**

1. import statistics
2. # list of set of the integers
3. dataset = [2, 1, 7, 6, 1, 9]
4. print("High median of data-set is %s "
5. % (statistics.median\_high(dataset)))

**Output:**

High median of the data-set is 6

**Python statistics module**

Python statistics module provides the functions to mathematical statistics of numeric data. There are some popular statistical functions defined in this module.

**mean() function**

The mean() function is used to calculate the arithmetic mean of the numbers in the list.

**Example**

1. import statistics
2. # list of positive integer numbers
3. datasets = [5, 2, 7, 4, 2, 6, 8]
4. x = statistics.mean(datasets)
5. # Printing the mean
6. print("Mean is :", x)

**Output:**

Mean is : 4.857142857142857

**median() function**

The median() function is used to return the middle value of the numeric data in the list.

**Example**

1. import statistics
2. datasets = [4, -5, 6, 6, 9, 4, 5, -2]
3. # Printing median of the
4. # random data-set
5. print("Median of data-set is : % s "
6. % (statistics.median(datasets)))

**Output:**

Median of data-set is : 4.5

**mode() function**

The mode() function returns the most common data that occurs in the list.

**Example**

1. import statistics
2. # declaring a simple data-set consisting of real valued positive integers.
3. dataset =[2, 4, 7, 7, 2, 2, 3, 6, 6, 8]
4. # Printing out the mode of given data-set
5. print("Calculated Mode % s" % (statistics.mode(dataset)))

**Output:**

Calculated Mode 2

**stdev() function**

The stdev() function is used to calculate the standard deviation on a given sample which is available in the form of the list.

**Example**

1. import statistics
2. # creating a simple data - set
3. sample = [7, 8, 9, 10, 11]
4. # Prints standard deviation
5. print("Standard Deviation of sample is % s "
6. % (statistics.stdev(sample)))

**Output:**

Standard Deviation of sample is 1.5811388300841898

**median\_low()**

The median\_low function is used to return the low median of numeric data in the list.

**Example**

1. import statistics
2. # simple list of a set of integers
3. set1 = [4, 6, 2, 5, 7, 7]
4. # Note: low median will always be a member of the data-set.
5. # Print low median of the data-set
6. print("Low median of data-set is % s "
7. % (statistics.median\_low(set1)))

**Output:**

Low median of the data-set is 5

**median\_high()**

The median\_high function is used to return the high median of numeric data in the list.

**Example**

1. import statistics
2. # list of set of the integers
3. dataset = [2, 1, 7, 6, 1, 9]
4. print("High median of data-set is %s "
5. % (statistics.median\_high(dataset)))

**Output:**

High median of the data-set is 6

# Python sys module

The python sys module provides functions and variables which are used to manipulate different parts of the Python Runtime Environment. It lets us access system-specific parameters and functions.

**import sys**

First, we have to import the sys module in our program before running any functions.

**sys.modules**

This function provides the name of the existing python modules which have been imported.

**sys.argv**

This function returns a list of command line arguments passed to a Python script. The name of the script is always the item at index 0, and the rest of the arguments are stored at subsequent indices.

**sys.base\_exec\_prefix**

This function provides an efficient way to the same value as exec\_prefix. If not running a virtual environment, the value will remain the same.

**sys.base\_prefix**

It is set up during Python startup, before site.py is run, to the same value as prefix.

**sys.byteorder**

It is an indication of the native byteorder that provides an efficient way to do something.

**sys.maxsize**

This function returns the largest integer of a variable.

**sys.path**

This function shows the PYTHONPATH set in the current system. It is an environment variable that is a search path for all the python modules.

**sys.stdin**

It is an object that contains the original values of stdin at the start of the program and used during finalization. It can restore the files.

**sys.getrefcount**

This function returns the reference count of an object.

**sys.exit**

This function is used to exit from either the Python console or command prompt, and also used to exit from the program in case of an exception.

**sys executable**

The value of this function is the absolute path to a Python interpreter. It is useful for knowing where python is installed on someone else machine.

**sys.platform**

This value of this function is used to identify the platform on which we are working.

# Python IDEs

IDE stands for Integrated Development Environment is defined as a coding tool that helps to automate the process of editing, compiling, testing, etc. in an SDLC and it provides ease to the developer to run, write and debug the code.

It is specially designed for software development that consists of several tools which is used for developing and testing the software.

There are some Python IDEs which are as follows:



* [PyCharm](https://www.javatpoint.com/python-ides#PyCharm)
* [Spyder](https://www.javatpoint.com/python-ides#Spyder)
* [PyDev](https://www.javatpoint.com/python-ides#PyDev)
* [Atom](https://www.javatpoint.com/python-ides#Atom)
* [Wing](https://www.javatpoint.com/python-ides#Wing)
* [Jupyter Notebook](https://www.javatpoint.com/python-ides#JupyterNotebook)
* [Thonny](https://www.javatpoint.com/python-ides#Thonny)
* [Rodeo](https://www.javatpoint.com/python-ides#Rodeo)
* [Microsoft Visual Studio](https://www.javatpoint.com/python-ides#MicrosoftVisualStudio)
* [Eric](https://www.javatpoint.com/python-ides#Eric)

## PyCharm



PyCharm was developed by the Jet Brains, and it is a cross-platform Integrated Development Environment (IDE) specially designed for python. It is the most widely used IDE and available in both paid version and free open-source as well. It saves ample time by taking care of routine tasks.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

It is a complete python IDE that is loaded with a rich set of features like auto code completion, quick project navigation, fast error checking and correction, remote development support, database accessibility, etc.

### Features

* Smart code navigation
* Errors Highlighting
* Powerful debugger
* Supports Python web development frameworks, i.e., Angular JS, Javascript

## Spyder



Spyder is an open-source that has high recognition in the IDE market and most suitable for data science. The full name of Spyder is Scientific Python Development Environment. It supports all the significant platforms Linux, Windows, and MacOS X.

It provides a set of features like localized code editor, document viewer, variable explorer, integrated console, etc. and supports no. of scientific modules like NumPy, SciPy, etc.

### Features

* Proper syntax highlighting and auto code completion
* Integrates strongly with IPython console
* Performs well in multi-language editor and auto code completion mode

## PyDev



PyDev is defined as one of the commonly used Python IDE, which is an external plugin for Eclipse. It is a natural choice of the Python developers that are coming from the Java background and very popular in the market as Python interpreter.

Aleksandar Totic is famous for his contribution to Mosaic browser and worked on Pydev project during 2003-2004.

Pydev has a feature which includes Django integration, automatic code completion, smart indents and block indents, etc.

### Features

* Strong Parameters like refactoring, debugging, code analysis, and code coverage function.
* It supports virtual environments, Mypy, and black formatter.
* Also supports PyLint integration, remote debugger, Unit test integration, etc.

## Atom



Atom is developed by GitHub, which is initially started as an open-source, cross-platform. It is based on a framework, i.e., Electron which enables cross-platform desktop application using Chromium and Node.js and generally known as "Hackable Text Editor for the 21st century".

### Features

* Visualize the results on Atom without open any other window.
* A plugin named "Markdown Preview Plus" provides built-in support for editing and visualizing Markdown files.

## Wing



It is defined as a cross-platform IDE that is packed with necessary features and with decent development support. Its personal edition is free of cost. The pro version comes with a 30 days trial for the developers to try it out.

It has several features that include auto-completion, syntax highlighting, indents, and debugging.

### Features

* Customizable and can have extensions as well.
* Supports remote development, test-driven development along with the unit test.

## Jupyter Notebook



Jupyter is one of the most used IPython notebook editors that is used across the Data Science industry. It is a web application that is based on the server-client structure and allows you to create and manipulate notebook documents. It makes the best use of the fact that python is an interpreted language.

### Features

* Supports markdowns
* Easy creation and editing of codes
* Ideal for beginners in data science

## Thonny



Thonny is another IDE which is best suited for learning and teaching programming. It is a software developed at the University of Tartu and supports code completion and highlight syntax errors.

### Features

* Simple debugger
* Supports highlighting errors and auto code completion

## Rodeo



Rodeo is defined as one of the best IDE for python that is most widely used for data science projects like taking data and information from different resources.

It supports cross-platform functionality and provides auto-completion of code.

### Features

* Allows the functions for comparing data, interact, plot, and inspect data.
* Auto code completion, syntax highlighter, visual file navigator, etc.

## Microsoft Visual Studio



Microsoft Visual Studio is an open-source code editor which was best suited for development and debugging of latest web and cloud projects. It has its own marketplace for extensions.

### Features

* Supports Python Coding in Visual studio
* Available in both paid and free version

## Eric Python



The Eric Python is an editor which is developed in Python itself and can be used for both professional and non-professional work.

### Features

* Offers configurable window layout, editors, source code folding
* Advanced project management capability, version control
* In-built debugger and task management support

**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. An array is popular in most programming languages like C/C++, JavaScript, etc.

Array is an idea of storing multiple items of the same type together and it makes easier to calculate the position of each element by simply adding an offset to the base value. A combination of the arrays could save a lot of time by reducing the overall size of the code. It is used to store multiple values in single variable. If you have a list of items that are stored in their corresponding variables like this:

car1 = "Lamborghini"

car2 = "Bugatti"

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

car3 = "Koenigsegg"

If you want to loop through cars and find a specific one, you can use the array.

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.

The Array can be created in Python by importing the array module to the 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])
4. print("Second element:", a[1])
5. print("Second last element:", a[-1])

**Output:**

First element: 2

Second element: 4

Second last element: 8

**Explanation:** In the above example, we have imported an array, defined a variable named as "a" that holds the elements of an array and print the elements by accessing elements through indices of an array.

**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])
4. # changing first element
5. numbers[0] = 0
6. print(numbers)    # Output: array('i', [0, 2, 3, 5, 7, 10])
8. # changing 3rd to 5th element
9. numbers[2:5] = arr.array('i', [4, 6, 8])
10. print(numbers)    # Output: array('i', [0, 2, 4, 6, 8, 10])

**Output:**

array('i', [0, 2, 3, 5, 7, 10])

array('i' ,[0, 2, 4, 6, 8, 10])

**Explanation:** In the above example, we have imported an array and defined a variable named as "numbers" which holds the value of an array. If we want to change or add the elements in an array, we can do it by defining the particular index of an array where you want to change or add the elements.

**Why to use arrays in Python?**

A combination of arrays saves a lot of time. The array can reduce the overall size of the code.

**How to delete elements from an array?**

The 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])

**Output:**

array('i', [10, 20, 40, 60])

**Explanation:** In the above example, we have imported an array and defined a variable named as "number" which stores the values of an array. Here, by using del statement, we are removing the third element [3] of the given array.

**Finding the length of an array**

The length of an array is defined as the number of elements present in an array. It returns an integer value that is equal to the total number of the elements present in that array.

**Syntax**

1. len(array\_name)

**Array Concatenation**

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

**Example**

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)

**Output:**

Array c= array('d', [1.1, 2.1, 3.1, 2.6, 7.8, 3.7, 8.6])

**Explanation**

In the above example, we have defined variables named as "a, b, c" that hold the values of an array.

**Example**

1. import array as arr
2. x = arr.array('i', [4, 7, 19, 22])
3. print("First element:", x[0])
4. print("Second element:", x[1])
5. print("Second last element:", x[-1])

**Output:**

First element: 4

Second element: 7

Second last element: 22

**Explanation:** In the above example, first, we have imported an array and defined a variable named as "x" which holds the value of an array and then, we have printed the elements using the indices of an array.

**Python Command line arguments**

The Python supports the programs that can be run on the command line, complete with command line arguments. It is the input parameter that needs to be passed to the script when executing them.

It means to interact with a command-line interface for the scripts.

It provides a **getopt** module, in which command line arguments and options can be parsed.

**What is argument passing?**

The command **ls** is often used to get a summary of files and folders present in a particular directory.

**Why to use argparse?**

It means to communicate between the writer of a program and user which does not require going into the code and making changes to the script. It provides the ability to a user to enter into the command-line arguments.

**Access command line arguments**

The Python sys module provides access to command-line arguments via sys.argv. It solves the two purposes:

**Python sys module**

It is a basic module that was shipped with Python distribution from the early days on. It is a similar approach as C library using argc/argv to access the arguments. The sys module implements command-line arguments in a simple list structure named sys.argv.

Each list element represents a single argument. The first one -- sys.argv[0] -- is the name of Python script. The other list elements are sys.argv[1] to sys.argv[n]- are the command line arguments 2 to n. As a delimiter between arguments, space is used. Argument values that contain space in it have to be quoted, accordingly.

It stores command-line arguments into a list; we can access it using **sys.argv**. This is very useful and a simple way to read command-line arguments as String.

1. import sys
2. print(type(sys.argv))
3. print('The command line arguments are:')
4. for i in sys.argv:
5. print(i)

**Python getopt module**

The Python getopt module extends the separation of the input string by parameter validation. Based on getopt C function, it allows both short and long options, including a value assignment.

It is very similar to C getopt() function for parsing command line parameters.

It is useful in parsing command line arguments where we want the user to enter some options.

**Code**

1. import getopt
2. import sys
3. argv = sys.argv[1:]
4. try:
5. opts, args = getopt.getopt(argv, 'hm:d', ['help', 'my\_file='])
6. print(opts)
7. print(args)
8. except getopt.GetoptError:
9. # Print a message or do something useful
10. print('Something went wrong!')
11. sys.exit(2)

**Python argparse module**

It offers a command-line interface with standardized output, whereas the former two solutions leave most of the work in your hands. argparse allows verification of fixed and optional arguments with a name checking as either UNIX or GNU style. It is the preferred way to parse command-line arguments. It provides a lot of option such as positional arguments, the default value for arguments, helps message, specifying the data type of argument etc.

It makes it easy to write the user-friendly command-line interfaces. It automatically generates help and usage messages and issues errors when a user gives invalid arguments to the program.

**getopt.getopt method**

This method is used for parsing the command line options and parameter list.

**Syntax:**

1. getopt.getopt(args, options, [long\_options])

**args**- It is an argument list that needs to be parsed.

**options**- A string of option letters that the script wants to recognize, with options that require an argument which should be followed by a colon(:).

**long\_options(optional)**- It must be a string with names of the long options, which should be supported.

* This method returns a value consisting of two elements, i.e. list of (**option, value**) pairs, list of program arguments left after option list was stripped.
* Each option-and-value pair are returned as an option as its first element, prefixed with a hyphen for short options (e.g.,'-x') or two hyphens for long options (e.g., '--long-option').

**Exception getopt.GetoptError**

This exception arises when an unrecognized option is found in the argument list or when any option requiring an argument is given none.

The argument to the exception is a string that indicates the cause of the error. The attributes **msg** and **opt** to give the error message and related option.

**Code**

1. #!/usr/bin/python
2. import sys, getopt
3. def main(argv):
4. inputfile = ''
5. outputfile = ''
6. try:
7. opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
8. except getopt.GetoptError:
9. print 'test.py -i <inputfile> -o <outputfile>'
10. sys.exit(2)
11. for opt, arg in opts:
12. if opt == '-h':
13. print 'test.py -i <inputfile> -o <outputfile>'
14. sys.exit()
15. elif opt in ("-i", "--ifile"):
16. inputfile = arg
17. elif opt in ("-o", "--ofile"):
18. outputfile = arg
19. print 'Input file is "', inputfile
20. print 'Output file is "', outputfile
22. if \_\_name\_\_ == "\_\_main\_\_":
23. main(sys.argv[1:])

**Output:**

$ test.py -h

usage: test.py -i <inputfile> -o <outputfile>

$ test.py -i BMP -o

usage: test.py -i <inputfile> -o <outputfile>

$ test.py -i inputfile

Input file is " inputfile

Output file is "

**How to use command line arguments in python?**

<="" p="">

|  |  |  |
| --- | --- | --- |
| **Module** | **Use** | **Python version** |
| sys | All arguments in sys.argv (basic) | All |
| argparse | Build a command line interface | >= 2.3 |
| docopt | Created command line interfaces | >= 2.5 |
| fire | Automatically generate command line interfaces (CLIs) | All |
| optparse | Deprecated | < 2.7 |

**Docopt**

Docopt is used to create command line interfaces.

1. from docopt import docopt
2. if \_\_name\_\_ == '\_\_main\_\_':
3. arguments = docopt(\_\_doc\_\_, version='Example 1')
4. print(arguments)

**Fire**

Python Fire automatically generates a command line interface; you only need one line of code. Unlike the other modules, it works instantly.

You don't need to define any arguments; all the methods are linked by default.

To install it type:

1. pip install fire

Define or use a class:

1. import fire
2. class Python(object):
3. def hello(self):
4. print("Hello")
5. def openfile(self, filename):
6. print("Open file '" + filename + "'")
8. if \_\_name\_\_ == '\_\_main\_\_':
9. fire.Fire(Python)

You have the options matching to the class methods:

1. python example.py hello
2. python example.py openfile filename.txt

**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.

We can integrate Python's built-in classes with our classes. The class which has inherited from the built-in class is known as a child class. A child class has access to all of the attributes of the parent class, including its methods. By utilizing the essential built-in features, we can customize some of the tasks of our class by using magic methods.

**\_\_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, just like constructors in various programming languages like the popular ones C++, Java, C#, PHP, etc. These methods are invoked after \_new\_ and therefore are referred to as initialising. We should define the instance parameters here.

**Code**

1. # Python program to show how \_\_init\_\_ method works
3. # Creating a class
4. class methods():
5. def \_\_init\_\_(self, \*args):
6. print ("Now called \_\_init\_\_ magic method, after tha initialised parameters")
7. self.name = args[0]
8. self.std = args[1]
9. self.marks = args[2]
11. Student = methods("Itika", 11, 98)
12. print(Student)
13. print(f"Name, standard, and marks of the student is: \n", Student.name, "\n", Student.std, "\n", Student.marks)

**Output:**

Now called \_\_init\_\_ magic method, after tha initialised parameters

<\_\_main\_\_.methods object at 0x3701290>

Name, standard, and marks of the student is:

Itika

11

98

**\_\_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.

**Code**

1. # Python program to show how \_\_new\_\_ method works
3. # Creating a class
4. class Method(object):
5. def \_\_new\_\_( cls ):
6. print( "Creating an instance by \_\_new\_\_ method")
7. return super(Method, cls).\_\_new\_\_(cls)
8. # Calling the init method
9. def \_\_init\_\_( self ):
10. print( "Init method is called here" )
12. Method()

**Output:**

Creating an instance by \_\_new\_\_ method

Init method is called here

<\_\_main\_\_.Method at 0x30dfb88>

**\_\_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.

Below is the code to show how we add two attributes of two instances of different classes without using the \_\_add\_\_ magic method.

**Code**

1. # Python program to show how to add two attributes
3. # Creating a class
4. class Method:
5. def \_\_init\_\_(self, argument):
6. self.attribute = argument
8. # Creating a second class
9. class Method\_2:
10. def \_\_init\_\_(self, argument):
11. self.attribute = argument
12. # creating the instances
13. instance\_1 = Method(" Attribute")
14. print(instance\_1.attribute)
15. instance\_2 = Method\_2(" 27")
16. print(instance\_2.attribute)
18. # Adding two attributes of the instances
19. print(instance\_2.attribute + instance\_1.attribute)

**Output:**

Attribute

27

27 Attribute

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

**Code**

1. # Python program to show how \_\_add\_\_ method works
3. # Creating a class
4. class Method:
5. def \_\_init\_\_(self, argument):
6. self.attribute = argument
7. def \_\_add\_\_(self, object1):
8. return self.attribute + object1.attribute
10. # Creating a second class
11. class Method\_2:
12. def \_\_init\_\_(self, argument):
13. self.attribute = argument
14. def \_\_add\_\_(self, object1):
15. return self.attribute + object1.attribute
16. instance\_1 = Method(" Attribute")
17. print(instance\_1)
18. instance\_2 = Method\_2(" 27")
19. print(instance\_2)
20. print(instance\_2 + instance\_1)

**Output:**

<\_\_main\_\_.Method object at 0x37470f0>

<\_\_main\_\_.Method\_2 object at 0x376beb8>

27 Attribute

Classes, Method and Method\_1 in the script above have a property called "attribute" that stores a string. We create two instances, instance\_1 and instances\_2, with corresponding attributes of " Attribute" and " 27" values. The \_\_add\_\_ method is used to translate the action instance\_1 + instance\_2 into instance\_1 + instance\_2.attribute, which produces output ( 27 Attribute).

**\_\_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.

**Code**

1. # Python program to show how \_\_repr\_\_ magic method works
3. # Creating a class
4. class Method:
5. # Calling \_\_init\_\_ method and initializing the attributes of the class
6. def \_\_init\_\_(self, x, y, z):
7. self.x = x
8. self.y = y
9. self.z = z
10. # Calling the \_\_repr\_\_ method and providing the string to be printed each time instance is printe
11. def \_\_repr\_\_(self):
12. return f"Following are the values of the attributes of the class Method:\nx = {self.x}\ny = {self.y}\nz = {self.z}"
13. instance = Method(4, 6, 2)
14. print(instance)

**Output:**

Following are the values of the attributes of the class Method:

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 ( such as lists, tuples, etc.).

**Code**

1. # Python code to show how the \_\_contains\_\_ magic method works
3. # Creating a class
4. class Method:
5. # Calling the \_\_init\_\_ method and initializing the attributes
6. def \_\_init\_\_(self, attribute):
7. self.attribute = attribute
9. # Calling the \_\_contains\_\_ method
10. def \_\_contains\_\_(self, attribute):
11. return attribute in self.attribute
12. # Creating an instance of the class
13. instance = Method([4, 6, 8, 9, 1, 6])
15. # Checking if a value is present in the container attribute
16. print("4 is contained in ""attribute"": ", 4 in instance)
17. print("5 is contained in ""attribute"": ", 5 in instance)

**Output:**

4 is contained in attribute: True

5 is contained in attribute: False

We have used the \_\_contanis\_\_ magic method in the program above to determine if a given integer is contained in the attribute "attribute". In this case, "attribute" is a list of integers. The integer 4 is present in the given list of integers passed to the class Method as an attribute. While 5 is not present in the list.

**\_\_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.

**Code**

1. # Python program to show how the \_\_call\_\_ magic method works
3. # Creating a class
4. class Method:
5. # Calling the \_\_init\_\_ method and initializing the attributes
6. def \_\_init\_\_(self, a):
7. self.a = a
8. # Calling the \_\_call\_\_ method to multiply a number to the attribute value
9. def \_\_call\_\_(self, number):
10. return self.a \* number
12. # Creating an instance and proving the value to the attribute a
13. instance = Method(7)
14. print(instance.a) # Printing the value of the attribute a
15. # Calling the instance while passing a value which will call the \_\_call\_\_ method
16. print(instance(5))

**Output:**

7

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.

**Code**

1. # Python program to show how the \_\_iter\_\_ method works
3. # Creating a class
4. class Method:
5. def \_\_init\_\_(self, start\_value, stop\_value):
6. self.start = start\_value
7. self.stop = stop\_value
8. def \_\_iter\_\_(self):
9. for num in range(self.start, self.stop + 1):
10. yield num \*\* 2
11. # Creating an instance
12. instance = iter(Method(3, 8))
13. print( next(instance) )
14. print( next(instance) )
15. print( next(instance) )
16. print( next(instance) )
17. print( next(instance) )
18. print( next(instance) )

**Output:**

9

16

25

36

49

64

We have calculated the squares of the numbers in the code. For the numbers in the specified range, squares are calculated in the program above (start and stop). The \_\_iter\_\_ method, which generates squares of the numbers between the given range, is called when we call the function iter(Method(3, 8)). In our example, we're using the range of 3 to 8; therefore, calling the next() method will produce the results 9, 16, 25, 36, 49, 64.

