# Basic Commands in Python | Magic Commands

When Python was introduced in 1991, it was general assumption that Python is "use at own risk" language. But situation has changed; Python is a dominant language at present time, it is used for data science, machine learning, and software development.

As we know that Python is a flexible language. So we can add new features and functions such as magic command.

This tutorial will discuss about the magic commands. These magic commands surely help to reduce the lot of headache.

## Introduction

Magic commands are easy designed syntax that facilitates us to perform routine task. These are basically created to accomplish some common task in data analysis using Python. In fact, they control the nature of IPython itself. It is used to running an external script or calculating the execution time of a piece of code.

We can use the two different forms of magic commands in IPython.

* **Line Magic -**The line magic commands represent using % prefix and operates on a particular line of input. It is used in the form of expression and their return value can be assigned to variable.
* **Cell Magic -**The cell magic commands represent using %% prefix and works on a complete cell or multiple lines of input. They receive the whole block as a string.

Let's learn about the most popular and interesting commands.

## Built -in Magic commands

### 1.%autocall [mode]

The **%autocall[mode]**magic function is used to make a function automatically callable without having to use parentheses.

### 2. %automagic

The magic function can be also callable without having to initial % if we set to 1. We need to set to 0 to deactivate it.

**Output:**

1. Automagic is OFF, % prefix IS needed **for** line magics.

### 3. %run

Suppose we have a file named **program\_test.py**with the following code.

1. def name()
2. print("Hello I am Stuart?)
3. name()

We use the following statement to run.

1. %run program\_test.py

The script will run without importing defined variables.

The above command will behave same as the python **program\_script.py**file.

#### Note - We can offer access to already defined variables using %run -i.

Now we can access all the variables of executed file in IPython shell.

### 4. %cd

This magic command changes the current directory. It automatically manages the internal list of directories we visit during IPython session.

**Output:**

1. C:\Users\DEVANSH SHARMA

**Usage -**

* %cd <dir> - Changes current working directory to <dir>
* %cd.. - It changes current directory to parent directory.
* %cd - It changes to last visited directory.

### 5. %dhist

It is a very useful magic command; it prints all the directories we have visited in current session. Every time %cd command is executed, this is updated in **\_dh**variable.

**Output:**

1. Directory history (kept in \_dh)
2. 0: C:\Users\DEVANSH SHARMA

### 6. %edit

The edit magic command unlocks the default text editor of current operating system **(Notepad for Window)**for editing a Python script. We can edit the current Python script in editor.

**Output:**

1. IPython will make a temporary file named: C:\Users\DEVANS~1\AppData\Local\Temp\ipython\_edit\_z41uo6o8\ipython\_edit\_zwb99en9.py

### 7. %env[GUINAME]

This magic command is used to enable and disables **IPython GUI**event loop integration. When we use the GUINAME argument, this command replaces the default GUI toolkits by the specified one.

### 8. %%timeit

The above command calculates the time taken by the **IPython**environment to execute a Python program. Let's understand the following program.

1. %%timeit
2. square\_numbers = [num\*\*2 **for** num in range(1,1000)]

**Output:**

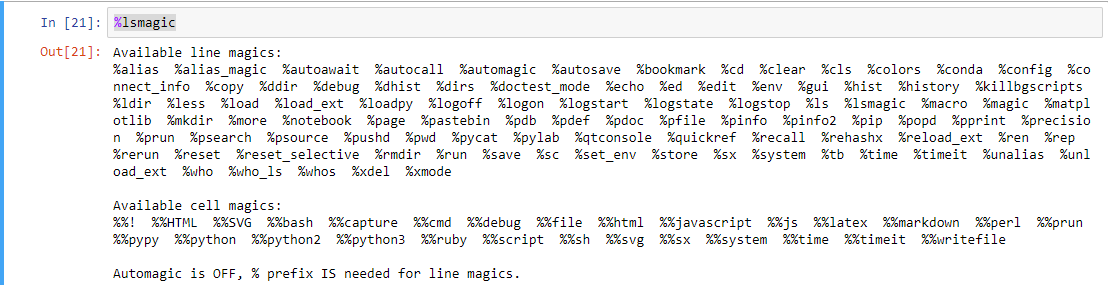
659 µs ± 19.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

### 9. %lsmagic

This command will return the list of all magic command. Let's see the following example.

1. %lsmagic

**Output:**



### 10. %who

The above command will return a list of all variables that are defined in the current notebook.

**Output:**

1. a    b   c   name    z

### 11. %pinfo <variable\_name>

It returns the detailed information about the variable. We can inspect the object that is stored in the particular variable name. Let's understand the following example.

**Command -**

1. %pinfo a

**Output:**

Type: int

String form: 10

Docstring:

int([x]) -> integer

int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments

are given. If x is a number, return x.\_\_int\_\_(). For floating point

numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string,

bytes, or bytearray instance representing an integer literal in the

given base. The literal can be preceded by '+' or '-' and be surrounded

by whitespace. The base defaults to 10. Valid bases are 0 and 2-36.

Base 0 means to interpret the base from the string as an integer literal.

>>> int('0b100', base=0)

4

### 13. %matplotlib inline

It is used to display the matplotlib graph in the **Jupyter**notebook. However, this command is available in the older version of **Jupyter**notebook. In the newer versions, this is no longer in use.

### 14. %hist

This command returns the history of the current notebook. Means, what we have done so far in the current notebook.

**Output -**

1. %dhist
2. %edit
3. %env[GUINAME]
4. %automagic
5. def name()
6. print('Hello I am Stuart')
7. name()
8. def name():
9. print('Hello I am Stuart')
10. name()
11. %cd
12. %%timeit
13. square\_number = []
14. **for** num in range(1,100):
15. sqaure\_number.append(num\*\*2)
16. %%timeit
17. square\_numbers = []
18. **for** num in range(1,100):
19. sqaure\_numbers.append(num\*\*2)
20. %%timeit
21. square\_numbers = [num\*\*2 **for** num in range(1,1000)]
22. %%html
23. <html>
24. <body>
25. <h3>Student Marks</h3>
26. <table>
27. <tr>
28. <th>Name</th>
29. <th>Roll</th>
30. <th>Age</th>
31. <th>Marks</th>
32. </tr>
33. <tr>
34. ..........................................
35. ..........................................
36. ..........................................

## Execute the Html Script in IPython

We can execute the HTML and JavaScript code using the magic command. It helps us to provide some simple UI elements to our code.

**Syntax -**

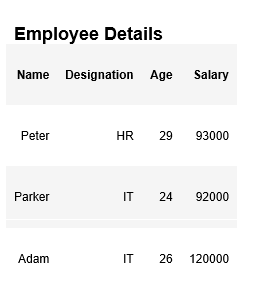
**%%html**allows us to execute the html script.

**%%js**allows us to execute the JS script.

Let's see the following example.

**Command**

1. %%html
2. <html>
3. <body>
4. <h3>Employee Details</h3>
5. <table>
6. <tr>
7. <th>Name</th>
8. <th>Designation</th>
9. <th>Age</th>
10. <th>Salary</th>
11. </tr>
12. <tr>
13. <td>Peter</td>
14. <td>HR</td>
15. <td>29</td>
16. <td>93000</td>
17. </tr>
18. <tr>
19. <td>Parker</td>
20. <td>IT</td>
21. <td>24</td>
22. <td>92000</td>
23. </tr>
24. <tr>
25. <td>Adam</td>
26. <td>IT</td>
27. <td>26</td>
28. <td>120000</td>
29. </tr>
30. </table>
31. </body>
32. </html>



### Working with Environment Variable

* **%env**

This command offers us to access and handle system environment variables.

**Output:**

{'ALLUSERSPROFILE': 'C:\\ProgramData',

'APPDATA': 'C:\\Users\\DEVANSH SHARMA\\AppData\\Roaming',

'C:\\PROGRAMDATA\\ANACONDA3\\SCRIPTS\\': 'C:\\ProgramData\\Anaconda3',

'C:\\USERS\\DEVANSH SHARMA\\APPDATA\\ROAMING\\NPM': 'C:\\Users\\DEVANSH SHARMA\\AppData\\Roaming\\npm',

'COMMONPROGRAMFILES': 'C:\\Program Files\\Common Files',

'COMMONPROGRAMFILES(X86)': 'C:\\Program Files (x86)\\Common Files',

'COMMONPROGRAMW6432': 'C:\\Program Files\\Common Files',

'COMPUTERNAME': 'DESKTOP-2VAN176',

'COMSPEC': 'C:\\WINDOWS\\system32\\cmd.exe',

'DART\_SDK': 'C:\\Program Files\\Dart\\dart-sdk',

'DRIVERDATA': 'C:\\Windows\\System32\\Drivers\\DriverData',

'FPS\_BROWSER\_APP\_PROFILE\_STRING': 'Internet Explorer',

'FPS\_BROWSER\_USER\_PROFILE\_STRING': 'Default',

'HOMEDRIVE': 'C:',

'HOMEPATH': '\\Users\\DEVANSH SHARMA',

'LOCALAPPDATA': 'C:\\Users\\DEVANSH SHARMA\\AppData\\Local',

'LOGONSERVER': '\\\\DESKTOP-2VAN176',

'NUMBER\_OF\_PROCESSORS': '4',

'ONEDRIVE': 'C:\\Users\\DEVANSH SHARMA\\OneDrive',

'OS': 'Windows\_NT',

'PATH': 'C:\\Users\\DEVANSH SHARMA\\Anaconda3;C:\\Users\\DEVANSH SHARMA\\Anaconda3\\Library\\mingw-w64\\bin;C:\\Users\\DEVANSH SHARMA\\Anaconda3\\Library\\usr\\bin;C:\\Users\\DEVANSH SHARMA\\Anaconda3\\Library\\bin;C:\\Users\\DEVANSH SHARMA..............

* **%env var -**It return value for specific variable. For example - **%env OS**

**Output:**

1. 'Windows\_NT'

* **%env var value -**It is used to set value for variable var.

### Conclusion

In this tutorial, we have covered the important magic commands of the Jupyter notebook. These commands make work easier and also provide the flexibility to use notebook efficiently.

# How to reverse a string in Python?

Python String is the collection of the Unicode character. Python has numerous functions for string manipulation, but Python string library doesn't support the in-built **"reverse()"** function. But there are various ways to reverse the string. We are defining the following method to reverse the [Python String](https://www.javatpoint.com/python-strings).

* **Using for loop**
* **Using while loop**
* **Using the slice operator**
* **Using the reversed() function**
* **Using the recursion**

### Using for loop

Here, we will reverse the given string using for loop.

1. **def** reverse\_string(str):
2. str1 = ""   # Declaring empty string to store the reversed string
3. **for** i **in** str:
4. str1 = i + str1
5. **return** str1    # It will return the reverse string to the caller function
7. str = "JavaTpoint"    # Given String
8. **print**("The original string is: ",str)
9. **print**("The reverse string is",reverse\_string(str)) # Function call

**Output:**

('The original string is: ', 'JavaTpoint')

('The reverse string is', 'tniopTavaJ')

**Explanation-**

In the above code, we have declared the **reverse\_string()** function and passed the **str** argument. In the function body, we have declared empty string variable **str1** that will hold the reversed string.

Next, the for loop iterated every element of the given string, join each character in the beginning and store in the str1 variable.

After the complete iteration, it returned the reverse order string **str1** to the caller function. It printed the result to the screen.

### Using while loop

We can also reverse a string using a while loop. Let's understand the following example.

**Example -**

1. # Reverse string
2. # Using a while loop
4. str = "JavaTpoint" #  string variable
5. **print** ("The original string  is : ",str)
6. reverse\_String = ""  # Empty String
7. count = len(str) # Find length of a string and save in count variable
8. **while** count > 0:
9. reverse\_String += str[ count - 1 ] # save the value of str[count-1] in reverseString
10. count = count - 1 # decrement index
11. **print** ("The reversed string using a while loop is : ",reverse\_String)# reversed string

**Output:**

('The original string is : ', 'JavaTpoint')

('The reversed string using a while loop is : ', 'tniopTavaJ')

**Explanation:**

In the above code, we have declared a **str** variable that holds string value. We initialized a while loop with a value of the string.

In each iteration, the value of **str[count - 1]** concatenated to the **reverse\_String** and decremented the count value. A while completed its iteration and returned the reverse order string.

### Using the slice ([]) operator

We can also reverse the given string using the **extended slice operator**. Let's understand the following example.

**Example -**

1. #  Reverse a string
2. # using  slice syntax
3. # reverse(str) Function to reverse a string
4. **def** reverse(str):
5. str = str[::-1]
6. **return** str
8. s = "JavaTpoint"
9. **print** ("The original string  is : ",s)
10. **print** ("The reversed string using extended slice operator  is : ",reverse(s))

**Output:**

('The original string is : ', 'JavaTpoint')

('The reversed string(using extended slice syntax) is : ', 'tniopTavaJ')

**Explanation:**

Generally, a slice operator accepts three parameters - start, stop and step. We provided the no value to start and end index, which indicates the start index is 0 and the end is n-1 by default. The step size is -1; it means the string continues the traverse from the end and goes to the 1 index position.

### Using reverse function with join

[Python](https://www.javatpoint.com/python-tutorial) provides the **reversed()** function to reverse the string. Let's understand the following example.

**Example -**

1. #reverse a string using reversed()
2. # Function to reverse a string
3. **def** reverse(str):
4. string = "".join(reversed(str)) # reversed() function inside the join() function
5. **return** string
7. s = "JavaTpoint"
9. **print** ("The original string is : ",s)
10. **print** ("The reversed string using reversed() is : ",reverse(s) )

**Output:**

('The original string is : ', 'JavaTpoint')

('The reversed string using reversed() is : ', 'tniopTavaJ')

**Explanation:**

In the function body, we declared the empty string separated by .dot operator. The **reversed()** string returned the reverse string it joined with the empty string separated using the **join()** function.

### Using recursion()

The string can also be reversed using the recursion. Recursion is a process where function calls itself. Consider the following example.

**Example -**

1. # reverse a string
2. # using recursion
4. **def** reverse(str):
5. **if** len(str) == 0: # Checking the lenght of string
6. **return** str
7. **else**:
8. **return** reverse(str[1:]) + str[0]
10. str = "Devansh Sharma"
11. **print** ("The original string  is : ", str)
12. **print** ("The reversed string(using recursion) is : ", reverse(str))

**Output:**

('The original string is : ', 'JavaTpoint')

('The reversed string(using reversed) is : ', 'tniopTavaJ')

**Explanation:**

In the above code, we have defined a function that accepts the string as an argument.

In the function body, we defined the base condition of recursion, if a length of a string is 0, then the string is returned, and if not then we called the function recursively.

The slice operator slice the string expects the first character and concatenated to the end of the slice string.

# How to take input in Python?

Taking input is a way of interact with users, or get data to provide some result. Python provides two [built-in](https://www.javatpoint.com/python-built-in-functions) methods to read the data from the keyboard. These methods are given below.

* input(prompt)
* raw\_input(prompt)

## input()

The input function is used in all latest version of the Python. It takes the input from the user and then evaluates the expression. The [Python](https://www.javatpoint.com/python-tutorial) interpreter automatically identifies the whether a user input a string, a number, or a list. Let's understand the following example.

**Example -**

1. # Python program showing
2. # a use of input()
4. name = input("Enter your name: ")
5. **print**(name)

**Output:**

Enter your name: Devansh

Devansh

The Python interpreter will not execute further lines until the user enters the input.

Let's understand another example.

**Example - 2**

1. # Python program showing
2. # a use of input()
3. name = input("Enter your name: ")  # String Input
4. age = int(input("Enter your age: ")) # Integer Input
5. marks = float(input("Enter your marks: ")) # Float Input
6. **print**("The name is:", name)
7. **print**("The age is:", age)
8. **print**("The marks is:", marks)

**Output:**

Enter your name: Johnson

Enter your age: 21

Enter your marks: 89

The name is: Johnson

The age is 21

The marks is: 89.0

**Explanation:**

By default, the **input()** function takes input as a string so if we need to enter the integer or float type input then the **input()** function must be type casted.

1. age = int(input("Enter your age: ")) # Integer Input
2. marks = float(input("Enter your marks: ")) # Float Input

We can see in the above code where we type casted the user input into **int** and **float**.

### How input() function works?

* The flow of the program has stopped until the user enters the input.
* The text statement which also knows as prompt is optional to write in **input()** function. This prompt will display the message on the console.
* The **input()** function automatically converts the user input into string. We need to explicitly convert the input using the type casting.
* **raw\_input() -** The raw\_input function is used in Python's older version like Python 2.x. It takes the input from the keyboard and return as a string. The Python 2.x doesn't use much in the industry. Let's understand the following example.

**Example -**

1. # Python program showing
2. # a use of raw\_input()
4. name = raw\_input("Enter your name : ")
5. **print** name

**Output:**

Enter your name: Peter

Peter

## How to check Python version?

To check the Python version, open command line (Windows), shell (Mac), or terminal (Linux/Ubuntu) and run **python -version**. It will display the corresponding Python version.

**Check Python version in the running script**

We can check the Python version in our running script. Consider the following ways to know Python version in all operating systems.

|  |  |  |
| --- | --- | --- |
| **Commands** | **Operating System/Environment** | **Output** |
| Python --version or Python -v or Python -vv | Window/Mac/Linux | Python 3.8.1 |
| import sys sys.version sys.version\_info | Python Script | 3.8.3 (default, May 13 2020, 15:29:51) [MSC v.1915 64 bit (AMD64)] |
| Import platform platform.python\_version() | Python Script | '3.8.1' |

# How to Convert Python List to String

There are instances when we need to convert the data elements gathered from one Python data type to another. Using any of the 4 methods, we can convert a list to a string in Python. These methods lead to shorter programs to perform the conversion. Iteration, comprehension, join(), and map() are all methods for converting a list into a string. Let's first define lists and strings before going into depth about all these methods and see an example of each method converting lists to strings.

## What is a List?

The list is one of the most significant data types in the Python programming language. Python lists are collections of data elements that are ordered and mutable. The list is iterable objects expressed in Python as values separated by a comma and enclosed in square brackets []. Python lists support negative indexing and can contain duplicate elements, unlike sets that do not. The fact that the elements of the list do not have to be of just one data type in a list is a key benefit of the list. Like string operations, list operations include cutting, concatenating, and other operations. We may also make nested lists, which are lists within lists.

## What is a String?

A string is described as a group of characters, each of which is a basic symbol. For instance, there are 26 characters in the English language. The computer system only works with binary numbers because it cannot read letters. Although we may see characters we typed on our monitor screens, they are a collection of 0s and 1s that are saved and processed within. An assortment of Unicode characters is a string in the Python programming language. It is a single- or double-quoted immutable iterable data type. This implies that when a string has been defined, we cannot change it.

## Methods to Convert List to String in Python

As was already described, there are four methods of converting a list to a string in Python. Let's examine each one carefully, supported by an example program and its output.

### Using join() Method

The join() method is perhaps the most Pythonic approach to converting a list to a string. join() method is designed to serve this exact purpose.

It accepts an input parameter. This parameter must be iterable. This method then joins the elements of that iterable and returns the result as a string. The iterable's values should be of the string data type.

join() method removes the elements separator of the iterable and combines them as a single string.

**Syntax of join() method:**

1. string.join( iterable )

**Code**

1. # Python program to convert a list to a string by using the join() method
3. # Creating a list with elements of string data type
4. a\_list = ["Python", "Convert", "List", "String", "Method"]
6. # Converting to string
7. string = " ".join( a\_list ) # this is read as join elements of a\_list with a separator (" ")
9. # Printing the string
10. **print** (string)
11. **print** (type(string))

**Output:**

Python Convert List String Method

<class 'str'>

We specified (" ") to be the separator for the elements of the list by placing it before the join() method. Therefore, a string containing the list's elements separated by a space is produced.

As stated earlier, let's try using join() by passing an iterable that contains some elements of the int data type. The result would be a typeerror.

**Code**

1. # Python program to convert a list to a string by using the join() method
3. # Creating a list with some elements of int data type
4. a\_list = ["Python", "Convert", 11, "List", 12, "String", "Method"]
6. # Converting to string
7. string = " ".join( a\_list )
9. # Printing the string
10. **print** (string)
11. **print** (type(string))

**Output:**

TypeError Traceback (most recent call last)

Input In [57], in ()

4 a\_list = ["Python", "Convert", 11, "List", 12, "String", "Method"]

6 # Converting to string

----> 7 string = " ".join( a\_list )

9 # Printing the string

10 print (string)

TypeError: sequence item 2: expected str instance, int found

### Using join() Method and map() Method

Using the combination of the map() and join() methods in Python gives a method for converting lists into strings. Unlike the join() method, we can use this method if the list has elements of int data type. See the example below.

We did use the map() method to convert the integer elements into a string before converting the whole list to a string because the join() method can only accept string elements.

For each value in the iterable, a given function is executed via the map() method. And in this instance, we have used it to convert every int element in the list to a string. In this example, too, we have used (" "), a space, as the separator for the string elements.

**Syntax of the map() method:**

1. map(function, iterable)

**Code**

1. # Python program to convert a list to a string by using the join() method and map method
3. # Creating a list with some elements of int data type
4. iterable = ["Python", "Convert", 11, "List", 12, "String", "Method"]
6. # Converting to string
7. string = " ".join (map (str, iterable))

10. # Printing the string
11. **print** (string)
12. **print** (type(string))

**Output:**

Python Convert 11 List 12 String Method

<class 'str'>

### Using List Comprehension

As was previously noted, if our example list contains integer elements, we cannot use the join() method. However, we can use the List comprehension and join() method together to handle this problem. In our case, we can construct an iterable list of elements using the example list that is already present with the help of list comprehension. Afterwards, a for loop is used to iterate through the components using element-wise rules. Using list comprehension, we can use the join() method to concatenate the list's elements to a blank string after the elements have been visited.

**Code**

1. # Python program to convert a list to string using the list comprehension and the join() method
3. # Creating a list with some elements of int data type
4. iterable = ["Python", "Convert", 11, "List", 12, "String", "Method"]
6. # Converting to string using list comprehension
7. string = " ".join ([str( elements ) **for** elements **in** iterable])
9. # Printing the string
10. **print** (string)
11. **print** (type(string))

**Output:**

Python Convert 11 List 12 String Method

<class 'str'>

### Iteration

The final method to convert a given list of elements into a string in Python is to loop through each element and concatenate it to the end of the string. This method is not advised because Python generates a new string whenever we append an element.

This approach is slow and should only be employed to learn how we can convert lists in Python to strings.

**Code**

1. # Python program to convert a list to string using the iteration method
3. # Creating a list with all elements of string data type
4. iterable = ["Python", "Convert", "List", "String", "Method"]
6. # Creating a blank string
7. string = ""
9. # Starting a for loop to traverse through the list elements
10. **for** element **in** iterable :
11. string = string + " " + element # Using " " as a separator for the elements of the string. However, it will add an extra space at the beginning of the string
13. # printing the string
14. **print** ( string )

**Output:**

Python Convert List String Method

The ideal approach would be to write a function that returns the results because it is very likely that converting lists to strings in Python will not be a one-time task.

Additionally, as was already noted, there aren't any big drawbacks that can be exploited to compare them. They do, however, accommodate many use situations.

If we are certain that we won't be working with any numeric numbers, we can use Methods 1 and 4. If we're unsure or our list contains integer values, use Methods 2 and 3 instead. The final method is just for us to comprehend how lists can be converted to strings in Python.

# How to append element in the list

Python provides built-in methods to append or add elements to the list. We can also append a list into another list. These methods are given below.

* **append(elmt) -** It appends the value at the end of the list.
* **insert(index, elmt) -** It inserts the value at the specified index position.
* **extends(iterable) -** It extends the list by adding the iterable object.

Let's understand these methods by the following example.

### 1. append(elmt)

This function is used to add the element at the end of the list. The example is given below.

**Example**

1. names = ["Joseph", "Peter", "Cook", "Tim"]
3. **print**('Current names List is:', names)
5. new\_name = input("Please enter a name:\n")
6. names.append(new\_name)  # Using the append() function
8. **print**('Updated name List is:', names)

**Output:**

Current names List is: ['Joseph', 'Peter', 'Cook', 'Tim']

Please enter a name:

Devansh

Updated name List is: ['Joseph', 'Peter', 'Cook', 'Tim', 'Devansh']

### 2. insert(index, elmt)

The insert() function adds the elements at the given an index position. It is beneficial when we want to insert element at a specific position. The example is given below.

**Example -**

1. list1 = [10, 20, 30, 40, 50]
3. **print**('Current Numbers List: ', list1)
5. el = list1.insert(3, 77)
6. **print**("The new list is: ",list1)
8. n = int(input("enter a number to add to list:\n"))
10. index = int(input('enter the index to add the number:\n'))
12. list1.insert(index, n)
14. **print**('Updated Numbers List:', list1)

**Output:**

Current Numbers List: [10, 20, 30, 40, 50]

The new list is: [10, 20, 30, 77, 40, 50]

enter a number to add to list:

45

enter the index to add the number:

1

Updated Numbers List: [10, 45, 20, 30, 77, 40, 50]

### 3. extend(iterable)

The extends() function is used to add the iterable elements to the list. It accepts iterable object as an argument. Below is the example of adding iterable element.

**Example -**

1. list1 = [10,20,30]
2. list1.extend(["52.10", "43.12" ])  # extending list elements
3. **print**(list1)
4. list1.extend((40, 30))  # extending tuple elements
5. **print**(list1)
6. list1.extend("Apple")  # extending string elements
7. **print**(list1)

**Output:**

[10, 20, 30, '52.10', '43.12']

[10, 20, 30, '52.10', '43.12', 40, 30]

[10, 20, 30, '52.10', '43.12', 40, 30, 'A', 'p', 'p', 'l', 'e']

# How to compare two lists in Python

Python provides multiple ways to compare the two lists. Comparison is the process when the data items of are checked against another data item of list, whether they are the same or not.

1. list1 - [11, 12, 13, 14, 15]
2. list2 - [11, 12, 13, 14, 15]
3. Output - The lists are equal

The methods of comparing two lists are given below.

* The cmp() function
* The set() function and == operator
* The sort() function and == operator
* The collection.counter() function
* The reduce() and map() function

### The cmp() function

The [Python](https://www.javatpoint.com/python-tutorial) cmp() function compares the two Python objects and returns the integer values -1, 0, 1 according to the comparison.

#### Note - It doesn't use in Python 3.x version.

### The set() function and == operator

[Python **set()** function](https://www.javatpoint.com/python-set-function) manipulate the list into the set without taking care of the order of elements. Besides, we use the equal to operator (==) to compare the data items of the list. Let's understand the following example.

**Example -**

1. list1 = [11, 12, 13, 14, 15]
2. list2 = [12, 13, 11, 15, 14]
4. a = set(list1)
5. b = set(list2)
7. **if** a == b:
8. **print**("The list1 and list2 are equal")
9. **else**:
10. **print**("The list1 and list2 are not equal")

**Output:**

The list1 and list2 are equal

### Explanation:

In the above example, we have declared the two lists to be compared with each other. We converted those lists into the set and compared each element with the help of == operator. All elements are equal in both lists, then if block executed and printed the result.

### The sort() method with == operator

Python **sort()** function is used to sort the lists. The same list's elements are the same index position it means; lists are equal.

#### Note - In the sort() method, we can pass the list items in any order because we are sorting the list before comparison.

Let's understand the following example -

Example -

1. **import** collections
3. list1 = [10, 20, 30, 40, 50, 60]
4. list2 = [10, 20, 30, 50, 40, 70]
5. list3 = [50, 10, 30, 20, 60, 40]
7. # Sorting the list
8. list1.sort()
9. list2.sort()
10. list3.sort()

13. **if** list1 == list2:
14. **print**("The list1 and list2 are the same")
15. **else**:
16. **print**("The list1 and list3 are not the same")
18. **if** list1 == list3:
19. **print**("The list1 and list2 are not the same")
20. **else**:
21. **print**("The list1 and list2 are not the same")

**Output:**

The list1 and list3 are not the same

The list1 and list2 are not the same

### The collection.counter() function

The collection module provides the **counter(),** which compare the list efficiently. It stores the data in dictionary format <value>:<frequency> and counts the frequency of the list's items.

#### Note - The order of the list's elements doesn't matter in this function.

**Example -**

1. **import** collections
3. list1 = [10, 20, 30, 40, 50, 60]
4. list2 = [10, 20, 30, 50, 40, 70]
5. list3 = [50, 10, 30, 20, 60, 40]

8. **if** collections.Counter(list1) == collections.Counter(list2):
9. **print**("The lists l1 and l2 are the same")
10. **else**:
11. **print**("The lists l1 and l2 are not the same")
13. **if** collections.Counter(list1) == collections.Counter(list3):
14. **print**("The lists l1 and l3 are the same")
15. **else**:
16. **print**("The lists l1 and l3 are not the same")

**Output:**

The lists list1 and list2 are not the same

The lists list1 and list3 are the same

### The reduce() and map()

The **map()** function accepts a function and Python iterable object (list, tuple, string, etc) as an arguments and returns a map object. The function implements to each element of the list and returns an iterator as a result.

Besides, The **reduce()** method implements the given function to the iterable object recursively.

Here, we will use both methods in combination. The **map()** function would implement the function (it can be user-define or lambda function) to every iterable object and the **reduce()** function take care of that would apply in recursive manner.

#### Note - We need to import the functool module to use the reduce() function.

Let's understand the following example.

**Example -**

1. **import** functools
3. list1 = [10, 20, 30, 40, 50]
4. list2 = [10, 20, 30, 50, 40, 60, 70]
5. list3 = [10, 20, 30, 40, 50]
7. **if** functools.reduce(**lambda** x, y: x **and** y, map(**lambda** a, b: a == b, list1, list2), True):
8. **print**("The list1 and list2 are the same")
9. **else**:
10. **print**("The list1 and list2 are not the same")
12. **if** functools.reduce(**lambda** x, y: x **and** y, map(**lambda** a, b: a == b, list1, list3), True):
13. **print**("The list1 and list3 are the same")
14. **else**:
15. **print**("The list1 and list3 are not the same")

**Output:**

The list1 and list2 are not the same

The list1 and list3 are the same

In this section, we have covered various methods of comparing two lists in Python.

# How to convert int to string in Python

We can convert an integer data type using the [Python built-in **str()** function](https://www.javatpoint.com/python-str-function). This function takes any data type as an argument and converts it into a string. But we can also do it using the "%s" literal and using the .format() function. Below is the syntax of the **str()** function.

**Syntax -**

1. str(integer\_Value)

Let's understand the following example.

**Example - 1 Using the str() function**

1. n = 25
2. # check  and print type of num variable
3. **print**(type(n))
4. **print**(n)
6. # convert the num into string
7. con\_num = str(n)
9. # check  and print type converted\_num variable
10. **print**(type(con\_num))
11. **print**(con\_num)

**Output:**

<class 'int'>

25

<class 'str'>

25

**Example - 2 Using the "%s" integer**

1. n = 10
3. # check and print type of n variable
4. **print**(type(n))
6. # convert the num into a string and print
7. con\_n = "% s" % n
8. **print**(type(con\_n))

**Output:**

<class 'int'>

<class 'str'>

**Example - 3: Using the .format() function**

1. n = 10
3. # check  and print type of num variable
4. **print**(type(n))
6. # convert the num into string and print
7. con\_n = "{}".format(n)
8. **print**(type(con\_n))

**Output:**

<class 'int'>

<class 'str'>

**Example - 4: Using f-string**

1. n = 10
3. # check  and print type of num variable
4. **print**(type(n))
6. # convert the num into string
7. conv\_n = f'{n}'
9. # print type of converted\_num
10. **print**(type(conv\_n))

**Output:**

<class 'int'>

<class 'str'>

We have defined all methods of converting the integer data type to the string type. You can use one of them according to your requirement.

# How to print in same line in Python

The Python's **print()** function is used to print the result or output to the screen. By default, it jumps to the newline to printing the next statement. It has a pre-defined format to print the output. Let's understand the following example.

**Example - 1**

1. **print**("Welcome")
2. **print**("To")
3. **print**("JavaTpoint")

**Output:**

Welcome

To

JavaTpoint

Or, we can write the complete statement in single **print()** function.

1. **print**("Welcome To JavaTpoint")

**Output:**

Welcome To JavaTpoint

The [Python](https://www.javatpoint.com/python-tutorial)

**print()** function has an argument called **end,** which prevents jump into the newline. Let's understand the following example.

**Example - 3:**

1. list1 = [10,11,12,13,14,15]
2. **for** i **in** list1:
3. **print**(i, end = " ")

**Output:**

10 11 12 13 14 15

In the above code, we declared a list and iterated each element using for loop. The print() function printed the first element of the list and then printed the end value which we assigned as ' ' whitespace and it will be printed till the element of the list.

We can assign any literal to the end. Let's understand the following example.

**Example - 4**

1. list1 = [10,11,12,13,14,15]
2. **for** i **in** list1:
3. **print**(i,end = "&")

**Output:**

10&11&12&13&14&15&

# How to concatenate two strings in Python

Python string is a collection of Unicode characters. Python provides many [built-in functions](https://www.javatpoint.com/python-built-in-functions) for string manipulation. String concatenation is a process when one string is merged with another string. It can be done in the following ways.

* Using + operators
* Using join() method
* Using % method
* Using format() function

Let's understand the following string concatenation methods.

### Using + operator

This is an easy way to combine the two strings. The + operator adds the multiple strings together. Strings must be assigned to the different variables because strings are immutable. Let's understand the following example.

**Example -**

1. # Python program to show
2. # string concatenation
4. # Defining strings
5. str1 = "Hello "
6. str2 = "Devansh"
8. # + Operator is used to strings concatenation
9. str3 = str1 + str2
10. **print**(str3)   # Printing the new combined string

**Output:**

Hello Devansh

**Explanation:**

In the above example, the variable str1 stores the string "Hello", and variable str2 stores the "Devansh". We used the + operator to combine these two string variables and stored in str3.

### Using join() method

The join() method is used to join the string in which the str separator has joined the sequence elements. Let's understand the following example.

**Example -**

1. # Python program to
2. # string concatenation
4. str1 = "Hello"
5. str2 = "JavaTpoint"
7. # join() method is used to combine the strings
8. **print**("".join([str1, str2]))
10. # join() method is used to combine
11. # the string with a separator Space(" ")
12. str3 = " ".join([str1, str2])
14. **print**(str3)

**Output:**

HelloJavaTpoint

Hello JavaTpoint

**Explanation:**

In the above code, the variable str1 stores the string "Hello" and variable str2 stores the "JavaTpoint". The join() method returns the combined string that is stored in the str1 and str2. The join() method takes only list as an argument.

### Using % Operator

The % operator is used for string formatting. It can also be used for string concatenation. Let's understand the following example.

**Example -**

1. # Python program to demonstrate
2. # string concatenation
4. str1 = "Hello"
5. str2 = "JavaTpoint"
7. # % Operator is used here to combine the string
8. **print**("% s % s" % (str1, str2))

**Output:**

Hello JavaTpoint

**Explanation -**

In the above code, the %s represents the string data-type. We passed both variables's value to the %s that combined the strings and returned the "Hello JavaTpoint".

### Using format() function

[Python](https://www.javatpoint.com/python-tutorial) provides the **str.format()** function, which allows use of multiple substitutions and value formatting. It accepts the positional arguments and concatenates the string through positional formatting. Let's understand the following example.

**Example -**

1. # Python program to show
2. # string concatenation
4. str1 = "Hello"
5. str2 = "JavaTpoint"
7. # format function is used here to
8. # concatenate the string
9. **print**("{} {}".format(str1, str2))
11. # store the result in another variable
12. str3 = "{} {}".format(str1, str2)
14. **print**(str3)

**Output:**

Hello JavaTpoint

Hello JavaTpoint

**Explanation:**

In the above code, the format() function combines both strings and stores into the str3 variable. The curly braces {} are used as the position of the strings.

# How to convert list to dictionary in Python

Lists and Dictionaries are two data structure which is used to store the Data. List stores the heterogeneous data type and Dictionary stores data in key-value pair. Here, we are converting the [Python](https://www.javatpoint.com/python-tutorial) list into dictionary. Since list is ordered and dictionary is unordered so output can differ in order. [Python list](https://www.javatpoint.com/python-lists) stores the element in the following way.

1. student\_marks = [56, 78, 96, 37, 85]

On the other hand, Dictionary is unordered, and stores the unique data. It stores the data in key value pair where each key is associated with it value. [Python Dictionary](https://www.javatpoint.com/python-dictionary) stores the data in following way.

1. student\_dict = {'Abhinay': 56, 'Sharma': 78, 'Himanshu': 96, 'Peter': 37}

In this tutorial, we will learn the conversion Python list to dictionary.

**Sample Input:**

1. Input : ['Name', 'Abhinay', 'age', 25, 'Marks', 90]
2. Output : {'Name', 'Abhinay', 'age', 25, 'Marks', 90}
4. Input : ['a', 10, 'b', 42, 'c', 86]
5. Output : {'a', 10, 'b', 42, 'c', 86}

Let's understand the following methods.

### Method - 1 Using Dictionary Comprehension

We can convert the list into dictionary using the dictionary comprehension. Let's understand the following code.

**Example -**

1. student = ["James", "Abhinay", "Peter", "Bicky"]
3. student\_dictionary = { stu : "Passed" **for** stu **in** student }
5. **print**(student\_dictionary)

**Output:**

{'James': 'Passed', 'Abhinay': 'Passed', 'Peter': 'Passed', 'Bicky': 'Passed'}

**Explanation -**

In the above code, we have created a student list to be converted into the dictionary. Using the dictionary compression, we converted the list in dictionary in a single line. The list elements tuned into key and passed as a value.

Let's understand another example.

**Example - 2**

1. list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2. square\_dict = {n: n\*n **for** n **in** list1}
3. **print**(square\_dict)

**Output:**

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}

**Explanation:**

In the above code, we have created **square\_dict** with number-square key/value pair.

### Method - 2 Using zip() function

The **zip()** function is used to zip the two values together. First, we need to create an iterator and initialize to any variable and then typecast to the **dict()** function.

Let's understand the following example.

**Example -**

1. **def** Convert\_dict(a):
2. init = iter(list1)
3. res\_dct = dict(zip(init, init))
4. **return** res\_dct

7. # Driver code
8. list1 = ['x', 1, 'y', 2, 'z', 3]
9. **print**(Convert\_dict(list1))

**Output:**

{'x': 1, 'y': 2, 'z': 3}

# How to declare a global variable in Python

## What is a Global Variable?

The global variables are the ones that are available both within and outside of any function, provided they are defined outside a function which is a global scope. Let's explore the creation of a global variable.

In this example, we will define and access a global variable.

**Code**

1. # Python program to show how to define a global variable
3. # defining a function
4. **def** func():
5. **print**( "Inside the defined function the string is: ", var ) # this is local scope
6. # This is global scope
7. var = "Declaring Global Variable"
8. # calling the function
9. func()
10. **print**("Outside the defined function the string is: ", var)

**Output:**

Inside the defined function the string is: Declaring Global Variable

Outside the defined function the string is: Declaring Global Variable

Accessing the Global Variable

The variable var is accessed both within and outside the function func(), and since it is declared in the global scope, it is designated as a global variable.

Since we have not created local variables, the value of the global variable will be applied; however, we must ensure that the names of the local and global variables match.

What if a variable with the same identifier is initialized globally and inside a defined function? What will occur if we modify the value of a variable within the function func()? The problem is whether changing the local variable will change the global variable or, conversely, changing the global variable will change the local variable. The following example of code is used to evaluate it:

**Code**

1. # Python program to show how to access global and local variables
3. # defining a function
4. **def** func():
5. var = "Global Variable" # declared variable in the local scope
6. **print**( var )

9. # Declared variable in the global scope
10. var = "I am learning Python"
11. func()
12. **print**( var )

**Output:**

Global Variable

I am leaning Python

Suppose a certain variable, having the name just as given to the variable in the global scope, is declared inside the function's local scope. In that case, it will only show the value provided within the defined function but not the global value.

## Changing Global Variable inside the Local Scope

What would happen if we attempted to alter the value of a variable defined in the global scope inside the function? Let's examine it with the help of the example below.

**Code**

1. # We will try to change the value of the variable defined in the global scope in the local scope
3. # defining a function
4. **def** func():
5. var = var + "Global Variable"
6. **print**( "Inside the defined function", var )

9. # Declaring a variable in the global scope
10. var = "Python Tutorial"
11. func()

**Output:**

4 def func():

----> 5 var = var + "Global Variable"

6 print( "Inside the defined function", var )

UnboundLocalError: local variable 'var' referenced before assignment

We must include the "global" Python keyword for the preceding application to run. Let us now study the global variable.

## Global Keyword

When assigning values to or changing the variables declared in global scope within a function, we must use the global Python keyword. The keyword is not required to display or access a global scope variable. Since we declared var inside the function func(), the Python interpreter "assumes" that we desire a local variable, which is why the first sentence raises the error. It is considered local if a variable is modified or declared inside a function without being declared a global variable using the global keyword. The following example demonstrates how to employ the keyword "global" to instruct the Python interpreter that we wish to access the global variable.

**Code**

1. # Python program to show the use of the global keyword to modify a variable, defined in global scope, inside a local scope.
2. # defining a function
3. **def** func():
4. **global** var
5. var = var + " " + "Global Variable"
6. **print**( "Inside the defined function: ", var )

9. # Declaring a variable in the global scope
10. var = "Python Tutorial"
11. func()
12. **print**( "Outside the function: ", var ) # this will show if the variable var is changed in global scope also or not

**Output:**

Inside the defined function: Python Tutorial Global Variable

Outside the function: Python Tutorial Global Variable

### Example to Show Use of Local and Global Variables

**Code**

1. # Python program to show how to use global and local variables
3. var = 10
5. # this will show the global value of var as there is no local var
6. **def** func1():
7. **print**('Inside the first function: ', var)
9. # declaring a local variable named var
10. **def** func2():
11. var = 15
12. **print**('Inside the second function: ', var)
14. # using the global keyword to modify var in global scope
15. **def** func3():
16. **global** var
17. var += 3
18. **print**("Inside the third function: ", var)

21. # Global scope
22. **print**('global value of variable: ', var)
23. func1()
24. **print**('global value of variable: ', var)
25. func2()
26. **print**('global value of variable: ', var)
27. func3()
28. **print**('global value of variable: ', var)

**Output:**

global value of variable: 10

Inside the first function: 10

global value of variable: 10

Inside the second function: 15

global value of variable: 10

Inside the third function: 13

global value of variable: 13

# How to reverse a number in Python

It is the most asked programming question in the interview. We can reverse the integer number in Python using the different methods.



Here we will write the program which takes input number and reversed the same. Let's understand the following methods of reversing the integer number.

* Using while loop
* Using recursion

### Reverse a number using Python while loop

First, we understand the algorithm of this program. It will make easy to understand the program logic. Once you get the logic, you can write the program in any language, not only [Python](https://www.javatpoint.com/python-tutorial).

### Algorithm

1. Input Integer:  number
2. (1) Initialize variable revs\_number = 0
3. (2) Loop **while** number > 0
4. (a) Multiply revs\_number by 10 **and** add remainder of number
5. divide by 10 to revs\_number
6. revs\_number = revs\_number\*10 + number%10;
7. (b) Divide num by 10
8. (3) Return revs\_number

Let's implement the above algorithm in program.

### Program

1. # Ask for enter the number from the use
2. number = int(input("Enter the integer number: "))
4. # Initiate value to null
5. revs\_number = 0
7. # reverse the integer number using the while loop
9. **while** (number > 0):
10. # Logic
11. remainder = number % 10
12. revs\_number = (revs\_number \* 10) + remainder
13. number = number // 10
15. # Display the result
16. **print**("The reverse number is : {}".format(revs\_number))

**Output:**

Enter the integer number: 12345

The reverse number is: 54321

**Explanation -**

Let's understand this program step by step.

We initialed a **number** variable for user input and variable **revs\_number** initial value to null.

**First Iteration**

Reminder = number %10  
Reminder = 12345%10 = 5  
Reverse = Reverse \*10 + Reminder Initial value of revs\_number is null  
Reverse = 0 \* 10 + 5 = 0 + 5 = 5  
Number = Number //10  
Number = 1234 //10 = 1234 // Now loop will iterate on this number.

**Second Iteration**

Now the number is 123, and the revs\_number is 5. The while checks its condition and executes for the next iteration.

Reminder = Number % 10  
Reminder = 1234 % 10 = 4  
Reverse = Reverse \*10+ Reminder = 5 \* 10 + 4  
Reverse = 50 + 4 = 54  
Number = Number //10 = 12345 //10  
Number = 123

**Third Iteration**

From the Second Iteration, the values of both Number and Reverse have been changed as: number = 123 and revs\_number = 54

Reminder = Number %10  
Reminder = 123%10 = 3  
Reverse = Reverse \*10+ Reminder = 54 \* 10 + 3  
Reverse = 540 + 3 = 543  
Number = Number //10 = 123//10  
Number = 12

**Fourth Iteration**

The modified number is 12 and the revs\_number is 543: Now while executes again.

Reminder = Number %10  
Reminder = 12 %10 = 2  
Reverse = Reverse \*10+ Reminder = 543 \* 10 + 2  
Reverse = 5430 + 2 = 5432  
Number = Number //10 = 12//10  
Number = 1

**Fifth Iteration**

Reminder = Number %10  
Reminder = 1 %1 0 = 1  
Reverse = Reverse \*10+ Reminder = 5432 \* 10 + 1  
Reverse = 54320 + 1 = 54321

while loop is terminated because if found the false as a Boolean result.

You can enter the different number and check the result.

### Reverse a Number Using Recursion

Let's understand the following example.

1. num = int(input("Enter the number: "))
2. revr\_num = 0    # initial value is 0. It will hold the reversed number
3. **def** recur\_reverse(num):
4. **global** revr\_num   # We can use it out of the function
5. **if** (num > 0):
6. Reminder = num % 10
7. revr\_num = (revr\_num \* 10) + Reminder
8. recur\_reverse(num // 10)
9. **return** revr\_num

12. revr\_num = recur\_reverse(num)
13. **print**("n Reverse of entered number is = %d" % revr\_num)

**Output:**

Enter the number: 5426

The Reverse of entered number is = 6245

Logic is same in both programs. Once you understand the logic, it will easy to do it by own.

# What is an object in Python

Python is an object-oriented programming language. Everything is in Python treated as an object, including variable, function, list, tuple, dictionary, set, etc. Every object belongs to its class. For example - An integer variable belongs to integer class. An object is a real-life entity. An object is the collection of various data and functions that operate on those data. An object contains the following properties.

* **State -** The attributes of an object represents its state. It also reflects the properties of an object.
* **Behavior -** The method of an object represents its behavior.
* **Identity -** Each object must be uniquely identified and allow interacting with the other objects.

Let's understand the object in the aspect of classes.

The classes and objects are the essential key to the object-oriented programming. Classes are the blueprint of the object. Classes are used to bundling the data and functionality together. Each newly created class must have its object. Let's understand real-life example of class and object.

A human is a class which may have may attributes such as walking, sleeping, thinking, etc. Suppose we want to name and age of 100 humans, so we don't need to create a class for every person. We just need to instantiate the multiple objects of that perticular class.

The class contains the user-defined data structure that holds the own data members such as variables, constructs, and member functions, which can be accessed by creating an object of the class.

The syntax of creating a class is given below. The syntax of creating a class is given below.

### Syntax:

1. **class** ClassName:
2. #statement\_suite

The class keyword is used to define the class and the user-define class name replaces ClassName.

### Creating an Object of class

The object is essential to work with the class attributes. Instantiate is a term used when we create the object of any class, and the instance is also referred to as an object. The object is created using the class name. The syntax is given below.

**Syntax:**

1. <object-name> = <**class**-name>(<arguments>)

In the following example, we have created the object of Person class.

**Example -**

1. **class** Person:
2. name = "John"
3. age = 24
4. **def** display (self):
5. **print**("Age: %d \nName: %s"%(self.age,self.name))
6. # Creating a emp instance of Employee class
7. per = Person()
8. per.display()

**Output:**

Age: 24

Name: John

**Explanation:**

In the above code, we have created a Person class which consisted of two attributes age, name and display function. We created the object of person class called **per** . Using the object along with the .dot operator, we accessed the class function.

# How to create a DataFrames in Python

A Data Frame is a two-dimension collection of data. It is a data structure where data is stored in tabular form. Datasets are arranged in rows and columns; we can store multiple datasets in the data frame. We can perform various arithmetic operations, such as adding column/row selection and columns/rows in the data frame.

We can import the DataFrames from the external storage; these storages can be referred to as the [SQL](https://www.javatpoint.com/sql-tutorial) Database, CSV file, and an Excel file. We can also use the lists, dictionary, and from a list of dictionary, etc.

In this tutorial, we will learn to create the data frame in multiple ways. Let's understand these different ways.

First, we need to install the [pandas](https://www.javatpoint.com/python-pandas) library into the [Python](https://www.javatpoint.com/python-tutorial) environment.

### An empty dataframe

We can create a basic empty Dataframe. The dataframe constructor needs to be called to create the DataFrame. Let's understand the following example.

**Example -**

1. # **import** pandas as pd
2. **import** pandas as pd
4. # Calling DataFrame constructor
5. df = pd.DataFrame()
7. print(df)

**Output:**

Empty DataFrame

Columns: []

Index: []

### Method - 2: Create a dataframe using List

We can create dataframe using a single list or list of lists. Let's understand the following example.

**Example -**

1. # importing pandas library
2. **import** pandas as pd
4. # string values in the list
5. lst = ['Java', 'Python', 'C', 'C++',
6. 'JavaScript', 'Swift', 'Go']
8. # Calling DataFrame constructor on list
9. dframe = pd.DataFrame(lst)
10. print(dframe)

**Output:**

0 Java

1 Python

2 C

3 C++

4 JavaScript

5 Swift

6 Go

### Method - 3: Create Dataframe from dict of ndarray/lists

The dict of ndarray/lists can be used to create a dataframe, all the **ndarray** must be of the same length. The index will be a range(n) by default; where n denotes the array length. Let's understand the following example.

**Example -**

1. **import** pandas as pd
3. # assign data of lists.
4. data = {'Name': ['Tom', 'Joseph', 'Krish', 'John'], 'Age': [20, 21, 19, 18]}
6. # Create DataFrame
7. df = pd.DataFrame(data)
9. # Print the output.
10. print(df)

**Output:**

Name Age

0 Tom 20

1 Joseph 21

2 Krish 19

3 John 18

### Method - 4: Create a indexes Dataframe using arrays

Let's understand the following example to create the indexes dataframe using arrays.

**Example -**

1. # DataFrame using arrays.
2. **import** pandas as pd
4. # assign data of lists.
5. data = {'Name':['Renault', 'Duster', 'Maruti', 'Honda City'], 'Ratings':[9.0, 8.0, 5.0, 3.0]}
7. # Creates pandas DataFrame.
8. df = pd.DataFrame(data, index =['position1', 'position2', 'position3', 'position4'])
10. # print the data
11. print(df)

**Output:**

Name Ratings

position1 Renault 9.0

position2 Duster 8.0

position3 Maruti 5.0

position4 Honda City 3.0

**Explanation -**

In the above code, we have defined the column name with the various car names and their ratings. We used the array to create indexes.

### Method - 5: Create Dataframe from list of dicts

We can pass the lists of dictionaries as input data to create the Pandas dataframe. The column names are taken as keys by default. Let's understand the following example.

**Example -**

1. # the example is to create
2. # Pandas DataFrame by lists of dicts.
3. **import** pandas as pd
5. # assign values to lists.
6. data = [{'A': 10, 'B': 20, 'C':30}, {'x':100, 'y': 200, 'z': 300}]
8. # Creates DataFrame.
9. df = pd.DataFrame(data)
11. # Print the data
12. print(df)

**Output:**

A B C x y z

0 10.0 20.0 30.0 NaN NaN NaN

1 NaN NaN NaN 100.0 200.0 300.0

Let's understand another example to create the pandas dataframe from list of dictionaries with both row index as well as column index.

**Example - 2:**

1. **import** pandas as pd
3. # assigns values to lists.
4. data = [{'x': 1, 'y': 2}, {'A': 15, 'B': 17, 'C': 19}]
6. # With two column indices, values same
7. # as dictionary keys
8. dframe1 = pd.DataFrame(data, index =['first', 'second'], columns =['x', 'y'])
10. # With two column indices with
11. # one index with other name
12. dframe2 = pd.DataFrame(data, index =['first', 'second'], columns =['x', 'y1'])
14. # print the first data frame
15. print (dframe1, "\n")
16. # Print the second DataFrame.
17. print (dframe2)

**Output:**

x y

first 1.0 2.0

second NaN NaN

x y1

first 1.0 NaN

second NaN NaN

Let's understand another example to create dataframe by passing lists of dictionary and rows.

**Example - 3**

1. # The example is to create
2. # Pandas DataFrame by passing lists of
3. # Dictionaries and row indices.
4. **import** pandas as pd
6. # assign values to lists
7. data = [{'x': 2, 'z':3}, {'x': 10, 'y': 20, 'z': 30}]
9. # Creates padas DataFrame by passing
10. # Lists of dictionaries and row index.
11. dframe = pd.DataFrame(data, index =['first', 'second'])
13. # Print the dataframe
14. print(dframe)

**Output:**

x y z

first 2 NaN 3

second 10 20.0 30

We have discussed the three ways to create the dataframe using the lists of dictionary.

### Method - 6: Create Dataframe using the zip() function

The zip() function is used to merge the two lists. Let's understand the following example.

**Example -**

1. # The example is to create
2. # pandas dataframe from lists using zip.
4. **import** pandas as pd
6. # List1
7. Name = ['tom', 'krish', 'arun', 'juli']
9. # List2
10. Marks = [95, 63, 54, 47]
12. #  two lists.
13. # and merge them by using zip().
14. list\_tuples = list(zip(Name, Marks))
16. # Assign data to tuples.
17. print(list\_tuples)
19. # Converting lists of tuples into
20. # pandas Dataframe.
21. dframe = pd.DataFrame(list\_tuples, columns=['Name', 'Marks'])
23. # Print data.
24. print(dframe)

**Output:**

[('john', 95), ('krish', 63), ('arun', 54), ('juli', 47)]

Name Marks

0 john 95

1 krish 63

2 arun 54

3 juli 47

### Method - 7: Create Dataframe from Dicts of series

The dictionary can be passed to create a dataframe. We can use the Dicts of series where the subsequent index is the union of all the series of passed index value. Let's understand the following example.

**Example -**

1. # Pandas Dataframe from Dicts of series.
3. **import** pandas as pd
5. # Initialize data to Dicts of series.
6. d = {'Electronics' : pd.Series([97, 56, 87, 45], index =['John', 'Abhinay', 'Peter', 'Andrew']),
7. 'Civil' : pd.Series([97, 88, 44, 96], index =['John', 'Abhinay', 'Peter', 'Andrew'])}
9. # creates Dataframe.
10. dframe = pd.DataFrame(d)
12. # print the data.
13. print(dframe)

**Output:**

Electronics Civil

John 97 97

Abhinay 56 88

Peter 87 44

Andrew 45 96

# How to Remove an Element from a List in Python

We can store items of several data types in an ordered sequence using the Python List data structure. Square brackets ([]) are used to encapsulate the data, while commas are used to separate the entries (,).

Python provides numerous methods to help us remove a specific item from a list. The three methods are remove(), pop(), and clear().

Along with the approaches outlined above, we can use the del keyword to remove items from a list.

## Python remove() method

A built-in method in Python that we can use with the list data type is remove(). It helps to eliminate the first item from the list that matches the given item.**Syntax:**

1. list.remove(element)

**element:-** The item of the list we would like to remove.

When there are duplicate elements in a list, the first item that matches the provided item is eliminated. If the supplied element does not exist in the list, an exception will be thrown stating that the element does not exist in the list. There is no value returned by the delete () function. The value is passed as a parameter to delete(). Therefore it must be of the right datatype.

**Code**

1. # Python program to remove an element from a list using the remove() function
3. my\_list = ['Javatpoint', 'Python', 'Tutorial', 'List',
4. 'Element', 'Removal']
5. **print**("Initial List is :", my\_list)
7. # through remove() deleting 'Python' from the my\_list
8. my\_list.remove('Python')
9. **print**( "After using the function :", my\_list )

**Output:**

Initial List is : ['Javatpoint', 'Python', 'Tutorial', 'List', 'Element', 'Removal']

After using the function : ['Javatpoint', 'Tutorial', 'List', 'Element', 'Removal']

### Python pop() method

Based on the provided index, the pop() function eliminates the element present at that index from the list.

**Syntax**

1. list.pop( index )

**index:** There is only one argument, named index, for the pop() method.

We must pass the item's index to eliminate that entry from the list. The index begins at zero. Give the index as 0 to fetch the first item from the list. We can give the index as -1 to remove the last item. The index parameter is optional. In the absence of a value, the last item from the list is delivered from the list, and the parameter's default value is taken to be -1. The pop() function returns an error with the message IndexError: pop index if the supplied index is not valid or out of limits.

**Code**

1. # Python program to show how to use the pop() function
3. lst = ["Python", "Remove", "Elements", "List", "Tutorial"]
4. **print**("Initial List is :", lst)
6. # using pop() function to remove element at index 2 of the list
7. element = lst.pop(2)
8. **print**("Element that is popped :", element)
9. **print**("List after deleting the element", lst)

**Output:**

Initial List is : ['Python', 'Remove', 'Elements', 'List', 'Tutorial']

Element that is popped : Elements

List after deleting the element ['Python', 'Remove', 'List', 'Tutorial']

### Python clear() method

No value is returned by this method. The clear() function is used to empty the list().

**Syntax:**

1. list.clear()

No parameters.

There is no return value. The list() is emptied using clear() method.

**Code**

1. # Python program to clear all the elements from the list
3. lst = ["Python", "Remove", "Elements", "List", "Tutorial"]
4. **print**("Initial List is :", lst)
6. # Using the clear() function
7. element = lst.clear()
8. **print**(element)
9. **print**(lst)

**Output:**

Initial List is : ['Python', 'Remove', 'Elements', 'List', 'Tutorial']

None

[]

### Using del Keyword

We can apply the del keyword of Python followed by the list's name to remove an item from the list. The item's index must be supplied to the list. The indexing in Python begins at zero.

**Syntax:**

1. **del** list[index]

We can also delete a part of the list using the del keyword. We can do this by using slicing. The keyword will erase the components within that range if the del keyword is provided with appropriate start and stop indices from the list. The syntax is as follows:

**Syntax:**

1. **del** list[start : stop]

Here is an illustration of how to use del to delete the elements from the list created.

**Code**

1. # Python program to show how to use del keyword
3. lst = ["Python", "Remove", "Elements", "List", "Tutorial", "Clear", "Pop", "Remove", "Delete"]
4. **print**("The Initial list is ", lst)
6. # Removing the first element of the list
7. **del** lst[0]
8. **print**("After removing the first element new list is", lst)
10. # Removing the last element from the list
11. **del** lst[-1]
12. **print**("After removing the last element new list is", lst)
14. # To remove the elements between a range
15. **del** lst[:3]
16. **print**("After removing element from index:5", lst)
18. # Removing the last two elements from the list
19. **del** lst[-2]
20. **print**("After removing the last 2 elements from the list", lst)
22. # Removing the elements between a range having the start and stop indices
23. **del** lst[1:5]
24. **print**("After removing elements present in the range 1:5", lst)

**Output:**

The Initial list is ['Python', 'Remove', 'Elements', 'List', 'Tutorial', 'Clear', 'Pop', 'Remove', 'Delete']

After removing the first element new list is ['Remove', 'Elements', 'List', 'Tutorial', 'Clear', 'Pop', 'Remove', 'Delete']

After removing the last element new list is ['Remove', 'Elements', 'List', 'Tutorial', 'Clear', 'Pop', 'Remove']

After removing element from index:5 ['Tutorial', 'Clear', 'Pop', 'Remove']

After removing the last 2 elements from the list ['Tutorial', 'Clear', 'Remove']

After removing elements present in the range 1:5 ['Tutorial']

# How to Round number in Python

Python provides the built-in round() function, which used to round off a number to a given number of digits. It takes the two arguments, first is n, second is n digits and then it returns number n after rounding it to ndigits. By default, it rounds off the number n to the nearest integer.

**For example -** If we want to round off a number, let's suppose 7.5. It will be rounded off to the nearest whole number is 7. However, the number 7.56 will be rounded off to 7.5 by one places to give.

The round() function is essential when working with the number of floats that may have many decimal places. The round() function makes easy and simple. The syntax is given below.

### Syntax:

1. round(number, number of digits)

The parameters are -

* number - It represents the given number to be rounded.
* number of digits(Optional) - It represents the number of digits up to which the given number is to be rounded.

Let's understand the following example -

### Example -

1. print(round(15))
3. # For floating point
4. print(round(25.8))
5. print(round(25.4))

**Output:**

15

26

25

Now, the second parameter is used.

### Example -

1. print(round(25.4654, 2))
3. # when the (ndigit+1)th digit is >=5
4. print(round(25.4276, 3))
6. # when the (ndigit+1)th digit is <5
7. print(round(25.4173, 2))

**Output:**

25.47

25.428

25.42

### The real-life example of the round() function

The round() function is most useful while changing fractions to decimals. We generally get the number of a decimal points such as if we do 1/3 then we get 0.333333334, but we use either two or three digits to the right of the decimal points. Let's understand the following example.

**Example -**

1. x = 1/6
2. print(x)
3. print(round(x, 2))

**Output:**

0.16666666666666666

0.17

Another example

**Example -**

1. print(round(5.5))
2. print(round(5))
3. print(round(6.5))

**Output:**

6

5

6

The **round()** function rounds 5.5 up to 6 and 6.5 down to 6. This is not a bug, the **round()** behaves like this way.

# How to sort a dictionary in Python

Python dictionary is the collection of data which stored in the key-value form. Each key is associated with its value. It is mutable in nature, which means we can change data after its creation.

It is the unordered collection of the data and allows storing duplicate values, but the key must be unique.

Dictionary is declared using the curly braces {}, and the key-value pair is separated by a comma.

1. dict1 = {'name': 'Devansh', 'age': 22, 'Rollno':90014}
2. print(dict1)

**Output:**

### Why need to sort the dictionary

* The search time complexity of the list is O(n), and the dictionary has search time complexity 0(1), which makes that the dictionary is faster than the list. The dictionary can be used in place for list whenever it needs.
* The sorting allows us to analyze the data efficiently when we are working with the data-structure.
* A sorted dictionary provides a better understanding to handle the complex operations.

Let's understand the various ways to sort the dictionary.

* Sorting by keys
* Sorting by values
* Sorting Algorithm
* Reversing the sorted order

### Sorting By Keys and Values

[Python](https://www.javatpoint.com/python-tutorial) offers the [built-in keys functions](https://www.javatpoint.com/python-built-in-functions) keys() and values() functions to sort the dictionary. It takes any iterable as an argument and returns the sorted list of keys. We can use the keys to sort the dictionary in the ascending order. Let's understand the following example.

**Example -**

1. names = {1:'Alice' ,2:'John' ,4:'Peter' ,3:'Andrew' ,6:'Ruffalo' ,5:'Chris' }
2. #print a sorted list of the keys
3. print(sorted(names.keys()))
4. #print the sorted list with items.
5. print(sorted(names.items()))

**Output:**

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

[(1, 'Alice'), (2, 'John'), (3, 'Andrew'), (4, 'Peter'), (5, 'Chris'), (6, 'Ruffalo')]

**Explanation -**

In the above code, we have declared a dictionary **names**. We used the built-in function along with the **sorted()** function that returned the list of the sorted keys. Next, we used the **items()** function to get the dictionary in the sorted order.

### Sorting Algorithm

There are various sorting algorithm to sort a dictionary; we can use other arguments in the sorted method. Let's understand the following example.

**Example -**

1. daynames = { 'one' : 'Monday' ,  'six' : 'Saturday' ,'three' : 'Wednesday' ,  'two' : 'Tuesday' , 'five': 'Friday' ,  'seven': 'Sunday' }
2. print(daynames)
3. number = { 'one' : 1 , 'two' : 2 , 'three' : 3 , 'four' : 4 , 'five' : 5 , 'six' : 6 , 'seven' : 7}
4. print(sorted(daynames , key=number.\_\_getitem\_\_))
5. print([daynames[i] **for** i in sorted(daynames , key=number.\_\_getitem\_\_)])

**Output:**

{'one': 'Monday', 'six': 'Saturday', 'three': 'Wednesday', 'two': 'Tuesday', 'five': 'Friday', 'seven': 'Sunday'}

['one', 'two', 'three', 'five', 'six', 'seven']

['Monday', 'Tuesday', 'Wednesday', 'Friday', 'Saturday', 'Sunday']

### Reverse the sorted Order

The dictionary can be reversed using the **reverse** argument. Let's understand the following example.

**Example -**

1. a = {'a':2 ,'b':1 ,'c':3 ,'d':4 ,'e':5 ,'f':6 }
2. print(sorted(a.values() ,  reverse= True))

**Output:**

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

In this tutorial, we have discussed how to sort the dictionary in Python. A sorted dictionary is easy to handle the large amount of data and gives us a fast search result.

# Strong Number in Python

In this tutorial, we will learn a Python program to find a given number is a Strong number or not.

### What is a strong number?

A Strong number is a special number whose sum of the all digit factorial should be equal to the number itself.

To find a whether given number is strong or not. We pick each digit from the given number and find its factorial, and we will do this every digit of the number.

Once we get the factorial of all digit, then we do the sum of factorials. If the sum is equal to the given number then the given number is strong otherwise not.

**For example -** The given number is 145, we have to pick each digit and find the factorial 1! = 1, 4! = 24, and 5! = 120.

Now, we will do the sum of the factorials, we get 1+24+120 = 145, which is exactly the same as the given number. So we can say that 145 is a strong number.

We got the logic of the strong number. Now implements it using the [Python Program](https://www.javatpoint.com/python-programs).

**Problem Approach**

1. Ask the user to enter an integer number.
2. Find the factorial of each digit in the number using the two while loop.
3. Now, sum up all the factorial number.
4. Check if it is equal to the given number.
5. Print the Output.
6. Exit

**Sample Input:** num = 132

**Sample Output:** Given number is not a strong number

**Explanation:** 1! + 3! + 2! = 9 which is not equal to the 132

**Sample Input:** num = 145

**Sample Output:** Given number is a strong number.

### Python Program to Find Strong Number

Below is the code of the [Python](https://www.javatpoint.com/python-tutorial) program to print the given number is a strong or not.

**Example -**

1. # Variable to store sum of the numbers
2. sum=0
3. # Ask user to enter the number
4. num=**int**(input("Enter a number:"))
5. # temporary variable  store copy of the original number
6. temp=num
7. # Using **while** loop
8. **while**(num):
9. # intialize with 1
10. i=1
11. # fact variable with 1
12. fact=1
13. rem=num%10
14. **while**(i<=rem):
15. fact=fact\*i   # Find factorial of each number
16. i=i+1
17. sum=sum+fact
18. num=num//10
19. **if**(sum==temp):
20. print("Given number is a strong number")
21. **else**:
22. print("Given number is not a strong number")

**Output:**

Enter a number: 145

Given number is a strong number.

**Explanation:**

In the above code

* We declared a variable integer value **num** for enter a number.
* Defined the sum variable with zero.
* A copy of the num value to the temp variable.
* In the first **while loop**, ensure that the given number is greater than 0.
* Inside the while loop, split the number and assigns the variable to find the factorial of each number.
* In the second while loop (nested while loop), finds the factorial of the each number.

Suppose user enter the value = 145 and sum = 0

**Assigning initial values**

1. i = 0
2. fact = 0
3. temp = num
4. temp = 145

Now understand the loop iteration. First Iteration

1. rem = temp % 10
2. rem = 145 % 10 = 5

Now, we entered in the nested while loop. It calculates the factorial of 5 is 120.

1. sum = sum + 120> 0+120
2. sum = 120
3. temp = temp//10 = 14
4. temp = 14

**Second Iteration**

1. temp = 14,
2. sum = 120
3. rem = 14 % 10 = 4

Now, it enters into nested While loop. Here, it calculates the factorial of 4 = 24.

1. sum = 120 + 24
2. sum = 144
4. temp = 14//10
5. temp = 1

**Third Iteration**

1. temp = 1
2. sum = 144
3. rem = 1 % 10 = 0

The factorial of 1 is 1

1. sum = 144 + 1
2. sum = 145
3. temp = 1 / 10
4. temp = 0

**Here, the** temp = 0 so, the while loop condition fails.

If (num == sum) Now, we check the condition whether the user enter number is exactly equal to sum or not. If this condition returns True, then it is strong Number, else it is not Strong Number.

We have complete the program using the while loop. We can also use [for loop](https://www.javatpoint.com/python-for-loop) to find whether a given number is strong or not.

## Strong Number Using the for loop

We can also find the strong number using for loop. The logic is the same as above program, the while loop is replaced by for loop.

**Example -**

1. # Python Program to find Strong Number
2. num = **int**(input(" Enter the Number:"))
3. sum = 0
4. temp = num
6. **while**(temp > 0):
7. fact = 1
8. rem = temp % 10
10. **for** i in range(1, rem + 1):
11. fact = fact \* i
13. print("Factorial of %d = %d" %(rem, fact))
14. sum = sum + fact
15. temp = temp // 10
17. print("\n Sum of Factorials of a Given Number %d = %d" %(num, sum))
19. **if** (sum == num):
20. print(" The given number is a Strong Number")
21. **else**:
22. print(" The given number is not a Strong Number")

**Output:**

Enter the Number:145

Factorial of 5 = 120

Factorial of 4 = 24

Factorial of 1 = 1

Sum of Factorials of a Given Number 145 = 145

The given number is a Strong Number

## Python Program to find strong number using factorial function

[Python math module](https://www.javatpoint.com/python-math-module) provides the built-in math module. By using this method, we can omit the use of the [nested while loop](https://www.javatpoint.com/python-while-loop).

**Example -**

1. # Python Program to find Strong Number
3. **import** math
4. num = **int**(input(" Enter the Number:"))
5. sum = 0
6. temp = num
8. **while**(temp > 0):
9. rem = temp % 10
10. fact = math.factorial(rem)  # Using the buitlt-in factorial() function
12. print("Factorial of %d = %d" %(rem, fact))
13. sum = sum + fact
14. temp = temp // 10
16. print("\n Sum of Factorials of a Given Number %d = %d" %(num, sum))
18. **if** (sum == num):
19. print(" The given number is a Strong Number")
20. **else**:
21. print(" The given number is not a Strong Number")

**Output:**

Enter the Number: 145

Factorial of 5 = 120

Factorial of 4 = 24

Factorial of 1 = 1

Sum of Factorials of a Given Number 145 = 145

The given number is a Strong Number

**Explanation -**

In the above code,

* We used the **factorial()** function and passed a reminder as an argument.
* In the first iteration of while loop, it returned the reminder 5 and passed to the factorial of 5.
* It will go on until the temp value is greater than zero. We don't need to use another while loop.

# Python Array vs. List

Python array and lists are the important data structure of Python. Both list and array and list are used to store the data in Python. These data structures allow us to indexing, slicing, and iterating. But they have little dissimilarity with each other. In this tutorial, we will learn the essential difference between the Python list and array.

## Introduction

As we know that, Python has extensive data structures such as lists, tuples, sets, and dictionaries that provide many features and functions. The lists are the most effective and easy-to-use data structure in [Python](https://www.javatpoint.com/python-tutorial).

On the other hand, Python doesn't provide built-in support to the array. We need to import the array module to use the **array module** or from the **NumPy package** in the Python program. And that's a primary difference between the array and list. Before diving deep into this topic, let's have a brief introduction to both data structures.

## Python List

A [list](https://www.javatpoint.com/python-lists) is a built-in, linear data structure of Python. It is used to store the data in a sequence manner. We can perform several operations to list, such as indexing, iterating, and slicing. The list comes with the following features.

* The list elements are enclosed with a square bracket, and each element is separated by a comma (,).
* It is a mutable type which means we can modify the list items after their creation.
* The lists are ordered, which means items are stored in a specific order. We can use indexing to access the list element.
* We can store the items of different data types. We can combine strings, integers, and objects in the same list.

Below is an example of a list.

**Example -**

1. list = [31, 60, 19, 12]
2. print(list)
3. print(type(list))

**Output:**

[31, 60, 19, 12]

<class 'list'>

**Example - 2**

1. # creating a list containing elements
2. # belonging to different data types
3. list1 = [1,"Yash",['a','e']]
4. print(list1)

**Output:**

[1, 'Yash', ['a', 'e']]

In the above list, the first element is an integer; the second is a string and third is a list of characters.

## Array in Python

An [array](https://www.javatpoint.com/python-arrays) is also a linear data structure that stores the data. It is also ordered, mutable, and enclosed in square brackets. It can store the non-unique items. But there are restrictions to store the different data type values.

To work with the Array in Python, we need to import either an array module or a Numpy.

1. **import** array as arr
2. or
3. **import** numpy as np

Elements are allocated in contiguous memory location that allows us to easy modification, addition, deletion, accessing of element. Moreover, we need to specify the data type. Let's understand the following example.

**Example -**

1. Import array as arr
2. array\_1 = arr.array("i", [31, 60,19, 12])
3. print(array\_1)
4. print(type(array\_1))

**Output:**

array('i', [31, 60, 19, 12])

<class 'array.array'>

**Example - 2: Using Numpy array**

1. **import** numpy as np
2. array\_sample = np.array(["str", 'sunil', 'sachin', 'megha', 'deepti'])
3. print (array\_sample)
4. print(type(array\_sample))

**Output:**

['numbers' 'sunil' 'sachin' 'megha' 'deepti']

<class 'numpy.ndarray'>

We have specified the string type and stored the string value.

## Difference between Array and List

Now, we have a brief introduction and features. Here, we will discuss the differences between the Array and List.

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **List** | **Array** |
| 1. | The list can store the value of different types. | It can only consist of value of same type. |
| 2. | The list cannot handle the direct arithmetic operations. | It can directly handle arithmetic operations. |
| 3. | We need to import the array before work with the array. | The lists are the build-in data structure so we don't need to import it. |
| 4. | The lists are less compatible than the array to store the data. | An array are much compatible than the list. |
| 5. | It consumes a large memory. | It is a more compact in memory size comparatively list. |
| 6. | It is suitable for storing the longer sequence of the data item. | It is suitable for storing shorter sequence of data items. |
| 7. | We can print the entire list using explicit looping. | We can print the entire list without using explicit looping. |
| 8. | It can be nested to contain different types of elements. | It must contain either all nested elements of same size. |

## Conclusion

We have discussed the differences between array and list. Both data types are essential in Python and both have some limitations. Python lists are easy to use in python where arrays are typically used for data analysis.

# What is duck typing in Python?

In this tutorial, we will learn about duck typing. It is a popular term in Python, and it comes from saying, **"If it walks like duck, swims like duck, looks like a duck, then it probably should be a duck."**

The above statement gives an idea to identify a duck. Here we don't need to have a genomic sequence of the duck. We draw our conclusion by its behavior and external appearances.

We will discuss what is exactly mean of duck typing in Python programming.

Python follows the **EAFP** (Easier to Ask Forgiveness than Permission) rather than the **LBLY** (Look Before You Leap) philosophy. The EAFP is somewhat linked to the "duck typing" style.

**Dynamic vs. Static Typing**

The main reason for using duck typing is to provide support for dynamic typing in [Python programming](https://www.javatpoint.com/python-tutorial). In Python, we don't need to specify the variable's data type and we can reassign the different data type values to same variable in further code. Let's see the following example.

**Example -**

1. x = 12000
2. print(type(x))
4. x = 'Dynamic Typing'
5. print(type(x))
7. x = [1, 2, 3, 4]
8. print(type(x))

**Output:**

<class 'int'>

<class 'str'>

<class 'list'>

As we can see in the above code, we assigned an integer to a variable x, making it of the **int** type. Then, we assigned a string and a list to the same variable. Python interpreter accepts the changes of data types of the same variable. This is a dynamic typing behavior.

Many other programming languages such as Java, swift are the static type. We need to declare variable with the data types. In the below example, we try to do the same thing using the Swift instead of Python.

**Example -**

1. # integer value assigning in JavaScript
2. var a = 10
4. # Assinging string in swift
5. a = 'Swift language'

Above code cannot be compiled, because we couldn't assign a string in Swift language. Because variable **a** was declared as an integer.

## Concept of Duck Typing

Earlier, we have discussed that Python is a dynamic typed language. However, we can use the dynamic approach with custom data types. Let's understand the following example.

**Example -**

1. **class** VisualStudio:
2. def execute(self):
3. print('Compiling')
4. print('Running')
5. print('Spell Check')
6. print('Convention Check')
8. **class** Desktop:
9. def code(self, ide):
10. ide.execute()

13. ide  = VisualStudio()
14. desk = Desktop()
15. desk.code(ide)

**Output:**

Compiling

Running

Spell Check

Convention Check

In the above code, we have created a **VisualStudio** class that has to **execute()** method. In the desktop-class, we have passed the ide as an argument in the code(). An **ide** is an object of **VisualStudio** class. With the help of ide, we called the **execute()** method of VisualStudio class.

Let's see another example.

**Example - 2**

1. **class** Duck:
2. def swim(self):
3. print("I'm a duck, and I can swim.")
5. **class** Sparrow:
6. def swim(self):
7. print("I'm a sparrow, and I can swim.")
9. **class** Crocodile:
10. def swim\_walk(self):
11. print("I'm a Crocodile, and I can swim, but not quack.")
13. def duck\_testing(animal):
14. animal.swim()

17. duck\_testing(Duck())
18. duck\_testing(Sparrow())
19. duck\_testing(Crocodile())

**Output:**

I'm a duck, and I can swim.

I'm a sparrow, and I can swim.

Traceback (most recent call last):

File "<string>", line 24, in <module>

File "<string>", line 19, in duck\_testing

AttributeError: 'Crocodile' object has no attribute 'swim'

In the above code, the Duck class's instance is reflected by calling **the duck\_testing** function. It also happens with the Sparrow class, which implements the **swim()** function. But in the case of the Crocodile class, it fails the duck testing evaluation because it doesn't implement the **swim()** function.

## How duck typing supports EAFP

The duck typing is the most appropriate style for the EAFP because we don't need to focus on the **"type"** of the object. We only need to take care of its **behavior** and **capability**. Let's see the following statements.

When we see a lot of if-else blocks, then it is an LBYL coding style.

But if we see a lot of try-except blocks, then it is a probability an EAFP coder.

# PEP 8 in Python | what is the purpose of PEP 8 in Python?

In this tutorial, we will learn what PEP-8 is and how we can use it in Python coding. We will discuss the guidelines for using PEP in programming-this tutorial is aimed at beginners to intermediate. We will also discuss the benefits of using PEP-8 while coding.

## What is PEP?

The PEP is an abbreviation form of **Python Enterprise Proposal**. Writing code with proper logic is a key factor of programming, but many other important factors can affect the code's quality. The developer's coding style makes the code much reliable, and every developer should keep in mind that Python strictly follows the way of order and format of the string.

Adaptive a nice coding style makes the code more readable. The code becomes easy for end-user.

PEP 8 is a document that provides various guidelines to write the readable in Python. PEP 8 describes how the developer can write beautiful code. It was officially written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan. The main aim of PEP is to enhance the readability and consistency of code.

## Why PEP 8 is Important?

PEP 8 enhances the readability of the [Python](https://www.javatpoint.com/python-tutorial) code, but why is readability so important? Let's understand this concept.

Creator of Python, Guido van Rossum said, **"Code is much more often than it is written."** The code can be written in a few minutes, a few hours, or a whole day but once we have written the code, we will never rewrite it again. But sometimes, we need to read the code again and again.

At this point, we must have an idea of why we wrote the particular line in the code. The code should reflect the meaning of each line. That's why readability is so much important.

We will describe few important guidelines for writing effective code that can be read by others as well.

## Naming Convention

When we write the code, we need to assign name to many things such as variables, functions, classes, packages, and a lot more things. Selecting a proper name will save time and energy. When we look back to the file after sometime, we can easily recall what a certain variable, function, or class represents. Developers should avoid choosing inappropriate names.

The naming convention in Python is slightly messy, but there are certain conventions that we can follow easily. Let's see the following naming convention.

**Example -**

Single lowercase letter

1. a = 10

Single upper case letter

1. A = 10

Lowercase

1. var = 10

Lower\_case\_with\_underscores

1. number\_of\_apple = 5

UPPERCASE

1. VAR = 6

UPPER\_CASE\_WITH\_UNDERSCORES

1. NUM\_OF\_CARS =20

CapitalizedWords (or CamelCase)

1. NumberOfBooks = 100

#### Note: While using abbreviations in CapWords, capitalize all the letters of the abbreviation. Thus HTTPServerError is better than HttpServerError.

## Name Style

Below is the table that specifies some of the common naming styles in Python. Consider the following table.

|  |  |  |
| --- | --- | --- |
| **Type** | **Naming Convention** | **Examples** |
| Function | We should use the lowercase words or separates words by the underscore. | myfunction, my\_function |
| Variable | We should use a lowercase letter, words, or separate words to enhance the readability. | a, var, variable\_name |
| Class | The first letter of class name should be capitalized; use camel case. Do not separate words with the underscore. | MyClass, Form, Model |
| Method | We should use a lowercase letter, words, or separate words to enhance readability. | class\_method, method |
| Constant | We should use a short, uppercase letter, words, or separate words to enhance the readability. | MYCONSTANT, CONSTANT, MY\_CONSTANT |
| Module | We should use a lowercase letter, words, or separate words to enhance the readability. | Module\_name.py, module.py |
| Package | We should use a lowercase letter, words, or separate words to enhance the readability. Do not separate words with the underscore. | package, mypackage, |

Above are some common naming conventions that are useful to beautify the Python code. For additional improvement, we should choose the name carefully.

## Code Layout

The code layout defines how much the code is readable. In this section, we will learn how to use whitespace to improve code readability.

### Indentation

Unlike other programming languages, the indentation is used to define the code block in Python. The indentations are the important part of the Python programming language and it determines the level of lines of code. Generally, we use the 4 space for indentation. Let's understand the following example.

**Example -**

1. x = 5
2. **if** x ==  5:
3. print('x is larger than 5')

In the above example, the indented print statement will get executed if the condition of **if statement** is true. This indentation defines the code block and tells us what statements execute when a function is called or condition trigger.

* **Tabs vs. Space**

We can also use the **tabs** to provide the consecutive spaces to indicate the indentation, but whitespaces are the most preferable. Python 2 allows the mixing of tabs and spaces but we will get an error in Python 3.

### Indentation following Line Break

It is essential to use indentation when using line continuations to keep the line to fewer than 79 characters. It provides the flexibility to determining between two lines of code and a single line of code that extends two lines. Let's understand the following example.

**Example -**

1. # Correct Way:
3. # Aligned with opening delimiter.
4. obj = func\_name(argument\_one, argument\_two,
5. argument\_three, argument\_four

We can use the following structure.

1. # first line doesn't has any argument
2. # We add 4 spaces from the second line to discriminate arguments from the rest.
3. def function\_name(
4. argument\_one, argument\_two, argument\_three,
5. argument\_four):
6. print(argument\_two)
8. # 4 space indentation to add a level.
9. foo = long\_function\_name(
10. var\_one, var\_two,
11. var\_three, var\_four)

### Use docstring

Python provides the two types of **document strings or docstring** - single line and multiple lines. We use the triple quotes to define a single line or multiline quotes. Basically, these are used to describe the function or particular program. Let's understand the following example.

**Example -**

1. def add(a, b):
2. """This is simple add method"""
4. """This is
5. a
6. simple add program to add
7. the two numbers. """

### Should a Line Break Before or After a Binary Operator?

The lines break before or after a binary operation is a traditional approach. But it affects the readability extensively because the operators are scattered across the different screens, and each operator is kept away from its operand and onto the previous line. Let's understand the following example.

**Example -**

1. # Wrong:
2. # operators sit far away from their operands
3. marks = (engilsh\_marks +
4. math\_marks +
5. (science\_marks - biology\_marks) +
6. Physics\_marks

As we can see in the above example, it seems quite messy to read. We can solve such types of problems by using the following structure.

**Example -**

1. # Correct:
2. # easy to match operators with operands
3. Total\_marks = (English\_marks
4. + math\_marks
5. + (science\_marks - biology\_marks)
6. + physics\_marks

Python allows us to break line before or after a binary operator, as long as the convention is consistent locally.

### Importing module

We should import the modules in the separates line as follows.

1. **import** pygame
2. **import** os
3. **import** sys

Wrong

1. **import** sys, os

We can also use the following approach.

1. from subprocess **import** Popen, PIPE

The import statement should be written at the top of the file or just after any module comment. Absolute imports are the recommended because they are more readable and tend to be better behaved.

1. **import** mypkg.sibling
2. from mypkg **import** sibling
3. from mypkg.sibling **import** example

However, we can use the explicit relative imports instead of absolutes imports, especially dealing with complex packages.

### Blank Lines

Blank lines can be improved the readability of Python code. If many lines of code bunched together the code will become harder to read. We can remove this by using the many blank vertical lines, and the reader might need to scroll more than necessary. Follow the below instructions to add vertical whitespace.

* **Top-level function and classes with two lines -** Put the extra vertical space around them so that it can be understandable.

1. **class** FirstClass:
2. pass

5. **class** SecondClass:
6. pass

9. def main\_function():
11. **return** None

* **Single blank line inside classes -** The functions that we define in the class is related to one another. Let's see the following example -

1. **class** FirstClass:
2. def method\_one(self):
3. **return** None
5. def second\_two(self):
6. **return** None

* **Use blank lines inside the function -** Sometimes, we need to write a complicated function has consists of several steps before the return statement. So we can add the blank line between each step. Let's understand the following example.

1. def cal\_variance(n\_list):
2. list\_sum = 0
3. **for** n in n\_list:
4. list\_sum = list\_sum + n
5. mean = list\_sum / len(n\_list)
7. square\_sum = 0
8. **for** n in n\_list:
9. square\_sum = square\_sum + n\*\*2
10. mean\_squares = square\_sum / len(n\_list)
12. **return** mean\_squares - mean\*\*2

The above way can remove the whitespaces to improve the readability of code.

## Put the Closing Braces

We can break lines inside parentheses, brackets using the Line continuations. PEP 8 allows us to use closing braces in implies line continuations. Let's understand the following example.

* Line up the closing brace with the first non-whitespace.

1. list\_numbers = [
2. 5, 4, 1,
3. 4, 6, 3,
4. 7, 8, 9
5. ]

* Line up the closing braces with the first character of line.

1. list\_numbers = [
2. 5, 4, 1,
3. 4, 6, 3,
4. 7, 8, 9
5. ]

Both methods are suitable to use, but consistency is key, so choose any one and continue with it.

## Comments

Comments are the integral part of the any programming language. These are the best way to explain the code. When we documented our code with the proper comments anyone can able to understand the code. But we should remember the following points.

* Start with the capital latter, and write complete sentence.
* Update the comment in case of a change in code.
* Limit the line length of comments and docstrings to 72 characters.

### Block Comment

Block comments are the good choice for the small section of code. Such comments are useful when we write several line codes to perform a single action such as iterating a loop. They help us to understand the purpose of the code.

PEP 8 provides the following rules to write comment block.

* Indent block comment should be at the same level.
* Start each line with the # followed by a single space.
* Separate line using the single #.

Let's see the following code.

1. **for** i in range(0, 5):
2. # Loop will iterate over i five times and print out the value of i
3. # **new** line character
4. print(i, '\n')

We can use more than paragraph for the technical code. Let's understand the following example.

### Inline Comments

Inline comments are used to explain the single statement in a piece of code. We can quickly get the idea of why we wrote that particular line of code. PEP 8 specifies the following rules for the inline comments.

* Start comments with the # and single space.
* Use inline comments carefully.
* We should separate the inline comments on the same line as the statement they refer.

Following is the example of inline comments.

1. a = 10    # The a is variable that holds integer value.

Sometimes, we can use the naming convention to replace the inline comment.

1. x = 'Peter Decosta' #This is a student name

We can use the following naming convention.

1. Student\_name = 'Peter Decosta'

Inline comments are essential but block comments make the code more readable.

## Avoid Unnecessary Adding Whitespaces

In some cases, use of whitespaces can make the code much harder to read. Too much whitespaces can make code overly sparse and difficult to understand. We should avoid adding whitespaces at the end of a line. This is known as trailing whitespaces.

Let's see the following example.

Example - 1

1. # Recommended
2. list1 = [1, 2, 3]
4. # Not recommended
5. List1 = [ 1, 2, 3, ]

**Example - 3**

1. x = 5
2. y = 6
4. # Recommended
5. print(x, y)
7. # Not recommended
8. print(x , y)

## Programming Recommendation

As we know that, there are several methods to perform similar tasks in Python. In this section, we will see some of the suggestions of PEP 8 to improve the consistency.

### Avoid comparing Boolean values using the equivalence operator

1. # Not recommended
2. bool\_value = 10 > 5
3. **if** bool\_value == True:
4. **return** '10 is bigger than 5'

We shouldn't use the equivalence operator == to compare the Boolean values. It can only take the True or False. Let's see following example.

1. # Recommended
2. **if** my\_bool:
3. **return** '10 is bigger than 5'

This approach is simple that's why PEP 8 encourages it.

### Empty sequences are false in if statements

If we want to check whether a given list is empty, we might need to check the length of list, so we need to avoid the following approach.

1. # Not recommended
2. list1 = []
3. **if** not len(list1):
4. print('List is empty!')

However, if there is any empty list, set, or tuple. We can use the following way to check.

1. # Recommended
2. list1 = []
3. **if** not list1:
4. print('List is empty!')

The second method is more appropriate; that's why PEP 8 encourages it.

### Don't use not is in if statement

There are two options to check whether a variable has a defined value. The first option is with x is not None, as in the following example.

1. # Recommended
2. **if** x is not None:
3. **return** 'x exists!'

A second option is to evaluate **x is None** and if statement based on not the outcome.

1. # Not recommended
2. **if** not x is None:
3. **return** 'x exists!'

Both options are correct but the first one is simple, so PEP 8 encourages it.

## Conclusion

We have discussed the PEP 8 guidelines to make the code remove ambiguity and enhance readability. These guidelines improve the code, especially when sharing the code with potential employees or collaborators. We have discussed what PEP is and why it uses, how to write code that is PEP 8 compliant. Moreover, we have a brief introduction to the naming conventions. If you want more information regarding the PEP 8, you can read the full documentation or visit [PEP8.org](http://www.pep8.org/).

# F String in Python

In this tutorial, we will learn about the string formatting mechanism. The f-string is the best way to format the string. The string's formatting style makes the string more readable, more concise, and less prone to error. It is also faster than the other.

Before diving deep into this topic, we will first discuss the techniques were used before the f string.

## Old String Formatting in Python

There are two main ways for formatting the string: % - formatting and **str.format().**But both methods have some limitations. Let's have a brief introduction of these given methods.

### 1.% - formatting

It is a traditional way of formatting the string, but the docs recommend this method, which contains the following statement.

**"The formatting operations described here exhibit a variety of quirks that lead to several common errors (such as failing to display tuples and dictionaries correctly).**

Using the newer formatted string literals or the **str.format()**interface helps avoid these errors." - **Official Documentation**.

### How to use % - formatting

We can perform built-in operation using the %-operator. Let's understand the following example.

1. student = "Michel"
2. "Hello, %s." % student

**Output:**

Hello Michel

We can also use the tuple to store the multiple variables. Let's see the following example.

1. name = "Sharma"
2. age = 24
3. "Hello, %s. You are %s." % (name, age)

**Output:**

Hello Sharma You are 24.

### Why %-formatting is not recommended

Because as the variable is increased and a longer string, our code becomes much less easily readable. The code looks messy. Let's see the following example.

**Example -**

1. first\_name = "Steve"
2. last\_name = "Rogers"
3. age = 70
4. profession = "Superhero"
5. group = "Marvel?
6. print(""Hello, %s %s. Your age is %s. You are a %s. You were a member of %s." %(first\_name, last\_name, age, profession)")

**Output:**

Hello, Steve Rogers. Your age is 70. You are a Superhero. You were a member of Marvel.

As we can see in the above code, the code became hard to read and more prone to the error. That's why this way of formatting string isn't good.

### 2.str.format() method

It is another popular way to format the string introduced in Python 2.6. Let's see how we can use it.

### How to Use str.format()

It is an improvement on %-formatting. It is like a normal built-in function called on the object and that object being converted to a string.

Let's see the following example.

**Example -**

1. first\_name = "Mathew"
2. last\_name = "Zukerburg"
3. age = 45
4. print("Hello, {}. You are {}.".format(name, age)")

To access the dictionary element inside the format method, we can pass the key in the format() method.

1. person = {'name': 'Peter', 'age': 17}
2. "Hello, {name}. You are {age}.".format(name=person['name'], age=person['age'])

**Output:**

Hello Peter, You are 17

### Why str.format() method is not recommended?

The str.format() is much efficient than the %-format method but it can be still be quite verbose when we deal with the multiple parameters.

## F-string Method

It is a new string formatting mechanism introduced by the PEP 498. It is also known as Literal String Interpolation or more commonly as F-strings **(f character preceding the string literal).**The primary focus of this mechanism is to make the interpolation easier.

When we prefix the string with the letter 'F, the string becomes the f-string itself. The f-string can be formatted in much same as the **str.format()**method. The F-string offers a convenient way to embed Python expression inside string literals for formatting.

**Example -**

1. # Python3 program introducing f-string
2. val = 'Geeks'
3. print(f"{val}for{val} is a portal for {val}.")
4. name = 'Tushar'
5. age = 23
6. print(f"Hello, My name is {name} and I'm {age} years old.")

**Output:**

Hello, My name is Tushar and I'm 28 years old.

In the above code, we have used the f-string to format the string. It evaluates at runtime; we can put all valid Python expressions in them.

We can use it in a single statement.

1. print(f"{2 \* 30})"

**Output:**

60

However, we could use in function.

1. def upercase(input):
2. **return** input.uper()
4. name = "Sachin Tendulkar"
5. f"{upercase(name)} is great."

**Output:**

Sachin Tendulkar is great

The f-string could also be used with the class object. Let's understand the following example.

**Example -**

1. **class** Actor:
2. def \_\_init\_\_(self, first\_name, last\_name, movie):
3. self.first\_name = first\_name
4. self.last\_name = last\_name
5. self.movie = movie
7. def \_\_str\_\_(self):
8. **return** f"{self.first\_name} {self.last\_name}'s superhit movie is {self.movie}."
10. def \_\_repr\_\_(self):
11. **return** f"{self.first\_name} {self.last\_name}  {self.movie}. Superhi!"
13. ac = Actor('Keenu', 'Reevs', 'Matrix')
14. print(f"{ac}")

**Output:**

Keenu Reevs's superhit movie is Matrix.

**Explanation -**

In the above code, we have used the \_\_str\_\_() and \_\_repr\_\_(), representing an object as a string. So we need to include at least one of those methods in the class definition. The f-string will use the \_\_str\_\_() method; we can also use the \_\_repr\_\_() by including the conversion flag ! r.

1. print(f"{ac}")
2. print(f"{ac}"!r)

**Output:**

Keenu Reevs's superhit movie is Matrix.

Keenu Reevs Matrix Superhit

## F-string in Dictionary

We have to take care while working with dictionary keys inside the f-string. There is a different quotation to use dictionary keys and f-string. Let's understand the following example.

**Example -**

1. detail = {"name": "John", "age": 19}
2. print(f"{detail['name']} is {detail['age']} years old.")

**Output:**

John is 19 years old.

Below method is not allowed in case of dictionary.

**Example -**

1. detail = {"name": "John", "age": 19}
2. print(f'{detail['name']} is {detail['age']} years old.')

**Output:**

File "", line 2

print(f'{detail['name']} is {detail['age']} years old.')

^

SyntaxError: invalid syntax

As we can see in the above code, we change a double quotation to single quotes, and it has thrown an error.

## Speed

The reason for adapting this formatting style is its speed. The f-string evaluates at runtime rather than constant values. It embeds expression inside string literals, using minimal syntax. It is fast because it evaluates at runtime, not a constant value.

Let's see the following string comparison.

**Example - 1:**

1. **import** timeit
2. print(timeit.timeit("""name = "Sachin"
3. age = 74
4. '%s is %s.' % (name, age)""", number = 10000))

**Output:**

0.0022497819736599922

**Example - 2:**

1. **import** timeit
2. print(timeit.timeit("""name = "Mathew"
3. age = 40
4. '{} is {}.'.format(name, age)""", number = 10000))

**Output:**

0.0025783719611354172

**Example - 3:**

1. **import** timeit
2. print(timeit.timeit("""name = "Rockey"
3. age = 74
4. f'{name} is {age}.'""", number = 10000))

**Output:**

0.0019360429723747075

As we can observe, the f-string is on the top of list.

## Braces

To make appear braces in the code, you should use the double quotes as follows. Let's understand the following example.

**Example -**

1. f"{{70 + 40}}"

**Output:**

{70 + 40}

If we use the triple braces, it will display single braces in our string. Let's understand the following example.

**Example -**

1. f"{{{90 + 4}}}"

**Output:**

{94}

We can display the more braces if we use more than triple braces.

**Example -**

1. f"{{{{70 + 4}}}}"

**Output:**

{{70 + 4}}

## Backslashes

We can use the backslash escapes in the string portion of an f-string. However, we can't use backslashes to escape in the expression part of an f-string. Let's understand the following example.

**Example -**

1. f"{\"Medric Pacalo\"}"
2. File "<stdin>", line 1
3. f"{\"Medric Pacalo\"}"

**Output:**

SyntaxError: f-string expression part cannot include a backslash

## Inline comments

We cannot include the # symbol in the expression. It will throw a syntax error. Let's understand the following example.

**Example -**

1. f"Tony is {10 \* 73 #Hate this!}."
2. f"Tonyr is {2 \* 37 #Hate this!}."

**Output:**

SyntaxError: f-string expression part cannot include '#'

### Conclusion

We can use any one method out of three, but the f-string method provides a more concise, readable, and convenient way. It is faster and less prone to error. We have explained almost every possible scenario off-string and why one should consider this approach in programming.

# Bubble Sort in Python

A Bubble sort is an easy algorithm among the various sorting algorithms. We learn it as a first sorting algorithm. It is easy to learn and highly intuitive. It can be easy to implement into the code, which is much beneficial for beginner software developers. But it is the worst algorithm for sorting the elements in every except because it checks every time the array is sorted or not.

Let's understand the concepts of the bubble sort.

## Concept of Bubble Sort

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

Let's understand it by an example -

Play Videox[[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack" \t "_blank)

**Example -**

We are creating a list of element, which stores the integer numbers

list1 = [5, 3, 8, 6, 7, 2]

Here the algorithm sort the elements -

**First iteration**

[**5, 3**, 8, 6, 7, 2]

It compares the first two elements and here 5>3 then swap with each other. Now we get new list is -

[**3, 5,** 8, 6, 7, 2]

In second comparison, 5 < 8 then swapping happen -

[3, **5, 8,** 6, 7, 2]

In third comparison, 8>6 then swap -

[3, 5, **6, 8,** 7, 2]

In fourth comparison, 8>7 then swap -

[3, 5, 6, **7, 8**, 2]

In fifth comparison, 8>2 then swap-

[3, 5, 6, 7, **2, 8**]

Here the first iteration is complete and we get the largest element at the end. Now we need to the len(list1) - 1

**Second Iteration**

[**3, 5**, 6, 7, 2, 8] - > [**3, 5**, 6, 7, 2, 8] here, 3<5 then no swap taken place

[3, **5, 6,** 7, 2, 8] - > [3, **5, 6,** 7, 2, 8] here, 5<6 then no swap taken place

[3, 5, **6, 7**, 2, 8] - > [3, 5, **6, 7**, 2, 8] here, 6<7 then no swap taken place

[3, 5, 6, **7, 2**, 8] - > [3, 5, 6, **2, 7**, 8] here 7>2 then swap their position. Now

[3, 5, 6, **2, 7**, 8] - > [3, 5, 6, 2, **7, 8**] here 7<8 then no swap taken place.

**Third Iteration**

[**3, 5**, 6, 2, 7, 8] - > [**3, 5**, 6, 7, 2, 8] here, 3<5 then no swap taken place

[3, **5, 6,** 2, 7, 8] - > [3, **5, 6,** 7, 2, 8] here, 5<6 then no swap taken place

[3, 5, **6, 2**, 7, 8] - > [3, 5, **2, 6**, 7, 8] here, 6<2 then swap their positions

[3, 5, 2, **6, 7**, 8] - > [3, 5, 2, **6, 7**, 8] here 6<7 then no swap taken place. Now

[3, 5, 2, 6, **7, 8**] - > [3, 5, 2, 6, **7, 8**] here 7<8 then swap their position.

It will iterate until the list is sorted.

**Fourth Iteration -**

[**3, 5**, 2, 6, 7, 8] - > [**3, 5,** 2, 6, 7, 8]

[3**, 5, 2**, 6, 7, 8] - > [3, **2, 5**, 6, 7, 8]

[3, 2, **5, 6**, 7, 8] - > [3, 2, **5, 6**, 7, 8]

[3, 2, 5, **6, 7**, 8] - > [3, 2, 5, **6, 7**, 8]

[3, 2, 5, **6, 7**, 8] - > [3, 2, 5, **6, 7**, 8]

**Fifth Iteration**

[**3, 2,** 5, 6, 7, 8] - > [**2, 3,** 5, 6, 7, 8]

Check the each element and as we can see that our list is sorted using the bubble sort technique.

## Implementation in Python Code

We have already described the technique of bubble sort. Now, we will implement the logic in the [Python](https://www.javatpoint.com/python-tutorial)

code.

**Program**

1. # Creating a bubble sort function
2. def bubble\_sort(list1):
3. # Outer loop for traverse the entire list
4. for i in range(0,len(list1)-1):
5. for j in range(len(list1)-1):
6. if(list1[j]**>**list1[j+1]):
7. temp = list1[j]
8. list1[j] = list1[j+1]
9. list1[j+1] = temp
10. return list1
12. list1 = [5, 3, 8, 6, 7, 2]
13. print("The unsorted list is: ", list1)
14. # Calling the bubble sort function
15. print("The sorted list is: ", bubble\_sort(list1))

**Output:**

The unsorted list is: [5, 3, 8, 6, 7, 2]

The sorted list is: [2, 3, 5, 6, 7, 8]

**Explanation:**

In the above code, we have defined a **bubble\_sort()** function which takes **list1** as an argument.

* Inside the function, we have defined two for loop - first for loop iterates the complete list and the second for loop iterates the list and the compare the two elements in every outer loop iterations.
* The for loop will be terminated when it reaches at the end.
* We have defined the condition in the inner for loop; if a first index value is greater than the second index value, swap their positions with each other.
* We called the function and passed a list; it iterated and returned the sorted list.

### Without using a temp variable

We can also swap the elements without using the temp variable. Python has a very unique syntax. We can use the following lines of code.

**Example -**

1. def bubble\_sort(list1):
2. # Outer loop for traverse the entire list
3. for i in range(0,len(list1)-1):
4. for j in range(len(list1)-1):
5. if(list1[j]**>**list1[j+1]):
6. # here we are not using temp variable
7. list1[j],list1[j+1] = list1[j+1], list1[j]
8. return list1
10. list1 = [5, 3, 8, 6, 7, 2]
11. print("The unsorted list is: ", list1)
12. # Calling the bubble sort function
13. print("The sorted list is: ", bubble\_sort(list1))

**Output:**

The unsorted list is: [5, 3, 8, 6, 7, 2]

The sorted list is: [2, 3, 5, 6, 7, 8]

## Optimization of Python Code Implementation

We can optimize the above code using the two techniques. The swaps are not done; it means list is sorted. In the previous technique - The previous technique will evaluate the complete list though it doesn't seem necessary to do.

We can prevent the unnecessary evaluation using the **Boolean** flag and checks if any swaps were made in the previous section.

**Example -**

1. def bubble\_sort(list1):
2. # We can stop the iteration once the swap has done
3. has\_swapped = True
5. while(has\_swapped):
6. has\_swapped = False
7. for i in range(len(list1) - 1):
8. if list1[i] **>** list1[i+1]:
9. # Swap
10. list1[i], list1[i+1] = list1[i+1], list1[i]
11. has\_swapped = True
12. return list1

15. list1 = [5, 3, 8, 6, 7, 2]
16. print("The unsorted list is: ", list1)
17. # Calling the bubble sort function
18. print("The sorted list is: ", bubble\_sort(list1))

**Output:**

The unsorted list is: [5, 3, 8, 6, 7, 2]

The sorted list is: [2, 3, 5, 6, 7, 8]

In the second technique, we consider the fact that the iteration is ended when the largest element of the list end up at the end of the list.

The first time, we pass the largest element at the end position using the n position. The second time, we pass through the n-1 position, the second largest element.

In each consecutive iteration, we can compare at one less element than before. More accurately, in the k-th iteration, only need to compare at the first **n - k + 1** elements:

**Example -**

1. def bubble\_sort(list1):
2. has\_swapped = True
4. total\_iteration = 0
6. while(has\_swapped):
7. has\_swapped = False
8. for i in range(len(list1) - total\_iteration - 1):
9. if list1[i] **>** list1[i+1]:
10. # Swap
11. list1[i], list1[i+1] = list1[i+1], list1[i]
12. has\_swapped = True
13. total\_iteration += 1
14. print("The number of iteraton: ",total\_iteration)
15. return list1
17. list1 = [5, 3, 8, 6, 7, 2]
18. print("The unsorted list is: ", list1)
19. # Calling the bubble sort funtion
20. print("The sorted list is: ", bubble\_sort(list1))

**Output:**

The unsorted list is: [5, 3, 8, 6, 7, 2]

The number of iteraton: 6

The sorted list is: [2, 3, 5, 6, 7, 8]

## Time Comparison

Let's see the time comparison between the above code snippets.

1. Unoptimized Bubble Sort Takes: 0.0106407
2. Optimize Bubble Sort Takes: 0.0078251
3. Bubble Sort with a Boolean flag and shortened list Takes: 0.0075207

All techniques are useful for the fewer elements, but if the list consists of many elements, then the second optimize technique make a huge difference.

# Logging in Python

In this tutorial, we will learn the fundamentals of the standard logging module.

## What is logging?

Logging is a [Python](https://www.javatpoint.com/python-tutorial) module in the standard library that provides the facility to work with the framework for releasing log messages from the [Python programs](https://www.javatpoint.com/python-programs). Logging is used to tracking events that occur when the software runs.

This module is widely used by the developers when they work to logging. It is very important tool which used in software development, running, and debugging.

Logging is beneficial to store the logging records. Suppose there is no logging record, and the program is interrupted during its execution, we will be unable to find the actual cause of the problem.

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

Somehow, we detect the cause of the crash but it will consume a lot of time to resolve this. Using the logging, we can leave a trace of breadcrumbs so that if the problem happens in the program, we can find the cause of the problem easily.

We can face many problems while running applications such as we suppose an integer, and we have been given a float, the service is under maintenance and many more. These problems are hard to determine and time-consuming.

## How Logging Works

The logging is a powerful module used by the beginners as well as enterprises. This module provides a proficiency to organize different control handlers and a transfer log messages to these handlers.

To releasing a log message, we need to import the logging module as follows.

1. import logging

Now, we will call the logger to log messages that we want to see. The logging module offers the five levels that specify the severity of events. Each event contains the parallel methods that can be used to log events at the level of severity. Let's understand the following events and their working.

1. **DEBUG -** It is used to provide detailed information and only use it when there is diagnosing problems.
2. **INFO -** It provides the information regarding that things are working as we want.
3. **WARNING -** It is used to warn that something happened unexpectedly, or we will face the problem in the upcoming time.
4. **ERROR -** It is used to inform when we are in some serious trouble, the software hasn't executed some programs.
5. **CRITICAL -** It specifies the serious error, the program itself may be incapable of remaining executing.

The above levels are sufficient to handle any types of problems. These corresponding numerical values of the levels are given below.

|  |  |
| --- | --- |
| **Level** | **Numeric Values** |
| NOTSET | 0 |
| DEBUG | 10 |
| INFO | 20 |
| WARNING | 30 |
| ERROR | 40 |
| CRITICAL | 50 |

The logging module offers many features. It consists of several constants, classes, and methods. The constants are represented by the all caps latter; the classes are represented by capital letters. The items with lowercase represent methods.

Let's have a look at the several logger objects offered by the module itself.

* **Logger.info(msg) :** It is used to log a message with level INFO on this logger.
* **Logger.warning(msg) :** It is used to log a message with level WARNING on this logger.
* **Logger.error(msg) :** It is used to log a message with level ERROR on this logger.
* **Logger.critical(msg) :** It is used to log a message with level CRITICAL on this logger.
* **Logger.log(lvl,msg) :** It is used to logs a message with integer level lvl on this logger.
* **Logger.exception(msg) :** It is used to log a message with level ERROR on this logger.
* **Logger.setLevel(lvl) :** It is used to sets the beginning of this logger to lvl. It will ignore all the messages which are written below.
* **Logger.addFilter(filt) :** It is used to add a specific filter filt to the to this logger.
* **Logger.removeFilter(filt) :** It is used to eliminates a specific filter filt to the to this logger.
* **Logger.filter(record) :** It put on the filter of logger to the record. If the record available and to be handled then returns True. Otherwise, it will return False.
* **Logger.addHandler(hdlr) :** It is used to add a particular handler hdlr to the to this logger.
* **Logger.removeHandler(hdlr) :** It is used to eliminate a particular handler hdlr to this logger.
* **Logger.hasHandlers() :** It is used to verify if the logger contains any handler configured or not.

Let's understand the following example.

**Example -**

1. import logging
3. logging.debug('The debug message is displaying')
4. logging.info('The info message is displaying')
5. logging.warning('The warning message is displaying')
6. logging.error('The error message is displaying')
7. logging.critical('The critical message is displaying')

**Output:**

WARNING:root:The warning message is displaying

ERROR:root:The error message is displaying

CRITICAL:root:The critical message is displaying

**Explanation:**

As we can see in the above output, each message is displayed along with the root, which is the logging module name given to its default logger. The message and the level name are separated by a colon (:) and print the messages in default output format.

We can notice that the **debug()** and **info()** message didn't display messages because, by default, the log module logs the messages with a severity level of **WARNING, ERROR and CRITICAL**.

## Basic Configurations

The main task of logging is to store the records events in a file. The logging module provides the **basicConfig(\*\*kwarg)**, used to configure the logging.

It accepts some of the commonly used argument as follows.

* **level -** The specified severity level is set by the root level.
* **filename -** It specifies a file.
* **filemode -** It opens a file in a specific mode. The default mode of the opening file is a, which means we can append the content.
* **format -** The format defines the format of the log message.

We can set the level of log messages by using the level parameter as we want to record. We need to pass the one constant in the class, which would permit all logging calls.

Let's understand the following example.

**Example -**

1. import logging
3. logging.basicConfig(level=logging.DEBUG)
4. logging.debug('The dubug message is logged')

**Output:**

1. DEBUG:root: The debug will be get logged

Similarly, we can log the message to a file instead of display on console, **filename** and **filemode** can be used in the **basicConfig()** function, and we can decide the format of the message using format attributes. Let's understand the following example.

**Example -**

1. import logging
3. logging.basicConfig(filename='msg.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')
4. logging.warning('This will get logged to a file')

**Output:**

1. root - WARNING - This will get logged to a file

**Explanation:**

The above output will be displayed in the **msg.log** file instead of console. We opened the file in **w**, which means the file is opened in the "write mode". If the **basicConfig()** is called multiple times, then each run of the program will rewrite the log file's output. The **basicConfig()** function can be modified by passing the additional arguments ([https://docs.python.org/3/library/logging.html#logging.basicConfig).](https://docs.python.org/3/library/logging.html#logging.basicConfig)

Let's understand the following example.

**Example -**

1. import logging
3. #Create and configure logger using the basicConfig() function
4. logging.basicConfig(filename="newfile.log",
5. format='%(asctime)s %(message)s',
6. filemode='w')
8. #Creating an object of the logging
9. logger=logging.getLogger()
11. #Setting the threshold of logger to DEBUG
12. logger.setLevel(logging.DEBUG)
14. #Test messages
15. logger.debug("This is a harmless debug Message")
16. logger.info("This is just an information")
17. logger.warning("It is a Warning. Please make changes")
18. logger.error("You are trying to divide by zero")
19. logger.critical("Internet is down")

**Output:**

2020-09-05 13:17:39,204 This is a harmless debug Message

2020-09-05 13:17:39,205 This is just an information

2020-09-05 13:17:39,205 It is a Warning. Please make changes

2020-09-05 13:17:39,205 You are trying to divide by zero

2020-09-05 13:17:39,205 Internet is down

**Explanation:**

The above code will generate a file, and we can see the output while opening a file.

## Formatting the Output

A string passed in the program as a message to log can be modified according to our requirements. There are some basic elements in the given string and part of the **Logrecord**. Let's understand the following example.

**Example -**

1. import logging
3. logging.basicConfig(format='%(process)d-%(levelname)s-%(message)s')
4. logging.warning('This is a Warning Message')

**Output:**

18472-WARNING-This is a Warning Message

The format argument can accept a string with **Logrecord** attributes in any form as we require.

Let's understand the following example -

**Example -**

1. import logging
3. logging.basicConfig(format='%(asctime)s - %(message)s', level=logging.INFO)
4. logging.info('Admin logged in')

**Output:**

2020-09-02 20:12:06,288 - Admin logged in

The %(asctime) attributes adds the time creation of the Logrecord. We can also customize the format using datefmt attributes, which provides the same function as the datetime module.

**Example -**

1. import logging
3. logging.basicConfig(format='%(asctime)s - %(message)s', datefmt='%d-%b-%y %H:%M:%S')
4. logging.warning('Admin logged out')

**Output:**

02-Sep-20 13:29:05 - Admin logged out

## Logging Variable Data

Sometimes, we want to include the dynamic information from the application in the log. The logging methods are accepted a string as an argument, and it is good practice to format a string with the variable data and passed to a log method.

But instead of that, we can also use a format string for the message and appending the variable data as an argument.

Let's understand the following example -

1. import logging
3. name = 'Peter Decosta'
5. logging.error('%s raised an error', name)

**Output:**

ERROR:root: Peter Decosta raised an error

**Explanation:**

The arguments passed to the method would be convoluted as variable data in the message.

We can use the f{string} to format the given string. It provides a short and easy way to handle the string.

**Example -**

1. import logging
3. name = 'Antonio Mino'
5. logging.error(f'{name} raised an error')

**Output:**

ERROR:root: Antonio Mino raised an error

## Capturing Stack Traces

We can capture the full stacks of traces in an application using the logging module. There is **an exc\_info** parameter in the logging function; if we set it as True, it can capture the **Exception information**.

Let's understand the following example -

**Example -**

1. import logging
3. a = 10
4. b = 0
6. try:
7. c = a / b
8. except Exception as e:
9. logging.error("Exception occurred", exc\_info=True)

**Output:**

ERROR:root:Exception occurred

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/loggingFile.py", line 224, in <module>

c = a / b

ZeroDivisionError: division by zero

**Explanation:**

If we don't set true to exc\_info, the output will not inform us about the exception. It would be hard to debug an error in thousand lines of code, if it displays only the following output.

1. ERROR:root:Exception occurred

There is also other option to get complete information about the exception. The logging module provides the **exception()** method, which logs a message with ERROR and attaches the exception information. To use it, call the **logging.exception()** method same as calling **logging.error(exc\_info = True)**.

Let's understand the following example.

**Example -**

1. import logging
3. a = 10
4. b = 0
6. try:
7. c = a / b
8. except Exception as e:
9. logging.exception("Exception occurred", exc\_info=True)

**Output:**

ERROR:root:Exception occurred

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/loggingFile.py", line 224, in <module>

c = a / b

ZeroDivisionError: division by zero

We can use any of one option in error(), debug(), or critical() methods to get information about the exception.

## Classes and Functions

We have seen so far the default logger called **root**. The logging module is used it whenever its functions are called such as **logging.debug(),** logging.error(), etc. We can also define own logger by creating an object of the **Logger** class. Here, we are defining the commonly used classes and functions.

Below are the classes and functions defined in the logging module.

* **Logger -** The logger object is used to call the functions directly.
* **LogRecord -** It creates automatically log record file which consists the information related to all event of being logged such as the logger's name, the function, the line number, the message, and more.
* **Handler -** The handlers are used to dispatch the **LogRecord** to the output endpoint. The **FileHandler, StreamHandler, HTTPHandler, SMTTPHandler** are the subclasses of a **Handler**.
* **Formatters -** The formatters are used to define the structure of the output. It is used the string formatting methods to specify the format of the log messages.

If we don't have the message to format, the default is to use the raw message. The default format date format is.

1. %Y-%m-%d %H:%M:%S

The following format is used to make the log message in the human -readable format.

1. '%(asctime)s - %(levelname)s - %(message)s'

We generally work with the objects of the Logger class, which are created using the **logging.getLogger(name)** function. If the **getLogger()** method is called multiple times with the same name, it will return the reference of the same logger object.

Let's understand the following example:

**Example -**

1. import logging
3. logger = logging.getLogger('first\_logger')
4. logger.warning('This is a warning message')

**Output:**

This is a warning message

**Explanation:**

We have created the own logger name **first\_logger**, but unlike the root logger, the **first\_logger** is not part of the output format. To display it, pass it into the configuration function. Then the output will look like as follows.

1. WARNING:first\_logger:This is a warning message

## Work With Handlers

Handlers are generally used to configure logger and transmit the logs to the many places at a time. It sends the log messages to the standard output stream or a file over HTTP or on email.

Let's understand the following example of creating handlers.

**Example:**

1. import logging
3. # Create a custom logger\_obj
4. logger\_obj = logging.getLogger(\_\_name\_\_)
6. # Create handlers
7. w\_handler = logging.StreamHandler()
8. e\_handler = logging.FileHandler('file.log')
9. w\_handler.setLevel(logging.WARNING)
10. e\_handler.setLevel(logging.ERROR)
12. # Create formatters and add it to handlers
13. c\_format = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
14. f\_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
15. w\_handler.setFormatter(c\_format)
16. e\_handler.setFormatter(f\_format)
18. # Add handlers to the logger\_obj
19. logger\_obj.addHandler(w\_handler)
20. logger\_obj.addHandler(e\_handler)
22. logger\_obj.warning('This is a warning message')
23. logger\_obj.error('This is an error message')

**Output:**

\_\_main\_\_ - WARNING - This is a warning message

\_\_main\_\_ - ERROR - This is an error message

**Explanation:**

In the following program, we have created a custom logger named the **logger\_obj** and created a LogRecord that stores the all record of the logging events and passed it to all the Handlers that it has: **w\_handlers** and **e\_handlers**.

The w\_handlers is a **stream handler** with the level WARNING. It accepts the log from the **LogRecord** to generate the output in the format string and print it to the screen.

The **e\_handler is a** file handler with the level **ERROR**. It disregards the LogRecord as its level WARNING.

## Conclusion

The logging module is flexible and easy to use. It is very useful for keeping track of the logging records and displaying the appropriate message to the user. It provides the flexibility to create custom log levels, handler classes, and many other useful methods.

It also provides basic logging for small projects.

In this tutorial, we have discussed all the essential concepts of the logging module. We have covered generate messages with different levels.

# Python \_\_call\_\_ method

### An Introduction to Magic Methods

Method Names having double underscores as prefixes and suffixes are the reserved methods for particular use in Python. For example, the **\_\_init\_\_** method is used for object constructors or the **\_\_call\_\_** method for making object callable. We call these methods **dunder methods**, where **dunder** refers to **Double Under (Underscores)**. These **dunder methods** are also known as **magic methods** - However, there is nothing magical related to them. There are many developers in the Python community who are not a big fan of the word 'magic', as it delivers a feeling that the use of this method is discouraged, but quite the contradiction is factual.

### Understanding the callable function

The Object in [Python](https://www.javatpoint.com/python-tutorial) is called callable when that object is defined within the **\_\_call\_\_()** function. The function for the same can be defined as **x(arg 1, arg 2, …)**, which is short for **x.\_\_call\_\_(arg1, arg2, …)**.

#### Note: The callable() method returns the Boolean value for whether the object appears to be callable. This function returns True if the object is callable; else, it returns False. Moreover, it is also possible that this function may return True even when the object is not callable. Still, if this method returns False, then the object is not callable.

Moreover, a Python class is always Callable. Thus, we can always utilize the **callable()** function with an object of the class, not the class itself.

Let us consider the following example in order to understand the behavior of the Python **callable()** function.

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

**Example:**

1. # creating a **class**
2. **class** Employee:
3. n = 0
4. # defining a function
5. def \_\_init\_\_(self, emp\_id):
6. self.i = emp\_id
8. # instantiating the **class**
9. x = Employee(15)
11. # checking **if** the **class** and object are callable or not
12. print('Employee Class is callable = ', callable(Employee))
13. print('Employee object is callable = ',

**Output:**

Employee Class is callable = True

Employee object is callable = False

**Explanation:**

In the above example, we have defined a class **Employee** that takes a variable **n = 0**. We have then defined a function and instantiated the class. At last, we have checked whether the class and its object are callable or not using the **callable()** function.

Let us consider an example of **\_\_call\_\_()** function.

**Example:**

1. # creating Person Class
2. **class** Person:
3. person\_id = 0
4. person\_name = ""
5. # defining an initializing function
6. def \_\_init\_\_(self, personid, personname):
7. self.person\_id = personid
8. self.person\_name = personname
10. # defining a callable function
11. def \_\_call\_\_(self, \*args, \*\*kwargs):
12. print('Printing Arguments')
13. print(\*args)
15. print('Printing Keyword Arguments')
16. **for** key, value in kwargs.items():
17. print("%s == %s" % (key, value))
19. # instantiating the **class**
20. m = Person(15, 'George')
22. # printing the object
23. print(m)
25. # checking **if** the object is callable or not
26. print("The Person object is callable: ", callable

**Output:**

<\_\_main\_\_.Person object at 0x000002706DF41FD0>

The Person object is callable: True

**Explanation:**

In the above example, we have defined the **Person** class and the variables **person\_id = 0** and **person\_name = " "**. We have then defined the initializing function along with the callable function. At last, we have instantiated the class, printed the object, and checked whether the object is callable or not. And as a result, the person object appears to be callable.

Moreover, we can observe that we have used the **\*args** to allow passing variable arguments and **\*\*kwargs** to allow passing named arguments to the **\_\_call\_\_()** method.

Now, let us consider another example where we will have used the **callable()** function to check whether the object is callable or not and then called the object as a function.

**Example:**

1. **if** callable(m):
2. m()
3. m(15, 30)
4. m.\_\_call\_\_(15, 30)
6. m(15, 30, {'a': 4, 'b': 8})
8. m(15, 'B', personname = 'George', personid = 50)

**Output:**

Printing Arguments

Printing Keyword Arguments

Printing Arguments

15 30

Printing Keyword Arguments

Printing Arguments

15 30

Printing Keyword Arguments

Printing Arguments

15 30 {'a': 4, 'b': 8}

Printing Keyword Arguments

Printing Arguments

15 B

Printing Keyword Arguments

personname == George

personid == 50

**Explanation:**

In the above example, we have defined an **if**-statement where if the object m is callable, then the object is called as a function without arguments, with only arguments, with **\_\_call\_\_()** function, with the arguments of different types and with arguments and keyword arguments. As a result, the required objects have been called successfully.

# Insertion Sort in Python

The Insertion sort is a straightforward and more efficient algorithm than the previous bubble sort algorithm. The insertion sort algorithm concept is based on the deck of the card where we sort the playing card according to a particular card. It has many advantages, but there are many efficient algorithms available in the data structure.

While the card-playing, we compare the hands of cards with each other. Most of the player likes to sort the card in the ascending order so they can quickly see which combinations they have at their disposal.

The insertion sort implementation is easy and simple because it's generally taught in the beginning programming lesson. It is an **in-place** and **stable algorithm** that is more beneficial for nearly-sorted or fewer elements.

The insertion sort algorithm is not so fast because of it uses nested loop for sort the elements.

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

Let's understand the following terms.

### What is the meaning of in-place and stable?

* **In-place:** The in-place algorithm requires additional space without caring for the input size of the collection. After performing the sorting, it rewrites the original memory locations of the elements in the collection.
* **Stable:** The stable is a term that manages the relative order of equal objects from the initial array.

The more important thing, the insertion sort doesn't require to know the array size in advance and it receives the one element at a time.

The great thing about the insertion sort is if we insert the more elements to be sorted - the algorithm arranges the in its proper place without performing the complete sort.

It is more efficient for the small (less than 10) size array. Now, let's understand the concepts of insertion sort.

## The Concept of Insertion Sort

The array spilled virtually in the two parts in the insertion sort - An **unsorted part** and **sorted** part.

The sorted part contains the first element of the array and other unsorted subpart contains the rest of the array. The first element in the unsorted array is compared to the sorted array so that we can place it into a proper sub-array.

It focuses on inserting the elements by moving all elements if the right-side value is smaller than the left side.

It will repeatedly happen until the all element is inserted at correct place.

To sort the array using insertion sort below is the algorithm of insertion sort.

* Spilt a list in two parts - sorted and unsorted.
* Iterate from arr[1] to arr[n] over the given array.
* Compare the current element to the next element.
* If the current element is smaller than the next element, compare to the element before, Move to the greater elements one position up to make space for the swapped element.

Let's understand the following example.

We will consider the **first element** in the **sorted array** in the following array.

[10, 4, 25, 1, 5]

The first step to **add 10** to the sorted subarray

[**10**, 4, 25, 1, 5]

Now we take the first element from the unsorted array - 4. We store this value in a new variable **temp. Now**, we can see that the 10>4 then we move the 10 to the right and that overwrite the 4 that was previously stored.

[**10**, 10, 25, 1, 5] (temp = 4)

Here the 4 is lesser than all elements in sorted subarray, so we insert it at the first index position.

[**4, 10,** 25, 1, 5]

We have two elements in the sorted subarray.

Now check the number 25. We have saved it into the temp **variable.** 25> 10 and also 25> 4 then we put it in the third position and add it to the sorted sub array.

[**4, 10, 25,** 1, 5]

Again we check the number 1. We save it in **temp.** 1 is less than the 25. It overwrites the 25.

[**4, 10, 25,** 25, 5] 10>1 then it overwrites again

[**4, 25, 10,** 25, 5]

[**25, 4, 10,** 25, 5] 4>1 now put the value of temp = 1

[**1, 4, 10, 25**, 5]

Now, we have 4 elements in the sorted subarray. 5<25 then shift 25 to the right side and pass **temp = 5** to the left side.

[**1, 4, 10, 25**, 25] put temp = 5

Now, we get the sorted array by simply putting the temp value.

**[1, 4, 5, 10, 25]**

The given array is sorted.

## Implementation

The implementation of insertion is relative easy. We will implement using the Python array of integers. Let's understand the following example -

**Python Program**

1. # creating a function for insertion
2. def insertion\_sort(list1):
4. # Outer loop to traverse through 1 to len(list1)
5. for i in range(1, len(list1)):
7. value = list1[i]
9. # Move elements of list1[0..i-1], that are
10. # greater than value, to one position ahead
11. # of their current position
12. j = i - 1
13. while j **>**= 0 and value **<** **list1**[j]:
14. list1[j + 1] = list1[j]
15. j -= 1
16. list1[j + 1] = value
17. return list1
18. # Driver code to test above
20. list1 = [10, 5, 13, 8, 2]
21. print("The unsorted list is:", list1)
23. print("The sorted list1 is:", insertion\_sort(list1))

**Output:**

The unsorted list is: [10, 5, 13, 8, 2]

The sorted list1 is: [2, 5, 8, 10, 13]

**Explanation:**

In the above code, we have created a function called **insertion\_sort(list1).** Inside the function -

* We defined for loop for traverse the list from 1 to **len(list1).**
* In for loop, assigned a values of list1 in **value** Every time the loop will iterate the new value will assign to the value variable.
* Next, we moved the elements of list1[0…i-1], that are greater than the **value,** to one position ahead of their current position.
* Now, we used the while to check whether the j is greater or equal than 0, and the **value** is smaller than the first element of the list.
* If both conditions are true then move the first element to the 0th index and reduce the value of j and so on.
* After that, we called the function and passed the list and printed the result.

## Sorting Custom Objects

Python provides the flexibility to change the algorithm using a custom object. We will create a custom class and redefine the actual comparison parameter and try to keep the same code as the above.

We would require to overload the operators in order to sort the objects in a different way. But, we can pass another argument to the **insertion\_sort()** function by using the **lambda** function. The lambda function is a convenient when calling the sorting method.

Let's understand the following example of sorting custom objects.

First, we are defining the **Point** class:

### Python Program

1. # Creating Point class
2. class Point:
3. def \_\_init\_\_(self, a, b):
4. self.a = a
5. self.b = b
7. def \_\_str\_\_(self):
8. return str.format("({},{})", self.a, self.b)
10. def insertion\_sort(list1, compare\_function):
11. for i in range(1, len(list1)):
12. Value = list1[i]
13. Position = i
15. while Position **>** 0 and compare\_function(list1[Position - 1], Value):
16. list1[Position] = list1[Position - 1]
17. PositionPosition = Position - 1
19. list1[Position] = Value
21. U = Point(2,3)
22. V = Point(4,4)
23. X = Point(3,1)
24. Y = Point(8,0)
25. Z = Point(5,2)
27. list1 = [U,V,X,Y,Z]
29. # We sort by the x coordinate, ascending
30. insertion\_sort(list1, lambda x, y: x.a **>** y.a)
32. for point in list1:
33. print(point)

**Output:**

The points are in the sorted order

(2,3)

(3,1)

(4,4)

(5,2)

(8,0)

Using the above code, we can sort the coordinate points. It will work for any type of the list.

## Time Complexity in Insertion Sort

Insertion sort is a slow algorithm; sometimes, it seems too slow for extensive dataset. However, it is efficient for small lists or array.

The time complexity of the insertion sort is - **O(n2).** It uses the two loops for iteration.

Another important advantage of the insertion sort is that; it is used by the popular sorting algorithm called **Shell sort.**

The auxiliary space in insertion sort: **O(1)**

## Conclusion

Insertion sort is a simple and inefficient algorithm that has many advantages, but there are more efficient algorithms are available.

In this tutorial, we have discussed the concept of the insertion sort and its implementation using the Python programming language.

[**Next →**](https://www.javatpoint.com/linear-search-in-python)[**← Prev**](https://www.javatpoint.com/insertion-sort-in-python)

# Binary Search in Python

This tutorial will learn how we can apply a binary search algorithm using Python to find an element's index position in the given list.

## Introduction

A binary search is an algorithm to find a particular element in the list. Suppose we have a list of thousand elements, and we need to get an index position of a particular element. We can find the element's index position very fast using the binary search algorithm.

There are many searching algorithms but the binary search is most popular among them.

The elements in the list must be sorted to apply the binary search algorithm. If elements are not sorted then sort them first.

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

Let's understand the concept of binary search.

## Concept of Binary Search

In the binary search algorithm, we can find the element position using the following methods.

* Recursive Method
* Iterative Method

The divide and conquer approach technique is followed by the recursive method. In this method, a function is called itself again and again until it found an element in the list.

A set of statements is repeated multiple times to find an element's index position in the iterative method. The **while** loop is used for accomplish this task.

Binary search is more effective than the linear search because we don't need to search each list index. The list must be sorted to achieve the binary search algorithm.

Let's have a step by step implementation of binary search.

We have a sorted list of elements, and we are looking for the index position of 45.

**[12, 24, 32, 39, 45, 50, 54]**

So, we are setting two pointers in our list. One pointer is used to denote the smaller value called **low** and the second pointer is used to denote the highest value called **high**.

Next, we calculate the value of the **middle** element in the array.

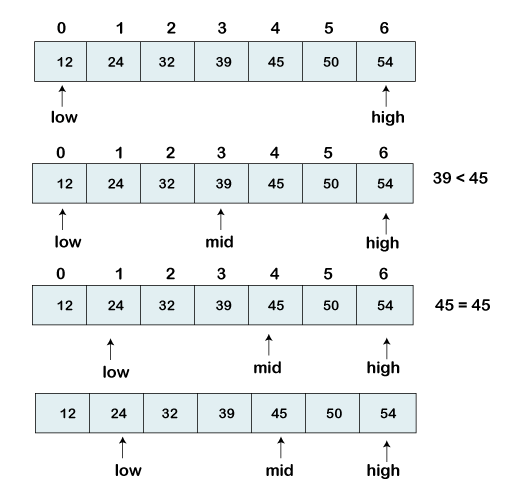
1. mid = (low+high)/2
2. Here, the low is 0 and the high is 7.
3. mid = (0+7)/2
4. mid = 3 (Integer)

Now, we will compare the searched element to the mid index value. In this case, **32** is not equal to **45.** So we need to do further comparison to find the element.

If the number we are searching equal to the mid. Then return **mid** otherwise move to the further comparison.

The number to be search is greater than the **middle** number, we compare the **n** with the middle element of the elements on the right side of **mid** and set low to **low = mid + 1.**

Otherwise, compare the **n** with the **middle element** of the elements on the left side of **mid** and set **high** to **high = mid - 1.**



Repeat until the number that we are searching for is found.

## Implement a Binary Search in Python

First, we implement a binary search with the iterative method. We will repeat a set of statements and iterate every item of the list. We will find the middle value until the search is complete.

Let's understand the following program of the iterative method.

### Python Implementation

1. # Iterative Binary Search Function method Python Implementation
2. # It returns index of n in given list1 if present,
3. # else returns -1
4. def binary\_search(list1, n):
5. low = 0
6. high = len(list1) - 1
7. mid = 0
9. while low **<**= high:
10. # for get integer result
11. mid = (high + low) // 2
13. # Check if n is present at mid
14. if list1[mid] **<** **n:**
15. low = mid + 1
17. # If n is greater, compare to the right of mid
18. elif list1[mid] **>** n:
19. high = mid - 1
21. # If n is smaller, compared to the left of mid
22. else:
23. return mid
25. # element was not present in the list, return -1
26. return -1

29. # Initial list1
30. list1 = [12, 24, 32, 39, 45, 50, 54]
31. n = 45
33. # Function call
34. result = binary\_search(list1, n)
36. if result != -1:
37. print("Element is present at index", str(result))
38. else:
39. print("Element is not present in list1")

**Output:**

Element is present at index 4

**Explanation:**

In the above program -

* We have created a function called **binary\_search()** function which takes two arguments - a list to sorted and a number to be searched.
* We have declared two variables to store the lowest and highest values in the list. The low is assigned initial value to 0, **high** to **len(list1)** - 1 and mid as 0.
* Next, we have declared the **while** loop with the condition that the **lowest** is equal and smaller than the **highest** The while loop will iterate if the number has not been found yet.
* In the while loop, we find the mid value and compare the index value to the number we are searching for.
* If the value of the mid-index is smaller than **n**, we increase the mid value by 1 and assign it to The search moves to the left side.
* Otherwise, decrease the mid value and assign it to the **high**. The search moves to the right side.
* If the n is equal to the mid value then return **mid**.
* This will happen until the **low** is equal and smaller than the **high**.
* If we reach at the end of the function, then the element is not present in the list. We return -1 to the calling function.

Let's understand the recursive method of binary search.

## Recursive Binary Search

The recursion method can be used in the binary search. In this, we will define a recursive function that keeps calling itself until it meets the condition.

Let's understand the above program using the recursive function.

### Python Program

1. # Python program for recursive binary search.
2. # Returns index position of n in list1 if present, otherwise -1
3. def binary\_search(list1, low, high, n):
5. # Check base case for the recursive function
6. if low **<**= high:
8. mid = (low + high) // 2
10. # If element is available at the middle itself then return the its index
11. if list1[mid] == n:
12. return mid
14. # If the element is smaller than mid value, then search moves
15. # left sublist1
16. elif list1[mid] **>** n:
17. return binary\_search(list1, low, mid - 1, n)
19. # Else the search moves to the right sublist1
20. else:
21. return binary\_search(list1, mid + 1, high, n)
23. else:
24. # Element is not available in the list1
25. return -1
27. # Test list1ay
28. list1 = [12, 24, 32, 39, 45, 50, 54]
29. n = 32
31. # Function call
32. res = binary\_search(list1, 0, len(list1)-1, n)
34. if res != -1:
35. print("Element is present at index", str(res))
36. else:
37. print("Element is not present in list1")

**Output:**

Element is present at index 2

**Explanation**

The above program is similar to the previous program. We declared a recursive function and its base condition. The condition is the lowest value is smaller or equal to the highest value.

* We calculate the middle number as in the last program.
* We have used the **if** statement to proceed with the binary search.
* If the middle value equal to the number that we are looking for, the middle value is returned.
* If the middle value is less than the value, we are looking then our recursive function **binary\_search()** again and increase the mid value by one and assign to low.
* If the middle value is greater than the value we are looking then our recursive function **binary\_search()** again and decrease the mid value by one and assign it to low.

In the last part, we have written our main program. It is the same as the previous program, but the only difference is that we have passed two parameters in the **binary\_search()** function.

This is because we can't assign the initial values to the low, high and mid in the recursive function. Every time the recursive is called the value will be reset for those variables. That will give the wrong result.

## Complexity

The complexity of the binary search algorithm is **O(1)** for the best case. This happen if the element that element we are looking find in the first comparison. The **O(logn)** is the worst and the average case complexity of the binary search. This depends upon the number of searches are conducted to find the element that we are looking for.

## Conclusion

A binary search algorithm is the most efficient and fast way to search an element in the list. It skips the unnecessary comparison. As the name suggests, the search is divided into two parts. It focuses on the side of list, which is close to the number that we are searching.

We have discussed both methods to find the index position of the given number.

Linear Search in Python

Python is one of the most popular and powerful languages. It takes a few lines to execute the code, which makes it much user-friendly language. In this tutorial, we will learn the linear search in Python. Searching is a technique to find the particular element is present or not in the given list.

There are two types of searching -

* Linear Search
* Binary Search

Both techniques are widely used to search an element in the given list.

What is a Linear Search?

Linear search is a method of finding elements within a list. It is also called a sequential search.  It is the simplest searching algorithm because it searches the desired element in a sequential manner.

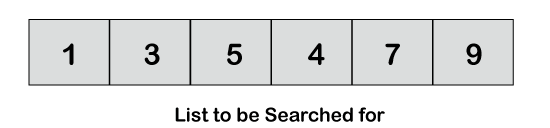
Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

It compares each and every element with the value that we are searching for. If both are matched, the element is found, and the algorithm returns the key's index position.

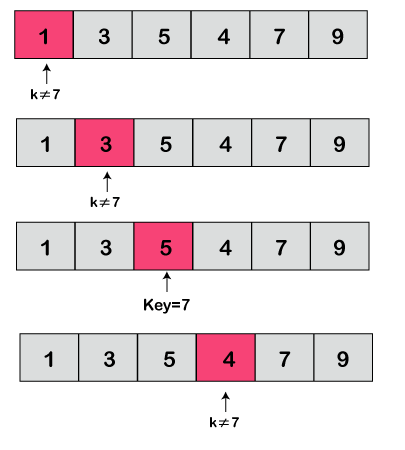
Concept of Linear Search

Let's understand the following steps to find the element key = 7 in the given list.

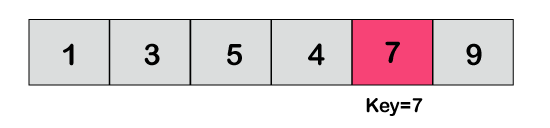
**Step - 1:** Start the search from the first element and Check key = 7 with each element of list x.



**Step - 2:** If element is found, return the index position of the key.



**Step - 3:** If element is not found, return element is not present.



Linear Search Algorithm

There is list of n elements and key value to be searched.

Below is the linear search algorithm.

1. LinearSearch(list, key)
2. for each item in the list
3. if item == value
4. return its index position
5. return -1

Python Program

Let's understand the following Python implementation of the linear search algorithm.

**Program**

1. def linear\_Search(list1, n, key):
3. # Searching list1 sequentially
4. for i in range(0, n):
5. if (list1[i] == key):
6. return i
7. return -1

10. list1 = [1 ,3, 5, 4, 7, 9]
11. key = 7
13. n = len(list1)
14. res = linear\_Search(list1, n, key)
15. if(res == -1):
16. print("Element not found")
17. else:
18. print("Element found at index: ", res)

**Output:**

Element found at index: 4

**Explanation:**

In the above code, we have created a function **linear\_Search(),** which takes three arguments - list1, length of the list, and number to search. We defined for loop and iterate each element and compare to the key value. If element is found, return the index else return -1 which means element is not present in the list.

Linear Search Complexity

Time complexity of linear search algorithm -

* Base Case - O(1)
* Average Case - O(n)
* Worst Case -O(n)

Linear search algorithm is suitable for smaller list (<100) because it check every element to get the desired number. Suppose there are 10,000 element list and desired element is available at the last position, this will consume much time by comparing with each element of the list.

To get the fast result, we can use the binary search algorithm.

We have discussed the basic concept of the linear search. In the next tutorial, we will learn the second and most popular searching algorithm named Binary search.

# Queue in Python

In this tutorial, we will discuss the Queue's basic concepts and built-in Queue class and implement it using the Python code.

### What is the Queue?

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

**For example -** There are 20 computers in the computer science lab and connected to a single printer. The students want to print their paper; the printer will print the first task and second, so on. If we are the last in line, we need to wait until all other tasks are completed that ahead of ours.

The operating system manages the queue for processing the various processes within a computer.

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

## Operations in Python

We can perform the following operations in the Queue.

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

## Methods Available in Queue

[Python](https://www.javatpoint.com/python-tutorial) provides the following methods, which are commonly used to perform the operation in Queue.

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

We will learn these functions in the below sections.

## The built-in Python List

The list can be used as the queue, but it is not suitable for a performance perspective. Python provides built-in methods **insert()** and **pop()** function to add and remove elements. Lists are quite slow because if we insert a new element to the list, all elements require shifting by one. It takes O(n) time. So lists are recommended in-place of queue. Let's understand the following example of how a list can be used as a queue.

**Example -**

1. que = []
3. que.append('Apple')
4. que.append('Mango')
5. que.append('Papaya')
7. print(que)
9. # List is slow!
10. print(que.pop(0))

**Output:**

['Apple', 'Mango', 'Papaya']

Apple

**Explanation -**

We have defined the empty list in the above code and inserted a few elements using **append()** method. It will add an element to the end of the list.

## Adding Element to a Queue (Enqueue)

We can add the element from to the rear end. This process is also called enqueue. We create a Queue class where we will implement the First-in-First-Out concept. Let's understand the following example.

**Example -**

1. class Queue:
3. def \_\_init\_\_(self):
4. self.queue = list()
6. def add\_element(self,val):
7. # Insert method to add element
8. if val not in self.queue:
9. self.queue.insert(0,val)
10. return True
11. return False
13. def size(self):
14. return len(self.queue)
16. TheQueue = Queue()
17. TheQueue.add\_element("Apple")
18. TheQueue.add\_element("Mango")
19. TheQueue.add\_element("Guava")
20. TheQueue.add\_element("Papaya")
22. print("The length of Queue: ",TheQueue.size())

**Output:**

The length of Queue: 4

## Removing Element from a Queue (Dequeue)

We can remove the element form the rear end. This process is called a dequeue. In the following example, we use the built-in pop() method to remove an element from the list.

**Example -**

1. class Queue:
3. def \_\_init\_\_(self):
4. self.queue = list()
6. def add\_element(self,val):
7. # Insert method to add element
8. if val not in self.queue:
9. self.queue.insert(0,val)
10. return True
11. return False
12. # Pop method to remove element
13. def remove\_element(self):
14. if len(self.queue)**>**0:
15. return self.queue.pop()
16. return ("Queue is Empty")
18. que = Queue()
19. que.add\_element("January")
20. que.add\_element("February")
21. que.add\_element("March")
22. que.add\_element("April")
24. print(que)
25. print(que.remove\_element())
26. print(que.remove\_element())

**Output:**

January

February

**Explanation -**

In the above code, we have defined a class named Queue and constructor in it. We assigned an list constructor to the **queue** variable. Then, we defined two methods - **add\_element()** and **remove\_element()**. In the **add\_element()** block, we check the condition if the value is not in Queue. If value is not present, insert the element.

In the **remove\_element()** function block, we check the condition of whether a queue is not underflow. If it returns false, then remove the element one by one.

## Sorting the Queue

In the following example, we have sorted the elements of the queue.

**Example -**

1. import queue
2. q = queue.Queue()
4. q.put(14)
5. q.put(27)
6. q.put(11)
7. q.put(4)
8. q.put(1)

11. # Here, we use bubble sort algorithm for sorting
12. n =  q.qsize()
13. for i in range(n):
14. # Remove the element
15. x = q.get()
16. for j in range(n-1):
17. # Remove the element
18. y = q.get()
19. if x **>** y :
20. # put the smaller element at the beginning of the queue
21. q.put(y)
22. else:
23. # the smaller one is put at the start of the queue
24. q.put(x)
25. x = y    # The greater element is replaced by the x and check again
26. q.put(x)
28. while (q.empty() == False):
29. print(q.queue[0], end = " ")
30. q.get()

**Output:**

1 4 11 14 27

## The Queue Module

Python provides the queue module to implement multi-producer, multi-consumer queues. The queue module offers Queue class which is especially used for the threaded programming. The Queue class implements all the required locking semantics.

We can perform all the operation using the in-built queue class.

## Working With queue.Queue Class

The queue module contains several classes. The Queue is one of the important classes of them. This is very useful in the parallel computing and multiprogramming. Let's understand the following example of the queue. Queue class0uii

**Example -**

1. from queue import Queue
2. que = Queue()
4. que.put('Apple')
5. que.put('Mango')
6. que.put('Papaya')
8. print(que)

11. print(que.get())
13. print(que.get())
15. print(que.get())
17. print(que.get\_nowait())
19. print(que.get())

**Output:**

<queue.Queue object at 0x00000114B30656A0>

Apple

Mango

Papaya

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/Queue.py", line 78, in <module>

print(que.get\_nowait())

File "C:\Python\lib\queue.py", line 198, in get\_nowait

return self.get(block=False)

File "C:\Python\lib\queue.py", line 167, in get

raise Empty

\_queue.Empty

## Working With collection.deque Class

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

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

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

**Example -**

1. from collections import deque
2. que = deque()
4. que.append('Apple')
5. que.append('Mango')
6. que.append('Banana')
8. print(que)
9. deque(['Apple ', 'Mango', 'Banana'])
11. print(que.popleft())
13. print(que.popleft())
15. print(que.popleft())

18. que.popleft()

**Output:**

deque(['Apple', 'Mango', 'Banana'])

Apple

Mango

Banana

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/Queue.py", line 101, in <module>

que.popleft()

IndexError: pop from an empty deque

## The multiprocessing.Queue Class

The multiprocessing.Queue class is used to implement queued items for processed in parallel by **multicurrent** workers. The multiprocessing.Queue shares data between processes and can store any pickle-able object. Let's understand the following example.

**Example -**

1. from multiprocessing import Queue
2. que = Queue()
4. que.put('Apple')
5. que.put('Mango')
6. que.put('Banana')
8. print(que)
10. print(que.get())
12. print(que.get())
14. print(que.get())

**Output:**

<multiprocessing.queues.Queue object at 0x000002CA073356A0>

Apple

Mango

Banana

## Priority Queue in Python

A priority queue is a special type of queue in the data-structure. As the name suggest, it sorts the elements and dequeues the elements based on their priorities.

Unlike normal queue, it retrieves the highest-priority element instead of the next element. The priority of individual elements is decided by ordering applied to their keys.

Priority queues are most beneficial to handling scheduling problems where some tasks will happen based on priorities.

**For example -** An operating system task is the best example of a priority queue - It executes the high precedence over lower-priority tasks (downloading updates in the background). The task scheduler can allow the highest-priority tasks to run first.

There are various ways to implement a priority queue in Python. Let's understand the following ways.

## Manually Sorted List

We can use the sorted Python list as the priority queue to quickly identify and delete the smaller and largest element. But inserting the new element is slow as it takes **O(n)** operations.

Therefore sorted list can be effective when there are will be few insertions into the priority queue.