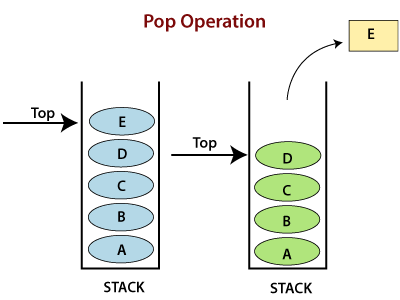
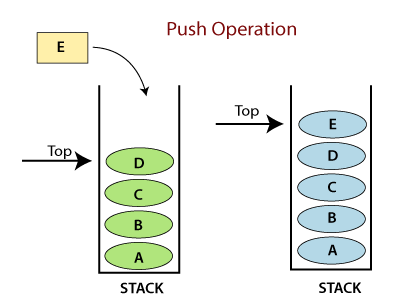
# Python Stack and Queue

Data structure organizes the storage in computers so that we can easily access and change data. Stacks and Queues are the earliest data structure defined in computer science. A simple Python list can act as a queue and stack as well. A queue follows FIFO rule (First In First Out) and used in programming for sorting. It is common for stacks and queues to be implemented with an array or linked list.

## 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.



**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:**

[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

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

**Code**

1. # Code to demonstrate Implementation of
2. # stack using list
3. x = ["Python", "C", "Android"]
4. x.push("Java")
5. x.push("C++")
6. print(x)
7. print(x.pop())
8. print(x)
9. print(x.pop())
10. print(x)

**Output:**

['Python', 'C', 'Android', 'Java', 'C++']

C++

['Python', 'C', 'Android', 'Java']

Java

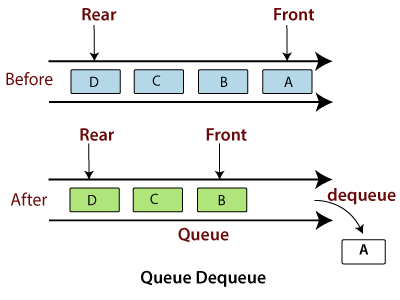
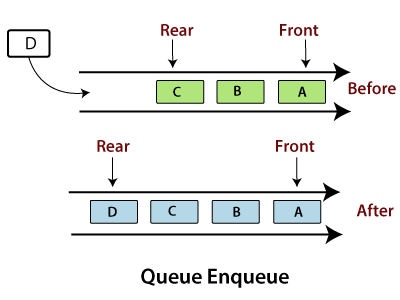
['Python', 'C', 'Android']

## 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.

**Characteristics**

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

#### Note: The implementation of a queue is a little bit different. A queue follows the "First-In-First-Out". Time plays an important factor here. The Stack is fast because we insert and pop the elements from the end of the list, whereas in the queue, the insertion and pops are made from the beginning of the list, so it becomes slow. The cause of this time difference is due to the properties of the list, which is fast in the end operation but slow at the beginning operations because all other elements have to be shifted one by one.

### Code

1. import queue
2. # Queue is created as an object 'L'
3. L = queue.Queue(maxsize=10)
5. # Data is inserted in 'L' at the end using put()
6. L.put(9)
7. L.put(6)
8. L.put(7)
9. L.put(4)
10. # get() takes data from
11. # from the head
12. # of the Queue
13. print(L.get())
14. print(L.get())
15. print(L.get())
16. print(L.get())

**Output:**

9

6

7

4

# 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)

**Output:**

+--------------------+--------------------+----------------+------------------+------------------+------------------+--------------------+-------------------+

| \_c0| \_c1| \_c2| \_c3| \_c4| \_c5| \_c6| \_c7|

+--------------------+--------------------+----------------+------------------+------------------+------------------+--------------------+-------------------+

| Email| Address| Avatar|Avg Session Length| Time on App| Time on Website|Length of Membership|Yearly Amount Spent|

|mstephenson@ferna...|835 Frank TunnelW...| Violet| 34.49726772511229| 12.65565114916675| 39.57766801952616| 4.0826206329529615| 587.9510539684005|

| hduke@hotmail.com|4547 Archer Commo...| DarkGreen| 31.92627202636016|11.109460728682564|37.268958868297744| 2.66403418213262| 392.2049334443264|

| pallen@yahoo.com|24645 Valerie Uni...| Bisque|33.000914755642675|11.330278057777512|37.110597442120856| 4.104543202376424| 487.54750486747207|

|riverarebecca@gma...|1414 David Throug...| SaddleBrown| 34.30555662975554|13.717513665142507| 36.72128267790313| 3.120178782748092| 581.8523440352177|

|mstephens@davidso...|14023 Rodriguez P...|MediumAquaMarine| 33.33067252364639|12.795188551078114| 37.53665330059473| 4.446308318351434| 599.4060920457634|

|alvareznancy@luca...|645 Martha Park A...| FloralWhite|33.871037879341976|12.026925339755056| 34.47687762925054| 5.493507201364199| 637.102447915074|

|katherine20@yahoo...|68388 Reyes Light...| DarkSlateBlue| 32.02159550138701|11.366348309710526| 36.68377615286961| 4.685017246570912| 521.5721747578274|

| awatkins@yahoo.com|Unit 6538 Box 898...| Aqua|32.739142938380326| 12.35195897300293| 37.37335885854755| 4.4342734348999375| 549.9041461052942|

|vchurch@walter-ma...|860 Lee KeyWest D...| Salmon| 33.98777289568564|13.386235275676436|37.534497341555735| 3.2734335777477144| 570.2004089636196|

+--------------------+--------------------+----------------+------------------+------------------+------------------+--------------------+-------------------+

only showing top 10 rows

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()

**Output:**

+------------------+

Independent Feature

+------------------+

|34.49726772511229 |

|31.92627202636016 |

|33.000914755642675|

|34.30555662975554 |

|33.33067252364639 |

|33.871037879341976|

|32.02159550138701 |

|32.739142938380326|

|33.98777289568564 |

+------------------+

1. z = featureassembler.transform(dataset)
2. finlized\_data = z.select("Indepenent feature", "Yearly Amount Spent",)
3. z.show()

**Output:**

+--------------------++-------------------+

|Independent Feature | Yearly Amount Spent|

+--------------------++-------------------+

|34.49726772511229 | 587.9510539684005 |

|31.92627202636016 | 392.2049334443264 |

|33.000914755642675 | 487.5475048674720 |

|34.30555662975554 | 581.8523440352177 |

|33.33067252364639 | 599.4060920457634 |

|33.871037879341976 | 637.102447915074 |

|32.02159550138701 | 521.5721747578274 |

|32.739142938380326 | 549.9041461052942 |

|33.98777289568564 | 570.2004089636196 |

+--------------------++-------------------+

PySpark provides the **LinearRegression()** function to find the prediction of any given dataset. The syntax is given below:

1. regressor = LinearRegression(featureCol = 'column\_name1', labelCol = 'column\_name2 ')

## 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. The below example is showing the use of MLlib K-Means Cluster library:

1. from pyspark.ml.clustering import KMeans
2. from pyspark.ml.evaluation import ClusteringEvaluator
3. # Loads data.
4. dataset = spark.read.format("libsvm").load(r"C:\Users\DEVANSH SHARMA\Iris.csv")
5. # Trains a k-means model.
6. kmeans = KMeans().setK(2).setSeed(1)
7. model = kmeans.fit(dataset)
8. # Make predictions
9. predictions = model.transform(dataset)
10. # Evaluate clustering by computing Silhouette score
11. evaluator = ClusteringEvaluator()
12. silhouette = evaluator.evaluate(predictions)
13. print("Silhouette with squared euclidean distance = " + str(silhouette))
14. # Shows the result.
15. centers = model.clusterCenters()
16. print("Cluster Centers: ")
17. for center in centers:
18. print(center)

### 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.

Let's consider the following example, where we load ratings data from the MovieLens dataset. Each row is containing a user, a movie, rating and a timestamp.

1. #importing the libraries
2. frompyspark.ml.evaluation import RegressionEvaluator
3. frompyspark.ml.recommendation import ALS
4. frompyspark.sql import Row
5. no\_of\_lines = spark.read.text(r"C:\Users\DEVANSH SHARMA\MovieLens.csv").rdd
6. no\_of\_parts = no\_of\_lines.map(lambda row: row.value.split("::"))
7. ratingsRDD = no\_of\_lines.map(lambda p: Row(userId=int(p[0]), movieId=int(p[1]),
8. rating=float(p[2]), timestamp=long(p[3])))
9. ratings = spark.createDataFrame(ratingsRDD)
10. (training, test) = ratings.randomSplit([0.8, 0.2])
12. # Develop the recommendation model using ALS on the training data
13. # Note we set cold start strategy to make sure that we don't get NaN evaluation metrics.
14. als = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating",
15. coldStartStrategy="drop")
16. model = als.fit(training)
18. # Calculate the model by computing the RMSE on the test data
19. predictions = model.transform(test)
20. evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
21. predictionCol="prediction")
22. rmse = evaluator.evaluate(predictions)
23. print("Root-mean-square error = " + str(rmse))
25. # Evaluate top 10 movie recommendations for each user
26. userRecs = model.recommendForAllUsers(10)
27. # Evaluate top 10 user recommendations for each movie
28. movieRecs = model.recommendForAllItems(10)
29. # Evaluate top 10 movie recommendations for a specified set of users
30. users = ratings.select(als.getUserCol()).distinct().limit(3)
31. userSubsetRecs = model.recommendForUserSubset(users, 10)
32. # Evalute top 10 user recommendations for a specified set of movies
33. movies = ratings.select(als.getItemCol()).distinct().limit(3)
34. movieSubSetRecs = model.recommendForItemSubset(movies, 10)

# 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.

Before understanding the **Decorator**, we need to know some important concepts of Python.

## 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")

**Output:**

Hii

Hii

In the above program, when we run the code it give the same output for both functions. The **func2** referred to function **func1** and act as function. We need to understand the following concept of the function:

* 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. Consider the following example:

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()
8. func2()
9. func()

**Output:**

We are in first function

This is first child function

This is second child 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.

A function that accepts other function as an argument is also called **higher order function**. Consider the following example:

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. print(operator(add,20))

**Output:**

9

21

In the above program, we have passed the **sub()** function and **add()** function as argument in **operator()** function.

A function can return another function. Consider the below example:

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

**Output:**

Hello

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

### Decorating functions with parameters

Let's have an example to understand the parameterized decorator function:

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)

**Output:**

2.0

### Syntactic Decorator

In the above program, we have decorated **out\_div()** that is little bit bulky. Instead of using above method, Python allows to **use decorator in easy way with @symbol**. Sometimes it is called "pie" syntax.

1. def outer\_div(func):
2. def inner(x,y):
3. if(x<y):
4. x,y = y,x
5. return func(x,y)
6. return inner
7. # syntax of generator
8. @outer\_div
9. def divide(x,y):
10. print(x/y)

**Output:**

2.0

### Reusing Decorator

We can reuse the decorator as well by recalling that decorator function. Let's make the decorator to its own module that can be used in many other functions. Creating a file called **mod\_decorator.py** with the following code:

1. def do\_twice(func):
2. def wrapper\_do\_twice():
3. func()
4. func()
5. return wrapper\_do\_twice

We can import mod\_decorator.py in other file.

1. from decorator import do\_twice
2. @do\_twice
3. def say\_hello():
4. print("Hello There")
5. say\_hello()

**Output:**

Hello There

Hello There

### Python Decorator with Argument

We want to pass some arguments in function. Let's do it in following code:

1. from decorator import do\_twice
2. @do\_twice
3. def display(name):
4. print(f"Hello {name}")
5. display()

**Output:**

TypeError: display() missing 1 required positional argument: 'name'

As we can see that, the function didn't accept the argument. Running this code raises an error. We can fix this error by using **\*args** and **\*\*kwargs** in the inner wrapper function. Modifying the **decorator.py** as follows:

1. def do\_twice(func):
2. def wrapper\_function(\*args,\*\*kwargs):
3. func(\*args,\*\*kwargs)
4. func(\*args,\*\*kwargs)
5. return wrapper\_function

Now **wrapper\_function()** can accept any number of argument and pass them on the function.

1. from decorator import do\_twice
2. @do\_twice
3. def display(name):
4. print(f"Hello {name}")
5. display("John")

**Output:**

Hello John

Hello John

### Returning Values from Decorated Functions

We can control the return type of the decorated function. The example is given below:

1. from decorator import do\_twice
2. @do\_twice
3. def return\_greeting(name):
4. print("We are created greeting")
5. return f"Hi {name}"
6. hi\_adam = return\_greeting("Adam")

**Output:**

We are created greeting

We are created greeting

## Fancy Decorators

Let's understand the fancy decorators by the following topic:

### 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. Let’s understand it by the following example:

Example: 1- **@property decorator** - By using it, we can use the class function as an attribute. Consider the following code:

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
9. stu = Student("John","B")
10. print("Name:", stu.name)
11. print("Grade:", stu.grade)
12. print(stu.display)

**Output:**

Name: John

Grade: B

John got grade B

Example: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. Consider the following code:

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

**Output:**

Hello Peter

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. Let's consider the following example:

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

In the above code, we have used the nested decorator by stacking them onto one another.

### 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. Let us consider the following example:

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

**Output:**

JavatPoint

JavatPoint

JavatPoint

JavatPoint

JavatPoint

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 the example where we are creating a decorator that counts how many times the function has been called.

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

**Output:**

Call 1 of 'say\_hello'

Say Hello

Call 2 of 'say\_hello'

Say Hello

In the above program, the state represented the number of calls of the function stored in **.num\_calls** on the wrapper function. When we call **say\_hello()** it will display the number of the call of the function.

### 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.

1. import functools
3. class Count\_Calls:
4. def \_\_init\_\_(self, func):
5. functools.update\_wrapper(self, func)
6. self.func = func
7. self.num\_calls = 0
9. def \_\_call\_\_(self, \*args, \*\*kwargs):
10. self.num\_calls += 1
11. print(f"Call{self.num\_calls} of {self.func.\_\_name\_\_!r}")
12. return self.func(\*args, \*\*kwargs)
14. @Count\_Calls
15. def say\_hello():
16. print("Say Hello")
18. say\_hello()
19. say\_hello()
20. say\_hello()

**Output:**

Call 1 of 'say\_hello'

Say Hello

Call 2 of 'say\_hello'

Say Hello

Call 3 of 'say\_hello'

Say Hello

The **\_\_init\_\_()** method stores a reference to the function and can do any other required initialization.

# Python Generators

## What is Python Generator?

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 quite simple to create a generator 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. Consider the following example:

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

**Output:**

0

2

4

6

8

### 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. Consider the following example.

1. def multiple\_yield():
2. str1 = "First String"
3. yield str1
5. str2 = "Second string"
6. yield str2
8. str3 = "Third String"
9. yield str3
10. obj = multiple\_yield()
11. print(next(obj))
12. print(next(obj))
13. print(next(obj))

**Output:**

First String

Second string

Third String

### Difference between Generator function and Normal function

* Normal function contains only one L**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.

Consider the following example:

1. list = [1,2,3,4,5,6,7]
3. # List Comprehension
4. z = [x\*\*3 for x in list]
6. # Generator expression
7. a = (x\*\*3 for x in list)
9. print(a)
10. print(z)

**Output:**

<generator object <genexpr> at 0x01BA3CD8>

[1, 8, 27, 64, 125, 216, 343]

In the above program, list comprehension has returned the list of cube of elements whereas generator expression has returned the reference of calculated value. Instead of applying a **for loop**, we can also call **next()** on the generator object. Let's consider another example:

1. list = [1,2,3,4,5,6]
3. z = (x\*\*3 for x in list)
5. print(next(z))
7. print(next(z))
9. print(next(z))
11. print(next(z))

**Output:**

1

8

27

64

#### Note:- When we call the next(), Python calls \_\_next\_\_() on the function in which we have passed it as a parameter.

In the above program, we have used the **next()** function, which returned the next item of the list.

**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
6. for i in table(15):
7. print(i)

**Output:**

15

30

45

60

75

90

105

120

135

150

In the above example, a generator function is iterating using for loop.

## Advantages of Generators

There are various advantages of Generators. Few of them are given below:

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

**Output:**

Memory in Bytes: 4508

Memory in Bytes: 56

We can observe from the above output that list comprehension is using 4508 bytes of memory, whereas generator expression is using 56 bytes of memory. It means that generator objects are much efficient than the list compression.

### 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, consider the following example:

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

**Output:**

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

.........

..........

315

316

317

Traceback (most recent call last):

File "C:\Users\DEVANSH SHARMA\Desktop\generator.py", line 33, in <module>

print(i)

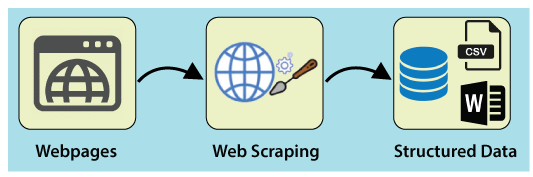
KeyboardInterrupt

In this tutorial, we have learned about the Python Generators.

# Web Scraping Using Python

## What is Web Scraping?

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.

## Is Web Scrapping legal?

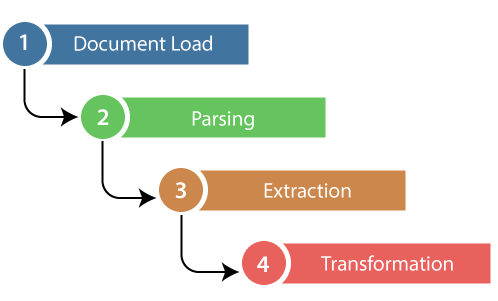
Here the question arises **whether the web scrapping is legal or not**. The answer is that some sites allow it when used legally. Web scraping is just a tool you can use it in the right way or wrong way.

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

There are several tools available to scrap data from websites, such as:

* Scrapping-bot
* Scrapper API
* Octoparse
* Import.io
* Webhose.io
* Dexi.io
* Outwit
* Diffbot
* Content Grabber
* Mozenda
* Web Scrapper Chrome Extension

## Why Web Scrapping?



As we have discussed above, web scrapping is used to extract the data from websites. But we should know how to use that raw data. That raw data can be used in various fields. Let's have a look at the 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**

eb 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**

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

## Why use Python for Web Scrapping?

There are other popular programming languages, but why we choose the [Python](https://www.javatpoint.com/python-tutorial) over other programming languages for web scraping? Below we are describing a list of Python's features that make the most useful programming language 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 and do manipulation.

* **Less Code**

The purpose of the 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.

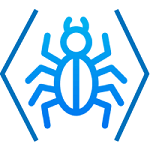
* **Open-Source Community**

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.

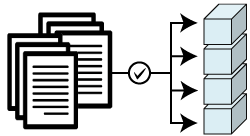
### The basics of web scraping

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. Let's understand about these two components of web scrapping:

* **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. Let's understand the working of 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

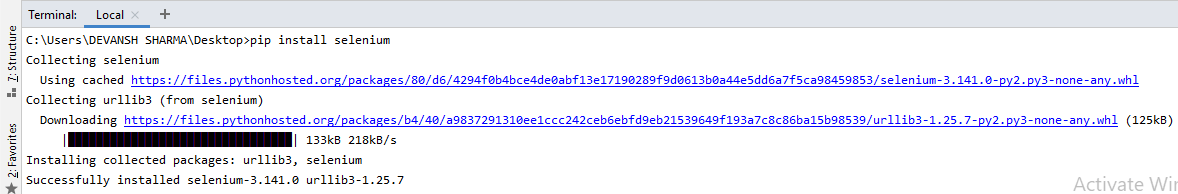
Python has a vast collection of libraries and also provides a very useful library for web scrapping. Let's understand the required library for Python.

**Library used for web scrapping**

* **Selenium-** Selenium is an open-source automated testing library. It is used to check browser activities. To install this library, type the following command in your terminal.

1. pip install selenium

#### Note - It is good to use the PyCharm IDE.



* **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 the parser to provide a natural way of navigating, searching, and modifying the parse tree. The latest version of BeautifulSoup is 4.8.1.

Let's understand the **BeautifulSoup** library in detail.

**Installation of BeautifulSoup**

You can install BeautifulSoup by typing the following command:

1. pip install bs4

**Installing a parser**

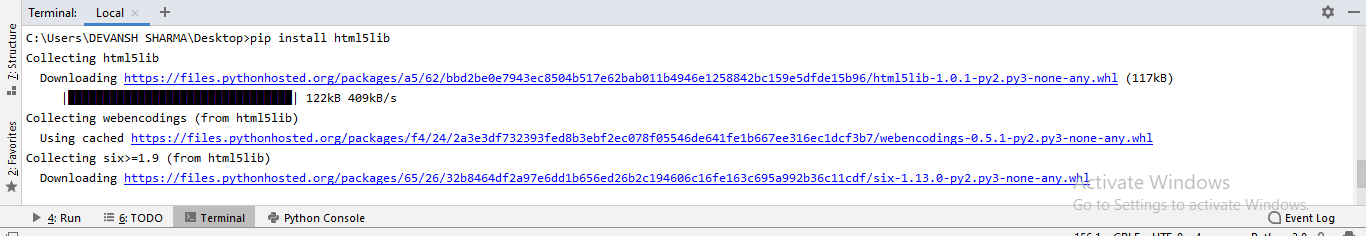
BeautifulSoup supports HTML parser and several third-party Python parsers. You can install any of them according to your dependency. The list of BeautifulSoup's parsers is the following:

|  |  |
| --- | --- |
| **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.

Type the following command in your terminal:

1. pip install html5lib



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)

**Output:**

<class "bs4.element.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

* **Attributes**

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 the tag as dictionary.

1. tag[id]

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

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

* **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**.