Let's understand the following example -

**Example -**

1. pri\_que = []
3. pri\_que.append((2, 'Apple'))
4. pri\_que.append((1, 'Mango'))
5. pri\_que.append((3, 'Banana'))
7. # NOTE: Remember to re-sort every time
8. #       a new element is inserted.
9. pri\_que.sort(reverse=True)
11. while pri\_que:
12. next\_item = pri\_que.pop()
13. print(next\_item)

**Output:**

(1, 'Mango')

(2, 'Apple')

(3, 'Banana')

## The queue.PriorityQueue Class

This priority queue implements uses **heapq** internally and shares the same time and space complexities.

The difference is the priority queue is coordinated and delivers locking semantics to backing multiple concurrent events and consumers.

**Example -**

1. from queue import PriorityQueue
2. q = PriorityQueue()
4. q.put((2, 'Apple'))
5. q.put((1, 'Banana'))
6. q.put((3, 'Mango'))
8. while not q.empty():
9. next\_item = q.get()
10. print(next\_item)

**Output:**

(1, 'Banana')

(2, 'Apple')

(3, 'Mango')

We can choose any priority queue implementation in the Python program but keep in mind that **queue.PriorityQueue** is good default choice.

## Conclusion

We have discussed all the basic concepts of queue and its implementation. It is similar to the standard list, but performance-wise it is always better. We have also defined the priority queue and its various ways of implementation.

# Stack in Python

In this tutorial, we will learn the basics of the stack and implement it using the Python code.

### What is a Stack?

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

We always pick the last plate from the stack of the plate. In stack, the new element is inserted at the one end and an element can be removed only that end.

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

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

### Methods of Stack

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

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

## Implementation of Stack

Python offers various ways to implement the stack. In this section, we will discuss the implementation of the stack using Python and its module.

We can implement a stack in Python in the following ways.

* List
* dequeu
* LifeQueue

### Implementation Using List

Python list can be used as the stack. It uses the **append()** method to insert elements to the list where stack uses the **push()** method. The list also provides the pop() method to remove the last element, but there are shortcomings in the list. The list becomes slow as it grows.

The list stores the new element in the next to other. If the list grows and out of a block of memory then Python allocates some memory. That's why the list become slow. Let's understand the following example -

1. # initial empty stack
2. my\_stack = []
4. # append() function to push
5. # element in the my\_stack
6. my\_stack.append('x')
7. my\_stack.append('y')
8. my\_stack.append('z')

11. print(my\_stack)
13. # pop() function to pop
14. # element from my\_stack in
15. # LIFO order
16. print('\nElements poped from my\_stack:')
17. print(my\_stack.pop())
18. print(my\_stack.pop())
19. print(my\_stack.pop())
21. print('\nmy\_stack after elements are poped:')
22. print(my\_stack)

**Output:**

['x', 'y', 'z']

Elements poped from my\_stack:

z

y

x

my\_stack after elements are poped:

[]

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/Stack.py", line 21, in <module>

print(my\_stack.pop())

IndexError: pop from empty list

**Explanation -**

In the above code, we have defined an empty list. We inserted the elements one by one using the **append()** method. That is similar to the **push()** method. We also removed the elements using the **pop()** method. The **pop()** method returns the last element of the list.

### Implementation using collection.deque

The collection module provides the deque class, which is used to creating Python **stacks.** The deque is pronounced as the "deck" which means "double-ended queue". The deque can be preferred over the list because it performs append and pop operation faster than the list. The time complexity is O(1), where the list takes O(n).

Let's understand the following example -

**Example**

1. from collections import deque
3. my\_stack = deque()
5. # append() function is used to push
6. # element in the my\_stack
7. my\_stack.append('a')
8. my\_stack.append('b')
9. my\_stack.append('c')
11. print('Initial my\_stack:')
12. print(my\_stack)
14. # pop() function is used to pop
15. # element from my\_stack in
16. # LIFO order
17. print('\nElements poped from my\_stack:')
18. print(my\_stack.pop())
19. print(my\_stack.pop())
20. print(my\_stack.pop())
22. print('\nmy\_stack after elements are poped:')
23. print(my\_stack)

**Output:**

Initial my\_stack:

deque(['a', 'b', 'c'])

Elements poped from my\_stack:

c

b

a

my\_stack after elements are poped:

deque([])

**Explanation:**

The above code is almost similar to the previous example. However, the only difference is that, we have imported the deque from the collection module.

### Deque Vs. list

The list stores the element right next to each other and uses the contiguous memory. This is most effective for the several operations, like indexing into the list. For example - Getting **list1[2]** is fast as Python knows the exact position of a particular element. The contiguous memory also allows slice to work well on the lists.

The list consumes extra time to **append()** some objects than others. If the block of contiguous memory is full, it will get another block that can take much a longer time than a normal **append()** function.

## Implementation Using queue module

The queue module has the LIFO queue, which is the same as the stack. Generally, the queue uses **the put()** method to add the data and **the ()** method to take the data.

Below are a few methods that available in the queue.

* **empty() -** If queue empty, returns true; otherwise return false.
* **maxsize() -** This method is used to set the maximum number of elements allowed in the queue.
* **get() -** It returns and removes the element from the queue. Sometime. The queue can be empty; it waits until element is available.
* **full() -** It returns True if the queue is full. The queue is defined as maxsize = 0 by default. In this case, it will not return **True**.
* **put(item) -** It adds the element to the queue; if the queue is full, wait until space is available.
* **put\_nowait(item) -** It adds the element into the queue without delaying.
* **qsize() -** It returns the size of the queue.

Let's understand the following example of the queue module.

**Example -**

1. # Implementing stack using the queue module
2. from queue import LifoQueue
4. # Initializing a my\_stack stack with maxsize
5. my\_stack = LifoQueue(maxsize = 5)
7. # qsize() display the number of elements
8. # in the my\_stack
9. print(my\_stack.qsize())
11. # put() function is used to push
12. # element in the my\_stack
13. my\_stack.put('x')
14. my\_stack.put('y')
15. my\_stack.put('z')
17. print("Stack is Full: ", my\_stack.full())
18. print("Size of Stack: ", my\_stack.qsize())
20. # To pop the element we used get() function
21. # from my\_stack in
22. # LIFO order
23. print('\nElements poped from the my\_stack')
24. print(my\_stack.get())
25. print(my\_stack.get())
26. print(my\_stack.get())
28. print("\nStack is Empty: ", my\_stack.empty())

**Output:**

0

Stack is Full: False

Size of Stack: 3

Elements poped from the my\_stack

z

y

x

Stack is Empty: True

**Explanation -**

In the above, we have imported the queue module that is a **LIFOqueue**. It works the same as the stack but this module includes some additional functions mentioned above. We defined a stack with the maxsize that means it can hold maximum five values in it.

The initial array size is zero; we pushed three elements in the stack using the put() method. Now, again we checked whether a stack is empty and size of the stack. We have three elements in the stack. We popped the element using the get() method. It removes the last added element first. After removing the entire elements, we get an empty stack.

## Python Stacks and Threading

We can also use Python stack in the multi-threaded program. It is an advanced topic but not frequently used so you can skip this section if you are not interested in threading.

The list and deque behave differently if we are using threading in our program. Using a list in multi-threading programming can be dangerous because it is not thread-safe.

On the other hand, the deque is a little bit complex because its **append()** and **pop()** methods are atomic, which means they will not interrupt by other thread.

We can build the multithreading program using the deque, but it may cause few complexities in the future.

Now the question arises, how we can build a program of Python stack with threading.

The answer is that we can use the **LIFOqueue** and we know that what LIFO stands for - Last In First Out. It uses the **put()** and **gets()** to add and remove the stack element.

## Which Implementation of Stack Should Consider?

We have mentioned three methods of implement the stack in Python. The above methods have their own advantages or disadvantages.

Let us cleat the confusion; we are using stack with the threading, you should use the **Lifoqueue** but make sure about its performance for popping and pushing elements. But if you are not using threading, use a **deque**.

We can also use the list to implement the stack but the list can have potential memory reallocation issues. The list and deque are same in the interface, but the deque doesn't memory allocation issues.

## Conclusion

We have defined a stack and its implementation in brief. Python stack can be used in real-life programs. We can choose either implement the method according to our requirements. We have also defined the Python stack with threading environment.

Heap Sort in Python

The heap sort is quite the same as the selection sort, where we find the maximum element and put it at the end. It is based on a comparison sorting algorithm which works on Binary heap data structure. It is the best example of an efficient sorting algorithm.

What is Heap Sort?

Heap sort is an efficient and popular sorting algorithm. The heap sort concept is to **"eliminate"** the element from the heap part of the list one-by-one and **insert** them to the sorted part of the list. Before learning more about the heap sorting algorithm, let's discuss the heap data structure.

It is an in-place algorithm, which means a fixed amount of memory is used to store the sorted list, or the memory size doesn't rely on the size of the preliminary list.

**For example -** We don't need the additional memory stack to store the sorted array and neither recursive call stack. The heapsort algorithm usually implements using the second array to sort the fixed values. This process is quick, simple, natural and easy to implement.



On the other hand, heap sort is unstable, which means it doesn't maintain the comparative order of elements with equal values. It can quickly sort primitive types such as integers and characters, but it has a problem with the complex types and objects.

Let's understand it by the following example -

We have a custom class **Student** with properties **age** and **name,** and several objects of that class in an array, including a student called **"Thomas"** ages "20" and also **"Peter,"** have aged 20 appear in the same order.

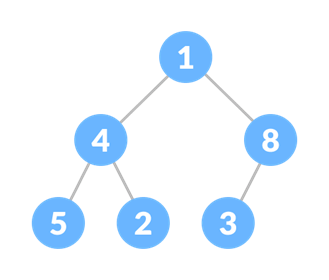
If we sort the array of people by age, then there is no guarantee that **"Thomas"** would appear before the **"Peter"** in the sorted array. It can be defined order, but there is no guarantee.

Heap Data Structure

The heap data structure is **a complete binary tree** that fulfills the heap property. It is also known as the binary heap.

A complete binary tree satisfies the following properties.

* Every level should be filled.
* All the nodes are as far left as possible.



As we can see in the above image of the heap, but it is not sorted. We will not dig in-depth this article because our focus is to explain the Heap sort algorithm, not a heap. In the heap sort, the next smallest element is always the first element.

The heap tree can be the two types - min-heap and max tree. A min-heap is kept a record of the maximum element. A max heap keeps track of the largest element. Heap mainly supports the following operations - delete\_minimum(), get\_minimum() and add().

The first element of the heap can delete after restoring it. It takes **O(log N)** time, and that is highly effective.

Implementation

Python provides the in-built functions for sorting elements using heap sort. The functions are given below.

* **heappush(list, item) -**It is used to add the heap element and re-sort it.
* **heappop(list) -** It is used to remove the element and return the element.
* **heapfy() -** It is used to turn the given list into a heap.

Consider the following example for heap sort.

**Example -**

1. from heapq import heappop, heappush
3. def heapsort(list1):
4. heap = []
5. for ele in list1:
6. heappush(heap, ele)
8. sort = []
10. # the elements are lift in the heap
11. while heap:
12. sort.append(heappop(heap))
14. return sort
16. list1 = [27, 21, 55, 15, 60, 4, 11, 17, 2, 87]
17. print(heapsort(list1))

**Output:**

[2, 4, 11, 15, 17, 21, 27, 55, 60, 87]

**Explanation**

In the above code, we have imported the **heapq** module which consist **heappop()** and **heappush()** method. We created **the Heapsort Heapsort ()** method, which takes list1 as an argument. A for loop iterated the list1 and pushed the elements to the **empty heap**. We used the while loop and sorted element added to the **empty sort**.

We called the **Heapsort Heapsort ()** function and passed a list. It returned the sorted list.

Sorting Custom Objects

Heap sort is useful for predefined data types, but it is more complicated to handle the user-define data types, such as class objects. We will sort the custom objects in this section.

As we can see, our implementation depends upon the built-in methods. Python provides the following methods.

* **heapq.nlargest(\*n\*, \*iterable\*, \*key = None) -** This method is used to get a list with the n largest element from the dataset, defined by the iterable.
* **heapq.nsmallest(\*n\*, \*iterable\*, \*key = None) -** This method is used to get a list with the n smallest elements from the dataset, which is defined by the iterable.

Let's understand the following implementation of custom objects.

**Example -**

1. from heapq import heappop, heappush
3. class Car:
4. def \_\_init\_\_(self, model, year):
5. self.model = model
6. self.year = year
8. def \_\_str\_\_(self):
9. return str.format("Model Name: {}, Year: {}", self.model, self.year)
11. def \_\_lt\_\_(self, other):
12. return self.year **<** **other.year**
14. def \_\_gt\_\_(self, other):
15. return other.\_\_lt\_\_(self)
17. def \_\_eq\_\_(self, other):
18. return self.year == other.year
20. def \_\_ne\_\_(self, other):
21. return not self.\_\_eq\_\_(other)

24. def heapsort(list1):
25. heap = []
26. for element in list1:
27. heappush(heap, element)
29. ordered = []
31. while heap:
32. ordered.append(heappop(heap))
34. return ordered

37. car1 = Car("Renault", 2001)
38. car2 = Car("Bentley", 2005)
39. car3 = Car("Kia", 2014)
40. car4 = Car("Maruti Suzuki", 1999);
41. car5 = Car("Nano", 2012)
43. list1 = [car1, car2, car3, car4, car5]
45. for c in Heapsort Heapsort (list1):
46. print(c)

**Output:**

Model Name: Maruti Suzuki, Year: 1999

Model Name: Renault, Year: 2001

Model Name: Bentley, Year: 2005

Model Name: Nano, Year: 2012

Model Name: Kia, Year: 2014

We have sorted the objects on the year base.

Comparison between Heap sort and Other Algorithm

One of the popular quick sort algorithms is also very efficient, but heap sort is legally used because of its reliability. The heap sort's key benefit is **O(nlogn)** upper bound as far as the time complexity is fretful.

The heap sort algorithm takes O(nlogn) time in both average and worst-case scenarios while the quick sort is 20% faster in the average case.

The quick sort algorithm becomes slow in predictable situations. There is a chance of the security breach in quick sort since the foul O(n2) can be easily triggered.

Now we compare to the Merge sort, which takes the same time as the heap sort.

Merge sort is much stable and intuitively parallelizable, where heap sort doesn't have such advantages.

Furthermore, Merge sort is faster than the Heap Sort in most cases since they have the same time complexity.

In contrast, HeapsortHeapsort can be implemented much quickly in-place than Marge sort can.

Conclusion

Heapsort is not so popular and faster, but it is more predictable than any other sorting algorithm. This algorithm is preferred where memory and security are a priority.

It can be quickly implemented using Python. We require to insert the elements in a heap and take them out.

[**Next →**](https://www.javatpoint.com/program-of-cumulative-sum-in-python)[**← Prev**](https://www.javatpoint.com/heap-sort-in-python)

# Palindrome program in python language

## What is a palindrome?

A palindrome is a number or letter that remains the same even if the number and letters are inverted.

### For example:

121, 11, 414, 1221, 74747 are the palindrome numbers.

MOM, DAD, MADAM, REFER are the palindrome letters.

JAVATPOINT, PROGRAM, JAVA are not the palindrome letters.

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

### Palindrome algorithm

* Read the number or letter.
* Hold the letter or number in a temporary variable.
* Reverse the letter or number.
* Compare the temporary variable with reverses letter or number.
* If both letters or numbers are the same, print "this string/number is a palindrome."
* Else print, "this string/number is not a palindrome."

### Palindrome Program

**Program 1: Palindrome string**

1. str = 'JaVaJ'
2. strstr = str.casefold()
4. # This string is reverse.
5. rev = reversed(str)
7. if list(str) == list(rev):
8. print("PALINDROME !")
9. else:
10. print("NOT PALINDROME !")

**Output:**

PALINDROME !

**Program 2: Palindrome string program**

1. string=input(("Enter a letter:"))
2. if(string==string[::-1]):
3. print("The letter is a palindrome")
4. else:
5. print("The letter is not a palindrome")

**Output:**

Enter a letter: javatpoint

The letter is not a palindrome

Enter a letter: MADAM

The letter is a palindrome

**Program 3: Palindrome number program using while loop**

1. Num = int(input("Enter a value:"))
2. Temp = num
3. Rev = 0
4. while(num **>** 0):
5. dig = num % 10
6. revrev = rev \* 10 + dig
7. numnum = num // 10
8. if(temp == rev):
9. print("This value is a palindrome number!")
10. else:
11. print("This value is not a palindrome number!")

**Output:**

Enter the value: 2551

This value is not a palindrome number!

Enter the value: 1221

This value is a palindrome number!

[**Next →**](https://www.javatpoint.com/merge-sort-in-python)[**← Prev**](https://www.javatpoint.com/palindrome-program-in-python)

# Program of Cumulative sum in python

### What is the cumulative sum?

The cumulative sum means "how much so far". The definition of the cumulative sum is the sum of a given sequence that is increasing or getting bigger with more additions. The real example of a cumulative sum is the increasing amount of water in a swing pool.

**Example:**

1. Input: 10, 15, 20, 25, 30
2. Output: 10, 25, 45, 70, 100
4. Input: 1, 2, 3, 4, 5 6, 7 ,8, 9, 10
5. Output: 1, 3, 6, 10, 15, 21, 28, 36, 45, 55

### Program: 1

1. # Cumulative sum
2. def Cumulative\_sum(lists):
3. cum\_list = []
4. lenlength = len(lists)
5. cum\_list = [sum(lists[0:x:1]) for x in range(0, length+1)]
6. return cum\_list[1:]
8. lists = [10, 15, 20, 25, 30]
9. print (Cumulative\_sum(lists))

**Output:**

10, 25, 45, 70, 100

### Program: 2

1. list=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2. cum\_list=[]
3. y = 0
4. for x in range(0,len(list)):
5. y+=list[x]
6. cum\_list.append(y)
8. print(cum\_list)

**Output:**

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

1, 3, 6, 10, 15, 21, 28, 36, 45, 55

### Program 3: User define program

1. i = []
2. n = int(input("enter the no of elements in list:"))
3. for x in range(0,n):
4. element=int(input("enter the element" + str(x+1) + ":"))
5. i.append(element)
6. j=[sum(i[0:x+1]) for x in range(0,len(i))]
7. print("Original list is: ",i)
8. print("Cumulative sum list is: ",j)

**Output:**

enter the no of elements in list: 10

enter the element1: 2

enter the element2: 3

enter the element3: 8

enter the element4: 6

enter the element5: 45

enter the element6: 32

enter the element7: 56

enter the element8: 32

enter the element9: 14

enter the element10: 25

Original list is: [2, 3, 8, 6, 45, 32, 56, 32, 14, 25]

Cumulative sum list is: [2, 5, 13, 19, 64, 96, 152, 184, 198, 223]

# Merge Sort in Python

Merge sort is similar to the quick sort algorithm as works on the concept of divide and conquer. It is one of the most popular and efficient sorting algorithm. It is the best example for divide and conquer category of algorithms.

It divides the given list in the two halves, calls itself for the two halves and then merges the two sorted halves. We define the **merge()** function used to merging two halves.

The sub lists are divided again and again into halves until we get the only one element each. Then we combine the pair of one element lists into two element lists, sorting them in the process. The sorted two element pairs is merged into the four element lists, and so on until we get the sorted list.

## Merge Sort Concept

Let's see the following Merge sort diagram.

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

We have divided the given list in the two halves. The list couldn't be divided in equal parts it doesn't matter at all.

Merge sort can be implement using the two ways - top-down approach and bottom-up approach. We use the top down approach in the above example, which is Merge sort most often used.

The bottom-up approach provides the more optimization which we will define later.

The main part of the algorithm is that how we combine the two sorted sublists. Let's merge the two sorted merge list.

* A : [**2**, 4, 7, 8]
* B : [**1**, 3, 11]
* sorted : empty

First, we observe the first element of both lists. We find the B's first element is smaller, so we add this in our sorted list and move forward in the B list.

* A : [**2**, 4, 7, 8]
* B : [1, **3**, 11]
* Sorted : 1

Now we look at the next pair of elements 2 and 3. 2 is smaller so we add it into our sorted list and move forward to the list.

* A : [**2**, 4, 7, 8]
* B : [1, **3**, 11]
* Sorted : 1

Continue this process and we end up with the sorted list of {1, 2, 3, 4, 7, 8, 11}. There can be two special cases.

* What if both sublists have same elements - In such case, we can move either one sublist and add the element to the sorted list. Technically, we can move forward in both sublist and add the elements to the sorted list.
* We have no element left in one sublist. When we run out the in a sublist simply add the element of the second one after the other.

We should remember that we can sort the element in the any order. We sort the given list in ascending order but we can easily sort in descending order.

## Implementation

The merge sort algorithm is implemented by suing the top-down approach. It can be look slightly difficult, so we will elaborate each step in details. Here, we will implement this algorithm on two types of collections - integer element's list (typically used to introduce sorting) and a custom objects (a more practical and realistic scenario).

### Sorting Array

The main concept of algorithm is to divide (sub)list into halves and sort them recursively. We continue the process until we end up lists that have only one element. Let's understand the following function for division -

1. **def** merge\_sort(array, left\_index, right\_index):
2. **if** left\_index >= right\_index:
3. **return** middle = (left\_index + right\_index)//2
4. merge\_sort(array, left\_index, middle)
5. merge\_sort(array, middle + 1, right\_index)
6. merge(array, left\_index, right\_index, middle)

Our primary focus to divide the list into subparts before the sorting happen. We need to get the integer value so we use the // operator for our indices.

Let's understand the above procedure by following steps.

* First step is to create copies of lists. The first list contains the lists from **[left\_index,...,middle]** and the second from **[middle+1,?,right\_index]**.
* We traverse both copies of list by using the pointer, select the smaller value of the two values and add them to the sorted list. Once we add the element to the list and we move forward in the sorted list regardless.
* Add the remaining elements in the other copy to the sorted array.

Let's implement the merge sort in [Python program](https://www.javatpoint.com/python-programs).

### Python Program

1. # funtion to divide the lists in the two sublists
2. **def** merge\_sort(list1, left\_index, right\_index):
3. **if** left\_index >= right\_index:
4. **return**
6. middle = (left\_index + right\_index)//2
7. merge\_sort(list1, left\_index, middle)
8. merge\_sort(list1, middle + 1, right\_index)
9. merge(list1, left\_index, right\_index, middle)

12. # Defining a function for merge the list
13. **def** merge(list1, left\_index, right\_index, middle):

16. # Creating subparts of a lists
17. left\_sublist = list1[left\_index:middle + 1]
18. right\_sublist = list1[middle+1:right\_index+1]
20. # Initial values for variables that we use to keep
21. # track of where we are in each list1
22. left\_sublist\_index = 0
23. right\_sublist\_index = 0
24. sorted\_index = left\_index
26. # traverse both copies until we get run out one element
27. **while** left\_sublist\_index < len(left\_sublist) **and** right\_sublist\_index < len(right\_sublist):
29. # If our left\_sublist has the smaller element, put it in the sorted
30. # part and then move forward in left\_sublist (by increasing the pointer)
31. **if** left\_sublist[left\_sublist\_index] <= right\_sublist[right\_sublist\_index]:
32. list1[sorted\_index] = left\_sublist[left\_sublist\_index]
33. left\_sublist\_index = left\_sublist\_index + 1
34. # Otherwise add it into the right sublist
35. **else**:
36. list1[sorted\_index] = right\_sublist[right\_sublist\_index]
37. right\_sublist\_index = right\_sublist\_index + 1

40. # move forward in the sorted part
41. sorted\_index = sorted\_index + 1

44. # we will go through the remaining elements and add them
45. **while** left\_sublist\_index < len(left\_sublist):
46. list1[sorted\_index] = left\_sublist[left\_sublist\_index]
47. left\_sublist\_index = left\_sublist\_index + 1
48. sorted\_index = sorted\_index + 1
50. **while** right\_sublist\_index < len(right\_sublist):
51. list1[sorted\_index] = right\_sublist[right\_sublist\_index]
52. right\_sublist\_index = right\_sublist\_index + 1
53. sorted\_index = sorted\_index + 1
55. list1 = [44, 65, 2, 3, 58, 14, 57, 23, 10, 1, 7, 74, 48]
56. merge\_sort(list1, 0, len(list1) -1)
57. **print**(list1)

**Output:**

[1, 2, 3, 7, 10, 14, 23, 44, 48, 57, 58, 65, 74]

## Sorting Custom Objects

We can also sort the custom objects by using the [Python](https://www.javatpoint.com/python-tutorial) class. This algorithm is almost similar to the above but we need to make it more versatile and pass the comparison function.

We will create a custom class, Car and add a few fields to it. We make few changes in the below algorithm to make it more versatile. We can do this by using the lambda functions.

Let's understand the following example.

### Python Program

1. **class** Car:
2. **def** \_\_init\_\_(self, make, model, year):
3. self.make = make
4. self.model = model
5. self.year = year
7. **def** \_\_str\_\_(self):
8. **return** str.format("Make: {}, Model: {}, Year: {}", self.make, self.model, self.year)
10. **def** merge(list1, l, r, m, comp\_fun):
11. left\_copy = list1[l:m + 1]
12. r\_sublist = list1[m+1:r+1]
14. left\_copy\_index = 0
15. r\_sublist\_index = 0
16. sorted\_index = l
18. **while** left\_copy\_index < len(left\_copy) **and** r\_sublist\_index < len(r\_sublist):
20. # We use the comp\_fun instead of a simple comparison operator
21. **if** comp\_fun(left\_copy[left\_copy\_index], r\_sublist[r\_sublist\_index]):
22. list1[sorted\_index] = left\_copy[left\_copy\_index]
23. left\_copy\_index = left\_copy\_index + 1
24. **else**:
25. list1[sorted\_index] = r\_sublist[r\_sublist\_index]
26. r\_sublist\_index = r\_sublist\_index + 1
28. sorted\_index = sorted\_index + 1
30. **while** left\_copy\_index < len(left\_copy):
31. list1[sorted\_index] = left\_copy[left\_copy\_index]
32. left\_copy\_index = left\_copy\_index + 1
33. sorted\_index = sorted\_index + 1
35. **while** r\_sublist\_index < len(r\_sublist):
36. list1[sorted\_index] = r\_sublist[r\_sublist\_index]
37. r\_sublist\_index = r\_sublist\_index + 1
38. sorted\_index = sorted\_index + 1

41. **def** merge\_sort(list1, l, r, comp\_fun):
42. **if** l >= r:
43. **return**
45. m = (l + r)//2
46. merge\_sort(list1, l, m, comp\_fun)
47. merge\_sort(list1, m + 1, r, comp\_fun)
48. merge(list1, l, r, m, comp\_fun)
50. car1 = Car("Renault", "33 Duster", 2001)
51. car2 = Car("Maruti", "Maruti Suzuki Dzire", 2015)
52. car3 = Car("Tata motor", "Jaguar", 2004)
53. car4 = Car("Cadillac", "Seville Sedan", 1995)
55. list1 = [car1, car2, car3, car4]
57. merge\_sort(list1, 0, len(list1) -1, **lambda** carA, carB: carA.year < carB.year)
59. **print**("Cars sorted by year:")
60. **for** car **in** list1:
61. **print**(car)
63. **print**()
64. merge\_sort(list1, 0, len(list1) -1, **lambda** carA, carB: carA.make < carB.make)
65. **print**("Cars sorted by make:")
66. **for** car **in** list1:
67. **print**(car)

**Output:**

Cars sorted by year:

Make: Cadillac, Model: Seville Sedan, Year: 1995

Make: Renault, Model: 33 Duster, Year: 2001

Make: Tata motor, Model: Jaguar, Year: 2004

Make: Maruti, Model: Maruti Suzuki Dzire, Year: 2015

Cars sorted by make:

Make: Cadillac, Model: Seville Sedan, Year: 1995

Make: Maruti, Model: Maruti Suzuki Dzire, Year: 2015

Make: Renualt, Model: 33 Duster, Year: 2001

Make: Tata motor, Model: Jaguar, Year: 2004

## Optimization

We can improve the performance of the merge sort algorithm. First let's understand the difference between the top-down and bottom-up merge sort. The bottom-up approach sorts the elements of adjacent lists iteratively where the top-down approach breaks down the lists into the two halves.

The given list is [10, 4, 2, 12, 1, 3], instead of breaking it down into [10], [4], [2], [12], [1], [3] - we divides into the sublists which may already sorted: [10, 4], [2], [1, 12], [3] and now are ready to sort them.

Merge sort is inefficient algorithm in both time and space for the smaller sublists. So insertion sort is more efficient algorithm than the merge sort for the smaller sublists.

## Conclusion

Merge sort is popular and efficient algorithm. It is more efficient algorithm for the large lists. It doesn't depend on the any unfortunate decisions that lead to bad runtimes.

There is one major demerit in the merge sort. It uses the additional memory that is used to store the temporary copies of lists before merging them. However Merge sort is widely used in the software. Its performance is fast and produces the excellent result.

We have discussed the merge sort concept in brief and implement it both on simple integer list and on custom objects via a lambda function used for comparison.

# Python Matrix

In this tutorial, we will learn about Python matrices. In Python, a matrix object is similar to nested lists as they are multi-dimensional. We will see how to create a matrix using Numpy arrays. Following this, we will see various matrix operations methods and examples for better understanding.

## What is a Matrix in Python?

A matrix in Python is a rectangular Numpy array. This array has to be two-dimensional. It contains data stored in the array's rows and columns. In a Python matrix, the horizontal series of items are referred to as "rows," while the vertical series of items are referred to as "columns." The rows and columns are stacked over each other just like a nested list. If a matrix contains r number of rows and c number of columns, where r and c are positive integers, then r x c determines the order of this matrix object.

We can store strings, integers, and objects of other data types in a matrix. Data is stored in the stacks of rows and columns in a matrix. The matrix is a crucial data structure for calculations in mathematics and science. In Python, we consider a list of lists or a nested list as a matrix since Python doesn't include any built-in type for a matrix object.

In the course of this tutorial, we will go through the following list of matrix operation methods.

* Matrix addition
* Matrix multiplication
* Matrix multiplication operator
* Matrix multiplication without Numpy
* Matrix inverse
* Matrix transpose
* Matrix to array

## How do Matrices in Python work?

We write data in a two-dimensional array to create a matrix. It is done as follows:

**Example**

1. [ 2 3 5 7 6
2. 3 2 6 7 2
3. 5 7 2 6 1 ]

It displays a matrix having 3 rows and 5 columns, so its dimension is 3×5. Integer data type objects make up the data in this matrix. Row1, the first row, has values (2, 3, 5, 7, 6), while Row2 has values (3, 2, 6, 7, 2) and Row3 has values 5, 7, 2, 6, 1. Regarding columns, Column1 has values (2, 3, 5), Column2 has values (3, 2, 7), and so on.

**Example**

1. [ 0, 0, 1
2. 0, 1, 0
3. 1, 0, 0 ]

It displays a matrix having 3 rows and 3 columns, so its dimension is 3×3. Such matrices having equal rows and columns are called square matrices.

Similarly, Python allows users to store their data within an m x n dimensional matrix. We can perform the addition of matrices, multiplication, transposition, and other operations on a matrix-like structure.

The implementation of a matrix object in Python is not straightforward. We can create a Python matrix by using arrays and similarly use them.

## NumPy Array

The scientific computing software NumPy supports a robust N-dimensional array object. Installing NumPy is a prerequisite to using it in our program.

NumPy can be used and imported after installation. Knowing the basics of Numpy Array will be helpful in understanding matrices.

Arrays having multiple dimensions of items are provided by NumPy. Here's an illustration:

**Code**

1. # Python program to show how to create a Numpy array
3. # Importing numpy
4. **import** numpy as np
6. # Creating a numpy array
8. array = np.array([4, 6, "Harry"])
9. **print**(array)
10. **print**("Data type of array object: ", type(array))

**Output:**

['4' '6' 'Harry']

Data type of array object: <class 'numpy.ndarray'>

As we can see, Numpy arrays belong to the ndarray class.

### Example to Create a Matrix using Numpy Array

Think about the scenario where we create a record of students' marks. We will record the student's name and marks in two subjects, Python programming, and Matrix. We will create a two-dimensional matrix using a numpy array and then reshape it.

**Code**

1. # Python program to create a matrix using numpy array
3. # Importing numpy
4. **import** numpy as np
6. # Creating the matrix
7. record = np.array( [['Itika', 89, 91],
8. ['Aditi', 96, 82],
9. ['Harry', 91, 81],
10. ['Andrew', 87, 91],
11. ['Peter', 72, 79]])
13. matrix = np.reshape(record, (5,3))
14. **print**("The matrix is: \n", matrix)

**Output:**

The matrix is:

[['Itika' '89' '91']

['Aditi' '96' '82']

['Harry' '91' '81']

['Andrew' '87' '91']

['Peter' '72' '79']]

### Example to Create a Matrix using Numpy Matrix Method

We can use the numpy.matrix to create a 2D matrix.

**Code**

1. # Python program to show how to create a matrix using the matrix method
3. # importing numpy
4. **import** numpy as np
6. # Creating a matrix
7. matrix = np.matrix('3,4;5,6')
8. **print**(matrix)

**Output:**

[[3 4]

[5 6]]

## Accessing Values of a Matrix

The indices of a matrix can be used to access the elements stored in it. Data stored in a matrix is accessible using the same approach we use for a two-dimensional array.

**Code**

1. # Python program to access elements of a matrix
3. # Importing numpy
4. **import** numpy as np
6. # Creating the matrix
7. record = np.array( [['Itika', 89, 91],
8. ['Aditi', 96, 82],
9. ['Harry', 91, 81],
10. ['Andrew', 87, 91],
11. ['Peter', 72, 79]])
13. matrix = np.reshape(record, (5,3))
15. # Accessing record of Itika
16. **print**( matrix[0] )
18. # Accessing marks in the matrix subject of Andrew
19. **print**( "Andrew's marks in Matrix subject: ", matrix[3][2] )

**Output:**

['Itika' '89' '91']

Andrew's marks in Matrix subject: 91

## Methods to Create a 2-D Numpy Array or a Matrix

There are several methods to create a two-dimensional NumPy array and hence a matrix. Providing entries for the rows and columns

We can provide integers, floats, or even complex numbers. Using the dtype attribute of the array method, we can specify the data type we want.

**Code**

1. # Python program to show how to create a Numpy array
3. # Importing numpy
4. **import** numpy as np
6. # Creating numpy arrays
7. array1 = np.array([[4, 2, 7, 3], [2, 8, 5, 2]])
8. **print**("Array of data type integers: \n", array1)
10. array2 = np.array([[1.5, 2.2, 3.1], [3, 4.4, 2]], dtype = "float")
11. **print**("Array of data type float: \n", array2)
13. array3 = np.array([[5, 3, 6], [2, 5, 7]], dtype = "complex")
14. **print**("Array of data type complex numbers: \n", array3)

**Output:**

Array of data type integers:

[[4 2 7 3]

[2 8 5 2]]

Array of data type float:

[[1.5 2.2 3.1]

[3. 4.4 2. ]]

Array of data type complex numbers:

[[5.+0.j 3.+0.j 6.+0.j]

[2.+0.j 5.+0.j 7.+0.j]]

### Array having Zeros and Ones

**Code**

1. # Python program to show how to create a Numpy array having zeroes and ones
3. # Importing numpy
4. **import** numpy as np
6. # Creating numpy arrays
7. zeores\_array = np.zeros( (3, 2) )
8. **print**(zeores\_array)
10. ones\_array = np.ones( (2, 4), dtype=np.int64 )
11. **print**(ones\_array)

**Output:**

[[0. 0.]

[0. 0.]

[0. 0.]]

[[1 1 1 1]

[1 1 1 1]]

Here, we have specified dtype to 64 bits.

### Using the arange() and shape() Methods

**Code**

1. # Python program to show how to create Numpy array using arrange() and shape() methods
3. # Importing numpy
4. **import** numpy as np
6. # Creating numpy arrays
7. array1 = np.arange( 5 )
8. **print**(array1)
10. array2 = np.arange( 6 ).reshape( 2, 3 )
11. **print**(array2)

**Output:**

[0 1 2 3 4]

[[0 1 2]

[3 4 5]]

## Python Matrix Operations

### Python Matrix Addition

We will add the two matrices and use the nested for loop through the given matrices.

**Code**

1. # Python program to add two matrices without using numpy
3. # Creating matrices in the form of nested lists
4. matrix1 = [[23, 43, 12],
5. [43, 13, 55],
6. [23, 12, 13]]
8. matrix2 = [[4, 2, -1],
9. [5, 4, -34],
10. [0, -4, 3]]
12. matrix3  = [[0,1,0],
13. [1,0,0],
14. [0,0,1]]
16. matrix4  = [[0,0,0],
17. [0,0,0],
18. [0,0,0]]
20. matrices\_length = len(matrix1)
22. #Adding the three matrices using nested loops
23. **for** row **in** range(len(matrix1)):
24. **for** column **in** range(len(matrix2[0])):
25. matrix4[row][column] = matrix1[row][column] + matrix2[row][column] + matrix3[row][column]
26. #Printing the final matrix
27. **print**("The sum of the matrices is = ", matrix4)

**Output:**

The sum of the matrices is = [[27, 46, 11], [49, 17, 21], [23, 8, 17]]

### Python Matrix Multiplication

**Python Matrix Multiplication Operator**

In Python @ is known as the multiplication operator. Let us see an example where we will use this operator to multiply two matrices.

**Code**

1. # Python program to show how to create a matrix using the matrix method.
3. # importing numpy
4. **import** numpy as np
6. # Creating the matrices
7. matrix1 = np.matrix('3,4;5,6')
8. matrix2 = np.matrix('4,6;8,2')
10. # Usng multiplication operator to multiply two matrices
11. **print**(matrix1 @ matrix2)

**Output:**

[[44 26]

[68 42]]

### Python Matrix Multiplication without using Numpy

Another way of multiplying two matrices is using nested loops. Here is an example to show.

**Code**

1. # Python program to show how to create a matrix using the matrix method
3. # importing numpy
4. **import** numpy as np
6. # Creating two matrices
7. matrix1 = [[4, 6, 2],
8. [7, 4, 8],
9. [6, 2, 7]]
11. matrix2 = [[4, 6, 8, 2],
12. [6, 5, 3, 7],
13. [7, 3, 7, 6]]
14. # Result will be a 3x4 matrix
15. output = [[0,0,0,0],
16. [0,0,0,0],
17. [0,0,0,0]]
19. # Iterating through the rows of matrix1
20. **for** i **in** range(len(matrix1)):
21. # iterating through the columns of matrix2
22. **for** j **in** range(len(matrix2[0])):
23. # iterating through the rows of matrix2
24. **for** k **in** range(len(matrix2)):
25. output[i][j] += matrix1[i][k] \* matrix2[k][j]
27. **for** row **in** output:
28. **print**(row)

**Output:**

[66, 60, 64, 62]

[108, 86, 124, 90]

[85, 67, 103, 68]

### Python Matrix Inverse

When an equation needs to be solved to get the value of an unknown variable that satisfies the equations, the inverse of a matrix, which is just the reciprocal of the matrix as we would in regular mathematics, is calculated. A matrix's inverse is the matrix that gives the identity matrix when we multiply with the original matrix. Only a non-singular matrix can have an inverse. A non-singular matrix has a non-zero determinant.

**Code**

1. # Python program to show how to calculate the inverse of a matrix
3. # Importing the required library
4. **import** numpy as np
6. # Creating a matrix
7. A = np.matrix("3, 4, 6; 6, 2, 7; 6, 4, 6")
9. # Calculating the inverse of A
10. **print**(np.linalg.inv(A))

**Output:**

[[-3.33333333e-01 -7.40148683e-17 3.33333333e-01]

[ 1.25000000e-01 -3.75000000e-01 3.12500000e-01]

[ 2.50000000e-01 2.50000000e-01 -3.75000000e-01]]

### Python Matrix Transpose

**Python Matrix Transpose without Numpy**

A matrix's transposition involves switching the rows and columns. It has the symbol X'. We will put the object in row i and column j of matrix X in row j and column i of matrix X'. Consequently, X' will become a 4x3 matrix if the original matrix X is a 3x4 matrix.

**Code**

1. # Python program to find the transpose of a matrix using nested loops
3. # Creating a matrix
4. matrix = [[4, 6, 7, 8],
5. [3, 7, 2, 7],
6. [7, 3, 7, 5]]
8. result = [[0, 0, 0],
9. [0, 0, 0],
10. [0, 0, 0],
11. [0, 0, 0]]
13. # iterating through the rows
14. **for** i **in** range(len(matrix)):
15. # iterating through the columns
16. **for** j **in** range(len(matrix[0])):
17. result[j][i] = matrix[i][j]
19. **for** row **in** result:
20. **print**(row)

**Output:**

[4, 3, 7]

[6, 7, 3]

[7, 2, 7]

[8, 7, 5]

**Python Matrix Transpose using Numpy**

We can use the matrix.transpose() method in Numpy to get the matrix's transpose.

**Code**

1. # Python program to find the transpose of a matrix
3. # importing the required module
4. **import** numpy as np
6. # Creating a matrix using matrix method
7. matrix = np.matrix('[5, 7, 6; 4, 2, 4]')
9. #finding transpose using matrix.transpose method
10. transpose = matrix.transpose()
12. **print**(transpose)

**Output:**

[[5 4]

[7 2]

[6 4]]

### Converting Python Matrix to Array

We can use ravel and flatten functions to convert a Python matrix to a Python array.

**Code**

1. # Python program to convert a matrix to an array
3. # importing the required module
4. **import** numpy as np
6. # Creating a matrix using numpy
7. matrix = np.matrix("[4, 6, 7; 5, 2, 6; 6, 3, 6]")
9. # Using ravel() function to covert matrix to array
10. array = matrix.ravel()
11. **print**(array)
13. # Using flatten() function to covert matrix to array
14. array = np.asarray(matrix).flatten()
15. **print**(array)
17. # Using reshape() function to covert matrix to array
18. array = (np.asarray(matrix)).reshape(-1)
19. **print**(array)

**Output:**

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

[4 6 7 5 2 6 6 3 6]

[4 6 7 5 2 6 6 3 6]

# Python Unit Testing

In this tutorial, we will implement unit testing using the Python. Unit testing using Python is a huge topic itself, but we will cover a few essential concepts.

### What is the Python unittest?

[Unit testing](https://www.javatpoint.com/unit-testing) is a technique in which particular module is tested to check by developer himself whether there are any errors. The primary focus of unit testing is test an individual unit of system to analyze, detect, and fix the errors.

[Python](https://www.javatpoint.com/python-tutorial) provides the **unittest module** to test the unit of source code. The unittest plays an essential role when we are writing the huge code, and it provides the facility to check whether the output is correct or not.

Normally, we print the value and match it with the reference output or check the output manually.

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

This process takes lots of time. To overcome this problem, Python introduces the **unittest** module. We can also check the application's performance by using it.

We will learn how to create a basic test, finds the bugs, and execute it before the code delivers to the users.

## Testing the Code

We can test our code using many ways. In this section, we will learn the basic steps towards advanced methods.

### Automate vs. Manual Testing

[Manual testing](https://www.javatpoint.com/manual-testing) has another form, which is known as exploratory testing. It is a testing which is done without any plan. To do the manual testing, we need to prepare a list of the application; we enter the different inputs and wait for the expected output.

Every time we give the inputs or change the code, we need to go through every single feature of the list and check it.

It is the most common way of testing and it is also time-consuming process.

On the other hand, the automated testing executes the code according to our code plan which means it runs a part of the code that we want to test, the order in which we want to test them by a script instead of a human.

Python offers a set of tools and libraries which help us to create automated tests for the application.

## Unit Tests vs. Integration Tests

Suppose we want to check the lights of the car and how we might test them. We would turn on the light and go outside the car or ask the friend that lights are on or not. The turning on the light will **consider** as the test step, and go outside or ask to the friend will know as the **test assertion.** In the [integration testing](https://www.javatpoint.com/integration-testing), we can test multiple components at once.

These components can be anything in our code, such as functions, classes and module that we have written.

But there is a limitation of the integration testing; what if an integration test doesn't give the expected result. In this situation, it will be very hard to recognize which part of the system is falling. Let's take the previous example; if the light didn't turn on, the battery might be dead, blub is broken, car's computer have failed.

That's why we consider unit testing to get to know the exact problem in the tested code.

Unit testing is a smaller test, it checks a single component that it is working in right way or not. Using the unit test, we can separate what necessities to be fixed in our system.

We have seen the two types of testing so far; an integration test checks the multiple components; where unit test checks small component in or application.

Let's understand the following example.

We apply the unit testing [Python built-in function](https://www.javatpoint.com/python-built-in-functions) [**sum()**](https://www.javatpoint.com/python-sum-function) against the known output. We check that the sum() of the number **(2, 3, 5)** equals 10.

1. **assert** sum([ 2, 3, 5]) == 10, "Should be 10"

Above line will return the right result because values are correct. If we pass the wrong arguments it will return the **Assertion error**. For example -

1. **assert** sum([1, 3, 5]) == 10, "Should be 10"
2. Traceback (most recent call last):
3. File "<stdin>", line 1, **in** <module>
4. AssertionError: Should be 10

We can put the above code into the file and execute it again at the command line.

1. **def** test\_sum():
2. **assert** sum([2, 3, 5]) == 10, "It should be 10"
4. **if** \_\_name\_\_ == "\_\_main\_\_":
5. test\_sum()
6. **print**("Everything passed")

**Output:**

$ python sum.py

Everything is correct

In the following example, we will pass the tuple for testing purpose. Create a new file named **test\_sum2.py.**

**Example - 2:**

1. **def** test\_sum2():
2. **assert** sum([2, 3, 5]) == 10, "It should be 10"
4. **def** test\_sum\_tuple():
5. **assert** sum((1, 3, 5)) == 10, "It should be 10"
7. **if** \_\_name\_\_ == "\_\_main\_\_":
8. test\_sum2()
9. test\_sum\_tuple()
10. **print**("Everything is correct")

**Output:**

Everything is correct

Traceback (most recent call last):

File "<string>", line 13, in <module>

File "<string>", line 9, in test\_sum\_tuple

AssertionError: It should be 10

**Explanation -**

In the above code, we have passed the wrong input to the **test\_sum\_tuple().** The output is dissimilar to the predicted result.

The above method is good but what if there are multiple errors. Python interpreter would give an error immediately if the first error is encountered. To remove this problem, we use the test runners.

Test runner applications specially designed for testing the output, running test and give tools for fixing and diagnosing tests and applications.

### Choosing a Test Runner

Python contains many test runners. The most popular build-in Python library is called **unittest.** The unittest is portable to the other frameworks. Consider the following three top most test runners.

* unittest
* nose Or nose2
* pytest

We can choose any of them according to our requirements. Let's have a brief introduction.

### unittest

The unittest is built into the Python standard library since 2.1. The best thing about the unittest, it comes with both a test framework and a test runner. There are few requirements of the unittest to write and execute the code.

* The code must be written using the classes and functions.
* The sequence of distinct assertion methods in the **TestCase** class apart from the built-in asserts statements.

Let's implement the above example using the unittest case.

**Example -**

1. **import** unittest
2. **class** TestingSum(unittest.TestCase):
4. def test\_sum(self):
5. self.assertEqual(sum([2, 3, 5]), 10, "It should be 10")
6. def test\_sum\_tuple(self):
7. self.assertEqual(sum((1, 3, 5)), 10, "It should be 10")
9. **if** \_\_name\_\_ == '\_\_main\_\_':
10. unittest.main()

**Output:**

.F

-

FAIL: test\_sum\_tuple (\_\_main\_\_.TestingSum)

--

Traceback (most recent call last):

File "<string>", line 11, in test\_sum\_tuple

AssertionError: 9 != 10 : It should be 10

----------------------------------------------------------------------

Ran 2 tests in 0.001s

FAILED (failures=1)

Traceback (most recent call last):

File "<string>", line 14, in <module>

File "/usr/lib/python3.8/unittest/main.py", line 101, in \_\_init\_\_

self.runTests()

File "/usr/lib/python3.8/unittest/main.py", line 273, in runTests

sys.exit(not self.result.wasSuccessful())

SystemExit: True

As we can see in the output, it shows the **dot(.)** for the successful execution and **F** for the one failure.

### nose

Sometimes, we need to write hundreds or thousands of test lines for application; it becomes so difficult to understand.

The nose test runner can be a suitable replacement of the unittest test runners because it is compatible with any tests writing using the unittest framework. There are two types of nose - nose and nose2. We recommend using nose2 because it is a latest version.

Working with the nose2, we need to install it using the following command.

1. pip install nose2

Run the following command in the terminal to test the code using nose2.

1. python -m nose2

The output is as follows.

FAIL: test\_sum\_tuple (\_\_main\_\_.TestSum)

--

Traceback (most recent call last):

File "test\_sum\_unittest.py", line 10, in test\_sum\_tuple

self.assertEqual(sum((2, 3, 5)), 10, "It should be 10")

AssertionError: It should be 10

--

Ran 2 tests in 0.001s

FAILED (failures=1)

The nose2 provides many command line flags for filtering the test. You can learn more from its official documentation.

### pytest

The pytest test runner supports the execution of **unittest** test cases. The actual benefit of the **pytest** is to writing **pytest** test cases. The pytest test cases are generally sequence of methods in the Python file starting.

The pytest provides the following benefits -

* It supports the built-in assert statement instead of using a special **assert\*()** methods.
* It also provides support for cleaning for test cases.
* It can rerun from the last cases.
* It has an ecosystem of hundreds of plugin to extend the functionality.

Let's understand the following example.

**Example -**

1. **def** test\_sum():
2. **assert** sum([2, 3, 5]) == 10, "It should be 10"
3. **def** test\_sum\_tuple():
4. **assert** sum((1, 2, 5)) == 10, "It should be 10"

## Writing the First Test

Here we will apply all the concepts that we have learned in earlier section. First, we need to create a file name **test.py** or anything. Then make inputs and execute the code being tested, capturing the output. After successfully run the code, match the output with an expected result.

First, we create the file **my\_sum** file and write code in it.

1. **def** sum(arg):
2. total = 0
3. **for** val **in** arg:
4. total += val
5. **return** total

We initialized the total variable which iterates over all the values in arg.

Now, we create a file name **test.py** with the following code.

**Example -**

1. **import** unittest
3. from my\_sum **import** sum

6. **class** CheckSum(unittest.TestCase):
7. def test\_list\_int(self):
9. data = [1, 2, 3]
10. result = sum(data)
11. self.assertEqual(result, 6)
13. **if** \_\_name\_\_ == '\_\_main\_\_':
14. unittest.main()

**Output:**

.

----------------------------------------------------------------------

Ran 1 test in 0.000s

OK

**Explanation:**

In the above code, we imported **sum()** from the **my\_sum package** that we created. We have defined the **Checkclass,** which inherits from **unittest.TestCase.** There is a test methods - **.test\_list\_int(),** to test the integer.

After running the code, it returns **dot(.)** which means there is no error in the code.

Let's understand another example.

**Example - 2**

1. **class** Person:
2. name1 = []
4. def set\_name(self, user\_name):
5. self.name1.append(user\_name)
6. **return** len(self.name1) - 1
8. def get\_name(self, user\_id):
9. **if** user\_id >= len(self.name1):
10. **return** ' No such user Find'
11. **else**:
12. **return** self.name1[user\_id]

15. **if** \_\_name\_\_ == '\_\_main\_\_':
16. person = Person()
17. print('Peter Decosta has been added with id ', person.set\_name('Peter'))
18. print('The user associated with id 0 is ', person.get\_name(0))

**Output:**

Peter Decosta has been added with id 0

The user associated with id 0 is Peter

## Python Basic Functions and Unit Test Output

The unittest module produces three possible outcomes. Below are the potential outcomes.

1. **OK -** If all tests are passed, it will return OK.
2. **Failure -** It will raise an **AssertionError** exception, if any of tests is failed.
3. **Error -** If any errors occur instead of Assertion error.

Let's see the following basic functions.

|  |  |
| --- | --- |
| **Method** | **Description** |
| .assertEqual(a, b) | a == b |
| .assertTrue(x) | bool(x) is True |
| .assertFalse(x) | bool(x) is False |
| .assertIs(a, b) | a is b |
| .assertIsNone(x) | x is None |
| .assertIn(a, b) | a in b |
| .assertIsInstance(a, b) | isinstance(a, b) |
| .assertNotIn(a, b) | a not in b |
| .assertNotIsInstance(a,b) | not isinstance(a, b) |
| .assertIsNot(a, b) | a is not b |

### Python Unit Test Example

1. **import** unittest
3. # First we **import** the **class** which we want to test.
4. **import** Person1 as PerClass
6. **class** Test(unittest.TestCase):
7. """
8. The basic **class** that inherits unittest.TestCase
9. """
10. person = PerClass.Person()  # instantiate the Person Class
11. user\_id = []  # This variable stores the obtained user\_id
12. user\_name = []  # This variable stores the person name
14. # It is a test **case** function to check the Person.set\_name function
15. def test\_0\_set\_name(self):
16. print("Start set\_name test\n")
18. **for** i in range(4):
19. # initialize a name
20. name = 'name' + str(i)
21. # put the name into the list variable
22. self.user\_name.append(name)
23. # extraxt the user id obtained from the function
24. user\_id = self.person.set\_name(name)
25. # check **if** the obtained user id is **null** or not
26. self.assertIsNotNone(user\_id)
27. # store the user id to the list
28. self.user\_id.append(user\_id)
29. print("The length of user\_id is = ", len(self.user\_id))
30. print(self.user\_id)
31. print("The length of user\_name is = ", len(self.user\_name))
32. print(self.user\_name)
33. print("\nFinish set\_name test\n")
35. # Second test **case** function to check the Person.get\_name function
36. def test\_1\_get\_name(self):
37. print("\nStart get\_name test\n")
39. # total number of stored user information
40. length = len(self.user\_id)
41. print("The length of user\_id is = ", length)
42. print("The lenght of user\_name is = ", len(self.user\_name))
43. **for** i in range(6):
44. # **if** i not exceed total length then verify the returned name
45. **if** i < length:
46. # **if** the two name not matches it will fail the test **case**
47. self.assertEqual(self.user\_name[i], self.person.get\_name(self.user\_id[i]))
48. **else**:
49. print("Testing for get\_name no user test")
50. # **if** length exceeds then check the 'no such user' type message
51. self.assertEqual('There is no such user', self.person.get\_name(i))
52. print("\nFinish get\_name test\n")

55. **if** \_\_name\_\_ == '\_\_main\_\_':
56. # begin the unittest.main()
57. unittest.main()

**Output:**

Start set\_name test

The length of user\_id is = 4

[0, 1, 2, 3]

The length of user\_name is = 4

['name0', 'name1', 'name2', 'name3']

Finish set\_name test

Start get\_name test

The length of user\_id is = 4

The lenght of user\_name is = 4

Testing for get\_name no user test

.F

======================================================================

FAIL: test\_1\_get\_name (\_\_main\_\_.Test)

----------------------------------------------------------------------

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/multiprocessing.py", line 502, in test\_1\_get\_name

self.assertEqual('There is no such user', self.person.get\_name(i))

AssertionError: 'There is no such user' != ' No such user Find'

- There is no such user

+ No such user Find

----------------------------------------------------------------------

Ran 2 tests in 0.002s

FAILED (failures=1)

## Advance Testing Scenario

We must follow the given step while creating test for the application.

* Generate necessary input
* Execute the code, taking the output.
* Match the output with an expected result.

Creating inputs such as static value for the input like a string or numbers is a slightly complex task. Sometimes, we need to create an instance of a class or a context.

The input data that we create is known as a fixture. We can reuse fixtures in our application.

When we run the code repeatedly and pass the different values each time and expecting the same result, this process is known as **parameterization.**

## Handling Expected Failures

In the earlier example, we pass the integer number to test **sum(); what** happens if we pass the bad value, such as a single integer or a string?

The **sum()** will throw an error as expected. It would happen due to failed test.

We can use the **.assertRaises()** to handle the expected errors. It is used inside **with** statement. Let's understand the following example.

**Example -**

1. **import** unittest
2. from my\_sum **import** sum
4. **class** CheckSum(unittest.TestCase):
5. def test\_list\_int(self):
7. #  Test that it can sum a list of integers
9. data = [1, 2, 3]
10. res = sum(data)
11. self.assertEqual(res, 6)
13. def test\_bad\_type(self):
14. data = "Apple"
15. with self.assertRaises(TypeError):
16. res = sum(data)
18. **if** \_\_name\_\_ == '\_\_main\_\_':
19. unittest.main()

**Output:**

..

----------------------------------------------------------------------

Ran 2 tests in 0.006s

OK

## Python unittest Skip Test

We can skip an individual test method or **TestCase** using the skip test technique. The fail will not count as a failure in TestResult.

Consider the following example to skip the method unconditionally.

**Example -**

1. **import** unittest
3. def add(x,y):
4. c = x + y
5. **return** c
7. **class** SimpleTest(unittest.TestCase):
8. @unittest.skip("The example skipping method")
9. def testadd1(self):
10. self.assertEquals(add(10,5),7)
12. **if** \_\_name\_\_ == '\_\_main\_\_':
13. unittest.main()

**Output:**

s

----------------------------------------------------------------------

Ran 1 test in 0.000s

OK (skipped=1)

**Explanation:**

In the above example, the **skip()** method prefixed by the @token. It takes the one argument a log message where we can describe the reason for skip. The **s** character denotes that a test has been successfully skipped.

We can skip a particular method or block based on the specific condition.

**Example - 2:**

1. **import** unittest

4. **class** suiteTest(unittest.TestCase):
5. a = 100
6. b = 40
8. def test\_add(self):
9. res = self.a + self.b
10. self.assertEqual(res, 100)
12. @unittest.skipIf(a > b, "Skip because a is greater than b")
13. def test\_sub(self):
15. res = self.a - self.b
16. self.assertTrue(res == -10)
18. @unittest.skipUnless(b == 0, "Skip because b is eqaul to zero")
19. def test\_div(self):
21. res = self.a / self.b
22. self.assertTrue(res == 1)
24. @unittest.expectedFailure
25. def test\_mul(self):
27. res = self.a \* self.b
28. self.assertEqual(res == 0)

31. **if** \_\_name\_\_ == '\_\_main\_\_':
32. unittest.main()

**Output:**

Fsx.

======================================================================

FAIL: test\_add (\_\_main\_\_.suiteTest)

----------------------------------------------------------------------

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/multiprocessing.py", line 539, in test\_add

self.assertEqual(res, 100)

AssertionError: 50 != 100

----------------------------------------------------------------------

Ran 4 tests in 0.001s

FAILED (failures=1, skipped=1, expected failures=1)

**Explanation:**

As we can see in the output, the conditions b == 0 and a>b is true so the **test\_mul()** method has skipped. On the other hand, **test\_mul** has been marked as an expected failure.

## Conclusion

We have discussed the all-important concept related to Python unit testing. As a beginner, we need to write the smart, maintainable methods to validate our code. Once we get a decent command over the Python unit test, we can switch to other frameworks such as **the pytest** and leverage more advanced features.

# Python Forensics and Virtualization | Hash Functions

In this tutorial, we will learn the Forensics science using Python, basic Python forensics applications, Hash functions, Cracking an Encryption, Visualization, Naming Conventions, Dshell and Scapy, Network Forensics with its detailed explanation.

## Introduction

Collecting and preserving evidence is most essential for cyber forensic investigation and analysis at the computer devices. It plays important role in a court room to be used against the criminal. Nowadays, technology facilitates us to get the information by just typing the query on the browser. But it also invites the cyber crooks. Cyber crooks are those who perform the malicious activity by using their system and internet. They can get your all information from sitting somewhere else.

With its wide applications, Python also provides the facility to work with the digital forensics. By using it, we can gather data, extract evidence, and also encrypt password. It will support us to reinstate the reliability of evidence.

Before go further, you must familiar with the [Python](https://www.javatpoint.com/python-tutorial) and its advance concepts.

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

## Introduction to Computational Forensics

Computational Forensics is a part of study which used to solve problems in various forensics disciplines. It uses computer-based modeling, analysis, computer simulation and recognition. Python Forensics was invented by the **Chet Homster**. There are also pattern evidence, such as fingerprints, shoeprints, toolmarks and any documents. It makes use of procedures, scope of objects, and substances. There are also physiological and behavioral patterns such as digital evidence, [DNA](https://www.javatpoint.com/dna-full-form), and crime scenes.



We can also use the various algorithms to deal with the signal and image processing. By using algorithms, we can also handle the, [data mining](https://www.javatpoint.com/data-mining), [computer graphics](https://www.javatpoint.com/computer-graphics-tutorial), [machine learning](https://www.javatpoint.com/machine-learning), computer vision data visualization, and statistical pattern recognition.

In a few words, the computation forensics is used to study the digital evidence, computational forensics deals with the various types of evidence.

## Naming Conventions for Python Forensics Application

We must familiar with the naming convention and patterns to follow the Python Forensics guidelines. Consider the following table.

|  |  |  |
| --- | --- | --- |
|  | **Naming Convention** | **Example** |
|  |  |  |
| Local variables | camelCase with optional underscore | studentName |
| Constant | Uppercase, words separated by underscores | STUDENT\_NAME |
| Global variable | Prefix with camelCase with optional underscores | my\_studentName |
| Function | PascalCase with optional underscores; active voice | MystudentName |
| Module | Prefix with the camel case | \_studentname |
| Class | Prefix class with Pascalcase; keep it sort | class\_MyStudentName |
| Object | Prefix ob\_with camelcase | ob\_studentName |

The hashing algorithm is one of the best ways of take as an input a stream of binary data. In the real life scenario, we can encrypt our password, file, or even any kinds of digital file or data. The algorithm takes an input and generates the encrypted message. Let's see the given example.

### Example

1. **import** sys,string,md5
2. print("Enter the name")
3. line=sys.stdin.readline()
4. line=line.rstrip()
5. md5\_object=md5.**new**()
6. md5\_object.update(line)
7. print(md5\_object.hexdigest())

## Python Hash Function

Python hash function is used to map a vast amount of data to a fixed value. An input returns the same output. It is a hash sum and stores features with precise information. Once we map the data to a fixed value, that cannot be revert. That's why we also refer it as one-way cryptographic algorithm.

Let's understand the following example -

**Example -**

1. **import** hashlib
2. **import** uuid
4. def hash\_pass(password):
5. s = uuid.uuid4().hex
6. **return** hashlib.sha256(s.encode() + password.encode()).hexdigest() + ':' + s
7. def verify\_password(hashed\_password, user\_password):
8. password, s = hashed\_password.split(':')
9. **return** password == hashlib.sha256(s.encode() + user\_password.encode()).hexdigest()
10. new\_password = input('Enter your password :')
12. hashed\_password = hash\_pass(new\_password)
13. print('The hash string to store in the db is: ' + hashed\_password)

**Output:**

Enter your password: sharma

The hash string to store in the db is: 947782bdb0c7a5ad642f1f26179b6aef2d9857427b45a09af4fce3b8f1346e91:8a8371941513482487e5ab8af2ae6466

Now, we will re-enter the password.

1. old\_password = input('Enter new password ')
2. **if** verify\_password(hashed\_password, old\_password):
3. print(' Entered password is correct')
4. **else**:
5. print('Passwords do not match')

**Output:**

Enter your password devansh

The hash string to store in the db is: 4762866edd3b49c7736163ef3d981e42629a09a9ca7e081f56d116e137d77b9c:ebbf5b16bd9f4b989505a495bf7ae9b9

Enter new password sharma

Passwords do not match

The hash function has the following properties.

* We can simply transform any hash value for any input value.
* It doesn't able to produce the same output as given hash value.
* It is unrealistic to transform the input without moving the hash value.

## Cracking an Encryption in Python

We must know how to encrypt the text data that we fetch during analysis and evidence. First, understand the basic cryptographic.

Generally, secret messages are sent by the army person to convey their plans without get read by their enemies. These messages are not in the human-readable format. The plain texts are encrypted by using the encryption algorithm and these texts are called cipher text.

Suppose a general commander sends a message to senior to save the text from their enemies. Here, we take shift the plain text letter four place in the alphabet. Now, the A will be E, each B is F and so no.

Let's understand the following example to crack the vector data.

**Example -**

1. **import** sys
2. def decryption(text,cipher):
3. simple\_text=''
4. **for** each in cipher:
5. x = (ord(each)-text) % 126
6. **if** x < 32:
7. x+=95
8. simple\_text += chr(x)
9. print(simple\_text)
10. cipher\_text = input('Enter the message: ')
11. **for** i in range(1,95,1):
12. decryption(i,cipher\_text)

**Output:**

Enter message: Yes

~

}

|

{

z

y

x

w

v

u

t

s

r

r~

q

q}

p

p|

o

o{

n

nz

m

my

l

lx

k

kw

j

jv

i

iu

h

ht

g

gs

f

fr

e

eq

d

dp

dp~

c

co

co}

b

bn

bn|

a

am

am{

`

`l

`lz

\_

\_k

\_ky

j

jx

i

iw

h

hv

g

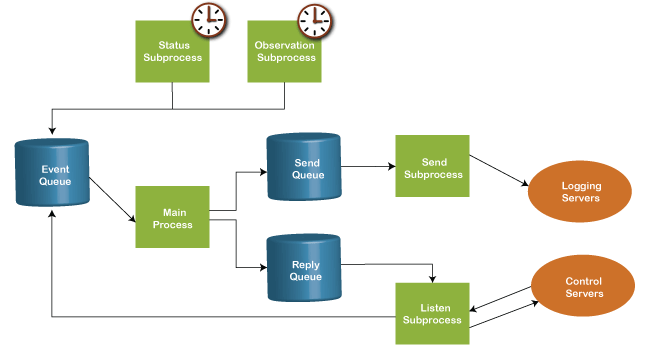
gu

f

ft

## Virtualization

A virtualization is an act of emulate IT system such as workstations, networks and storage. We make the virtual instance of such a resource. It can be done with the help of hypervisor.



The virtualization of hardware plays very important role in the computer forensics. By using the virtualization, we can get following advantages.

* We can use the workstation in a validate state for each investigation.
* We can recover deleted data by including **dd** images of a drive on a virtual machine.
* The virtual machine can turn into the recovery device that will help to gather evidence.

We define the following steps to create virtual machine using Python

**Step - 1:** Suppose we consider our local machine as **"dummy"**. Each Virtual Machine will have at least 512 MB of memory.

1. virmach\_memory = 512 \* 1024 \* 1024

**Step - 2:** Now, we attach this virtual machine to the default cluster.

1. virmach\_cluster = api.clusters.get(name = "Default")

**Step - 3:** Next, boot the virtual machine from the virtual HDD.

1. vm\_os = params.OperatingSystem(boot = [params.Boot(dev = "hd")])

Now, we will combine the above steps into a virtual machine parameter object. Let's understand the following example.

**Example -**

1. from ovirtsdk.xml **import** params
2. from ovirtsdk.api **import** API

5. **try**:
6. # We need to provide Api credentials **for** virtual machine
7. api = API(url="https://HOST",
8. username="Example",
9. password="example123",
10. ca\_file="ca.crt")
12. virmach\_name = "dummy"
13. virmach\_memory = 512 \* 1024 \* 1024  # calculating the memory in bytes
14. virmach\_cluster = api.clusters.get(name="Default")
15. virmach\_template = api.templates.get(name="Blank")
17. # here we are assigning the parameters to operating system
18. virmach\_os = params.OperatingSystem(boot=[params.Boot(dev="hd")])
20. virmach\_params = params.VM(name=virmach\_name,
21. memory=virmach\_memory,
22. cluster=virmach\_cluster,
23. template=virmach\_template,
24. os = virmach\_os)
26. **try**:
27. api.vms.add(vm=virmach\_params)
28. print("Virtual machine '%s' added successfully." % virmach\_name)
29. except Exception as ex:
30. print("Adding virtual machine '%s' failed: %s" % (virmach\_name, ex))
31. api.disconnect()
33. except Exception as ex:

**Output:**

Virtual Machine dummy added successfully.

## Network Forensics in Python

Python also provide the facility to work with the network forensics. In the modern days, Python network forensics environment investing can come across many difficulties. These problems can be responding to a breach report, executing assessments pertaining to susceptibility, or validating regularity compliances. Let's understand the basic terminology of network forensics.

**Client -** The client runs personal computer and workstation.

**Server -** The server executes the client's request.

**Protocols -** Protocols are the set of rule that must be followed while data transfer.

**Websockets -** A websockets are protocol that provides the full-duplex communication and runs over the TCP connection. We can send the bi-direction messages using the websockets.

With the help of those protocols, we can authenticate the information and sent or received by the third party users. But, encryption is necessary to secure channels.

Let's understand the following example of network

**Example -**

1. **import** socket
2. # creating a socket object
3. sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)
4. # getting local machine name
5. host = socket.gethostname()
6. port = 8080
7. # connection to hostname on the port.
8. sock.connect((host, port))
9. # Receive no more than 1024 bytes
10. temp = sock.recv(1024)
11. print("The client waits for connection")
12. sock.close()

**Output:**

The client waits for connection

## Python Scapy and Dshell

Let's understand the brief introduction Python Scapy and Dshell.

### Python Scapy

A **scapy** is Python-based tool which analyze and manipulate network traffic. With the help of scapy, we can analyze packet manipulation. We can also capture and decode the packets of a wide number of protocols. The benefit of using scapy is to get the detailed report about network traffic to the investigator. The third-party tools such as OS fingerprint app can be also used in Scapy. Let's understand the following example.

**Example -**

1. #Imports scapy and GeoIP toolkit
2. **import** scapy, GeoIP
3. from scapy **import** \*
5. geoIp = GeoIP.**new**(GeoIP.GEOIP\_MEMORY\_CACHE)
6. def locatePackage(pkg):
7. # extracts the source IP address
8. source = pkg.getlayer(IP).src
9. # extracts the destination IP address
10. destination = pkg.getlayer(IP).dst
11. # gets Country details of source
12. srcCountry = geoIp.country\_code\_by\_addr(source)
13. dstCountry = geoIp.country\_code\_by\_addr(destination)
14. # gets country details of destination
15. print src+"("+sourceCountry+") >> "+destination+"("+destinationcountry+")\n"

**Output:**

source INDIA >> destination USA

### Python Dshell

The Dshell is a Python-based network forensics analysis toolkit. It was developed by the US army research laboratory and released it open-source in 2014. It makes the forensics investigation very easy. Dshell provides the following decoders.

* **reservedips -** It is used to identify solutions for the DNS problems.
* **rip-http -** It extracts the files from HTTP traffic.
* **large-flows -** It is a decoder that represents the list net flows.
* **Protocols -** It identifies the non-standard protocols.
* **dns -** It extracts DNS-related queries.

## Python Searching

Searching is the most important part of the forensics investigation. Nowadays, the good search is upon the investigator who is running the evidence. Keyword searching from the message is a pillar of the investigation. We can find the strong evidence with the help of a keyword.

The experience and knowledge both are required to get the information from the deleted messages.

Python provides the various built-in modules to support search operation. The investigator can find the result using the keywords such as "who", "what", "where", "when", "which", etc. Let's understand the following example.

**Example -**

1. # Searching a particular word from a message
2. str1 = "This is an example for Computational forensics of gathering evidence!"
3. str2 = "string"
5. print(str1.find(str2))
6. print(str1.find(str2, 10))
7. print(str1.find(str2, 40))

**Output:**

11

11

-1

## Python Indexing

Indexing is feature that the investigator can use to gather potential evidence from the files. The evidence can be restricted within the memory snapshot, a disk image, a file, or a network trace. It is very helpful to reduce time for time-consuming tasks like keyword searching. The indexing also used to locate the keywords in interactive searching phase. In the following example, we have explained indexing in Python.

**Example -**

1. list1 = [123, 'example', 'creative', 'indexing']
3. print("Index example : ", list1.index('example'))
4. print("Index for indexing : ", list1.index('indexing'))
6. str1 = "This is a message for forensic investigation indexing"
7. str2 = "message"
9. print("Index of the character keyword found is ")
10. print(str1.index(str2))

**Output:**

Index example : 1

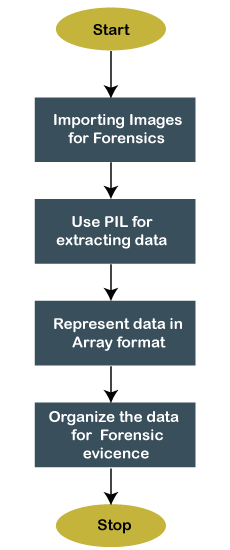
Index for indexing : 3

Index of the character keyword found is

10

## Python Image Library

The real meaning of forensics investigation is to extract the valuable information from the available resources. Getting all the relevant information from the resource is essential for the report. It helps us to derive appropriate result.



Resource data can be either simple data structure such as databases or complex data structures such as JPEG image.

Investigator can easily access the information from the simple data structure but extracting information from the complex data structure is tedious task.

Python provides the Image library which is known as PIL. It is used to add image processing capabilities to out Python interpreter. It also support the file formats, graphics capabilities and also provides powerful image processing. Let's understand the following image to extracting data from images.

We define the programming example to explain how it actually works.

**Step - 1:** Suppose we have a following image where we need to extract the details.



**Step - 2:** An image consists of various pixel values. The PIL library uses to extract the image details for gather evidence. Let's understand the following example.

**Example -**

1. from PIL **import** Image
2. im = Image.open('penguin.jpeg', 'r')
3. pix\_val = list(im.getdata())
4. pix\_val\_flat = [x **for** sets in pix\_val **for** x in sets]
5. print(pix\_val\_flat)

**Output:**

[255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255]

The output is returned in the form of a list. It is a pixel value of the RGB combination that gives a better picture of what data is needed.

## Python Multiprocessing Support

Forensics experts find difficulties to apply digital solutions to large digital evidence on the common crime. Most of the digital evidences are the single threaded that mean we can execute only one command at time. Let's see the brief [introduction of multiprocessing](https://www.javatpoint.com/python-multiprocessing).

### Multiprocessing

Multiprocessing is an ability of the system that support more than one process. It enables the several programs to run concurrently. There are two types of the multiprocessing - **symmetric and asymmetric processing**.

Let's understand the following example of multiprocessing.

**Example -**

1. **import** random
2. **import** multiprocessing

5. def list\_append(count, id, out\_list):
6. # count number of process at a time
7. **for** i in range(count):
8. out\_list.append(random.random())
10. **if** \_\_name\_\_ == "\_\_main\_\_":
11. size = 810
12. procs = 2
13. jobs = []
15. **for** i in range(0, procs):
16. out\_list = list()  # list of processes
17. process1 = multiprocessing.Process(
18. target=list\_append, args=(size, i, out\_list))
20. # appends the list of processes
21. jobs.append(process1)
23. # Calculate the random number of processes
24. **for** j in jobs:
25. j.start()  # initiate the process
27. # After the processes have finished execution
28. **for** j in jobs:
29. j.join()
30. print("List processing complete.")

**Output:**

List processing complete

## Mobile Forensics in Python

Forensics investing is not only limited to the standard computer hardware such as hard disk, CPUs, etc. Hardware is followed with the help of techniques to analyze non-standard hardware or transient evidence.

Nowadays, smartphones are widely used in digital investigation, but they still meant as non-standard. With the proper research of smartphones, we can extract photos, smartphones, and messages.

The android smartphones uses the PIN, or alphanumeric password. The password can be between 4 and 16 digits/characters.

In the following example, we will get through a lock screen to extract data. The smartphone password generally stores inside a file **password.key** in **/data/system**.

Android stores a salted SHA1-hashsum and MD5-hashsum of this password. Let's see the following example.

**Example -**

1. **public** **byte**[] passwordToHash(String password) {
2. **if** (password == **null**) {
3. **return** **null**;
4. }
5. String algo = **null**;
6. **byte**[] hashed = **null**;
7. **try** {
8. **byte**[] saltedPassword = (password + getSalt()).getBytes();
9. **byte**[] sha1 = MessageDigest.getInstance(algo = "SHA-1").digest(saltedPassword);
10. **byte**[] md5 = MessageDigest.getInstance(algo = "MD5").digest(saltedPassword);
11. hashed = (toHex(sha1) + toHex(md5)).getBytes();
12. } **catch** (NoSuchAlgorithmException e) {
13. Log.w(TAG, "Failed to encode string because of missing algorithm: " + algo);
14. }
15. **return** hashed;
16. }

The above code is a sample code of crack smartphone password. The dictionary attack won't be affected to crack the password since hashed password is stored in a **salt file**. The salt file is a string of hexadecimal representation of a random integer of 64 bit. The Rooted smartphones or JTAG Adapter can access the **salt** file.

## Rooted Smartphones

The file's dump /data/system/password.key is stored in SQLite database under the lock **screen.password\_salt.** The Password is stored under **settings.db**.

## JTAG Adapter

The JTAG stands for Joint Test Action Group which can be used to access the salt. Similarly, a Riff-Box or a JIG-Adapter can be used to access the sale files. We can find the position of the encrypted data using the obtained information from Riff-box. The rules are given below.

* Find the associated string "**password\_salt**".
* The width of the salt file represents in the bytes. This is its length.
* This is the length which is actually searched to get the stored password/pin of the smartphones.

## Memory and Forensics

Python forensics primarily focuses on the volatile memory with the help of Volatility which is a Python based framework.

### Volatile Memory

Volatile memory is a type of memory that erased when the system's power is turned off or interrupted. In the simple words, if we are working on a document that has not been saved to the hard disk and suddenly the power goes off, we will lose our data.

The volatile memory follows the same pattern as the other forensics investigations.

* First, it needs to be selected the target of the investing.
* Acquire the forensics data.
* Forensics Analysis

The [RAM](https://www.javatpoint.com/ram-full-form) dump is tool which used to analysis the gathered data from the [RAM](https://www.javatpoint.com/ram).

### YARA Rules

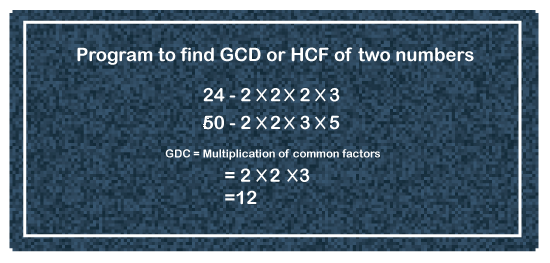
YARA is a tool which used to examine the suspected files/ directories and match strings. It is based on the pattern matching implementation. It plays an important role in forensics analysis.

**Example -**

1. **import** operator
2. **import** os
3. **import** sys
5. sys.path.insert(0, os.getcwd())
6. **import** plyara.interp as interp
8. # Plyara is a script that lexes and parses a file consisting of one more Yara
10. **if** \_\_name\_\_ == '\_\_main\_\_':
11. file\_to\_analyze = sys.argv[1]
12. Dictrules = interp.parseString(open(file\_to\_analyze).read())
13. authors = {}
14. imps = {}
15. meta\_keys = {}
16. max\_strings = []
17. max\_string\_len = 0
18. tags = {}
19. rule\_count = 0
21. **for** rule in Dictrules:
22. rule\_count += 1
24. # Imports
25. **if** 'imports' in rule:
26. **for** imp in rule['imports']:
27. imp = imp.replace('"', '')
29. **if** imp in imps:
30. imps[imp] += 1
31. **else**:
32. imps[imp] = 1
33. # Tags
34. **if** 'tags' in rule:
35. **for** tag in rule['tags']:
36. **if** tag in tags:
37. tags[tag] += 1
38. **else**:
39. tags[tag] = 1
41. # Metadata
42. **if** 'metadata' in rule:
43. **for** key in rule['metadata']:
44. **if** key in meta\_keys:
45. meta\_keys[key] += 1
46. **else**:
47. meta\_keys[key] = 1
49. **if** key in ['Author', 'author']:
50. **if** rule['metadata'][key] in authors:
51. authors[rule['metadata'][key]] += 1
52. **else**:
53. authors[rule['metadata'][key]] = 1
55. # Strings
56. **if** 'strings' in rule:
57. **for** strr in rule['strings']:
58. **if** len(strr['value']) > max\_string\_len:
59. max\_string\_len = len(strr['value'])
60. max\_strings = [(rule['rule\_name'], strr['name'], strr['value'])]
61. elif len(strr['value']) == max\_string\_len:
62. max\_strings.append((rule['rule\_name'], strr['key'], strr['value']))
64. print("\nThe number of rules implemented" + str(rule\_count))
65. ordered\_meta\_keys = sorted(meta\_keys.items(), key=operator.itemgetter(1),
66. reverse = True)
67. ordered\_authors = sorted(authors.items(), key=operator.itemgetter(1),
68. reverse = True)
69. ordered\_imps = sorted(imps.items(), key=operator.itemgetter(1), reverse=True)
70. ordered\_tags = sorted(tags.items(), key=operator.itemgetter(1), reverse=True)

# GCD of two number in python

**Greatest Common Divisor** (GCD) is a mathematical term to find the greatest common factor that can perfectly divide the two numbers. A GCD is also known as the **Highest Common Factor (HCF)**. For example, the HCF/ GCD of two numbers 54 and 24 is 6. Because 6 is the largest common divisor that completely divides 54 and 24.



## GCD Using gcd() Function

In [python](https://www.javatpoint.com/python-tutorial), a gcd() is an inbuilt function offered by the math module to find the greatest common divisor of two numbers.

**Syntax**

1. gcd(a, b)

Where a and b are the two integer number passes as an argument to the function gcd().

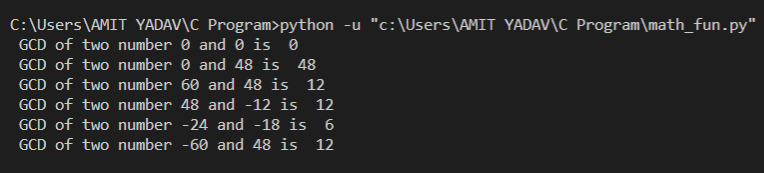
Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

Let's create a program to print the GCD of two number using the inbuilt function of math.gcd() in python.

**math\_fun.py**

1. # create a program to print the gcd of two number in python using the math.gcd() function.
2. **import** math
3. print(" GCD of two number 0 and 0 is ", math.gcd(0, 0)) #math.gcd(a, b), a and b are the two integer number
4. print(" GCD of two number 0 and 48 is ", math.gcd(0, 48))
5. a = 60 # assign the number to variable a
6. b = 48 # assign the number to variable b
7. print(" GCD of two number 60 and 48 is ", math.gcd(a, b)) # pass the variable a and b to math.gcd() function.
8. print(" GCD of two number 48 and -12 is ", math.gcd(48, -12)) # pass the integer number
9. print(" GCD of two number -24 and -18 is ", math.gcd(-24, -18))
10. print(" GCD of two number -60 and 48 is ", math.gcd(-60, 48))

**Output:**



In the above example, math.gcd() function generates the GCD of two given numbers. In the gcd() function, a and b pass as an argument that returns the greatest common divisor of two integer numbers, completely dividing the numbers.

## GCD Using recursion

Recursion is a memory consuming function defined in python that calls itself via self-referential expression. It means that the function will continuously call and repeat itself until the defined condition is met to return the greatest common divisor of the number.

### Pseudo Code of the Algorithm

Step 1: Take two inputs, x and y, from the user.

Step 2: Pass the input number as an argument to a recursive function.

Step 3: If the second number is equal to zero (0), it returns the first number.

Step 4: Else it recursively calls the function with the second number as an argument until it gets the remainder, which divides the second number by the first number.

Step 5: Call or assign the gcd\_fun() to a variable.

Step 6: Display the GCD of two numbers.

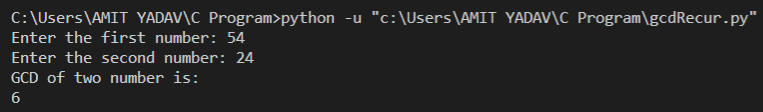
Step 7: Exit from the program.

Let's understand the program to find the GCD of two number using the recursion.

**gcdRecur.py**

1. # write a program to understand the GCD of two number in python using the recursion.
2. def gcd\_fun (x, y):
3. **if** (y == 0): # it divide every number
4. **return** x  # **return** x
5. **else**:
6. **return** gcd\_fun (y, x % y)
7. x =**int** (input ("Enter the first number: ")) # take first no.
8. y =**int** (input ("Enter the second number: ")) # take second no.
9. num = gcd\_fun(x, y) # call the gcd\_fun() to find the result
10. print("GCD of two number is: ")
11. print(num) # call num

**Output:**



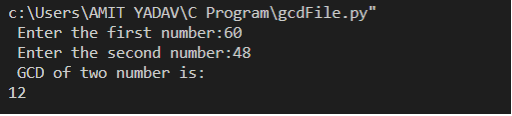
## GCD Using the Loop

Let's create program to find the GCD of two number in python using the loops.

**gcdFile.py**

1. def GCD\_Loop( a, b):
2. **if** a > b:  # define the **if** condition
3. temp = b
4. **else**:
5. temp = a
6. **for** i in range(1, temp + 1):
7. **if** (( a % i == 0) and (b % i == 0 )):
8. gcd = i
9. **return** gcd
10. x = **int**(input (" Enter the first number: ") ) # take first no.
11. y =**int** (input (" Enter the second number: ")) # take second no.
12. num = GCD\_Loop(x, y) # call the gcd\_fun() to find the result
13. print("GCD of two number is: ")
14. print(num) # call num

**Output:**



As we can see in the above program, we take two values as input and pass these numbers to the GCD\_Loop () function to return a GCD.

## GCD Using Euclid's algorithm or Euclidean Algorithm

Euclid's algorithm is an efficient method to find the greatest common divisor of two numbers. It is the oldest algorithm that divides the greater number into smaller ones and takes the remainder. Again, it divides the smaller number from the remainder, and this algorithm continuously divides the number until the remainder becomes 0.

For example, suppose we want to calculate the H.C.F of two numbers, 60 and 48. Then we divide the 60 by 48; it returns the remainder 12. Now we again divide the number 24 by 12, and then it returns the remainder 0. So, in this way, we get the H.C.F is 12.

### Pseudo Code of the Euclid Algorithm

Step 1: There are two integer numbers, such as a and b.

Step 2: if a = 0, then the GCD(a, b) is b.

Step 3: if b = 0, the GCD(a, b) is a.

Step 4: a mod b find the

Step 5: Suppose a = b and b = R

Step 6: Repeat steps 4 and 3 until a mod b is equal or greater than 0.

Step 7: GCD = b and then print the result.

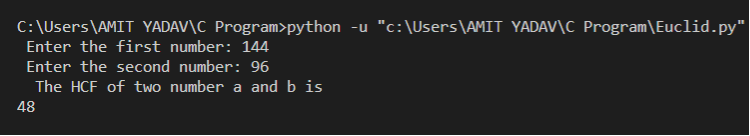
Step 8: Stop the program.

Let's find the H.C.F or GCD of two numbers using Euclid's algorithm in python.

**Euclid.py**

1. # Create a program to find the GCD of two number in python using the Euclid's Algorithm.
2. def find\_hcf(a,b):
3. **while**(b):
4. a, a = b, a % b
5. **return** a
6. a = **int**(input (" Enter the first number: ") ) # take first no.
7. b = **int**(input (" Enter the second number: ")) # take second no.
8. num = find\_hcf(a, b) # call the find\_hcf() to get the result
9. print("  The HCF of two number a and b is ")
10. print(num) # call num

**Output:**



# Python Program to generate a Random String

A random refers to the collection of data or information that can be available in any order. The [**random** module in python](https://www.javatpoint.com/python-random-module) is used to generate random strings. The random string is consisting of numbers, characters and punctuation series that can contain any pattern. The random module contains two methods **random.choice()** and **secrets.choice()**, to generate a secure string. Let's understand how to generate a random string using the random.choice() and secrets.choice() method in [python](https://www.javatpoint.com/python-tutorial).



## Using random.choice()

The **random.choice()** function is used in the [python string](https://www.javatpoint.com/python-strings) to generate the sequence of characters and digits that can repeat the string in any order.

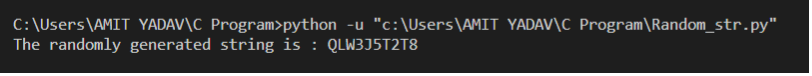
**Create a program to generate a random string using the random.choices() function.**

**Random\_str.py**

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

1. **import** string
2. **import** random # define the random module
3. S = 10  # number of characters in the string.
4. # call random.choices() string module to find the string in Uppercase + numeric data.
5. ran = ''.join(random.choices(string.ascii\_uppercase + string.digits, k = S))
6. print("The randomly generated string is : " + str(ran)) # print the random data

**Output:**



Following are the method used in the random module to generate the random string.

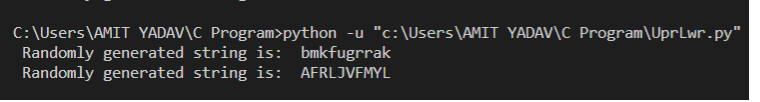
|  |  |
| --- | --- |
| **Methods** | **Description** |
| **String.ascii\_letters** | It returns a random string that contains both uppercase and lowercase characters. |
| **String\_ascii\_uppercase** | It is a random string method that only returns a string in uppercase characters. |
| **String.ascii\_lowercase** | It is a random string method that returns a string only in lowercase characters. |
| **String.digits** | It is a random string method that returns a string with numeric characters. |
| **String.punctuation** | It is a random string method that returns a string with punctuation characters. |

**Generate a random string of upper case and lower-case letters**

**UprLwr.py**

1. # write a program to generate the random string in upper and lower **case** letters.
2. **import** random
3. **import** string
4. def Upper\_Lower\_string(length): # define the function and pass the length as argument
5. # Print the string in Lowercase
6. result = ''.join((random.choice(string.ascii\_lowercase) **for** x in range(length))) # run loop until the define length
7. print(" Random string generated in Lowercase: ", result)
9. # Print the string in Uppercase
10. result1 = ''.join((random.choice(string.ascii\_uppercase) **for** x in range(length))) # run the loop until the define length
11. print(" Random string generated in Uppercase: ", result1)
13. Upper\_Lower\_string(10) # define the length

**Output:**

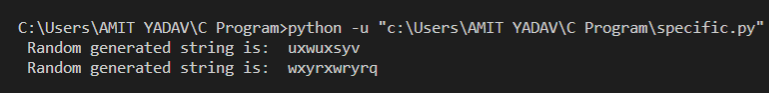


### Random String of Specified Characters

**Specific.py**

1. # create a program to generate the random string of given letters.
2. **import** random
3. **import** string
4. def specific\_string(length):
5. sample\_string = 'pqrstuvwxy' # define the specific string
6. # define the condition **for** random string
7. result = ''.join((random.choice(sample\_string)) **for** x in range(length))
8. print(" Randomly generated string is: ", result)
10. specific\_string(8) # define the length
11. specific\_string(10)

**Output:**



#### Note: The random.choice() method is used in the python program to repeat the same characters strings. If we don't want to display repetitive characters, we should use random.sample() function.

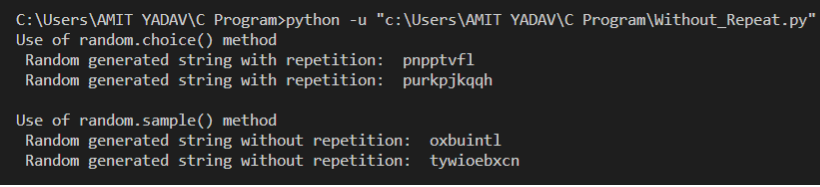
**Generate a random string without repeating the same characters**

**WithoutRepeat.py**

1. # create a program to generate a string with or without repeating the characters.
2. **import** random
3. **import** string
4. print("Use of random.choice() method")
5. def specific\_string(length):
7. letters = string.ascii\_lowercase # define the lower **case** string
8. # define the condition **for** random.choice() method
9. result = ''.join((random.choice(letters)) **for** x in range(length))
10. print(" Random generated string with repetition: ", result)
12. specific\_string(8) # define the length
13. specific\_string(10)

16. print("") # print the space
17. print("Use of random.sample() method")
18. def WithoutRepeat(length):
19. letters = string.ascii\_lowercase # define the specific string
20. # define the condition **for** random.sample() method
21. result1 = ''.join((random.sample(letters, length)))
22. print(" Random generated string without repetition: ", result1)
24. WithoutRepeat(8) # define the length
25. WithoutRepeat(10)

**Output:**



As we can see in the above output, the random.sample() method returns a string in which all characters are unique and non-repeating. Whereas, the random.choice() method returns a string that may contain repetitive characters. So, we can say that if we want to generate a unique random string, use **random.sample**() method.

### Generate a random alphanumeric string consisting of fixed letters and digits

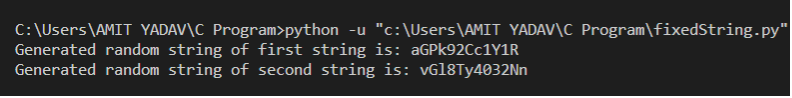
For example, suppose we want a randomly generated alphanumeric string that contains five letters and four digits. We need to define these parameters into the function.

Let's write a program to generate an alphanumeric string that contains a fixed number of letters and digits.

**fixedString.py**

1. **import** random
2. **import** string
3. def random\_string(letter\_count, digit\_count):
4. str1 = ''.join((random.choice(string.ascii\_letters) **for** x in range(letter\_count)))
5. str1 += ''.join((random.choice(string.digits) **for** x in range(digit\_count)))
7. sam\_list = list(str1) # it converts the string to list.
8. random.shuffle(sam\_list) # It uses a random.shuffle() function to shuffle the string.
9. final\_string = ''.join(sam\_list)
10. **return** final\_string
12. # define the length of the letter is eight and digits is four
13. print("Generated random string of first string is:", random\_string(8, 4))
15. # define the length of the letter is seven and digits is five
16. print("Generated random string of second string is:", random\_string(7, 5))

**Output:**



## Using secrets.choice()

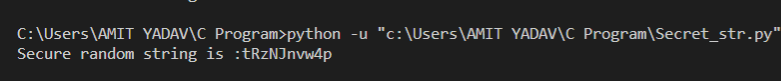
A secrets.choice() method is used to generate a more secure random string than random.choice(). It is a cryptographically random string generator that ensures no two processes can obtain the same results simultaneously using secrets.choice() method.

Let's write a program to print a secure random string using the secrets.choice method.

**Secret\_str.py**

1. **import** random
2. **import** string
3. **import** secrets # **import** **package**
4. num = 10 # define the length of the string
5. # define the secrets.choice() method and pass the string.ascii\_letters + string.digits as an parameters.
6. res = ''.join(secrets.choice(string.ascii\_letters + string.digits) **for** x in range(num))
8. # print the Secure string
9. print("Secure random string is :"+ str(res))

**Output:**



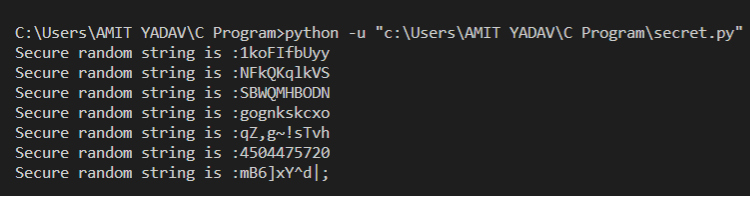
### Use the different method of the random module to generate a safe random string.

Let's write a program to print secure random strings using different methods of secrets.choice().

**Secret.py**

1. # write a program to display the different random string method using the secrets.choice().
2. # imports necessary packages
3. **import** random
4. **import** string
5. **import** secrets
6. num = 10 # define the length of the string
7. # define the secrets.choice() method and pass the string.ascii\_letters + string.digits as an parameters.
8. res = ''.join(secrets.choice(string.ascii\_letters + string.digits) **for** x in range(num))
9. # Print the Secure string with the combination of ascii letters and digits
10. print("Secure random string is :"+ str(res))
12. res = ''.join(secrets.choice(string.ascii\_letters) **for** x in range(num))
13. # Print the Secure string with the combination of ascii letters
14. print("Secure random string is :"+ str(res))
16. res = ''.join(secrets.choice(string.ascii\_uppercase) **for** x in range(num))
17. # Print the Secure string in Uppercase
18. print("Secure random string is :"+ str(res))
20. res = ''.join(secrets.choice(string.ascii\_lowercase) **for** x in range(num))
21. # Print the Secure string in Lowercase
22. print("Secure random string is :"+ str(res))
24. res = ''.join(secrets.choice(string.ascii\_letters + string.punctuation) **for** x in range(num))
25. # Print the Secure string with the combination of letters and punctuation
26. print("Secure random string is :"+ str(res))
28. res = ''.join(secrets.choice(string.digits) **for** x in range(num))
29. # Print the Secure string using string.digits
30. print("Secure random string is :"+ str(res))
32. res = ''.join(secrets.choice(string.ascii\_letters + string.digits + string.punctuation) **for** x in range(num))
33. # Print the Secure string with the combonation of letters, digits and punctuation
34. print("Secure random string is :"+ str(res))

**Output:**



# Yield Keywords in Python

Python includes many tools that vastly simplify the life of programmers. One such tool is the yield keyword in Python. We can use this keyword in place of return statements in regular Python procedures. We will learn about the yield keyword, its application in generator functions, the difference between a return statement and a yield statement, and when we can replace a return statement with a yield statement.

## What is the Yield Keyword in Python?

The yield keyword of Python is comparable to another keyword called return, which we use to return an expression or object, usually in functions. There is, although, a little variation. Instead of just returning a value to the call of the function that includes the yield statement, the yield statement of a function returns a generator object.

When we call a function in the program with a yield statement, the function's execution suspends when the Python interpreter encounters a yield statement. The caller receives an object from the generator class. To put it another way, the yield keyword will transform any expression supplied with it into a generator object and then return that generator object to the caller. Therefore, we must iterate through the generator object to obtain the values.

Please be aware that a function using the term yield is called a generator function. The yield keyword won't ruin the states of the local variables. The latest yield expression will be used as the starting point for the execution every time a function is called.

Every time we call a function with a return statement, a fresh set of variables is introduced. The function's execution will resume where it left off if a generator function is used in place of a regular function.

We can use the combination of generator function and yield keyword to return multiples function through a function. The yield expression returns multiple values. They return a single value, pause the execution, save the current local state of the variable declared, and then continue.

### Syntax of the yield Keyword in Python

1. **def** function():
2. **yield** < expression > # writing an yield statement
3. **print**( function )

## Brief on Generator Functions in Python

Generator functions in Python are those that, unlike the regular functions that return a single expression, return an iterable object, which we call a generator object. Using a basic Python loop, or the methods like the next() or list(), or the variables stored within a generator object one at a time, we can approach or read the values from the generator function. We created a generator function using the def keyword. A good example is provided below.

### Example of yield Keyword with a Generator Function

We will print certain strings when we call the functions.

**Code**

1. # Python program to show the use of yield keyword with the help of a generator function
3. # Defining a generator function
4. **def** generator\_func():
5. **yield** "Yield"
6. **yield** "Keyword"
7. **yield** "in"
8. **yield** "Python"
10. # calling the generator function and creating a generator object
11. generator\_object = generator\_func()
12. **print**( type(generator\_object) ) # Printing the type of generator object
13. **for** i **in** generator\_object:
14. **print**( i )

**Output:**

<class 'generator'>

Yield

Keyword

in

Python

In the program mentioned above, we made a basic generator function, and after using several yield expressions to return several values, Python then saved within a generator object we made. The values inside this generator function can then be displayed on the console using a loop over the object.

### Filtering Odd Numbers Using yield Keywords

Let's build a new generator function that uses the yield keyword. We'll attempt to eliminate every odd number from a list of integers. Additionally, it is crucial in this situation to output the information held inside the generator object using various methods like list(), for-in, and next().

Look at the example below.

**Code**

1. # Python program to show the use of yield keyword with the help of a generator function
3. # Defining a generator function
4. **def** filtering\_odd( numbers ):
5. **for** num **in** range( numbers ):
6. **if** num % 2 != 0:
7. **yield** num
8. # Calling the generator function, passing a vale and declaring a generator object.
9. odd\_num = filtering\_odd( 34 )
10. **print**(list( odd\_num ))

**Output:**

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33]

## How to Call Functions which have yield?

We can call functions in place of returning values with yield. Imagine, for instance; we define a function square() which returns the square of a number given to it on execution. This code has a return keyword. There is one more function square() with a yield keyword to give squares of a range of values. In this situation, we may develop a straightforward program by combining the yield expression with the square() function. Examine the example code in the section below.

**Code**

1. # Python program to show how to call a function with yield
3. # defining a function with a return keyword
4. **def** square( number ):
5. **return** number\*\*2
7. # defining a function with yield keyword
8. **def** range\_square( range\_of\_numbers ):
9. **for** num **in** range(range\_of\_numbers):
10. **yield** square( num )
12. # calling the function and passing values
13. squared = range\_square( 8 )
14. **print**( list( squared ) )

**Output:**

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

## How return Keyword is Different from yield Keyword

The Python Yield keyword works similarly to the return keyword used for values, except that rather than only providing a value in return of the function, the generator function that includes the yield keyword inside it returns a generator object. Their key distinction is the return keyword of Python returns the value, which stops the function's execution. The yield keyword, however, delays the function's execution. Statements after return keywords are never performed, which is another distinction. When the function continues its execution, yield statements are carried out.

### When should One Use yield Instead of return Keywords?

The yield keyword pauses the function's execution and returns a value while keeping a sufficient state for the generator function to pick up from where yield paused it. When execution is continued, the function starts again right where the previous yield execution ended. Instead of calculating values simultaneously and returning them as a list data type in output, this enables its program to output a succession of values throughout time.

**Advantages of Using yield:**

* Memory allocation load is kept within check because it retains local variable states.
* Because the former state is kept, the process doesn't have to start over, which reduces time.

**Disadvantages of Using yield:**

* We need to call the function correctly. Otherwise, using yield can occasionally become incorrect.
* Coding complexity increases due to time and storage optimization, making the reasoning behind it occasionally difficult to comprehend.

Let us look at an illustrative example:

**Code**

1. # Python program to show the difference between yield keyword and return keyword
3. # defining a function with return keyword
4. **def** square\_root( number ):
5. **return** number\*\*(1/2)
7. # defining a function with yield keyword
8. **def** generator\_square\_root( number ):
9. **for** i **in** range(number):
10. **yield** i\*\*(1/2)
12. # Calling the functions and passing values to them
13. **print**( "Function with return keyword: ", square\_root( 9 ) )
14. **print**( "Function with yield keyword: ", list(generator\_square\_root( 9 )) ) # converting generator object to list object

**Output:**

Function with return keyword: 3.0

Function with yield keyword: [0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.23606797749979, 2.449489742783178, 2.6457513110645907, 2.8284271247461903]

While yield might generate a series of results, the return keyword returns a specific value to its caller. If we want to iterate through a series but don't want to keep the complete series in memory, we must use yield.

Python generator functions use yield return output. When a generator function is defined using the def keyword, and it has to create a value, it uses the yield keyword instead of the return keyword. When the yield is included in a def's body, the function transforms into a generator function.

## List of Differences between yield Keywords and return Keywords

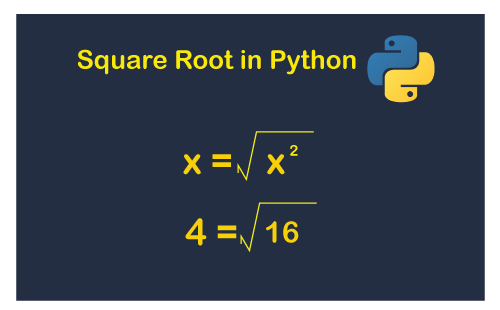
Here are a few distinctions between Python yield and return.

|  |  |  |
| --- | --- | --- |
| **Point of Difference** | **Yield Keyword** | **Return Keyword** |
| **Definition** | When we call a generator function, it creates a generator object by combining all the values returned by the yield keyword. Additionally, code begins to run only after the caller iterates through the object. | The function execution ends as soon as it hits the return statement, returning just one value to us. |
| **Function** | The initial yield is executed, and the function terminates when we call the generator function. The generator object, which contains the value given by yield, is then returned to us. The subsequent yield statement is then executed, and the cycle is repeated when we have to access or iterate over this variable. | A regular function's execution starts when we call it and finishes when it has reached the return statement. The value is then returned to us. |
| **Number of Keywords per Function** | In a generator function, we can use many yield statements. | In a typical function, we can use just one return statement. |
| **Storage** | When using yield keywords, no storage space is allocated. | Memory is allotted for every returned value. |
| **Application** | Exceptionally memory-efficient, particularly when working with huge data sets. | We should use this keyword on smaller data sets. |
| **Execution** | Using the yield keyword speeds up execution times for huge data sets. | Larger data sizes result in longer execution times because additional computation is required. |

# How to write square root in Python?

Python has a predefined **sqrt()** function that returns the square root of a number. It defines the square root of a value that multiplies itself to give a number. The **sqrt()** function is not used directly to find the square root of a given number, so we need to use a **math** module to call the **sqrt()** function in [Python](https://www.javatpoint.com/python-tutorial).

For example, the square root of 144 is 12.



## Using math.sqrt() method

The sqrt() function is an inbuilt function that returns the square root of any number. Following are the steps to find the square root of a number.

1. Start the program
2. Define any number whose square root to be found.
3. Invoke the **sqrt()** function and pass the value that you have defined in step 2 and store the result in a variable.
4. Print the square root.
5. Terminate the program.

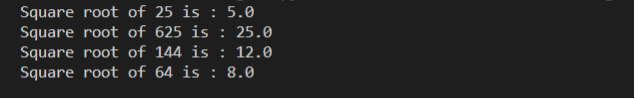
Let's create a Python program to find the square root of a number.

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

**SqrRoot.py**

1. **import** math # **import** math module
2. N = 25 # define the value to the variable N
3. result = math.sqrt(N) # use math.sqrt() function and pass the variable.
4. print(" Square root of 25 is :", result) # prints the square root of a given number
5. M = 625 # define the value
6. result = math.sqrt(M) # use math.sqrt() function and pass the variable
7. print(" Square root of 625 is :", result) # prints the square root of a given number
9. P = 144 # define the value
10. result = math.sqrt(P) # use math.sqrt() function and pass the variable
11. print(" Square root of 144 is :", result) # prints the square root of a given number
13. S = 64 # define the value
14. result = math.sqrt(S) # use math.sqrt() function and pass the variable
15. print(" Square root of 64 is :", result) # prints the square root of a given number

**Output:**

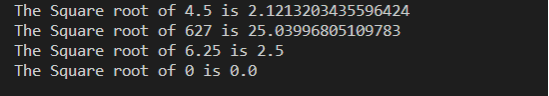


Let's create a python program that finds the square root of a decimal numbers.

**SqrRoot.py**

1. **import** math
2. print(" The Square root of 4.5 is", math.sqrt(4.5)) # Pass the decimal number
3. print(" The Square root of 627 is", math.sqrt(627)) # Pass the decimal number
4. print(" The Square root of 6.25 is", math.sqrt(6.25)) # Pass the decimal number
6. print(" The Square root of 0 is", math.sqrt(0)) # Pass number as 0

**Output:**

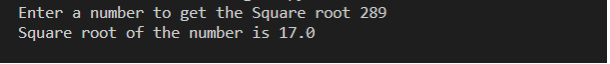


In the following program, we have read a number form the user and find the square root.

**SqRoot\_Usr.py**

1. **import** math # **import** math module
2. a = **int**(input("Enter a number to get the Square root")) # take an input
3. res = math.sqrt(a) # Use math.sqrt() function and pass the variable a.
4. print("Square root of the number is", res) # print the Square Root

**Output:**



## Using math.pow() function

The **pow()** is an inbuilt function that is used in Python to return the power of a number. It has two parameters. The first parameter defines the number and second parameter defines the power raise to that number.

**Pow\_Sqrt.py**

1. **import** math # **import** the math module
2. num = **float**(input("Enter the number :")) # take an input
3. SquareRoot = math.pow(num, 0.5) # Use the math.pow() function and pass the value and 0.5 (which is equal to √) as an parameters
4. print(" The Square Root of the given number {0} = {1}" .format(num, SquareRoot)) # print the Square Root.

**Output:**

How to write square root in Python

## Using \*\* Operator

We can also use the exponent operator to find the square root of the number. The operator can be applied between two operands. For example, x\*\*y. It means that left operand raised to the power of right.

Following are the steps to find the square root of a number.

**Step 1**. Define a function and pass the value as an argument.

**Step 2**. If the defined number is less than 0 or negative, it returns nothing.

**Step 3**. Use the exponential \*\* sign to find the power of a number.

**Step 4**. Take the numeric value from the user.

**Step 5**. Call the function and store its output to a variable.

**Step 6**. Display the Square Root of a number in Python.

**Step 7**. Exit from the program.

Let's implement the above steps in a Python program and calculate the square root of a number.

**SqrtFun.py**

1. **import** math # **import** the math **package** or module
2. def sqrt\_fun(num): # define the sqrt\_fun() and pass the num as an argument
3. **if** num < 0:  # **if** num is less than 0 or negative, it returns nothing
4. **return**
5. **else**:
6. **return** num \*\* 0.5 # Use the exponent operator
7. num = **int**(input (" Enter a numeric value: ") ) # take an input from the user
9. res = sqrt\_fun(num) # call the sqrt\_fun() to find the result
10. print("  Square Root of the {0} = {1}".format(num, res)) # print the Square Root of the variable

**Output:**

How to write square root in Python

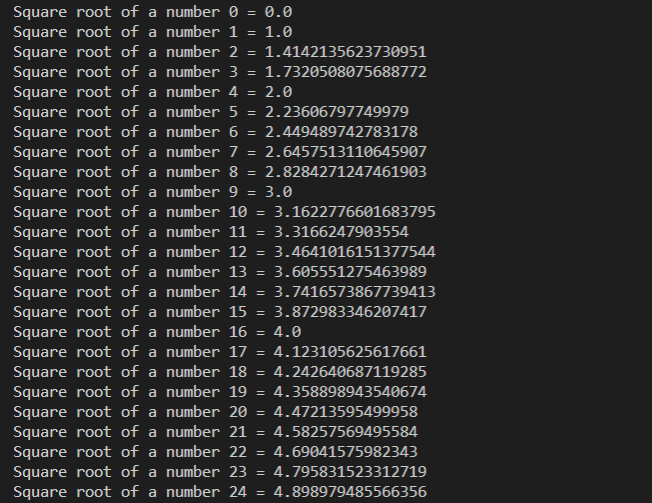
As we can see in the above example, first we take an input (number) from the user and then use the exponent \*\* operator to find out the power of a number. Where 0.5 is equal to √ (root symbol) to raise the power of a given number.

Let's create a Python program that finds the square root of between the specified range. In the following program, we have found the square root of all the number between 0 to 50.

**Sqrloop.py**

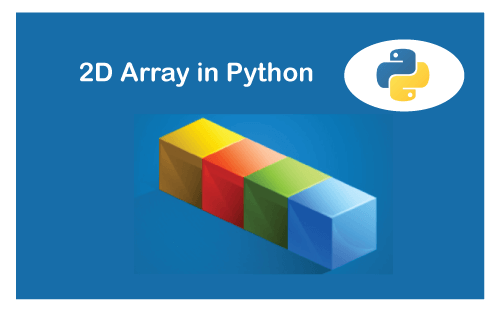
1. **import** math
2. **for** i in range(50):
3. print("Square root of a number {0} = {1}".format(i,math.sqrt(i)))

**Output:**



# Python 2D array

**An array** is a collection of linear data structures that contain all elements of the same data type in contiguous memory space. It is like a container that holds a certain number of elements that have the same data type. An array's index starts at 0, and therefore, the programmer can easily obtain the position of each element and perform various operations on the array. In this section, we will learn about 2D (two dimensional) arrays in Python.



### Two-Dimensional Array (2D Array)

**A 2D array** is an array of arrays that can be represented in matrix form like rows and columns. In this array, the position of data elements is defined with two indices instead of a single index.

**Syntax**

Array\_name = [rows][columns] # declaration of 2D array

Arr-name = [ [m1, m2, m3, … . mn], [n1, n2, n3, … .. nn] ]

Where **m** is the row and **n** is the column of the table.

Play Videox[[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack" \t "_blank)

### Access Two-Dimensional Array

In [Python](https://www.javatpoint.com/python-tutorial)

, we can access elements of a two-dimensional array using two indices. The first index refers to the indexing of the list and the second index refers to the position of the elements. If we define only one index with an array name, it returns all the elements of 2-dimensional stored in the [array](https://www.javatpoint.com/python-arrays)

.

Let's create a simple program to understand **2D** (two dimensional) arrays in Python.

**2dSimple.py**

1. Student\_dt = [ [72, 85, 87, 90, 69], [80, 87, 65, 89, 85], [96, 91, 70, 78, 97], [90, 93, 91, 90, 94], [57, 89, 82, 69, 60] ]
2. #print(student\_dt[])
3. print(Student\_dt[1]) # print all elements of index 1
4. print(Student\_dt[0]) # print all elements of index 0
5. print(Student\_dt[2]) # print all elements of index 2
6. print(Student\_dt[3][4]) # it defines the 3rd index and 4 position of the data element.

**Output:**



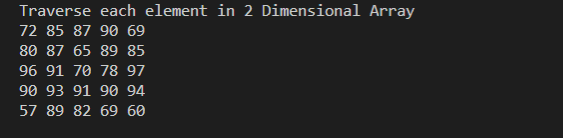
In the above example, we passed 1, 0, and 2 as parameters into 2D array that prints the entire row of the defined index. And we have also passed **student\_dt[3][4]** that represents the 3rd index and 4th position of a 2-dimensional array of elements to print a particular element.

### Traversing the element in 2D (two dimensional)

**Program.py**

1. # write a program to traverse every element of the two-dimensional array in Python.
2. Student\_dt = [ [72, 85, 87, 90, 69], [80, 87, 65, 89, 85], [96, 91, 70, 78, 97], [90, 93, 91, 90, 94], [57, 89, 82, 69, 60] ]
3. # Use **for** loop to print the entire elements of the two dimensional array.
4. **for** x in Student\_dt:  # outer loop
5. **for** i in x:  # inner loop
6. print(i, end = " ") # print the elements
7. print()

**Output:**



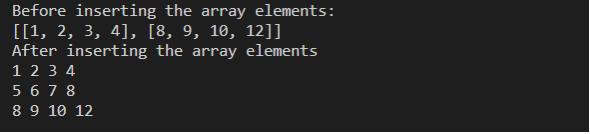
### Insert elements in a 2D (Two Dimensional) Array

We can insert elements into a 2 D array using the **insert()** function that specifies the element' index number and location to be inserted.

**Insert.py**

1. # Write a program to insert the element into the 2D (two dimensional) array of Python.
2. from array **import** \* # **import** all **package** related to the array.
3. arr1 = [[1, 2, 3, 4], [8, 9, 10, 12]]  # initialize the array elements.
4. print("Before inserting the array elements: ")
5. print(arr1) # print the arr1 elements.
6. # Use the insert() function to insert the element that contains two parameters.
7. arr1.insert(1, [5, 6, 7, 8])  # first parameter defines the index no., and second parameter defines the elements
8. print("After inserting the array elements ")
9. **for** i in arr1: # Outer loop
10. **for** j in i: # inner loop
11. print(j, end = " ") # print inserted elements.
12. print()

**Output:**



### Update elements in a 2 -D (Two Dimensional) Array

In a 2D array, the existing value of the array can be updated with a new value. In this method, we can change the particular value as well as the entire index of the array. Let's understand with an example of a 2D array, as shown below.

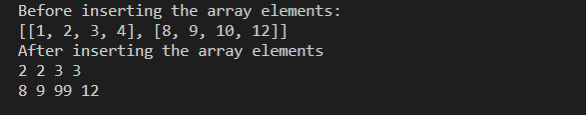
Create a program to update the existing value of a 2D array in Python.

**Update.py**

1. from array **import** \* # **import** all **package** related to the array.
2. arr1 = [[1, 2, 3, 4], [8, 9, 10, 12]]  # initialize the array elements.
3. print("Before inserting the array elements: ")
4. print(arr1) # print the arr1 elements.

7. arr1[0] = [2, 2, 3, 3] # update the value of the index 0
8. arr1[1][2] = 99 # define the index [1] and position [2] of the array element to update the value.
9. print("After inserting the array elements ")
10. **for** i in arr1: # Outer loop
11. **for** j in i: # inner loop
12. print(j, end = " ") # print inserted elements.
13. print()

**Output:**



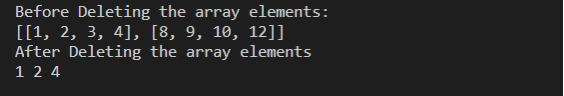
### Delete values from a 2D (two Dimensional) array in Python

In a 2- D array, we can remove the particular element or entire index of the array using **del()** function in Python. Let's understand an example to delete an element.

**Delete.py**

1. from array **import** \* # **import** all **package** related to the array.
2. arr1 = [[1, 2, 3, 4], [8, 9, 10, 12]]  # initialize the array elements.
3. print("Before Deleting the array elements: ")
4. print(arr1) # print the arr1 elements.
6. del(arr1[0][2]) # delete the particular element of the array.
7. del(arr1[1]) # delete the index 1 of the 2-D array.
9. print("After Deleting the array elements ")
10. **for** i in arr1: # Outer loop
11. **for** j in i: # inner loop
12. print(j, end = " ") # print inserted elements.
13. print()

**Output:**



### Size of a 2D array

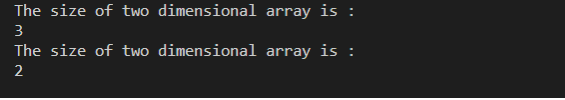
A **len**() function is used to get the length of a two-dimensional array. In other words, we can say that a **len**() function determines the total index available in 2-dimensional arrays.

Let's understand the len() function to get the size of a 2-dimensional array in Python.

**Size.py**

1. array\_size = [[1, 3, 2],[2,5,7,9], [2,4,5,6]] # It has 3 index
2. print("The size of two dimensional array is : ")
3. print(len(array\_size)) # it returns 3
5. array\_def = [[1, 3, 2], [2, 4, 5, 6]] # It has 2 index
6. print("The size of two dimensional array is : ")
7. print(len(array\_def)) # it returns 2

**Output:**

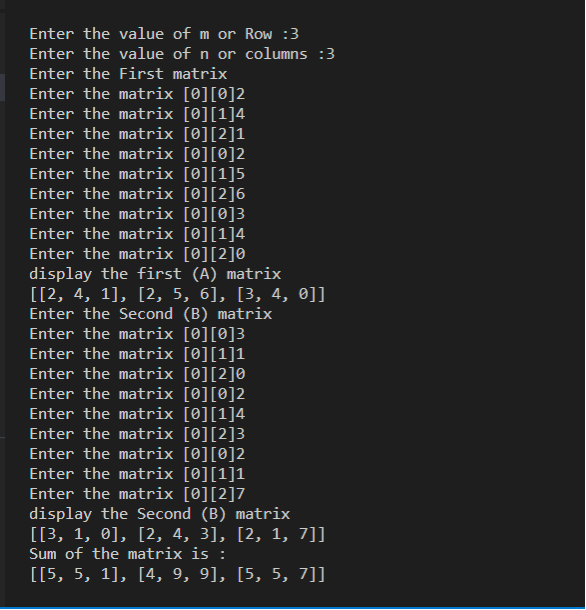


Write a program to print the sum of the 2-dimensional arrays in Python.

**Matrix.py**

1. def two\_d\_matrix(m, n): # define the function
2. Outp = []  # initially output matrix is empty
3. **for** i in range(m): # iterate to the end of rows
4. row = []
5. **for** j in range(n): # j iterate to the end of column
6. num = **int**(input(f "Enter the matrix [{0}][{j}]"))
7. row.append(num) # add the user element to the end of the row
8. Outp.append(row) # append the row to the output matrix
9. **return** Outp
11. def sum(A, B): # define sum() function to add the matrix.
12. output = [] # initially, it is empty.
13. print("Sum of the matrix is :")
14. **for** i in range(len(A)): # no. of rows
15. row = []
16. **for** j in range(len(A[0])): # no. of columns
18. row.append(A[i][j] + B[i][j]) # add matrix A and B
19. output.append(row)
20. **return** output    # **return** the sum of both matrix
22. m = **int**(input("Enter the value of m or Row\n")) # take the rows
23. n = **int**(input("Enter the value of n or columns\n")) # take the columns
25. print("Enter the First matrix ") # print the first matrix
26. A = two\_d\_matrix(m, n) # call the matrix function
27. print("display the first (A) matrix")
28. print(A) # print the matrix
30. print("Enter the Second (B) matrix ")
31. B = two\_d\_matrix(m, n) # call the matrix function
32. print("display the Second (B) matrix")
33. print(B) # print the B matrix
35. s= sum(A, B) # call the sum function
36. print(s) # print the sum of A and B matrix.

**Output:**



# Difference between Property and Attributes in Python

In this tutorial, we will understand the difference between property and attributes in Python. Everything in Python is an object, and each class has attributes, methods, or functions. When we work with the object-oriented programming language, we encounter the terms - Attributes and Properties. In terms of English, properties and attributes can be utilized similarly, but in terms of programming, both are different.

So we will discuss some critical differences between them. Let's move on to the introduction to attributes.

## Attributes in Python

Attributes are defined by the data variables such as name, age, height, etc. In Python, there are two types of attributes -

* Class Attributes
* Instance Attributes

### Class Attributes

Class attributes are the attributes that are created inside the class definition and belong to class it. These attributes are shared between all other objects of the same class. Let's understand the following example -

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

**Example -**

1. # declare a class
2. **class** Student:
4. # class attribute
5. count = 0
7. # define a method
8. **def** increment(self):
9. Student.count += 1
11. # create an Student
12. # class object
13. a1 = Student()
15. # calling object's method
16. a1.increment()
18. # print value of class attribute
19. **print**(a1.count)
20. a2 = Student()
22. a2.increment()
23. **print**(a2.count)
24. **print**(Student.count)

**Output:**

1

2

2

**Explanation -**

In the above code, we have created a class named Student, and assigned class attributes to count. We can access it using the instance of the class or class name itself.

### Instance Attribute

An object is nothing but an instance of the class in Python. Instance Attributes are unique to each instance, and every object/instance contains its attributes and can be changed without modifying other instances. Let's understand the following example.

**Example -**

1. # create a class
2. **class** Student:
4. # constructor
5. **def** \_\_init\_\_(self):
7. # instance attribute
8. self.name = 'JavaTpoint'
9. self.salary = 50000
11. # define a method
12. **def** show(self):
13. **print**(self.name)
14. **print**(self.salary)
16. # create an object of
17. # Student class
18. x = Student()
20. # method calling
21. x.show()

**Output:**

JavaTpoint

50000

Now we have a brief idea about idea attributes; let's understand the example.

## Properties in Python

Properties are unique attributes that contain getter, setter, and deleter methods such as \_\_get\_\_, \_\_set\_\_, and \_\_delete\_\_ methods. Python provides the @property decorator, which can be used to define properties in Python code. A decorator changes the behaviour of a wrapped function without changing its actual definition.

Let's understand the following example -

**Example -**

1. # create a class
2. **class** PropertyDeco:
4. # constructor
5. **def** \_\_init\_\_(self, message):
6. self.\_message = message
8. # getting the messages
9. @property
10. **def** message(self):
11. **print**('Getting message')
12. **return** self.\_message
14. # setting the messages
15. @message.setter
16. **def** message(self, message):
17. **print**('Setting message to ' + message)
18. self.\_message = message
20. # deleting the messages
21. @message.deleter
22. **def** message(self):
23. **print**('Deleting message')
24. **del** self.\_message

27. # create an object of class
28. x = PropertyDeco('Welcome to JavaTpoint')
29. **print**(x.message)
31. x.message = 'Happy Learning!'
33. # deleting the message
34. **del** x.message

**Output:**

Getting message

Welcome to JavaTpoint

Setting message to Happy Learning!

Deleting message

**Explanation -**

In the above code, we created a class that will display the message. First, we initialized the message attribute and defined the property using the @property decorator. Then we explained the setter method that will set the method when we modify the message. Then we described the deleter method. These are the unique attributes of the property decorator.

We can also define the properties using the property() method.

**Example -**

1. # create a class
2. **class** PropertyDeco:
4. # constructor
5. **def** \_\_init\_\_(self, message):
6. self.\_message = message
8. # getting the messages
9. @property
10. **def** getter(self):
11. **print**('Getting message')
12. **return** self.\_message
14. # setting the messages
15. **def** setter(self, message):
16. **print**('Setting message to ' + message)
17. self.\_message = message
19. # deleting the messages
20. **def** deleter(self):
21. **print**('Deleting message')
22. **del** self.\_message
24. message = property(getter, setter, deleter, )

27. # create an object of class
28. x = PropertyDeco('Welcome to JavaTpoint')
29. **print**(x.message)
31. x.message = 'Happy Learning!'
33. # deleting the message
34. **del** x.message

**Output:**

Getting message

Welcome to JavaTpoint

Setting message to Happy Learning!

Deleting message

The above code is similar to the previous code. The only difference is that, we created the getter, setter, and deleter() methods instead of a decorator.

## Attributes V/s Property

Below is the table of the attributes and properties.

|  |  |
| --- | --- |
| **Attribute** | **Property** |
|  |  |

|  |  |
| --- | --- |
| Attributes are defined by data variables like name, age, height etc. | Properties are special type of attributes. |
| There are two types of attributes -   * Class attributes * Instance attributes | Property method comes with the getter, setter and delete methods like \_\_get\_\_, \_\_set\_\_, and \_\_delete\_\_ methods. |
| Class attributes are defined in the class body not in the functions. | We can define getters, setters, and delete methods with the **property()** function. |
| Instance attributes are defined in the class body using the self keyword usually it the **\_\_init\_\_()** method. | To read the property, we can use the **@property** decorator which can be added above our method. |

## Conclusion

Thus, in this tutorial, we learned the significant difference between attributes and properties in Python. This tutorial included the types of attributes in Python and what are the properties in Python. It will help to grasp the object-oriented programming concepts in a better way.

# Python Packing and Unpacking Arguments in Python

In this tutorial, we will learn about the packing and unpacking of the arguments. Python provides a unique feature to pack or unpack the arguments. Suppose we have a function that takes five arguments; we create a list of five elements and pass it to the function, which will throw an error because elements are packed within the list. We need to pass five separate arguments to the function.

In the past, Python developers referred to this feature as tuple unpacking, but it became quite useful and popular among them. Hence, this feature generalized to all kinds of iterables; nowadays, it is referred to as **iterable unpacking.**

We will discuss how we can use iterable unpacking to make our code more readable.

In Addition, we will also explore some practical examples for loops, function definition, and function calls.

48.4M

824

History of Java

## Packing and Unpacking in Python

Python provides the facility to declare the variables on the left side of assignment operation. These variables can be in the same number of tuple element. Each variable in the tuple can have one value (or more if use the \* operator) from iterable on the right side of the assignment. Python developers called this process a tuple unpacking. Below is the definition of the packing and unpacking arguments.

**Packing -** As its name suggests, it wraps all the arguments into a single variable, and this function call gets into a tuple called args. We can use any other name in place of args.

* **Example:** def func( \*args )

**Unpacking -** Unpacking is the term which refers to an operation list or tuple assigned to a single variable.

* **Example -** (a, b, c) = List/tuple

Unpacking makes the code more readable that why it got popularity among Python developers. Let's have an example to understand how unpacking works in Python.

### Unpacking Tuples

Python allows us to assign tuples of the variable on the left side of the assignment operator and tuple values on the right side. If the variables are the same in number as the tuple value, then the value is automatically assigned to the left according to their position on the left. Let's understand the following example.

**Example -**

1. (a, b, c, d, e, f) = (1, 2, 3, 4, 5, 6)
2. **print**(a)
3. **print**(b)
4. **print**(c)
5. **print**(d)
6. **print**(e)
7. **print**(f)

**Output:**

1

2

3

4

5

6

As we can see, we have put the tuple on both sides of an assignment operator. The values of the tuple are assigned to the variables according to their relative position. Such as a will be 1, b will be 2 and so on.

We don't need to create the tuple on both sides; we can simply use the following ways.

**Example -**

1. (a, b, c, d, e, f) = (1, 2, 3, 4, 5, 6)
2. a, b, c, d, e, f = (1, 2, 3, 4, 5, 6)
3. (a, b, c, d, e, f) = 1, 2, 3, 4, 5, 6

#### Note - The number of variables on the left side must equal the number of elements of the tuple on the right side of the assignment operator. Otherwise, it will throw a ValueError.

In the example below, we use the four variables on the left and five values on the right. It will through a ValueError as there are too many values to unpack.

**Example -**

1. (a, b, c, d) = (1, 2, 3, 4, 5)

**Output:**

ValueError: too many values to unpack (expected 4)

## Unpacking Iterables

Tuple unpacking operation extended to other iterables of Python. The only requirement is that item yields the same one item per variable in the receiving in the tuple (or list).

Let's understand the following example -

**Example -**

1. a, b, c = '123'
2. **print**(a)
3. **print**(b)
4. **print**(c)
5. **print**("Unpacking lists")
6. a, b, c = [1, 2, 3]
7. **print**(a)
8. **print**(b)
9. **print**(c)
10. **print**("Unpacking generators")
11. gen = (i \*\* 2 **for** i **in** range(3))
12. a, b, c = gen
13. **print**(a)
14. **print**(b)
15. **print**(c)
16. **print**("Unpacking dictionaries (keys, values, and items")
17. my\_dict = {'one': 1, 'two':2, 'three': 3}
18. a, b, c = my\_dict  # Unpack keys
19. **print**(a)
20. **print**(b)
21. **print**(c)
22. **print**("Unpack values")
23. a, b, c = my\_dict.values()
24. **print**(a)
25. **print**(b)
26. **print**(c)
27. **print**("Unpacking key-value pairs")
28. a, b, c = my\_dict.items()
29. **print**(a)
30. **print**(b)
31. **print**(c)

**Output:**

1

2

3

Unpacking lists

1

2

3

Unpacking generators

0

1

4

Unpacking dictionaries (keys, values, and items

one

two

three

Unpack values

1

2

3

Unpacking key-value pairs

('one', 1)

('two', 2)

('three', 3)

On the other hand, we can use the following way to unpack the argument where the list/tuple defines on the left of the assignment operator and values on the right side.

**Example -**

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

**Output:**

1

2

3

However, it is not recommended in the actual code because it may be a little bit confusing for beginner Python developers.

We can also use the set unpacking operations since sets are unordered collections. So the order of assignments can throw subtle bugs. Let's understand the following example.

**Example -**

1. a, b, c = {'x', 'y', 'z'}
2. **print**(a)
3. **print**(b)
4. **print**(c)

**Output:**

y

z

x

The set returns the unordered elements, so it will also happen to the unpacking argument. So it is recommended to avoid the set unpacking operation.

## Packing with \* Operators

The \* operator is the tuple (or iterables) unpacking operator. It allows packing multiple values into a single variable. We pack a tuple of values into a single variable using the \* operators in the following example.

**Example -**

1. \*a, = ('x', 'y', 'z')
2. **print**(a)

**Output:**

['x', 'y', 'z']

The left side of the assignment must be a tuple (or a list). That's we use a trailing comma.

Let's see another way of packing arguments.

**Example -**

1. a, b, \*c= 1, 2, 'x', 'y', 'z'
2. **print**(a)
3. **print**(b)
4. **print**(c)

**Output:**

1

2

['x', 'y', 'z']

In the above code, we pack the trailing values in b.

1. \*a, b, c = 1, 2, 3
2. **print**(a)
3. **print**(b)
4. **print**(c)

**Output:**

[1]

2

3

Packing no values in a (a defaults to []) because b, c, and d are mandatory.

**Example -**

1. \*a, b, c, d = 1, 2, 3
2. **print**(a)
3. **print**(b)
4. **print**(c)
5. **print**(d)

**Output:**

[]

1

2

3

If we don't pass the value for the required variable, it will throw an error.

**Example -**

1. \*a, b, c, d, e = 1, 2, 3

**Output:**

ValueError: not enough values to unpack (expected at least 4, got 3)

Packing the value in the single variable with the \* operator can be handy when we want to access the elements from the generator in a single variable without using the **list()** function. Let's understand the following example.

**Example -**

1. gen = (2 \*\* x **for** x **in** range(15))
2. **print**(gen)
3. \*g, = gen
4. **print**(g)
5. range = range(10)
6. \*r, = range
7. **print**(r)

**Output:**

at 0x7fcc4aec0ac0>

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384]

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

**Explanation -**

In the above code, we packed the element in gen into g and ran into r, respectively. Using this, we can omit the **list(**) function to create a list of values from the range object.

## Working of Packing and Unpacking Simultaneously

We have discussed packing and unpacking individually and now we will implement both concepts in practice. Packing and unpacking allows the programmer to write more readable, clean, and Pythonic code. Let's see some example of packing and unpacking.

### Assigning in Parallel

Assignment in parallel is the most common use-cases of unpacking in Python. As discussed earlier, we can assign multiple variables according to an iterable (tuple or list).

Let's understand the following example.

**Example -**

1. emp = ["Mathew Wade", "400000$", "Software Engineer"]
2. name = emp[0]
3. age = emp[1]
4. profile = emp[2]
5. **print**("The name is: ", name)
6. **print**("The age is: ", age)
7. **print**("The profile is: ", profile)

**Output:**

The name is: Mathew Wade

The age is: 400000$

The profile is: Software Engineer

The above code is quite traditional and quite clumsy, hard to type. We can make it easier and more readable as below.

**Example -**

1. name, age, profile = ["Mathew Wade", "400000$", "Software Engineer"]
2. **print**("The name is: ", name)
3. **print**("The age is: ", age)
4. **print**("The profile is: ", profile)

**Output:**

The name is: Mathew Wade

The age is: 400000$

The profile is: Software Engineer

As we can see, using the unpacking, we reduced the number of lines and made it simple and elegant.

### Swapping values Between Variables

We can swap the values in Python without using a temporary or auxiliary variable. Let's understand the following example of swapping.

**Example - Without Swapping**

1. a = 10
2. b = 20
3. temp = a
4. b = temp
5. **print**(a)
6. **print**(b)
7. **print**(temp)
8. Output:

**Output:**

10

10

10

**Example -**

1. a = 100
2. b = 200
3. a, b = b, a
4. **print**(a)
5. **print**(b)

**Output:**

200

100

In statement a, b = b, a, we're reassigning a to b and b to a in one line of code. It is a lot more readable and straightforward. Also, notice that with this technique, there is no need for a new temporary variable.

### Dropping Unnecessary Values With \*

We can use the \* to drop the unneeded values. Let's understand the following example.

**Example -**

1. a, b, \*\_ = 6, 8, 0, 0, 0, 0
2. **print**(a)
3. **print**(b)

**Output:**

6

8

### Returning Tuple in the Functions

We can return the multiple values separate by the commas from Python function. Since we can define the tuple object without using parentheses, this kind of operation can be interpreted as returning a tuple of values. If we code a function that returns multiple values, then we can perform iterable packing and unpacking operations with the returned values.

Let's understand the following example.

**Example -**

1. **def** powers(number):
2. **return** number, number \*\* 2, number \*\* 3
3. # Packing returned values in a tuple
4. result = powers(8)
5. **print**(result)
6. # Unpacking returned values to multiple variables
7. number, square, cube = powers(8)
8. **print**(number)
9. **print**(square)
10. **print**(cube)
11. \*\_, cube = powers(2)
12. **print**(cube)

**Output:**

(8, 64, 512)

8

64

512

8

### Merging Iterables With the \* Operators

Another interesting use case for the unpacking operator \* is the ability to merge several iterables into a final sequence. This functionality works for lists, tuples, and sets. Take a look at the following examples:

**Example -**

1. tup = (1, 2, 3)
2. **print**((0, \*tup, 4))
3. list1 = [1, 2, 3]
4. **print**([0, \*list1, 4])
5. my\_set = {1, 2, 3}
6. **print**({0, \*my\_set, 4})
7. **print**([\*my\_set, \*list1, \*tup, \*range(1, 4)])
8. my\_str = "123"
9. **print**([\*my\_set, \*list1, \*tup, \*range(1, 4), \*my\_str])

**Output:**

(0, 1, 2, 3, 4)

[0, 1, 2, 3, 4]

{0, 1, 2, 3, 4}

[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, '1', '2', '3']

We can use the iterable unpacking operator \* when defining sequences to unpack the elements of a subsequence (or iterable) into the final sequence. It will allow us to create sequences on the fly from existing sequences without calling methods like **append() and insert(),** and so on.

## Unpacking Dictionaries With the \* Operator

Python provides the \*\* operator to unpack the dictionary. This operator is also known as the dictionary unpacking operator. The use of this operator was extended by the PEP 448. Now, we can use it in function calls and list comprehension.

**Example -**

1. numbers = {"one": 1, "two": 2, "three": 3}
2. letters = {"a": "Apple", "b": "Bat", "c": "Cat"}
3. combine = {\*\*numbers, \*\*letters}
4. **print**(combine)

**Output:**

{'one': 1, 'two': 2, 'three': 3, 'a': 'Apple', 'b': 'Bat', 'c': 'Cat'}

**Explanation -**

In the above code, we created two dictionaries; we unpacked both the dictionary and created a combined new dictionary that includes elements of both dictionaries.

## Unpacking the For loops

We can also use iterable unpacking in the context of for loops. When we run a for loop, the loop assigns one item of its iterable to the target variable in every iteration. If the item to be assigned is an iterable, then we can use a tuple of target variables.

Let's understand the following example -

**Example -**

1. students = [("Mathew", 45, 90), ("Warner", 67, 88), ("Kapil", 75, 100)]
2. **for** mark **in** students:
3. **print**(f"Marks of {mark[0]} is: {mark[1] + mark[2]}")

**Output:**

Marks of Mathew is: 135

Marks of Warner is: 155

Marks of Kapil is: 175

## Conclusion

Iterable unpacking is an essential and pretty useful feature in Python. This feature allows us to unpack an iterable into several variables. In this tutorial, we have discussed some crucial concepts of iterable unpacking and packing. We have seen some valuable examples and how to write more readable, maintainable, and Pythonic code.

This knowledge can solve common problems like parallel assignment and swapping values between variables. We can also use this feature in other Python concepts like loops.

# Mirror Character of a String in Python

In this tutorial, we will learn about mirror characters in Python. Mirror characters problem can be asked in the technical interview. First, we understand the mirror characters in the string.

If the two strings are the same in the alphabetical position, one from the front and the other from the back. For example - z is a mirror character of a, y is a mirror character of b, and so on. Let's write the Python program to find the mirror character of the strings.

## Find Mirror Characters in Python

In the alphanumeric character, we have to mirror the characters from the Nth position up to the length of the string. Let's understand the problem statement.

**Example -**

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

1. Input : N = 3
2. paradox
3. Output : paizwlc
4. We mirror characters **from** position 3 to end.
6. Input : N = 6
7. pneumonia
8. Output : pnefnlmrz

Following are the different characters and their mirror characters.

1. a b c d e f g h i j k l m || n o p q r s t u v w x y z

Mirroring refers to alphabetical order corresponds to z, b corresponds to y. The first character becomes the last character; the second character becomes the second last character, and so on. To write the program, we will create a string (or a character array) which contains the English alphabet in lower case. Now we take the pivot point up to the length and reverse the alphabetical order of the character by using the ASCII value as the index. Let's understand the following example.

**Example -**

1. **def** mirror\_compute(st, n):
2. # Creating a string having reversed
3. # alphabetical order
4. reverse\_string = "zyxwvutsrqponmlkjihgfedcba"
5. l = len(st)
7. result = ""
8. **for** i **in** range(0, n):
9. result = result + st[i];
11. **for** i **in** range(n, l):
12. result = (result +
13. reverse\_string[ord(st[i]) - ord('a')]);
15. **return** result;
17. st = "javaTpoint"
18. n = 4
19. answer = mirror\_compute(st, n - 1)
20. **print**(answer)

**Output:**

javzmklrmg

**Explanation**

In the above code, we created a function **mirror\_compute()** that takes two arguments - string and its length. Then we created the string that has reversed alphabetical order. We initialized a result as an empty string that stores the result of the given string. A given string remains unchanged up to a point specified, and from point up to the length to the string, we reverse the alphabetical order.

### Time Complexity

Time complexity is O(n), where n represents the size of the given string.

**Auxiliary Space:** O(1), no extra space is required, so it is a constant.

**Method - 2**

In this method, we will use the dict() and zip() method to create the dictionary where characters in alphabetical order are the key and the character in the reverse order will be the value. Let's understand the following example.

**Example -**

1. **def** mirror\_compute(st, n):
2. # create dictionary
3. order = 'abcdefghijklmnopqrstuvwxyz'
4. reverse = 'zyxwvutsrqponmlkjihgfedcba'
5. dictChars = dict(zip(order,reverse))
7. # divide string at k
8. pre = string[0:k-1]
9. suffix = string[k-1:]
10. mirror = ''
12. # change suffix into mirror char
13. **for** i **in** range( len(suffix)):
14. mirror = mirror + dictChars[suffix[i]]
16. # join prefix and mirrored part
17. **return** pre+mirror
19. st = "javaTpoint"
20. n = 4
21. answer = mirror\_compute(st, n - 1)
22. **print**(answer)

**Output:**

javzmklrmg

## Conclusion

This tutorial introduced the mirror characters for a given string and how to mirror the character present in the possible k. We have written the python program to find the mirror characters.

# Python Property Decorator

In this tutorial, we will learn about the property decorator in Python. Previously, we have discussed decorators in Python; if you are familiar with decorators, you should read [this tutorial](https://www.javatpoint.com/python-decorator) before moving forward.

Let's have an introduction to property decorators.

## Python @property Decorator

Let's understand it by an example - Suppose we have a class Employee who has three properties **first\_name, last\_name**, and **department\_name.** And we have another function **email()** function that generates an email address for an employee using its first\_name and **last\_name**. Let's see the following code.

**Example -**

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

1. **class** Student:
2. **def** \_\_init\_\_(self, first\_name, last\_name, full\_name):
3. self.first\_name =first\_name
4. self.last\_name = last\_name
5. self.department\_name = full\_name
7. # generate email using first and last name
8. **def** email(self):
9. **return** '{}.{}@avenger.com'.format(self.first\_name, self.last\_name)

Now create the object of class and call the **email()** function.

1. obj = Employee('Bruce', 'Banner', 'Bruce Banner')
3. **print**("First Name is:", obj.first\_name)
4. **print**("Last Name is:", obj.last\_name)
5. **print**("Depatement Name is", obj.full\_name)
7. **print**(obj.email())

**Output:**

First Name is: Bruce

Last Name is: Banner

Full Name is: Bruce Banner

Bruce.Banner@avenger.com

In the above code, we have defined the three attributes first\_name, last\_name, and department\_name and **email()** are the derived attributes.

The full\_name is declared as the variable and email() is declared as a function. When we run the program, we get the first\_name, last\_name, and full\_name. The full\_name is derived from the first\_name and the last\_name so email is. Now, let's make some changes in the program.

1. obj.first\_name='Natasha'
2. **print**("Full Name is", obj.full\_name)
3. **print**(obj.email())

Changing the first\_name to Natasha and printing the **full\_name** and **email** will give the following output.

**Output:**

Full Name is Bruce Banner

Natasha.Banner@avenger.com

We can see that the first name is changed, and the email changes automatically. Still, the full\_name doesn't change despite using the first\_name attribute because email() is a function called when we want the email to be returned, while first\_name is set at the time of initialization of the object. We can fix this problem by creating another function of the full\_name we created for the email().

**Example -**

1. **class** Employee:
2. **def** \_\_init\_\_(self, first\_name, last\_name):
3. self.first\_name =first\_name
4. self.last\_name = last\_name
6. **def** full\_name(self):
7. **return** self.first\_name + ' ' + self.last\_name
9. # generate email using first and last name
10. **def** email(self):
11. **return** '{}.{}@avenger.com'.format(self.first\_name, self.last\_name)
13. obj = Employee('Bruce', 'Banner')
15. **print**("First Name is:", obj.first\_name)
16. **print**("Last Name is:", obj.last\_name)
18. **print**(obj.full\_name())
19. **print**(obj.email())
21. **print**('After changing the first name:')
22. obj.first\_name='Natasha'
23. **print**(obj.full\_name())
24. **print**(obj.email())

**Output:**

First Name is: Bruce

Last Name is: Banner

Bruce Banner

Bruce.Banner@avenger.com

After changing the first name:

Natasha Banner

Natasha.Banner@avenger.com

We get changed **full\_name** and updated email, but it is not a Pythonic way of solving this problem. Here the @property decorator comes into play to resolve such a problem.

## Using @property Decorator

The property decorator returns the property attributes of a class from the stated getter and setter and deletes them as a parameter. We will use the @property decorator to solve this problem. Now let's see the above example using the @property decorator.

**Example -**

1. **class** Employee:
2. **def** \_\_init\_\_(self, first\_name, last\_name):
3. self.first\_name =first\_name
4. self.last\_name = last\_name
6. @property
7. **def** full\_name(self):
8. **return** self.first\_name + ' ' + self.last\_name
10. # generate email using first and last name
11. **def** email(self):
12. **return** '{}.{}@avenger.com'.format(self.first\_name, self.last\_name)
14. obj = Employee('Bruce', 'Banner')
16. **print**("First Name is:", obj.first\_name)
17. **print**("Last Name is:", obj.last\_name)
19. **print**("Full Name is:", obj.full\_name)
20. **print**("Email is:", obj.email())
22. **print**('After changing the first name:')
23. obj.first\_name='Natasha'
24. **print**("Full Name is:", obj.full\_name)
25. **print**("Email is:", obj.email())

**Output:**

First Name is: Bruce

Last Name is: Banner

Full Name is: Bruce Banner

Email is: Bruce.Banner@avenger.com

After changing the first name:

Full Name is: Natasha Banner

Email is: Natasha.Banner@avenger.com

**Explanation -**

In the above code, the @property decorator is used on the full\_name function, and now this function changed into the attribute and can also work as a getter because of the @property decorator.

## Use setter and deleter methods with @property Decorator

The function that applies the @property decorator is known as the getter. In the previous example, the full\_name acts a getter. In this section, we will understand the getter and setter.

As the name suggests, the setter method set the value of the attributes and deleter method deleter method removes the attributes from memory. Let's implement a setter and getter method for full\_name attributes.

**Example -**

1. **class** Student:
2. **def** \_\_init\_\_(self, name, age):
3. self.name = name
4. self.age = age
6. @property
7. **def** age(self):
8. **return** self.\_age
10. @age.setter
11. **def** age(self, value):
12. **if** value <= 0:
13. **raise** ValueError('Please Enter the Positive Number')
14. self.\_age = value
16. @age.deleter
17. **def** age(self):
18. self.age = None
19. **print**("The age is deleted")
21. @property
22. **def** name(self):
23. **return** self.\_name
25. @name.setter
26. **def** name(self, value):
27. **if** value.strip() == '':
28. **raise** ValueError('Please Enter the Valid String')
29. self.\_name = value
31. @name.deleter
32. **def** name(self):
33. self.name = None
34. **print**("The name is deleted")
36. s1 = Student('Rishabh Pant', 21)
37. **print**(s1.age)
38. **print**(s1.name)
39. s1.name = 'KL Rahul'
40. **print**(s1.name )
41. s1.name = ''
42. **print**(s1.name)

**Output:**

21

Rishabh Pant

KL Rahul

Traceback (most recent call last):

File "d:/Python Project/property.py", line 557, in

s1.name = ''

File "d:/Python Project/bubble\_sort.py", line 544, in name

raise ValueError('Please Enter the Valid String')

ValueError: Please Enter the Valid String

**Explanation -**

In the above code, we have initialized a class Student where we passed name and age in the constructor. First, we created the **age()** method that applied the @property decorator; it will get the age and returns. Then we created the setter method for the age where we checked if the user entered the value as negative then raised an error. Then we created the deleter method for the age. The same thing we have done with the name attributes. In the setter method, we checked if the user enters the name as an empty string raises an error otherwise, set the given name.

**Example - 2**

1. **class** Employee:
2. **def** \_\_init\_\_(self, first\_name, last\_name):
3. self.first\_name =first\_name
4. self.last\_name = last\_name
6. @property
7. **def** full\_name(self):
8. **return** self.first\_name + ' ' + self.last\_name
10. #setter for the full\_name
11. @full\_name.setter
12. **def** full\_name(self, name):
13. # split the name from space
14. first\_name, last\_name = name.split(" ")
15. self.first = first\_name
16. self.last = last\_name
18. #deleter for full\_name
19. @full\_name.deleter
20. **def** full\_name(self):
21. self.first = None
22. self.last = None
23. **print**('Deleted the full\_name')
25. # generate email using first and last name
26. **def** email(self):
27. **return** '{}.{}@avenger.com'.format(self.first\_name, self.last\_name)
29. obj = Employee('Steve', 'Rogers')
30. **print**('Fullname is:', obj.full\_name)
31. **print**('Email address is: ', obj.email())
33. # now updating the obj object's first name
34. obj.first\_name = 'Bruce'
35. **print**('Fullname of obj is: ', obj.full\_name)
36. **print**('And email address:', obj.email())
38. #setting new value of full\_name
39. obj.full\_name = 'Hello World'
40. **print**('New Fullname of obj is:', obj.full\_name)
42. #deleting the full\_name
43. **del** obj.full\_name

**Output:**

Fullname is: Steve Rogers

Email address is: Steve.Rogers@avenger.com

Fullname of obj is: Bruce Rogers

And email address: Bruce.Rogers@avenger.com

New Fullname of obj is: Peter Parker

Deleted the full\_name

In the above code, we have created the getter, setter, and deleter using the property decorator.