1. class\_is\_multi= { '\*' : 'class'}
2. xml\_soup = BeautifulSoup('<p class="body strikeout"></p>', 'xml', multi\_valued\_attributes=class\_is\_multi)
3. xml\_soup.p['class']
4. # [u'body', u'strikeout']

* **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. # u'Extremely bold'
3. type(tag.string)
4. # <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)

**Output:**

?xml version="1.0" encoding="utf-8"?>

# <document><content/><footer>Here's the footer</footer></document>

### 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. Consider the following code:

1. #importing the BeautifulSoup Library
3. importbs4
4. import requests
6. #Creating the requests
8. res = requests.get("https://en.wikipedia.org/wiki/Machine\_learning")
9. print("The object type:",type(res))
11. # Convert the request object to the Beautiful Soup Object
12. soup = bs4.BeautifulSoup(res.text,'html5lib')
13. print("The object type:",type(soup)

**Output:**

The object type <class 'requests.models.Response'>

Convert the object into: <class 'bs4.BeautifulSoup'>

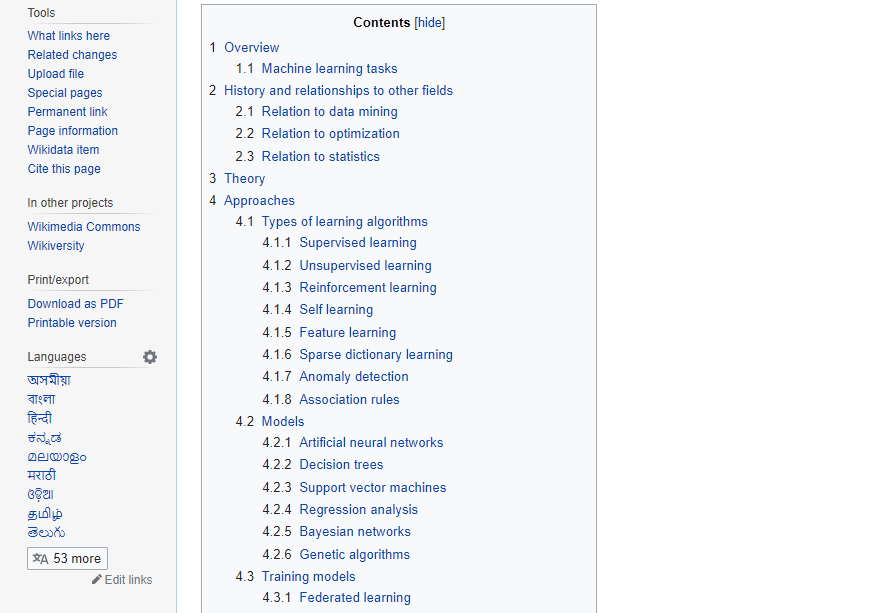
In the following lines of code, we are extracting all headings of a webpage by class name. Here front-end knowledge plays an essential role in inspecting the webpage.

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

**Output:**

Overview,Machine learning tasks,History and relationships to other fields,Relation to data mining,Relation to optimization,Relation to statistics, Theory,Approaches,Types of learning algorithms,Supervised learning,Unsupervised learning,Reinforcement learning,Self-learning,Feature learning,Sparse dictionary learning,Anomaly detection,Association rules,Models,Artificial neural networks,Decision trees,Support vector machines,Regression analysis,Bayesian networks,Genetic algorithms,Training models,Federated learning,Applications,Limitations,Bias,Model assessments,Ethics,Software,Free and open-source software,Proprietary software with free and open-source editions,Proprietary software,Journals,Conferences,See also,References,Further reading,External links,

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.



**Webpage of Wikipedia Learning**

Let's understand another example; we will make a GET request to the URL and create a parse Tree object (soup) with the use of BeautifulSoup and Python built-in **"html5lib"** parser.

Here we will scrap the webpage of given link [(https://www.javatpoint.com/).](https://www.javatpoint.com/) Consider the following code:

1. following code:
2. # importing the libraries
3. from bs4 import BeautifulSoup
4. import requests
6. url="https://www.javatpoint.com/"
8. # Make a GET request to fetch the raw HTML content
9. html\_content = requests.get(url).text
11. # Parse the html content
12. soup = BeautifulSoup(html\_content, "html5lib")
13. print(soup.prettify()) # print the parsed data of html

The above code will display the all html code of javatpoint homepage.

Using the **BeautifulSoup** object, i.e. **soup**, we can collect the required data table. Let's print some interesting information using the **soup** object:

* Let's print the title of the web page.

1. print(soup.title)

**Output:** It will give an output as follow:

<title>Tutorials List - Javatpoint</title>

* In the above output, the HTML tag is included with the title. If you want text without tag, you can use the following code:

1. print(soup.title.text)

**Output:** It will give an output as follow:

Tutorials List - Javatpoint

* We can get the entire link on the page along with its attributes, such as href, title, and its inner Text. Consider the following code:

1. for link in soup.find\_all("a"):
2. print("Inner Text is: {}".format(link.text))
3. print("Title is: {}".format(link.get("title")))
4. print("href is: {}".format(link.get("href")))

**Output:** It will print all links along with its attributes. Here we display a few of them:

href is: https://www.facebook.com/javatpoint

Inner Text is:

The title is: None

href is: https://twitter.com/pagejavatpoint

Inner Text is:

The title is: None

href is: https://www.youtube.com/channel/UCUnYvQVCrJoFWZhKK3O2xLg

Inner Text is:

The title is: None

href is: https://javatpoint.blogspot.com

Inner Text is: Learn Java

Title is: None

href is: https://www.javatpoint.com/java-tutorial

Inner Text is: Learn Data Structures

Title is: None

href is: https://www.javatpoint.com/data-structure-tutorial

Inner Text is: Learn C Programming

Title is: None

href is: https://www.javatpoint.com/c-programming-language-tutorial

Inner Text is: Learn C++ Tutorial

### Demo: Scraping Data from Flipkart Website

In this example, we will scrap the mobile phone prices, ratings, and model name from Flipkart, which is one of the popular e-commerce websites. Following are the prerequisites to accomplish this task:

**Prerequisites:**

* Python 2.x or Python 3.x with **Selenium, BeautifulSoup, Pandas** libraries installed.
* Google - chrome browser
* Scrapping Parser such as html.parser, xlml, etc.

**Step - 1: Find the desired URL to scrap**

The initial step is to find the URL that you want to scrap. Here we are extracting mobile phone details from the flipkart. The URL of this page is https://www.flipkart.com/search?q=iphones&otracker=search&otracker1=search&marketplace=FLIPKART&as-show=on&as=off.

**Step -2: Inspecting the page**

It is necessary to inspect the page carefully because the data is usually contained within the tags. So we need to inspect to select the desired tag. To inspect the page, right-click on the element and click **"inspect"**.

**Step - 3: Find the data for extracting**

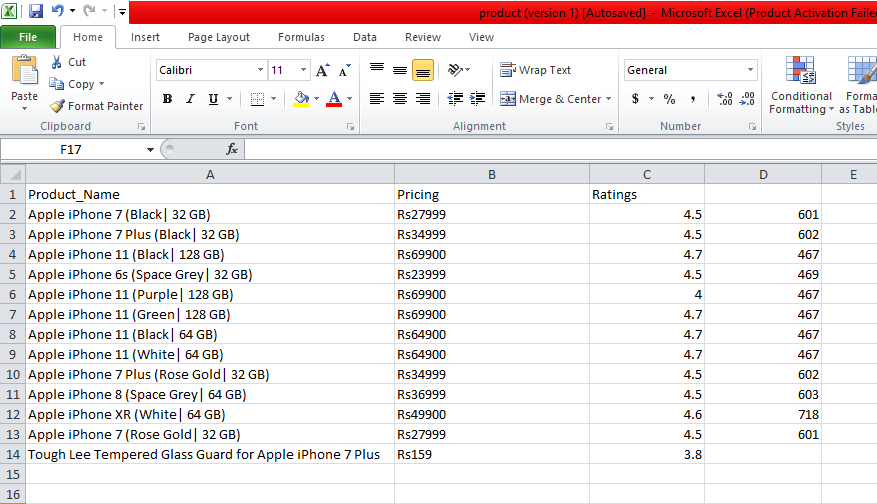
Extract the Price, Name, and Rating, which are contained in the "div" tag, respectively.

**Step - 4: Write the Code**

1. from bs4 import BeautifulSoupas soup
2. from urllib.request import urlopen as uReq
4. # Request from the webpage
5. myurl = "https://www.flipkart.com/search?q=iphones&otracker=search&otracker1=search&marketplace=FLIPKART&as-show=on&as=off"

8. uClient  = uReq(myurl)
9. page\_html = uClient.read()
10. uClient.close()
12. page\_soup = soup(page\_html, features="html.parser")
14. # print(soup.prettify(containers[0]))
16. # This variable held all html of webpage
17. containers = page\_soup.find\_all("div",{"class": "\_3O0U0u"})
18. # container = containers[0]
19. # # print(soup.prettify(container))
20. #
21. # price = container.find\_all("div",{"class": "col col-5-12 \_2o7WAb"})
22. # print(price[0].text)
23. #
24. # ratings = container.find\_all("div",{"class": "niH0FQ"})
25. # print(ratings[0].text)
26. #
27. # #
28. # # print(len(containers))
29. # print(container.div.img["alt"])
31. # Creating CSV File that will store all data
32. filename = "product1.csv"
33. f = open(filename,"w")
35. headers = "Product\_Name,Pricing,Ratings\n"
36. f.write(headers)
38. for container in containers:
39. product\_name = container.div.img["alt"]
41. price\_container = container.find\_all("div", {"class": "col col-5-12 \_2o7WAb"})
42. price = price\_container[0].text.strip()
44. rating\_container = container.find\_all("div",{"class":"niH0FQ"})
45. ratings = rating\_container[0].text
47. # print("product\_name:"+product\_name)
48. # print("price:"+price)
49. # print("ratings:"+ str(ratings))
51. edit\_price = ''.join(price.split(','))
52. sym\_rupee = edit\_price.split("?")
53. add\_rs\_price = "Rs"+sym\_rupee[1]
54. split\_price = add\_rs\_price.split("E")
55. final\_price = split\_price[0]
57. split\_rating = str(ratings).split(" ")
58. final\_rating = split\_rating[0]
60. print(product\_name.replace(",", "|")+","+final\_price+","+final\_rating+"\n")
61. f.write(product\_name.replace(",", "|")+","+final\_price+","+final\_rating+"\n")
63. f.close()

**Output:**



We scrapped the details of the iPhone and saved those details in the CSV file as you can see in the output. In the above code, we put a comment on the few lines of code for testing purpose. You can remove those comments and observe the output.

In this tutorial, we have discussed all basic concepts of web scrapping and described the sample scrapping from the leading online ecommerce site flipkart.

# 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 is easy to learn and the most effective way to interchange the data. It can be used with various programming languages such as [**Python**](https://www.javatpoint.com/python-tutorial)**,** [**Perl**](https://www.javatpoint.com/perl-tutorial)**,** [**Java**](https://www.javatpoint.com/java-tutorial)**,** etc.

[JSON](https://www.javatpoint.com/json-tutorial) mainly supports 6 types of data type In [JavaScript](https://www.javatpoint.com/javascript-tutorial):

* **String**
* **Number**
* **Boolean**
* **Null**
* **Object**
* **Array**

JSON is built on the two structures:

* 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.

JSON data representation is similar to the Python dictionary. Below is an example of JSON data:

1. {
2. "book": [
3. {
4. "id": 01,
5. "language": "English",
6. "edition": "Second",
7. "author": "Derrick Mwiti"
8. ],
9. {
10. {
11. "id": 02,
12. "language": "French",
13. "edition": "Third",
14. "author": "Vladimir"
15. }
16. }

## 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.

This module includes many built-in functions.

Let's have a look at these functions:

1. import json
2. print(dir(json))

**Output:**

['JSONDecodeError', 'JSONDecoder', 'JSONEncoder', '\_\_all\_\_', '\_\_author\_\_', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_path\_\_', '\_\_spec\_\_', '\_\_version\_\_', '\_default\_decoder', '\_default\_encoder', 'codecs', 'decoder', 'detect\_encoding', 'dump', 'dumps', 'encoder', 'load', 'loads', 'scanner']

In this section, we will learn the following methods:

* **load()**
* **loads()**
* **dump()**
* **dumps()**

## Serializing JSON

Serialization is the technique to convert the Python objects to JSON. Sometimes, computer need to process lots of information so it is good to store that information into the file. 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.

Python objects are converted into the following JSON objects. The list is given below:

|  |  |  |
| --- | --- | --- |
| **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.

Let's consider the simple serialization example:

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

**Output:**

{"Name" : "Peter", "Roll\_no" : "0090014" , "Grade" : "A", "Age" : 20, "Subject" : ["Computer Graphics", "Discrete Mathematics", "Data Structure"] }

In the above program, we have opened a file named **data.json** in writing mode. We opened this file in write mode because if the file doesn't exist, it will be created. The **json.dump()** method transforms dictionary into JSON string.

* **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. Let's consider the following example:

1. import json
2. # Key:value mapping
3. student  = {
4. "Name" : "Peter",
5. "Roll\_no" : "0090014",
6. "Grade" : "A",
7. "Age": 20
8. }
9. b = json.dumps(student)
11. print(b)

**Output:**

{"Name": "Peter", "Roll\_no": "0090014", "Grade": "A", "Age": 20}

JSON supports primitive data types, such as strings and numbers, as well as nested list, tuples and objects.

1. import json
3. #Python  list conversion to JSON  Array
4. print(json.dumps(['Welcome', "to", "javaTpoint"]))
6. #Python  tuple conversion to JSON Array
7. print(json.dumps(("Welcome", "to", "javaTpoint")))
9. # Python string conversion to JSON String
10. print(json.dumps("Hello"))
12. # Python int conversion to JSON Number
13. print(json.dumps(1234))
15. # Python float conversion to JSON Number
16. print(json.dumps(23.572))
18. # Boolean conversion to their respective values
19. print(json.dumps(True))
20. print(json.dumps(False))
22. # None value to null
23. print(json.dumps(None))

**Output:**

["Welcome", "to", "javaTpoint"]

["Welcome", "to", "javaTpoint"]

"Hello"

1234

23.572

true

false

null

### 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:

|  |  |  |
| --- | --- | --- |
| **SR.** | **JSON** | **Python** |
| **1.** | Object | dict |
| **2.** | Array | list |
| **3.** | String | str |
| **4.** | number(int) | int |
| **5.** | true | True |
| **6.** | false | False |
| **7.** | null | None |

The above table shows the inverse of the serialized table but technically it is not a perfect conversion of the JSON data. It means that if we encode the object and decode it again after sometime; we may not get the same object back.

Let's take real-life example, one person translates something into Chinese and another person translates back into English, and that may not be exactly translated. Consider the simple example:

1. import json
2. a = (10,20,30,40,50,60,70)
3. print(type(a))
4. b = json.dumps(a)
5. print(type(json.loads(b)))

**Output:**

<class 'tuple'>

<class 'list'>

* **The load() function**

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

1. import json
2. # Key:value mapping
3. student  = {
4. "Name" : "Peter",
5. "Roll\_no" : "0090014",
6. "Grade" : "A",
7. "Age": 20,
8. }
10. with open("data.json","w") as write\_file:
11. json.dump(student,write\_file)
13. with open("data.json", "r") as read\_file:
14. b = json.load(read\_file)
15. print(b)

**Output:**

{'Name': 'Peter', 'Roll\_no': '0090014', 'Grade': 'A', 'Age': 20}

In the above program, we have encoded Python object in the file using **dump()** function. After that we read JSON file using **load()** function, where we have passed **read\_file** as an argument.

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. Consider the following example:

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

**Output:**

['Mathew', 'Peter', [10, 32.9, 80], {'Name': 'Tokyo'}]

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

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

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

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

## Python Pretty Print JSON

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

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

Consider the following example:

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

**Output:**

{

"Age": 23,

"City": "English",

"Name": "Andrew",

"Number": 90014,

"Subject": [

"Data Structure",

"Computer Graphics",

"Discrete mathematics"

]

}

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 the preferred user by decoding it. Encoding is also known as **serialization** and decoding is also called **deserialization**. Encoding and decoding are done for JSON(object) format. Python provides a popular package for such operations. We can install it on Windows by the following command:

1. pip install demjson

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

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

**Example:1 - Encoding using demjson package**

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

**Output:**

[{"Age":20,"Name":"Peter","Subject":"Electronics"}]

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

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

**Output:**

['Peter', 'Smith', 'Ricky', 'Hayden']

In this tutorial, we have learned about the Python JSON. JSON is the most effective way to transmit data between the client and the web server.

# Python Itertools

Itertool is one of the most amazing Python 3 standard libraries. This library has pretty much coolest functions and nothing wrong to say that it is the gem of the Python programing language. Python provides excellent documentation of the itertools but in this tutorial, we will discuss few important and useful functions or iterators of itertools.

The key thing about itertools is that the functions of this library are used to make memory-efficient and precise code.

Before learning the Python itertools, you should have knowledge of the Python iterator and generators. In this article, we will describe itertools for beginners are well as for professionals.

## Introduction

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.

x[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

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

**Output:**

[(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 iter() function](https://www.javatpoint.com/python-iter-function) is used to call on the iterable and return iterator object of the iterable.

1. a = iter('Hello')
2. print(a)

**Output:**

<str\_iterator object at 0x01505FA0>

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

### 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 |

* **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 the number of steps will be skipped. Consider the following example:

1. import itertools
3. for i in itertools.count(10,5):
4. if i == 50:
5. break
6. else:
7. print(i,end=" ")

**Output:**

10 15 20 25 30 35 40 45

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

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=' ')
8. temp = temp+1

**Output:**

1 2 3 1 2 3 1 2 3 1 2

**Example - 2: Using next() function**

1. import itertools
3. val = ['Java', 'T', 'Point']
5. iter = itertools.cycle(val)
7. for i in range(6):
8. # Using next function
9. print(next(iter), end = " ")

**Output:**

Java T Point Java T Point

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

1. import itertools
2. print("Printing the number repeadtly:")
3. print(list(itertools.repeat(40,15)))

**Output:**

[40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40]

**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:

* **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. Consider the following example:

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

**Output:**

Computing cartesian product using repeat Keyword Argument:

[(1, 1), (1, 2), (2, 1), (2, 2)]

Computing cartesian product of the containers:

[('Java', '5'), ('T', '5'), ('point', '5')]

Computing product of the containers:

[('C', 4), ('C', 5), ('D', 4), ('D', 5)]

* **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
3. print("Computing all permutation of the following list")
4. print(list(permutations([3,"Python"],2)))
5. print()
7. print("Permutations of following string")
8. print(list(permutations('AB')))
9. print()
11. print("Permutation of the given container is:")
12. print(list(permutations(range(4),2)))

**Output:**

Computing all permutation of the following list

[(3, 'Python'), ('Python', 3)]

Permutations of following string

[('A', 'B'), ('B', 'A')]

Permutation of the given container is:

[(0, 1), (0, 2), (0, 3), (1, 0), (1, 2), (1, 3), (2, 0), (2, 1), (2, 3), (3, 0), (3, 1), (3, 2)]

* **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)))
3. print()
5. print("Combination of string in sorted order",list(combinations("ZX",2)))
6. print()
8. print("Combination of list in sorted order",list(combinations(range(20),1)))

**Output:**

Combination of list in sorted order(without replacement) [('B', 3)]

Combination of string in sorted order [('Z', 'X')]

Combination of list in sorted order [(0,), (1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,), (9,)]

* **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
3. print("Combination of string in sorted order(with replacement) is:")
4. print(list(combinations\_with\_replacement("XY", 3)))
5. print()
7. print("Combination of list in sorted order(with replacement) is:")
8. print(list(combinations\_with\_replacement([4, 2], 3)))
9. print()
11. print("Combination of container in sorted order(with replacement) is:")
12. print(list(combinations\_with\_replacement(range(3), 2)))

**Output:**

Combination of string in sorted order(with replacement) is:

[('X', 'X', 'X'), ('X', 'X', 'Y'), ('X', 'Y', 'Y'), ('Y', 'Y', 'Y')]

Combination of list in sorted order(with replacement) is:

[(4, 4, 4), (4, 4, 2), (4, 2, 2), (2, 2, 2)]

Combination of container in sorted order(with replacement) is:

[(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)]

### 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:

* **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
4. # initializing list 1
5. list1 = [1, 4, 5, 7, 9, 11]
7. # using accumulate() that will prints the successive summation of elements
8. print("The sum is : ", end="")
9. print(list(itertools.accumulate(list1)))
11. # using accumulate() that will prints the successive multiplication of elements
12. print("The product is : ", end="")
13. print(list(itertools.accumulate(list1, operator.mul)))

16. # using accumulate() that will prints the successive summation of elements
17. print("The sum is : ", end="")
18. print(list(itertools.accumulate(list1)))
20. # using accumulate() that will prints the successive multiplication of elements
21. print("The product is : ", end="")
22. print(list(itertools.accumulate(list1, operator.mul)))

**Output:**

The sum is : [1, 5, 10, 17, 26, 37]

The product is : [1, 4, 20, 140, 1260, 13860]

The sum is : [1, 5, 10, 17, 26, 37]

The product is : [1, 4, 20, 140, 1260, 13860]

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

1. import itertools
3. # declaring list 1
4. list1 = [1, 2, 3, 4]
6. # declaring list 2
7. list2 = [1, 5, 6, 8]
9. # declaring list 3
10. list3 = [9, 10, 11, 12]
12. # using chain() function that will to print all elements of lists
13. print("The output is : ", end="")
14. print(list(itertools.chain(list1, list2, list3)))

**Output:**

The output is: [1, 2, 3, 4, 1, 5, 6, 8, 9, 10, 11, 12]

* **dropwhile(func, seq)** - It starts printing the character only after the **func**. Consider the following argument:

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

**Output:**

The output is : [5, 7, 8]

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

1. import itertools
3. # declaring list
4. list1 = [12, 14, 15, 27, 28]
6. # using filterfalse() iterator that will print false values
7. print("The Output is: ", end="")
8. print(list(itertools.filterfalse(lambda x: x % 2 == 0, list1)))

**Output:**

The Output is : [15, 27]

* **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. # Declaring list
3. list1 = [12, 34, 65, 73, 80, 19, 20]
4. # using islice() iterator that will slice the list acc. to given argument
5. # starts printing from 3nd index till 8th skipping 2
6. print("The sliced list values are : ", end="")
7. print(list(itertools.islice(list1, 2, 8, 2)))

**Output:**

The sliced list values are : [34, 73, 19]

* **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. Consider the following example.

1. import itertools
3. # Declaring list that contain tuple as element
4. list1 = [(10, 20, 15), (18, 40, 19), (53, 42, 90), (16, 12, 27)]
6. # using starmap() iterator for selection value acc. to function
7. # selects max of all tuple values
8. print("The values acc. to function are : ", end="")
9. print(list(itertools.starmap(max, list1)))

**Output:**

The values acc. to function are : [20, 40, 90, 27]

* **takewhile(func, iterable)** - It is visa-versa of **dropwhile().** It will print values until it returns false condition. Consider the following example:

1. import itertools
3. # Defining a list
4. list1 = [20, 42, 64, 77, 8, 10, 20]
6. # takewhile() iterator is used  to print values till condition return false.
7. print("Print until 1st false value returned : ", end="")
8. print(list(itertools.takewhile(lambda x: x % 2 == 0, list1)))

**Output:**

The list values until false value return : [20, 42, 64]

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

1. import itertools
3. # Declaring list
4. li = [1, 2, 3, 4, 5, 6, 7]
6. # storing list in iterator
7. iti = iter(li)
8. # using tee() iterator to create a list of iterators
9. # Creating list of 3 iterators having similar values.
10. it = itertools.tee(iti, 3)
11. # It will print object of iterator
12. print(it)
13. print("The iterators are : ")
14. for i in range(0, 2):
15. print(list(it[i]))

**Output:**

(<itertools.\_tee object at 0x01B88D88>, <itertools.\_tee object at 0x01B88DA8>, <itertools.\_tee object at 0x01B88BA8>)

The iterators are :

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

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

* **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(" The combined value of iterrables is :")
3. print(\*(itertools.zip\_longest('Java', 'Tpoint', fillvalue='\_')))

**Output:**

The combined value of iterables is :

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

In this tutorial, we have discussed several useful iterators along with itertools.

# Python Multiprocessing

In this article, we will learn how we can achieve multiprocessing using Python. We also discuss its advanced concepts.

### What is 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 the tasks between more than one process.

Processing units share the main memory and peripherals to process programs simultaneously. Multiprocessing Application breaks into smaller parts and runs independently. Each process is allocated to the processor by the operating system.