### The property() Function

We can use the **property()** function to create getters, setters, and deleter in-place of @property decorator. The syntax is given below.

**Syntax -**

1. property(fget=None, fset=None, fdel=None, doc)

**Parameters -**

* **fget() -** It is used to get the attribute's value the same as the getters.
* **fset() -** It is used to set the value of the attributes the same as the setters.
* **fdel() -** It is used to delete the attribute value.
* **doc() -** It represents the string that contains the documentation (docstring) for the attributes.

It returns a property attribute from the given getter, setter, and deleter.

Let's understand the following example.

**Example -**

1. **class** Employee:
3. **def** \_\_init\_\_(self, first\_name, last\_name):
4. self.first\_name = first\_name
5. self.last\_name = last\_name
6. self.full\_name = first\_name+' '+last\_name
8. **def** full\_name\_getter(self):
9. **return** self.first\_name +' '+ self.last\_name
11. **def** full\_name\_setter(self,name):
12. firstname, lastname = name.split()
13. self.first\_name = firstname
14. self.last\_name = lastname
16. **def** full\_name\_deleter(self):
17. self.first\_name = None
18. self.last\_name = None
19. **print**('Deleted the full\_name.')
21. **def** email(self):
22. **return** '{}.{}@email.com'.format(self.first\_name, self.last\_name)
24. full\_name = property()
25. full\_name = full\_name.getter(full\_name\_getter)
26. full\_name = full\_name.setter(full\_name\_setter)
27. full\_name = full\_name.deleter(full\_name\_deleter)
29. # this can be done in a single line too
30. # full\_name = property(full\_name\_getter, full\_name\_setter, full\_name\_deleter)
32. obj = Employee('Nick', 'Fury')
33. **print**('Full Name is: ', obj.full\_name)
34. **print**('Email Address: ', obj.email())
36. # now updating the obj object's first name
37. obj.first = 'Oddin'
38. **print**('Full Name is:  ', obj.full\_name)
39. **print**('Email Address: ', obj.email())
41. #setting new value of full\_name
42. obj.full\_name = 'Bruce Banner'
43. **print**('New Full Name is: ', obj.full\_name)
45. #deleting the full\_name
46. **del** obj.full\_name

**Output:**

Full Name is: Nick Fury

Email Address: Nick.Fury@email.com

Full Name is: Nick Fury

Email Address: Nick.Fury@email.com

New Full Name is: Bruce Banner

Deleted the full\_name.

We can also implement these methods using the single line of code.

full\_name = property(full\_name\_getter, full\_name\_setter, full\_name\_deleter)

## Important Tips

We don't need to create all three methods for every property in the code. We can define read-only properties by only including a getter method. We can also avoid the deleter method and use getter and setter. We can avoid the setter method if we want to set the attribute when the instance is created or if it should be modified within the class.

The users are free to choose any method depending on their working context.

## Conclusion

This tutorial has covered property decorators and their examples in detail. The **property()** method helps create getter, setter, and deleter methods. We can define the properties with the @property decorator syntax, which is more compatible and straightforward. It is considered the **'pythonic'** way to define getters, setters, and deleter.

Using the properties, we can modify the class's internal implementation without affecting it. The getter, setter, and deleter help avoid accessing or modifying the data directly.

# Python Property Decorator

In this tutorial, we will learn about the property decorator in Python. Previously, we have discussed decorators in Python; if you are familiar with decorators, you should read [this tutorial](https://www.javatpoint.com/python-decorator) before moving forward.

Let's have an introduction to property decorators.

## Python @property Decorator

Let's understand it by an example - Suppose we have a class Employee who has three properties **first\_name, last\_name**, and **department\_name.** And we have another function **email()** function that generates an email address for an employee using its first\_name and **last\_name**. Let's see the following code.

**Example -**

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

1. **class** Student:
2. **def** \_\_init\_\_(self, first\_name, last\_name, full\_name):
3. self.first\_name =first\_name
4. self.last\_name = last\_name
5. self.department\_name = full\_name
7. # generate email using first and last name
8. **def** email(self):
9. **return** '{}.{}@avenger.com'.format(self.first\_name, self.last\_name)

Now create the object of class and call the **email()** function.

1. obj = Employee('Bruce', 'Banner', 'Bruce Banner')
3. **print**("First Name is:", obj.first\_name)
4. **print**("Last Name is:", obj.last\_name)
5. **print**("Depatement Name is", obj.full\_name)
7. **print**(obj.email())

**Output:**

First Name is: Bruce

Last Name is: Banner

Full Name is: Bruce Banner

Bruce.Banner@avenger.com

In the above code, we have defined the three attributes first\_name, last\_name, and department\_name and **email()** are the derived attributes.

The full\_name is declared as the variable and email() is declared as a function. When we run the program, we get the first\_name, last\_name, and full\_name. The full\_name is derived from the first\_name and the last\_name so email is. Now, let's make some changes in the program.

1. obj.first\_name='Natasha'
2. **print**("Full Name is", obj.full\_name)
3. **print**(obj.email())

Changing the first\_name to Natasha and printing the **full\_name** and **email** will give the following output.

**Output:**

Full Name is Bruce Banner

Natasha.Banner@avenger.com

We can see that the first name is changed, and the email changes automatically. Still, the full\_name doesn't change despite using the first\_name attribute because email() is a function called when we want the email to be returned, while first\_name is set at the time of initialization of the object. We can fix this problem by creating another function of the full\_name we created for the email().

**Example -**

1. **class** Employee:
2. **def** \_\_init\_\_(self, first\_name, last\_name):
3. self.first\_name =first\_name
4. self.last\_name = last\_name
6. **def** full\_name(self):
7. **return** self.first\_name + ' ' + self.last\_name
9. # generate email using first and last name
10. **def** email(self):
11. **return** '{}.{}@avenger.com'.format(self.first\_name, self.last\_name)
13. obj = Employee('Bruce', 'Banner')
15. **print**("First Name is:", obj.first\_name)
16. **print**("Last Name is:", obj.last\_name)
18. **print**(obj.full\_name())
19. **print**(obj.email())
21. **print**('After changing the first name:')
22. obj.first\_name='Natasha'
23. **print**(obj.full\_name())
24. **print**(obj.email())

**Output:**

First Name is: Bruce

Last Name is: Banner

Bruce Banner

Bruce.Banner@avenger.com

After changing the first name:

Natasha Banner

Natasha.Banner@avenger.com

We get changed **full\_name** and updated email, but it is not a Pythonic way of solving this problem. Here the @property decorator comes into play to resolve such a problem.

## Using @property Decorator

The property decorator returns the property attributes of a class from the stated getter and setter and deletes them as a parameter. We will use the @property decorator to solve this problem. Now let's see the above example using the @property decorator.

**Example -**

1. **class** Employee:
2. **def** \_\_init\_\_(self, first\_name, last\_name):
3. self.first\_name =first\_name
4. self.last\_name = last\_name
6. @property
7. **def** full\_name(self):
8. **return** self.first\_name + ' ' + self.last\_name
10. # generate email using first and last name
11. **def** email(self):
12. **return** '{}.{}@avenger.com'.format(self.first\_name, self.last\_name)
14. obj = Employee('Bruce', 'Banner')
16. **print**("First Name is:", obj.first\_name)
17. **print**("Last Name is:", obj.last\_name)
19. **print**("Full Name is:", obj.full\_name)
20. **print**("Email is:", obj.email())
22. **print**('After changing the first name:')
23. obj.first\_name='Natasha'
24. **print**("Full Name is:", obj.full\_name)
25. **print**("Email is:", obj.email())

**Output:**

First Name is: Bruce

Last Name is: Banner

Full Name is: Bruce Banner

Email is: Bruce.Banner@avenger.com

After changing the first name:

Full Name is: Natasha Banner

Email is: Natasha.Banner@avenger.com

**Explanation -**

In the above code, the @property decorator is used on the full\_name function, and now this function changed into the attribute and can also work as a getter because of the @property decorator.

## Use setter and deleter methods with @property Decorator

The function that applies the @property decorator is known as the getter. In the previous example, the full\_name acts a getter. In this section, we will understand the getter and setter.

As the name suggests, the setter method set the value of the attributes and deleter method deleter method removes the attributes from memory. Let's implement a setter and getter method for full\_name attributes.

**Example -**

1. **class** Student:
2. **def** \_\_init\_\_(self, name, age):
3. self.name = name
4. self.age = age
6. @property
7. **def** age(self):
8. **return** self.\_age
10. @age.setter
11. **def** age(self, value):
12. **if** value <= 0:
13. **raise** ValueError('Please Enter the Positive Number')
14. self.\_age = value
16. @age.deleter
17. **def** age(self):
18. self.age = None
19. **print**("The age is deleted")
21. @property
22. **def** name(self):
23. **return** self.\_name
25. @name.setter
26. **def** name(self, value):
27. **if** value.strip() == '':
28. **raise** ValueError('Please Enter the Valid String')
29. self.\_name = value
31. @name.deleter
32. **def** name(self):
33. self.name = None
34. **print**("The name is deleted")
36. s1 = Student('Rishabh Pant', 21)
37. **print**(s1.age)
38. **print**(s1.name)
39. s1.name = 'KL Rahul'
40. **print**(s1.name )
41. s1.name = ''
42. **print**(s1.name)

**Output:**

21

Rishabh Pant

KL Rahul

Traceback (most recent call last):

File "d:/Python Project/property.py", line 557, in

s1.name = ''

File "d:/Python Project/bubble\_sort.py", line 544, in name

raise ValueError('Please Enter the Valid String')

ValueError: Please Enter the Valid String

**Explanation -**

In the above code, we have initialized a class Student where we passed name and age in the constructor. First, we created the **age()** method that applied the @property decorator; it will get the age and returns. Then we created the setter method for the age where we checked if the user entered the value as negative then raised an error. Then we created the deleter method for the age. The same thing we have done with the name attributes. In the setter method, we checked if the user enters the name as an empty string raises an error otherwise, set the given name.

**Example - 2**

1. **class** Employee:
2. **def** \_\_init\_\_(self, first\_name, last\_name):
3. self.first\_name =first\_name
4. self.last\_name = last\_name
6. @property
7. **def** full\_name(self):
8. **return** self.first\_name + ' ' + self.last\_name
10. #setter for the full\_name
11. @full\_name.setter
12. **def** full\_name(self, name):
13. # split the name from space
14. first\_name, last\_name = name.split(" ")
15. self.first = first\_name
16. self.last = last\_name
18. #deleter for full\_name
19. @full\_name.deleter
20. **def** full\_name(self):
21. self.first = None
22. self.last = None
23. **print**('Deleted the full\_name')
25. # generate email using first and last name
26. **def** email(self):
27. **return** '{}.{}@avenger.com'.format(self.first\_name, self.last\_name)
29. obj = Employee('Steve', 'Rogers')
30. **print**('Fullname is:', obj.full\_name)
31. **print**('Email address is: ', obj.email())
33. # now updating the obj object's first name
34. obj.first\_name = 'Bruce'
35. **print**('Fullname of obj is: ', obj.full\_name)
36. **print**('And email address:', obj.email())
38. #setting new value of full\_name
39. obj.full\_name = 'Hello World'
40. **print**('New Fullname of obj is:', obj.full\_name)
42. #deleting the full\_name
43. **del** obj.full\_name

**Output:**

Fullname is: Steve Rogers

Email address is: Steve.Rogers@avenger.com

Fullname of obj is: Bruce Rogers

And email address: Bruce.Rogers@avenger.com

New Fullname of obj is: Peter Parker

Deleted the full\_name

In the above code, we have created the getter, setter, and deleter using the property decorator.

### The property() Function

We can use the **property()** function to create getters, setters, and deleter in-place of @property decorator. The syntax is given below.

**Syntax -**

1. property(fget=None, fset=None, fdel=None, doc)

**Parameters -**

* **fget() -** It is used to get the attribute's value the same as the getters.
* **fset() -** It is used to set the value of the attributes the same as the setters.
* **fdel() -** It is used to delete the attribute value.
* **doc() -** It represents the string that contains the documentation (docstring) for the attributes.

It returns a property attribute from the given getter, setter, and deleter.

Let's understand the following example.

**Example -**

1. **class** Employee:
3. **def** \_\_init\_\_(self, first\_name, last\_name):
4. self.first\_name = first\_name
5. self.last\_name = last\_name
6. self.full\_name = first\_name+' '+last\_name
8. **def** full\_name\_getter(self):
9. **return** self.first\_name +' '+ self.last\_name
11. **def** full\_name\_setter(self,name):
12. firstname, lastname = name.split()
13. self.first\_name = firstname
14. self.last\_name = lastname
16. **def** full\_name\_deleter(self):
17. self.first\_name = None
18. self.last\_name = None
19. **print**('Deleted the full\_name.')
21. **def** email(self):
22. **return** '{}.{}@email.com'.format(self.first\_name, self.last\_name)
24. full\_name = property()
25. full\_name = full\_name.getter(full\_name\_getter)
26. full\_name = full\_name.setter(full\_name\_setter)
27. full\_name = full\_name.deleter(full\_name\_deleter)
29. # this can be done in a single line too
30. # full\_name = property(full\_name\_getter, full\_name\_setter, full\_name\_deleter)
32. obj = Employee('Nick', 'Fury')
33. **print**('Full Name is: ', obj.full\_name)
34. **print**('Email Address: ', obj.email())
36. # now updating the obj object's first name
37. obj.first = 'Oddin'
38. **print**('Full Name is:  ', obj.full\_name)
39. **print**('Email Address: ', obj.email())
41. #setting new value of full\_name
42. obj.full\_name = 'Bruce Banner'
43. **print**('New Full Name is: ', obj.full\_name)
45. #deleting the full\_name
46. **del** obj.full\_name

**Output:**

Full Name is: Nick Fury

Email Address: Nick.Fury@email.com

Full Name is: Nick Fury

Email Address: Nick.Fury@email.com

New Full Name is: Bruce Banner

Deleted the full\_name.

We can also implement these methods using the single line of code.

full\_name = property(full\_name\_getter, full\_name\_setter, full\_name\_deleter)

## Important Tips

We don't need to create all three methods for every property in the code. We can define read-only properties by only including a getter method. We can also avoid the deleter method and use getter and setter. We can avoid the setter method if we want to set the attribute when the instance is created or if it should be modified within the class.

The users are free to choose any method depending on their working context.

## Conclusion

This tutorial has covered property decorators and their examples in detail. The **property()** method helps create getter, setter, and deleter methods. We can define the properties with the @property decorator syntax, which is more compatible and straightforward. It is considered the **'pythonic'** way to define getters, setters, and deleter.

Using the properties, we can modify the class's internal implementation without affecting it. The getter, setter, and deleter help avoid accessing or modifying the data directly.

Python List Comprehension vs. Generator Expression

In this tutorial, we will discuss some important differences between list comprehension and generator expression. Both are quite similar in syntax, but they have some significant differences. Let's have a brief introduction to list comprehension and generator expression.

What is List Comprehension?

List comprehension is an excellent way to create a list. It allows us to create a list using the loop and conditional statement. We can perform the task using the lesser code. While a normal list takes the 3-4 lines of code, it can be done in a single line. Let's understand the following code.

**Example -**

1. # initialize the empty list
2. list = []
4. **for** i **in** range(20):
5. **if** i % 2 == 0:
6. list.append(i)
8. # print elements
9. **print**(list)

**Output:**

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

We can do the same thing using the list comprehension with just a single line of code.

**Example -**

1. list = [i **for** i **in** range(20) **if** i % 2 == 0]
2. **print**(list)

**Output:**

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

As we can see, the syntax of the list compression takes for loop along with the if condition. We can also use the nested for loop but it will reduce the readability.

Let's see another example.

**Example - Using the Normal List**

1. square\_cube=[]
2. **for** num **in** range(10):
3. **if** num%2==0:
4. square\_cube.append(num\*\*2)
5. **else**:
6. square\_cube.append(num\*\*3)
8. **print**(f"square\_cube = {square\_cube}")

**Output:**

square\_cube = [0, 1, 4, 27, 16, 125, 36, 343, 64, 729]

**Example - Using List Comprehension**

1. square\_cube=[n\*\*2 **if** n%2==0 **else** n\*\*3 **for** n **in** range(10)]
2. **print**(f"With List-Conprehension = {square\_cube}")

**Output:**

With List Comprehension = [0, 1, 4, 27, 16, 125, 36, 343, 64, 729]

The both code do the same thing, filters the number based on condition num%2 ==0 and accordingly execute either **square\_cube.append(num\*\*2)** or **square\_cube.append(num\*\*3).**

What are Generator Expressions?

The Generator Expression is similar to the list compression, but it uses parentheses () and square brackets []. Unlike a list, it returns the generator object. Instead of holding the entire sequence in the memory, the generator generates the next element when needed. The normal function terminates when the return statement is called. But the generator function uses the yield statement, which saves the state of the function and can be picked up from the same state whenever the function is called.

The Generator expression provides the facility to create the generator without using the yield keyword.

Let's understand the following example.

**Example -**

1. list\_comprehension = [i **for** i **in** range(20) **if** i % 2 == 0]
2. **print**(list\_comprehension)

**Output:**

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

**Example - Generator Expression**

1. # Generator Expression
2. generator\_expression = (i **for** i **in** range(20) **if** i % 2 == 0)
4. **print**(generator\_expression)

**Output:**

<generator object <genexpr> at 0x7fa0e3f3b580>

The above code returns the generator object; to print the value of the expression, we need to iterate the generator object.

1. **for** i **in** generator\_expression:
2. **print**(i)

**Output:**

0

2

4

6

8

10

12

14

16

18

Here we get all the elements. Now we see the difference between the Generator expression and List Comprehension.

Difference between Generator Expression and List Comprehension

Both the syntaxes are quite similar, the only difference being that the list comprehension is enclosed in square brackets, whereas the generator expression is enclosed in parentheses.

The generator yields the elements, which means it evaluates the demand value. The normal function with the return statement is terminated whenever it encounters a return statement, whereas Python restores the memory for the entire list.

The main advantage of the generator expression over a list is memory efficiency. The generator expression is memory more efficient than the list. Let's understand the following example.

**Example -**

1. # import getsizeof from sys module
2. **from** sys **import** getsizeof
4. comp = [i **for** i **in** range(20000)]
5. gen = (i **for** i **in** range(20000))
7. # gives size for list comprehension
8. x = getsizeof(comp)
9. **print**("x = ", x)
11. #gives size for generator expression
12. y = getsizeof(gen)
13. **print**("y = ", y)

**Output:**

We can see the difference while creating a list and Python reserves memory for the whole list and calculates the entire memory that it will consume. The generator expression generates the next element of the sequence on demand. It calculates memory of the next value.

Generators are slower as the element of the sequence is calculated and yielded every time. The state of the function has to be saved to generate the value for the next time.

**Example -**

1. # List creation
2. **import** timeit
3. **print**(timeit.timeit('''''list\_com = [i for i in range(100) if i % 2 == 0]''', number=1000000))

**Output:**

14.614160199999986

**Example -**

1. **import** timeit
2. **print**(timeit.timeit('''''gen\_exp = (i for i in range(100) if i % 2 == 0)''', number=1000000))

**Output:**

0.9265093000000206

As we can see, there is a significant difference in the execution time. Generator expressions are quite faster than list comprehension and time-efficient.

Conclusion

This tutorial included the basic introduction of the list comprehension and generator expression. We have also learned about some important differences between them. List comprehensions effectively reduce the number of lines, whereas generator expression generates the values on demand. The comprehensions are considered a more Pythonic way of coding by some. It is up to you to choose which one is best for you.

# How to Send Push Notification in Python

In this tutorial, we will learn how to send or push notifications; implement the functionality of push notifications and why we use the user notification.

Push notifications are a beneficial method of communication with the user. These are used to send or notify the user of a significant event or display a symbol and brief details of the text that the user may click on to access the website. Push notifications can include the action button in the notice so that users can get direct access to the site or application without returning. Push notifications are essential in increasing user engagement with the application or website. Let's have a brief introduction to push notifications.

## What are Push Notifications?

Push notifications are clickable pop-up messages that generally include a call to action (CTA). These are beneficial from both customer and company perspectives, and the companies can increase their conversion rate. These text-based messages may appear as browser notifications on your desktop/laptop. We can customize any system, whether a window desktop Mozilla Firefox browser or Windows mobile opera browser.

Below are some use cases of the push notification -

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

## Behavioral Characteristics

Have you ever noticed that you have been looking for a pair of shoes over the website for a long time and don't add to the cart yet? Or you want to have it in the near future. Once the website notices such users, it sends a push notification to the user with social media proof claiming that shoes are popular. We can determine if they are interested in other items by analyzing the user's choice or behavior.

### Location

Gift vouchers are the perfect example of increasing user interaction based on location. This hyper-personalized method analyses visitor movement on the internet. If potential buyers are near your location, sending a push notification with a gift certificate only redeemable at a physical location is a solid strategy.

### Previous Purchases

Every business's growth depends on the repeat customer. The websites are not meant to be someone who visited the website and completed a purchase; we break the communication with them. For example, many businesses send regular users to push notifications to inform them that the brand's most popular goods are on sale. Suppose the user has booked a flight to New York on our website. We can send the customize push Message to any blog or website in the New York area. We can also mention the city's service's expertise to increase customer engagement.

### Reminder to the Inactive Users

Today's average applications have dismally poor retention rates. Popular apps have the high retention and they often leads to the uninstallation of scarcely used application. To avoid a situation, where there is possibility of inactive user can uninstall our application, we can send regular push notifications reminding them how useful our application is to them and how many new features we have added in our application that can enhance the user application.

## Multi-Channel Push Notification

In the current digital era, notifications are the essential attributes of many software products like Instagram and e-commerce shops. Many businesses would not be able to survive without push notifications. They are using push notifications to alert users about the new feature or piece of information they need to know to keep them engaged and excited.

However, multi-channel notifications are a bit tricky. For example - an email works asynchronously; the user may prefer a summary of a hotel room booking activity via email. However, an action requires quick attention, such as a change in room availability status. A website would like to prefer email notification in favor of a push notification.

## How to Send Push Notification

Now we will implement the push notification using Python, which is the motive of this tutorial. We will follow the below steps to implement each type of notification.

### Web Push Notification

Web push notification enables user engagement to the website and timely updates users about the new update on the web. To send the push notification to the users, we must perform an API request to a push notification. This API is used to specify the data to be transmitted, who will receive the Message, and any other scenario to send the Message.

A push service takes a network request, validates it, and sends a push message to the corresponding browser. If the browser is unavailable, the Message waits until the browser becomes available.

The developer can control the browser to send the push notification. This tutorial will use the **Pushbullet** module to send push notifications. The Pushbullet can also be used to send push notifications on mobile devices.

**Prerequisite**

Your system must have the following libraries -

* Pushbullet
* Pywebio
* Python

**Implementation**

Let's implement the web push notification -

**Step - 1:** Install the Pushbullet library using the following command.

1. pip install Pushbullet.py

**Step - 2:** Create an account on Pushbullet if you don't have and get the access token.

**Step - 3:** Add the Pushbullet extension to preferred browser.

**Step - 4:** Navigate to Account in the setting tab and copy your access token.

Let's understand the following code.

**Example -**

1. # Import the following modules
2. **from** pushbullet **import** PushBullet
3. **from** pywebio.input **import** \*
4. **from** pywebio.output **import** \*
5. **from** pywebio.session **import** \*
6. **import** time
7. # Get the access token from Pushbullet.com
8. access\_token = "access\_token "
9. # Taking input from the user
10. data = input('Enter the title: ')
11. # Taking large text input from the user
13. text = textarea(
15. "Text", rows=3, placeholder="Write some text...",
17. required=True)
19. # Get the instance using the access token
21. pb = PushBullet(access\_token)
23. # Send the data by passing the main title
25. # and text to be send
27. push = pb.push\_note(data, text)
29. # Put a success message after sending
31. # the notification
32. put\_success("Message sent successfully...")
33. # Sleep for 3 seconds
34. time.sleep(3)
35. # Clear the screen
36. clear()
37. # Give the pop at last
38. toast("Thanks for Using :)")
39. # hold the session until the whole work finishes
40. hold()

After running the code, the browser will open the new tab and ask to write a notification message. Now, we can enter the notification subject and click on **Submit** to send. Once submitted, we should get the push notification message from the browser.

## Multi-Channel Push Notification

In this section, we will learn how we can send the multi-channel notification using Python.

**Prerequisites**

Following requirements must be in fulfilled your system.

* Python
* An AWS SNS Account
* trycourier - (Python package)

**Implementation**

In this section, we will learn how to send multi-channel notifications using Python.

**Prerequisites**

The following requirements must be fulfilled in your system.

* Python
* An AWS SNS Account
* trycourier - (Python package)

**Implementation**

**Step - 1:** Create an account on [Courier](https://app.courier.com/signup).

**Step - 2:** Now navigate to the integrated tab on the page; here, you will find all notification channel providers, then select AWS SNS.

**Step - 3:** Provide the AWS SNS account and SNS topic and click on Install Provider.

**Step - 4:** Navigate the designer tab and click the Create Notification button.

**Step - 5:** Click on the Publish Changes button.

**Step - 6:** Enter a message for AWS SNS and Gmail in the **Design** tab for the notification you just created; click on the **Publish Changes** button to save each message.

**Step - 7:** Now, create a subscription on your amazon SNS account for the topic you used to create the notification.

**Step - 8:** Copy the following code and replace the token in the code.

**Example -**

1. **from** trycourier **import** Courier
3. client = Courier(auth\_token="your auth token")
5. resp = client.send(
7. event="notification\_id",
9. recipient="recepient\_email",
11. profile={
13. "email": "your\_profile\_email"
15. }
17. )

When we run the above code, you should get the notification.

## Conclusion

This tutorial taught us about push notification, their use cases, and their advantages. We have also implemented the push notification on the web and multi-channel push notification and how to implement it using Courier.

This tutorial will give you a brief idea of push notifications using the Push bullet module and the courier. It also has analytics tools that give you helpful information and metrics about how users engage your notification.

# Python sleep() Function

The sleep() method pauses (waits) the current thread's execution for a specified period of time.

Python contains a time module that offers a number of helpful functions to manage time-related tasks. Sleep is one of the common activities among them ().

The sleep() function pauses the current thread's execution for a specified period of time.

The Python time sleep function is being used to delay a program's execution. To stop the running of the programme for a certain amount of time in seconds, we can use the sleep function in Python. Keep in mind that the Python time sleep function only halts the current thread's execution, not the entire programme.

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

Python comes with built-in functionality for sleeping our programme. The sleep() function in the time module allows us to pause the caller thread's execution for as many seconds we want.

**Code 1: Python sleep()**

1. **import** time
2. **print**("Printed time immediately.")
3. time.sleep(2.5)
4. **print**("Printed after 2.5 seconds.")

**Output:**

"Printed time immediately" is printed

Suspends (Delays) execution for 2.5 seconds.

"Printed after 2.5 seconds" is printed.

The aforementioned example demonstrates that sleep() accepts a floating-point integer as an input. Prior to Python 3.5, the suspension time could actually be shorter than the value supplied as an argument to the time() method.

The suspension period is at least the set number of seconds starting with Python 3.5. The core developers of Python 3.5 made a little adjustment to the way time.sleep() behaves. Even though the sleep is stopped by a signal, the newer Python sleep() system will continue for at least the amount of seconds we've chosen. However, if the message itself produces an exception, then this does not apply.

**Code 2: Python in Create a Digital Clock**

1. **import** time
2. **while** True:
3. disttime = time.disttime()
4. ans = time.strftime("%I:%M:%S %p", disttime)
5. **print**(ans)
6. time.sleep(1)

Within the endless while loop of the programme shown above, we computed and reported the current local time. The programme then waits for a full second. The current local time is once more calculated and printed. This procedure continues.

**Output:**

02:10:50 PM

02:10:51 PM

02:10:52 PM

02:10:53 PM

02:10:54 PM

**Code 2: Improved Version**

Here is a significantly improved version of the previously mentioned software.

1. **import** time
2. **while** True:
3. disttime = time.disttime()
4. ans = time.strftime("%I:%M:%S %p", disttime)
5. **print**(ans, end="", flush=True)
6. **print**("\r", end="", flush=True)
7. time.sleep(1)

**Output:**

02:10:50 PM

02:10:51 PM

02:10:52 PM

02:10:53 PM

02:10:54 PM

## Multithreading in Python:

Let's speak about groups and thread first before discussing sleep() in multithreaded systems. There are further instances in which we would want to provide a thread a Python sleep() function. Maybe we're using a production database containing millions of documents and a migration script.

We choose to use threads because we would not want to create any downtime but we really do not want to wait any longer than required for the migration to be finished.

Each instruction in a computer programme is a separate unit. The carrying out of those directives constitutes a procedure. A part of the process is called a thread. One or even more threads may be present in a process.

**Code 3: Python Multithreading**

The tutorial programmes listed above are all single-threaded programmes. Here is an instance of a Python programme with several threads.

1. **import** threading
2. **def** print\_hola\_3\_times():
3. **for** a **in** range(3):
4. **print**("Hola")
5. **def** print\_hey\_3\_times():
6. **for** a **in** range(3):
7. **print**("Hey")
8. th1 = threading.Thread(target=print\_hola\_3\_times)
9. th2 = threading.Thread(target=print\_hey\_3\_times)
10. th1.start()
11. th2.start()

**Output:**

Hola

Hola

Hey

Hola

Hey

Hey

The software mentioned above has two threads, th1 and th2. The statements th1.start() and th2.start() are used to start these threads. Keep in mind that th1 and th2 may run concurrently and produce different results.

### time.sleep() in Python Multithreaded Programs

The sleep() function pauses the current thread's execution for a specified period of time.

Sleep() pauses the operation of the thread and processes in single-threaded programmes. In multithreaded programmes, the function halts a thread instead of the entire operation.

**Code 4: sleep() in a Multithreaded Program**

1. **import** time
2. **from** threading **import** Thread
3. **class** make(Thread):
4. **def** run(self):
5. **for** m **in** xrange(0,11):
6. **print** m
7. time.sleep(1)
8. **class** made(Thread):
9. **def** run(self):
10. **for** m **in** xrange(100,103):
11. **print** m
12. time.sleep(5)
13. **def** run():
14. make().start()
15. made().start()

**Output:**

0

100

>>> 1

2

3

4

5

101

6

7

8

9

10

102

# Find Key from Value in Dictionary

Python dictionary is a type of data structure that makes it simple to write highly effective code. Since we can hash its keys, a dictionary data structure is a type of hash table in several other languages. A data structure called a hash table uses associative coding to store data. Each item of the dictionary is kept in a hash table in an array format with a distinct index value. If we get to know the index for the desired data, accessing it becomes pretty quick.

The keys and their corresponding are arranged in pairs in Python dictionaries. We can compare them to words and their interpretations from an ordinary dictionary. The interpreter maps the keys and values to each other. For example, the key "Data Science" is linked to the description of science that uses computer tools to look for patterns in large amounts of data in an actual dictionary.

We can use Python's dict\_name[key\_name] syntax to retrieve the values contained in a dictionary using its keys. But even if we have values, there is no direct way to retrieve the key that goes with them. In this post, we'll examine the strategies for obtaining the key through the value of that key.

## Methods

Let's examine the Python Dictionary's key-by-value retrieval method.

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

### Identify A Key Using Its Value by Searching the Dictionary Items

The easiest way to exchange a key for its value is in this way. We will use this approach to examine each key-value combination to identify the key corresponding to the present value. We shall employ the items() function of the Python dictionary for this operation. Using the items() method, we can get a list of tuples with key-value pairs in the dictionary. We will find the key linked to our value by searching each tuple in the manner described below.

**Code**

1. # Python program to get a key by its value by searching for the value in the dictionary
3. # Creating a dictionary
4. Dict = {"Name": "Javatpoint", "Language": "Python", "Data Structure": "Dictionary"}
5. # Printing the dictionary
6. **print**("Our dictionary is:", Dict)
8. # Getting the tuples of key:value pairs present in the dictionary using the items() function
9. items = Dict.items()
11. # Storing the value whose key is to be retrieved
12. **print**("Value whose key is to be retrieved:")
13. Value = "Python"
14. **print**(Value)
16. # Getting the key of the stored value
17. **for** key, value **in** items:
18. **if** value == Value:
20. **print**(f"The key of {Value} in the dictionary is: ", key)

**Output:**

Our dictionary is: {'Name': 'Javatpoint', 'Language': 'Python', 'Data Structure': 'Dictionary'}

Value whose key is to be retrieved:

Python

The key of Python in the dictionary is: Language

### Using Python Lists, Obtain The Key from A Value

We can make a separate Python list of keys and values and then use the index() function to get the keys from the supplied value. For this assignment, we will first use the keys() function to generate a Python list of the keys contained in the given dictionary, after which we will use the values() function to make a Python list of the values included in the dictionary.

Using the index() function, we will retrieve the supplied value's index from the list of values. The index of the dictionary's values in a Python list of values will match the index of the related key in the Python list of keys because we know that the ordering of the keys in the Python list of keys corresponds to the order of the values added in the Python list of values.

Therefore, after locating the value's index inside the Python list of values, one can identify the key's index in the list of keys in the exact location. The next is how to accomplish this.

**Code**

1. # Python program to get a key by its value using lists of keys and values of the dictionary and index() method
3. # Creating a dictionary
4. Dict = {"Name": "Javatpoint", "Language": "Python", "Data Structure": "Dictionary"}
6. # Printing the dictionary
7. **print**("Our dictionary is:", Dict)
9. # Storing the keys and the values of the dictionary in separate Python lists
10. keys = list(Dict.keys())
11. values = list(Dict.values())
13. # Storing the value whose key is to be retrieved
14. **print**("Value whose key is to be retrieved:")
15. Value = "Python"
16. **print**(Value)
18. # Getting the index of the stored value from the list of values
19. index = values.index(Value)
21. # Getting the key of the value
22. Key = keys[index]
23. **print**(f"The key of {Value} in the dictionary is: ", Key)

**Output:**

Our dictionary is: {'Name': 'Javatpoint', 'Language': 'Python', 'Data Structure': 'Dictionary'}

Value whose key is to be retrieved:

Python

The key of Python in the dictionary is: Language

### Using List Comprehension to Determine the Key from A Value

We can use list comprehension to obtain the keys connected to the supplied value in place of the index() method. We will use list comprehension to generate a list of keys with associated values identical to the given value to find the key.

**Code**

1. # Python program to get a key by its value using list comprehension
3. # Creating a dictionary
4. Dict = {"Name": "Javatpoint", "Language": "Python", "Data Structure": "Dictionary"}
6. # Printing the dictionary
7. **print**("Our dictionary is:", Dict)
9. # Getting the key:value pairs present in the dictionary
10. items = Dict.items()
12. # Storing the value whose key is to be retrieved
13. **print**("Value whose key is to be retrieved:")
14. Value = "Python"
15. **print**(Value)
17. # Getting the key of the value
18. Key = [key **for** key, value **in** items **if** value == Value]
19. **print**(f"The key of {Value} in the dictionary is: ", Key)

**Output:**

Our dictionary is: {'Name': 'Javatpoint', 'Language': 'Python', 'Data Structure': 'Dictionary'}

Value whose key is to be retrieved:

Python

The key of Python in the dictionary is: ['Language']

# What Is Sleeping Time in Python?

Have you ever been forced to delay the execution of a Python programme? You typically wish your code to run as rapidly as possible. There are, however, instances when it serves your best interests to put your program to sleep for a while.

For instance, to mimic a pause in our programme, we may use Python's sleep() method. We may need to wait while data uploads or downloads; the graphic loads or is shown to the screen. We may need to take a break even amid requests to a web API or database operations. Each of these situations and many more can benefit from inserting Python sleep() methods into our code.

## Python time.sleep()

The Python time sleep method is applied to pause a program's execution. We can employ the sleep method of the time module in our Python programs to stop the execution of the programme for a certain amount of time in seconds. Remember that the Python time sleep method halts the current thread's execution, not the entire programme.

### Python time.sleep() Function Syntax

The sleep() function is a method of the time module. Therefore, before using this function, we must first import the time module into our program. Following is the syntax of using the sleep function.

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

1. **import** time
2. time.sleep(t)

The sleep() method's input, t, is the time the programme is to be delayed which is given in seconds. That indicates that the subsequent code line would run after the expression time.sleep(t) has been executed after t seconds of execution of the previous code line. See the example below:

**Code**

1. # Python program to experience the delayed execution due to sleep function
3. # importing the required module
4. **import** time
6. # printing time at the beginning of the execution
7. **print**(time.ctime())
9. **print**("Before delay of the program")
10. time.sleep(50)
11. **print**("After delay of the program")
13. # printing time at the end of the execution
14. **print**(time.ctime())

**Output:**

Sun Aug 28 17:07:33 2022

Before delay of the program

After delay of the program

Sun Aug 28 17:08:23 2022

The second print statement is executed after 50 seconds when the mentioned code is invoked. Consequently, we can add delays as needed to our code. To have a more precise delay in execution, we can also give the parameter t in a floating point number. For instance, if we wish to add a delay of 10 milliseconds (or 0.01 seconds), do as follows:

**Code**

1. # Python program to delay the program by 100 milliseconds
3. # importing the required module
4. **import** time
6. # printing epoch time at the beginning of the execution
7. **print**(time.time())
9. **print**("Before delay of the program")
10. time.sleep(0.1)
11. **print**("After delay of the program")
13. # printing epoch time at the end of the execution
14. **print**(time.time())

**Output:**

1661706874.95425

Before delay of the program

After delay of the program

1661706875.0580375

### Python time.sleep Example

Let's look at the next Python time.sleep() function example.

**Code**

1. # Python program to use sleep function inside the Python loop
3. # Importing the time module
4. **import** time
6. # Time at the beginning of the execution of the program
7. start = time.ctime()
8. **print**(start)
10. **for** i **in** range(10):
11. # using the time.sleep() method to halt the execution
12. time.sleep(5)
13. **print**(i)
14. end = time.ctime()
15. **print**(end)

**Output:**

Sun Aug 28 17:19:50 2022

0

1

2

3

4

5

6

7

8

9

Sun Aug 28 17:20:40 2022

Due to the for loop's 5-second timeouts, the total time spent is greater than 5. The additional time is a result of the program's processing time, operating system thread sequencing, etc.

### Creating a Delay in Time in a Python List

**Code**

1. # Python program to create a time delay in a Python list
3. # importing the required package
4. **import** time
6. # Marking the beginning of the time
7. b = time.ctime()
8. **print**(b)
10. # creating a delay of 10 seconds
11. time.sleep(10)
13. # Initializing the Python list
14. my\_List = ["Time", "Sleep", "Python"]
16. # Printing the list
17. **print**(my\_List)
19. # Marking the end of the program
20. e = time.ctime()
21. **print**(e)
22. # List will be displayed after a time difference of 10 seconds from the beginning of the program

**Output:**

Thu Sep 1 12:28:05 2022

['Time', 'Sleep', 'Python']

Thu Sep 1 12:28:15 2022

### Creating a Delay in Time in a Python Tuple

**Code**

1. # Python program to create a time delay in a Python tuple
3. # importing the required package
4. **import** time
6. # Recording the beginning of the program
7. b = time.ctime()
8. **print**(b)
10. # creating a delay of 15 seconds in the execution
11. time.sleep(15)
13. # Initializing the Python tuple
14. my\_tuple = ("Time", "Sleep", "Python")
16. # Printing the tuple
17. **print**(my\_tuple)
19. # Recording the end of the program
20. e = time.ctime()
21. **print**(e)
22. # Tuple will be displayed after 15 seconds of delay in the execution

**Output:**

Thu Sep 1 12:32:31 2022

('Time', 'Sleep', 'Python')

Thu Sep 1 12:32:46 2022

### Creating a Delay in Time Delay in List Comprehension

**Code**

1. # Python program to create a time delay in a list comprehension
3. # importing the required package
4. **import** time
6. # Displaying the beginning of the execution
7. b = time.ctime()
8. **print**(b)
10. # creating and Initializing a list
11. listt = ['Delay', 'Time', 'Sleep', 'Python', 'Time', 'Module']
13. # a time delay of 10 seconds is inserted in between creating the list comprehension and printing the components of the list
14. listt = [(time.sleep(10), **print**(l)) **for** l **in** listt]

17. # Displaying the end of the execution
18. e = time.ctime()
19. **print**(e)

**Output:**

Thu Sep 1 12:38:03 2022

Delay

Time

Sleep

Python

Time

Module

Thu Sep 1 12:39:03 2022

### Creating Many Time Delays in a Program

**Code**

1. # Python program to create a time delay in a list comprehension
3. # importing the required package
4. **import** time
6. # Displaying the start of the execution
7. b = time.ctime()
8. **print**(b)
10. # creating and Initializing a list
11. languages = ['Python', 'C', 'C++', 'R', 'Java', 'JavaScript', 'HTML']
13. # Halting the execution for 2 seconds here
14. time.sleep(2)
16. # Interpreter will execute this command after 2 seconds
17. **print**(languages)
19. **for** l **in** languages:
20. # Halting the execution for 5 seconds here
21. time.sleep(5)
22. # A component of the list will be delayed after every 5 seconds
23. **print**(l)
25. # Displaying the end of the execution
26. # There must be a delay of 37 seconds in the complete execution of the program
27. e = time.ctime()
28. **print**(e)

**Output:**

Thu Sep 1 12:44:39 2022

['Python', 'C', 'C++', 'R', 'Java', 'JavaScript', 'HTML']

Python

C

C++

R

Java

JavaScript

HTML

Thu Sep 1 12:45:16 2022

### Python Thread Sleep

The sleep() method in Python is a crucial multithreading technique. The straightforward example demonstrates how the Python time sleep method merely halts the operation of the currently running thread in multithreaded coding.

**Code**

1. # Python program to show how to use the sleep() function in a thread
3. # Importing the required modules
4. **import** time
5. **from** threading **import** Thread
7. # Beginning a thread
8. **class** a\_thread(Thread):
9. **def** func(self):
10. **for** i **in** range(10):
11. **print**(i)
12. time.sleep(5)
14. **class** Delay(Thread):
15. **def** func(self):
16. **for** i **in** range(100, 105):
17. **print**(i)
18. time.sleep(10)
20. **print**("Starting the thread a\_thread")
21. a\_thread().start()
22. **print**("Starting the delay thread")
23. Delay().start()
24. **print**("Complete")

### Including a Python time.sleep() Request With Decorators

We are occasionally required to try a failed operation again. This is a frequent use scenario when we have to repeat a file downloading since the server was overloaded. Including a Python sleep() function between every request is preferable because we usually won't like to send requests to the server frequently.

I've also had to monitor the status of a UI while running an automated test, yet another scenario. The user interface may load quicker or slower than expected based on the machine we are using to test, which could alter the information displayed on the screen when my software is checking anything.

In this situation, we can instruct the application to take a little break and verify everything again a moment or two later. This may determine whether a test is passed or failed.

In either of these scenarios, we may add a Python sleep() method using a decorator. Let's examine an illustration:

**Code**

1. **import** time
3. **def** sleep(timeout, retry = 4):
4. **def** real\_decorator(function):
5. **def** a\_wrapper(\*args, \*\*kwargs):
6. retry\_no = 0
7. **while** retry\_no < retry:
8. **try**:
9. v = function(\*args, \*\*kwargs)
10. **if** v **is** None:
11. **return**
12. **except**:
13. **print**(f'This is sleeping for {timeout} seconds')
14. time.sleep(timeout)
15. retry\_no += 1
16. **return** a\_wrapper
17. **return** real\_decorator

This is the decorator in the above code: sleep(). It receives a time window and a retry count, with a default setting of three. The real decorator(), a function that takes the decorated method, is contained within the sleep() method.

Last but not least, the inner method wrapper() takes both keyword arguments and parameters passed to the decorated method. The magic happens right here! We try invoking the procedure once more by using a while loop. If there is an exception raised, we should declare it over. Try executing the function once more after using sleep() and increasing the number of retries.

# Attributes Meaning in Python

If we have been working with OOPS paradigm languages, we should've heard the word "attributes" quite often. There are two types of attributes-**class attributes and instance attributes**. This tutorial explains what attributes are and their types, along with examples.

As Python is a huge supporter of the OOPS paradigm, everything is regarded as an object. We create classes and objects. Before understanding attributes, understanding classes and objects is the foundation. In a simple sentence**, a class is like a blueprint and to use that blueprint, we create objects for that class**.

**Example:**

1. **class** student:
2. sch\_name = "Delhi public school"
3. place = "Delhi"
4. grade = 5
5. **def** display(self):
6. **print**("School:", self.sch\_name)
7. **print**("Place:", self.place)
8. **print**("Grade:", self.grade)
9. s1 = student()
10. s2 = student()
11. **print**(s1.sch\_name)
12. s1.display()
13. **print**(s2.sch\_name)
14. s2.display()

**Output:**

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

Delhi public school

School: Delhi public school

Place: Delhi

grade: 5

Delhi public school

School: Delhi public school

Place: Delhi

grade: 5

**Understanding:**

In the above example, we created a class called "student" with three variables and a function that prints the values of the three variables**. Every object we create for the class has access to the variables in the class. These variables are called the "Attributes"**, and the functions are called methods (also referred to as functions). In the above program, we created the class's two objects, s1 and s2, and accessed the attribute sch\_name and the method display() printed the values of the attributes.

## Types of Attributes:

### Class Attributes:

* In the previous example, we declared attributes that don't change, which means that for any object we create for the class, **the values of the attributes remain the same.**
* Such attributes are called the "**Class attributes**".

**Example:**

1. **class** Address:
2. University = "Harvard"
3. City = "Cambridge"
4. State = "Massachusetts"
5. Capital = "Boston"
6. Country = "United States"
7. student1 = Address()
8. **print**("Student1: Ryan")
9. **print**(student1.University,"",student1.City,"",student1.State,"",student1.Capital,"",student1.Country)
10. student2 = Address()
11. **print**("Student2: Tony")
12. **print**(student2.University,"",student2.City,"",student2.State,"",student2.Capital,"",student2.Country)

**Output:**

Student1: Ryan

Harvard Cambridge Massachusetts Boston United States

Student2: Tony

Harvard Cambridge Massachusetts Boston United States

* Observe that the information in the class is storing the same values representing the address of the university and we used the same information for all the objects-students.

### Instance Attributes:

* What if we wanted to create a class that allows **every object to have its attributes**? It is possible too. For example, every object should hold information about an individual student.
* Such class attributes that can let objects hold different values for the same attribute are called the "**Instance attributes**".
* We need to use **the \_\_init\_\_ method and self variable** to declare these attributes.
* **\_\_init\_\_ method:** It is a constructor that allows objects to have different values for the same attribute. It is executed as soon as an object is created for the class without any manual call.
* **self variable:** It is like a placeholder for the object created for the class inside the \_\_init\_\_ method:

**Example:**

1. **class** Student\_info:
2. **def** \_\_init\_\_(self, name, age, college):
3. self.name = name;
4. self.age = age;
5. self.college = college;
6. student1 = Student\_info("Ryan", 18, "NYU")
7. student2 = Student\_info("Roy", 18, "Duke")
8. **print**("Student1: ")
9. **print**(student1.name, student1.age, student1.college)
10. **print**("Student2: ")
11. **print**(student2.name, student2.age, student2.college)

**Output:**

Student1:

Ryan 18 NYU

Student1:

Roy 18 Duke

* We created two objects, student1 and student2, and the two objects hold different information about two individuals.
* What happens here is that once an object is created, \_\_init\_\_ method is invoked, and the object will replace the self variable in the method:

**When we created student1:**

1. **def** \_\_init\_\_(student1, "Ryan", 18, "NYU"):
2. student1.name = "Ryan";
3. student1.age = 18;
4. student1.college = "NYU";

**When we created student2:**

1. **def** \_\_init\_\_(student2, "Roy", 18, "Duke"):
2. student2.name = "Roy";
3. student2.age = 18;
4. student2.college = "Duke";

Both Instance attributes and Class attributes play a prominent role in creating a sound class. Here is an example using both class variables and instance variables to create a class:

1. **class** Address:
2. University = "Harvard"
3. City = "Cambridge"
4. State = "Massachusetts"
5. Capital = "Boston"
6. Country = "United States"
7. **def** \_\_init\_\_(self, name, age, CGPA):
8. self.name = name
9. self.age = age
10. self.CGPA = CGPA
11. student1 = Address("Ryan", 18, 9.2)
12. **print**("Student1:")
13. **print**(student1.name, student1.age, student1.CGPA)
14. **print**(student1.University, student1.City, student1.State, student1.Capital, student1.Country)
15. **print**("\nStudent2:")
16. student2 = Address("Roy", 18, 9.1)
17. **print**(student2.name, student2.age, student2.CGPA)
18. **print**(student2.University, student2.City, student2.State, student2.Capital, student2.Country)

**Output:**

Student1:

Ryan 18 9.2

Harvard Cambridge Massachusetts Boston United States

Student2:

Roy 18 9.1

Harvard Cambridge Massachusetts Boston United States

* We used class attributes to store the data common to all the objects and instance attributes for storing distinct data of every single object.
* Also, observe that:

1. Class attributes will always be outside the \_\_init\_\_ method.
2. Instance attributes are defined inside the \_\_init\_\_ method.

* **In the Above Code:**

**Class attributes:**

1. University
2. City
3. State
4. Capital
5. Country

**Instance attributes:**

1. name
2. age
3. CGPA

**Function to display the attributes in a program:**

Python provides two built-in functions that can print the attributes of a particular object:

1. **dir()** -> It **returns both class attributes and instance attributes** of the object along with the attributes of the ancestor classes of the object.
2. **vars()** -> It returns the **instance attributes** of an object in the form of a dictionary.

**Example:**

1. **print**(vars(student1))
2. **print**()
3. **print**(dir(student1))

**Output:**

{'name': 'Ryan', 'age': 18, 'CGPA': 9.2}

['CGPA', 'Capital', 'City', 'Country', 'State', 'University', '\_\_class\_\_', '\_\_delatt

# Arguments and Parameters in Python

Be it any programming language, Arguments and Parameters are the two words that cause a lot of confusion to programmers. Sometimes, these two words are used interchangeably, but actually, they have two different yet similar meanings. This tutorial explains the differences between these two words and dives deep into the concepts with examples.

Both arguments and parameters are variables/ constants passed into a function. The difference is that:

1. Arguments are the variables passed to the function in the function call.
2. Parameters are the variables used in the function definition.
3. The number of arguments and parameters should always be equal except for the variable length argument list.

### Example:

1. **def** add\_func(a,b):
2. sum = a + b
3. **return** sum
4. num1 = int(input("Enter the value of the first number: "))
5. num2 = int(input("Enter the value of the second number: "))
6. **print**("Sum of two numbers: ",add\_func(num1, num2))

**Output:**

Enter the value of the first number: 5

Enter the value of the second number: 2

Sum of two numbers: 7

### Points to grasp from the Example:

1. (num1, num2) are in the function call, and (a, b) are in the function definition.
2. (num1, num2) are arguments and (a, b) are parameters.

### Mechanism:

Observe that in the above example, num1 and num2 are the values in the function call with which we called the function. When the function is invoked, a and b are replaced with num1 and num2, the operation is performed on the arguments, and the result is returned.

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

Functions are written to avoid writing frequently used logic again and again. To write a general logic, we use some variables, which are **parameters**. They belong to the function definition. When we need the function while writing our program, we need to apply the function logic on the variables we used in our program, called the **arguments**. We then call the function with the arguments.

## Types of Arguments:

Based on how we pass arguments to parameters, arguments are of two types:

1. Positional arguments
2. Keyword arguments

* Given some parameters, if the respective arguments are passed **in order** one after the other, those arguments are called the "Positional arguments."
* If the arguments are passed by **assigning** them to their respective parameters in the function call with no significance to the passing order, they are called "Keyword arguments".

**Example:**

1. **def** details(name, age, grade):
2. **print**("Details of student:", name)
3. **print**("age: ", age)
4. **print**("grade: ", grade)
5. details("Raghav", 12, 6)
6. details("Santhosh", grade = 6, age = 12)

**Output:**

Details of student: Raghav

age: 12

grade: 6

Details of student: Santhosh

age: 12

grade: 6

**Points to grasp from the Example:**

**First Function Call:**

* The function has three parameters-name, age and grade. So, it accepts three arguments.
* In the first function call:

1. details("Raghav", 12, 6)

The arguments are passed **position-wise** to the parameters, which mean according to the passed order:

* name is replaced with "Raghav."
* age is replaced with 12 and
* grade is replaced with 6
* In the first function call, the order of passing the arguments matter. The parameters accept the arguments in the given order only.
* **In the Second Function Call:**

1. details("Santhosh", grade = 6, age = 12)

Here, the first argument, "Santhosh", is passed based on its position to name, and the next two arguments are passed by assignment to their respective parameters. As you can observe, here, the position didn't matter.

### Important Point:

* **Keyword arguments must always follow positional arguments**. If not, Python will raise a syntax error:

If we write: details("Santhosh", age = 6, 12)

1. details("Santhosh", age = 6, 12)
2. ^
3. SyntaxError: positional argument follows keyword argument

### Call by Value and Call by Reference:

This is the most important concept on arguments and parameters. Based on the type of arguments passed to the parameters, there are two methods of invoking/ calling functions-Call by value and Call by reference.

When the values of arguments are passed into parameters in the function, the values are copied into parameters. This method is called "**Call by value**".

In this method, arguments and parameters are different and are stored in different memory locations.

* Changes done on parameters inside the function do not affect the arguments in the program and vice versa.
* **Java** functions/ methods follow only Call by value.

When the addresses of the arguments are passed into parameters instead of values, this method of invoking a function is called "**Call by Reference**".

* Both the arguments and parameters refer to the same memory location.
* Changes to the parameters (pointers) will affect the values of the arguments in the program.
* By default, **C language** follows Call by value, but using the indirection operator and pointers; we can simulate Call by reference.

## Which Method does Python Follow?

Python doesn't use Call by value or Call by reference. It follows a method called "**Call by assignment**". In Python, every single entity is an **object**. Objects are divided into Mutable and Immutable objects. What happens in Python when we assign a value to a variable is different from other low-level languages like C or Java.

Suppose, in the statement:

a = 20

a is the variable, and 20 is the value assigned. Here in a memory location, 20 is saved, and a is the name we're giving to the reference we're making to the memory location. Now, if we say:

a = 21

The name stops referring to the memory location with 20 and starts to refer to another memory location with 21.

In other languages like C, variables are the memory locations that store the values.

### Example:

**In C:**

1. #include<stdio.h>
2. **int** main()
3. {
4. **int** a;
5. a = 20;
6. printf("%p", (**void**\*)&a);
7. a = 22;
8. printf("\n%p", (**void**\*)&a);
9. }

**Output:**

000000000062FE1C

000000000062FE1C

**In Python:**

1. a = 20
2. **print**(id(a))
3. a = 21
4. **print**(id(a))

**Output:**

140714950863232

140714950863264

* As you can observe: In C, after reassigning the value, the variable is still in the same memory location, while in Python, it refers to a different memory location. (id -> address in Python).
* But that's not all. There are other types of objects too.

Now comes the concept of Mutable and Immutable objects in Python.

### Mutable and Immutable Objects in Python:

1. Mutable objects are those objects/ data types in Python that we can modify after creating them Ex: Lists, Dictionaries, Sets
2. Immutable objects, on the other hand, are objects that can't be modified once created. Ex: int, float, strings, tuples

**Example:**

**Mutable Objects:**

1. a = [23, 45, 89]
2. **print**(id(a))
3. a.append(49)
4. **print**(id(a))

**Output:**

2253724439168

2253724439168

**Understanding:**

A list is immutable, which means we can alter or modify it after creating it. As you can observe, when created with the name a, it is saved in the address "2253724439168". Using append(), we altered it by appending another value. It is still in the same memory location, meaning the same object is modified.

**Immutable Objects:**

1. a = 20
2. **print**(id(a))
3. a += 23
4. **print**(id(a))

**Output:**

140714950863232

140714950863968

**Understanding:**

This is the case we discussed before in the tutorial. An int object is immutable, meaning we can't modify it once created. You might wonder we still added 23 in the above code. Observe that the object when created is not the same object after adding. Both are in different memory locations which means they are different objects.

**So, how are arguments passed to the parameters when a function is invoked?**

With all the knowledge about assignment operation in Python:

1. The passing is like a "Call by Reference" if the arguments are mutable.
2. The passing is like "Call by Value" if the arguments are immutable.

**Example:**

1. **def** details(name, age, grade, marks):
2. marks.append(26)
3. name += " Styles"
4. **print**("Details of the student: ")
5. **print**("name: ",name)
6. **print**("age: ",age)
7. **print**("grade: ", grade)
8. **print**("marks: ", marks)
9. name = "Harry"
10. age = 15
11. grade = 10
12. marks = [25, 29, 21, 30]
13. details (name, age, grade, marks)
14. **print**(grade)
15. **print**(marks)

**Output:**

Details of the student:

name: Harry Styles

age: 15

grade: 10

marks: [25, 29, 2F1, 30, 26]

10

[25, 29, 21, 30, 26]

**Understanding:**

The function accepts 4 arguments. Notice the arguments **grade** and **marks**. grade is an integer value which means it is immutable. Hence, once created, we can't modify it. It follows "Call by Value". As we discussed earlier in the tutorial, when following Call by reference, "**Changes done on the parameters (pointers) will not affect the values of the arguments in the program**". Hence, the original value of grade in the program is not modified after concatenating the string in the function definition.

In the case of marks, it is a list and is mutable. So, it follows "Call by Reference," which means, "**Changes done on the parameters (pointers) will affect the values of the arguments in the program"**. Hence, the change is reflected in the original program after appending the list in the function definition.

# any() in Python

Imagine a situation where we must check 100 conditions to operate. Traditionally, we use conditional statements (if, if-else, elif, nested if) to check if conditions are true. But, for that many conditions, the code becomes lengthy and too many if statements take away the readability of the code.

For such situations, the programmer's choice has to be to use Python's any() function. This tutorial explains the use and functionality of the any () function with examples.

### Example for any():

1. list1 = [True, False, True, False]
2. **print**(any(list1))
3. list1 = [False, False, False, False]
4. **print**(any(list1))
5. list1 = [True, True, True, True]
6. **print**(any(list1))

**Output:**

True

False

True

### Understanding:

In the three cases in the above code, any() returned False only in the second scenario where all the conditions are False. The function requires at least one condition to be True to return True.

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

### Mechanism:

**Lazy evaluation:** When we give more than one expression/value to the function, it keeps evaluating each expression from left to right, and once it finds a True expression, it stops evaluation and returns True. It won't even check the succeeding expressions.

**Syntax:**

1. any(iterable)

**iterable:** Any object that can be iterated using loops like a tuple, list, dictionary, set etc. (int, float etc., are not iterable).

* The function returns True if at least one of the conditions or values evaluates to True else, it returns False. (returns Boolean-True or False)

**Equivalent Boolean values:**

|  |  |
| --- | --- |
| **Token** | **Equivalent Boolean value** |
| 0, 0.0 | False |
| Non-zero integer, float, double... | True |
| An empty string, list, tuple...any iterable | False |
| Non-empty iterable | True |
| None | False |

* Using the bool(x) function, we can find the equivalent Boolean values of any token in Python.

**Sample Code:**

1. **print**("0: ", bool(0))
2. **print**("Some non-zero integer: ", bool(78))
3. **print**("0.0: ", bool(0.0))
4. **print**("Some non-zero float: ", bool(8.90))
5. **print**("Empty string: ", bool(""))
6. **print**("Non-empty string: ", bool("Hi!"))
7. **print**("Empty list: ", bool([]))
8. **print**("Non-empty list: ", bool([2, 3, 5]))
9. **print**("None: ", bool(None))

**Output:**

0: False

Some non-zero integer: True

0.0: False

Some non-zero float: True

Empty string: False

Non-empty string: True

Empty list: False

Non-empty list: True

None: False

* Knowing these values, we can use the any () function accurately.

**Example usage Scenarios:**

Generally, if a code is successful, it returns 0; if any unexpected error occurs, it returns some number. Imagine running many test cases and seeing if all the test cases returned 0. We can use any() to see if any test case returned a non-zero value.

**Code:**

1. **print**("Find if all the test cases returned zero")
2. results = [0, 0, 0, 3.8343, 0, 0, 0]
3. **if**(any(results)==True):
4. **print**("At least one test case returned a non-zero number")
5. **else**:
6. **print**("All the test cases returned success")

**Output:**

Find if all the test cases returned zero

At least one test case returned a non-zero number

To find if any non-empty string is included in a set of empty strings:

**Code:**

1. **print**("Find if any empty string is created by accident")
2. results = ["", "", "", "H"]
3. **if**(any(results) == True):
4. **print**("At least one character is included in some string")
5. **else**:
6. **print**("All are empty strings")

**Output:**

Find if any empty string is created by accident

Atleast one character is included in some string

**any with lists, tuples and sets:**

**Example:**

1. **print**("Lists: ")
2. list1 = [0, 0, 0.0, ""]
3. list2 = [1, 4, 9.0, "Hi"]
4. **print**(list1,":",any(list1))
5. **print**(list2,":",any(list2))
6. **print**("Tuples: ")
7. tuple1 = (0, 0, 0.0, "")
8. tuple2 = (1, 4, 9.0, "Hi")
9. **print**(tuple1,":",any(tuple1))
10. **print**(tuple2,":",any(tuple2))
11. **print**("Sets: ")
12. set1 = {0, 0, 0.0, ""}
13. set2 = {1, 4, 9.0, "Hi"}
14. **print**(set1,":",any(set1))
15. **print**(set2,":",any(set2))

**Output:**

Lists:

[0, 0, 0.0, ''] : False

[1, 4, 9.0, 'Hi'] : True

Tuples:

(0, 0, 0.0, '') : False

(1, 4, 9.0, 'Hi') : True

Sets:

{0, ''} : False

{1, 4, 9.0, 'Hi'} : True

### any() with Dictionary:

In the case of a dictionary, any checks if at least one key in the dictionary is True. If all the keys are False, it returns False.

* If an empty dictionary is given as an argument, it returns False

**Example:**

1. **print**("Dictionary: ")
2. dict1 = {0: "Hi", False: "Hey"}
3. dict2 = {}
4. dict3 = {0: "", 1: "Hello"}
5. **print**(dict1,":",any(dict1))
6. **print**(dict2,":",any(dict2))
7. **print**(dict3,":",any(dict3))

**Output:**

Dictionary:

{0: 'Hey'} : False

{} : False

{0: '', 1: 'Hello'} : True

### any() with a Condition:

We can use the any () function to check if at least one element in the iterable satisfies a specified condition.

**Example:**

1. list1 = [0, -4, -3, 2, -1]
2. **print**("The list: ", list1)
3. result = any(i > 0 **for** i **in** list1)
4. **if**(result==True):
5. **print**("At least one element in the list is greater than 0")
6. **else**:
7. **print**("Not even one element in the list is greater than 0")

**Output:**

The list: [0, -4, -3, 2, -1]

At least one element in the list is greater than 0

### OR and any:

'OR' is one of Python's Boolean operators. It returns true if at least one condition evaluates to True. Both OR and any follow the lazy approach, and both have the same type of functionality and are not quite the same. Here are some examples showing the similarity between both.

**Example:**

1. **print**(True **or** False)
2. **print**(any([True, False]))
3. **print**(True **or** False **or** True **or** False)
4. **print**(any([True, False, True, False]))

**Output:**

True

True

True

True

**Here are the differences:**

|  |  |
| --- | --- |
| **OR** | **ANY()** |
| It is an operator | It is a function |
| It returns the first True value **print(78 or 0)** O/P: 78 | It returns the Boolean True or False **print(any([78, 0]))** O/P: True |
| It is used between values as it is an operator. | It accepts only one argument, which has to be iterable with any number of values-list, tuple… |
| **print(0 or 0 or 0 or 0 or "Hi")** O/P: Hi | **print(0 or 0 or 0 or 0 or "Hi")** O/P: True |

**Example:**

1. list1 = ["Hi", "Hey", "Hello"]
2. **print**(any(list1))
3. **print**(list1[0] **or** list1[1] **or** list1[2])

**Output:**

True

Hi

* Both OR and any() stops checking the values after the first element-"Hi" as it is already True-Lazy evaluation.

### any() without using any()

It is also important to understand how to implement the functionality of any () without actually using the function:

* We can achieve that using a simple for loop and an if-else.

**Example:**

1. list1 = [0, -4, -3, 2, -1]
2. **print**("The list: ", list1)
3. **for** i **in** list1:
4. **if** i:
5. **print**("True")
6. **break**
7. **if**(i==len(list1)):
8. **print**("False")

**Output:**

The list: [0, -4, -3, 2, -1]

True

* More efficiently, we can use a function.

[**Next →**](https://www.javatpoint.com/any-in-python)[**← Prev**](https://www.javatpoint.com/tkinter-application-to-switch-between-different-page-frames-in-python)

# Append (key: value) Pair to Dictionary

Dictionary is one of the most used data types in Python. It is an unordered collection of key: value pairs. Every value has a corresponding key that identifies it. **A dictionary is a mutable collection, meaning we can modify the values.** One factor that makes a dictionary unique among other data types is that it stores the mapping of key: value pairs while other data types store a single value as an element.

### Example:

1. dictionary = {101: 'Ramya', 102: 'Sanya', 103: 'Sree'}
2. **print**(dictionary)
3. **print**(dictionary[101])
4. **print**(dictionary[102])
5. **print**(dictionary[103])

**Output:**

{101: 'Ramya', 102: 'Sanya', 103: 'Sree'}

Ramya

Sanya

Sree

Given that a dictionary is mutable, we should be able to **alter the values of existing keys and add new key: value pairs to the dictionary**. This tutorial discusses how we can add a new key: value pair, into a dictionary.

### 1. The Traditional way of using the key Subscripts:

We can assign the existing keys with the values we want. Python forgets the old value and updates the new value to the key. Using the same way, we can assign values to new keys, thus appending new pairs.

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

**Example:**

1. dictionary = {101: 'Ramya', 102: 'Sanya', 103: 'Sree'}
2. **print**(dictionary)
3. dictionary[102] = "Priyanka"
4. dictionary[103] = "Ujjwala"
5. dictionary[104] = "Sanya"
6. dictionary[105] = "Sree"
7. **print**(dictionary)

**Output:**

{101: 'Ramya', 102: 'Sanya', 103: 'Sree'}

{101: 'Ramya', 102: 'Priyanka', 103: 'Ujjwala', 104: 'Sanya', 105: 'Sree'}

### 2. update() Method:

It is an inbuilt dictionary method designed to modify a dictionary. **The method takes the {key: value} pair as an argument and adds it to the dictionary.**

* We can also update the values of pre-existing keys using this method.
* The method can **accept any number of arguments**, which means it can append a dictionary with any number of {key: value} pairs at once.

**Example:**

1. dictionary = {101: 'Ramya', 102: 'Sanya', 103: 'Sree'}
2. **print**(dictionary)
3. dictionary.update({102: 'Priyanka'})
4. dictionary.update({103: 'Ujjwala'})
5. dictionary.update({104: 'Sanya'})
6. dictionary.update({105: 'Sree'})
7. **print**(dictionary)

**Output:**

{101: 'Ramya', 102: 'Sanya', 103: 'Sree'}

{101: 'Ramya', 102: 'Priyanka', 103: 'Ujjwala', 104: 'Sanya', 105: 'Sree'}

* In the above example, the dictionary is modified 4 times using the update method. The first two statements updated the values of already existing keys, while the next two added two new pairs to the dictionary.

**What if we want to add thousands of new key: value pairs?**

* We can't keep adding individual pairs one after the other. It takes a lot of time, making the code complex and lengthy.
* The simple way is to create a new dictionary with all the new pairs we want to add and then append/ **merge** it into the dictionary.

**Example:**

1. dictionary = {101: 'Ramya', 102: 'Sanya', 103: 'Sree'}
2. **print**("The original dictionary: ")
3. **print**(dictionary)
4. **print**("New key: value pairs: ")
5. append = {104: 'Jeevani', 105: 'Rishitha', 106: 'Nikitha'}
6. **print**(append)
7. dictionary.update(append)
8. **print**("The updated dictionary: ")
9. **print**(dictionary)

**Output:**

The original dictionary:

{101: 'Ramya', 102: 'Sanya', 103: 'Sree'}

New key: value pairs:

{104: 'Jeevani', 105: 'Rishitha', 106: 'Nikitha'}

The updated dictionary:

{101: 'Ramya', 102: 'Sanya', 103: 'Sree', 104: 'Jeevani', 105: 'Rishitha', 106: 'Nikitha'}

### 3. The OOPS Way:

* We use the OOPS nature of Python to **create a function capable of adding new key: value pairs into the dictionary.**
* The logic is the same traditional way, but the difference is we create an object for the dictionary, and the user can input the key: value pairs to add or modify the dictionary.

**Example:**

1. **class** append(dict):
2. **def** \_\_init\_\_(self, dictionary):
3. self.dictionary = dictionary
4. **def** add(self, key, value):
5. self.dictionary[key] = value
6. dictionary = {101: 'Ramya', 102: 'Sanya', 103: 'Sree'}
7. **print**(dictionary)
8. obj = append(dictionary)
9. obj.key = int(input("Enter the key you want to modify: "))
10. obj.value = input("Enter the value to update: ")
11. obj.add(obj.key, obj.value)
12. **print**(obj.dictionary)

**Output:**

#updating existing key:

{101: 'Ramya', 102: 'Sanya', 103: 'Sree'}

Enter the key you want to modify: 102

Enter the value to update: Priyanka

{101: 'Ramya', 102: 'Priyanka', 103: 'Sree'}

#adding new key:value pair:

{101: 'Ramya', 102: 'Sanya', 103: 'Sree'}

Enter the key you want to modify: 104

Enter the value to update: Sanya

{101: 'Ramya', 102: 'Sanya', 103: 'Sree', 104: 'Sanya'}

* Rather than using an existing dictionary, by **using dict()**, we can create a dictionary only from the inputs of the user.

**Example:**

1. **class** append(dict):
2. **def** \_\_init\_\_(self):
3. self = dict()
4. **def** add(self, key, value):
5. self[key] = value

# Filter List in Python

While using shopping apps, suppose we want to buy a T-shirt; we type it in the search bar, and hundreds of results appear. We'll have some criteria in our mind on what we want to choose like the amount of money we want to spend on the T-shirt, colour etc. Hence, shopping apps provide a '**filter**' option using which we can only view the results that already satisfy our pre-defined criteria.

In the same way, while working on huge amounts of data, we'll sometimes have to filter the data based on some criteria. For example, In the 3 billion population of India, if we want information about only people of a certain age group, we can filter the 3 billion profiles based on age group.

This tutorial discusses how to filter the data stored in a list based on the need in the program with examples.

First, a list is a built-in data structure in Python in which we can store data of different types.

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

1. Students = ["Jeevani", "Ramya", "Sanya"]

### Example 1:

Suppose we have a list with marks of 60 students in a particular subject, and we want to filter out the marks less than 20. So here, the criterion is to eliminate all the values less than 20.

**Code:**

1. marks = [30, 40, 56, 54, 34, 12, 13, 18]
2. res = []
3. **for** i **in** marks:
4. **if**(i > 20):
5. res.append(i)
6. **else**:
7. **continue**
8. **print**(res)

**Output**

[30, 40, 56, 54, 34]

### Example 2:

Suppose we have some characters in a list, and we want to filter all the consonants out of the list.

1. characters = ['a', 't', 'e', 'j', 'i', 'o', 'l']
2. vowels = ['a', 'e', 'i', 'o', 'u']
3. res = []
4. **for** i **in** characters:
5. **if**(i **in** vowels):
6. res.append(i)
7. **else**:
8. **continue**
9. **print**(res)

**Output**

['a', 'e', 'i', 'o']

In both the above examples, we created a new list and appended the elements from the original list that satisfied the criteria we wanted.

### Another Approach:

We can also create a copy of the original list and delete the elements that do not satisfy the criteria.

**Example:**

1. characters = ['a', 't', 'e', 'j', 'i', 'o', 'k']
2. copy = characters
3. vowels = ['a', 'e', 'i', 'o', 'u']
4. **for** i **in** copy:
5. **if**(i **in** vowels):
6. **continue**
7. **else**:
8. copy.remove(i)
9. **print**(copy)

**Output**

['a', 'e', 'i', 'o']

* The above code creates a copy of the original characters list and removes the elements if they don't exist in the vowels list.

### Python's built-in Function:

These are the general approaches, but Python is a language that makes everything simpler. It provides a built-in function called filter() that filters the lists based on the criterion we give.

**Syntax:**

1. filter(criterion, sequence)

**criterion:** The testing standard we provide to filter the list. The function checks if every sequence element returns True when tested with the criterion.

**sequence:** The data structure we provide can be a list, tuple, set or any other iterable data type.

* The function returns the filtered sequence.

**Let us take the two examples we used earlier and see how filter() works:**

1. #Filtering marks:
2. **def** criterion(orig):
3. **if**(orig > 20):
4. **return** True
5. **else**:
6. **return** False
8. marks = [30, 40, 56, 54, 34, 12, 13, 18]
9. **print**(filter(criterion, marks))
10. res = list(filter(criterion, marks))
11. **print**(res)

**Output**

[30, 40, 56, 54, 34]

* In Python 3.x, filter() returns an object. Hence, we need to use the list() to convert the object into a list.
* In the above code, we created a function criterion that checks if the given variable is greater than 20. Now, we can use filter() with the criterion on any data we want with a single line.

**Example:**

1. #Filtering consonants
2. **def** criterion(orig):
3. vowels = ['a', 'e', 'i', 'o', 'u']
4. **if**(orig **in** vowels):
5. **return** True
6. **else**:
7. **return** False
8. characters = ['a', 't', 'e', 'j', 'i', 'o', 'k']
9. res = list(filter(criterion, characters))
10. **print**(res)

**Output**

['a', 'e', 'i', 'o']

### Lambda Function + filter():

Creating a function every time we need the filter() can be tiring, mostly if we won't need it any other time. Hence, we mostly use lambda functions with the function.

**If you don't know about lambda functions, here is a small promo:**

A lambda function is a function without a name-an anonymous function. It can take any number of arguments and permits a single expression. We use the lambda function to create a function needed for a short time without much long-term significance.

**Syntax:**

1. **lambda** arguments : expression

**Example:**

1. x = **lambda** a, b : a + b
2. **print**(x(3, 2))

**Output**

5

**Here are the same two examples we used earlier made easier using the lambda functions:**

1. #Filtering marks
2. marks = [30, 40, 56, 54, 34, 12, 13, 18]
3. res = list(filter(**lambda** i : i>20, marks))
4. **print**(res)

**Output**

[30, 40, 56, 54, 34]

1. #Filtering consonants
2. characters = ['a', 't', 'e', 'j', 'i', 'o', 'k']
3. res = list(filter(**lambda** i: i **in** ['a', 'e', 'i', 'o', 'u'], characters))
4. **print**(res)

**Output**

['a', 'e', 'i', 'o']

### Filtering the List of Lists:

Suppose we are doing an investigation and we have a list of possible criminals. We know that the age of the accused is between 20 and 30. Now, we need to filter the lists:

1. criminals = [["Andy", 34], ["Roberto", 23], ["Joseph", 37], ["Ross", 21], ["Chandler", 23], ["Joey", 27]]
2. filtered = list(filter(**lambda** i : i[1] <= 30 **and** i[1] >= 20, criminals))
3. **print**(filtered)

**Output**

[['Roberto', 23], ['Ross', 21], ['Chandler', 23], ['Joey', 27]]

# \_name\_ \_main\_ in Python

The **main() function** is required in programmes developed in the **C family of languages (C, C++, Java, C#, etc.)** to designate where the execution should begin.

However, because **Python is an interpreter-based language** that can also be used in an interactive shell, there is **no such thing as a main() method** in it. Multiple statements can be found in the **.py extension Python programme file**. The **first statement of the Python programme file** is where the execution begins.

Python interpreter analyses source file and defines some special variables/global variables before running code. The Python interpreter set the special **\_\_name\_\_** variable to have the value **"\_\_main\_\_"** if that module (the source file) is being **executed as the main application. \_\_name\_\_** will be set to the **name of the module** if the same file is being fetched from another module. The **\_\_name\_\_ global variable** accepts the module name as a value.

A file containing **definitions and statements in Python** is known as a module. The module name with the **.py** extension becomes the file name.

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

### Example :

1. # Python program to demonstrate the execution of
2. # main directly
3. **print** ("Executed Always")
5. **if** \_\_name\_\_ == "\_\_main\_\_":
6. **print** ("It is executed when it's invoked directly")
7. **else**:
8. **print** ("It is Executed when it's imported")

**Output:**

Executed Always

It is executed when it's invoked directly

**Explanation :**

This whole block of code **indented at level 0 [Block 1] is run**. Although specified functions and classes are well defined, **none of their code actually executes**.

As we followed the instructions here. The **\_\_main\_\_ variable** will be used directly by Python. As a result, the code in this **if block [Block 2]** will only be executed if that module serves as your **program's entry point**.

So, by checking the **\_\_name\_\_ variable**, you may determine if your script is getting **executed manually or** whether it is being **imported by another programme**.

**If script is imported by another module** at that moment, the module **name will be \_\_name\_\_.**

## Why is it required?

It is usually required when we want to **execute a function directly**. For instance, **we are creating a script** that will be utilised as a module:

**Example :**

1. # Python program to demonstrate the execution of a
2. # function directly
3. **def** this\_function():
4. **print** ("I'm inside this function")
6. # We can test this function by calling it.
7. this\_function()

**Output:**

I'm inside this function

## Advantages of using \_name\_ \_main\_ :

* Every Python module has a specified **\_\_name\_\_** and if this is '**\_\_main\_\_'**, it indicates that the module is being **executed independently by the user,** allowing us to take the necessary action.
* The **\_\_name\_\_ is assigned to the name of the script/module** if you acquire this script as a module in another script.
* Python files can function as **independent applications** or as **reusable modules**.
* If the file was executed directly, without being imported, then the statement **"if \_\_name\_\_ == "main"** is used to run some code.

## Why was this designed this way?

Naturally, we might ponder the **rationale behind the design**. Well, there are instances when we wish to **create a .py file** that can be used as both a **module by other programmes and/or modules** and the **main application itself**.

This behaviour helps us **test and improve our code** more quickly. Since it enables us to **run unit tests directly in script mode**, it also aids in **debugging**.

Additionally, the fact that setting up a single variable is all it takes to **launch a module in Python directly** is elegant.

## Conclusion :

The **\_\_name\_\_** variable has a particularly excellent use case, whether you need a file that can be imported by the other modules or launched as the primary programme. When the modules are imported, we can allow or disallow running specific sections of code by using an **if \_\_name\_\_ == "\_\_main\_\_"** block.

The **\_\_name\_\_** variable is set to the name of the **imported** module or **\_\_main\_\_** if the module is being launched whenever the **Python interpreter reads any file**. All top-level code is **executed when reading the file**, but **functions and classes are not** (as they will only be imported).

By employing the top-level scope **\_\_main\_\_** and the special variable **\_\_name\_\_,** the **reusability is increased**. Both independently and when imported as a module, the Python script file can be run from the **command line or terminal**.

# \_\_name\_\_ in Python

Python's **\_\_name\_\_** special variable stores the **name of the currently running Python script or module**. The Python **\_\_name\_\_** variable was added in **Python 3.0** and is not present in Python 2.x. When a Python script or module is being executed, the **\_\_name\_\_ variable is given the value \_\_main\_\_** for the present Python script or module.

## What does \_name\_ mean ?

Python has a **built-in variable called \_\_name\_\_** that records the name of the currently running module or script. The **\_\_name\_\_** variable merely holds the name of the module or script unless the current module is **executing**, in which case the value **\_\_main\_\_** is set to it.

Therefore, if a **Python script** is imported into another Python script, its **\_\_name\_\_** variable should always have the **value \_\_main\_\_** when that Python script is running. Otherwise, it would have the **name of the module**.

### Example :

To further understand this, let's use an example. Make a **script in Python called testing.py,** and append the following code to it :

1. # Code to define a function
2. **def** anything():
3. **print**('Value of the \_\_name\_\_ : ', \_\_name\_\_)
5. anything()

**Output:**

Value of the \_\_name\_\_ : \_\_main\_\_

**Explanation :**

The value of the **\_\_name\_\_** variable was set to **\_\_main\_\_** when we ran the test.py script.

Let's now build another Python script called **mains.py** and import the previous one into it.

### Example :

1. # importing testing.py
2. **import** testing
4. testing.anything()

**Output:**

Value of the \_\_name\_\_ : testing

**Explanation :**

Because we displayed the **testing,py** module's value of the **\_\_name\_\_ variable** , we can see from the output of the code above that the value of the variable is **testing**.

## Using the condition if name == main :

We use the **if statement** and the condition **\_\_name\_\_ == \_\_main\_\_** to declare that **certain Python code** should only be **performed when the script is run directly**.

### Example :

1. # importing testing.py
2. **import** testing
4. **if** \_\_name\_\_ == \_\_main\_\_:
5. testing.anything()

Here, the string **\_\_main\_\_** is used to determine if the present module or script is executing independently or not. The two underscores on each side of name in the **\_\_name\_\_ variable** were there to let the Python interpreter know that it's a **reserved or special keyword.**

## Code example for name in Python :

As previously said, when we run a code file, the value of the **\_\_name\_\_ variable** changes to **\_\_main\_\_** because the code is **executed directly**, without even being imported into another file.

**Code :** Here is **ScriptP1.py**, a code file.

1. # defining a function
2. **def** anything():
3. **print**('It is a function in the ScriptP1.')
5. **if** \_\_name\_\_=='\_\_main\_\_':
6. anything()
7. **print**('Called from the ScriptP1.')
8. **else**:
9. **print**('ScriptP1 is imported into another file.')

**Output:**

It is a function in the ScriptP1.

Called from the ScriptP1.

Let's now create a new Python script file called **ScriptP2.py**, import **ScriptP1.py** into it, and attempt to invoke the function **anything()** that is defined in **ScriptP1.**

**Code :** The **ScriptP2.py** code is provided here:

1. **import** ScriptP1
3. **if** \_\_name\_\_=='\_\_main\_\_':
4. ScriptP1.anything()
5. **print**('Called from the ScriptP2.')

**Output:**

ScriptP1 is imported into another file.

It is a function in the ScriptP1.

Called from the ScriptP2.

The **\_\_name\_\_ variable** had the value **ScriptP1** (the name of the module) when the import statement for **ScriptP1** was run inside of **ScriptP2**, but as the **ScriptP2** was the first script to be executed, it will now have the value **\_\_main\_\_.**

## Printing the Value of \_\_name\_\_ :

Let's print the value of the \_\_name\_\_ variable at each stage of execution to help you understand it better.

**Example :** The ScriptP1.py Python script's source code is provided below :

1. **print**('Value or the variable \_\_name\_\_ : ' + \_\_name\_\_)

**Output:**

Value or the variable \_\_name\_\_ : \_\_main\_\_

**Example 2 :** And here is the script ScriptP2.py's source code :

1. # importing the file ScriptP1.py
2. **import** ScriptP1
4. **print**('Value or the variable \_\_name\_\_ : ' + \_\_name\_\_)

**Output:**

Value or the variable \_\_name\_\_ : \_\_main\_\_

## Summary :

The primary method or function is typically used as the point at which any program is executed in most programming languages. What about Python, though? A Python program (script) typically begins **execution at the first line**, which is the **program's indentation level 0**. A **\_\_name\_\_** variable is generated **prior to the execution** of a Python program, though. In Python, **this variable could be used in place of the main method.**

# Automate Software Testing with Python

The process of discovering faults in a created product is known as software testing. Additionally, it assesses if the actual outcomes correspond to the outcomes expected and aids in the identification of errors, needs that are lacking, or gaps. The final step before a product is made available on the market is testing. It involves looking at, studying, observing, and judging a variety of product components. Software testers in the industry combine human and automated testing methods. After running the tests, the testers inform the development team of the results. Software testing is essential since the end goal is to give the user a high-quality product.

Many entrepreneurs neglect to test their products. They may claim that their financial constraints are to blame for skipping this crucial phase. They believe it will have little impact. It must, however, be excellent in order to establish a robust and favorable initial impression. And this necessitates thorough bug testing of the product.

### Enhance Product Quality

Only when the product supplied is perfect can a company provide value to its clients. And, in order to accomplish so, businesses must ensure that users have no problems when utilizing their products. Making your goods bug-free is the foolproof approach to achieving it.

Before delivering a product, companies must focus on testing apps and fixing any flaws that are discovered during testing. The quality of the output improves when the team fixes issues before the product reaches the consumer.

Play Videox[[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack" \t "_blank)

### Improve Security

Customers are bound to divulge some type of personal information when they utilize the product. Before the program is deployed, security testing is required to prevent hackers from gaining access to this information. When a company follows a thorough testing procedure, it assures a safe product, which helps customers feel comfortable while using it.  
Payment information is required by banking apps and e-commerce stores, for example. It can result in enormous financial loss if the developers do not repair security-related flaws.  
The other aspect of security is preventing data loss. People nowadays frequently save data on cloud storage. Your photographs and files are presumably also stored on iCloud or Google Drive.  
What if anything goes wrong and all your data is lost? Isn't this one of your worst nightmares? A product's security ensures that data is not only protected against hackers but also that it is not lost or damaged.

Customers no longer work exclusively on large desktop computers. Testing a product's device compatibility is essential in today's mobile-first world. Consider the case where your company created a website. The tester must see if the webpage works on various device resolutions. It should also be compatible with a variety of browsers. The growing number of browser alternatives is another reason why testing is becoming increasingly important. What works with Chrome may not be compatible with Safari or Internet Explorer. This necessitates cross-browser testing, which involves validating the application's interoperability across many browsers.

### Classifications of Software Testing:

Software testing is a broad term that encompasses a variety of activities. Instead, it occurs in a variety of forms that may be classified based on a number of factors. You can divide testing into manual and automated categories, for example. When it comes to automated testing, you may choose between code-based and code-free ways, as well as hybrid approaches that combine the best of both worlds. Tests may also be classified based on how much information they have about the system under the test's internal implementation. We may divide tests into three categories based on this criterion: white-box, black-box, and grey-box. Finally, we may divide tests into functional and non-functional categories based on whether they confirm the application's business requirements.

**Black-box Testing:**

You have the least amount of knowledge about how the product is developed in this sort of testing. You have no knowledge of the product's structure, coding, or logic. You would utilize the product as if you were a customer. Black-box testing is used for functional testing since you have the same amount of knowledge as your consumer. Only after the code is run can this form of testing take place. As a result, dynamic testing is employed. Dynamic testing is a sort of testing in which you must run the code and test the product while it is running. It's largely done to see how it'll work once it's up and running, as well as how the user will react to it.

**Grey-box Testing:**

In this sort of testing, you only have a limited understanding of the product. This form of testing is useful for detecting issues that the user would be unaware of.

To give you an example, suppose you created an element with a blue shade but it really has a green shade. The user would be unaware that there is a flaw since they would believe that is as it should be. However, your limited understanding of the product will aid you in detecting such flaws.

### Gorilla Testing:

Gorilla testing is a method of software testing in which a module is repeatedly tested using a variety of random inputs to ensure that it is error-free. Only a few selected modules of the system are submitted to testing with the purpose of evaluating whether or not the module is operating properly. This form of testing is done manually and frequently. Torture testing, fault tolerance testing, and unpleasant testing are all terms used to describe Gorilla testing.

Now that you know what testing is all about, it's time to learn how to carry out software testing.

* **Planning:** An operation always starts with planning. You compile all the relevant product details in this phase. You create a list of tasks that need to be tested first. If you are testing after a bug fix, you should be aware of what the problem was and what the ideal behavior is. The issue of prioritizing your to-do list is another. The division of tasks during this stage can also be done when the entire team is involved.
* **Preparation:** After you've figured out what you need to accomplish, you'll need to lay the groundwork for testing. Preparing the test environment, gathering test cases, and studying product features and test cases are all part of this process. Gathering and familiarising yourself with testing tools and methodologies should also be done here.
* **Execution:** This is when you truly put the thing through its paces. You run test scenarios and record the outcomes. The findings are then compared to the intended outcome to evaluate if the product is performing as expected. You keep track of all successful and unsuccessful tests and test cases.
* **Reporting:** This is the final stage of software testing when you must document all of your results and send them to the appropriate individuals. The failures of test cases are of particular importance in this scenario. There should be a proper and clear description of the tests that were done and the results that were obtained. Steps to replicate the mistake, screenshots, and any other relevant information should be included in more sophisticated tests.

### Benefits of Software testing:

* **Cost-Effective:** It is one of the most significant benefits of software testing. Any IT project that is tested on time enables you to save money in the long run. It is less expensive to correct issues discovered early in the software testing process.
* **Security:** It is software testing's most fragile and sensitive advantage. People are seeking items that they can trust. It aids in the early detection of hazards and issues. Another important reason why software testing should be avoided is because of this. It is said to be the most fragile and sensitive area of the body. There are a variety of instances in which users' information and details are taken and utilized for personal gain. It is said to be the reason why customers seek things that have been thoroughly examined and proven to be dependable. As a product undergoes testing, the user may be confident that they will obtain a trustworthy product. The user's personal information is protected. With the help of software testing, users may acquire goods that are free of vulnerabilities.
* **Product quality:** It is a prerequisite for any software product. Testing guarantees that buyers receive a high-quality product.
* **Customer Satisfaction:** Any product's primary goal is to provide client happiness. The optimal user experience is ensured via UI/UX testing. The fundamental goal of the product owner is to provide the highest level of customer satisfaction. The importance of software testing may be attributed to the fact that it provides the required and ideal user experience. You will be able to establish the reputation of dependable clients if you choose the ideal project in a crowded market. As a result, choosing software testing will provide you with long-term benefits. Earning a client's trust is no easy feat, especially if the product is proven to be malfunctioning at all times. You've probably tried a lot of items and had a number of bad experiences as a result of which you may have removed the app. In today's market, the market is extremely saturated. The initial impression is really crucial, and if you fail to make one, people will look for another product that meets all of their needs.

# \_\_dict\_\_ in Python

In Python, a dictionary is an **unordered set of data values** that could be **used to store data values** similarly to a map. Unlike other data types, which can only include a single value per element, dictionaries can also contain a **key:value** pair. To make the dictionary **more efficient**, Key-Value is offered.

When it pertains to attribute access, the dot **"."** (as in **x.any\_attribute**) is all that the majority of users are familiar with. Simply said, **attribute access is the process of obtaining an object that is connected to one you already have**. It may look very simple to somebody who uses Python without going too far into the specifics. However, there is a lot going on behind the scenes for this relatively simple process.

## What is \_dict\_ ?

Every module has a unique property called **\_\_dict\_\_.** This dictionary **contains the symbol table for the module**. The (writable) characteristics of an item are stored in a dictionary or the other mapping object.

To put it simply, every object in Python has a **property that is indicated by the symbol \_\_dict\_\_**. Furthermore, this object has every property that has been specified for it. Another name for \_\_dict\_\_ is **mappingproxy** object. We can use the dictionary via **applying the \_\_dict\_\_ property** to a class object.

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

### Syntax :

1. object.\_\_dict\_\_

### Example :

1. **class** AnimalClass:
2. **def** \_\_init\_\_(self,identity,age):
3. self.identity = identity
4. self.age = age
6. **def** feature(self):
7. **if** self.age == "10":
8. **return** True
9. **else**:
10. **return** False
11. ac = AnimalClass('Lion','10')
12. **print**(ac.\_\_dict\_\_)

**Output:**

{'identity': 'Lion', 'age': '10'}

### Example 2 :

This example will demonstrate that by the means of the **\_\_dict\_\_ attribute**, one could create a dictionary out of any object :

1. # class Flowers is defined
2. **class** Flowers:
4. # constructor
5. **def** \_\_init\_\_(self):
7. # keys are being initialized with
8. # their corresponding values
9. self.Rose = 'red'
10. self.Lily = 'white'
11. self.Lotus = 'pink'
13. **def** displayit(self):
14. **print**("The Dictionary from object fields belongs to the class Flowers :")

17. # object animal of class Animals
18. flower = Flowers()
20. # calling displayit function
21. flower.displayit()
22. # calling the attribute \_\_dict\_\_ on flower
23. # object and making it print it
24. **print**(flower.\_\_dict\_\_)

**Output:**

The Dictionary from object fields belongs to the class Flowers :

{'Rose': 'red', 'Lily': 'white', 'Lotus': 'pink'}

### Example 3 :

1. **def** funct():
3. **pass**
5. funct.practice = 1
7. **print**(funct.\_\_dict\_\_)
9. **class** PracticeClass:
11. x = 1
13. **def** practice\_function(self):
15. **pass**
17. **print**(PracticeClass.\_\_dict\_\_)

**Output:**

{'practice': 1}

{'\_\_module\_\_': '\_\_main\_\_', 'x': 1, 'practice\_function': , '\_\_dict\_\_': , '\_\_weakref\_\_': , '\_\_doc\_\_': None}

## Using Dictionary without the use of \_\_dict\_\_ in Python :

### Creating a Dictionary :

In Python, a dictionary may be made by enclosing a **list of entries inside curly brackets and separating them with a comma**. Dictionary stores pairs of values, where one pair element is the Key and another one is its **Key:value**. In contrast to **keys,** which **cannot be repeated and must be immutable, values** in dictionaries **can be of any type of data and can be duplicated**.

The elements are **separated by commas**, each key is distinguished from its value **by a colon (:),** and the entire structure is contained in **curly brackets**. A dictionary that is completely devoid of all words is written as follows: **{}**.

The keys of the dictionary must be **immutable**, such as **integers, tuples, or strings**, although the values can be of any type. In Python dictionaries, the same key name spelled differently is considered as **a distinct key.** Please take note that **dictionary keys are case-sensitive**; keys with the similar name but distinct case will be handled differently.

**Example :**

1. # Creating a Dictionary
2. # using Integer Keys only
3. Dict = {1: 'JAVA', 2: 'T', 3: 'POINT'}
4. **print**("\nCreating a Dictionary by using Integer Keys : ")
5. **print**(Dict)
7. # Creating a Dictionary
8. # using various Mixed keys
9. Dict = {'Company': 'JavaTpoint', 7: [22, 35, 46, 97]}
10. **print**("\nCreating a Dictionary by using Mixed Keys : ")
11. **print**(Dict)

**Output:**

Creating a Dictionary by using Integer Keys :

{1: 'JAVA', 2: 'T', 3: 'POINT'}

Creating a Dictionary by using Mixed Keys :

{'Company': 'JavaTpoint', 7: [22, 35, 46, 97]}

The **built-in method dict()** also allows for the **creation of dictionaries**. Simply putting **two curly braces {}** together will result in an **empty dictionary**.

### Example :

1. # Creating an empty Dictionary
2. myDict = {}
3. **print**("This is an Empty Dictionary: ")
4. **print**(myDict)
6. # Creating a Dictionary
7. # using the dict() method
8. myDict = dict({1: 'JAVA', 2: 'T', 3: 'POINT'})
9. **print**("\nCreating a Dictionary by using the dict() method : ")
10. **print**(myDict)
12. # Creating a Dictionary
13. # using each item as a different Pair
14. myDict = dict([(1, 'JavaTpoint'), (2, 'Great')])
15. **print**("\nCreating a Dictionary by using each item as a different pair : ")
16. **print**(myDict)

**Output:**

This is an Empty Dictionary:

{}

Creating a Dictionary by using the dict() method :

{1: 'JAVA', 2: 'T', 3: 'POINT'}

Creating a Dictionary by using each item as a different pair :

{1: 'JavaTpoint', 2: 'Great'}

### Complexities for creating a Dictionary :

* Time complexity : **O(length(dict))**
* Space Complexity : **O(n)**

### Nested dictionaries :

It is a form of dictionary where one or more than one **keys has a dictionary attached to it as a value** of the key.

**Example :**

1. # Creating a Nested Dictionary
2. # as mentioned above using a dictionary as a value to a key in
3. # a dictionary
4. myDict = dict({1: 'JAVA', 2: 'T', 3: 'POINT', 4: {1: 'JavaTpoint', 2: 'Great'}})
5. **print**("\nCreating a Nested Dictionary : ")
6. **print**(myDict)

**Output:**

Creating a Nested Dictionary : {1: 'JAVA', 2: 'T', 3: 'POINT', 4: {1: 'JavaTpoint', 2: 'Great'}}

### Addition of elements to a dictionary :

There are **several methods for adding elements** to a Python dictionary. By specifying the value and the key together, for example, **Dict[Key] = "Value",** one value may be added to a dictionary at a time. Using the **built-in update() function**, one can modify an existing value in a Dictionary. An existing Dictionary may also be expanded with **nested key values**.

#### Note : When adding a value, the value is updated if the key-value combination already exists. If not, a new key and value are added to the dictionary.

**Example :**

1. # Creating Empty Dictionary
2. myDict = {}
3. **print**("Empty Dictionary: ")
4. **print**(myDict)
6. # Adding elements only one at a time
7. myDict[0] = 'Java'
8. myDict[3] = 'T'
9. myDict[6] = 41
10. **print**("\nDictionary after the addition of 3 elements: ")
11. **print**(myDict)
13. # Adding a set of values
14. # to a particular Key
15. myDict['settingValues'] = 7, 8, 9
16. **print**("\nDictionary after the adding a set of values to a key : ")
17. **print**(myDict)
19. # Updating the existing Key's Value
20. myDict[3] = 'tPoint'
21. **print**("\nDictionary after Updated key value: ")
22. **print**(myDict)
24. # Adding Nested Key value to Dictionary
25. myDict[8] = {'Nested' :{'A' : 'boy', 'B' : 'Girl'}}
26. **print**("\nDictionary after Addition of a Nested Key: ")
27. **print**(myDict)

**Output:**

Empty Dictionary:

{}

Dictionary after the addition of 3 elements:

{0: 'Java', 3: 'T', 6: 41}

Dictionary after the adding a set of values to a key :

{0: 'Java', 3: 'T', 6: 41, 'settingValues': (7, 8, 9)}

Dictionary after Updated key value:

{0: 'Java', 3: 'tPoint', 6: 41, 'settingValues': (7, 8, 9)}

Dictionary after Addition of a Nested Key:

{0: 'Java', 3: 'tPoint', 6: 41, 'settingValues': (7, 8, 9), 8: {'Nested': {'A': 'boy', 'B': 'Girl'}}}

### Complexities for adding elements to a Dictionary :

* Time Complexity : **O(1)/O(n)**
* Space Complexity : **O(1)**

### Accessing Dictionary Elements :

A dictionary employs **keys**, whereas other data types require indexing to retrieve values. Keys can be utilised with the **get() function** or **inside square brackets [].**

In the event that a **key cannot be found** in the dictionary, **KeyError is produced** if we use **square brackets [].** On the other hand, if the key cannot be located, the **get() function returns None.**

**Example :**

1. # Python program to demonstrate the
2. # accessing of an element, from a Dictionary
4. # Creating a Dictionary
5. myDict = {1: 'Java', 'name': 'T', 2: 'Point', 4: 'Website'}
7. # accessing an element using key
8. **print**("Accessing an element using the key:")
9. **print**(myDict['name'])
11. **print**("Accessing another element using the key:")
12. **print**(myDict[4])
14. # accessing an element using the get() method
15. **print**("Accessing an using the get() method:")
16. **print**(myDict.get(2))
18. **print**("Accessing another using the get() method:")
19. **print**(myDict.get(1))

**Output:**

Accessing an element using the key:

T

Accessing another element using the key:

Website

Accessing an using the get() method:

Point

Accessing another using the get() method:

Java

### Complexities for accessing elements in a Dictionary :

* Time Complexity : **O(1)**
* Space Complexity : **O(1)**

### Accessing a nested dictionary's element :

We can take the help of the **indexing [] technique** to get the value of an existing key in the **nested dictionary**.

**Example :**

1. # Creating a Dictionary
2. myDict = {'myDict1': {3: 'JavatPoint'},
3. 'myDict2': {'Info.': 'Website'}}
5. # Accessing the elements using the key
6. **print**(myDict['myDict1'])
7. **print**(myDict['myDict1'][3])
8. **print**(myDict['myDict2']['Info.'])

**Output:**

{3: 'JavatPoint'}

JavatPoint

Website

## Built-in Dictionary methods :

**clear() :**

The function dict.clear() **eliminates every key-value pair** from the dictionary.

**copy() :**

A **shallower copy of the dictionary** is returned by the dict.copy() method.

**fromkeys() :**

**Using the supplied iterable** (string, list, set, or tuple) as keys and the specified value, the function dict.fromkeys() **creates a new dictionary**.

**get() :**

This **gives the value associated** with the given key.

**items() :**

A **dictionary view object**, that offers a dynamic presentation of the dictionary items as a list of key-value pairs **is returned** by the function dict.items(). When the dictionary is updated, this **view object is also updated**.

**dict.keys() :**

The function dict.keys() **returns a dictionary view object** with the dictionary's **list of keys.**

**pop() :**

This **returns the value of the key after removing it**. If a key is missing from the dictionary, it either throws a **KeyError** or returns the **default value** if one was provided.

**popitem() :**

This **removes one item from the dictionary and returns a tuple** of (key, value) pairs. The **Last In First Out (LIFO)** sequence is used for returning pairs.

**setdefault() :**

This **returns the dictionary's value for the given key**. If the key cannot be discovered, the key with the supplied **defaultvalue** is added. It sets **None** as the **defaultvalue** if it is not supplied.

**values() :**

The **dictionary view object that offers a dynamic view** of each value which is **present in the dictionary, is returned** by the function dict.values(). When the dictionary is updated, this **view object is also updated**.

**update() :**

A **dictionary or any iterable with key-value pairs**, such as a tuple, **can be updated** using the dict.update() function.

### Example :

1. # Example to demonstrate all dictionary methods
2. #Creating a Dictionary
3. mydict1={1:"HTML",2:"CSS",3:"Javascript",4:"Python"}
5. #copy method
6. mydict2=mydict1.copy()
7. **print**(mydict2)
9. #clear method
10. mydict1.clear()
11. **print**(mydict1)
13. #get method
14. **print**(mydict2.get(1))
16. #items method
17. **print**(mydict2.items())
19. #keys method
20. **print**(mydict2.keys())
22. #pop method
23. mydict2.pop(4)
24. **print**(mydict2)
26. #popitem method
27. mydict2.popitem()
28. **print**(mydict2)
30. #update method
31. mydict2.update({2:"C++"})
32. **print**(mydict2)
34. #values method
35. **print**(mydict2.values())

**Output:**

{1: 'HTML', 2: 'CSS', 3: 'Javascript', 4: 'Python'}

{}

HTML

dict\_items([(1, 'HTML'), (2, 'CSS'), (3, 'Javascript'), (4, 'Python')])

dict\_keys([1, 2, 3, 4])

{1: 'HTML', 2: 'CSS', 3: 'Javascript'}

{1: 'HTML', 2: 'CSS'}

{1: 'HTML', 2: 'C++'}

dict\_values(['HTML', 'C++'])

## Differenciating between a dictionary and a list :

Data structures such as a **list and a dictionary** are fundamentally dissimilar. An **ordered series of items** can be stored in a **list** so that we can index into it or iterate over it. Lists can also be changed even after they have already been generated since they are a changeable type. The **Python dictionary** is a **key-value storage** and an implementation of a **hash table**. It does not follow any particular sequence and requires hashable keys. Additionally, it is quick for key lookups.

A **list's** elements contain the following features :

* Unless specifically reordered, they **keep their current order** (for instance, by sorting the list).
* They might be of whatever type, or even a **combination of sorts**.
* **Through numerical (zero based) indexes**, we may access them.

The characteristics of **dictionary** elements are as follows :

* Each entry has a **value and a key**.
* Orders are **not warranted**.
* **Key values** are used to access elements.
* **Any hashtable type (other than a dict)** may be used for key values, and types may be combined.
* **Any kind of value**, including other dicts, **is allowed**, and types can be combined.

### Usage :

We use a **dictionary** if we have a **set of distinct keys that correspond to values**, but a **list** when we have an **ordered group of things**.

## Conclusion :

* In a computer language, **dictionaries are a sort of data structure used to hold information** that is somehow related.
* Every module has a unique property called **\_\_dict\_\_.**
* \_\_dict\_\_ **contains the symbol table for the module**.
* The properties of an element are stored in a **mapping object.**
* Every object in Python has a **property that is indicated by the symbol \_\_dict\_\_**.
* Another name for \_\_dict\_\_ is also known as **mappingproxy object.**
* The two components of a Python dictionary are called **Keys and Values**.
* You **might not receive** your data back in the similar order that you input it since dictionaries do not keep their data in any specific order.
* Keys will consist of just **one thing.**
* Values can be **integers, lists, lists inside lists**, etc.
* There can be no more than **one entry per key** ( no duplicate key is allowed)
* The keys of the dictionary must be **immutable**, such as **integers, tuples, or strings**, although the values can be of any type.
* Dictionary **keys are case-sensitive**; in Python dictionaries, the same key name spelled differently is considered as a **distinct key**.

\_\_init\_\_ in python

If you have been working with the Object-oriented programming, you might have come across \_init\_ word quite a few times*.****\_\_init\_\_****is a Python method. It is similar to the constructors in languages like Java and C++.* Knowing classes and objects in Python will make the \_\_init\_\_ method understandable.

**Here is some required pre-requisite knowledge:**

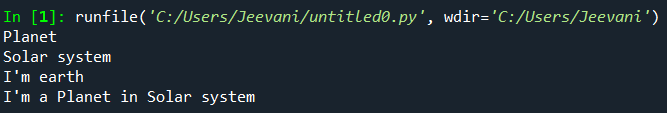
* A **class** is like a blueprint with variables/ attributes and functions/ methods declared. To use the class, we need to create objects for the created class.
* Using the objects, we can call the methods in the class and access the declared attributes.
* Every object can have its values for the attributes in the class. We can pass the values we want as arguments when creating the object.

Here is a simple example of a class and an object:

1. **class** planet:
2. var1 = "Planet"
3. var2 = "Solar system"
4. **def** function (self):
5. **print** ("I'm earth")
6. **print** ("I'm a", self. var1, "in", self. var2)
8. earth = planet ()
9. **print** (earth. var1)
10. **print** (earth. var2)
11. earth. function ()

**Output:**

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



**Analysis:**

We created a class named planet. In the class:

1. We declared two variables, var1 and var2.
2. We created a function where we printed two strings with the declared variables inside the class.

Now, we created object **earth** and accessed the two variables and the method from the class **without passing any arguments.**

* **The object we created doesn't have its variables.**

***Now, what is self in the class?***

When we create an object for the class and call the function, the self is replaced with the created object. It is like a placeholder for the object. In the class we created, we have two variables common to all the objects we create. Hence, even if we called the variables with the object name, we will get the same values for all the objects.

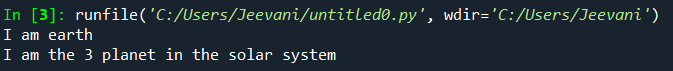
**Now, let us understand what '\_\_init\_\_' can do?**

1. We discussed above that every object could have its values for the attributes of a class. This functionality can be achieved using the \_\_init\_\_ method.
2. It is a constructor, allowing a class to hold objects with different values.
3. We need not call it like we call a normal method. It is similar to a method inside a class. It is executed as soon as an object is created for the class.

Now, let us see the above example with \_\_init\_\_ method:

1. **class** planet:
2. **def** \_\_init\_\_ (self, name, number):
3. self. name = name
4. self. number = number
5. **def** function (self):
6. **print** ("I am", self. name)
7. **print** ("I am the", self. number, "planet in the solar system")
9. earth = planet ('earth', 3)
10. earth. function ()

**Output:**



**Understanding:**

1. We passed arguments to the object when we created it.
2. As we discussed earlier, when an object is created:
3. \_\_init\_\_ method is executed.
4. 'self' is replaced with the created object.

When we created the object 'earth':

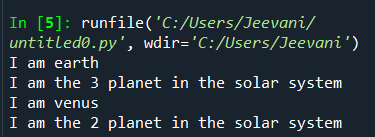
1. **def** \_\_init\_\_ (earth, 'earth', 3):
2. earth. name = 'earth'
3. earth. number = 3

This is the inner mechanism in the \_\_init\_\_ method.

This way, the object earth () can have its attributes.

Now, if we create another object:

1. venus = planet ('venus', 2)
2. venus. function ()



Hence, we can create any number of objects, and every object can have its values for the attributes.

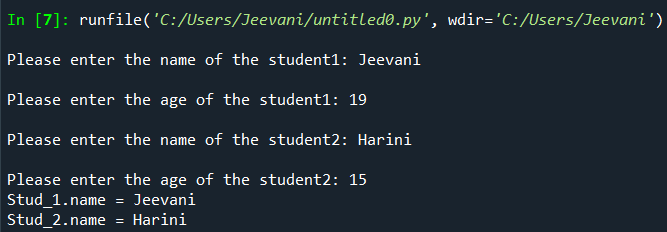
This is the functionality of the \_\_init\_\_ method in object-oriented programming in Python.

**Let us see one more example:**

* We can take the values as inputs from the user and then pass them as attributes to an object.

1. **class** Student:
2. **def** \_\_init\_\_ (self, name, age, email):
3. self. name = name
4. self. age = age
5. self. email = email
6. name = input ("Please enter the name of the student1: ")
7. age = int (input ("Please enter the age of the student1: "))
8. stud = Student (name, age, 'Jeevani@gmail.com')
9. name = input ("Please enter the name of the student2: ")
10. age = int (input ("Please enter the age of the student2: "))
11. stud2 = Student (name, age, 'Harini@gmail.com')
12. **print** ("Stud\_1. name =", stud. name)
13. **print** ("Stud\_2. name =", stud2. name)

**Output:**



**Understanding:**

In the program, we created a class called "Student" with three attributes: name, age and email. Using the self variable, we defined the attributes in the \_\_init\_\_ method. We created two objects, stud and stud2. For both the objects, we already gave the value for email and asked the user for inputs for name and age attributes and then passed the values to the objects.

**Note:**

* We can create any number of objects and any number of attributes and functions inside a class. But, **there can only be one explicit \_\_init\_\_ method inside a class.**
* Even if we write multiple \_\_init\_\_ methods, the latest one will overwrite the previous \_\_init\_\_ methods.

Python's Module Configparser

Python's in-built ConfigParser library is part of the base module. The library provides a console parser for easy file configuration consisting of pairs of name-key values.

The popular worldwide accepted convention supported by this library is the "INI" syntax, utilized on the Microsoft platform most commonly. The library was initially made to help with a different language structure in which the items in each segment looked like RFC 822 headers.

The execution permits named values to be determined utilizing either grammar. This verifiable mishap and absence of an unmistakable determination of the "INI" design have made this module extremely monotonous to keep up with, so the execution has been changed in pretty much every Python discharge, and the way of behaving of the module has demonstrated exceptionally dreary to portray.

**Important points**

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

* The ongoing way of behaving of the defaults has forever been planned.
* The module was intended to help a set-up of projects that could give some data either utilized straightforwardly or to frame portions of different qualities (for example, way names that incorporated the host name of the PC running a program).
* At the time, there was a compelling reason to recognize various purposes of "default" values, as MartinManey proposes beneath. (Not satisfactory to me presently, there's a genuine need to do as such since the API given by ConfigParser is so extra.)
* The utilization of an "enchantment segment name" for the current "defaults" in the ongoing execution is something I think about sad; the default word reference ought to be different from the word reference of areas.
* Ought to extra name gap ("sorts of defaults") at any point be thought of, I'd trust they don't turn out to be essential for arranging segment names by idiotic mishap.
* Incorporating the defaults by the compose() technique is sad; these were expected to be a way for an application to supply processed values that would be utilized on the off chance that a particular set-up didn't supersede them.
* From Python's standard library, the configparser module characterizes usefulness for perusing and composing set-up documents as utilized by Windows OS Microsoft.
* Such documents typically have .INI expansion.
* The INI document comprises segments driven by a [section] header. Between square sections, we can put the part's name.
* The segment is trailed by key/esteem passages isolated by = or: character. It might incorporate remarks prefixed by # or; image.

**An example INI record is displayed underneath:**

1. [Settings]
2. # Setting log detailed for extra debugging information
3. DetailedLog=1
4. RunStatus=1
5. StatusPort=6090
6. StatusRefresh=10
7. Archive=1
8. # Setting the location of the log file MV\_FTP
9. LogFile=/opt/ecs/mvuser/MV\_IPTel/log/MV\_IPTel.log
10. Version=0.9 Build 4
11. ServerName=Unknown
12. [FTP]
13. # Setting the server active FTP
14. RunFTP=1
15. # defining the control port FTP
16. FTPPort=21
17. # Setting the locations of the data directory FTP
18. FTPDir=/opt/ecs/mvuser/MV\_IPTel/data/FTPdata
19. # Setting the Name of admin
20. UserName=admin
21. # Setting the protected Password
22. Password=admin

The ConfigParser class belongs to the configparser module. It must manage the parsed content database and parse all configuration documents.

Any Object of class ConfigParser is created by the below line -

1. Parser\_name = configparser.ConfigParser()

**Following are the methods used with the objects of class ConfigParser:**

|  |  |
| --- | --- |
| write() method | To know the state of configuration in .ini format. |
| read\_dict() method | Reading configuration from a dictionary. Key concepts are important for the segment names, and values are word references with keys and values that ought to be available in the segment. |
| read\_string() method | Reading configuration from any string. |
| remove\_option() method | Remove any option from any part of the section. |
| has\_option() method | Returning whether any option exists in any part of the section. |
| has\_section() method | Returning whether any part of the section exists. |
| read() method | Reading and parsing any configuration document. |
| items() method | Returning a tuple with ( value, name) for each option in the part of the section. |
| getboolean() method | Like the get() method, but converts the value to a Boolean value. Returning True or False value |
| options() method | Returning list of configurations option for the name part of the section. |
| read\_file() method | Reading and parsing one configuration document, any as a document object. |
| getint() method | Like get(), but convert values to integer. |
| get() method | Returning string values for the option named |
| getfloat() method | Like get(), but convert values to float. |
| sections() method | Returning all the configuration parts of the section names. |
| remove\_section() method | Removing any document part of the section and all its options. |
| set() method | Setting any options. |

**Following script parses and reads the 'sampleconfigfile.ini' file**

1. **import** configparser\_name
2. parser\_name = configparser\_name.ConfigParser\_name()
3. parser\_name.read('sampleconfig.ini')
4. **for** sect **in** parser\_name.section():
5. **print**('Sections:', sect\_)
6. **for** k1,v1 **in** parser\_name.items(sect\_):
7. **print**(' {} = {}'.format(k1,v1))
8. **print**()

**Output:**

Sections: Settings

detailedlog = 1

runstatus = 1

statusport = 6099

statusrefresh = 10

archive = 1

logfile = /opt/ecs/mvuser/MV\_IPTel/log/MV\_IPTel.log

version = 0.9 Build 4

servername = Unknown

Section: FTP

runftp = 1

ftpport = 21

ftpdir = /opt/ecs/mvuser/MV\_IPTel/data/FTPdata

username = admin

password = admin

To build a configuration file, write() method is used. Below code configures the parser\_name's object and performs write operation to a file object named 'testing.ini'

1. **import** configparser\_name\_name
2. parser\_name\_name = configparser\_name\_name.ConfigParser\_name\_name()
3. parser\_name\_name.add\_section('Client')
4. parser\_name\_name.set('Client', 'Name', 'Asho Kulkarn')
5. parser\_name\_name.set('Client', 'email', 'asho@gmail.com')
6. parser\_name\_name.set('Client', 'password', 'secret')
7. fp=open('test.ini','w')
8. parser\_name\_name.write(fp)
9. fp.close()

**Supported Datatypes**

In configuration files, Config parser\_names can't assume the datatypes of value, always preserving them as strings internally. This implies that if we need other datatypes, we must convert implicitly:

1. > > > int(top\_secrets['Port'])
2. 50225
3. > > > float(top\_secrets['CompressionLevels'])
4. 8.0

Since this job is so normal, config parser\_names give a scope of helpful getter techniques to deal with numbers, floats, and booleans. The last one is the most fascinating because passing the worth to bool() would do no decent since bool('true') is still false. This is the reason config parser\_names additionally give getboolean(). This technique is case-harsh and perceives Boolean qualities from 'no'/'yes', 'false'/'true', 'off'/'on' and '0'/'1'.

**For example:**

1. > > >
2. > > > top\_secret.getboolean('ForwardX11')
3. False
4. > > > config['bitbucket.org'].getboolean('ForwardX11')
5. True
6. > > > config.getboolean('bitbucket.org', 'Compression')
7. True

Not only method getboolean(), config parser\_names also provide equivalent methods like getfloat()and getint().

Fallback Values

In a dictionary, to provide fallback\_ values, we will use a section's function get():

1. > > >
2. > > > top\_secret.get('Port')
3. '500222'
4. > > > top\_secret.get('CompressionLevel')
5. '9'
6. > > > top\_secret.get('Cipher')
7. > > > top\_secret.get('Cipher', '3degs-cbc')
8. '3degs-cbc'

If it's not too much trouble, note that default values have priority over fallback\_ values. For example, in our model, the 'CompressionLevel' key was determined exclusively in the 'DEFAULT' area. On the off chance that we attempt to get it from the segment 'top\_secret.server.com', we will constantly get the default, regardless of whether we determine a fallback\_:

1. > > > top\_secret.get('CompressionLevel', '3')
2. '9'

Another thing to know about is that the parser\_name-level get() technique gives a custom, more mind-boggling connection point, kept up with for in reverse similarity. While utilizing this technique, a fallback\_ worth can be given utilizing the fallback\_ catchphrase just contention:

1. > > >
2. > > > config.get('bitbucket.org', 'monster\_',
3. ...            fallback\_='No such thing as monster\_s')
4. 'No such thing as monster\_s'

The same fallback\_ argument can be used with the getint(), getfloat() and getboolean() methods, for example:

1. > > >
2. > > > 'BatchMode\_\_' **in** top\_secret
3. False
4. > > > top\_secret.getboolean('BatchMode\_\_', fallback\_=True)
5. True
6. > > > config['DEFAULT']['BatchMode\_\_'] = 'no'
7. > > > top\_secret.getboolean('BatchMode\_\_', fallback\_=True)
8. False

INI File Supported format

Each configuration document has sections, each defined by a header-> section, followed by entries key/value type, which is then partitioned by a string (: or = ). Trailing and leading whitespace is separated from values and keys.

Values can be erased on the off chance that the parser\_name is designed to permit it one, in which case the worth/key full stop may likewise be forgotten about. Values can likewise traverse different lines, for however long they are indented further than the main line of the worth. Contingent upon the parser\_name's mode, clear lines might be treated as parts of multiline esteems or overlooked.

Any section name could be any string that may not has ']' or '\n', by default.

Design documents could contain remarks prefixed by a specific person (; and # ). Remarks might show up on their own on a generally vacant line, potentially indented.

**For example**

1. [Simple Values]
2. key=value
3. gap **in** keys=allowed
4. the gap **in** values=allowed as well
5. the gap around the fullstop= obviously
6. you can also use: to delimit keys **from** values
8. [All Values Are Strings]
9. values like this: 1000000
10. **or** this: 3.14159265359
11. are they treated as a number? : no
12. integer, float **and** boolean are held as strings
13. can use the API to get converted values directly: true
15. [Multiline Values]
16. Servers are PC programming **or** equipment that cycles demand **and** convey information to a client over an organization. Different kinds of servers exist, the most well-known ones being web servers, information base servers, application servers, **and** exchange servers.
17. [No Values]
18. key\_without\_values
19. empty value of string here =
21. [You can use comments]
22. # like this
23. ; **or** this
25. # By default, only in an empty line.
26. # Inline comments can be harmful because they prevent users
27. # from using the delimiting characters as parts of values.
28. # That being said, this can be customized.
30. [Sections Can Be Indented]
31. can\_values\_be\_as\_well = True
32. does\_that\_mean\_anything\_special = False
33. purposes = format **for** purpose of readability
34. multilines\_values = are
35. handle just good as
36. huge as they indent
37. deep the lines are prescribed
38. of a value

Exceptions

1. **Exception name: DuplicateSectionError**  
   Description: if the add\_section() function is called with the name of a section Exception raised that is already present or in strict parser\_names when sections are found once in a single input file, string, or dictionary.  
   New in version 3.1: Optional lineno, source attributes, and arguments to \_\_init\_\_() were added.
2. **Exception name: NoSectionError**  
   Description: Exception is raised when a specified section is not found.
3. **Exception name: NoOptionError**  
   Description: Exception is raised when a specified option is not found in the specified section.
4. **Exception name: DuplicateOptionError**  
   Description: error raised by severe parser\_names on the off chance that a solitary choice shows up twice during perusing from a single document, string, or word reference. This gets incorrect spellings and case awareness-related mistakes; for example, a word reference might have two keys addressing a similar case-inhumane set-up key.
5. **Exception name: InterpolationDepthError**  
   Description: when any length of string interpolations cannot be finished due to the number of iterations exceeding MAX\_INTERPOLATION\_DEPTHs, an Exception is raised. The subclass of the error InterpolationError is called.
6. **Exception name: InterpolationError**  
   Description: The base class is raised when errors happen when performing string interpolations.
7. **Exception name: InterpolationMissingOptionError**  
   Description: when an option reference from a value can't exist, this Exception is raised. The subclass of the error InterpolationError is called.
8. **Exception name: MissingSectionHeaderError**  
   Description: when attempting to parse a document, an Exception is raised, with no sections header.
9. **Exception name: InterpolationSyntaxError**  
   Description: when the text source into which substitutions are made doesn't conform to the needed syntax, an Exception is raised. The subclass is of InterpolationError.
10. **Exception name: Error**  
    Description: Base class for all other configparser\_name **error**s.
11. **Exception name: ParsingError**  
    Description: When errors happen, Exceptions raise, endeavoring to parse a document.  
    Changed in variant 3.2: The filename quality and \_\_init\_\_() parameters were named to source for consistency.

Python Program to Find Difference between Two Strings

In this tutorial, we will write a Python program to find the difference between the two given strings. This problem can be asked in the interview. Let's understand the problem statement and then we will approach to the solution.

Problem Statement -

There are two strings given **s** and **t.** String t is generated by random shuffling string s and then added one more character at any random position. We need to write a Python program that returns the letter added to **t.**

**Example -**

1. Input: s = "zxyc", t = "zxyce"
2. Output: "e"
3. Explanation: 'e' **is** the letter that was added.

**Example -**

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

1. Input: s = "uvw", t = "wyu"
2. Output: "y"
3. Explanation: 'e' **is** the letter that was added.

**Constraints:**

The following constraints should be followed -

* 0 <= s.length <= 1000
* t.length == s.length + 1
* s and t consist of lowercase English letters.

Python Program

Let's understand the following Python program.

**Example -**

1. **class** Solution(object):
2. **def** findTheDifference(self, s, t):
3. ls\_s = [s[i] **for** i **in** range(len(s))]
4. ls\_t = [t[i] **for** i **in** range(len(t))]
5. **for** elem **in** ls\_s:
6. ls\_t.remove(elem)
7. **return**(ls\_t[0])
8. obj = Solution()
9. s = "zxyc"
10. t = "zxyce"
11. **print**(obj.findTheDifference(s, t)

**Output:**

'e'

**Explanation -**

In the above code, we defined the findThedifference() function that takes two strings as arguments. We used the list comprehension to convert the strings into list. Now, we iterate **ls\_s** list, pick single element and remove that element to the second list **ls\_t.** If all element removed from the second element, it means both given strings are same, otherwise return the first element of the second list.

**Solution - 2**

Let's see another solution of the problem.

1. **class** Solution:
2. **def** findTheDifference(self, s: str, t: str) -> str:
3. #sort both the strings
4. s\_list = sorted(s)
5. t\_list = sorted(t)
6. s\_list.append(0) #to make the length equal else we will get list index out of bounds (1 extra char in string2)
7. **for** i **in** range(len(t\_list)):
8. **if** s\_list[i] != t\_list[i]: #if character at i not same for both the strings, we get our answer
9. **return** t\_list[i]
10. obj = Solution()
11. s = "zxyc"
12. t = "zxyce"
13. **print**(obj.findTheDifference(s, t)

**Output:**

e

**Explanation -**

In this tutorial, we used the **sorted()** method, which converts the string into a list of characters in a sorted manner. We created the two lists of strings and added an extra element as 0 to make the length equal; else, we will get the list index out of bounds. Now we iterated the t\_list and checked if the **s\_list** element is not equal to t\_list; if the condition is matched, it returns that element.

# 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

Play Videox[[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack" \t "_blank)

**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

# Split, Sub, Subn functions of re module in python

Before looking into the Split, Sub, Subn functions of the re module in python, let us understand a little bit about the re module offered by python.

A regex or Regular Expression (RE) is a particular text string that is very useful in defining a search pattern in a computer language. It's great for extracting data from text, such as code, files, logs, spreadsheets, and even papers. Regular expressions (also known as REs, regexes, or regex patterns) are a small, highly specialized computer language that is integrated into Python and accessible through the re module. You provide the criteria for the collection of potential strings that you wish to match using this small language; this set may include English phrases, e-mail addresses, TeX instructions, or anything else you desire. Then you may ask things like, "Does this string match the pattern?" or "Is there a pattern match everywhere in this string?" REs can also be used to change or split a string in a variety of ways. Regular expression patterns are converted into bytecodes, which are subsequently performed by a C-based matching engine. To create a bytecode that runs quicker, it may be required to pay close attention to how the engine will execute a particular RE and write the RE in a specific style for advanced use.

Because the regular expression language is short and limited, regular expressions cannot be used to do all string processing jobs. Some tasks can be accomplished using regular expressions, but the expressions are quite complex. In certain circumstances, creating Python code to perform the processing may be preferable; while Python code is slower than a complex regular expression, it is also likely to be more intelligible.

The majority of letters and characters will simply match. The regular expression test, for example, will perfectly match the string test. (A case-insensitive mode may be enabled, allowing this RE to match Test or TEST as well; more on that later.) There are some exceptions to this rule; certain characters are special metacharacters that don't match. Instead, they indicate that something unusual should be matched, or they have an effect on other parts of the RE by repeating or modifying their meaning. [and] are the first strings that are going to be observed by us. They're used to identify a character class, which is a collection of characters to match. Individual characters can be stated, or a range of characters can be indicated by using two characters and a '-' to separate them. [abc], for example, will match any of the letters a, b, or c; this is the same as [a-c], which expresses the same set of characters using a range. Your RE would be [a-z] if you just wanted to match lowercase letters. Inside of classes, metacharacters are inactive.

Pause

Unmute

Current TimeÂ 0:00

/

DurationÂ 4:57

Loaded: 100.00%

Â

Fullscreen

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

By complementing the set, you can match characters who aren't included in the class. An " as the initial character of the class indicates this. For instance, [5] will match any character other than the letter '5.' The caret has no special significance if it appears elsewhere in a character class. [5] will match either a '5' or a ", for instance.

The backslash is maybe the most essential metacharacter. The backslash can be followed by various characters to signify various specific sequences, just like in Python string literals.

the escaping of all the matter characters in a regular expression can be then with the help of the backslash it plays a very important role in defining various characters in the regular expression.

Both special and conventional characters can be used in regular expressions. The simplest regular expressions are the most common characters, such as 'A', 'a', or '0'; they simply match themselves. Ordinary characters can be concatenated, so last matches the string 'last'. Some characters, such as '|' and '(,' are unique. Special characters either represent classes of ordinary characters or impact the interpretation of regular expressions around them.

Repetition qualifiers (\*, +,?, m,n, and so on) cannot be nested directly. This eliminates ambiguity with the non-greedy suffix?, as well as other modifiers in other implementations. Parentheses can be used to add a second repetition to an inner repetition. The equation (?:a6\*) matches any multiple of six 'a' letters, for example.

Those characters discussed above are:

* . (Dot.) This matches any character except a newline in the default mode. This fits any character, including one with a newline, if the DOT ALL flag is set.
* $ Matches the string's end or right before the newline at the end of the string, as well as before a newline in MULTILINE mode. The regular expression foo$ matches only 'foo', while foo matches both 'foo' and 'foobar'.
* ^ (Caret.) Matches the beginning of the string, as well as immediately after each newline in MULTILINE mode.
* + The resulting RE must match one or more repeats of the previous RE. ab+ will match any non-zero number of 'b's after 'a;' it will not match just 'a.'
* \* Causes the resulting RE to match 0 or more repeats of the preceding RE, up to the maximum number of repetitions. ab\* will match any number of 'b's after 'a', 'ab', or 'a'.
* {m} Specifies that exactly m copies of the previous RE should be matched; if there are fewer matches, the entire RE will fail to match. For instance, a6 will match six 'a' characters exactly, but not five.
* ? Causes the resulting RE to match the previous RE's 0 or 1 repetitions. ab? 'a' or 'ab' will be matched.
* {m,n} Causes the resulting RE to match the preceding RE from m to n repeats, striving to match as many as feasible. For instance, a3,5 will match between 3 and 5 'a' letters. When m is omitted, the lower bound is set to zero, and when n is omitted, the upper bound is set to infinity. a4,b, for example, will match 'aaaab' or a thousand 'a' letters followed by a 'b,' but not 'aaab.' If the comma is removed, the modifier will be mistaken with the form described previously.
* {m,n}? Causes the resulting RE to match the preceding RE from m to n repeats, to match as few repetitions as feasible. This qualifier is the non-greedy counterpart to the previous one. For example, a3,5 will match 5 'a' characters in the 6-character string 'aaaaaa,' whereas a3,5? will only match 3 characters.
* | A|B creates a regular expression that matches either A or B, where A and B are arbitrary REs. The '|' can be used to separate an arbitrary number of REs. This can also be used within groups (see below). REs separated by '|' are tried from left to right as the target string is scanned. When one pattern matches perfectly, a branch is allowed. This means that once A matches, B isn't tested again, even if it would result in a lengthier overall match. The '|' operator, in other words, is never greedy. Use | or enclose it inside a character class, as in [|], to match a literal '|'.
* (...) The contents of a group can be recovered after a match has been completed, and can be matched later in the string with the number special sequence, as detailed below. Use (or ) to match the literals ( or ) or wrap them in a character class: [(], [)].
* (?...) This is an extension notation (otherwise, a '?' after a '(' is meaningless). The first character after the '?' determines the construct's meaning and further syntax. The lone exception is (?Pname>...), which does not establish a new group by default. The extensions that are currently supported are listed below.
* (?:...) Regular parentheses in a non-capturing form. The substring matched by the group cannot be retrieved or referenced later in the pattern since it Matches whatever regular expression is inside the parentheses.
* (?<=...) If a match for... that ends at the current location before the current place in the string, it matches. A positive look behind the statement is what it's termed. Because the look behind will back up three characters and verify if the contained pattern matches, (?=abc)def will find a match in 'abcdef'. The contained pattern can only match strings of a certain length, so abc or a|b are acceptable, but a\* and a3,4 are not.

Now let us see the respective codes for all the functions of the re module.

### The split () method in re module in python:

The built-in re module has the split() method, which splits a text based on regular expression matches.

The split() function has the following syntax:

**Syntax:**

1. split(pattern, string, maxsplit=0, flags=0)

This is the syntax:

* **the pattern** is a regular expression with matches that will be utilized as split separators.
* **the string** is the input string that will be split.
* **Maxsplit** determines the maximum number of splits. If maxsplit is one, the resulting list will often have two elements. The resulting list will have three elements if the maxsplit is two, and so on.
* The **flags** parameter is optional and has a value of zero by default. One or more regex flags can be passed to the flags parameter. The flags option modifies the regex engine's behavior while matching a pattern.

The code for the split() method in the re module in python is like this,

**Code:**

1. # A sample python code to demonstrate the use of the split() method of the re module of python
3. # re module is imported to use variables and functions offered by re module
4. **import** re
6. # A sample **class** is written that will have different functions representing the different use **case** scenarios of the split() method of the re module of python
7. **class** NirnayREClass:
9. # A constructor is written that can be used **for** initialization of the **class** variables
10. def \_\_init\_\_(self):
11. pass
13. # the use\_split\_default() method is the first method of the NirnayREClass **class** that will reflect the
14. # usage of split() method with **default** parameters only, these **default** parameters are the regex pattern
15. # and the input string, the split operation is performed on the according to the provided regular expression
16. def use\_split\_default(self):
17. print("Enter the string that you want to split.")
18. ip\_str = input()
19. print("Enter the regular expression for performing the split operation on the input string.")
20. pattern = input()
21. list\_after\_split = re.split(pattern,ip\_str)
22. print("The result after the split operation::")
23. print(list\_after\_split)
25. # the use\_split\_with\_maxsplit\_parameter() method is the second method of the NirnayREClass **class** that will reflect the
26. # usage of split() method with **default** parameters and maxsplits parameter, the **default** parameters are the regex pattern
27. # and the input string, the split operation is performed on the according to the provided regular expression and split is performed only upto specified number of max splits
28. def use\_split\_with\_maxsplit\_parameter(self):
29. print("Enter the string that you want to split.")
30. ip\_str = input()
31. print("Enter the regular expression for performing the split operation on the input string.")
32. pattern = input()
33. print("Enter the maximum number of splits that you want.")
34. max\_splits = **int**(input())
35. list\_after\_split = re.split(pattern,ip\_str,max\_splits)
36. print("The result after the split operation with maxsplit parameter::")
37. print(list\_after\_split)

40. # the use\_split\_with\_maxsplit\_parameter() method is the second method of the NirnayREClass **class** that will reflect the
41. # usage of split() method with **default** parameters and maxsplits and flag parameters, the **default** parameters are the regex pattern
42. # and the input string, the split operation is performed on the according to the provided regular expression and split is performed only
43. # upto specified number of max splits and according to the flag parameter
44. def use\_split\_with\_maxsplit\_and\_flag\_parameter(self):
45. print("Enter the string that you want to split.")
46. ip\_str = input()
47. print("Enter the regular expression for performing the split operation on the input string.")
48. pattern = input()
49. print("Enter the maximum number of splits that you want.")
50. max\_splits = **int**(input())
51. list\_after\_split = re.split(pattern,ip\_str,max\_splits,flags=re.IGNORECASE)
52. print("The result after the split operation with maxsplit and flag parameter::")
53. print(list\_after\_split)

56. # In the main function the object of the above-written **class** is created and all the member methods()
57. # of that **class** are called with the use of the object of the NirnayREClass **class**.
58. def main():
60. re\_object = NirnayREClass()
62. **while**(True):
63. print("Select among the options printed below::")
64. print("1. To use split() method of re module.")
65. print("2. To use split() method of re module with maxsplit parameter.")
66. print("3. To use split() method of re module with maxsplit and flag parameter.")
67. print("4. To finish the code execution and exit.")
68. menu\_choice = input()
69. menu\_choice = **int**(menu\_choice)
71. **if** menu\_choice == 1:
72. re\_object.use\_split\_default()
74. elif menu\_choice == 2:
75. re\_object.use\_split\_with\_maxsplit\_parameter()
77. elif menu\_choice == 3:
78. re\_object.use\_split\_with\_maxsplit\_and\_flag\_parameter()
80. elif menu\_choice == 4:
81. sys.exit()
83. print("Move further with code execution enter (y/n) as the input")
84. continue\_or\_exit = input()
86. **if** continue\_or\_exit == 'y' or continue\_or\_exit == 'Y':
87. pass
88. elif continue\_or\_exit == 'n' or continue\_or\_exit == 'N':
89. sys.exit()

92. #the main fucntion is called to start the execution of the code.
93. **if** \_\_name\_\_ == '\_\_main\_\_':
94. main()

**Output:**

nirnay@superbook:~$ python3 re1.py

Select among the options printed below::

1. To use split() method of re module.

2. To use split() method of re module with maxsplit parameter.

3. To use split() method of re module with maxsplit and flag parameter.

4. To finish the code execution and exit.

1

Enter the string that you want to split.

Hi my name is nirnay khajuria and I'm author of this python code

Enter the regular expression for performing the split operation on the input string.

\s+

The result after the split operation::

['Hi', 'my', 'name', 'is', 'nirnay', 'khajuria', 'and', "I'm", 'author', 'of', 'this', 'python', 'code']

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use split() method of re module.

2. To use split() method of re module with maxsplit parameter.

3. To use split() method of re module with maxsplit and flag parameter.

4. To finish the code execution and exit.

2

Enter the string that you want to split.

This example will show the use case of maxsplit parameter of the split() method of re module.

Enter the regular expression for performing the split operation on the input string.

\s+

Enter the maximum number of splits that you want.

6

The result after the split operation with maxsplit parameter::

['This', 'example', 'will', 'show', 'the', 'use', 'case of maxsplit parameter of the split() method of re module.']

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use split() method of re module.

2. To use split() method of re module with maxsplit parameter.

3. To use split() method of re module with maxsplit and flag parameter.

4. To finish the code execution and exit.

2

Enter the string that you want to split.

This example will show the use case of maxsplit parameter of the split() method of re module.

Enter the regular expression for performing the split operation on the input string.

\s+

Enter the maximum number of splits that you want.

9

The result after the split operation with maxsplit parameter::

['This', 'example', 'will', 'show', 'the', 'use', 'case', 'of', 'maxsplit', 'parameter of the split() method of re module.']

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use split() method of re module.

2. To use split() method of re module with maxsplit parameter.

3. To use split() method of re module with maxsplit and flag parameter.

4. To finish the code execution and exit.

3

Enter the string that you want to split.

This example will show the use case of maxsplit parameter of the split() method of re module.

Enter the regular expression for performing the split operation on the input string.

\s+

Enter the maximum number of splits that you want.

20

The result after the split operation with maxsplit and flag parameter::

['This', 'example', 'will', 'show', 'the', 'use', 'case', 'of', 'maxsplit', 'parameter', 'of', 'the', 'split()', 'method', 'of', 're', 'module.']

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use split() method of re module.

2. To use split() method of re module with maxsplit parameter.

3. To use split() method of re module with maxsplit and flag parameter.

4. To finish the code execution and exit.

4

**Explanation:**

So in the above-written code, we have seen the usage of the split method and how we can use this method with different parameters. In the above-written code, we have created a class that has different functions representing the different use case scenario of the split function with its different parameters the first function is used to display the usage of the split method with its default input parameters there are two defaults input parameters which are the input string and the regular expression these two input parameters are used to split the input string based on the regular expression specified the second function represent the usage of the split method with the max split parameter in this scenario the splitting of the input string based on provided regular expression is limited up to the max split parameter specified by the user and in the last function we have used the flag parameter of the split function.

### Sub() function of re module in python:

Return the string obtained by replacing the replacement repl with the leftmost non-overlapping instances of the pattern in the string. If the pattern isn't found, the string is left alone. In other words, n becomes a single newline character, r becomes a carriage return, and so on. Unknown ASCII letter escapes are set aside for future use and are viewed as mistakes. Other undiscovered escapes, such as &, are left to their own devices.

**Syntax:**

1. re.sub(pattern, repl, string, count=0, flags=0)

* A regular expression that you want to match is called a pattern. The pattern can also be a Pattern object, in addition to a regular expression.
* The count parameter indicates the maximum number of matches that the sub() method should replace.
* repl is the replacement string. The sub() function will replace all matches if the count parameter is set to 0 or omitted entirely.
* flags are one or more regex flags that change the pattern's default behavior.

The sub() function searches the string for a pattern and replaces the matched strings with the replacement string (repl). If the sub() function fails to discover a match, the original string is returned. Otherwise, the sub() function replaces the matches and returns the string. The leftmost non-overlapping repetitions of the pattern are replaced with the sub() function. In the following example, you'll see it in further detail.

**Code:**

1. # A sample python code to demonstrate the use of the sub() method of the re module of python
3. # re module is imported to use variables and functions offered by re module
4. **import** re
6. # A sample **class** is written that will have different functions representing the different use **case** scenarios of the sub() method of the re module of python
7. **class** NirnayREClass:
9. # A constructor is written that can be used **for** initialization of the **class** variables
10. def \_\_init\_\_(self):
11. pass
13. # the use\_sub\_default() method is the first method of the NirnayREClass **class** that will reflect the
14. # usage of sub() method with **default** parameters only, these **default** parameters are the regex pattern
15. # and the input string, the replace operation is performed on the input string according to the
16. # provided regular expression
17. def use\_sub\_default(self):
18. print("Enter the string on which you want to perform the replacement.")
19. ip\_str = input()
20. print("Enter the regular expression according to which you want to do replace on the input string.")
21. pattern = input()
22. list\_after\_sub = re.sub(pattern,ip\_str)
23. print("The result after the sub operation::")
24. print(list\_after\_sub)
26. # the use\_sub\_with\_repl\_parameter() method is the second method of the NirnayREClass **class** that will reflect the
27. # usage of sub() method with **default** parameters and rel parameter, the **default** parameters are the regex pattern
28. # and the input string, the replace operation is performed on the according to the provided regular expression
29. # and matching string is replaced with the repl parameter
30. def use\_sub\_with\_repl\_parameter(self):
31. print("Enter the string on which you want to perform the replacement.")
32. ip\_str = input()
33. print("Enter the regular expression according to which you want to do replace on the input string.")
34. pattern = input()
35. print("Enter repl string.")
36. repl\_str = **int**(input())
37. list\_after\_sub = re.sub(pattern,repl\_str,ip\_str)
38. print("The result after the sub operation::")
39. print(list\_after\_sub)


43. # the use\_sub\_with\_repl\_and\_count\_parameter() method is the second method of the NirnayREClass **class** that will reflect the
44. # usage of sub() method with **default** parameters and rel parameter, the **default** parameters are the regex pattern
45. # and the input string, the replace operation is performed on the according to the provided regular expression
46. # and matching string is replaced with the repl parameter upto count parameter
47. def use\_sub\_with\_repl\_and\_count\_parameter(self):
48. print("Enter the string on which you want to perform the replacement.")
49. ip\_str = input()
50. print("Enter the regular expression according to which you want to do replace on the input string.")
51. pattern = input()
52. print("Enter repl string.")
53. repl\_str = input()
54. print("Enter max count.")
55. max\_c = **int**(input())
56. list\_after\_sub = re.sub(pattern,repl\_str,ip\_str,max\_c)
57. print("The result after the sub operation::")
58. print(list\_after\_sub)

61. # In the main function the object of the above-written **class** is created and all the member methods()
62. # of that **class** are called with the use of the object of the NirnayREClass **class**.
63. def main():
65. re\_object = NirnayREClass()
67. **while**(True):
68. print("Select among the options printed below::")
69. print("1. To use sub() method of re module.")
70. print("2. To use sub() method of re module with repl parameter.")
71. print("3. To use sub() method of re module with repl and count parameters.")
72. print("4. To finish the code execution and exit.")
73. menu\_choice = input()
74. menu\_choice = **int**(menu\_choice)
76. **if** menu\_choice == 1:
77. re\_object.use\_sub\_default()
79. elif menu\_choice == 2:
80. re\_object.use\_sub\_with\_repl\_parameter()
82. elif menu\_choice == 3:
83. re\_object.use\_sub\_with\_repl\_and\_count\_parameter()
85. elif menu\_choice == 4:
86. sys.exit()
88. print("Move further with code execution enter (y/n) as the input")
89. continue\_or\_exit = input()
91. **if** continue\_or\_exit == 'y' or continue\_or\_exit == 'Y':
92. pass
93. elif continue\_or\_exit == 'n' or continue\_or\_exit == 'N':
94. sys.exit()

97. #The main function is called to start the execution of the code.
98. **if** \_\_name\_\_ == '\_\_main\_\_':
99. main()

**Output:**

nirnay@superbook:~$ python3 re2.py

Select among the options printed below::

1. To use sub() method of re module.

2. To use sub() method of re module with repl parameter.

3. To use sub() method of re module with repl and count parameters.

4. To finish the code execution and exit.

1

Enter the string on which you want to perform the replacement.

This-is-a-simple-string-having-hyphen-instead-of-space

Enter the regular expression according to which you want to do replace on the input string.

\-

The result after the sub-operation::

This is a simple string having a hyphen instead of a space

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use sub() method of re module.

2. To use sub() method of re module with repl parameter.

3. To use sub() method of re module with repl and count parameters.

4. To finish the code execution and exit.

2

Enter the string on which you want to perform the replacement.

This-is-a-simple-string-having-hyphen-instead-of-space

Enter the regular expression according to which you want to do replace on the input string.

\-

Enter repl string.

\_

The result after the sub-operation::

This\_is\_a\_simple\_string\_having\_hyphen\_instead\_of\_space

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use sub() method of re module.

2. To use sub() method of re module with repl parameter.

3. To use sub() method of re module with repl and count parameters.

4. To finish the code execution and exit.

3

Enter the string on which you want to perform the replacement.

This-is-a-simple-string-having-hyphen-instead-of-space

Enter the regular expression according to which you want to do replace on the input string.

\-

Enter repl string.

\_

Enter max count.

5

The result after the sub-operation::

This\_is\_a\_simple\_string\_having-hyphen-instead-of-space

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use sub() method of re module.

2. To use sub() method of re module with repl parameter.

3. To use sub() method of re module with repl and count parameters.

4. To finish the code execution and exit.

1

Enter the string on which you want to perform the replacement.

Only replace the hyphen in this-sentence

Enter the regular expression according to which you want to do replace on the input string.

\-

The result after the sub-operation::

Only replace the hyphen in this sentence

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use sub() method of re module.

2. To use sub() method of re module with repl parameter.

3. To use sub() method of re module with repl and count parameters.

4. To finish the code execution and exit.

2

Enter the string on which you want to perform the replacement.

Replace +this string+

Enter the regular expression according to which you want to do replace on the input string.

\+

Enter repl string.

\*

The result after the sub-operation::

Replace \*this string\*

Move further with code execution enter (y/n) as the input

Y

Select among the options printed below::

1. To use sub() method of re module.

2. To use sub() method of re module with repl parameter.

3. To use sub() method of re module with repl and count parameters.

4. To finish the code execution and exit.

3

Enter the string on which you want to perform the replacement.

My m@il is email@website.com

Enter the regular expression according to which you want to replace on the input string.

\@

Enter repl string.

a

Enter max count.

1

The result after the sub-operation::

My mail is email@website.com

Move further with code execution enter (y/n) as the input

N

**Explanation:**

For the above-written code, we have seen the usage of the sub method and its usage with different parameters. In the above-written code, we have created a class that has different functions representing the different use case scenario of the sub function with its different parameters the first function is used to display the usage of the sub method with its default input parameters there are two defaults input parameters which are the input string and the regular expression these two input parameters are used to replace the input string based on the regular expression specified, the second function represent the usage of the sub method with the repl parameter in this scenario the replacement of the input string is based on provided regular expression is limited up to the count parameter is depicted in the last function.

### Subn function of re module:

The regular expressions (RE) module in Python has a function called subn() that defines strings or a group of strings or patterns that match it. The RE module must be imported before we can utilize this function. The subn() method is similar to the sub() function, but it additionally gives you a count of how many replacements you've done.

**Syntax:**

1. re.subn(pattern, repl, string, count=0, flags=0)

* The first argument, pattern, specifies the text or pattern to replace.
* The second option, repl, specifies the string/pattern that will be used to replace the pattern.
* The third option, string, specifies the string that will be used for the subn() function.
* The fourth option, count, specifies how many replacements should be made before the subn() action is performed.
* The fifth argument, flags, aids in code reduction and performs similar roles to split operations.