[Python](https://www.javatpoint.com/python-tutorial) provides the built-in package called multiprocessing which supports swapping processes. Before working with the multiprocessing, we must aware with the process object.

### Why Multiprocessing?

Multiprocessing is essential to perform the multiple tasks within the Computer system. Suppose a computer without multiprocessing or single processor. We assign various processes to that system at the same time.

It will then have to interrupt the previous task and move to another to keep all processes going. It is as simple as a chef is working alone in the kitchen. He has to do several tasks to cook food such as cutting, cleaning, cooking, kneading dough, baking, etc.

Therefore, multiprocessing is essential to perform several task at the same time without interruption. It also makes easy to track all the tasks. That is why the concept of multiprocessing is to arise.

* 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.

In the multiprocessing, the [CPU](https://www.javatpoint.com/central-processing-unit) can assign multiple tasks at one each task has its own processor.

## Multiprocessing In Python

Python provides the multiprocessing module to perform multiple tasks within the single system. It offers a user-friendly and intuitive API to work with the multiprocessing.

Let's understand the simple example of multiple processing.

Example -

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()

**Output:**

'Hello !! Welcome to Python Tutorial'

**Explanation:**

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.

Let's understand the following example of the multiprocessing with arguments.

**Example - 2**

1. # Python multiprocessing example
2. # importing the multiprocessing module
4. import multiprocessing
5. def cube(n):
6. # This function will print the cube of the given number
7. print("The Cube is: {}".format(n \* n \* n))
9. def square(n):
10. # This function will print the square of the given number
11. print("The Square is: {}".format(n \* n))
13. if \_\_name\_\_ == "\_\_main\_\_":
14. # creating two processes
15. process1 = multiprocessing.Process(target= square, args=(5, ))
16. process2 = multiprocessing.Process(target= cube, args=(5, ))
18. # Here we start the process 1
19. process1.start()
20. # Here we start process 2
21. process2.start()
23. # The join() method is used to wait for process 1 to complete
24. process1.join()
25. # It is used to wait for process 1 to complete
26. process2.join()
28. # Print if both processes are completed
29. print("Both processes are finished")

**Output:**

The Cube is: 125

The Square is: 25

Both processes are finished

**Explanation -**

In the above example, We created the two functions - the **cube()** function calculates the given number's cube, and the **square()** function calculates the square of the given number.

Next, we defined the process object of the Process class that has two arguments. The first argument is a **target** that represents the function to be executed, and the second argument is **args that** represents the argument to be passed within the function.

1. process1 = multiprocessing.Process(target= square, args=(5, ))
2. process2 = multiprocessing.Process(target= cube, args=(5, ))

We have used the **start()** method to start the process.

1. process1.start()
2. process2.start()

As we can see in the output, it waits to completion of **process one** and then **process 2**. The last statement is executed after both processes are finished.

## 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.

Let's see the simple example of a get number of CPUs currently in the system.

**Example -**

1. import multiprocessing
2. print("The number of CPU currently working in system : ", multiprocessing.cpu\_count())

**Output:**

('The number of CPU currently woking in system : ', 32)

The above number of CPUs can vary for your pc. For us, the number of cores is 32.

## Python Multiprocessing Using Queue Class

We know that Queue is important part of the data structure. Python multiprocessing is precisely the same as the data structure queue, which based on the "First-In-First-Out" concept. Queue generally stores the Python object and plays an essential role in sharing data between processes.

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. Let's understand the following example.

**Example -**

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

**Output:**

pushing items to the queue:

('item no: ', 1, ' ', 'Apple')

('item no: ', 2, ' ', 'Orange')

('item no: ', 3, ' ', 'Guava')

('item no: ', 4, ' ', 'Papaya')

('item no: ', 5, ' ', 'Banana')

popping items from the queue:

('item no: ', 0, ' ', 'Apple')

('item no: ', 1, ' ', 'Orange')

('item no: ', 2, ' ', 'Guava')

('item no: ', 3, ' ', 'Papaya')

('item no: ', 4, ' ', 'Banana')

**Explanation -**

In the above code, we have imported the **Queue** class and initialized the list named fruits. Next, we assigned a **count** to 1. The count variable will count the total number of elements. Then, we created the queue object by calling the **Queue()** method. This object will used to perform operations in the Queue. In for loop, we inserted the elements one by one in the queue using the **put()** function and increased the count by 1 with each iteration of loop.

## 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.

In the following example, we will merge all the multiprocessing classes together. Let's see the below example.

**Example -**

1. from multiprocessing import Lock, Process, Queue, current\_process
2. import time
3. import queue

6. def jobTodo(tasks\_to\_perform, complete\_tasks):
7. while True:
8. try:
10. # The try block to catch task from the queue.
11. # The get\_nowait() function is used to
12. # raise queue.Empty exception if the queue is empty.
14. task = tasks\_to\_perform.get\_nowait()
16. except queue.Empty:
18. break
19. else:
21. # if no exception has been raised, the else block will execute
22. # add the task completion

25. print(task)
26. complete\_tasks.put(task + ' is done by ' + current\_process().name)
27. time.sleep(.5)
28. return True

31. def main():
32. total\_task = 8
33. total\_number\_of\_processes = 3
34. tasks\_to\_perform = Queue()
35. complete\_tasks = Queue()
36. number\_of\_processes = []
38. for i in range(total\_task):
39. tasks\_to\_perform.put("Task no " + str(i))
41. # defining number of processes
42. for w in range(total\_number\_of\_processes):
43. p = Process(target=jobTodo, args=(tasks\_to\_perform, complete\_tasks))
44. number\_of\_processes.append(p)
45. p.start()
47. # completing process
48. for p in number\_of\_processes:
49. p.join()
51. # print the output
52. while not complete\_tasks.empty():
53. print(complete\_tasks.get())
55. return True

58. if \_\_name\_\_ == '\_\_main\_\_':
59. main()

**Output:**

Task no 2

Task no 5

Task no 0

Task no 3

Task no 6

Task no 1

Task no 4

Task no 7

Task no 0 is done by Process-1

Task no 1 is done by Process-3

Task no 2 is done by Process-2

Task no 3 is done by Process-1

Task no 4 is done by Process-3

Task no 5 is done by Process-2

Task no 6 is done by Process-1

Task no 7 is done by Process-3

## 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)**. Consider the following example of a multiprocessing Pool.

**Example -**

1. from multiprocessing import Pool
2. import time
4. w = (["V", 5], ["X", 2], ["Y", 1], ["Z", 3])

7. def work\_log(data\_for\_work):
8. print(" Process name is %s waiting time is %s seconds" % (data\_for\_work[0], data\_for\_work[1]))
9. time.sleep(int(data\_for\_work[1]))
10. print(" Process %s Executed." % data\_for\_work[0])

13. def handler():
14. p = Pool(2)
15. p.map(work\_log, w)
17. if \_\_name\_\_ == '\_\_main\_\_':
18. handler()

**Output:**

Process name is V waiting time is 5 seconds

Process V Executed.

Process name is X waiting time is 2 seconds

Process X Executed.

Process name is Y waiting time is 1 seconds

Process Y Executed.

Process name is Z waiting time is 3 seconds

Process Z Executed.

Let's understand another example of the multiprocessing Pool.

**Example - 2**

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

**Output:**

[1, 8, 27]

## Proxy Objects

The proxy objects are referred to as shared objects which reside in a different process. This object is also called as a proxy. Multiple proxy objects might have a similar referent. A proxy object consists of various methods which are used to invoked corresponding methods of its referent. Below is the example of proxy objects.

**Example -**

1. from multiprocessing import Manager
2. manager = Manager()
3. l = manager.list([i\*i for i in range(10)])
4. print(l)
5. print(repr(l))
6. print(l[4])
7. print(l[2:5])

**Output:**

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

<ListProxy object, typeid 'list' at 0x7f063621ea10>

16

[4, 9, 16]

The proxy objects are picklable so we can pass them between processes. These objects are also used for level of control over the synchronization.

## Commonly Used Functions of Multiprocessing

So far, we have discussed the basic concepts of multiprocessing using Python. Multiprocessing is a broad topic itself and essential for performing various tasks within a single system. We are defining a few essential functions that are commonly used to achieve 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 and 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() | This method 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 |

# How to Calculate Distance between Two Points using GEOPY.

The **geopy** is a Python library which helps to calculate geographical distance. In this tutorial, we will discuss different methods of how the user can calculate the distance between two places on the earth.

First, the user has to install the **geopy** by using the following command:

1. pip install geopy

After successful installation, we are ready to work with the geopy library.

## Calculate Distance between Two Points

Below are the 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.

**Example:**

1. # First, import the geodesic module from the geopy library
2. from geopy.distance import geodesic as GD
4. # Then, load the latitude and longitude data for New York & Texas
5. New\_York = (40.7128, 74.0060)
6. Texas = (31.9686, 99.9018)
8. # At last, print the distance between two points calculated in kilo-metre
9. print ("The distance between New York and Texas is: ", GD(New\_York, Texas).km)

**Output:**

The distance between New York and Texas is: 2507.14797665193

### 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. The following example shows how the user can calculate great circle distance by using longitude and latitude data of two points.

**Example:**

1. # First, import the great\_circle module from the geopy library
2. from geopy.distance import great\_circle as GC
4. # Then, load the latitude and longitude data for New York & Texas
5. New\_York = (40.7128, 74.0060)
6. Texas = (31.9686, 99.9018)
8. # At last, print the distance between two points calculated in kilo-metre
9. print ("The distance between New York and Texas is: ", GC(New\_York, Texas).km)

**Output:**

The distance between New York and Texas is: 2503.045970189156

### Method 3: By using Haversine Formula

The orthodromic distance is used for calculating the shortest distance between two latitudes and longitudes points on the earth's surface.

Using this method, the user needs to have the coordinates of two points **(P and Q).**

First, they have to convert the values of latitude and longitude points from decimal degrees to radians and then divide the values of latitude and longitude by (180/π). The user should use the value of "π = 22/7". Then, the value of (180/π) will be "57.29577". If the user wants to calculate the distance in miles, they can use the value of the radius of Earth, that is, "3,963". And if the user wants to calculate the distance in Kilo-metre, they can use the value "6,378.80".

**Formulas:**

1. How to calculate the value of latitude in radians:
2. The value of Latitude in Radian: Latitude (La1) = La1 / (180/?)
3. OR
4. The value of Latitude in Radian: Latitude (La1) = La1 / 57.29577
5. How to calculate the value of longitude in radians:
6. The value of Longitude in Radian: Longitude (Lo1) = Lo1 / (180/?)
7. OR
8. The value of Longitude in Radian: Longitude (Lo1) = Lo1 / 57.29577

The user needs the coordinates of P point and Q points in terms of longitude and latitude, then using the above formula for converting them into radians.

Now, calculate the distance between two points by using the following formula.

**Formula:**

**For miles:**

1. Distance (D) = 3963.0 \* arccos[(sin(La1) \* sin(La2)) + cos(La1) \* cos(La2) \* cos(Lo2 - Lo1)]

**For kilometre:**

1. Distance (D) = 3963.0 \* arccos[(sin(La1) \* sin(La2)) + cos(La1) \* cos(La2) \* cos(Lo2 - Lo1)]

Thus, the user can calculate the shortest distance between the two given points on Earth by using Haversine Formula.

**Example:**

1. from math import radians, cos, sin, asin, sqrt
2. # For calculating the distance in Kilometres
3. def distance\_1(La1, La2, Lo1, Lo2):
5. # The math module contains the function name "radians" which is used for converting the degrees value into radians.
6. Lo1 = radians(Lo1)
7. Lo2 = radians(Lo2)
8. La1 = radians(La1)
9. La2 = radians(La2)
11. # Using the "Haversine formula"
12. D\_Lo = Lo2 - Lo1
13. D\_La = La2 - La1
14. P = sin(D\_La / 2)\*\*2 + cos(La1) \* cos(La2) \* sin(D\_Lo / 2)\*\*2
16. Q = 2 \* asin(sqrt(P))
18. # The radius of earth in kilometres.
19. R\_km = 6371
21. # Then, we will calculate the result
22. return(Q \* R\_km)


26. # driver code
27. La1 = 40.7128
28. La2 = 31.9686
29. Lo1 = -74.0060
30. Lo2 = -99.9018
31. print ("The distance between New York and Texas is: ", distance\_1(La1, La2, Lo1, Lo2), "K.M")
32. # For calculating the distance in Miles
33. def distance\_2(La1, La2, Lo1, Lo2):
35. # The math module contains the function name "radians" which is used for converting the degrees value into radians.
36. Lo1 = radians(Lo1)
37. Lo2 = radians(Lo2)
38. La1 = radians(La1)
39. La2 = radians(La2)
41. # Using the "Haversine formula"
42. D\_Lo = Lo2 - Lo1
43. D\_La = La2 - La1
44. P = sin(D\_La / 2)\*\*2 + cos(La1) \* cos(La2) \* sin(D\_Lo / 2)\*\*2
46. Q = 2 \* asin(sqrt(P))
47. # The radius of earth in Miles.
48. R\_Mi = 3963
50. # Then, we will calculate the result
51. return(Q \* R\_Mi)
52. print ("The distance between New York and Texas is: ", distance\_2(La1, La2, Lo1, Lo2), "Miles")

**Output:**

The distance between New York and Texas is: 2503.04243426357 K.M

The distance between New York and Texas is: 1556.985899699659 Miles

## Conclusion

In this tutorial, we have discussed various methods for calculating the distance between two points on the earth's surface by using the geopy library. We have shown examples of each method.

# Gmail API in Python

In this tutorial, we are going to learn about Gmail API in Python, and we will also learn how we can use Gmail APIs in Python to perform many Gmail operations such as sending an email, searching an email, deleting an email, etc. For this, we will learn to set up Gmail API in our Python script. First, let us have a brief of Gmail API and its basic introduction.

## Gmail APIs

Gmail is the most popular mail service in today's world, and it is used by almost all of us and many organizations. Over past years, many Gmail features are enhanced with the use of AI, including suggestions while composing emails and security features (detecting fraud and spam emails).

Gmail API is APIs based on [RESTful](https://www.javatpoint.com/restful-web-services-tutorial) APIs that allow its users to interact with our Gmail account, and it helps us to use its features using a Python script.

### Prerequisites of Using Gmail APIs in Python

We must fulfil the following requirements for using Gmail APIs in our Python script:

* We should have a Python version higher or equal to 2.6.
* We must have a google account with Gmail service enabled of it.
* The system must be installed **BeautifulSoup** library (if not, then we should use 'pip install bsp' syntax in the command terminal to install it in our device).
* We should have basic knowledge of Google OAuth libraries and Google API client.

### Installation of Required libraries:

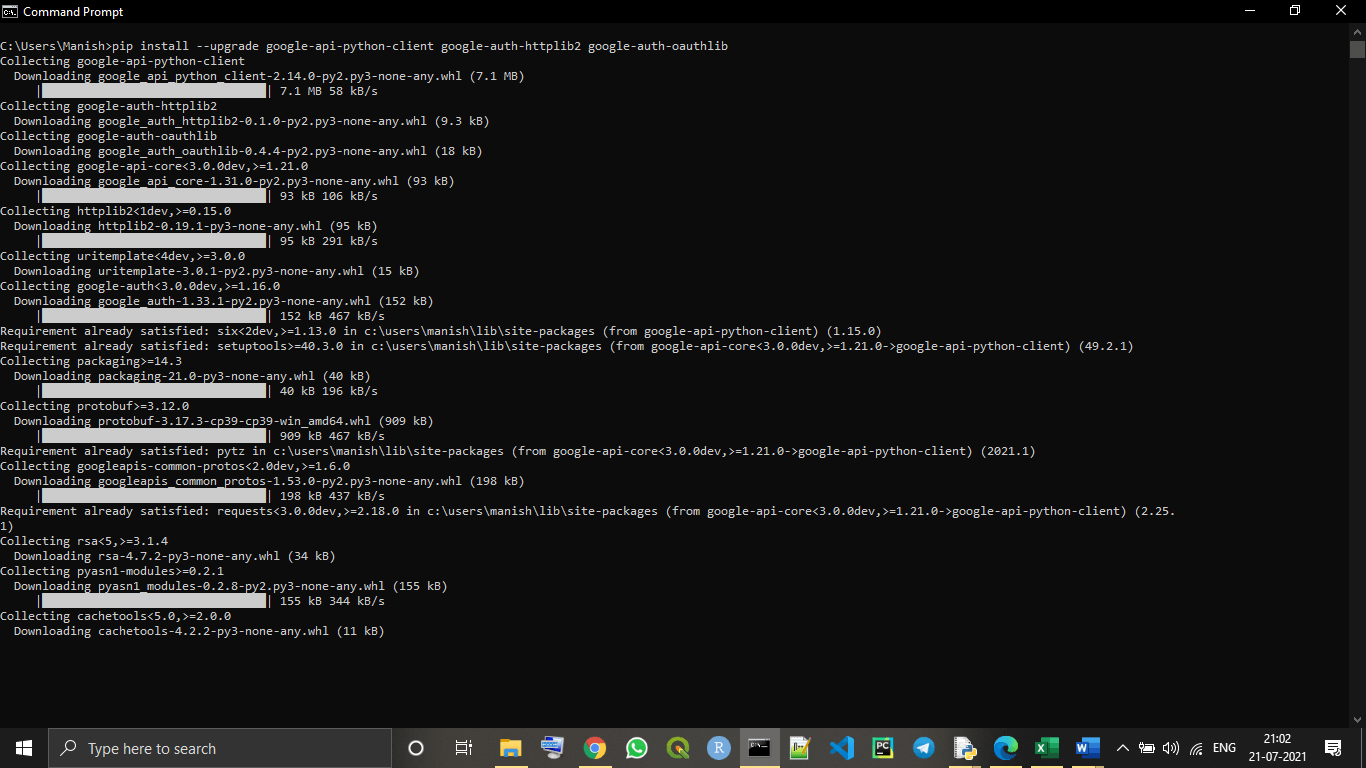
Before enabling the Gmail APIs to use them in our [Python](https://www.javatpoint.com/python-tutorial) script, let's first install the pre-required libraries in our system. To install the pre-required libraries for enabling the Gmail APIs, we should follow the following steps:

**Step 1:** Open the command prompt terminal of the system and make sure that our device has an active internet connection.

**Step 2:** Write down the following command in the terminal:

1. pip install --upgrade google-api-python-client google-auth-httplib2 google-auth-oauthlib

Now, press enter to start the installation of libraries.



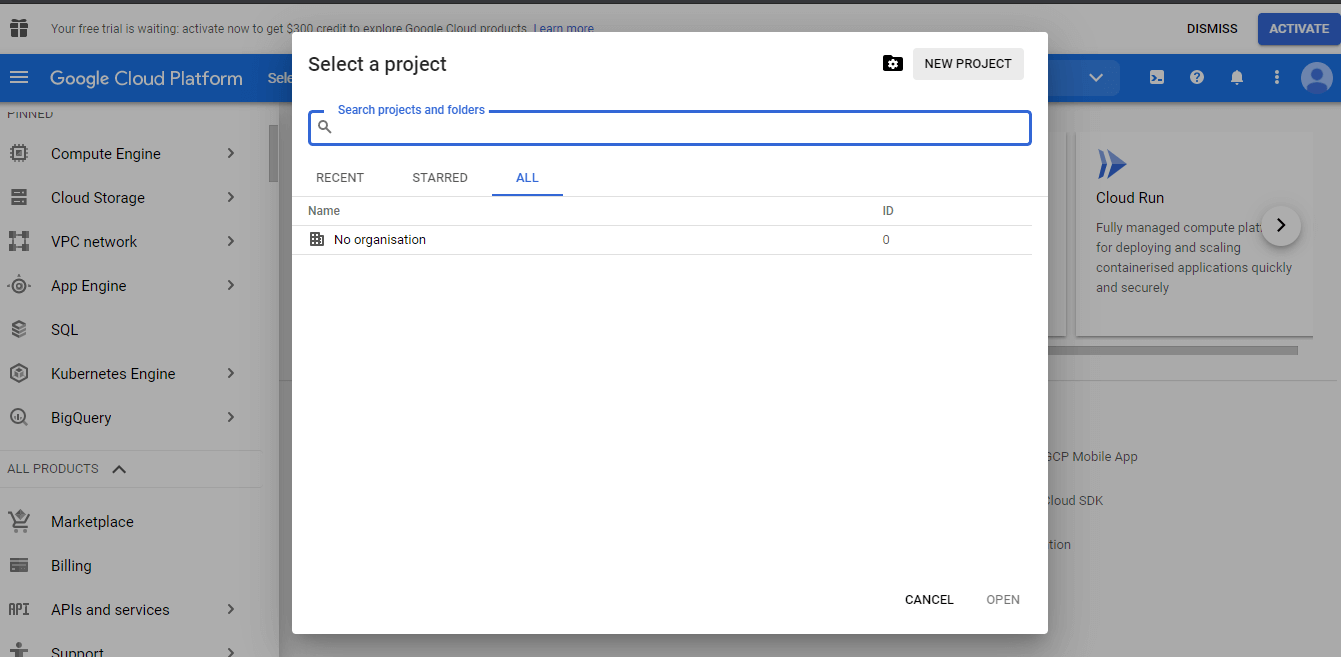
As we can see that, the pre-required libraries for enabling Gmail APIs are successfully installed in our system. Now, we can proceed with enabling Gmail APIs part in this tutorial.

### Enabling Gmail APIs in our device

We have to follow the following given steps to enable Gmail APIs in our device so that we can use these APIs in our Python script:

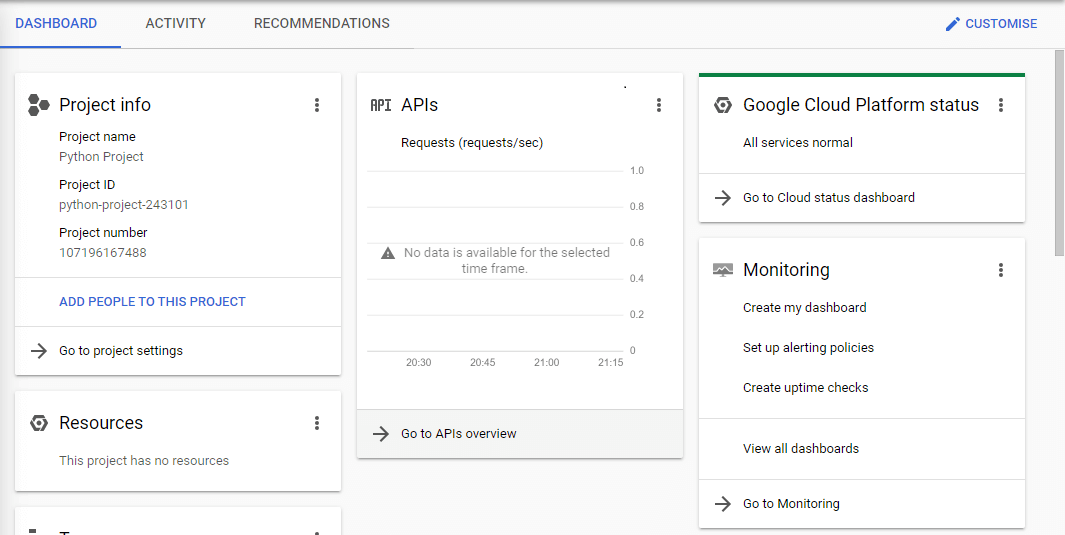
**Step 1: Creating New Project on Google Cloud console:**

In this step, first, we have to login into the Google cloud console **(https://console.cloud.google.com/?pli=1)** with our Google account, and then we have to click on 'New Project' to create a new project.