**Code:**

1. # A sample python code to demonstrate the use of the subn() method of the re module of python
3. # re module is imported to use variables and functions offered by re module
4. **import** re
5. **import** sys
7. # A sample **class** is written that will have different functions representing the different use **case** scenarios of the subn() method of the re module of python
8. **class** NirnayREClass:
10. # A constructor is written that can be used **for** initialization of the **class** variables
11. def \_\_init\_\_(self):
12. pass
14. # the use\_subn\_default() method is the first method of the NirnayREClass **class** that will reflect the
15. # usage of subn() method with **default** parameters only, these **default** parameters are the regex pattern
16. # and the input string, the replace operation is performed on the input string according to the
17. # provided regular expression
18. def use\_subn\_default(self):
19. print("Enter the string on which you want to perform the replacement.")
20. ip\_str = input()
21. print("Enter the regular expression according to which you want to do replace on the input string.")
22. pattern = input()
23. list\_after\_sub = re.sub(pattern,ip\_str)
24. print("The result after the sub operation::")
25. print(list\_after\_sub)
27. # the use\_subn\_with\_repl\_parameter() method is the second method of the NirnayREClass **class** that will reflect the
28. # usage of subn() method with **default** parameters and rel parameter, the **default** parameters are the regex pattern
29. # and the input string, the replace operation is performed on the according to the provided regular expression
30. # and matching string is replaced with the repl parameter
31. def use\_subn\_with\_repl\_parameter(self):
32. print("Enter the string on which you want to perform the replacement.")
33. ip\_str = input()
34. print("Enter the regular expression according to which you want to do replace on the input string.")
35. pattern = input()
36. print("Enter repl string.")
37. repl\_str = **int**(input())
38. list\_after\_sub = re.sub(pattern,repl\_str,ip\_str)
39. print("The result after the sub operation::")
40. print(list\_after\_sub)


44. # the use\_subn\_with\_repl\_and\_count\_parameter() method is the second method of the NirnayREClass **class** that will reflect the
45. # usage of subn() method with **default** parameters and rel parameter, the **default** parameters are the regex pattern
46. # and the input string, the replace operation is performed on the according to the provided regular expression
47. # and matching string is replaced with the repl parameter upto count parameter
48. def use\_subn\_with\_repl\_and\_count\_parameter(self):
49. print("Enter the string on which you want to perform the replacement.")
50. ip\_str = input()
51. print("Enter the regular expression according to which you want to do replace on the input string.")
52. pattern = input()
53. print("Enter repl string.")
54. repl\_str = input()
55. print("Enter max count.")
56. max\_c = **int**(input())
57. list\_after\_sub = re.sub(pattern,repl\_str,ip\_str,max\_c)
58. print("The result after the sub operation::")
59. print(list\_after\_sub)

62. # In the main function the object of the above-written **class** is created and all the member methods()
63. # of that **class** are called with the use of the object of the NirnayREClass **class**.
64. def main():
66. re\_object = NirnayREClass()
68. **while**(True):
69. print("Select among the options printed below::")
70. print("1. To use subn() method of re module.")
71. print("2. To use subn() method of re module with repl parameter.")
72. print("3. To use subn() method of re module with repl and count parameters.")
73. print("4. To finish the code execution and exit.")
74. menu\_choice = input()
75. menu\_choice = **int**(menu\_choice)
77. **if** menu\_choice == 1:
78. re\_object.use\_subn\_default()
80. elif menu\_choice == 2:
81. re\_object.use\_subn\_with\_repl\_parameter()
83. elif menu\_choice == 3:
84. re\_object.use\_subn\_with\_repl\_and\_count\_parameter()
86. elif menu\_choice == 4:
87. sys.exit()
89. print("Move further with code execution enter (y/n) as the input")
90. continue\_or\_exit = input()
92. **if** continue\_or\_exit == 'y' or continue\_or\_exit == 'Y':
93. pass
94. elif continue\_or\_exit == 'n' or continue\_or\_exit == 'N':
95. sys.exit()

98. #The main function is called to start the execution of the code.
99. **if** \_\_name\_\_ == '\_\_main\_\_':
100. main()

**Output:**

Select among the options printed below::

1. To use subn() method of re module.

2. To use subn() method of re module with repl parameter.

3. To use subn() method of re module with repl and count parameters.

4. To finish the code execution and exit.

1

Enter the string on which you want to perform the replacement.

This is a sample string to show the usage of subn() function.

Enter the regular expression according to which you want to do replace on the input string.

()

The result after the subm operation::

('This is a sample string to show the usage of subn function.', 1)

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use subn() method of re module.

2. To use subn() method of re module with repl parameter.

3. To use subn() method of re module with repl and count parameters.

4. To finish the code execution and exit.

2

Enter the string on which you want to perform the replacement.

The repl p@r@meter is used to repl@ce with a specific string

Enter the regular expression according to which you want to do replace on the input string.

\@

Enter repl string.

a

The result after the subn operation::

('The repl parameter is used to replace with a specific string', 3)

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use subn() method of re module.

2. To use subn() method of re module with repl parameter.

3. To use subn() method of re module with repl and count parameters.

4. To finish the code execution and exit.

3

Enter the string on which you want to perform the replacement.

My m@il is email@website.com

Enter the regular expression according to which you want to do replace on the input string.

\@

Enter repl string.

a

Enter max count.

1

The result after the subn operation::

('My mail is email@website.com', 1)

Move further with code execution enter (y/n) as the input

y

Select among the options printed below::

1. To use subn() method of re module.

2. To use subn() method of re module with repl parameter.

3. To use subn() method of re module with repl and count parameters.

4. To finish the code execution and exit.

4

**Explanation:**

For the above-written code, we have seen the usage of the subn method and its usage with different parameters. In the above-written code, we have created a class that has different functions representing the different use case scenario of the subn function with its different parameters the first function is used to display the usage of the subn method with its default input parameters there are two defaults input parameters which are the input string and the regular expression these two input parameters are used to replace the input string based on the regular expression specified, the second function represent the usage of the subn method with the repl parameter in this scenario the replacement of the input string is based on provided regular expression is limited up to the count parameter is depicted in the last function.

## Conclusion:

So, in this article, we understood the usage of the Split, Sub, Subn functions of the re module in python. And we have also seen the sample python code to use these functions in the different scenarios.

# Namespace in Python

In this tutorial, we will learn about the namespace in Python, the structure used to organize the symbolic names assigned to objects in a Python program, why namespace is important, and how we can use them in our Python program. Let's have a brief introduction to a namespace.

## What is Namespace?

A namespace is a way of providing the unique name for each object in Python. Everything in Python is an object, i.e., a variable or a method. In other words, it is a collection of the defined symbolic names along with the information about the object that each name references. A namespace can be understood as a dictionary where a name represents a key and objects are values. Let's understand it with a real-life example - A namespace is like a surname. A "Peter" name might be difficult to find in the class if there are multiple "Peter," but when we particularly ask for "Peter Warner" or "Peter Cummins,". It might be rare to find the same name and surname in a class for multiple students.

The namespace helps the Python interpreter to understand what exact method or variable is trying to point out in the code. So its name gives more information - **Name** (which means name, a unique identifier) + **Space** (related to scope).

In Python, there are four types of namespaces which are given below.

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

* Built-in
* Global
* Enclosing
* Local

As these namespace various have lifetimes, Python interpreter creates namespaces as necessary and deletes them when they are no longer needed.

Let's understand the various types of namespace in Python.

### The Built-in Namespace

As its name suggests, it contains pre-defined names of all of Python's built-in objects already available in Python. Let's list these names with the following command.

Open the Python terminal and type the following command.

**Command -**

1. dir(\_\_builtins\_\_)

**Output:**

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',

'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '\_\_build\_class\_\_', '\_\_debug\_\_', '\_\_doc\_\_', '\_\_import\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable',

'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

The built-in namespace creates by the Python interpreter when its starts up. These are terminated when Python interpreter terminates.

## The Global Namespace

The global namespace consists of any names in Python at any level of the main program. It is created when the main body executes and remains in existence until the interpreter terminates.

The Python interpreter creates a global namespace for any module that our Python loads with the import statement. To get more information, visit our Python Module.

## The Local and Enclosing Namespaces

The function uses the local namespaces; the Python interpreter creates a new namespace when the function is executed. The local namespaces remain in existence until the function terminates. The function can also consist of another function. We can define one function inside another as below.

**Example -**

1. def f():
2. print('Initiate f()')
4. def g():
5. print('Initiate g()')
6. print('End g()')
7. **return**
9. g()
11. print('Initiate f()')
12. **return**
13. f()

In the above example, the function **g()** is defined within the body of f(). Inside the **f()** we called the **g()** and called the main **f()** function. Let's understand the working of the above function -

* When we calls f(), Python creates a new namespace for f().
* Similarly, the f() calls g(), g() gets its own separate namespace.
* Here the g() is a local namespace created for f() is the enclosing namespace.

Each of these namespace is terminated when the function is terminated.

## Scope of the Object/Variable

The scope is term which defines the coding region from a particular Python object is accessible. Every object/variable has its scope where we can access that particular variable in the program. For example - A variable in a function can only access inside the function. Let's understand the following example -

**Example -**

1. def scope\_func():
2. print("Inside scope\_func")
3. def scope\_inner\_func():
4. var = 20
5. print("Inside inner function, value of var:",var)
6. scope\_inner\_func()
7. print("Try printing var from outer function: ",var)
8. scope\_func()

**Output:**

Inside scope\_func

Inside inner function, value of var: 20

Traceback (most recent call last):

File "d:/Python Project/listproblems.py", line 343, in

scope\_func()

File "d:/Python Project/listproblems.py", line 342, in scope\_func

print("Try printing var from outer function: ",var)

NameError: name 'var' is not defined

## Python Namespace Dictionaries

In the earlier tutorial, we have discussed that namespaces are as the dictionaries in which keys are the object names and values are the objects themselves. Python implements global and local namespaces as dictionaries. Python comes with the **globals()** and **locals()** methods that allow us to access global and local namespace dictionaries.

## The globals() Method

The globals() method returns a reference to the current global namespace dictionary. We can use it to access the objects in the global namespace. Let's see the below example.

**Example -**

1. >>> type(globals())
2. <**class** 'dict'>
3. >>> globals()
4. {'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <**class** '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>}

As we can see that, there are many built-in entries in **globals()** method. It may be differ according to your operating system and Python version. Now let's define the global variable and observe the differences.

1. >>> a = 20
2. >>> globals()
3. {'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <**class** '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, 'a': 20}

After the assignment of a = 20, a new global variable assigned to the global namespace dictionary. We can access the values as we access in the dictionaries. Let's see the below example.

1. >>> a = 20
2. >>> globals()
3. {'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <**class** '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, 'a': 20}
4. >>> a
5. 20
6. >>> globals()['a']
7. 20

We can modify the dictionary value using the globals() function.

1. >>> globals()['a'] = 100
2. >>> a
3. 100

Now the new value of a will be appeared in the global dictionaries.

## The locals() Function

Python also provides the locals() method similar to globals() but accesses objects in the local namespace instead. Let's see the following example.

**Example -**

1. >>> def func(a, b):
2. ...     str1 = "Hello"
3. ...     loc = locals()
4. ...     print(loc)
5. ...
6. >>> func(10, 20)
7. {'a': 10, 'b': 20, 'str1': 'Hello'}

When we call the func(10, 20), the locals() return the dictionary representing the function's local namespace. In the function scope, we defined the local variable str1; the local namespace included the function arguments since they are local to the func().

However, when we call the **locals()** function, it behaves the same as the **globals()** function. There is a small difference between **globals()** and **locals()** function. The **globals()** save the return value and subsequently define additional variables. The new variables will show up in the dictionary along with their value. Let's see the below example.

**Example -**

1. >>> glob\_var = globals()
2. >>> glob\_var
3. {'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <**class** '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, 'a': 100, 'func': <function func at 0x000001670FB85160>, 'glob\_var': {...}, 'x': 100, 'y': 'JavaTpoint'}
4. >>> x = 100
5. >>> glob\_var
6. {'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <**class** '\_frozen\_importlib.BuiltinImporter'>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, 'a': 100, 'func': <function func at 0x000001670FB85160>, 'glob\_var': {...}, 'x': 100, 'y': 'JavaTpoint'}

Here the **glob\_var** is a reference to the global namespace dictionary. The new assignment statements **x** and **y** appeared in the **glob\_var** dictionary.

## Changing Variables Out of Scope

The function can change the argument in the calling environment by passing different value and sometimes it can't change the value.

* A function cannot modify an immutable argument.
* A **mutable argument** cannot be redefined wholesale, but it can be modified in place.

Let's understand the following scenario.

**Example -**

1. x = 20
2. def func():
3. x = 40
4. print(x)
6. func()
8. print(x)

**Output:**

40

20

We define a global variable x = 20 and also in function with the same name. When the func() execute, it creates the new local variable reference to an integer object whose value is 40. Inside the **func()** body, the assignment statement won't affect the global object.

However, a function can modify an object of a mutable type outside its local scope. Let's understand the below example.

**Example -**

1. my\_list = ['Hello', 'From', 'JavaTpoint']
2. def func():
3. my\_list[1] = 'Welcome to'
4. **return** my\_list
6. print(func())

The my\_list is a list and it is mutable type. The func() can modify inside my\_list even though its outside the local scope. But, if we try to reassigned the my\_list, it will create the new local object and won't modify the global my\_list. Let's see the below example.

**Example -**

1. my\_list = ['Hello', 'from', 'JavaTpoint']
2. def func():
3. my\_list = ['A', 'B', 'C', 'D', 'E', '']
4. **return** my\_list
6. print(func())

**Output:**

['A', 'B', 'C', 'D', 'E']

## Conclusion

We have covered the namespace, how we can use it, and the variable's scope. A short Python program will create many different objects. In a complex Python program, this number can be in the thousand. Python namespace helps the interpreter keep track of these objects and their names.

[**Next →**](https://www.javatpoint.com/namespace-in-python)[**← Prev**](https://www.javatpoint.com/snakeviz-library-in-python)

# Difference between Materialized View and View

Views are the most important concept of the database management system. In the interview, it is a popular and commonly asked question, much like truncate vs. delete, correlated, correlated vs. noncorrelated subquery, or primary key vs. unique key. This tutorial will learn about the difference between view and materialized view.

The original table is stored in the physical memory of the database. In which we can access all attributes by running queries. But sometimes, we have to restrict the user to fetch some data from the table and allow them to get only permitted attributes. Suppose we have a student table, and the user can search for student names, marks, branches, ages, and other information about the student. But the user must not be allowed to fetch the student's mobile numbers and addresses.

In such cases, it is a good way to create a view that can show the data of the required attributes of the table. We can achieve such functionality by creating a virtual table or view and Materialized view.

Let's understand the concepts of the views.

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

## What is View?

Views are the virtual table of the database that acts as an actual relation. It is not part of the logical relational model of the database system. They are created using the select query, but the result is not stored in the physical memory. Every time we fire the query to the view, the view returns the updated and latest data from the original table. It means the views are generated every time the view is accessed. In this view, the query definition is stored in the database itself. We can create as many views as we want.

The point is to remember that when we make any change in the virtual table, it will reflect in the original table. If we make any changes in the original or base table, the changes will be reflected in the view. This makes the query performance very slow. For example - we create a view from the join of the two more tables. In this case, we have to resolve the joins and create them every time we try to fetch the data.

However, it provides some advantages, such as it doesn't require memory to store the data. We can create the view using the below syntax.

### Syntax -

1. Create view as <select query expression>

#### Note - Views that are created using the DISTINCT clause, Group By clause, check constraint, read only option can't be modified.

### Advantages of View

The following are some important advantages of the views.

* Views can simply the base table that consists of a huge amount of data. We can define a virtual table as per the data requirements.
* Views can easily define by joining one of more tables.
* Views can hide the complexity of the data.
* Views can limit its degree of exposure for underlying tables to the outer world.

## What is Materialized View?

Materialized views are also known as virtual tables, but the result of the query expression is saved in physical memory. The query definition is also stored in the database. We can also consider them a Physical copy of the original base tables. It is primarily used in the context of warehousing of data. There is no standard view to define materialized view in SQL. However, few database management systems offer custom extensions to use materialized views. Unlike the normal view, they are not updated each time they are used. Instead, we need to update it manually or with the help of the trigger. The process of updating the Materialized view is known as Materialized View Maintenance.

It stores the result in the physical memory, it responds faster than the normal view because the normal view is created whenever we run the query. It is mainly used for summarizing, pre-computing, replicating and distributing data, etc.

Let's understand the syntax of the materialized view.

1. Create Materialized View view\_name
2. Build [clause] Refresh [ type]
3. ON [trigger ]
4. As <query expression>

In the above syntax, the Build clause decides when to populate the materialized view. It contains two options -

* **IMMEDIATE -** It populate the materialized view immediately.
* **DEFFERED -** Need to refresh materialized view manually at least once.

Refresh type define the how to update the materialized view. There are three options -

* **FAST -** The materialized view logs is required against the source table in advance, without logs, the creation fails. A fast refresh is attempted. A fast refresh is attempted.
* **COMPLETE -** The table segment supporting the materialized view is truncated and repopulated completely using the associated query.
* **FORCE -** The materialized logs is not required. A fast refresh is attempted.

On trigger defines when to update the materialized view. The refresh can be triggered in the two ways -

* **ON COMMIT -** When the data change is committed in one of the dependent tables. The refresh is triggered.
* **ON DEMAND -** A refresh happens when we schedule task or a manual request.

We have discussed the basic concept of the normal view and materialized view. Now, let's see the difference between normal view and materialized view.

### Advantages of Materialized View

The following are the some important advantages of the materialized view.

* Materialized view optimizes the query performance, using the same sub-query results every time.
* The data is not updated frequent in the materialized view, user needs to update data manually or using the trigger clause. It reduces the chances of any error and returns the efficient outcome.
* Materialized views are transparent and automatically maintained with the help of snowflake, which is a background services.

## Difference between View and Materialized View

The following are the important difference between the view and materialized view.

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **View** | **Materialized View** |
| 1. | Views are the virtual projection of the base table. The query expressions are stored in the database but not the resulting data of query expression. | The resulting data and query expression both are saved in the physical memory (database system). |
| 2. | Views are generally created by joining one or more tables. | Materialized views are primary used for the warehousing of data. |
| 3. | A view is virtual table that is based on the select query | It is also known as snapshot view of the data which provides access to duplicate, physical data in a separate table. |
| 4. | DML commands cannot be used if they are created using the multiple tables. | DML commands can be used in materialized view no matter how they are created. |
| 5. | There is no updation cost involve in normal view. | It does have the updation cost associated with it. |
| 6. | Views respond slowly. It causes the query performances. | Materialized view responds faster. Because it store data in the database. |
| 7. | Views are defined according to the fixed design architecture approach which is based on the SQL standard defining a view. | There is no predefined SQL standard to defining it, the database provides the functionality in the form of the extension. |
| 8. | Views are more effective when the data is accessed infrequently and data in table get updated on frequent basis. | Materialized views are mostly used when data is accessed more frequently and data is not updated frequently. |

## Conclusion

Views are played a crucial role in data retrieval. In this tutorial, we have covered the definition of the views and materialized view and its few important advantages. We have defined both views with a suitable example. Here you can understand the primary difference between both views and can choose according to your requirements.

# How to Sort Tuple in Python

Tuples are a type of data type of a variable that lets us store multiple objects in one place. A tuple is an ordered and immutable (we cannot update elements in a tuple) collection of items. There are 4 in-built Python data structures to store elements, one of them is a tuple, and the others are List, Dictionary, and Set, each with its own set of properties and use. They are written in round brackets.

## Sorting a Tuple in Python

### Using sort()

The sort() method is often used to sort elements of a list in ascending order, with the first element being sorted by default. We can sort a tuple by converting it to a list first and then apply this function. This function sorts the list in place and returns None.

**Input**

1. tuple\_ = ('Itika', 'Arshia', 'Peter', 'Parker')
2. list(tuple\_).sort()
3. print(tuple\_)
4. print(type(tuple\_))

**Output:**

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

('Itika', 'Arshia', 'Peter', 'Parker')

<class 'tuple'>

### Using sorted()

In Python, use the sorted() built-in function to sort a Tuple. The tuple should be passed as an argument to the sorted() function. The tuple items are sorted (by default) in ascending order in the list returned by the function. We can use a tuple to convert this list data type to a tuple ().

The reverse parameter to the sorted() function can also specify the sorting order. Ascending is the default sorting order. The items are sorted in descending order when reverse=True is set. We can also specify a key function whose returned values are used to sort items. We take a tuple, tuple\_, having integer values, and sort it in ascending order in the following program.

**Input**

1. tuple\_ = (5, 2, 24, 3, 1, 6, 7)
2. sorted\_ = tuple(sorted(tuple\_))
3. print('Sorted Tuple :', sorted\_)
4. print(type(sorted\_))

**Output:**

Sorted Tuple : (1, 2, 3, 5, 6, 7, 24)

<class 'tuple'>

Now we are sorting the tuple in descending order using same function. Pass reverse=True to the sorted() function to sort the tuple in descending order.

**Input**

1. tuple\_ = (5, 2, 24, 3, 1, 6, 7)
2. sorted\_ = tuple(sorted(tuple\_, reverse=True))
3. print('Sorted Tuple :', sorted\_)
4. print(type(sorted\_))

**Output:**

Sorted Tuple : (24, 7, 6, 5, 3, 2, 1)

<class 'tuple'>

### Sorting a Tuple Based on a Key Function

A key is a function that takes a value and returns a value. For each of the items in tuple, this key function is applied, and the returned value is used for comparison to sort the items. In the following program, we sort tuple of strings based on the length of strings. For this case, we may use len() builtin function as key.

**Input**

1. tuple\_ = ('abhd', 'sbchcwsc', 'sjs', 'sxshs')
2. sorted\_ = tuple(sorted(tuple\_, key=len))
3. print('Sorted Tuple :', sorted\_)
4. print(type(sorted\_))

**Output:**

Sorted Tuple : ('sjs', 'abhd', 'sxshs', 'sbchcwsc')

<class 'tuple'>

## Sorting List of Tuples

### Using sorted()

Let's look at how to sort a list in Python using a tuple. Consider the following scenario: we wish to sort the list of tuples. We must sort tuples according to any key given to us. This can be accomplished using the sorted() function, which sorts items using a key and stores the key index for sorting the given tuples. The Python execution of this approach is as follows:

**Input**

1. # Sorting list of tuples according to a key
2. def middle(n):
3. **return** n[1]
5. # function to sort the tuple
6. def sort(list\_of\_tuples):
7. **return** sorted(list\_of\_tuples, key = middle)
9. # driver code
10. list\_ = [(34, 21, 56), (24, 12, 32), (42, 34, 42), (27, 11, 32)]
12. print("Sorted list of Tuples:"),
13. print(sort(list\_))

**Output:**

Sorted list of Tuples:

[(27, 11, 32), (24, 12, 32), (34, 21, 56), (42, 34, 42)]

### Using Bubble Sort

Bubble sort is just a sorting algorithm for sorting a list of any number of elements. If the adjoining item in a given list are in the incorrect order, it swaps them. It then repeats this process until all of the elements have been sorted.

In this example, we'll use the bubble sort algorithm to sort a list of tuples.

**Input**

1. roll = [('Arshia', 26), ('Itika', 53), ('Peter', 82), ('Parker', 74), ('MJ', 45)]
2. first = 0
3. last = len(roll)
4. **for** k in range(0, last):
5. **for** l in range(0, last-k-1):
6. **if** (roll[l][first] > roll[l + 1][first]):
7. new\_item = roll[l]
8. roll[l]= roll[l + 1]
9. roll[l + 1]= new\_item
10. print(roll)

**Output:**

[('Arshia', 26), ('Itika', 53), ('MJ', 45), ('Parker', 74), ('Peter', 82)]

# Write Python Program to Search an Element in Sorted Array

In this tutorial, we will solve one of the interesting problems of the sorted array. But there is one twist; the given array may be rotated at the some index position. It means the few elements of the sorted array might be rotated at the given index. To understand it better, let's understand the following problem statement.

## Problem Statement -

An array is given as list1 sorted in the ascending order with the distinct values. It can be rotated at an unknown pivot index k (1 <= k < list1.length) such that the resulting array is [list1[k], list1[k+1], ..., list1[n-1], list1[0], list1[1], ..., list1[k-1]] (**0-indexed**).

Suppose the given array is [4, 5, 6, 7, 8, 9, 10, 11] and it might be rotated at pivot index 3 and become [8, 9, 10, 11, 4, 5, 6, 7].

Given an array list1 after the possible rotation and an integer target, return the index of the target; if exist otherwise, return -1.

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

**Example - 1:**

**Input:** list1 = [4, 5, 6, 7, 0, 1, 2], target = 0

**Output:** 4

**Example - 2:**

**Input:** list1 = [4, 5 ,6, 7, 0, 1, 2], target = 3**Output:** -1

Now we will find the best approach to solve this problem. It is a slightly tricky question, but we can easily write the code when we break down the solution. Let's understand the following solution.

## Solution

To the search related problems, we think to implement the binary search algorithm first because it is easy and quite efficient algorithm to search an element. However, the given list must be sorted, in our case the array is sorted but rotated at some place. Let's see the following array.

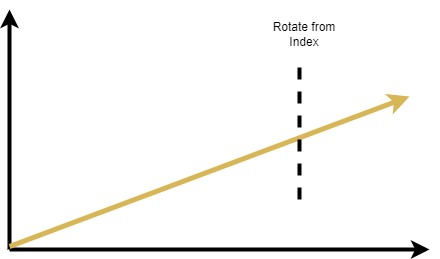
**list1 = [4, 5, 6, 7, 0, 1, 2]**

If we observe closely, we can see that the normal array will be [0, 1, 2, 4, 5, 6, 7], and it is rotated at the index of 3.

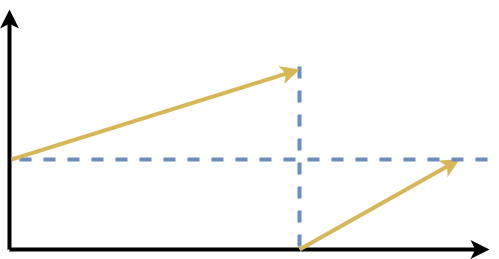
Now the array can see that the array is split into two parts, the left portion is sorted as [4, 5, 6, 7,] and the right portion is [0, 1, 2]

Let's represents it using the graph to take an advantage of binary search algorithm for this problem.

We draw a basic graph that has the contiguously basic increasing line might not be necessary linear but always in increasing order.



Hence the new graph will be formed as below.



Now we can find a pattern that can help us write the solution. As we know that there are three-pointers in the binary search - left, right, and mid. There are two portions in the array, both independently sorted. As we know that there are three pointers in the binary search - left, right, and mid.

Suppose in the given array **[4, 5, 6, 7, 0, 1, 2]** the target value is **0** and the mid is **6,** it means if the target value is greater than 6, it won't be exited on the left side.

Now we can search the element on the right side. What if the target value is less than the mid?

In that scenario, 4, 5 are less than 6 and 0, 1, 2 less than 6, so how do we know which side to be searched for?

So here check the leftmost value of the list, if it is less than the target value then we don't need to search it at left side anymore.

The search will be placed at right side from mid + 1. But if the target value is greater than the leftmost value, we search the element at the left side.

Now consider the mid-value as 1; the only element is less than 1 is 0. So the search will happen on the left side. We check the leftmost value and compare it with the target value if it is greater than the rightmost value. Let's implement it using Python code.

#### Note - To check whether the middle value belongs to the left portion or right, we can use compare with the leftmost value to mid. If the mid-value is greater than the leftmost value, it must be belonged to the left portion and vice-versa.

### Python Code -

1. **class** Solution:
2. def search(self, list1: List[**int**], target: **int**) -> **int**:
3. l, r = 0, len(list1) - 1
5. **while** l<=r:
6. mid = (l + r) // 2
7. **if** target == list1[mid]:
8. **return** mid
10. # left sorted portion
12. **if** list1[l] <= list1[mid]:
13. **if** target > list1[mid] or target < list1[l]:
14. l = mid + 1
15. **else**:
16. r = mid - 1
17. **else**:
19. **if** target < list1[mid] or target > list1[r]:
20. r = mid - 1
21. **else**:
22. l = mid + 1
24. **return** -1
25. list1 = [4, 5, 6, 7, 0, 1, 2]
26. target = 0
27. obj = Solution()
28. print("The element is at the", obj.search(list1, target), "index")

**Output:**

The element is at the 4 index

It might seem slightly complicated but once understand the concept. It will be clear.

## Time Complexity

The time complexity will be O(logN) because binary search requires log n comparison to find the element. The space complexity will be O(1) because we don't require the extra space.

# Write Python Program to Search an Element in Sorted Array

In this tutorial, we will solve one of the interesting problems of the sorted array. But there is one twist; the given array may be rotated at the some index position. It means the few elements of the sorted array might be rotated at the given index. To understand it better, let's understand the following problem statement.

## Problem Statement -

An array is given as list1 sorted in the ascending order with the distinct values. It can be rotated at an unknown pivot index k (1 <= k < list1.length) such that the resulting array is [list1[k], list1[k+1], ..., list1[n-1], list1[0], list1[1], ..., list1[k-1]] (**0-indexed**).

Suppose the given array is [4, 5, 6, 7, 8, 9, 10, 11] and it might be rotated at pivot index 3 and become [8, 9, 10, 11, 4, 5, 6, 7].

Given an array list1 after the possible rotation and an integer target, return the index of the target; if exist otherwise, return -1.

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

**Example - 1:**

**Input:** list1 = [4, 5, 6, 7, 0, 1, 2], target = 0

**Output:** 4

**Example - 2:**

**Input:** list1 = [4, 5 ,6, 7, 0, 1, 2], target = 3**Output:** -1

Now we will find the best approach to solve this problem. It is a slightly tricky question, but we can easily write the code when we break down the solution. Let's understand the following solution.

## Solution

To the search related problems, we think to implement the binary search algorithm first because it is easy and quite efficient algorithm to search an element. However, the given list must be sorted, in our case the array is sorted but rotated at some place. Let's see the following array.

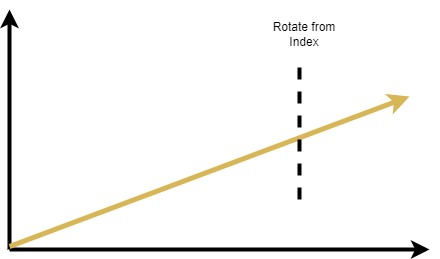
**list1 = [4, 5, 6, 7, 0, 1, 2]**

If we observe closely, we can see that the normal array will be [0, 1, 2, 4, 5, 6, 7], and it is rotated at the index of 3.

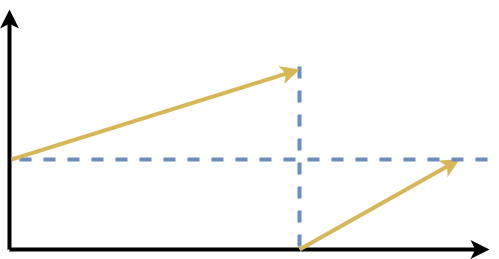
Now the array can see that the array is split into two parts, the left portion is sorted as [4, 5, 6, 7,] and the right portion is [0, 1, 2]

Let's represents it using the graph to take an advantage of binary search algorithm for this problem.

We draw a basic graph that has the contiguously basic increasing line might not be necessary linear but always in increasing order.



Hence the new graph will be formed as below.



Now we can find a pattern that can help us write the solution. As we know that there are three-pointers in the binary search - left, right, and mid. There are two portions in the array, both independently sorted. As we know that there are three pointers in the binary search - left, right, and mid.

Suppose in the given array **[4, 5, 6, 7, 0, 1, 2]** the target value is **0** and the mid is **6,** it means if the target value is greater than 6, it won't be exited on the left side.

Now we can search the element on the right side. What if the target value is less than the mid?

In that scenario, 4, 5 are less than 6 and 0, 1, 2 less than 6, so how do we know which side to be searched for?

So here check the leftmost value of the list, if it is less than the target value then we don't need to search it at left side anymore.

The search will be placed at right side from mid + 1. But if the target value is greater than the leftmost value, we search the element at the left side.

Now consider the mid-value as 1; the only element is less than 1 is 0. So the search will happen on the left side. We check the leftmost value and compare it with the target value if it is greater than the rightmost value. Let's implement it using Python code.

#### Note - To check whether the middle value belongs to the left portion or right, we can use compare with the leftmost value to mid. If the mid-value is greater than the leftmost value, it must be belonged to the left portion and vice-versa.

### Python Code -

1. **class** Solution:
2. def search(self, list1: List[**int**], target: **int**) -> **int**:
3. l, r = 0, len(list1) - 1
5. **while** l<=r:
6. mid = (l + r) // 2
7. **if** target == list1[mid]:
8. **return** mid
10. # left sorted portion
12. **if** list1[l] <= list1[mid]:
13. **if** target > list1[mid] or target < list1[l]:
14. l = mid + 1
15. **else**:
16. r = mid - 1
17. **else**:
19. **if** target < list1[mid] or target > list1[r]:
20. r = mid - 1
21. **else**:
22. l = mid + 1
24. **return** -1
25. list1 = [4, 5, 6, 7, 0, 1, 2]
26. target = 0
27. obj = Solution()
28. print("The element is at the", obj.search(list1, target), "index")

**Output:**

The element is at the 4 index

It might seem slightly complicated but once understand the concept. It will be clear.

## Time Complexity

The time complexity will be O(logN) because binary search requires log n comparison to find the element. The space complexity will be O(1) because we don't require the extra space.

# The randint() Function in Python

In this tutorial, we will learn about the "randint()" function in Python.

The "randint()" is a built-in function of the random module in Python. The random module is used for getting access to various functions like generating random numbers by using the randint() function.

First, we have to import the random module in Python to use the randint() function. This function is basically used for creating pseudo-randomness.

### Syntax:

1. randint(start\_range, end\_range)

### Parameters:

**(start\_range, end\_range):** Both the parameters must be integer type values.

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

### Parameters:

It will return the random integer in the range [start\_range, end\_range], including both starting and ending numbers.

### Errors and Exceptions:

**ValueError:** Returns a ValueError when the user passes floating point as parameters.

**TypeError:** Returns a TypeError when the user passes anything other than integer value as parameters.

### Example 1:

To get the random number between the given range of two positive numbers, two negative numbers, and one positive and one negative number.

1. **import** random as rnd
2. # First, we will generate the random number between any positive number range
3. random\_1 = rnd.randint(55, 75)
4. print ("The any random number between 55 and 75 is % s" % (random\_1))
6. # Then, we will generate the random number between two given negative number range
7. random\_2 = rnd.randint(-40, -20)
8. print ("The any random number between -40 and -20 is % s" % (random\_2))
10. # We will now, generate the random number between a positive number and a negative number range
11. random\_3 = rnd.randint(-20, 20)
12. print ("The any random number between -20 and 20 is % s" % (random\_3))

**Output:**

**1#**

The any random number between 55 and 75 is 74

The any random number between -40 and -20 is -40

The any random number between -20 and 20 is -12

**2#**

The any random number between 55 and 75 is 74

The any random number between -40 and -20 is -29

The any random number between -20 and 20 is -2

### Example 2:

In this example, we will see how the users can get the ValueError in the Python program while using the randint() function.

1. # First, we will **import** the random module
2. **import** random as rnd
4. # If the user passes any floating point values as the parameters in the randint() # function.
6. random\_1 = rnd.randint(2.543, 12.786)
7. print (random\_1)

**Output:**

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

in

4 # If the user passes any floating point values as the parameters in the randint() function.

5

----> 6 random\_1 = rnd.randint(2.543, 12.786)

7 print(random\_1)

c:\users\User Name\appdata\local\programs\python\python39\lib\random.py in randint(self, a, b)

336 """

337

--> 338 return self.randrange(a, b+1)

339

340

c:\users\user name\appdata\local\programs\python\python39\lib\random.py in randrange(self, start, stop, step)

300 istart = int(start)

301 if istart != start:

--> 302 raise ValueError("non-integer arg 1 for randrange()")

303 if stop is None:

304 if istart > 0:

ValueError: non-integer arg 1 for randrange()

### Example 3:

In this example, we will see how the user can get the TypeError in Python while using the randint() function.

1. # First, we will **import** the random module
2. **import** random as rnd
4. # If the user passes any string or character value as the parameters in the
5. # randint() function
7. random\_2 = rnd.randint('String', 'Character')
8. print (random\_2)

**Output:**

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

in

4 # If the user passes any string or character value as the parameters in the randint() function

5

----> 6 random\_2 = rnd.randint('String', 'Character')

7 print (random\_2)

c:\users\user name\appdata\local\programs\python\python39\lib\random.py in randint(self, a, b)

336 """

337

--> 338 return self.randrange(a, b+1)

339

340

TypeError: can only concatenate str (not "int") to str

## Applications:

The users can use the randint() function for simulating a lucky draw game.

Suppose we have participated in a lucky draw game such as "Casino Game". The player will get three chances to guess the number between 1 to 36. If we have guessed the number correctly, we will win, else we will lose the game.

**Example: code for application**

1. # First, we will **import** the randint function
2. # from the random module in Python
3. from random **import** randint as rdt
5. # We will create a function which can generate a **new**
6. # random number everytime the code will execute
7. def generator\_1():
8. **return** rdt(1, 36)
10. # Now, we will create a function which takes the input from the user and returns
11. # **true** or **false** depending whether the
12. # user has guessed the correct number and wins the lucky draw or not.
13. def random\_guess():
15. # The calls generator\_1() which returns a
16. # random integer between 1 and 36
17. random\_number\_1 = generator\_1()
19. # here, we will define the number of
20. # guesses the user will get
21. guess\_left\_1 = 3
23. # now, we will set the flag variable **for** checking
24. # the win-condition **for** the user
25. flag\_1 = 0
27. # Then, we will loop the number of times
28. # the user will get the chances
29. **while** guess\_left\_1 > 0:
31. # Here, we will take a input from the user.
32. guess\_1 = **int** (input ("Please select your number to "
33. "enter the lucky draw game \n"))
35. # then, we will check whether the guess of the user
36. # matched the generated win-condition or not.
37. **if** guess\_1 == random\_number\_1:
39. # Then, we will set the flag as 1 **if** the user have guessed
40. # the correct number and then loop will broke
41. flag\_1 = 1
42. **break**
44. **else**:
46. # If the guess of the user does not matched
47. # the win-condition then it will print
48. print ("You have guessed Wrong Number!!")
50. # then, we will decrease the number of
51. # guesses left by 1
52. guess\_left\_1 -= 1
54. # If the condition of winning is satisfied then,
55. # the function random\_guess will **return** "True"
56. **if** flag\_1 == 1:
57. **return** True
59. # Otherwise, the function will **return** "False"
60. **else**:
61. **return** False
63. # Driver code
64. **if** \_\_name\_\_ == '\_\_main\_\_':
65. **if** random\_guess() == True:
66. print ("Congratulation!! You have Won the game.")
67. **else** :
68. print ("Sorry, You have Lost the game!")

**Output:**

**1#**

Please select your number to enter the lucky draw game

3

You have guessed Wrong Number!!

Please select your number to enter the lucky draw game

2

You have guessed Wrong Number!!

Please select your number to enter the lucky draw game

34

You have guessed Wrong Number!!

Sorry, you have Lost the game!

**2#**

Please select your number to enter the lucky draw game

14

You have guessed Wrong Number!!

Please select your number to enter the lucky draw game

12

You have guessed Wrong Number!!

Please select your number to enter the lucky draw game

3

Congratulation!! You have Won the game.

## Conclusion

In this tutorial, we have discussed the randint() function of Python's random module. We have shown the types of errors the user can get while using the randint() function. We have also discussed how the randint() function is used for creating an application of lucky draw games.

# Python Program for How to Check if a Given Number is Fibonacci Number or Not

In this tutorial, we will learn how to check if the given number is a Fibonacci number or not.

Here, we have a number "n", and we have to check if it is a Fibonacci number. Starting number of Fibonacci series are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, and so on.

**Example:**

1. Input number: 5
2. Output: Yes, the given number is a Fibonacci\_Number.
4. Input number:  22
5. Output: No, the given number is not a Fibonacci\_Number.
7. Input number: 55
8. Output: Yes, the given number is a Fibonacci\_Number.

We can also use the following property of the Fibonacci number for checking if the given number is a Fibonacci number or not:

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

* A number is Fibonacci number only if one of (5 \* R \* R + 4) or ( 5 \* R \* R - 4) or both of them are the perfect square.

### Python Program to Check if Given Number if Fibonacci or Not:

1. **import** math as m
3. # Here, we will create a utility function that will **return** **true** **if** K is a perfect square
4. def is\_Perfect\_Square(K):
5. s = **int**(m.sqrt(K))
6. **return** s \* s == K
8. # Now, we will create a function which will **return** "true" **if** R is a Fibinacci Number,
9. # **else** it will **return** "false"
10. def is\_Fibonacci(R):
12. # R is a Fibinacci number only **if** one of (5 \* R \* R + 4) or (5 \* R \* R - 4) or both
13. # of them are perferct square
14. **return** is\_Perfect\_Square(5 \* R \* R + 4) or is\_Perfect\_Square(5 \* R \* R - 4)
16. # Now, we will create a utility function **for** testing the above functions
17. **for** J in range(1, 22):
18. **if** (is\_Fibonacci(J) == True):
19. print ("Number:", J, ": Yes, the given number is a Fibonacci\_Number")
20. **else**:
21. print ("Number:", J, ": No, the given number is not a Fibonacci\_Number")

**Output:**

Number: 1 : Yes, the given number is a Fibonacci\_Number

Number: 2 : Yes, the given number is a Fibonacci\_Number

Number: 3 : Yes, the given number is a Fibonacci\_Number

Number: 4 : No, the given number is not a Fibonacci\_Number

Number: 5 : Yes, the given number is a Fibonacci\_Number

Number: 6 : No, the given number is not a Fibonacci\_Number

Number: 7 : No, the given number is not a Fibonacci\_Number

Number: 8 : Yes, the given number is a Fibonacci\_Number

Number: 9 : No, the given number is not a Fibonacci\_Number

Number: 10 : No, the given number is not a Fibonacci\_Number

Number: 11 : No, the given number is not a Fibonacci\_Number

Number: 12 : No, the given number is not a Fibonacci\_Number

Number: 13 : Yes, the given number is a Fibonacci\_Number

Number: 14 : No, the given number is not a Fibonacci\_Number

Number: 15 : No, the given number is not a Fibonacci\_Number

Number: 16 : No, the given number is not a Fibonacci\_Number

Number: 17 : No, the given number is not a Fibonacci\_Number

Number: 18 : No, the given number is not a Fibonacci\_Number

Number: 19 : No, the given number is not a Fibonacci\_Number

Number: 20 : No, the given number is not a Fibonacci\_Number

Number: 21 : Yes, the given number is a Fibonacci\_Number

## Conclusion

In this tutorial, we have discussed how a user can check if the given number is a Fibonacci number or not by using Python.

# Python Program for Calculating the Sum of Squares of First n Natural Numbers

In this tutorial, we will learn how to calculate the sum of squares of first n natural numbers using Python.

We have a positive integer "N", and our task is to calculate (12+ 22 + 32 + 42 + 52 +… + N2)

**Example:**

1. Input: N = 6
2. Output: 91
3. 12 + 22 + 32 + 42 + 52 + 62
4. = 1 + 4 + 9 + 16 + 25 +36
5. = 91
7. Input:  N = 2
8. Output: 5
9. 12 + 22
10. = 1 + 4
11. = 5

**Method 1:** O(N)

In this method, a user has to run the loop from 1 to "N" natural number, and for each K, 1 <= K <= N. The user has to find the K2 to calculate the sum.

**Example:**

1. # First, we will create a function
2. # which will **return** the sum of
3. # squares of first "NN"
4. # natural numbers
5. def square\_sum(NN) :
7. # here, we will iterate K from 1
8. # and NN **for** finding
9. # the square numbers from K to NN and
10. # then add to sum.
11. sum\_1 = 0
12. **for** K in range(1, NN + 1) :
13. sum\_1 = sum\_1 + (K \* K)
15. **return** sum\_1
17. # Driven Program
18. NN = **int** (input ("Please enter the 'N' natural number: "))
19. print (square\_sum(NN)

**Output:**

Please enter the 'N' natural number: 56

60116

**Method 2:** O(1)

In this method, the user can calculate the sum of the square of the first "N" natural numbers by using the following formula:

1. (NN \* (NN + 1) \* (2 \* NN + 1)) / 6

**For Example:**

1. For NN = 6, sum\_1 = (6 \* (6 + 1) \* (2 \* 6 + 1)) / 6
2. = (6 \* (7) \* (13)) / 6
3. = (546) / 6
4. = 91
5. For NN = 56, sum\_1 = (56 \* (56 + 1) \* (2 \* 56 + 1)) / 6
6. = (56 \* (57) \* (113)) / 6
7. = (3,60,696) / 6
8. = 60116

**Code:**

1. # First, we will create a function
2. # which will **return** the sum of
3. # squares of first "NN"
4. # natural numbers
5. def square\_sum(NN) :
6. **return** (NN \* (NN + 1) \* (2 \* NN + 1)) // 6
8. # Driven Program
9. NN = **int**( input("Please enter the 'N' natural number: "))
10. print (square\_sum(NN))

**Output:**

Please enter the 'N' natural number: 87

223300

### How to Avoid Early Overflow:

For the large "NN" natural number, the value of [(NN \* (NN + 1) \* (2 \* NN + 1)) / 6] would probably overflow. The users can avoid this situation by using the fact that (NN \* (NN + 1)) must be divisible by 2.

**Example:**

1. # First, we will create a function
2. # which will **return** the sum of
3. # squares of first "NN"
4. # natural numbers
6. def square\_sum(NN):
7. **return** (NN \* (NN + 1) \* (2 \* NN + 1)) // 3
9. # Driven Program
10. NN = **int**( input("Please enter the 'N' natural number: "))
11. print (square\_sum(NN))

**Output:**

Please enter the 'N' natural number: 567

121844520

## Conclusion

In this tutorial, we have explained two methods for calculating the sum of squares of "N" natural numbers using python and avoiding the code's overflow.

# Python Program to Count the Number of Matching Characters in a Pair of String

In this tutorial, we will discuss how the users can write a Python program for counting the number of matching characters in the given pair of strings.

We will pass the pair of non-empty strings. The program will count the number of matching characters in that pair of strings. Here, we will consider that the strings we passed are having duplicates of characters in them.

**Example:**

1. Input: string\_1 = 'Javtpoint'
2. strint\_2 = 'Juvpionk'
3. Output: no. matching characters in the pairs of strings: 6
4. (That is, the matching characters are: - J, v, p, i, o, n)
5. Input: string\_1: 'zyxw531@7#'
6. string\_2: 'xwuv234#'
7. Output: no. matching characters in the pairs of strings: 4
8. (That is, the matching characters are: - x, w, 3, #)

### Approach 1:

* **Step 1:** we will initialize the counter variable with 0.
* **Step 2:** we will iterate over the first string from the first character to the last character.
* **Step 3:** If the character extracted from the string\_1 is found in the string\_2. And if the first occurrence index of that character in the string\_1 is the same as that of the index of the current extracted character, then it will increase the value of the counter by 1.

We will then use string.find('character') in Python to find the same characters. This will return the first occurrence index of the character in the string if found; otherwise, it will return "-1".

Play Videox[[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack" \t "_blank)

**Example:**

1. string\_1 = 'zyxwvwwv'
2. string\_1find('w') ? 3
3. string\_1find('v') ? 4
4. string\_1find('t') ? -1

* **Step 4:** print the output value of the counter.

**Example: Method 1**

1. # First, we will define the count function
2. def count\_1(string\_1, string\_2):
3. count, find\_1 = 0, 0
5. # The loop will execute till the length of string\_1 and it will
6. # Stores the value of string\_1 character by character and stores in "store\_1" at every iteration.
7. **for** store\_1 in string\_1:
9. # This will check **if** the extracted from of the characters of string\_1
10. # is present in string\_2 or not.
11. **if** string\_2.find(store\_1) >= 0 and find\_1 == string\_1.find(store\_1):
12. count += 1
13. find\_1 += 1
14. print ('The no. matching characters in the pairs of strings: ', count)
16. # Main function
17. def main():
18. string\_1 = str(input ("Please enter the characters for String 1: "))
19. string\_2 = str(input ("Please enter the characters for String 2: "))
20. count\_1(string\_1, string\_2) # At last, calling the count function

23. # Driver Code
24. **if** \_\_name\_\_ == "\_\_main\_\_":
25. main()

**Output:**

Please enter the characters for String 1: ajg 78y

Please enter the characters for String 2: gjy 23r

The no. matching characters in the pairs of strings: 2

### Approach 2:

* **Step 1:** In this method, we will use the set() function for removing the duplicate on the given strings.
* **Step 2:** We will use the set(intersection) on both strings.
* **Step 3:** We will use the len() function for calculating the length of the "matched\_characters\_1" string.

**Example 2: Method 2**

1. # First, we will define the count function
2. def count\_1(string\_1, string\_2):
3. # The set of characters of string\_1
4. set\_string\_1 = set(string\_1)
6. # The set of characters of string2
7. set\_string\_2 = set(string\_2)
9. # We will use "&" intersection mathematical operation on the sets
10. # the unique characters present in both the strings
11. # will be stored in matched\_characters\_1 set variable
12. matched\_characters\_1 = set\_string\_1 & set\_string\_2
14. #Then, we will print the length of matched\_characters\_1 set
15. # which will give the number of matched characters in the pair of strings.
16. Print ('The number matching characters in the pairs of strings: ' + str (len (matched\_characters\_1)))

19. # Driver code
20. **if** \_\_name\_\_ == "\_\_main\_\_" :
22. string\_1 = str(input ("Please enter the characters for String 1: "))
23. string\_2 = str(input ("Please enter the characters for String 2: "))
25. count\_1(string\_1, string\_2) # At last, calling the count function

**Output:**

Please enter the characters for String 1: awe ret #$65

Please enter the characters for String 2: rty urw @!34

The number matching characters in the pairs of strings: 4

### Approach 3:

* **Step 1:** We will import the Re module.
* **Step 2:** We will use the re.search() function to check if any character of string\_1 is present in string\_2 and if it does, then 1 will be added into the counter variable.

**Example 3: Method 3**

1. **import** re
2. string\_1 = str(input ("Please enter the characters for String 1: "))
3. string\_2 = str(input ("Please enter the characters for String 2: "))
5. count = 0
6. **for** store\_1 in string\_1:
7. **if** re.search(store\_1, string\_2):
8. count = count + 1
9. print ('The number matching characters in the pairs of strings: ', count)

**Output:**

Please enter the characters for String 1: learning

Please enter the characters for String 2: working

The number matching characters in the pairs of strings: 5

## Conclusion

In this tutorial, we have discussed different methods of writing a Python program for counting the number of matching characters in the given pair of strings.

Tuple to Dictionary in Python

In this tutorial, we will discuss how we can convert a tuple to a dictionary in Python.

We know that the elements of tuples are enclosed within parentheses and the elements of a dictionary are present in the form of a key-value pair and are enclosed within curly brackets.

We will use the following techniques to convert a tuple to a dictionary in Python-

1. Using setdefault()
2. Using dict()
3. Using Dictionary Comprehension
4. Using zip() and dict()

Let's get started with the first one,

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

Using setdefault()

The function of setdefault() is to return the value associated with a key and if the key is not present it is inserted with a default value.

The following program illustrates how it can be used in a Python program.

1. #creating a function
2. def convert\_dict(tup, dic):
3. **for** i, j in tup:
4. dic.setdefault(i, []).append(j)
5. **return** dic
6. #initialising the tuple values
7. tuple\_values = [("English", 2001), ("Hindi", 2002), ("Mathematics", 2003),
8. ("Computer Science", 2004), ("Physics", 2005), ("Chemistry", 2006)]
9. res\_dictionary = {}
10. #displaying the resultant dictionary
11. print ("The converted dictionary is: ", convert\_dict(tuple\_values,res\_dictionary))

**Output:**

The converted dictionary is: {'English': [2001], 'Hindi': [2002], 'Mathematics': [2003], 'Computer Science': [2004], 'Physics': [2005], 'Chemistry': [2006]}

**Explanation-**

1. In the first step, we have created a function that takes tuple and dictionary as an input.
2. After this, we have used for loop to use the **setdefault()** and append the subject name and the subject code.
3. Now we have initialized the values of our tuple and declared the resultant dictionary as {}.
4. On executing the above program, the expected output is displayed.

In the second program, we will learn how dict() can be used for the same.

Using dict()

The dict() is used to create a dictionary in Python, let's see how it can add meaning to our program.

Consider the program given below,

1. #creating a function
2. def convert\_dict(tup, dic):
3. #use of dict()
4. dic = dict(tup)
5. **return** dic
6. #initialising the tuple values
7. tuple\_values = [("English", 2001), ("Hindi", 2002), ("Mathematics", 2003),
8. ("Computer Science", 2004), ("Physics", 2005), ("Chemistry", 2006)]
9. res\_dictionary = {}
10. #displaying the resultant dictionary
11. print ("The converted dictionary is: ", convert\_dict(tuple\_values,res\_dictionary))

**Output:**

The converted dictionary is: {'English': 2001, 'Hindi': 2002, 'Mathematics': 2003, 'Computer Science': 2004, 'Physics': 2005, 'Chemistry': 2006}

**Explanation-**

1. In the first step, we have created a function that takes tuple and dictionary as an input.
2. After this, we have used for loop to use the dict() that takes tuple as a parameter and returns a dictionary.
3. Now we have initialized the values of our tuple and declared the resultant dictionary as {}.
4. On executing the above program, the expected output is displayed.

In the third program, we will see how dictionary comprehension can help us.

Using Dictionary Comprehension

The program below shows the same,

1. #initialising the tuple values
2. sub\_names = ('English', 'Hindi', 'Mathematics', 'Computer Science', 'Physics', 'Chemistry')
3. sub\_codes = (2001, 2002, 2003, 2004, 2005, 2006)
4. #displaying the tuples
5. print("The values in sub\_names are: ", sub\_names)
6. print("The values in sub\_codes are: ", sub\_codes)
7. **if** len(sub\_names) == len(sub\_codes):
8. res\_dict = {sub\_names[i]: sub\_codes[i] **for** i, \_ in enumerate(sub\_codes)}
9. #displaying the resultant dictionary
10. print("The resultant dictionary is: ", res\_dict)

**Output:**

The values in sub\_names are: ('English', 'Hindi', 'Mathematics', 'Computer Science', 'Physics', 'Chemistry')

The values in sub\_codes are: (2001, 2002, 2003, 2004, 2005, 2006)

The resultant dictionary is: {'English': 2001, 'Hindi': 2002, 'Mathematics': 2003, 'Computer Science': 2004, 'Physics': 2005, 'Chemistry': 2006}

**Explanation-**

1. Firstly, we have initialized the two tuples, sub\_names, and sub\_codes, and displayed them.
2. After this, the decision-making keyword, if is used to check whether the length of both the tuples is the same or not, if it is the same then the functionality defined in the dictionary comprehension is performed.
3. On executing the given program, the desired output is displayed.

In the last program, we will learn how zip() and dict() can be used in the Python program.

Using zip() and dict()

We have understood how the dict() works, here we will be applying both dict() and zip(), the zip() method takes the iterable items and appends them to form a single tuple.

The following program illustrates the same-

1. #initialising the tuple values
2. sub\_names = ('English', 'Hindi', 'Mathematics', 'Computer Science', 'Physics', 'Chemistry')
3. sub\_codes = (2001, 2002, 2003, 2004, 2005, 2006)
4. #displaying the tuples
5. print("The values in sub\_names are: ", sub\_names)
6. print("The values in sub\_codes are: ", sub\_codes)
7. **if** len(sub\_names) == len(sub\_codes):
8. #using zip() and dict()
9. res\_dict = dict(zip(sub\_names, sub\_codes))
10. #displaying the resultant dictionary
11. print("The resultant dictionary is: ", res\_dict)

**Output:**

The values in sub\_names are: ('English', 'Hindi', 'Mathematics', 'Computer Science', 'Physics', 'Chemistry')

The values in sub\_codes are: (2001, 2002, 2003, 2004, 2005, 2006)

The resultant dictionary is: {'English': 2001, 'Hindi': 2002, 'Mathematics': 2003, 'Computer Science': 2004, 'Physics': 2005, 'Chemistry': 2006}

**Explanation-**

1. Firstly, we have initialized the two tuples, sub\_names, and sub\_codes, and displayed them.
2. After this, the decision-making keyword, if is used to check whether the length of both the tuples is the same or not, if it is the same then the functionality involving zip() and dict() is performed.
3. On executing the given program, the desired output is displayed.

Conclusion

In this tutorial, we learned the different methods of converting a tuple to a dictionary in Python.

Reverse a tuple in Python

We know that tuples are the data structures present in Python in which elements of different data types can be enclosed in parenthesis.

In this tutorial, we will learn how we can reverse a tuple in Python.

Consider the following examples to understand our objective-

1. Input - (21, 54, 1, 'apple', 'fruits', 11.7)
2. Output - (11.7, 'fruits', 'apple', 1, 54, 21)

**Explanation -**

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

We can observe here that since we have reversed the elements of the tuple, the float value present at the last comes in the first place.

Let's see one more example-

1. Input - (20, 30, 50, 19.3, 41, 'comic books')
2. Output - ('comic books', 41, 19.3, 50, 30, 20)

**Explanation -**

We can observe here that since we have reversed the elements of the tuple, the string value present at the last comes in the first place.

We will use the following methods to reverse a tuple in Python,

1. Using Slicing
2. Using reversed() method
3. Using Generator in Python
4. Using the indexing technique

So, let's get started with the first one:

Using Slicing

The slicing technique used in Python to slice the elements that lie in a specific range.

The following program illustrates how they can be used.

1. #initializing the tuple
2. tuple\_values = (1, 2, 'Python', 'Java', 23.4, 77, 10)
3. #displaying the original tuple
4. print("The original tuple is: ", tuple\_values)
5. #using slicing
6. tuple\_values = tuple\_values[::-1]
7. #displaying the tuple
8. print("The reversed tuple is: ", tuple\_values)

**Output:**

The original tuple is: (1, 2, 'Python', 'Java', 23.4, 77, 10)

The reversed tuple is: (10, 77, 23.4, 'Java', 'Python', 2, 1)

**Explanation-**

It's time to have a look at the explanation of the above program-

1. In the first step, we have initialized our tuple with different values.
2. After this, we have displayed our tuple and then used slicing to print the elements of our tuple in reverse order by specifying the step as -1.
3. Finally, we have displayed the reversed tuple.

In the second program, we will learn how method can be used.

Using reversed() method

Consider the program given below,

1. #initializing the tuple
2. tuple\_values = (1, 2, 'Python', 'Java', 23.4, 77, 10)
3. #displaying the original tuple
4. print("The original tuple is: ", tuple\_values)
5. #using slicing
6. tuple\_values = tuple(reversed(tuple\_values))
7. #displaying the tuple
8. print("The reversed tuple is: ", tuple\_values)

**Output:**

The original tuple is: (1, 2, 'Python', 'Java', 23.4, 77, 10)

The reversed tuple is: (10, 77, 23.4, 'Java', 'Python', 2, 1)

**Explanation-**

Let's understand what happened in the above program,

1. In the first step, we have initialized our tuple with different values.
2. After this, we have displayed our tuple and then used the **reversed()** to print the elements of our tuple in reverse order.
3. Finally, we have displayed the reversed tuple.

In the third program, we will see how generators can be used for the same.

Using Generator in Python

The program given below demonstrates how generators can be used in our Python program.

1. #initializing the tuple
2. tuple\_values = (1, 2, 'Python', 'Java', 23.4, 77, 10)
3. #displaying the original tuple
4. print("The original tuple is: ", tuple\_values)
5. #creating a function
6. def reverse\_tup(tuple\_values):
7. **for** i in reversed(range(len(tuple\_values))):
8. yield tuple\_values[i]
9. #displaying the items in reverse order
10. **for** i in reverse\_tup(tuple\_values):
11. print("The element of a reversed tuple are: ", i)

**Output:**

The original tuple is: (1, 2, 'Python', 'Java', 23.4, 77, 10)

The elements of a reversed tuple are: 10

The elements of a reversed tuple are: 77

The elements of a reversed tuple are: 23.4

The elements of a reversed tuple are: Java

The elements of a reversed tuple are: Python

The elements of a reversed tuple are: 2

The elements of a reversed tuple are: 1

**Explanation-**

It's time to have a glance at the explanation,

1. In the first step, we have initialized our tuple with different values.
2. After this, we have displayed our tuple and then created a function that takes a tuple as its parameter and helps us to obtain the tuple in reversed order using the concept of generators.
3. Finally, we have displayed the reversed tuple.

Finally, we will see how indexing can help us to meet our objective.

Using the indexing technique

The following program illustrates how it can be done

1. #initializing the tuple
2. tuple\_values=(1, 2, 'Python', 'Java', 23.4, 77, 10)
3. #displaying the original tuple
4. print("The original tuple is: ", tuple\_values)
5. #displaying the items in reverse order
6. **for** i in range(len(tuple\_values)):
7. print("The element of a reversed tuple is: ", tuple\_values[-(i+1)])

**Output:**

The original tuple is: (1, 2, 'Python', 'Java', 23.4, 77, 10)

The element of a reversed tuple is: 10

The element of a reversed tuple is: 77

The element of a reversed tuple is: 23.4

The element of a reversed tuple is: Java

The element of a reversed tuple is: Python

The element of a reversed tuple is: 2

The element of a reversed tuple is: 1

**Explanation-**

It's time to have a look at the explanation of the above program-

1. In the first step, we have initialized our tuple with different values.
2. After this, we have displayed our tuple and then created a function that takes the elements of our tuple, uses for loop to traverse the tuple, and then print the elements in reverse order by specifying the index as -1.
3. Finally, we have displayed the reversed tuple.

Conclusion

In this tutorial, we discussed the different approaches of reversing a tuple in Python.

# Difference between module and function in Python

Python is a programming language that is considered to be progressive and known for its optimization capabilities. Python skims down redundant characteristics of programming and makes the tools rich in utilization. In the following tutorial, we will discuss the difference between module and function in the Python programming language.

## What is a Module?

A **module** is simply a Python file with a **.py** extension that a programmer can import inside another Python program.

The name of the Python file becomes the name of the module.

The module consists of definitions and implementation of -

Play Videox[[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack" \t "_blank)

1. **Classes**
2. **Variables**
3. **Functions**

That can be utilized within another program.

**Benefits of modules**

1. **Reusability:**Working with modules makes the code reusable.
2. **Simplicity:**The module targets a small proportion of the problem rather than aiming at the complete problem.
3. **Scoping:**A Module defines a distinct namespace that supports avoiding collisions between identifiers.

## What is a Function?

A **function** is a block of organized and reusable code that we can perform a single, associated activity. The functions are classified into different types -

1. User-defined functions
2. Built-in functions
3. Lambda functions
4. Recursive functions

### User-defined functions:

Functions that are defined by us in order to perform a particular activity are termed User-defined functions.

**Benefits of User-defined functions**

1. We can use the User-defined functions to decompose a large program into small segments, which makes the program easy to understand, maintain and debug.
2. Suppose repeated code occurs in a program. We can use the function to include those codes and execute when required by calling that function.

### Built-in functions:

Python has different functions that are readily available for use. These functions are known as Built-in functions.

**List of Built-in functions**

1. abs(), delattr(), hash(), memoryview(), set(), all(), dict(), help(), min(), setattr(), any(), dir(), hex(), next(), slice(), ascii(), divmod(), id(), object(), sorted(), bin(), enumerate(), input(), oct(), staticmethod(), bool(), eval(), **int**(), open(), str(), breakpoint(), exec(), isinstance(), ord(), sum(), bytearray(), filter(), issubclass(), pow(),**super**(), bytes(), **float**(), iter(), print(), tuple(), callable(), format(), len(), property(), type(), chr(), frozenset(), list(), range(), vars(), classmethod(), getattr(), locals(), repr(), zip(), compile(), globals(), map(), reversed(), \_\_import\_\_(), complex(), hasattr(), max(), round()

### Lambda functions:

Lambda functions are known as Anonymous functions that are defined without a name.

While we can define normal functions using the def keyword in Python, we can define anonymous functions with the help of the lambda keyword.

**Use of Lambda function in Python:**

To need a nameless function for a short period of time. In Python, we usually use it as a parameter to a higher-order function (a function that accepts other functions as parameters).

Lambda functions are utilized along with built-in functions such as **filter(), map()**, and a lot more.

1. **filter()\:** As the name suggests, it is used to filter the iterables as per the requirements. The **filter()** function filters the original iterable and passes the elements that return TRUE for the function given to the filter.
2. **map()\:** The **map()** function executes all the requirements of a function on the elements in the iterable and allows us to apply a function on it and then passes it to the result, which can have identical as well as different values.

### Recursion functions:

A Recursive function is a function defined in terms of itself through self-referential expressions. This means that the function will continue to call itself and repeat its behavior until some requirements are met to return an output.

## Defining and Using a Module in Python

In the following section, we will understand how we can define and use a module in Python.

In order to begin, we will create a Python program file with a **.py** extension and save it in the local repository. Now we can utilize this program file to import it into the application to involve the functionality of the module in the application.

We can use the import command in order to include multiple modules.

**Syntax:**

1. **import** module\_1, module\_2, ...

#### Note: As soon as we have included the module in the program code, we have executed the code without calling any function. That is because we have defined and called the functions in the file. Therefore, the entire file is executed first treated as a larger scale function, being executed while being called.

Let us consider the following example.

**Example:**

1. # importing a module
2. **import** math

We can delete the calls of the function from the module we created and access the functions attributes, classes, and all the other valuables using the dot (**.**) operator.

**Syntax:**

1. module\_1.func\_1()
2. module\_2.name

Let us consider the following example for the same.

**Example:**

1. # importing a module
2. **import** math
4. # using the sqrt() function of the math module
5. print("Square root of 16:", math.sqrt(16))

**Output:**

Square root of 16: 4.0

**Explanation:**

In the above snippet of code, we have imported the **math** module. We have then used the **sqrt()** function of the **math** module to find the square root of **16** and printed the value for the users.

## Defining and Calling a Function in Python

In the following section, we will understand how we can define and call a function in Python.

Let us consider the following example in order to understand the definition and calling of a function.

**Example:**

1. # defining a function
2. def my\_func():
3. print("Greetings User! Welcome to Javatpoint.")
5. # calling the function
6. my\_func()

**Output:**

Greetings User! Welcome to Javatpoint.

**Explanation:**

In the above snippet of code, we have defined a simple function that prints a string and terminates. The code of this function can also be altered in order to make it quite more useful.

We can pass a parameter to the function and make the above block of code more reusable by not hard coding the string.

Let us consider the following example for the same.

**Example:**

1. # defining a function
2. def my\_func(name):
3. print("Greetings " + name + ", Welcome to Javatpoint.")
5. # calling the function
6. my\_func("Daniel")
7. my\_func("Michelle")
8. my\_func("Chris")
9. my\_func("Dana")

**Output:**

Greetings Daniel, Welcome to Javatpoint.

Greetings Michelle, Welcome to Javatpoint.

Greetings Chris, Welcome to Javatpoint.

Greetings Dana, Welcome to Javatpoint.

**Explanation:**

In the above snippet of code, we have defined a function that accepts a parameter as '**name**' and printed a statement including that parameter. We have then called the function by specifying the value of the **name** parameter and printed different statements from one function.

As we can observe, we do not have to rewrite the function or define it repeatedly for different strings. We can call this function multiple times in the whole application.

## Conclusion

Modules and functions may appear similar to their purpose, which is reusability. However, modules are on a larger scale because of their use in different classes, functions, and attributes to fulfil larger functionalities. At the same time, functions are more particular to specific activities on a smaller scale.

Multiply All Elements in list of Python

In this tutorial, we will learn how we can multiply all the elements of a list in Python.

Let us have a look at some examples to understand our objective-

1. Input - [2, 3, 4]
2. Output - 24

We can observe that in the output we have obtained the product of all the elements present in the list.

1. Input - [3, 'a']
2. Output - aaa

Since the first element is three, a is printed three times in the output.

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

We will learn the following methods-

1. Traversing the list
2. Using NumPy
3. Using lambda

Let's start with the first one,

Traversing the list

Consider the program given below-

1. #creating a function
2. def multiply\_ele(list\_value1):
3. #multiply the elements
4. prod=1
5. **for** i in list\_value1:
6. prod = prod\*i
7. **return** prod
8. #initializing the list
9. list\_value1 = [10, 11, 12, 13, 14]
10. list\_value2 = [2, 3, 4, 5, 6, 7]
11. #displaying the resultant values
12. print("The multiplication of all the elements of list\_value1 is: ", multiply\_ele(list\_value1))
13. print("The multiplication of all the elements of list\_value2 is: ", multiply\_ele(list\_value2))

**Output:**

The multiplication of all the elements of list\_value1 is: 240240

The multiplication of all the elements of list\_value2 is: 5040

**Explanation-**

It's time to have a look at the explanation of the above program-

1. In the first step, we have created a function that will make the list as an input.
2. In the function definition, we have used a for loop that takes each element from the list, multiplies it with one initially, and then prints the resultant value of the product.
3. In the next step, we have initialized the lists and then passed them into our function.
4. On executing this program, the desired output is displayed.

In the second program, we will see how NumPy can help us to implement the same.

Using NumPy

The following program illustrates how it can be done in Python.

1. #importing the NumPy module
2. **import** numpy
3. #initializing the list
4. list\_value1 = [10, 11, 12, 13, 14]
5. list\_value2 = [2, 3, 4, 5, 6, 7]
6. #using numpy.prod()
7. res\_list1 = numpy.prod(list\_value1)
8. res\_list2 = numpy.prod(list\_value2)
9. #displaying the resultant values
10. print("The multiplication of all the elements of list\_value1 is: ", res\_list1)
11. print("The multiplication of all the elements of list\_value2 is: ", res\_list2)

**Output:**

The multiplication of all the elements of list\_value1 is: 240240

The multiplication of all the elements of list\_value2 is: 5040

**Explanation-**

Let's understand what we have done in the above program.

1. In the first step, we have imported the NumPy module.
2. In the next step, we have initialized the values of the two lists, list\_value1 and list\_value2.
3. After this, we will use **prod()** that will compute the product of the elements present in the lists.
4. On executing the program, the expected output is displayed.

Finally, we will learn how lambda can be used to multiply the elements of our list.

Using lambda

The program given below demonstrates the same-

1. #importing the module
2. from functools **import** reduce
3. #initializing the list
4. list\_value1 = [10, 11, 12, 13, 14]
5. list\_value2 = [2, 3, 4, 5, 6, 7]
6. #using numpy.prod()
7. res\_list1 = reduce((lambda a, b:a\*b), list\_value1)
8. res\_list2 = reduce((lambda a, b:a\*b), list\_value2)
9. #displaying the resultant values
10. print("The multiplication of all the elements of list\_value1 is: ", res\_list1)
11. print("The multiplication of all the elements of list\_value2 is: ", res\_list2)

**Output:**

The multiplication of all the elements of list\_value1 is: 240240

The multiplication of all the elements of list\_value2 is: 5040

**Explanation-**

Let us understand what happened in the above program.

1. In the first step, we have imported reduce from
2. After this, we have initialized the two lists, **list\_value1** and **list\_value2**.
3. We have used the precise way of defining functions, which is lambda, and then provided the required functionality.
4. On executing the program, the desired values are displayed.

Conclusion

In this tutorial, we learned the various approaches of multiplying the elements present in the list in Python.

How to Convert float to int in Python

We have used different numeric data types in Python, in this tutorial we will learn how we can convert a float value to an integer value.

Let's have a look at the approaches to implement the same-

1. Using **trunc()**
2. Using **floor()**
3. Using **ceil()**
4. Using **int()**

So, let's get started with the first one-

Using trunc()

The following program shows how we can use **trunc()** to convert a float value to an integer in Python.

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

1. #**import** trunc
2. from math **import** trunc
3. #initialising the values
4. a = 20.33
5. b = 12.46
6. c = 9.54
7. res\_sum = a + b + c
8. #displaying the sum value
9. print("The result of a + b + c is ", res\_sum)
10. #using trunc
11. print("The converted value of a is: ", trunc(a))
12. print("The converted value of b is: ", trunc(b))
13. print("The converted value of c is: ", trunc(c))
14. print("The converted value of sum is: ", trunc(res\_sum))

**Output:**

The result of a + b + c is 42.33

The converted value of a is: 20

The converted value of b is: 12

The converted value of c is: 9

The converted value of sum is: 42

**Explanation-**

It's time to understand what happened in the above program-

1. Since we have to use **trunc(),** we have imported math in the first step.
2. After this, we have initialized three float values and then calculated their sum.
3. Finally, we have passed the variables **a, b, c**, and **res\_sum** in **trunc()** to obtain the integer values.
4. On executing the program, we obtain the desired output.

In the next program, we will make use of floor().

Using floor()

First, let's understand what happens when we pass a float value in floor()?

When a float number is passed in **floor()**, it rounds off the number down to the nearest integer.

Consider the program given below,

1. #**import** floor
2. from math **import** floor
3. #initialising the values
4. a = 20.33
5. b = 12.46
6. c = 9.54
7. res\_sum = a + b + c
8. #displaying the sum value
9. print("The result of a + b + c is ", res\_sum)
10. #using floor
11. print("The converted value of a is: ", floor(a))
12. print("The converted value of b is: ", floor(b))
13. print("The converted value of c is: ", floor(c))
14. print("The converted value of sum is: ", floor(res\_sum))

**Output:**

The result of a + b + c is 42.33

The converted value of a is: 20

The converted value of b is: 12

The converted value of c is: 9

The converted value of sum is: 42

**Explanation-**

Let's have a glance at the explanation of this program.

1. Since we have to use **floor(),** we have imported math in the first step.
2. After this, we have initialized three float values and then calculated their sum.
3. Finally, we have passed the variables **a, b, c,** and **res\_sum** in the **floor()** to obtain the integer values.
4. On executing the program, we obtain the desired output.

Now, we shall see how **ceil()** can be used to do the same.

Using ceil()

First, let's understand what happens when we pass a float value in ceil()?

When a float number is passed in **ceil(),** it rounds off the number up to the nearest integer.

1. #**import** ceil
2. from math **import** ceil
3. #initialising the values
4. a = 20.33
5. b = 12.46
6. c = 9.54
7. res\_sum = a + b + c
8. #displaying the sum value
9. print("The result of a + b + c is ", res\_sum)
10. #using ceil
11. print("The converted value of a is: ", ceil(a))
12. print("The converted value of b is: ", ceil(b))
13. print("The converted value of c is: ", ceil(c))
14. print("The converted value of sum is: ", ceil(res\_sum))

**Output:**

The result of a + b + c is 42.33

The converted value of a is: 21

The converted value of b is: 13

The converted value of c is: 10

The converted value of sum is: 43

**Explanation-**

Let's understand what we have done in this program.