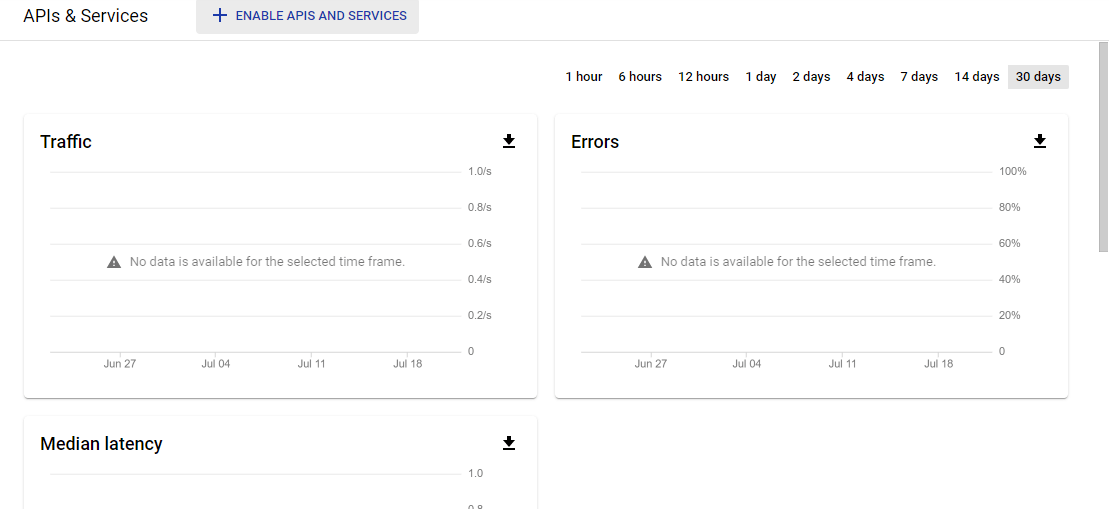


If we already have an existing project, then we can also continue with the existing project.

**Step 2:** Now, we have to go to the API and services option from the Project menu that we have created.

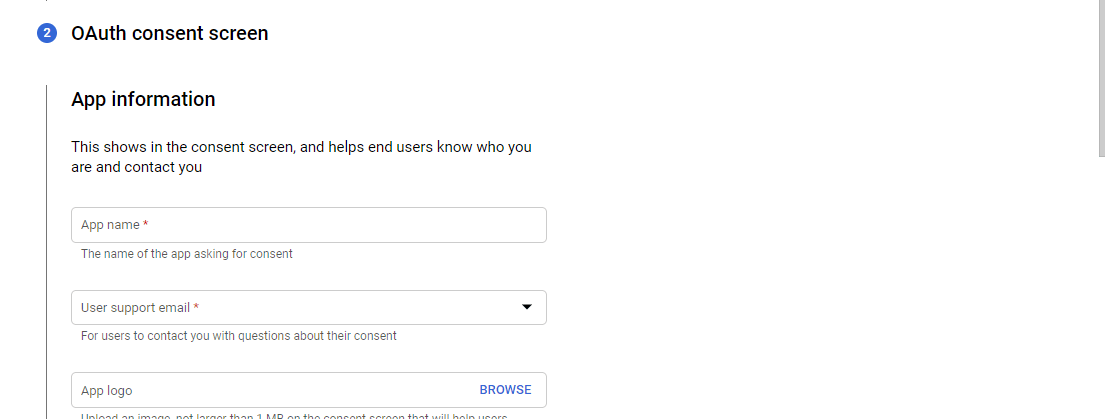


**Step 3:** Now, we can see the option 'Enable Gmail API and services,' and we have to choose this option to enable Gmail APIs for the project.



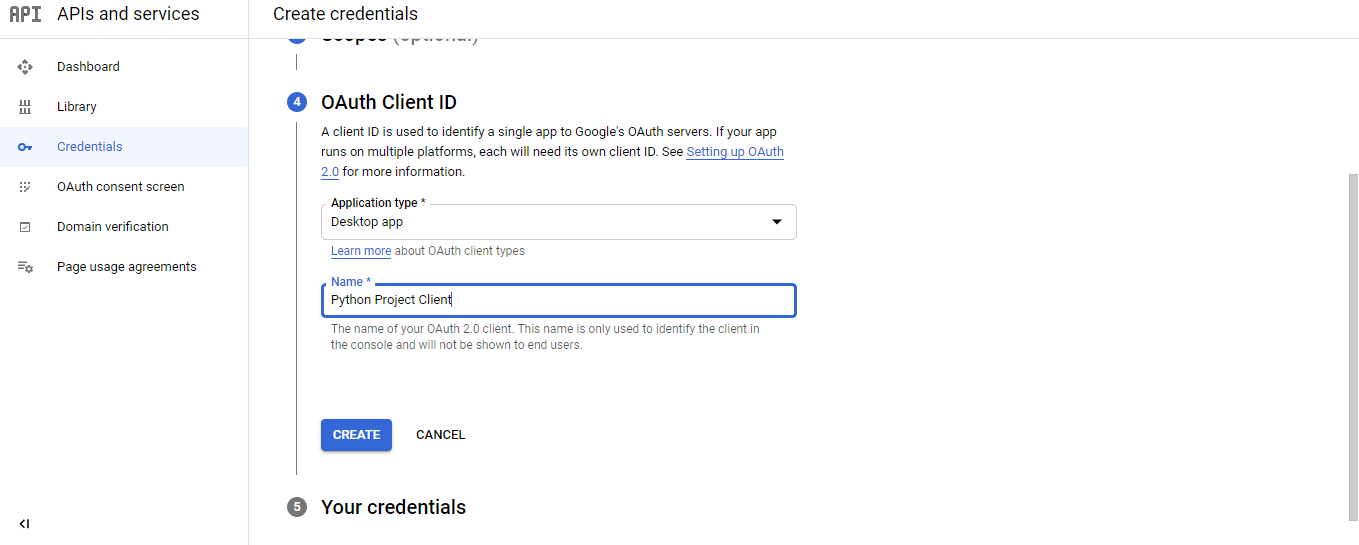
**Step 4: Configuration of Consent screen:**

Now, in this step, we will configure the consent screen of the project we created by clicking on the **'OAuth Consent Screen'** option given in the menu. We can only see this option if the consent screen is not already configured.



**Step 5:** Now, we have to enter the application name of our choice and save the project.

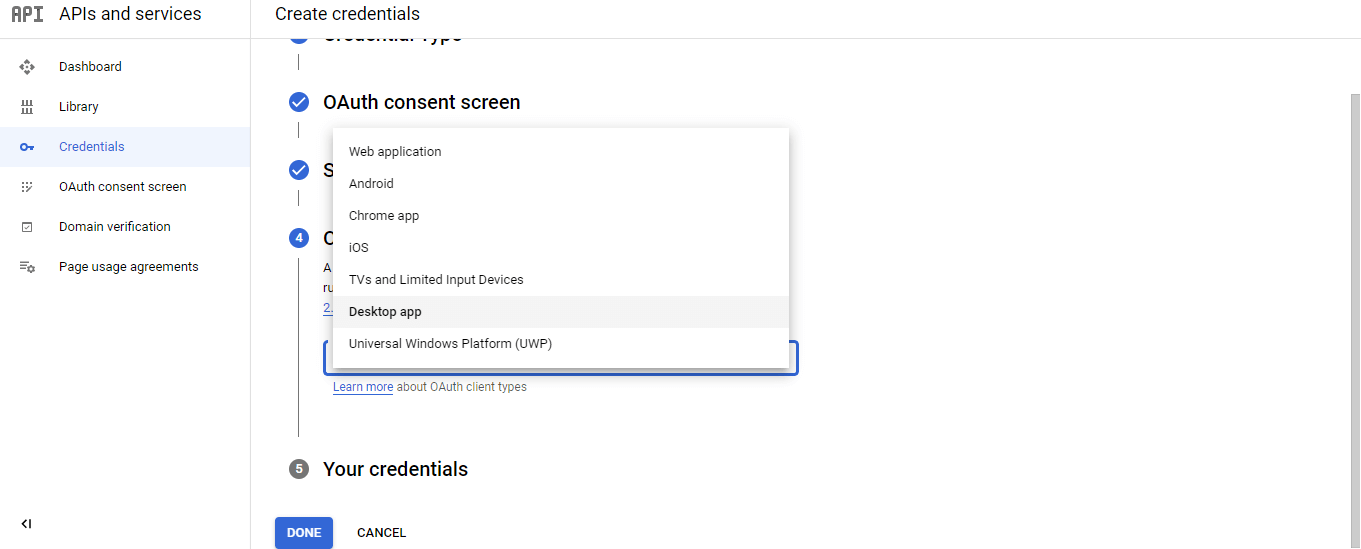
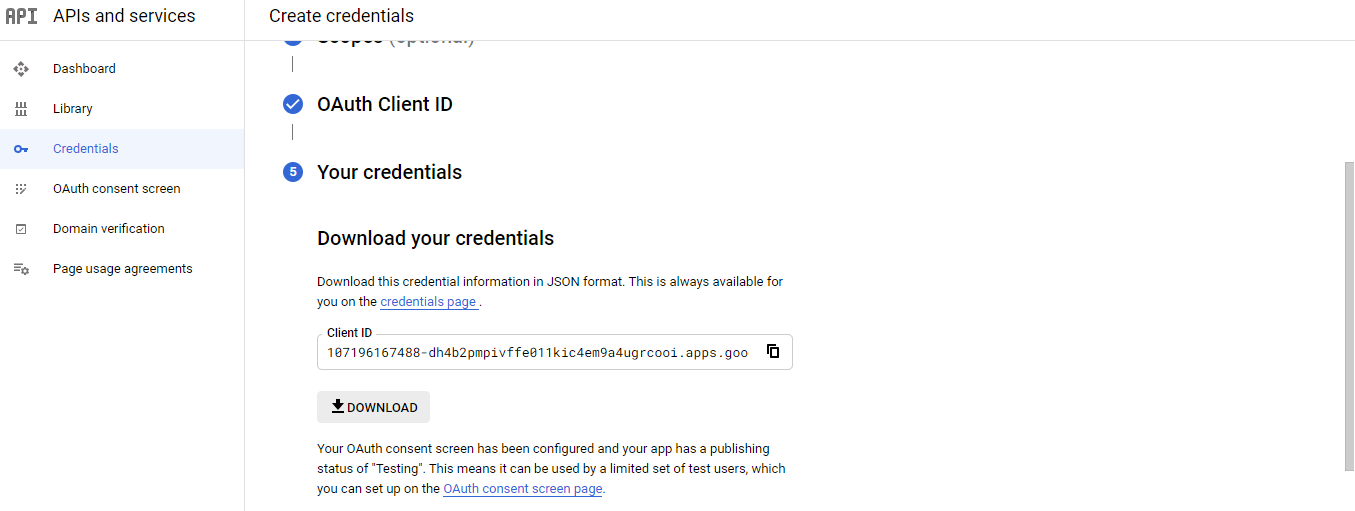
**Step 6:** Now, click on the credentials option and go to credentials.



**Step 7: Creating an OAuth Client ID:**

Now, we click on the **'create credentials'** option and go to the OAuth Client ID to create it.

We perform this by following the below sequential procedure to create a new OAuth Client ID for our project:

* First, we choose the application type as the desktop application for the project.  
  
* After that, we enter the application name (can be the same as we have set in the above steps or can be different) and click on the create button.
* Now, the OAuth client ID will be created for our project, and we download it and save it with the 'credentials.json' name and format for future references.  
  

Now, we are done with all the steps of enabling Gmail APIs, and we can start using Gmail APIs in our Python script.

#### Note: We have to save the client ID and password so that we can use them in future references if required.

**Importing Necessary Modules**

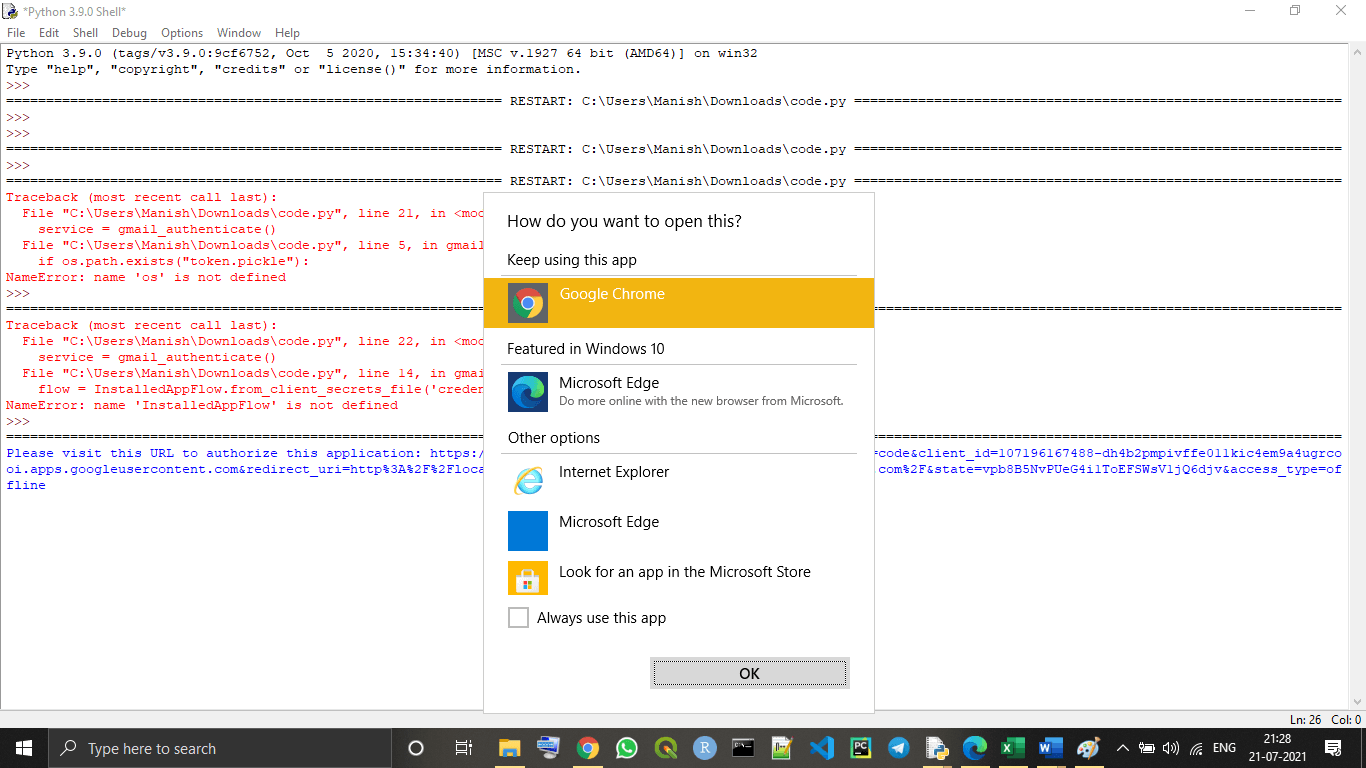
Now, we have set up all the necessary APIs and we should forward with importing all the necessary modules. Let's see the below example of importing modules.

**Example -**

1. # Importing os and pickle module in program
2. import os
3. import pickle
4. # Creating utils for Gmail APIs
5. from googleapiclient.discovery import build
6. from google\_auth\_oauthlib.flow import InstalledAppFlow
7. from google.auth.transport.requests import Request
8. # Importing libraries for encoding/decoding messages in base64
9. from base64 import urlsafe\_b64decode, urlsafe\_b64encode
10. # Importing libraries for dealing with the attachment of MIME types in Gmail
11. from email.mime.text import MIMEText
12. from email.mime.multipart import MIMEMultipart
13. from email.mime.image import MIMEImage
14. from email.mime.audio import MIMEAudio
15. from email.mime.base import MIMEBase
16. from email.mime.multipart import MIMEMultipart
17. from mimetypes import guess\_type as guess\_mime\_type
19. # Request all access from Gmail APIs and project
20. SCOPES = ['https://mail.google.com/']
21. OurEmailID = 'OurMail@gmail.com' # giving our Gmail Id
23. # using a default function to authenticate Gmail APIs
24. def authenticateGmailAPIs():
25. creds = None
26. # Authorizing the Gmail APIs with tokens of pickles
27. if os.path.exists("token.pickle"): # using if else statement
28. with open("token.pickle", "rb") as token:
29. creds = pickle.load(token)
30. # If there are no valid credentials available in device, we will let the user sign in manually
31. if not creds or not creds.valid:
32. if creds and creds.expired and creds.refresh\_token:
33. creds.refresh(Request())
34. else:
35. flow = InstalledAppFlow.from\_client\_secrets\_file('client\_secret\_107196167488-dh4b2pmpivffe011kic4em9a4ugrcooi.apps.googleusercontent.com.json', SCOPES) # downloaded credential name
36. creds = flow.run\_local\_server(port=0) # running credentials
37. # Save the credentials for the next run
38. with open("token.pickle", "wb") as token:
39. pickle.dump(creds, token)
40. return build('Gmail', 'v1', credentials=creds) # using Gmail to authenticate
42. # Get the Gmail API service by calling the function
43. service = authenticateGmailAPIs()

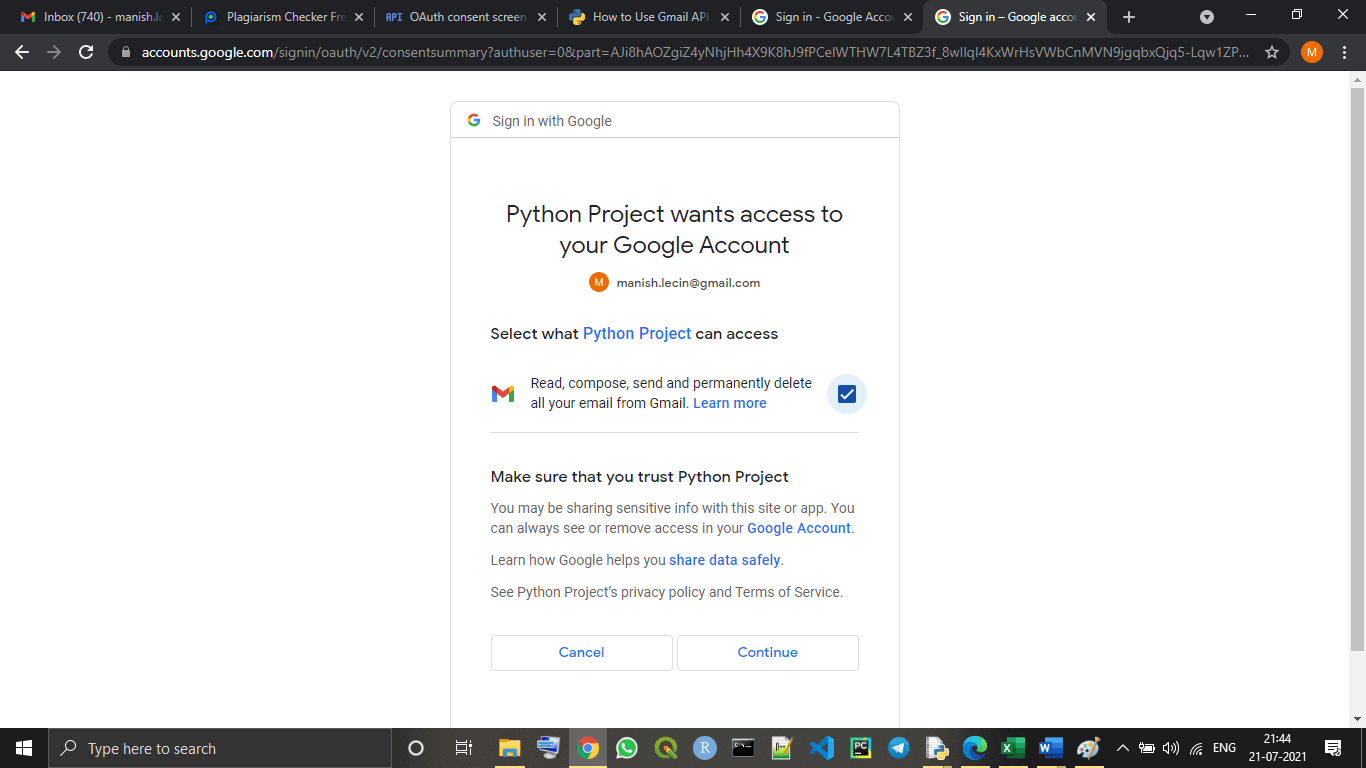
**Output:**

Please visit this URL to authorize this application: https://accounts.google.com/o/oauth2/auth?response\_type=code&client\_id=107196167488-dh4b2pmpivffe011kic4em9a4ugrcooi.apps.googleusercontent.com&redirect\_uri=http%3A%2F%2Flocalhost%3A55991%2F&scope=https%3A%2F%2Fmail.google.com%2F&state=kfXlNyjvbKetyUK0op7OF9WY7shrKS&access\_type=offline

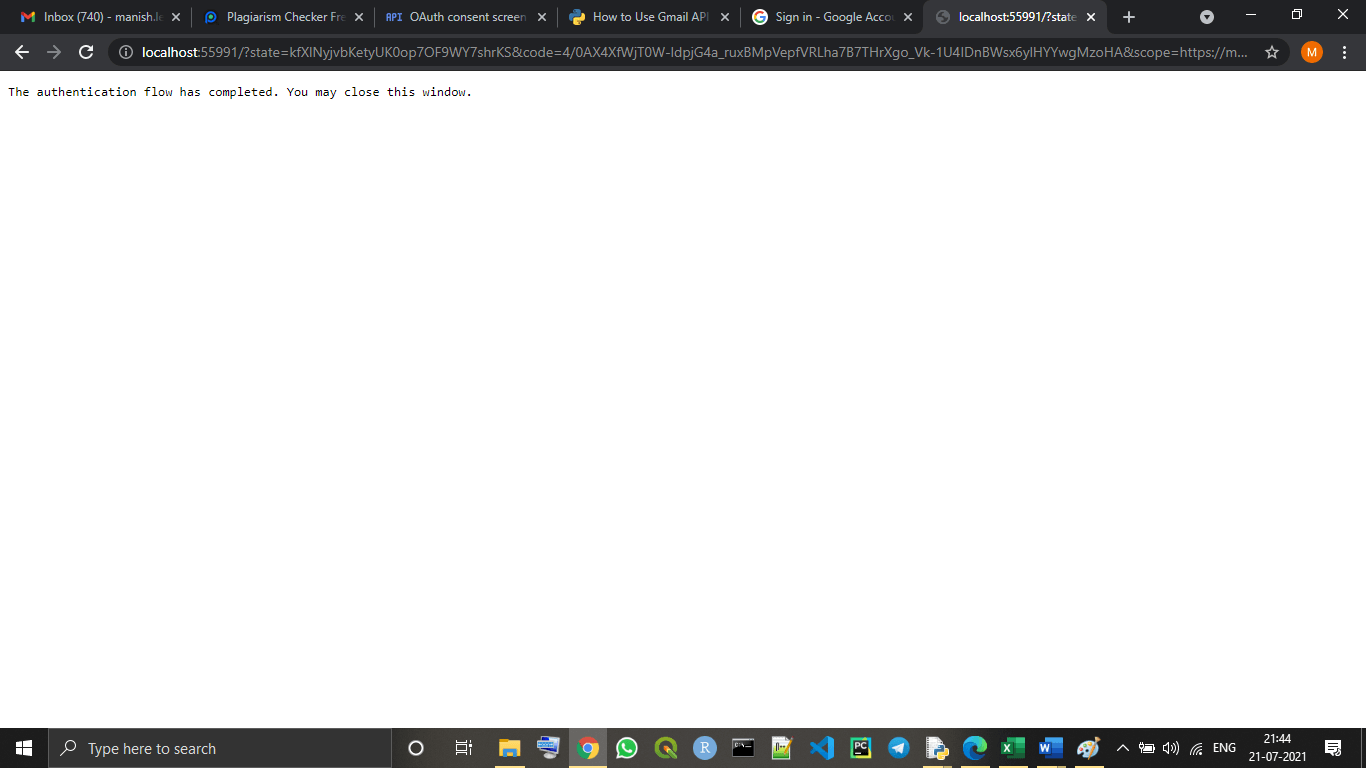


**Explanation -**

When we run the above given program, we will see an option to choose the browser, as we can see in the above image, and if we don't see an option like this, we need to click on the link given in the output. Then, we can select the browser of our choice or the default browser of the system to continue the process. Now, when we select the browser, we will be redirected to our browser and can see the following tab opened in it:



Now, we tick the checkbox option shown in the dialogue box to give the required permissions, and then, we will have to click on the continue option. After clicking on continue, we can see the following window will open in the same tab:



As the window is showing, the authentication part for enabling Gmail API is completed, and we have linked our Gmail account with the project for Gmail APIs we created.

#### Note: Of course, we have to put our mail that we can connect to Gmail APIs and use for future references for working with Gmail APIs, in the place of 'OurMail@gmail.com' as provided in the above program.

## Performing Actions using Gmail APIs in Python

Now, we have completely set up and enabled Gmail APIs in our project with Python script. Now, we can perform many actions from our Gmail account with a Python program.

We can perform the following Gmail actions with our Python script using Gmail APIs in it:

* Sending an email
* Searching an email
* Deleting an email or entire emails history
* Reading an email
* Marking read/unread an email etc.

In this tutorial, we will only about sending an email using Gmail APIs in our Python program, and we will learn to write the code to perform this action with Python script.

### Sending an email

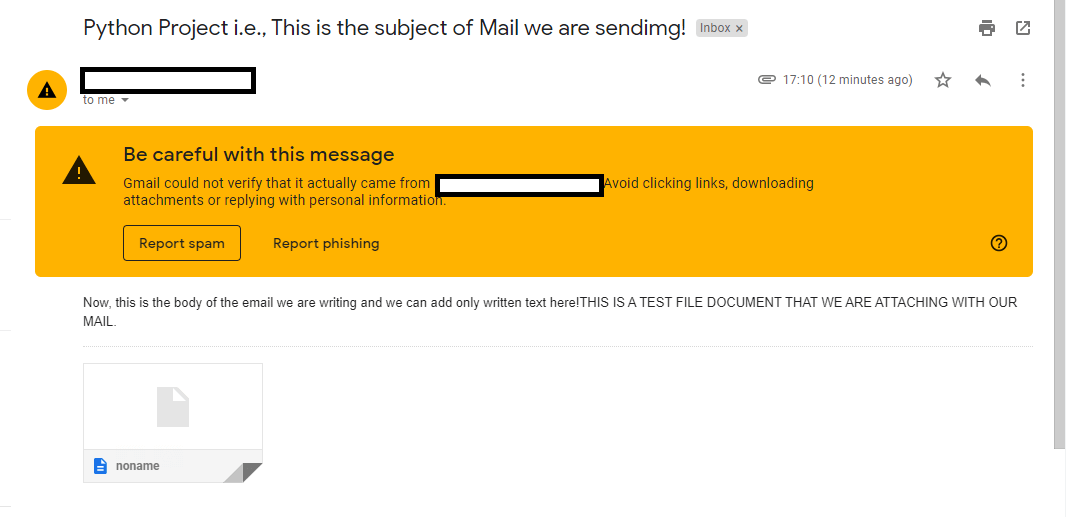
We can simply write and send an email by writing a Python program and using enabled Gmail APIs in it. Here, in this section, we will write a Python program by which we can send emails from our Gmail account just by running the program.

Look at the following Python program for a better understanding of it:

**Example -**

1. # importing os and pickle module in program
2. import os
3. import pickle
4. # Creating utils for Gmail APIs
5. from googleapiclient.discovery import build
6. from google\_auth\_oauthlib.flow import InstalledAppFlow
7. from google.auth.transport.requests import Request
8. # Importing libraries for encoding/decoding messages in base64
9. from base64 import urlsafe\_b64decode, urlsafe\_b64encode
10. # Importing libraries for dealing with the attachment of MIME types in Gmail
11. from email.mime.text import MIMEText
12. from email.mime.multipart import MIMEMultipart
13. from email.mime.image import MIMEImage
14. from email.mime.audio import MIMEAudio
15. from email.mime.base import MIMEBase
16. from email.mime.multipart import MIMEMultipart
17. from mimetypes import guess\_type as guess\_mime\_type
19. # Request all access from Gmail APIs and project
20. SCOPES = ['https://mail.google.com/'] # providing the scope for Gmail APIs
21. OurEmailID = 'OurMail@gmail.com' # giving our Gmail Id
23. # using a default function to authenticate Gmail APIs
24. def authenticateGmailAPIs():
25. creds = None
26. # authorizing the Gmail APIs with tokens of pickles
27. if os.path.exists("token.pickle"): # using if else statement
28. with open("token.pickle", "rb") as token:
29. creds = pickle.load(token)
30. # if there are no valid credentials available in device, we will let the user sign in manually
31. if not creds or not creds.valid:
32. if creds and creds.expired and creds.refresh\_token:
33. creds.refresh(Request())
34. else:
35. flow = InstalledAppFlow.from\_client\_secrets\_file('client\_secret\_107196167488-dh4b2pmpivffe011kic4em9a4ugrcooi.apps.googleusercontent.com.json', SCOPES) # downloaded credential name
36. creds = flow.run\_local\_server(port=0) # running credentials
37. # save the credentials for the next run
38. with open("token.pickle", "wb") as token:
39. pickle.dump(creds, token)
40. return build('gmail', 'v1', credentials=creds) # using Gmail to authenticate
42. # Get the Gmail API service by calling the function
43. ServicesGA = authenticateGmailAPIs()
45. # Using a default funnction to add attachments in Mail
46. def AddAttachment(mail, NameofFile):
47. content\_type, encoding = guess\_mime\_type(NameofFile)
48. if content\_type is None or encoding is not None: # defining none file type attachment
49. content\_type = 'application/octet-stream'
50. main\_type, sub\_type = content\_type.split('/', 1)
51. if main\_type == 'text': # defining text file type attachment
52. fp = open(NameofFile, 'rb') # opening file
53. msg = MIMEText(fp.read().decode(), \_subtype = sub\_type)
54. fp.close()
55. elif main\_type == 'image': # defining image file type attachment
56. fp = open(NameofFile, 'rb')
57. msg = MIMEImage(fp.read(), \_subtype = sub\_type)
58. fp.close()
59. elif main\_type == 'audio': # defining audio file type attachment
60. fp = open(NameofFile, 'rb')
61. msg = MIMEAudio(fp.read(), \_subtype = sub\_type) # reading file
62. fp.close()
63. else:
64. fp = open(NameofFile, 'rb')
65. msg = MIMEBase(main\_type, sub\_type)
66. msg.set\_payload(fp.read())
67. fp.close() # closing file
68. NameofFile = os.path.basename(NameofFile)
69. msg.add\_header('Content-Disposition', 'attachment', NameofFile = NameofFile)
70. mail.attach(msg) # composing the mail with given attachment
72. # Creating mail with a default function
73. def CreateMail(RecieverMail, SubofMail, BodyofMail, attachments=[]): # various import content of mail as function's parameter
74. # Using if else to check if there is any attachment in mail or not
75. if not attachments: # no attachment is given in the mail
76. mail = MIMEText(BodyofMail) # Body of Mail
77. mail['to'] = RecieverMail # mail ID of Reciever
78. mail['from'] = OurEmailID # our mail ID
79. mail['subject'] = SubofMail # Subject of Mail
80. else: # attachment is given in the mail
81. mail = MIMEMultipart()
82. mail['to'] = RecieverMail
83. mail['from'] = OurEmailID
84. mail['subject'] = SubofMail
85. mail.attach(MIMEText(BodyofMail))
86. for NameofFile in attachments:
87. AddAttachment(mail, NameofFile)
88. return {'raw': urlsafe\_b64encode(mail.as\_bytes()).decode()}
90. # Creating a default function to send a mail
91. def SendMail(ServicesGA, RecieverMail, SubofMail, BodyofMail, attachments=[]):
92. return ServicesGA.users().messages().send(
93. userId = "me",
94. body = CreateMail(RecieverMail, SubofMail, BodyofMail, attachments)
95. ).execute() # Body of the mail with execute() function
97. # Sending an email by adding important content, i.e., Reciever's mail, Subject, Body, etc.
98. SendMail(ServicesGA, "Reciever@gmail.com", "Python Project i.e., This is the subject of Mail we are sendimg!",
99. "Now, this is the body of the email we are writing and we can add only written text here!", ["test.txt", "client\_secret\_107196167488-dh4b2pmpivffe011kic4em9a4ugrcooi.apps.googleusercontent.com.json"]) # calling out default SendMail() function

**Output:**



If we put our mail in the place of the receiver's mail, i.e., Reciever@gmail.com, we will find that the mail is actually sent to the mail we entered as receiver's mail when we run the program, same as what we can see in the above output image.

## Conclusion

To use the Gmail APIs with our Python script or simply in Python, first, we have to enable them, and create a Project in Google cloud with our Gmail account.

We can also perform many other actions like reading, deleting, etc., using Gmail APIs in our Python program like sending emails. We can also modify many things into our Gmail account that we authenticated with our Gmail APIs project, just by running our Python scripts (enabled with Gmail APIs).

# How to Plot the Google Map using folium package in Python

The folium package is built on the **data wrangling** strengths of the Python ecosystem and the mapping strengths of the **Leaflet.js library** of JavaScript language. The user can manipulate their data by using Python and then visualize it by using **Leaflet.js** map through folium package. Folium package is an easy approach of visualizing the data on Leaflet.js map, which has been manipulated by using Python.

### Required Module and Libraries

**Folium:** The user can install the Folium package by using the following command.

1. pip install folium

**Geopy:** The geopy module of Python makes it easy for [Python](https://www.javatpoint.com/python-tutorial) users to locate the coordinates of landmarks, cities, countries on the earth's surface. For installing the geopy module, the user can use the following command:

1. pip install geopy

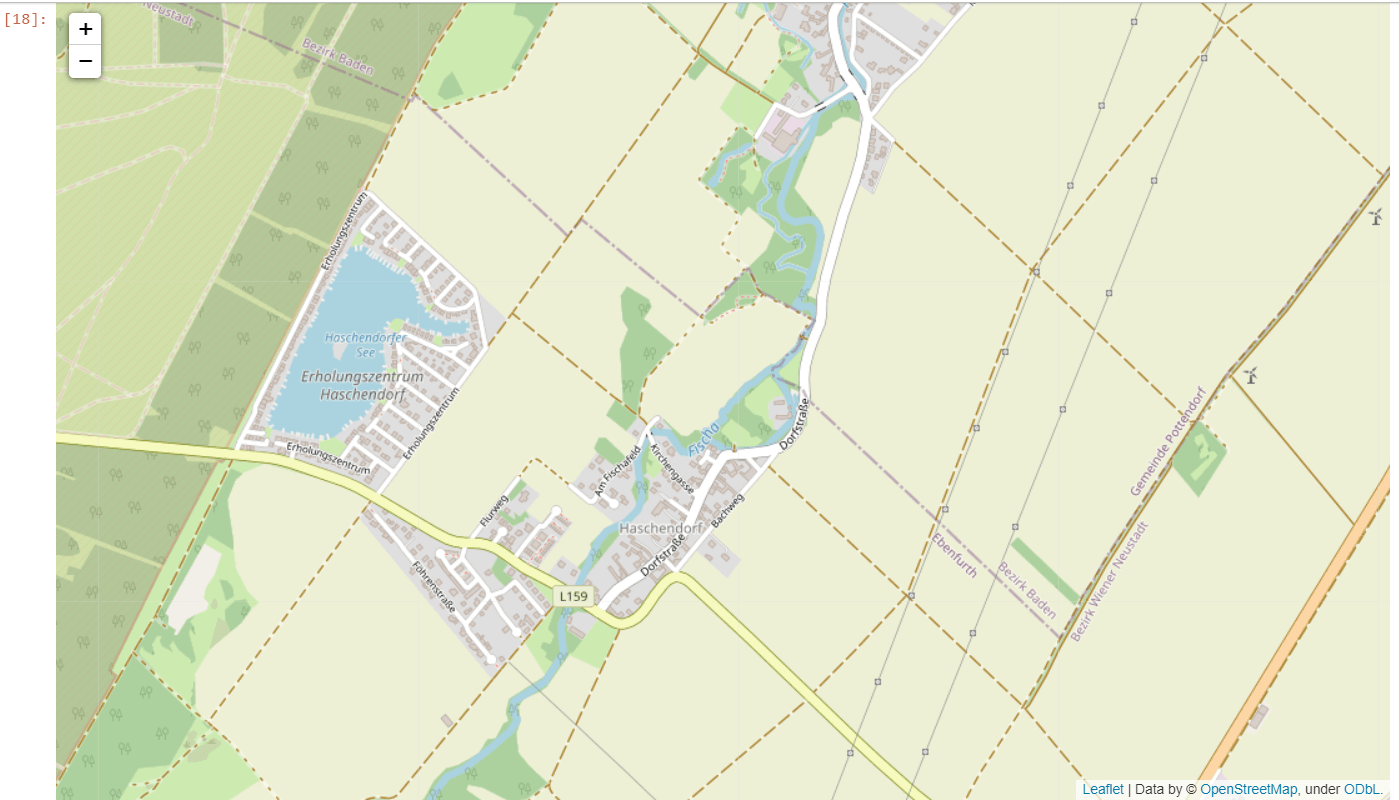
After successful installation of the both libraries, we follow the below steps to plot Google map.

### Step 1: Create the Base map

The user can create the base map by using the following program:

1. import os
2. # First, import folium package
3. import folium
4. from geopy.geocoders import Nominatim as NT
5. # Initialize Nominatim API
6. geo\_locator = NT(user\_agent = "geoapiExercises")
7. # write the place
8. place\_1 = "Yemen"
10. location\_1 = geo\_locator.geocode(place\_1)
11. # now, it will search for the location by using the latitude and longitude, with zoom\_start = 15
12. user\_map1 = folium.Map(location = [location\_1.longitude, location\_1.latitude],
13. zoom\_start = 15 )
14. # At last, open the base map
15. user\_map1

**Output:**

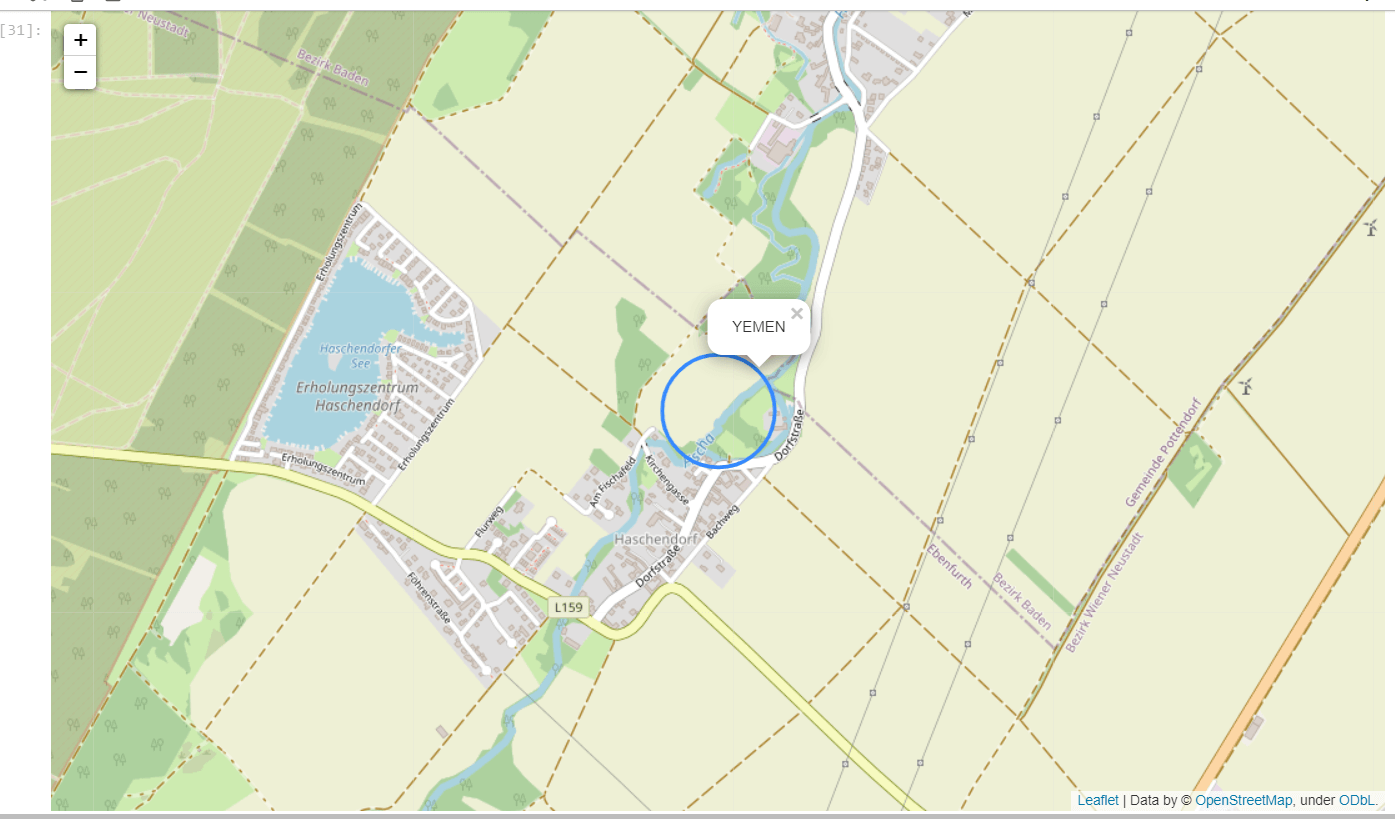


### Step 2: Add a Circular Marker

The user can mark the area with the circle and popup text by using the following code:

1. import folium
3. from geopy.geocoders import Nominatim as NT
5. geo\_locator = NT(user\_agent = "geoapiExercises")
7. place\_1 = "Yemen"
9. location\_1 = geo\_locator.geocode(place\_1)
11. user\_map1 = folium.Map(location = [location\_1.longitude, location\_1.latitude],
12. zoom\_start = 15 )
14. # CircleMarker with radius
15. folium.CircleMarker(location = [location\_1.longitude, location\_1.latitude],
16. radius = 45, popup = ' YEMEN ').add\_to(user\_map1)
18. # Now, open the Map with circular Mark
19. user\_map1

**Output:**



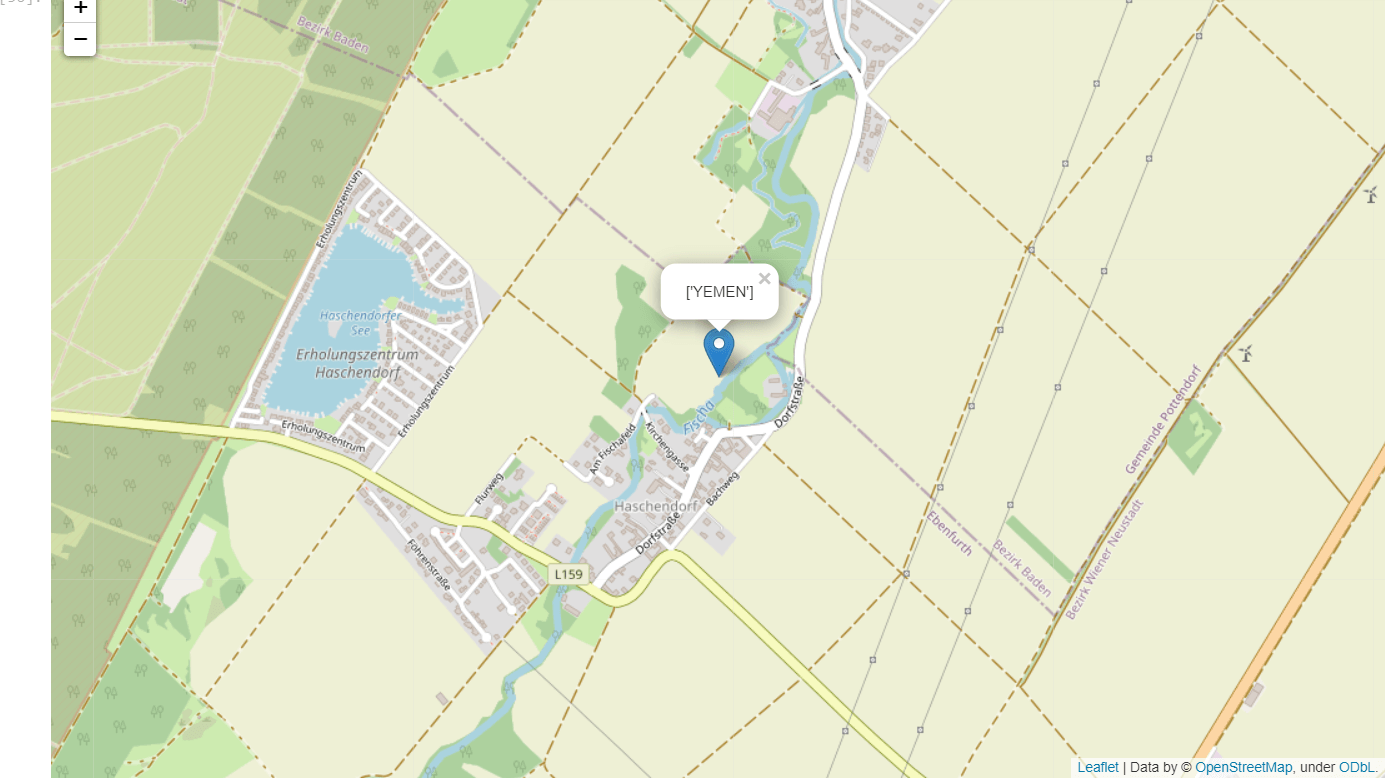
### Step 3: Add the simple marker for the parachute style marker with the popup text

The user can use the following code.

**Example -**

1. import os
2. import folium
4. from geopy.geocoders import Nominatim as NT
6. geo\_locator = NT(user\_agent = "geoapiExercises")
8. place\_1 = "Yemen"
10. location\_1 = geo\_locator.geocode(place\_1)
12. user\_map1 = folium.Map(location = [location\_1.longitude, location\_1.latitude],
13. zoom\_start = 15)
14. #Now, we will pass the string in popup parameter
15. folium.Marker([location\_1.longitude, location\_1.latitude],
16. popup = ['YEMEN']).add\_to(user\_map1)
17. # now, open the map
18. user\_map1

**Output:**



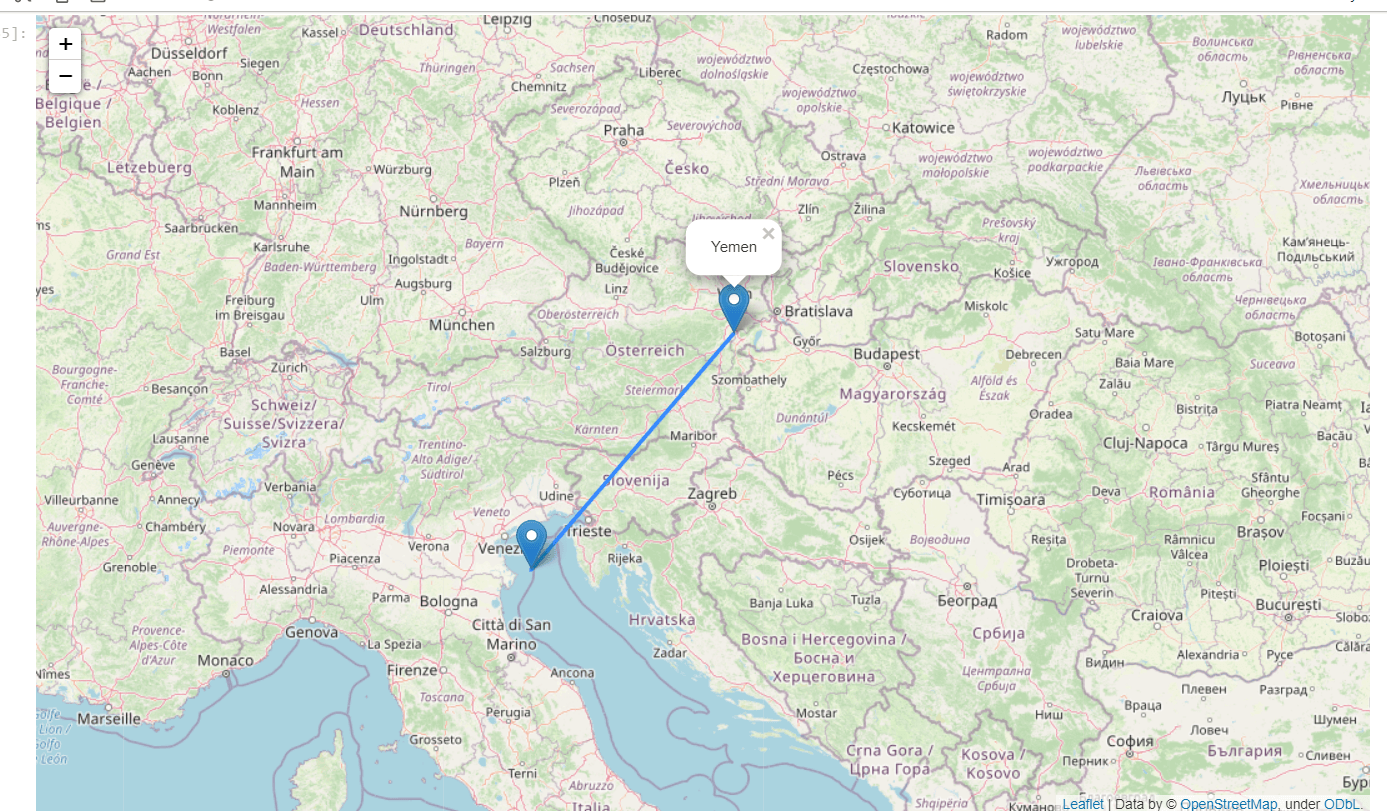
### Step 4: Add the line on the map

The user can use the following code for adding the line on the map to join the two coordinates.

**Example -**

1. # First, import folium package
2. import folium
3. import os
4. from geopy.geocoders import Nominatim as NT
6. geo\_locator = NT(user\_agent = "geoapiExercises")
8. place\_1 = "Aden"
9. place\_2 = "Yemen"
10. location\_1 = geo\_locator.geocode(place\_1)
11. location\_2 = geo\_locator.geocode(place\_2)
12. user\_map1 = folium.Map(location = [location\_1.longitude, location\_1.latitude],
13. zoom\_start = 6)
15. folium.Marker([location\_1.longitude, location\_1.latitude],
16. popup = ['Aden']).add\_to(user\_map1)
18. folium.Marker([location\_2.longitude, location\_2.latitude],
19. popup = 'Yemen').add\_to(user\_map1)
21. # Now, we will add the line on the map by using Polyline method .
22. # it will connect both coordinates by the line
24. folium.PolyLine(locations = [[location\_1.longitude, location\_1.latitude], [location\_2.longitude, location\_2.latitude]],
25. line\_opacity = 0.5).add\_to(user\_map1)
26. # now, open the map
27. user\_map1

**Output:**



**Explanation**

We used the **geopy** library to get the latitude and longitude of the location. Then we used the "**folium.map**" method of the **folium** package for creating the base of Google Maps.

In step 2, we used "**folium.CircleMarker**" for marking the circular mark on the location with the pop-up text. In step 3, we used "**folium.Marker**" to add a parachute style mark on the mentioned location. In the last step, we used "**folium.PolyLine**" for joining two marks on two different locations on the map.

## Conclusion

In this tutorial, we have shown how the user can Plot the Google map and add different required functionalities on the map like a circular mark, parachute mark, pop-up text, and the line joining the two coordinates on the map.

# Grid Search in Python

In this tutorial, we will discuss the Grid Search for the purpose of hyperparameter tuning. We will also learn about the working of Grid Search along with the implementation of it in optimizing the performance of the method of Machine Learning (ML).

**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 the Machine Learning model grows in complexity.

So let us begin by understanding Grid Search.

## 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.

## Installing the required libraries

Before we start implementing the Grid Search in the [Python programming language](https://www.javatpoint.com/python-tutorial), let us briefly discuss some of the necessary libraries and frameworks that need to be installed in the system.

These libraries and frameworks are as follows:

1. Python 3
2. [NumPy](https://www.javatpoint.com/numpy-tutorial)
3. [Pandas](https://www.javatpoint.com/python-pandas)
4. [Keras](https://www.javatpoint.com/keras)
5. Scikit-Learn

They are all quite simple to install. We can use the pip installer in order to install these libraries as shown below:

1. $ pip install numpy tensorflow pandas scikit-learn keras

#### Note: If any issues arise while executing any package, try reinstalling and referring to each package's official documentation.

Now, let us begin implementing the Grid Search in Python

## Implementation of Grid Search in Python

In the following section, we will understand how to implement Grid Search on an actual application. We will simply be executing the code and discuss in-depth regarding the section where Grid Search comes in rather than discussing Machine Learning and Data Pre-processing section.

So, let's get started.

We will use the Diet Dataset containing data regarding the height and weight of different people based on various attributes such as gender, age, and type of diet. We can directly import the data from an online resource with the help of the Pandas **read\_csv()** function.

But before that, we have to import the required packages:

**File: mygrid.py**

1. import sys
2. import pandas as pd
3. import numpy as np
4. from sklearn.model\_selection import GridSearchCV, KFold
5. from keras.models import Sequential
6. from keras.layers import Dense, Dropout
7. from keras.wrappers.scikit\_learn import KerasClassifier
8. from keras.optimizers import Adam

**Explanation:**

In the above snippet of code, we have imported the required packages and libraries necessary for the project. One can also save the program file and execute it in order to check if the libraries and packages are installed and imported properly.

Once the packages are imported successfully, we have to use the following snippet of code in order to import the dataset and print the first five rows of it.

**File: mygrid.py**

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

**Output:**

Person gender Age Height pre.weight Diet weight6weeks

0 25 41 171 60 2 60.0

1 26 32 174 103 2 103.0

2 1 0 22 159 58 1 54.2

3 2 0 46 192 60 1 54.0

4 3 0 55 170 64 1 63.3

**Explanation:**

In the above snippet of code, we have imported the dataset using the **read\_csv()** of the **pandas** library and stored it within the **mydf** variable. We have then printed the first five rows using the **head()** function along with the **mydf** variable.

Now, let us divide the data into the feature and label sets in order to apply the standard scaling on the dataset.

The snippet of code for the same is shown below:

**File: mygrid.py**

1. # converting dataframe into numpy array
2. mydataset = mydf.values
4. X = mydataset[:, 0:6]
5. Y = mydataset[:, 6].astype(int)
7. # Normalizing the data using sklearn StandardScaler
8. from sklearn.preprocessing import StandardScaler
10. myscaler = StandardScaler().fit(X)
12. # Transforming and displaying the training data
13. X\_stdized = myscaler.transform(X)
15. mydata = pd.DataFrame(X\_stdized)

**Explanation:**

In the above snippet of code, we have converted the **pandas** dataframe into a **NumPy** array. We have then imported the **StandardScaler** module from the **sklearn** library and use the function to normalize the data. We have then transformed and displayed the training data using the **transform()** function.

Now, let us consider the following snippet of code in order to create a simple deep learning model.

**File: mygrid.py**

1. # defining the function to create model
2. def create\_my\_model(learnRate, dropoutRate):
3. # Creating model
4. mymodel = Sequential()
5. mymodel.add(Dense(6, input\_dim = 6, kernel\_initializer = 'normal', activation = 'relu'))
6. mymodel.add(Dropout(dropoutRate))
7. mymodel.add(Dense(3, input\_dim = 6, kernel\_initializer = 'normal', activation = 'relu'))
8. mymodel.add(Dropout(dropoutRate))
9. mymodel.add(Dense(1, activation = 'sigmoid'))
11. # Compiling the model
12. my\_Adam = Adam(learning\_rate = learnRate)
13. mymodel.compile(loss = 'binary\_crossentropy', optimizer = my\_Adam, metrics = ['accuracy'])
14. return mymodel

**Explanation:**

The following snippet of code has defined a function as **create\_my\_model()** accepting two parameters, i.e., **learnRate** and **dropoutRate**, respectively. We have then created the model as **mymodel** using the **Sequential()** function. We have also used the **add()** along with the **Dense()** and **Dropout()** function. We have then compiled the model using the **compile()** function.

As a result, when we execute the code, this will lead to loading the dataset, preprocessing it, and creating a machine learning model. Since we are only interested in understanding the working of Grid Search, we haven't performed the train/test split, and we had fitted the model on the complete dataset.

Now, we will understand how Grid Search makes the programmer's life easier by optimizing the parameters in the next section.

### Training the Model without Grid Search

In the snippet of code shown below, we will create a model with the help of parameter values that we decided on randomly or based on our intuition and see how our model performs:

**File: mygrid.py**

1. # Declaring the values of the parameter
2. dropoutRate = 0.1
3. epochs = 1
4. batchSize = 20
5. learnRate = 0.001
7. # Creating the model object by calling the create\_my\_model function we created above
8. mymodel = create\_my\_model(learnRate, dropoutRate)
10. # Fitting the model onto the training data
11. mymodel.fit(X\_stdized, Y, batch\_size = batchSize, epochs = epochs, verbose = 1)

**Output:**

4/4 [==============================] - 41s 14ms/step - loss: 0.9364 - accuracy: 0.0000e+00

**Explanation:**

In the above snippet of code, we have declared the values of the parameter, i.e., **dropoutRate, epochs, batchSize**, and **learnRate**, respectively. We have then created the model object by calling the **create\_my\_model()** function. We have then fitted the model onto the training data.

As a result, the accuracy we got is 0.0000e+00.

### Optimizing Hyper-parameters using Grid Search

If we are not using the Grid Search method, we can directly call the **fit()** function on the model we have created above. But in order to utilize the Grid Search method, we have to pass in few arguments to the **create\_my\_model()** function. Moreover, we have to declare the grid with various options to try for every parameter. Let us perform that in different parts.

First of all, we will try modifying the **create\_my\_model()** function in order to accept arguments from the calling function as shown below:

**File: mygrid.py**

1. def create\_my\_model(learnRate, dropoutRate):
2. # Creating the model
3. mymodel = Sequential()
4. mymodel.add(Dense(6, input\_dim = 6, kernel\_initializer = 'normal', activation = 'relu'))
5. mymodel.add(Dropout(dropoutRate))
6. mymodel.add(Dense(3, input\_dim = 6, kernel\_initializer = 'normal', activation = 'relu'))
7. mymodel.add(Dropout(dropoutRate))
8. mymodel.add(Dense(1, activation = 'sigmoid'))
10. # Compile the model
11. myadam = Adam(learning\_rate = learnRate)
12. mymodel.compile(loss = 'binary\_crossentropy', optimizer = myadam, metrics = ['accuracy'])
13. return mymodel
15. # Creating the model object
16. mymodel = KerasClassifier(build\_fn = create\_my\_model, verbose = 1)

**Explanation:**

In the above snippet of code, we have made some changes to the previous **create\_my\_model** function and used the **KerasClassifier** to create the model object.

Now, let us implement the algorithm for Grid Search and fit the dataset on it:

**File: mygrid.py**

1. # Defining the arguments that we want to use in Grid Search along
2. # with the list of values that we want to try out
3. learnRate = [0.001, 0.02, 0.2]
4. dropoutRate = [0.0, 0.2, 0.4]
5. batchSize = [10, 20, 30]
6. epochs = [1, 5, 10]
8. # Making a dictionary of the grid search parameters
9. paramgrid = dict(learnRate = learnRate, dropoutRate = dropoutRate, batch\_size = batchSize, epochs = epochs )
11. # Building and fitting the GridSearchCV
12. mygrid = GridSearchCV(estimator = mymodel, param\_grid = paramgrid, cv = KFold(random\_state = None), verbose = 10)
14. gridresults = mygrid.fit(X\_stdized, Y)
16. # Summarizing the results in a readable format
17. print("Best: " + gridresults.best\_score\_ + " using " + gridresults.best\_params\_)
19. means = gridresults.cv\_results\_['mean\_test\_score']
20. stds = gridresults.cv\_results\_['std\_test\_score']
21. params = gridresults.cv\_results\_['params']
23. for mean, stdev, param in zip(means, stds, params):
24. print(mean + "(" + stdev + ")" + " with: " + param)

**Output:**

Best: 0.00347268912077, using {batch\_size=10, dropoutRate=0.4, epochs=5, learnRate=0.2}

**Explanation:**

The above output shows the parameter combination which yields the best accuracy.

At last, we can conclude that the Grid Search is quite easy to implement in the Python programming language and saved us a lot of time in human labor. We can list down all the arguments we wanted to tune, declare the values that need to be tested, execute the code, and forget about it. The process is so easy and convenient that it requires less input from the programmer's side. Once the best argument combination has been found, we can utilize that for the final model.

# Python High Order Function

As we must be aware of the basic concept of Python functions, we should move forward with some advanced concepts related to Python functions. In this tutorial, we are going to discuss the important aspects of High order functions in Python, like what high order functions are, how we can define them in Python, how we can use them in Python and what the properties of high order functions are.

### Prerequisites:

To learn about high order functions in Python, we must have basic knowledge of the following concepts:

* [Python functions](https://www.javatpoint.com/python-functions)
* Parameters
* Python objects
* [Python decorators](https://www.javatpoint.com/python-decorator)

First, let's start with the first thing, i.e., High order functions, and understand a basic about them.

## High order functions

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.

A fact about the High order function is that a high order function is applicable to both functions as well as to the methods that take a function as their parameter or return a function as the result of them. In [Python](https://www.javatpoint.com/python-tutorial), this concept of high-order functions is supported with every aspect.

### Properties of High order functions in Python

Now, in this section, we will discuss some of the important properties of high order functions that are applicable in Python as well.

* 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.
* In high order 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.

## High order functions in Python

Now, in this section, we will talk specifically about the Python high order functions and how we can define them. We will discuss the methods and means by which we will define and use high order functions in our Python program.

Following are the ways to define High order functions in a Python code that we are going to discuss in this tutorial.

1. Using functions as objects in High order function
2. Returning function as a result in high order function
3. Functions as a parameter for another function
4. Decorators as high order function

Now, we will discuss each of the above-given methods in detail and learn about their implementation as high order functions in a Python program.

### 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.

Look at the following example program to learn the implementation of method we discussed above:

**Example -**

1. # a default function to take another function parameter
2. def spell(text):
3. # Making text in upper
4. return text.upper()
5. # Taking text as user input
6. text = input("Enter a text to print it in uppercase and double: ")
7. # Spell function with text
8. print(spell(text))
9. # Assigning variable with the default function
10. scream = spell
11. # Scream with text variable
12. print(scream(text))

**Output:**

Enter a text to print it in uppercase and double: JavaTPoint

JAVATPOINT

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.

Look at the following program to understand the implementation of above-given method:

**Example -**

1. # Default function for making text uppercase
2. def scream(word):
3. return word.upper()
4. # Default function for making text lowercase
5. def spell(word):
6. return word.lower()
7. # A third function that work as a high order function
8. def speak(funct):
9. # Storing the function in a variable in high order function
10. speaking = funct("Hello Python Developers! You are welcomed to JavaTpoint")
11. print(speaking)
12. # Printing text in uppercase
13. speak(scream)
14. # Printing text in lowercase
15. speak(spell)

**Output:**

HELLO PYTHON DEVELOPERS! YOU ARE WELCOMED TO JAVATPOINT

hello python developers! you are welcomed to javatpoint

### 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.

Look at the following example program to learn the implementation of method we discussed above:

**Example -**

1. # A default function for addition
2. def Adding(a):
3. # Nested function with second number
4. def Addition(b):
5. return a + b # addition of two numbers
6. return Addition # Result
8. # Taking both number variable as user input
9. a = int(input("Enter First Number: "))
10. b = int(input("Enter Second Number: "))
11. # Assigning nested adding function to a variable
12. AddVariable = Adding(a)
13. # Using variable as high order function
14. Result = AddVariable(b)
15. # Printing result
16. print("Sum of Two numbers given by you is: ", Result)

**Output:**

Enter First Number: 24

Enter Second Number: 26

Sum of Two numbers given by you is: 50

### Method 4: Decorators as high order function

We can use decorators as the high order function as the most commonly used high order function in Python. 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. Look at the following exemplar syntax for a decorator defined in a Python program.

**Syntax**

1. # Using a decorator
2. @JTP\_Decorator
3. def Python\_Decorator():
4. .
5. .

The above syntax for the decorator is equivalent to the following Python code for a high order function.

1. # Using Python default function as Python decorators
2. def Python\_Decorator():
3. .
4. .
5. 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.

Look at the following program to understand the implementation of above given method:

**Example -**

1. # Using default function as Python decorators
2. def Python\_Decorator(funct):
3. # Inner nested function
4. def inner():
5. print("This line of code will be printed before the execution of high order function")
6. funct()
7. print("This line of code will be printed after the execution of high order function")
8. return inner
9. # A default function as decorator
10. def JTP\_Decorator():
11. print("This line of code will be printed inside the execution of high order function")
12. JTP\_Decorator = Python\_Decorator(JTP\_Decorator) # Python decorator as high order function
13. # Python decorator calling out as high order function
14. JTP\_Decorator()

**Output:**

This line of code will be printed before the execution of high order function

This line of code will be printed inside the execution of high order function

This line of code will be printed after the execution of high order function

# nsetools in Python

In the following tutorial, we will discuss the **nsetools** library in the Python programming language. We will understand its features and work with some examples.

So, let's get started.

## Understanding the nsetools library

NSE or National Stock Exchange of India Limited is the leading stock exchange of India, situated in Mumbai, Maharashtra. NSE was established in the year 1992 as the first dematerialized electronic exchange in the country.

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. The data accuracy is only as correct as provided on the official website of the National Stock Exchange of India Limited. (<http://www.nseindia.com>)

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

## Main features of the Python nsetools library

Some of the key features of the Python **nsetools** library are stated as follows:

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 and indices traded on the National Stock Exchange.
4. Moreover, it also provides a set of:
   1. Top losers
   2. Top gainers
   3. Most active
5. It also delivers several helpful Application Programming Interfaces (APIs) in order to validate a stock code and index code.
6. The library optionally returns data in JSON format.
7. It has a hundred per cent Unit test coverage.

## How to install the Python nsetools library?

The installation part of the **nsetools** library is quite easy, and it has no external dependencies. All the dependencies of the library are part of standard distribution packages of Python. We can install the **nsetools** library using the pip installer as shown in the following syntax:

**Syntax:**

1. $ pip install nsetools

### Updating the library

If some of us already have installed the **nsetools** library in their systems, then the following command will allow them to update the library.

**Syntax:**

1. $ pip install nsetools -upgrade

### Python 3 support

Python 3 support for the library has been included from version 1.0.0 and so on. Now, this library is able to work for both Python 2 as well as Python 3.

## Creating an NSE object

We can create an NSE object using the **Nse()** function offered by the **nsetools** library. The same can be seen in the following example:

**Example:**

1. # importing the Nse() function from the nsetools library
2. from nsetools import Nse
4. # creating an NSE object
5. nse\_obj = Nse()
7. # printing the value of the object
8. print("NSE Object:", nse\_obj)

**Output:**

NSE Object: Driver Class for National Stock Exchange (NSE)

**Explanation:**

In the above snippet of code, we have imported the required function from the library. We have then defined a variable that uses the **Nse()** function to create an NSE object. We have then printed the value of the variable for the users.

## Getting Information using the nsetools library

Let us consider an example demonstrating the use of **nsetools** for gathering Information.

**Example:**

1. # importing the Nse() function from the nsetools library
2. from nsetools import Nse
4. # creating an NSE object
5. nse\_obj = Nse()
7. # getting quotation of the company
8. the\_quotation = nse\_obj.get\_quote('sbin')
10. # printing the name of the company
11. print(the\_quotation["companyName"])
13. # printing average price
14. print("Average Price: " + str(the\_quotation["averagePrice"]))

**Output:**

State Bank of India

Average Price: 431.97

**Explanation:**

In the above snippet of code, we have imported the required module and created an NSE object using the **Nse()** function. We have then defined another variable that uses the **get\_quote()** function on the NSE object to get the quotation of the specified company. We have then printed the required details for the users.

**Python program to find the nth Fibonacci Number**

In the following tutorial, we will understand how to find the nth Fibonacci Number using Python. We can define a Fibonacci Number, where the following number is the sum of the preceding two numbers.

The first two elements of the Fibonacci series are 0 and 1, respectively. We can calculate the third element of the series by adding the preceding two elements and will get the third term as 0 + 1, which is equal to 1. Similarly, the fourth term will be the sum of the second and third terms, which is 1 + 1 = 2 and so on. The series of such numbers is known as a Fibonacci Series.

The recurrence relation defines a Fibonacci number as shown below:

**Fn = Fn - 1 + Fn - 2**

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

There are different ways to find the nth Fibonacci Number using the Python programming language. Some of them are as follows:

1. Finding nth Fibonacci Number using Recursion
2. Finding nth Fibonacci Number using dynamic programming
3. Finding nth Fibonacci Number using dynamic programming and space optimization
4. Finding nth Fibonacci Number using arrays

Of these ways, the two most fundamental are the Recursion method and the Dynamic method.

Let us understand the working of these methods in detail with examples.

**Finding nth Fibonacci Number using Recursion**

The term recursion is used to define something within itself. In a programming language like Python, Recursion refers to the process of a function calling itself. With the proper and correct code, the Recursion will create a finite loop.

Let us consider the following snippet of code for better understanding.

**Example:**

1. # defining the function for Fibonacci Series
2. def Fibonacci\_Series(n):
3. # using if-else conditional statement
4. if n < 0:
5. print("Oops! Incorrect input")
6. # First Fibonacci number is 0
7. elif n == 0:
8. return (0)
9. # Second Fibonacci number is 1
10. elif n == 1:
11. return (1)
12. else:
13. return (Fibonacci\_Series(n - 1) + Fibonacci\_Series(n - 2))
14. # printing the 12th element of the Fibonacci Series
15. print("12th Element of the Fibonacci Series:", Fibonacci\_Series(12))

**Output:**

12th Element of the Fibonacci Series: 144

**Explanation:**

In the above snippet of code, we have defined a function as **Fibonacci\_Series()** that accepts a parameter as **n**.

Moreover, we are aware that the first two elements of the Fibonacci are **0** and **1**. In the event of the input as **n = 1** or **n = 2** (First or Second terms of Fibonacci series), we have used the **if-else** conditional statement to return **0** or **1**. In case the value of **n** is greater than **2**, the function will call itself with a lower input value. As we can observe that the code returns **(Fibonacci\_Series(n - 1) + Fibonacci\_Series(n - 2))**. Here, the function calls itself with a lower value unless it reaches the base value of **n = 1** and **n = 2**, and as we know from before, **n = 1** returns **0** and **n = 2** returns **1**. The returned values are then continuously added to produce the sequence of the Fibonacci Series.

**Finding the nth Fibonacci Number using Dynamic Programming**

Dynamic Programming utilizes Recursion as well; however, it mainly utilizes **if-els**e conditional statements. Within the statements, the value of the Fibonacci number is stored in a variable. With the help of Recursion, the repeated addition allows us to obtain this Fibonacci number.

Let us consider the following example to understand the same.

**Example:**

1. # defining the function to find the nth Fibonacci Number
2. def Fibonacci\_series(x):
3. # Taking First two terms of the Fibonacci Series as 0 and 1
4. fib\_Array = [0, 1]
5. # Here, as we know that the first two terms of Fibonacci Series are 0 and 1,
6. # we append the remaining values (Fibonacci numbers from index 2 to x)
7. # in the array using recursion and return the last element.
8. # In the range function, we take range(2, x + 1) instead of range(2, x).
9. # This is because range function in python iterates until the value
10. # before the upper limit. So, if we take from 2 to x, it would only
11. # iterate from second to (x - 1)th element.
12. for n in range(2, x + 1):
13. fib\_Array.append(fib\_Array[n - 1] + fib\_Array[n - 2])
14. return fib\_Array[x]
15. print("12th Term of Fibonacci Series:", Fibonacci\_series(12))

**Output:**

12th Term of Fibonacci Series: 144

**Explanation:**

In the above snippet of code, we defined the function as **Fibonacci\_series()**, which accepts the parameter as variable **x**. We created a one-dimensional array as **fib\_Array** with data elements **0** and **1** in its zeroth and first indices. Then, if the provided input ('**x**') is less than or equal to **2**, which is also the length of the array **fib\_Array**, it returns **0** as the first number for **x = 1** and **1** as the second number for **x = 2**. If the value of **x** is greater than **2**, we have used recursion to call and insert the preceding two data elements. However, rather than returning the nth Fibonacci number directly, we append each of the summated elements to the **fib\_Array** array. At last, we have returned the last element of the array (i.e., the nth element) and printed the value for the users.

**Finding the nth Fibonacci Number using Dynamic Programming and Space Optimization**

This method is almost completely identical to Dynamic Programming. However, dynamic programming utilizes recursion to accomplish recurring addition, whereas this method utilizes the for-loop.

Let us consider the following example to understand the same.

**Example:**

1. # defing the function to return the nth element of the Fibonacci Series
2. def Fibonacci\_series(x):
3. # assiging the variables
4. m = 0
5. n = 1
6. # using the if-elif-else conditional statements
7. if x < 0:
8. print("Wrong input")
9. elif x == 0:
10. return m
11. elif x == 1:
12. return n
13. else:
14. # using the for-loop
15. for i in range(2, x + 1):
16. o = m + n
17. m = n
18. n = o
19. return n
20. # printing the twelveth term of the Fibonacci Series
21. print("12th element of the Fibonacci Series:", Fibonacci\_series(12))

**Output:**

12th element of the Fibonacci Series: 144

**Explanation:**

In the above snippet of code, we have defined a function and assigned two variables, **m = 0** and **n = 1**. These elements are the first and second elements of the Fibonacci Series. We have then used the **if-elif-else** conditional statements where the program returns **0** for input value **x = 1** and **1** for input value **x = 2**. If the value of **x** is greater than **2**, we have used the **for-loop** of **i** in the range **(2, x + 1)**. We have taken a variable o to store the sum of the preceding two elements in the series. Once **o** takes the value of **m + n**, the value of **m** is reassigned to **n**. Subsequently, the value of **n** is reassigned to the value of **o**. This process continues, and value 3 keeps reassigning until the loop terminates. Once the loop is terminated, the function returns the value of **n**, which stores the value of the nth Fibonacci Number.

**Finding the nth Fibonacci Number using Array**

In this method, we create an array of size **x** by repeated addition using the **for-loop**. Hence, the nth Fibonacci Number is returned.

Let us consider the following example to understand the same.

**Example:**

1. # defining the function
2. def Fibonacci\_series(x):
3. # creating an array in the function
4. fib\_Array = [0] \* (x + 1)
5. fib\_Array[1] = 1
6. # adding elements of the series to the array using addition of previous two elements.
7. for n in range (2, x + 1):
8. fib\_Array[n] = fib\_Array[n - 1] + fib\_Array[n - 2]
9. return fib\_Array[x]
10. if \_\_name\_\_ == "\_\_main\_\_":
11. print("12th element of the Fibonacci series:", Fibonacci\_series(12))

**Output:**

12th element of the Fibonacci series: 144

**Explanation:**

In the above snippet of code, we have defined the function. Within the function, we have created an array to find the nth element of the Fibonacci Series. We have then used the **for-loop** to add elements of the series to the array by repeating the addition of the preceding two elements. At last, the nth element is returned and printed for the users.

# 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. We will only focus to object detection from images using OpenCV in this tutorial. We will learn about how we can use OpenCV to do object detection from a given image using a Python program.

## 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

We have learned about object detection in the previous section, and in this section, we will learn that how we can do object detection in an image or video using the OpenCV library. We will first import the OpenCV library in the Python program, and then we will use functions to perform object detection on an image file given to us. But, before using and importing the library functions, let's first install the requirements for using the Object detection technique.

In this tutorial, we will use the Haar cascade technique to do object detection. Let's learn in brief about the Haar cascade technique first.

### 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.

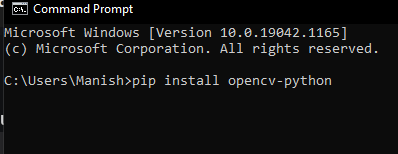
### Requirements for object detection with Python OpenCV:

We have to install first some important libraries in our system as it is an important requirement for doing object detection tasks. We have to install the following libraries into our system as the requirement for performing object detection:

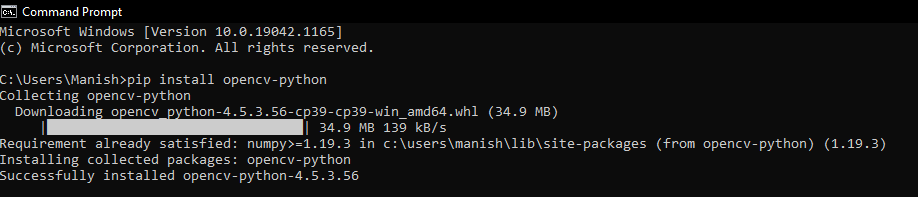
### 1. Installation of OpenCV library:

First and foremost, the requirement to perform object detection using the OpenCV library is that the OpenCV library should be present in our device so that we can import it into a Python program and use its object detection functions. If this library is not present in our system, we can use the following command in our command prompt terminal to install it:

1. pip install opencv-python



When we press the enter key after writing this command in the terminal, the pip installer in the command prompt will start installing the OpenCV library into our system.

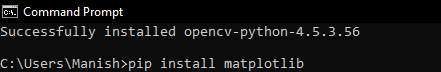


As we can see that, the OpenCV library is successfully installed in our system, and now we can import it into a Python program to use its functions.

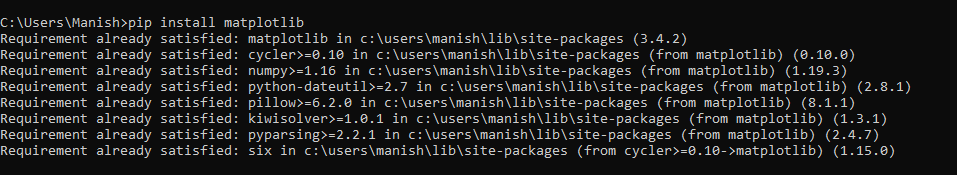
### 2. Installation of matplotlib library:

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. If the matplotlib library is not present in our system, we have to use the following command in our command prompt terminal to install it:

1. pip install matplotlib



When we press the enter key after writing this command in the terminal, the pip installer in the command prompt will start installing it into our system.



As we can see that, the matplotlib library is successfully installed in our system, and now we can import it into a Python program to use its functions for opening, reading etc., images.

We have installed all the required libraries for performing object detection, and now we can move ahead with the implementation part of this task.

### Implementation of Object detection in Python:

In this part, we will write the Python programs to do the object detection and understand the implementation of it. We will use the following image in our Python program to perform the object detection on it:



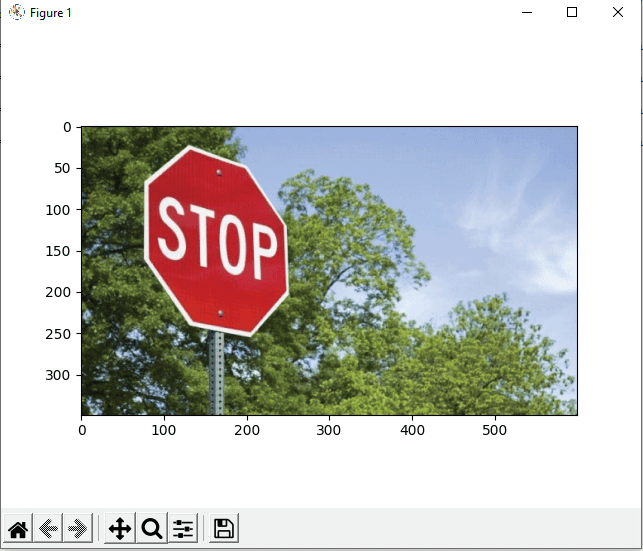
### Opening the Image

We will first open the image given above and create the environment of the picture to show it in the output. Let's first look at an example program to understand the implementation, and then we will look at the explanation part.

**Example 1:** Opening the image using OpenCV and matplotlib library in a Python program:

1. # Import OpenCV module
2. import cv2
3. # Import pyplot from matplotlib as pltd
4. from matplotlib import pyplot as pltd
5. # Opening the image from files
6. imaging = cv2.imread("opencv-od.png")
7. # Altering properties of image with cv2
8. img\_gray = cv2.cvtColor(imaging, cv2.COLOR\_BGR2GRAY)
9. imaging\_rgb = cv2.cvtColor(imaging, cv2.COLOR\_BGR2RGB)
10. # Plotting image with subplot() from plt
11. pltd.subplot(1, 1, 1)
12. # Displaying image in the output
13. pltd.imshow(imaging\_rgb)
14. pltd.show()

**Output:**



**Explanation:**

First, we have imported the OpenCV (as cv2) and matplotlib (as plt) libraries into the program to use their functions in the code. After that, 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 the detectMultiScale() in the program to detect the object present in the image. Following is the syntax for using detectMultiScale() function in the code:

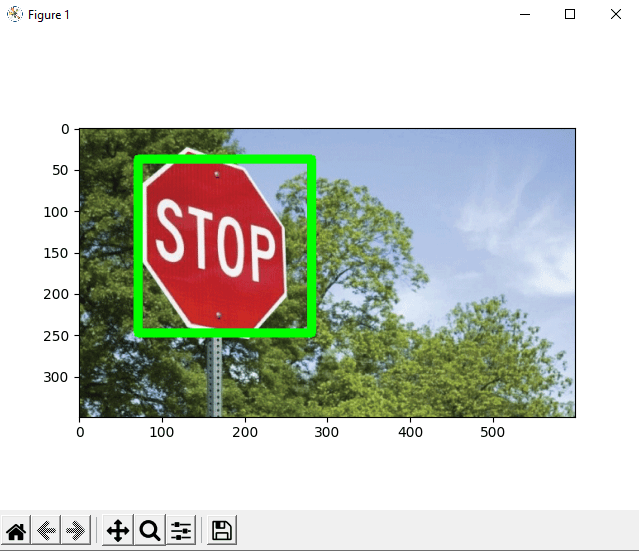
1. found = xml\_data.detectMultiScale(img\_gray,
2. minSize = (30, 30))

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. Let's understand the implementation of object detection in the image through an example program.

**Example 2:** Object detection in the image using the detectMultiScale() in the following Python program:

1. # Import OpenCV module
2. import cv2
3. # Import pyplot from matplotlib as plt
4. from matplotlib import pyplot as pltd
5. # Opening the image from files
6. imaging = cv2.imread("opencv-od.png")
7. # Altering properties of image with cv2
8. imaging\_gray = cv2.cvtColor(imaging, cv2.COLOR\_BGR2GRAY)
9. imaging\_rgb = cv2.cvtColor(imaging, cv2.COLOR\_BGR2RGB)
10. # Importing Haar cascade classifier xml data
11. xml\_data = cv2.CascadeClassifier('XML-data.xml')
12. # Detecting object in the image with Haar cascade classifier
13. detecting = xml\_data.detectMultiScale(imaging\_gray,
14. minSize = (30, 30))
15. # Amount of object detected
16. amountDetecting = len(detecting)
17. # Using if condition to highlight the object detected
18. if amountDetecting != 0:
19. for (a, b, width, height) in detecting:
20. cv2.rectangle(imaging\_rgb, (a, b), # Highlighting detected object with rectangle
21. (a + height, b + width),
22. (0, 275, 0), 9)
23. # Plotting image with subplot() from plt
24. pltd.subplot(1, 1, 1)
25. # Displaying image in the output
26. pltd.imshow(imaging\_rgb)
27. pltd.show()

**Output:**



**Explanation:**

After opening the image in the program, we have imported the cascade classifier XML file into the program. Then, we used the **detectMultiScale()** function with the imported cascade file to detect the object present in the image or not.

We used if condition in the program to check that object is detected or not, and if the object is detected, we have highlighted the detected object part using for loop with cv2 functions. After highlighting the detected object part in the image, we have displayed the processed image using the plt **show()** and **imshow()** function.

As we can see in the output, the image with the object detected part as highlighted is shown to us when we run the program.

# Python SimpleImputer module

In this tutorial, we are going to learn about the SimpleImputer module of the Sklearn library, and it was previously known as impute module but updated in the latest versions of the Sklearn library. We will discuss the SimpleImputer class and how we can use it to handle missing data in a dataset and replace the missing values inside the dataset using a Python program.

## SimpleImputer class

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.

### Syntax for SimpleImputer() method:

To implement the SimpleImputer() class method into a Python program, we have to use the following syntax:

1. 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.

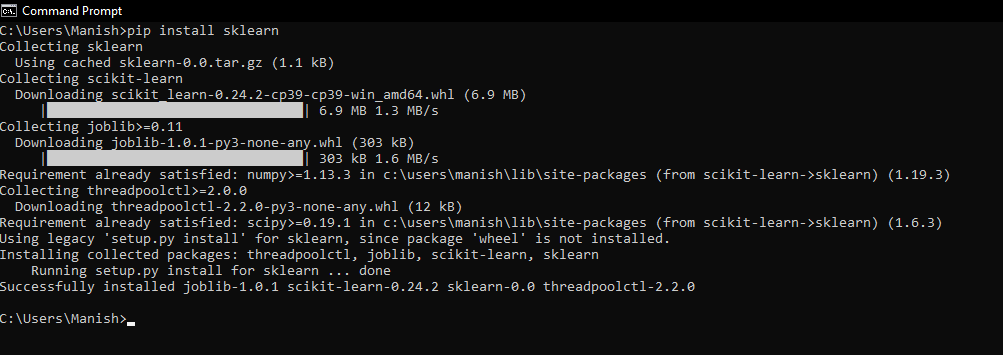
SimpleImputer class is the module class of Sklearn library, and to use this class, first we have to install the Sklearn library in our system if it is not present already.

### Installation of Sklearn library:

We can install the Sklearn by using the following command inside the command terminal prompt of our system:

1. pip install sklearn

After pressing the enter key, the sklearn module will start installing in our device, as we can see below:



Now, the Sklearn module is installed in our system, and we can move ahead with the SimpleImputer class function.

### 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. Let's understand the implementation of this through an example Python program.

**Example 1:** Look at the following Python program with a dataset having NaN values defined in it:

1. # Import numpy module as nmp
2. import numpy as nmp
3. # Importing SimpleImputer class from sklearn impute module
4. from sklearn.impute import SimpleImputer
5. # Setting up imputer function variable
6. imputerFunc = SimpleImputer(missing\_values = nmp.nan, strategy ='mean')
7. # Defining a dataset
8. 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]]
9. # Print original dataset
10. print("The Original Dataset we defined in the program: \n", dataSet)
11. # Imputing dataset by replacing missing values
12. imputerFunc = imputerFunc.fit(dataSet)
13. dataSet2 = imputerFunc.transform(dataSet)
14. # Printing imputed dataset
15. print("The imputed dataset after replacing missing values from it: \n", dataSet2)

**Output:**

The Original Dataset we defined in the program:

[[32, nan, 34, 47], [17, nan, 71, 53], [19, 29, nan, 79], [nan, 31, 23, 37], [19, nan, 79, 53]]

The imputed dataset after replacing missing values from it:

[[32. 30. 34. 47. ]

[17. 30. 71. 53. ]

[19. 29. 51.75 79. ]

[21.75 31. 23. 37. ]

[19. 30. 79. 53. ]]

**Explanation:**

We have firstly imported the numpy module (to define a dataset) and sklearn module (to use the SimpleImputer class method) into the program. Then, we defined the imputer to handle the missing values using the SimpleImputer class method, and we used the 'mean' strategy to replace the missing values from the dataset. After that, we have defined a dataset in the program using the numpy module function and gave some missing values (NaN values) in the dataset. Then, we printed the original dataset in the output. After that, we have imputed and replaced the missing values from the dataset with the imputer that we have defined earlier in the program with SimpleImputer class. After imputing the dataset and replacing the missing values from it, we have printed the new dataset as a result.

As we can see in the output, the imputed value dataset having mean values in the place of missing values, and that's how we can use the SimpleImputer module class to handle NaN values from a dataset.

## Conclusion

We have read about the SimpleImputer class method in this method, and we learned how we could use it to handle the NaN values present in a dataset. We learned about the strategy value parameter, which we use to define the method for replacing the NaN values of the dataset. We have also learned about the installation of the Sklearn library, and then last, we used the SimpleImputer class method in an example to impute the dataset.

**Second Largest Number in Python**

When we have a lot of elements in our list, the thought of finding the highest or lowest element can come to our mind and Python has made it much easier for us.

In this article, we shall how we can use to find the second largest number in Python from a list.

1. Sorting the list and then print the second last number.
2. Removing the maximum element.
3. Finding the maximum element.
4. Traversing the list.

Let us have a look at the first approach-

**Sorting the list and then print the second last number**

The following program illustrates how we can do it in Python-

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Example -**

1. #program to find the second largest number of list
2. # declaring the list
3. list\_val = [20, 30, 40, 25, 10]
4. # sorting the list
5. list\_val.sort()
6. #displaying the second last element of the list
7. print("The second largest element of the list is:", list\_val[-2])

**Output:**

The second largest element of the list is: 30

It's time to go for the explanation part-

1. We have declared the list from which we want to take out the second last element.
2. After this, we used the sort method so that all the elements of our list are arranged in ascending order.
3. Now we make use of negative indexing since the second-largest number will come at the second last position.

The second method is to obtain the second largest element of the list by removing the maximum element.

Let us see how we can do it.

**Removing the maximum element**

**Example -**

1. #program to find the second largest number of list
3. # declaring the list
4. list\_val = [20, 30, 40, 25, 10]
6. # new\_list is a set of list1
7. res\_list = set(list\_val)
9. #removing the maximum element
10. res\_list.remove(max(res\_list))
12. #printing the second largest element
13. print(max(res\_list))

**Output:**

30

**Explanation -**

Let us understand what we have done in the above program-

1. We have declared the list from which we want to take out the second last element.
2. After this, we used the set method to take all the unique elements of the list.
3. Now we make use of max() to get the maximum value from the list and then remove it.
4. After this, we print the maximum of the resultant list which will give us the second-largest number.

In the third method, we will use for loop and find the second largest number from the list.

**Example -**

1. # declaring empty list
2. list\_val = []
4. # user provides the number of elements to be added in the list
5. num\_list = int(input("Enter number of elements in list: "))

8. for i in range(1, num\_list + 1):
9. element = int(input("Enter the elements: "))
10. list\_val.append(element)

13. # sort the list
14. list\_val.sort()
16. # print second largest element
17. print("Second largest element is:", list\_val[-2])

**Output:**

Enter number of elements in list: 5

Enter the elements: 10

Enter the elements: 20

Enter the elements: 30

Enter the elements: 40

Enter the elements: 50

The second largest element is: 40

**Explanation -**

Let us have a glance at what we have done here-

1. We have declared an empty list in which we will insert the elements.
2. After this, we ask the user to provide us the number of elements we would like to add to our list.
3. After this, we use the sort method so that all the elements of our list are arranged in ascending order.
4. Now we make use of negative indexing since the second-largest number will come at the second last position.

**Traversing the list**

In the last program, we will traverse the list to find out the largest number and then make use of conditional statements to find the second largest number from the list.

The following program illustrates the same-

**Example -**

1. def calc\_largest(arr):
2. second\_largest = arr[0]
3. largest\_val = arr[0]
4. for i in range(len(arr)):
5. if arr[i] > largest\_val:
6. largest\_val = arr[i]
8. for i in range(len(arr)):
9. if arr[i] > second\_largest and arr[i] != largest\_val:
10. second\_largest = arr[i]
12. return second\_largest
13. print(calc\_largest([20, 30, 40, 25, 10]))

**Output:**

30

**Explanation -**

Let us understand what we have done in the above program-

1. The first step is to create a function that checks the largest number from the list by traversing it.
2. In the next for loop, we traverse the list again for finding the highest number but this time excludes the previous one since here our objective is to find the second largest function.
3. Finally, we pass our list in the function.

So, in this article, we got the chance to think out of the box and discover some new ways to develop the logic for finding the second largest number in Python.