1. Since we have to use **ceil(),** we have imported math in the first step.
2. After this, we have initialized three float values and then calculated their sum.
3. Finally, we have passed the variables **a, b, c,** and **res\_sum** in **ceil()** to obtain the integer values.
4. On executing the program, we obtain the desired output.

Finally, in the last program, we will use the most basic approach of converting a float value to an integer which is using **int()**.

Using int()

The program given below illustrates how it can be used in a program.

1. #initialising the values
2. a = 20.33
3. b = 12.46
4. c = 9.54
5. res\_sum = a + b + c
6. #displaying the sum value
7. print("The result of a + b + c is ", res\_sum)
8. #using **int**()
9. print("The converted value of a is: ", **int**(a))
10. print("The converted value of b is: ", **int**(b))
11. print("The converted value of c is: ", **int**(c))
12. print("The converted value of sum is: ", **int**(res\_sum))

**Output:**

The result of a + b + c is 42.33

The converted value of a is: 20

The converted value of b is: 12

The converted value of c is: 9

The converted value of sum is: 42

**Explanation-**

1. We have initialized three float values and then calculated their sum.
2. Finally, we have passed the variables **a, b, c,** and res\_sum in **int()** to obtain the integer values.
3. On executing the program, we obtain the desired output.

Conclusion

In this tutorial, we learned the interesting methods of converting a float to an integer in Python.

String to List in Python

So far, we have discussed various conversions in Python. In this tutorial, we will learn another one, which is converting a string to a list in Python.

We will use the following methods to meet our objective-

1. Using **split()**
2. Using **split()** with a separator
3. Using **strip()**
4. Using **map()**

Let us discuss each one of them.

In the first program, we will make use of **split()** to convert the string to a list in Python.

# The sqrt(): Math Function of Python

The **sqrt()** function is a built-in function in Python for performing operations related to math. The **sqrt()** function is used for returning the square root of any imported number.

In this tutorial, we will discuss how we can use the **sqrt()** function in Python.

### Syntax:

1. math.sqrt(N)

### Parameter:

'N' is any number which is greater or equal to 0 (N >= 0).

### Returns:

It will return the square root of the number "N", that is passed as the parameter.

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

### Example:

1. # importing the math module
2. **import** math as m
4. # printing the square root of 7
5. print ("The square root of 7: ", m.sqrt(7))
7. # printing the square root of 49
8. print ("The square root of 49: ", m.sqrt(49))
10. # printing the square root of 115
11. print ("The square root of 115: ", m.sqrt(115))

**Output:**

The square root of 7: 2.6457513110645907

The square root of 49: 7.0

The square root of 115: 10.723805294763608

### Error

We get an error while executing the **sqrt()** function if the number passed as a parameter is less than "0" due to a runtime error.

**Example:**

1. # **import** the math module
2. **import** math as m
4. print ("The square root of 16: ", m.sqrt(16))
6. # printing the error when x less than 0
7. print ("The square root of -16: ", m.sqrt(-16))

**Output:**

The square root of 16: 4.0

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-3-79d85b7d1d62> in <module>

5

6 # printing the error when x less than 0

----> 7 print ("The square root of -16: ", m.sqrt(-16))

8

ValueError: math domain error

**Explanation:**

As at first, we passed "16" as a parameter, and we got the output as "4", but for "-16", which is less than "0", we got an error, saying "math domain error".

## Application

We can also use the **sqrt()** function for creating an application for performing math functions. Suppose we have a number "N", of which we have to check whether it is a prime number or not.

### Approach:

We will use the following approach:

* **Step1:** we will import the math module.
* **Step 2:** we will create a function "check\_if" for checking if the given number is prime or not.
* **Step 3:** we will check if the number is equal to 1. If "yes", it will return false.
* **Step 4:** we will run a for loop from range "2" to "sqrt(N)".
* **Step 5:** we will check if any number between the given range divides "N".

**Example:**

1. **import** math as M
3. # Creating a function **for** checking **if** the number is prime or not
4. def check\_if(N):
5. **if** N == 1:
6. **return** False
8. # Checking from 1 to sqrt(N)
9. **for** K in range(2, (**int**)(M.sqrt(N)) + 1):
10. **if** N % K == 0:
11. **return** False
12. **return** True
14. # driver code
15. N = **int**( input("Enter the number you want to check: "))
16. **if** check\_if(N):
17. print ("The number is prime")
18. **else**:
19. print ("The number is not prime")

**Output:**

**#1:**

Enter the number you want to check: 12

The number is not prime

**#2:**

Enter the number you want to check: 11

The number is prime

**#3:**

Enter the number you want to check: 53

The number is prime

## Conclusion

In this tutorial, we have discussed how to use Python's sqrt() function for calculating the square root of any number greater than 0.

Using split()

The program given below illustrates how it can be done.

1. # Initialising the string values
2. str\_val1 = "Let us study programming."
3. str\_val2 = "But before that it is essential to have a basic knowledge of computers."
4. str\_val3 = "So first study what is IPO cycle."
5. str\_val4 = "Then learn about the generation of computers."
6. #using split()
7. print(str\_val1.split())
8. print(str\_val2.split())
9. print(str\_val3.split())
10. print(str\_val4.split())

**Output:**

['Let', 'us', 'study', 'programming.']

['But', 'before', 'that', 'it', 'is', 'essential', 'to', 'have', 'a', 'basic', 'knowledge', 'of', 'computers.']

['So', 'first', 'study', 'what', 'is', 'IPO', 'cycle.']

['Then', 'learn', 'about', 'the', 'generation', 'of', 'computers.']

**Explanation-**

1. In the first step, we have initialized the four strings that we would like to convert.
2. After this, we have used the split() so that we obtain a list in which each word of a string represents an element of the list.

In the second program, we have specified a separator in split().

Using split() with a Separator

Consider the given program,

1. # Initializing the string values
2. str\_val1="Let @ us @ study @ programming."
3. str\_val2="But # before # that # it # is # essential # to # have # a # basic # knowledge # of # computers."
4. str\_val3="So $ first $ study $ what $ is $ IPO $ cycle."
5. str\_val4="Then % learn % about % the % generation % of % computers."
6. # Using split()
7. print(str\_val1.split("@"))
8. print(str\_val2.split("#"))
9. print(str\_val3.split("$"))
10. print(str\_val4.split("%"))

**Output:**

['Let ', ' us ', ' study ', ' programming.']

['But ', ' before ', ' that ', ' it ', ' is ', ' essential ', ' to ', ' have ', ' a ', ' basic ', ' knowledge ', ' of ', ' computers.']

['So ', ' first ', ' study ', ' what ', ' is ', ' IPO ', ' cycle.']

['Then ', ' learn ', ' about ', ' the ', ' generation ', ' of ', ' computers.']

**Explanation-**

The approach is similar to the previous program, the only difference it takes an element in the list whenever a separator occurs.

In this program, the separators in the strings were @, #, $ & %.

Now, let's see how to strip() can be used.

Using strip()

The following program illustrates the same-

1. # Initialising the string values
2. str\_val1 = "Let us study programming."
3. str\_val2 = "But before that it is essential to have a basic knowledge of computers."
4. # Using list()
5. print(list(str\_val1.strip()))
6. print(list(str\_val2.strip()))

**Output:**

['L', 'e', 't', ' ', 'u', 's', ' ', 's', 't', 'u', 'd', 'y', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g', '.']

['B', 'u', 't', ' ', 'b', 'e', 'f', 'o', 'r', 'e', ' ', 't', 'h', 'a', 't', ' ', 'i', 't', ' ', 'i', 's', ' ', 'e', 's', 's', 'e', 'n', 't', 'i', 'a', 'l', ' ', 't', 'o', ' ', 'h', 'a', 'v', 'e', ' ', 'a', ' ', 'b', 'a', 's', 'i', 'c', ' ', 'k', 'n', 'o', 'w', 'l', 'e', 'd', 'g', 'e', ' ', 'o', 'f', ' ', 'c', 'o', 'm', 'p', 'u', 't', 'e', 'r', 's', '.']

**Explanation-**

1. In the first step, we have initialized the two strings that we would like to convert.
2. After this, we have used the strip() so that we obtain a list, where each character represents of the string represents an element in the list.

Convert Strings to List of Lists using map()

1. # Initializing the string values
2. str\_val1="Let us study programming."
3. str\_val2="But before that it is essential to have a basic knowledge of computers."
4. #using split()
5. str\_val1 = str\_val1.split()
6. str\_val2 = str\_val2.split()
7. list\_str1 = list(map(list,str\_val1))
8. list\_str2 = list(map(list,str\_val2))
9. #displaying the list values
10. print(list\_str1)
11. print(list\_str2)

**Output:**

[['L', 'e', 't'], ['u', 's'], ['s', 't', 'u', 'd', 'y'], ['p', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g', '.']]

[['B', 'u', 't'], ['b', 'e', 'f', 'o', 'r', 'e'], ['t', 'h', 'a', 't'], ['i', 't'], ['i', 's'], ['e', 's', 's', 'e', 'n', 't', 'i', 'a', 'l'], ['t', 'o'], ['h', 'a', 'v', 'e'], ['a'], ['b', 'a', 's', 'i', 'c'], ['k', 'n', 'o', 'w', 'l', 'e', 'd', 'g', 'e'], ['o', 'f'], ['c', 'o', 'm', 'p', 'u', 't', 'e', 'r', 's', '.']]

**Explanation-**

1. In the first step, we have initialized the two strings that we would like to convert.
2. After this, we have used the split() method followed by map() so that we map the list functionality to each element of the string.
3. On executing the given program, the desired output is displayed.

Finally, in the last program we have used the string of integers,

Converting String of Integers

Consider the program given below,

1. #initialising the string values
2. str\_val1 = "1 2 3 4 5 6 7 8 9"
3. str\_val2 = "12 21 32 44 54 76 83"
4. #using split()
5. str\_val1 = str\_val1.split()
6. str\_val2 = str\_val2.split()
7. list\_str1 = list(map(**int**,str\_val1))
8. list\_str2 = list(map(**int**,str\_val2))
9. #displaying the list values
10. print(list\_str1)
11. print(list\_str2)

**Output:**

[1, 2, 3, 4, 5, 6, 7, 8, 9]

[12, 21, 32, 44, 54, 76, 83]

**Explanation-**

The logic is similar to the above program but here we have passed a string of integers and applied the 'int' functionality on all the elements.

Conclusion

In this tutorial, we learned the simple techniques of converting a string to a list in Python.

Shuffle in Python

**Shuffling** refers to the rearranging of elements in a random order which means no specific sequence is considered while arranging the elements.

In this tutorial, we will learn how we can shuffle the elements of a list using Python.

The different approaches that we will use to shuffle the elements are as follows-

1. Using Fisher-Yates shuffle algorithm
2. Using **shuffle()**
3. Using **sample()**
4. Random selection of elements and then appending them in a list

We will discuss each method in detail.

So, let's begin with the first one,

Using Fisher-Yates Shuffle Algorithm

1. # using Fisher-Yates shuffle Algorithm to shuffle a list
2. **import** random
3. # initializing the list
4. list\_values1 = [11,20,19,43,22,10]
5. # Printing the list
6. print ("The initialized list is: ",(list\_values1))
7. **for** i in range(len(list\_values1)-1, 0, -1):
9. # choosing a random index from 0 to i
10. k = random.randint(0, i + 1)
11. list\_values1[i], list\_values1[k] = list\_values1[k], list\_values1[i]
13. # Printing the shuffled list
14. print ("The shuffled list is : ",(list\_values1))

**Output:**

The initialized list is : [11, 20, 19, 43, 22, 10]

The shuffled list is : [11, 43, 20, 19, 10, 22]

**Explanation**

Let's understand what we have done in the above program.

1. In the first step, we have imported the random module.
2. After this, we have an initialized list that contains different numeric values.
3. In the next step, we used the for loop and then randomly selected an element and then swapped it with the element at random index.
4. Finally, we have displayed the shuffled list in the output.

Using shuffle()

In the second method, we will see how **shuffle()** can be used to shuffle the elements of our list.

Consider the program given below-

1. **import** random
2. # initializing the list
3. list\_values1 = [11,20,19,43,22,10]
4. # Printing the list
5. print ("The initialized list is : ",(list\_values1))
6. #using shuffle()
7. random.shuffle(list\_values1)
8. # Printing the shuffled list
9. print ("The shuffled list is : ",(list\_values1))

**Output:**

The initialized list is : [11, 20, 19, 43, 22, 10]

The shuffled list is : [22, 10, 20, 11, 19, 43]

**Explanation**

Let's understand what we have done in the above program,

1. In the first step, we have imported the random module.
2. After this, we have an initialized list that contains different numeric values.
3. In the next step, we used the **shuffle()** and passed 'list\_values1' as a parameter.
4. Finally, we have displayed the shuffled list in the output.

Using random.sample()

In the third approach, we will use **random.sample()** to do the same.

The following program illustrates how it can be done-

1. **import** random
2. # initializing the list
3. list\_values1 = [11,20,19,43,22,10]
4. # Printing the list
5. print ("The initialized list is : ",(list\_values1))
6. #using random.sample()
7. res\_list=random.sample(list\_values1,len(list\_values1))
8. # Printing the shuffled list
9. print ("The shuffled list is : ",(res\_list))

**Output:**

The initialized list is : [11, 20, 19, 43, 22, 10]

The shuffled list is : [43, 20, 19, 11, 10, 22]

**Explanation**

It's time to understand the above program-

1. In the first step, we have imported the random module.
2. After this, we have an initialized list that contains different numeric values.
3. In the next step, we used the **sample()** and passed 'list\_values1' and the length of the list as its parameters.
4. Finally, we have displayed the shuffled list in the output.

Finally, it's time to discuss the last approach, it's indeed an interesting one, let's see how….

Random Selection of Elements and then Appending them in a list

1. **import** random
2. # initializing the list
3. list\_values1 = [11,20,19,43,22,10]
4. # Printing the list
5. print ("The initialized list is: ",(list\_values1))
6. len\_list=len(list\_values1)
7. **for** i in range(len\_list):
8. k=random.randint(0,len\_list-1)
9. e=list\_values1.pop(k)
10. list\_values1.append(e)
11. # Printing the shuffled list
12. print ("The shuffled list is : ",(list\_values1))

**Output:**

The initialized list is : [11, 20, 19, 43, 22, 10]

The shuffled list is : [19, 22, 20, 43, 10, 11]

**Explanation**

Now, let's see the explanation of this code,

1. In the first step, we have imported the random module.
2. After this, we have an initialized list that contains different numeric values and then calculated the length of the list.
3. In the next step, we used the **sample()** and passed 'list\_values1' and the length of the list as its parameters.
4. Now we have used for loop that randomly selects an element, removes it, and then appends it to the list.
5. Finally, we have displayed the shuffled list in the output.

Conclusion

In this tutorial, we learned the different methods of shuffling the elements of a list in Python.

Remove Multiple Characters from a String in Python

We already know that strings are defined as a sequence of characters, and we can perform a variety of operations on them.

In this tutorial, we will learn one more interesting task that can be accomplished using strings in Python.

Here we will see how we can remove multiple characters from it.

We have listed below the methods that we will learn to meet our objective.

1. Using **nested replace()**
2. Using **translate()** & **maketrans()**
3. Using **subn()**
4. Using **sub()**

Using nested replace()

In the program given below, we will see how **replace()** is used to remove multiple characters from the string.

1. #initializing the string
2. string\_val = 'learnpythonlearn'
3. #displaying the string value
4. print("The initialized string is ", string\_val)
5. #using nested replace()
6. res\_str = string\_val.replace('l', '%temp%').replace('l', 'e').replace('%temp%', 'e')
7. #printing the resultant string
8. print ("The string after replacing the characters is ", res\_str)

**Output:**

The initialized string is learnpythonlearn

The string after replacing the characters is eearnpythoneearn

**Explanation-**

1. In the first step, we have initialized the string whose characters we would like to replace.
2. After this, we have displayed the original string so that we can easily understand the difference between this and the expected output.
3. Now we have used **replace()** and specified the characters that we wish to remove or change.
4. On executing the program, the desired output is displayed.

In the second program, we will see how **translate()** and **maketrans()** can be used to do the same. The users must keep this thing in their mind that it only works in Python 2.

Using translate() and maketrans()

The following program shows how it can be done.

1. **import** string
2. #initializing the string
3. string\_val='learnpythonlearn'
4. #displaying the string value
5. print("The initialized string is ",string\_val)
6. #using translate() & maketrans()
7. res\_str=string\_val.translate(string.maketrans('le','el'))
8. #printing the resultant string
9. print("The string after replacing the characters is ",res\_str)

**Output:**

The initialized string is learnpythonlearn

The string after replacing the characters is eearnpythoneearn

**Explanation-**

1. In the first step, we have initialized the string whose characters we would like to replace.
2. After this, we have displayed the original string so that we can easily understand the difference between this and the expected output.
3. Now we have used **replace()** and specified the characters that we wish to remove or change.
4. On executing the program, the desired output is displayed.

Now we will discuss how **re.subn()** can become an aid for this. The **subn()** returns a new string with the total number of replacements.

Using re.subn()

The program given below shows how it can be done.

1. #importing the re module
2. **import** re
3. #initializing the string value
4. string\_val = "To get the result 100, we can multiply 10 by 10"
5. #defining the function
6. def pass\_str(string\_val):
7. string\_val, n = re.subn('[0-9]', 'A', string\_val)
8. print (string\_val)
9. #displaying the resultant value
10. pass\_str(string\_val)

**Output:**

To get the result AAA, we can multiply AA by AA

**Explanation-**

1. In the first step, we have imported the re module that will help us to use the required functions.
2. After this, we have initialized the string whose characters we would like to replace or remove.
3. The next step is to define a function that takes the string value as its parameter.
4. In the function definition, we have used **subn()** that takes three parameters. The first parameter is the pattern that we wish to replace, the second is with what element or number we want to replace it and the third is the string.
5. Finally, the print statement at the end shows that the processed string is displayed.
6. We have passed this string at the end so that we get the expected output.

In the last program, we will do the same thing using **sub()**

Using re.sub()

The following program illustrates how it can be done-

1. #importing the re module
2. **import** re
3. #initializing the string value
4. string\_val = "To get the result 100, we can multiply 10 by 10"
5. #defining the function
6. def pass\_str(string\_val):
7. string\_val = re.sub('[0-9]', 'Z', string\_val)
8. print(string\_val)
9. #displaying the resultant value
10. pass\_str(string\_val)

**Output:**

To get the result ZZZ, we can multiply ZZ by ZZ

**Explanation-**

1. In the first step, we have imported the re module that will help us to use the required functions.
2. After this, we have initialized the string whose characters we would like to replace or remove.
3. The next step is to define a function that takes the string value as its parameter.
4. In the function definition, we have used **sub()** that takes three parameters. The first parameter is the pattern that we wish to replace, the second is with what element or number we want to replace it and the third is the string.
5. Finally, the print statement at the end shows that the processed string is displayed.
6. We have passed this string at the end so that we get the expected output.

Conclusion

In this tutorial, we learned how we can remove multiple characters from a string using Python.

How to Remove Duplicates from a list in Python

When an element occurs more than once in a list, we refer to it as a duplicate.

In this tutorial, we will learn different methods of removing these duplicates from a list in Python.

1. The Basic Approach
2. Using List Comprehension
3. Using **Set()**
4. Using **enumerate()**
5. Using **OrderedDict**

Let us discuss each one of them in detail.

The Basic Approach

In the first method, we will discuss the basic approach of removing duplicates from the list using Python.

1. #initializing the list
2. list\_value1=[12,15,11,12,8,15,3,3]
3. print("The initialized list is ",list\_value1)
4. res\_list=[]
5. **for** i in list\_value1:
6. **if** i not in res\_list:
7. res\_list.append(i)
8. #printing the list after removing duplicate elements
9. print("The resultant list after removing duplicates is ",res\_list)

**Output:**

The initialized list is [12, 15, 11, 12, 8, 15, 3, 3]

The resultant list after removing duplicates is [12, 15, 11, 8, 3]

**Explanation**

So, now it's time to have a glance at the explanation of the above program.

1. The first step is to initialize the list that contains duplicate values.
2. After this, we have printed this list so that it becomes easy for us to compare the lists in the output.
3. In the next step, we have declared an empty list that will hold all the unique values and then used for loop followed by decision making to check whether a particular element exists in the list or not, if it doesn't exist, we will append it in the list.
4. Finally, we have displayed the resultant list.

In the second program, we will make use of list comprehension to meet our objective.

Using List Comprehension

The following program illustrates the same-

1. #initializing the list
2. list\_value1=[12,15,11,12,8,15,3,3]
3. print("The initialized list is ",list\_value1)
4. res\_list=[]
5. #using list comprehension
6. [res\_list.append(i) **for** i in list\_value1 **if** i not in res\_list]
7. #printing the list after removing duplicate elements
8. print("The resultant list after removing duplicates is ",res\_list)

**Output:**

The initialized list is [12, 15, 11, 12, 8, 15, 3, 3]

The resultant list after removing duplicates is [12, 15, 11, 8, 3]

**Explanation**

Let's understand what we have done in the above program-

1. The first step is to initialize the list that contains duplicate values.
2. After this, we have printed this list so that it becomes easy for us to compare the lists in the output.
3. In the next step, we have used list comprehension to build the same logic.
4. Finally, we have displayed the resultant list.

In the third approach, we will see how **set()** can be used to obtain a list of distinct elements.

Using Set()

The program given below shows how it can be done-

1. #initializing the list
2. list\_value1 = [12,15,11,12,8,15,3,3]
3. print("The initialized list is ",list\_value1)
4. res\_list = []
5. #using set()
6. list\_value1 = list(set(list\_value1))
7. #printing the list after removing duplicate elements
8. print("The resultant list after removing duplicates is ",list\_value1)

**Output:**

The initialized list is [12, 15, 11, 12, 8, 15, 3, 3]

The resultant list after removing duplicates is [12, 15, 11, 8, 3]

**Explanation**

Let's try and understand what happened here,

1. The first step is to initialize the list that contains duplicate values.
2. After this, we have printed this list so that it becomes easy for us to compare the lists in the output.
3. In the next step, we have used **set()** inside **list()** and passed list\_value1 as a parameter so that we can obtain a list of distinct elements.
4. Finally, we have displayed the resultant list.

Using enumerate()

Now let's see how **enumerate()** can be used to meet our objective.

The following program illustrates the same-

1. #initializing the list
2. list\_value1 = [12,15,11,12,8,15,3,3]
3. print("The initialized list is ",list\_value1)
4. res\_list = [x **for** n,x in enumerate(list\_value1) **if** x not in list\_value1[:n]]
5. #printing the list after removing duplicate elements
6. print("The resultant list after removing duplicates is ",res\_list)

**Output:**

The initialized list is [12, 15, 11, 12, 8, 15, 3, 3]

The resultant list after removing duplicates is [12, 15, 11, 8, 3]

**Explanation**

1. The first step is to initialize the list that contains duplicate values.
2. After this, we have printed this list so that it becomes easy for us to compare the lists in the output.
3. In the next step, we have used **enumerate()** inside list comprehension that will return a list containing unique elements.
4. Finally, we have displayed the resultant list.

Finally, in the last program we will see how OrderedDict can be used to remove duplicates from list using Python.

Using OrderedDict

The program given below shows how it can be done-

1. from collections **import** OrderedDict
2. #initializing the list
3. list\_value1 = [12,15,11,12,8,15,3,3]
4. print("The initialized list is ",list\_value1)
5. #using OrderedDict
6. res\_list = list(OrderedDict.fromkeys(list\_value1))
7. #printing the list after removing duplicate elements
8. print("The resultant list after removing duplicates is ",res\_list)

**Output:**

The initialized list is [12, 15, 11, 12, 8, 15, 3, 3]

The resultant list after removing duplicates is [12, 15, 11, 8, 3]

**Explanation**

1. The first step is to import OrderedDict from collections so that we can use it to remove duplicates from the list.
2. In the next step, we have initialized the list that contains duplicate values.
3. After this, we have printed this list so that it becomes easy for us to compare the lists in the output.
4. Now we have passed list\_values1 inside OrderedDict.fromkeys().
5. Finally, we have displayed the resultant list that does not contain any duplicate elements.

Conclusion

In this tutorial, we explored and learned some interesting methods to remove duplicates from a list in Python.

# Difference between Append, Extend and Insert in Python

**List:** are like an array of dynamic size, declared in another programming language such as vector in C++ or Arraylist in Java. It is not necessary for a list to be homogeneous, and this is the main reason that makes it the most powerful tool in Python. A single list can contain different data types such as strings, integers, and objects.

As we know, lists are mutable, so they can be altered even after creating them. Lists has several methods, among which **append()**, **insert()**, **extend()** are most common ones.

In this tutorial, we will learn how append(), expend(), and insert() functions are different from each other in Python's lists.

### Append() Function

The append() function is used for adding an element at the end of the list. The argument we pass in the append() function is added as a single element at the end of the list, and the length of the list will increase by 1.

**Syntax**

1. list\_name1.append(element\_1)

The "element\_1" can be an integer, tuple, string or another list.

**Example:**

1. list\_1 = ['The', 'list\_1']
3. # Using the method
4. list\_1.append('is')
5. list\_1.append('an')
6. list\_1.append('example')
8. # Displaying the list
9. print ("The added elements in the given list are: ", list\_1)

**Output:**

The added elements in the given list are: ['The', 'list\_1', 'is', 'an', 'example']

### Insert() Function

The **insert()** function is used for inserting the value at any desired position in the list. We have to pass two arguments in it; the first is for index where we want to inter the element, and the second is for element to be inserted.

**Syntax**

1. list\_name1.insert(index, element\_1)

The "element\_1" can be an integer, tuple, string, or object.

**Example:**

1. list\_1 = ['The', 'is']
3. # Using the method
4. list\_1.insert(3,'an')
5. list\_1.insert(1, 'list\_1')
6. list\_1.insert(4, 'example')
8. # Displaying the list
9. print ("The inserted elements in the given list are: ", list\_1)

**Output:**

The inserted elements in the given list are: ['The', 'list\_1', 'is', 'an', 'example']

### Extend() Function

The **extend()** function is used for appending each element of the iterable (list, string, or tuple) to the end of the list. This will increase the length of the list by the number of elements of the iterable are passed as an argument.

**Syntax**

1. list\_name1.extend(iterable)

**Example:**

1. list\_1 = ['The', 'list\_1']
3. # Using the method
4. list\_1.extend(['is', 'an', 'example'])
5. list\_1.extend('javatpoint')
7. # Displaying the list
8. print ("The extended elements in the given list are: ", list\_1)

**Output:**

The extended elements in the given list are: ['The', 'list\_1', 'is', 'an', 'example', 'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't']

## Difference Between Append(), Insert() and Extend()

|  |  |  |
| --- | --- | --- |
| **append() Function** | **insert() Function** | **extend() Function** |
| The element passed in the argument is added at the end of the list. | The element passed in the argument is added at the mentioned index of the list. | Each element of the iterable is passed as ab argument is added at the end of the list. |
| The element passed as an argument will be added as a single element without any changes. | The element passed in the argument will be added as a single element at the desired location without any changes. | The iterable passed as an argument will append each of its elements at the end of the list. |
| The length of the list will increase by 1. | The length of the list will increase by 1. | The length of the list will increase by the number of elements in the iterable. |
| The append() function has a constant time complexity of O(1). | The insert() function has linear complexity of O(n). | The extend() function has a time complexity of O(k), where "k" is the length of the iterable. |

**Let's compare all the three methods in a single program:**

1. list\_name1 = ['this', 'is', 'LIST\_1']
2. list\_name2 = ['this', 'is', 'of', 'LIST\_2']
3. list\_name3 = ['this', 'is', 'LIST\_3']
5. S = ['Example\_1', 'Example\_2']
7. # Using methods
8. list\_name1.append(S)
9. list\_name2.insert(2, S)
10. list\_name3.extend(S)
12. # Displaying lists
13. print ("The appended elements in the given list are: ", list\_name1)
14. print ("The inserted elements in the given list are: ", list\_name2)
15. print ("The extended elements in the given list are: ", list\_name3)

**Output:**

The appended elements in the given list are: ['this', 'is', 'LIST\_1', ['Example\_1', 'Example\_2']]

The inserted elements in the given list are: ['this', 'is', ['Example\_1', 'Example\_2'], 'of', 'LIST\_2']

The extended elements in the given list are: ['this', 'is', 'LIST\_3', 'Example\_1', 'Example\_2']

## Conclusion

In this tutorial, we discussed different methods that can be used for modifying lists in Python. We have also explained the difference between append(), insert(), and extend() functions in the list of Python.

Average of list in Python

In this tutorial, we will discuss how we can compute the average of the list in Python.

The average of a list is defined as the sum of elements present in the list divided by the number of elements present in the list.

Here, we will make use of three different approaches to calculate the average of the elements present in the list using Python.

1. Using **sum()**
2. Using **reduce()**
3. Using **mean()**

So, let's get started…

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

Using sum()

In the first method, we have used the sum() and len() to calculate the average.

The following program illustrates the same-

1. # Python program to get average of a list
2. def calc\_average(lst):
3. **return** sum(lst) / len(lst)
4. lst = [24, 19, 35, 46, 75, 29, 30, 18]
5. average = calc\_average(lst)
6. # Printing the average value of the list
7. print("The average of the list is ", round(average, 3))

**Output:**

The average of the list is 34.5

**Explanation-**

It's time to have a look at what we have done in the above program-

1. In the first step, we have created a function that takes a list as its parameter and then returns the average using **sum()** and **len().** We know that **sum()** is used to calculate the sum of elements and len() tells us about the length of the list.
2. After this, we have initialized the list whose average, we would like to calculate.
3. In the next step, we have passed this list as a parameter in our function.
4. Finally, we printed the resultant value.

In the next program, we will see how **reduce()** can help us to do the same.

Using reduce()

The program given below shows how it can be done-

1. # Python program to obtain the average of a list
2. # Using reduce() and lambda
3. from functools **import** reduce
4. def calc\_average(lst):
5. **return** reduce(lambda a, b: a + b, lst) / len(lst)
6. #initializing the list
7. lst = [24, 19, 35, 46, 75, 29, 30, 18]
8. average = calc\_average(lst)
9. # Printing average of the list
10. print("The Average of the list is ", round(average, 2))

**Output:**

The average of the list is 34.5

**Explanation**

Let's understand what we have done here-

1. In the first step, we have imported reduce from functools so that we can use it in our program to compute the average of elements.
2. Now, we have created a function calc\_average that takes a list as its parameter and uses lambda(a precise way of writing functions in python) inside reduce to calculate the average.
3. After this we have initialized whose average, we would like to calculate.
4. In the next step, we have passed this list as a parameter in our function.
5. Finally, we printed the resultant value.

In the last program, we will learn how **mean()** can be used to calculate the average of list

Using mean()

The following program shows how it can be done-

1. # Python program to obtain the average of a list
2. # Using mean()
3. from statistics **import** mean
4. def calc\_average(lst):
5. **return** mean(lst)
6. lst = [24, 19, 35, 46, 75, 29, 30, 18]
7. average = calc\_average(lst)
8. # Printing the average of the list
9. print("The average of the list is ", round(average, 2))

**Output:**

The average of the list is 34.5

**Explanation-**

It's time to have a look at what we have done in the above program-

1. In the first step, we have imported mean from statistics so that we can use it in our program to compute the average of elements.
2. Now, we have created a function calc\_average that takes a list as its parameter and uses **mean()** to calculate the average.
3. After this we have initialized whose average, we would like to calculate.
4. In the next step, we have passed this list as a parameter in our function.
5. Finally, we printed the resultant value.

Conclusion

In this tutorial, we learned the different methods of calculating the average of elements present in the list using Python.

Difference between Modules and Packages in Python

It is often that many coders and amateur programmers may confuse between a module and a package. The problem generally arises when it becomes hard to identify when and where a module or a package should be implemented.

In the following tutorial, we will discuss a clear set of **differences in modules and packages in the Python programming language** that will make it easy for the programmers to work more professionally while dealing with both modules and packages.

Understanding the Modules in Python

A **module** is a pythonic statement containing different functions. Modules act as a pre-defined library in the script, which is accessible to both the programmers as well as the user.

The python modules also store pre-defined functions from the library when the code is being executed.

Let us consider an example demonstrating the use of a module in Python

**Example:**

1. # importing the library and module
2. import math
3. from math import pow
4. # using the pow() function
5. pow(3, 5)
6. # printing pow()
7. print(pow)

**Output:**

<built-in function pow>

**Explanation:**

In the above snippet of code, we have imported the required module and used the **pow()** function to calculate the powers of the given number as arguments. We have then printed the value of the **pow** for the user.

Understanding the Packages in Python

A **package** is considered a collection of tools that allows the programmers to initiate the code. A Python package acts as a user-variable interface for any source code. This feature allows a Python package to work at a defined time for any functional script in the runtime.

Let us consider the following example demonstrating the package in Python.

**Example:**

1. # importing the package
2. import math
3. # printing a statement
4. print("We have imported the math package")

**Output:**

We have imported the math package

**Explanation:**

In the above snippet of code, we have imported the **math** package that consists of various modules and functions for the programmers and printed a statement for the users.

Understanding the differences between Python Modules and Packages

1. A Package consists of the **\_\_init\_\_.py** file for each user-oriented script. However, the same does not apply to the modules in runtime for any script specified to the users.
2. A module is a file that contains a Python script in runtime for the code specified to the users. A package also modifies the user interpreted code in such a manner that it gets easily operated in the runtime.

A python "module" contains a unit namespace, with the variables that are extracted locally along with some parsed functions like:

1. Constants and Variables
2. Any old or new value
3. Class definitions of properties
4. A module generally corresponds to a single file
5. A debugging tool in the user interface library.

There are some usually utilized tools that allow the programmers to build a new platform with the help of the modules for better code executions. This installs and distributes packages throughout the library in runtime as well.

It becomes easier for us to employ the user-specific tools with the help of a well-structured and standard layout for the package.

# How to Remove Decimal in Python

We have worked with different kind of numbers in Python and modified their type as per our need.

In this tutorial, we will discuss how we can remove decimal in Python.

Let's start with a simple program,

1. a = 24
2. print(type(a))
3. b = 19.4
4. print(type(b))
5. c = 3+4j
6. print(type(c))

**Output:**

<class 'int'>

<class 'float'>

<class 'complex'>

**Explanation:**

In the above program, we have declared a,b and c as 24, 19.4, and 3+4j respectively.

On checking their type, we came to know a belongs to class 'int', b belongs to class 'float' and c belongs to class 'complex'.

Here we have to work on the float numbers, so let's list out the different methods of removing decimals from numbers.

1. Using trunc() function
2. Using int()
3. Using split()

Let's discuss each one of them in detail-

### Using trunc() Function

In the first program, we will make use of **trunc()** function and remove the decimal present in the numbers.

The following program illustrates the same-

1. **import** math
2. num\_value1 = math.trunc(523.771)
3. print (num\_value1)
4. print (type(num\_value1))
5. num\_value2 = math.trunc(21.67)
6. print (num\_value2)
7. print (type(num\_value2))
8. num\_value3 = math.trunc(182.33)
9. print (num\_value3)
10. print (type(num\_value3))
11. num\_value4 = math.trunc(211.54)
12. print (num\_value4)
13. print (type(num\_value4))
14. num\_value5 = math.trunc(19.1)
15. print (num\_value5)
16. print (type(num\_value5))

**Output:**

523

<class 'int'>

21

<class 'int'>

182

<class 'int'>

211

<class 'int'>

19

<class 'int'>

**Explanation:**

Let's have a look at the explanation of the above program-

1. Since we have to use the **trunc()** function, we have imported the math module.
2. We have provided five different decimal values to five variables and checked their type after they are passed in the trunc() function.
3. On executing the program, it displays the required output.

### Using int()

It's time to know the second approach which is removing decimal using int().

The program given below shows how it can be done-

1. num\_value1 = 523.771
2. num\_value2 = 21.67
3. num\_value3 = 182.33
4. print (type(num\_value1))
5. print (type(num\_value2))
6. print (type(num\_value3))
7. new\_value1 = **int**(num\_value1)
8. new\_value2 = **int**(num\_value2)
9. new\_value3 = **int**(num\_value3)
10. print (new\_value1)
11. print (new\_value2)
12. print (new\_value3)
13. print (type(new\_value1))

**Output:**

<class 'float'>

<class 'float'>

<class 'float'>

523

21

182

<class 'int'>

**Explanation:**

Let's understand what we have done here-

1. In the first step, we have provided float values to three variables and checked their type.
2. After this, we have passed each variable to **int()** and stored them to a new variable.
3. Finally, we have printed the values stored in these variables.
4. On executing this program, the expected output is displayed.

### Using split()

Finally, in the last approach, we will use the interesting split() to obtain the integer values.

The following program illustrates the same-

1. num\_values=[523.771,21.67,182.33,211.54,19.1]
2. sp\_lst = []
3. **for** ele in num\_values:
4. sp\_lst.append(str(ele).split('.')[0])
5. res\_list = [**int**(i) **for** i in sp\_lst]
6. print("The resultant list is: ",res\_list)

**Output:**

The resultant list is: [523, 21, 182, 211, 19]

**Explanation:**

Let's have a look at the explanation of the above program-

1. In the first step, we have created a list that contains all the decimal values.
2. After this, we have declared an empty list and appended the values in it.
3. In the next step, we have taken each element from that list and passed it into an int().
4. Finally, we have displayed the resultant list that contains the numbers without decimal.

## Conclusion

In this Tutorial, we started with a general idea of the type of numbers we use in Python and then learned the various methods of removing decimals from the numbers.

# Fabs in Python

The fabs method in Python is used to return the absolute value of a number.

It can be used by importing the **math** module.

The **math** module in Python can be used to implement different basic mathematical operations like addition, subtraction, division, and multiplication.

It can also be used to perform calculations involving exponents, logarithms, and trigonometry.

### Syntax

The syntax for using fabs in Python is-

1. math.fabs(x)

It's time to have a look at some programs for a better understanding.

### Program 1:

In the first program, we will discuss the basics of the fabs method.

1. #working of fabs()
2. **import** math
3. # initializing the variables a, b, and c
4. a = -20.3
5. b = -30.66
6. c = 19.13
7. #printing the values
8. print ("The fabs of a is: ", math.fabs(a))
9. print ("The fabs of b is: ", math.fabs(b))
10. print ("The fabs of c is: ", math.fabs(c))

**Output:**

The fabs of a is: 20.3

The fabs of b is: 30.66

The fabs of c is: 19.13

**Explanation:**

Let's have a look at the explanation of this program-

1. Since we have to use **fabs()**, we have imported the math module.
2. After this, we have initialized the variables a, b, and c with positive and negative values.
3. In the next step, we have passed these values to **fabs()** and printed them.
4. On executing the program, it displays the desired output.

### Program 2:

In the next program, we will perform some basic operations on these numbers that involve the usage of **fabs().**

The following program illustrates the same-

1. #working of fabs()
2. **import** math
3. # initializing the variables a, b, and c
4. a = -20.3
5. b = -30.66
6. c = 19.13
7. # calculating the sum
8. print ("The sum of a and b is: ", (a + b))
9. print ("The sum of b and c is: ", (b + c))
10. # printing the values
11. d = math.fabs(a)
12. e = math.fabs(b)
13. f = math.fabs(c)
14. #calculating sum after fabs
15. print ("The sum of d and e is: ", (d + e))
16. print ("The sum of e and f is: ", (e +  f))

**Output:**

The sum of a and b is: -50.96

The sum of b and c is: -11.530000000000001

The sum of d and e is: 50.96

The sum of e and f is: 49.79

**Explanation:**

Let us see what we have done in the above program-

1. Since we have to use **fabs(),** we have imported the math module.
2. After this, we have initialized the variables a, b, and c with positive and negative values.
3. In the next step, we have printed the sum of a + b and b + c.
4. Now we have passed these three variables in **fabs()** and stored the returned values in variables d, e, and f.
5. We have calculated the sum again to check the difference between the results obtained using the values that were used for evaluation before and after passing into fabs.
6. On executing the program, it displays the desired output.

### Program 3:

Finally, in the last program, we will perform one more mathematical operation on the numbers.

1. #working of fabs()
2. **import** math
3. #initialising the variables a, b and c
4. a = -20.3
5. b = -30.66
6. c = 19.13
7. #calculating the product
8. print ("The product of a and b is: ", (a \* b))
9. print ("The product of b and c is: ", (b \* c))
10. #printing the values
11. d = math.fabs(a)
12. e = math.fabs(b)
13. f = math.fabs(c)
14. #calculating product after fabs
15. print ("The product of d and e is: ", (d \* e))
16. print ("The product of e and f is: ", (e \* f))

**Output:**

The product of a and b is: 622.398

The product of b and c is: -586.5258

The product of d and e is: 622.398

The product of e and f is: 586.5258

**Explanation:**

Let's have a look at the explanation of this program-

1. Since we have to use **fabs(),** we have imported the math module.
2. After this, we have initialized the variables a, b, and c with positive and negative values.
3. In the next step, we have printed the product of a \* b and b \* c.
4. Now we have passed these three variables in **fabs()** and stored the returned values in variables d, e, and f.
5. We have calculated the product again to check the difference between the results obtained using the values that were used for evaluation before and after passing into fabs.
6. On executing the program, it displays the desired output.

## Conclusion

In this Tutorial, we learned **fabs()** and how they can be used in Python programs to make our work much easier.

# Convert String to Float in Python

We are quite familiar with the data types available in Python and what kind of different operations we can perform on them.

We know that,

Strings are a sequence of characters that are denoted using inverted commas ''. They are immutable which means they cannot be changed once declared.

And float values are nothing but the decimal values.

We can check the types using the program given below-

1. x = 2.71
2. # printing the type of x
3. print(type(x))
4. y = 'Moscow'
5. #printing the type of y
6. print(type(y))

**Output:**

<class 'float'>

<class 'str'>

In this tutorial, we will discuss the different scenarios of converting a string to float in Python.

1. Using float function
2. Converting a string with commas
3. Converting to a float list
4. Converting a list of strings to float
5. Converting using NumPy
6. Converting in specified decimal points

### Using Float Function

Let us look at the first case where we will use the float function,

The following program illustrates the same-

1. x = '2.71'
2. res = **float**(x)
3. print("The float value is ",res)

**Output:**

The float value is 2.71

**Explanation:**

Let's understand what we have done in the above program-

1. The first step is to declare the string value that we would like to convert.
2. After this, we will make use of a float() in which we'll then pass the variable that holds the string value.
3. On executing the program, we can observe that it displays the required value.

### Converting a String with Commas

Let us look at the second case where we will convert a string with commas,

The following program illustrates the same-

1. x = '27,.71'
2. res = **float**(x.replace(',', ''))
3. print("The float value is ",res)

**Output:**

The float value is 2.71

**Explanation:**

Let's understand what we have done in the above program-

1. The first step is to declare the string value that we would like to convert, this time a comma is present before the decimal point.
2. After this, we will make use replace() inside float() in which we'll specify both, the value we would like to replace and with what it would be replaced.
3. On executing the program, we can observe that it displays the required value.

### Converting to a Float List

Let us look at the third case where we will convert the string to a float list,

The following program illustrates the same-

1. x='1.21, 2.34, 3.42, 9.22, 5.43'
2. res=[**float**(i) **for** i in x.split(', ')]
3. print("The list having float values: "+str(res))

**Output:**

The list having float values: [ 1.21, 2.34, 3.42, 9.22, 5.43]

**Explanation:**

Let's understand what we have done in the above program-

1. The first step is to declare the string value that we would like to convert, this time there is a list of different decimal values separated with a comma.
2. After this, we will make use of list comprehension that will take each value and convert it into the required type.
3. On executing the program, we can observe that it displays the required value.

### Converting List of Strings to Float

Let us look at the fourth case where we will convert list of strings to float,

The following program illustrates the same-

1. x=['1.21', '2.34', '3.42', '9.22', '5.43']
2. res=[]
3. **for** i in x:
4. res.append(**float**(i))
5. print('The list having float values:',res)

**Output:**

The list having float values: [ 1.21,2.34,3.42,9.22,5.43]

Let's understand what we have done in the above program-

1. The first step is to declare the list that contains all the string values as it's elements
2. After this, we have declared an empty list and used for loop that takes each element, converts it into float value, and appends it into res.
3. On executing the program, we can observe that it displays the required value.

### Converting Using NumPy

Let us look at the fifth case where we will convert the string using numpy,

The following program illustrates the same-

1. **import** numpy as np
2. x=np.array(['1.21', '2.34', '3.42', '9.22', '5.43'])
3. res=x.astype(**float**)
4. print('The list having float values:',res)

**Output:**

The list having float values: [1.21 2.34 3.42 9.22 5.43 ]

**Explanation:**

Let's understand what we have done in the above program-

1. The first step is to import the numpy library as np.
2. After this, we have declared the array that has all the string values.
3. Once this is done, the next step is to use astype() and specify float in it since this is the functionality/feature we wish to apply on all the elements.
4. On executing the program, we can observe that it displays the required value.

### Converting in Specified Decimal Points

Finally, in the last program we will see how we can convert a string to specified decimal points,

The following program illustrates the same-

1. x='2.88731'
2. val=**float**(x)
3. res="{:.2f}".format(val)
4. print(res)

**Output:**

2.89

**Explanation:**

Let's understand what we have done in the above program-

1. The first step is to declare the string value that we would like to convert.
2. After this, we will make use of a float() and pass the variable x.
3. Since we want our output in two decimal places, we will mention this in the next statement.
4. On executing the program, we can observe that it displays the required value.

## Conclusion

So, in this tutorial, we understood the different scenarios and learned approaches to convert a string to float in Python.

# choice() in Python

In this tutorial, we will discuss the usage of the **choice()** method in Python.

To use this in our program, we need to first import the random module.

The function of **choice()** is to pick or generate a random element that can be anything a number or a string from a given collection.

### Syntax

The syntax for using **choice()** is

1. random.choice(x)

Let us have a look at some programs that will make it's application clear.

### Program 1:

Consider the following program-

1. **import** random
2. list\_values=["C","C++","Python","Java","PHP"]
3. print(random.choice(list\_values))

**Output:**

Java

**Explanation:**

It's time to have a look at the explanation of the program given above-

1. In the first step, we have imported the random module.
2. After this, we have initialized a list that contains different string values.
3. Finally, we have used the **choice()** to pick a certain string from the provided list and display its value.

### Program 2:

In the second program, we will see how it can be used to take a random character from the string.

1. **import** random
2. string\_value="Python at JavaTpoint"
3. print(random.choice(string\_value))

**Output:**

y

**Explanation:**

Let's understand what happened here-

1. In the first step, we have imported the random module.
2. After this, we have initialized a string called "Python at JavaTpoint.
3. Finally, we have used the **choice()** to pick a certain character from the provided string and display it's value.

### Program 3:

Now, let's have a look at one more program and discuss another approach of writing this method.

1. **import** random
2. print("The random element from(1,2,3,4,5) is  ",random.choice([1,2,3,4,5]))
3. print("The random element from(a,b,c,d,e,f) is  ",random.choice(['a','b','c','d','e','f']))

**Output:**

The random element from(1,2,3,4,5) is 5

The random element from(a,b,c,d,e,f) is f

**Explanation:**

Let's see what we have done in this program-

1. In the first step, we have imported the random module.
2. After this, we have written two print statements that will display the value.
3. And then we can observe that we have used the random.choice() in the print function itself to pick a certain character or element from the provided list and display its value.

### Program 4:

In the next program, we shall see how we can choose a definite number of values to be displayed in the output.

1. #importing random
2. **import** random
3. #initializing the list
4. list\_values=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
5. #taking a random value from the list
6. res=random.choices(list\_values, k=4)
7. print(res)

**Output:**

[2,2,3,2]

**Explanation:**

It's time to have a look at the explanation of the program given above-

1. In the first step, we have imported the random module.
2. After this, we have initialized a list that contains different numeric values.
3. Finally, we have used the random.choice() in which we have specified the list name and k value as parameters.
4. On executing the program, we can observe that the number of elements in the returned list depends on the value of k.

### Program 5:

Now, we will see how a for loop can be used to meet the same objective.

The following program illustrates the same-

1. #importing random
2. **import** random
3. #initializing the list
4. list\_values=[1,2,3,4,5,6,7,8,9,10]
5. #using **for** loop
6. **for** i in range(6):
7. print(random.choice(list\_values))

**Output:**

1

3

1

3

1

4

### Program 6:

In the next program, we will see how a random value from a given range of numbers can be taken.

1. #importing random
2. **import** random
3. #taking a random number
4. num\_value = random.choice(range(10,120))
5. print("A random number from the given range is",num\_value)

**Output:**

A random number from the given range is 31.

**Explanation:**

Let's see what we have done in this program-

1. In the first step, we have imported the random module.
2. After this, we have specified the range of numbers in **choice().**
3. On executing the given program, it displays the expected output.

Now, we will learn how to get the Boolean values using **choice().**

## Displaying Boolean Result

The following program shows how we can obtain the Boolean values.

1. #importing random
2. **import** random
3. #taking a random **boolean** value
4. val=random.choice([True,False])
5. print("The boolean value is",val)

**Output:**

The boolean value is False

**Explanation:**

It's time to have a look at the explanation of the program given above-

1. In the first step, we have imported the random module.
2. After this, we have specified True and False inside **choice().**
3. On executing the given program, the expected output is displayed.

### Program 7:

Finally, the last program illustrates how **random.getrandbits()** can be used for the same purpose.

1. #importing random
2. **import** random
3. #taking a random **boolean** value
4. val=random.getrandbits(1)
5. print(bool(val))

**Output:**

False

The procedure of the above program is similar to the previous program, the only difference is here we have used **random.getrandbits().**

## Conclusion

In this tutorial, we learned the different ways of using **choice()** in our Python program.

# Decorators with Parameters in Python

In this tutorial, we will discuss Decorators with parameters in Python, but before starting this topic, users must study [Decorators in Python](https://www.javatpoint.com/python-decorator), Function Decorators.

Decorators are a very powerful and useful tool in Python because it allows users to modify the behaviour of function or class.

Python functions can be treated as supreme citizens as objects.

* Functions can be referenced to a variable.
* Functions can be passed as an argument to other functions.
* Functions can be returned from a function.

Decorators with parameters are like normal decorators.

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

### Syntax:

1. @decorator(params)
2. def function\_name():
3. '''Function implementation'''

### Code Implementation of Decorators with Parameters:

1. def function\_name():
2. '''Function implementation'''
3. function\_name = (decorator(params))(function\_name)
4. """

As we execute this code, the execution will start from left to right, which will call **decorator(params)** for returning the function object **func\_obj.** the **func\_obj(function\_name)** will be called by using **func\_obj**. Inside the inner function, required operations will be performed, and the actual function reference will return for assigning to the **function\_name**. Now, the users can use the **function\_name()** to call the function with decorator applied.

### How to Implement Decorators with Parameters:

First, we will see what we could get as output if we run a code of parameters directly without implementing any value.

1. def decorators\_1(\*args, \*\*kwargs):
2. def inner\_1(func\_1):
3. '''
4. doing operations with func\_1
5. '''
6. **return** func\_1
7. **return** inner\_1 # **this** is the function\_object mentioned in the above content
9. @decorators\_1(params)
10. def func\_1():
11. """
12. function implementation
13. """

Here, in the above code, as params is empty, we might get some errors.

Let's understand this stepwise:

1. def decorator\_function(function\_name):
2. print("Inside the Decorator: ")
4. def inner\_1(\*args, \*\*kwargs):
5. print("Inside the Inner Function: ")
6. print("'Decorated the function'")
7. # perform **this** operations with function\_name
9. function\_name()
11. **return** inner\_1
13. @decorator\_function
14. def function\_to():
15. print("Inside the actual function")
17. function\_to()

**Output:**

Inside the Decorator:

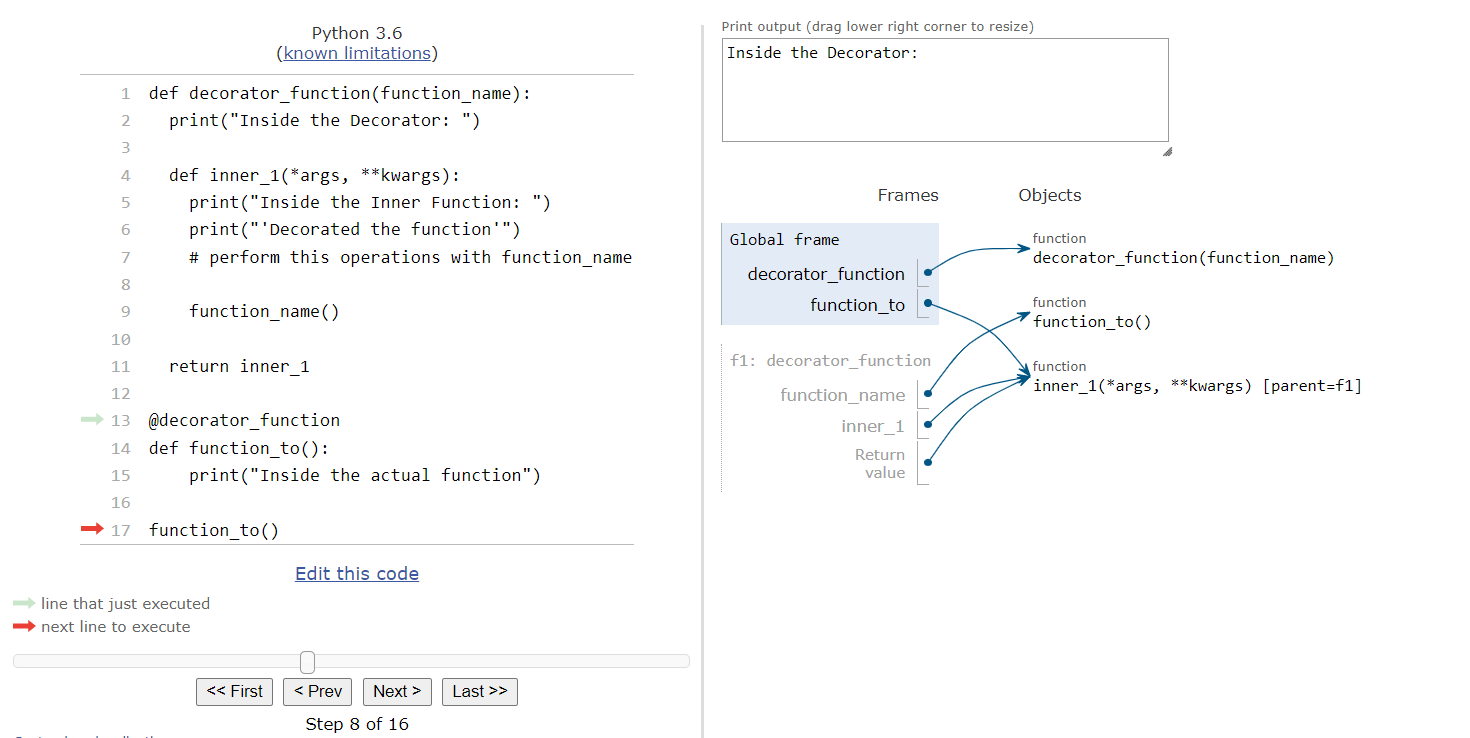
Inside the Inner Function:

'Decorated the function'

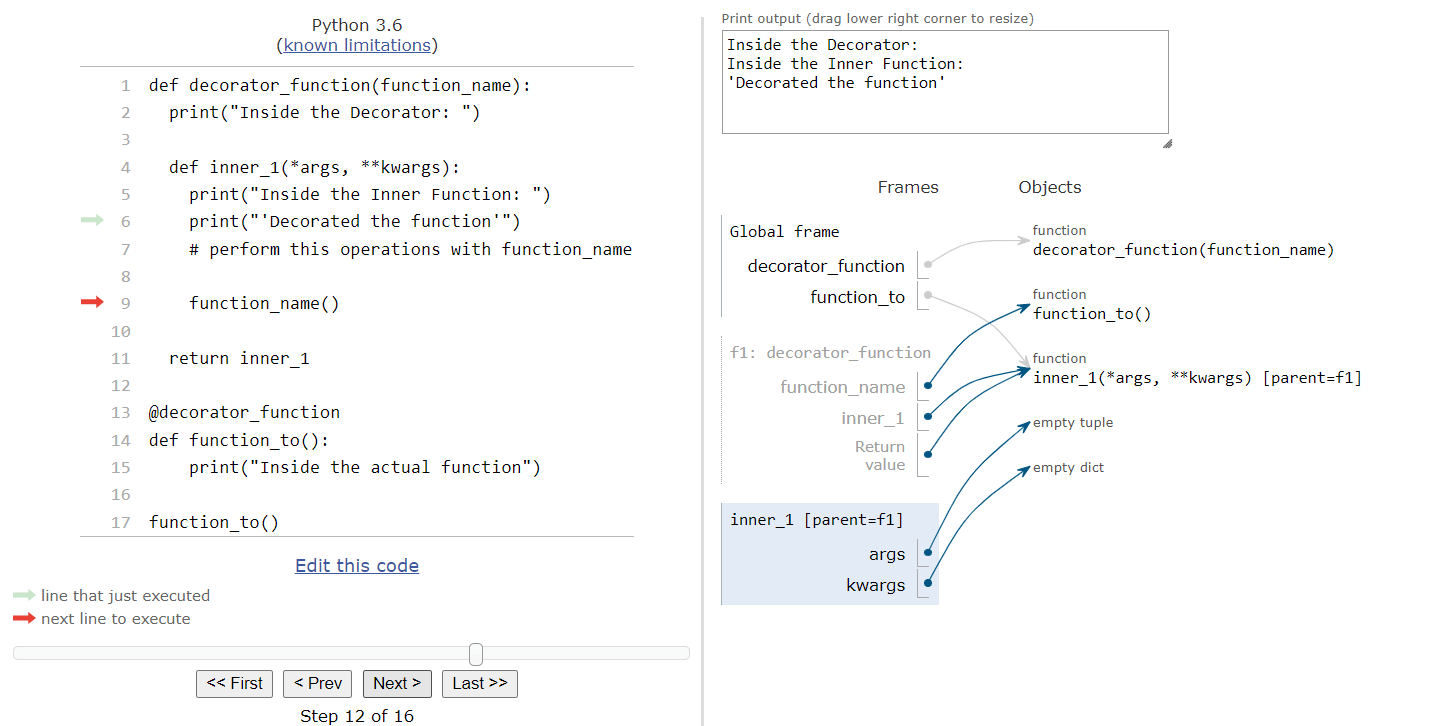
Inside the actual function

### Visual Representation of the Code Execution:

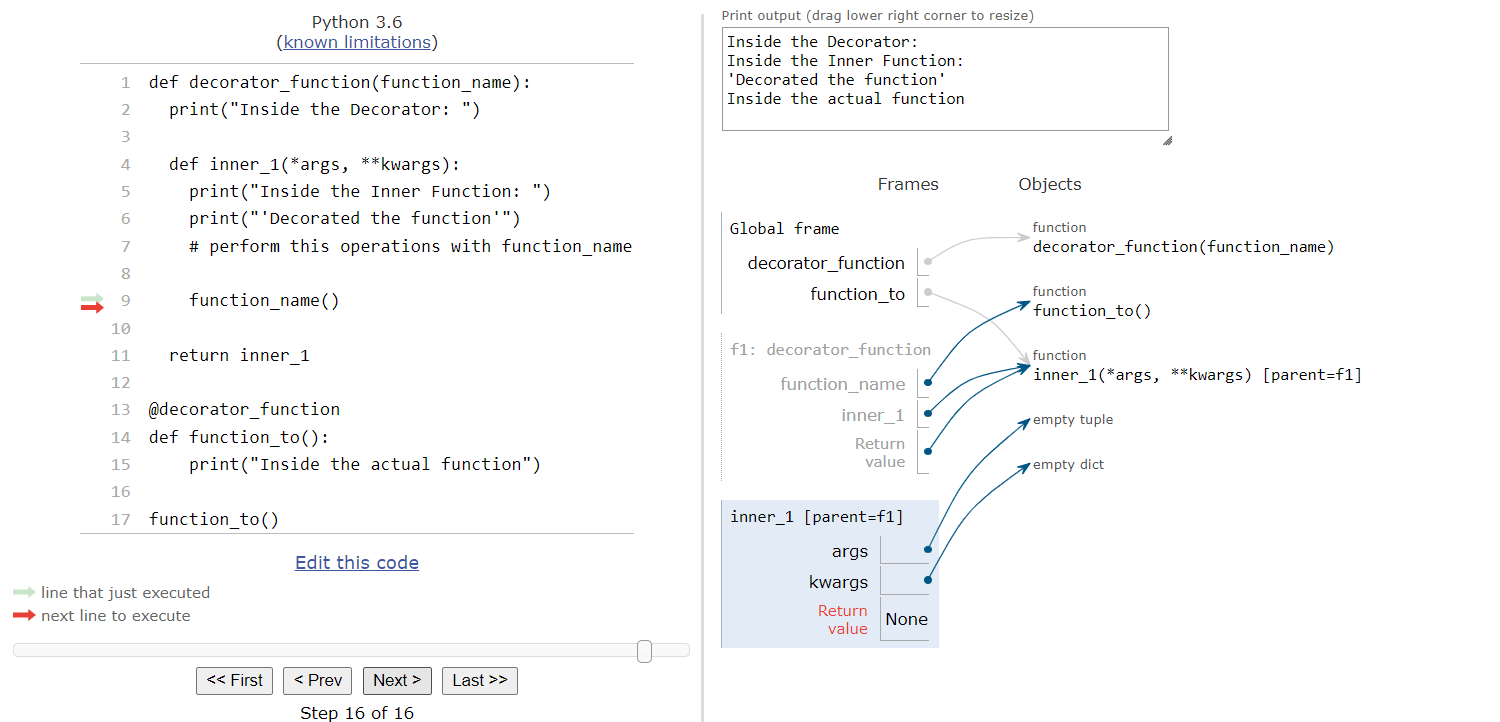
**Inside the Decorator Execution:**



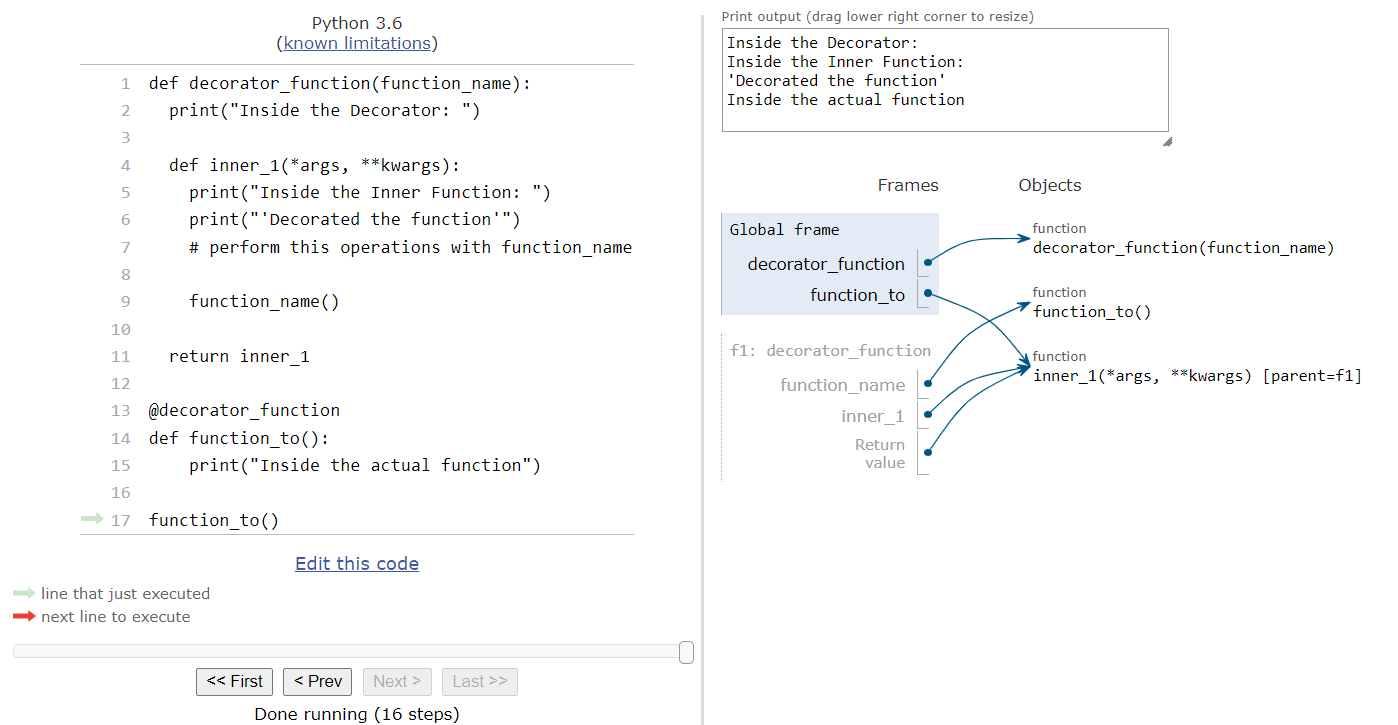
**Inside Inner Function Execution:**



**Decorator Function Execution:**



**Final Output Execution:**



In the above code, we will get the outputs from the functions called using decorators with parameters.

### Alternate Way:

In the following code, we will see how to write a code for using decorators for functions in an alternative way.

1. def decorator\_fun(function\_name):
2. print ("Inside the Decorator: ")
4. def inner\_1(\*args, \*\*kwargs):
5. print ("Inside the Inner Function: ")
6. print ("'Decorated the function'")
7. # Perform **this** operations with function\_name
9. function\_name()
11. **return** inner\_1
13. def function\_to():
14. print ("Inside the actual function")
16. # This is another way of using decorators
17. decorator\_fun(function\_to)()

**Output:**

Inside the decorator

Inside the inner function

Decorated the function

Inside the actual function

Now we will see different examples of using decorations with the parameter for better understanding the concept.

**Example 1:**

1. def decorator\_1(\*args, \*\*kwargs):
2. print("Inside the Decorator")
4. def inner\_1(function\_1):
6. # Here, we will see the functionality of the code:
7. print ("Inside the inner function")
8. print ("I am studying ", kwargs['JTP'])
10. function\_1()
12. # Returning the inner function
13. **return** inner\_1
15. @decorator\_1(JTP = "COMPUTER SCIENCE AND ENGINEERING ")
16. def my\_function():
17. print ("Inside the actual function")

**Output:**

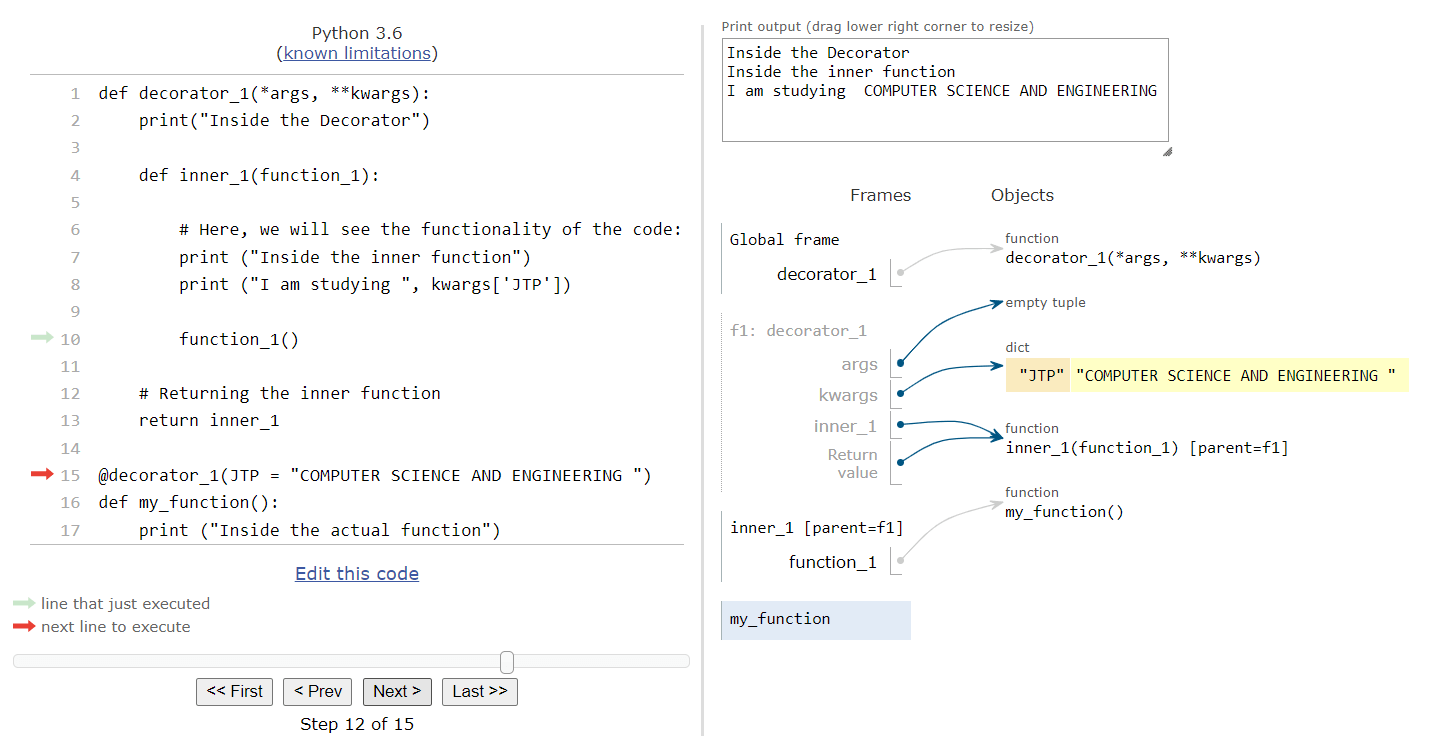
Inside the Decorator

Inside the inner function

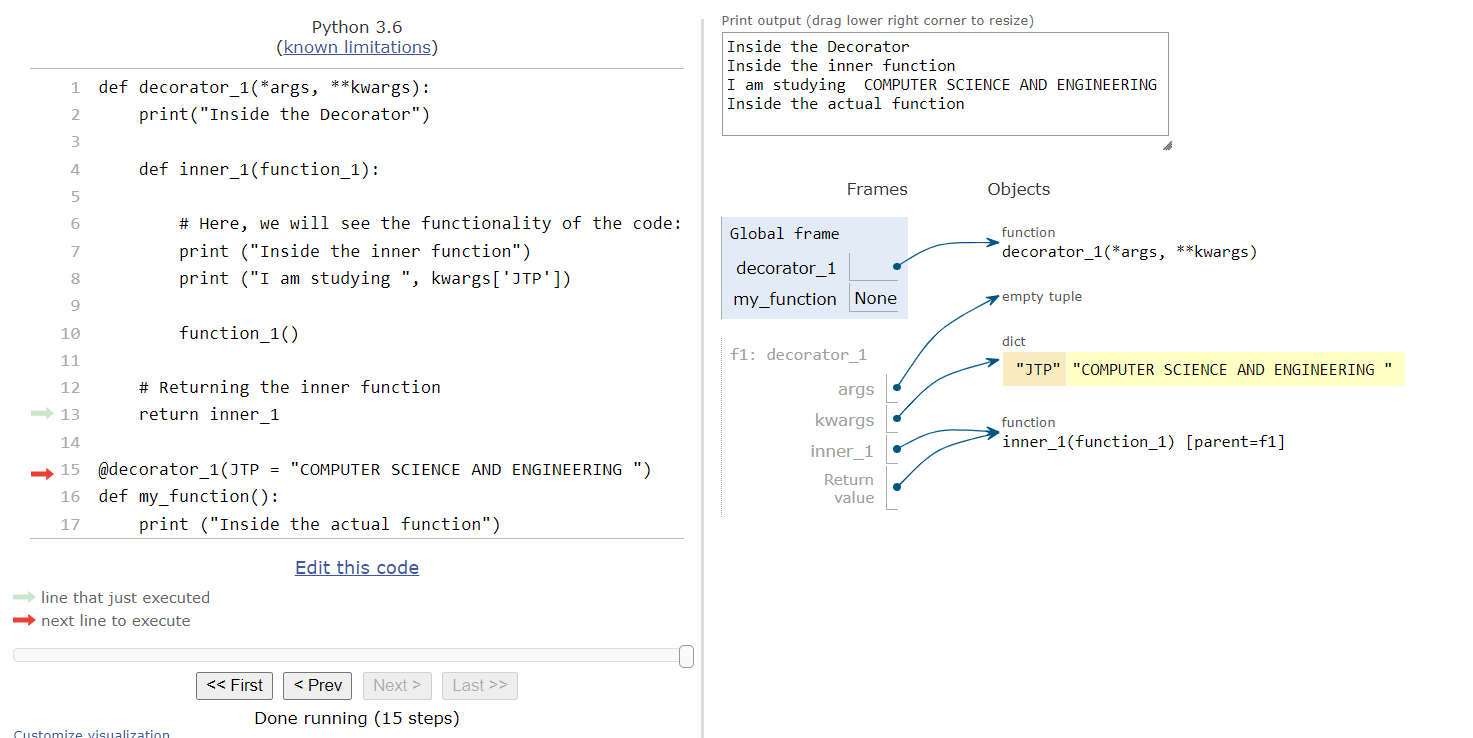
I am studying COMPUTER SCIENCE AND ENGINEERING

Inside the actual function

### Visual Representation of the Code Execution:



**Final Output Execution:**



### Example 2:

1. def decorator\_function(A, B):
3. def Inner\_1(function\_1):
5. def wrapper\_1(\*args, \*\*kwargs):
6. print ("I am studying COMPUTER SCIENCE AND ENGINEERING ")
7. print ("Summation of values - {}".format(A + B) )
9. function\_1(\*args, \*\*kwargs)
11. **return** wrapper\_1
12. **return** Inner\_1

15. # here, we are not using decorator
16. def my\_function(\*args):
17. **for** ele in args:
18. print (ele)
20. # another way of using decorators
21. decorator\_function(22, 14)(my\_function)('Computer', 'Science', 'and', 'Engineering')

**Output:**

I am studying COMPUTER SCIENCE AND ENGINEERING

Summation of values - 36

Computer

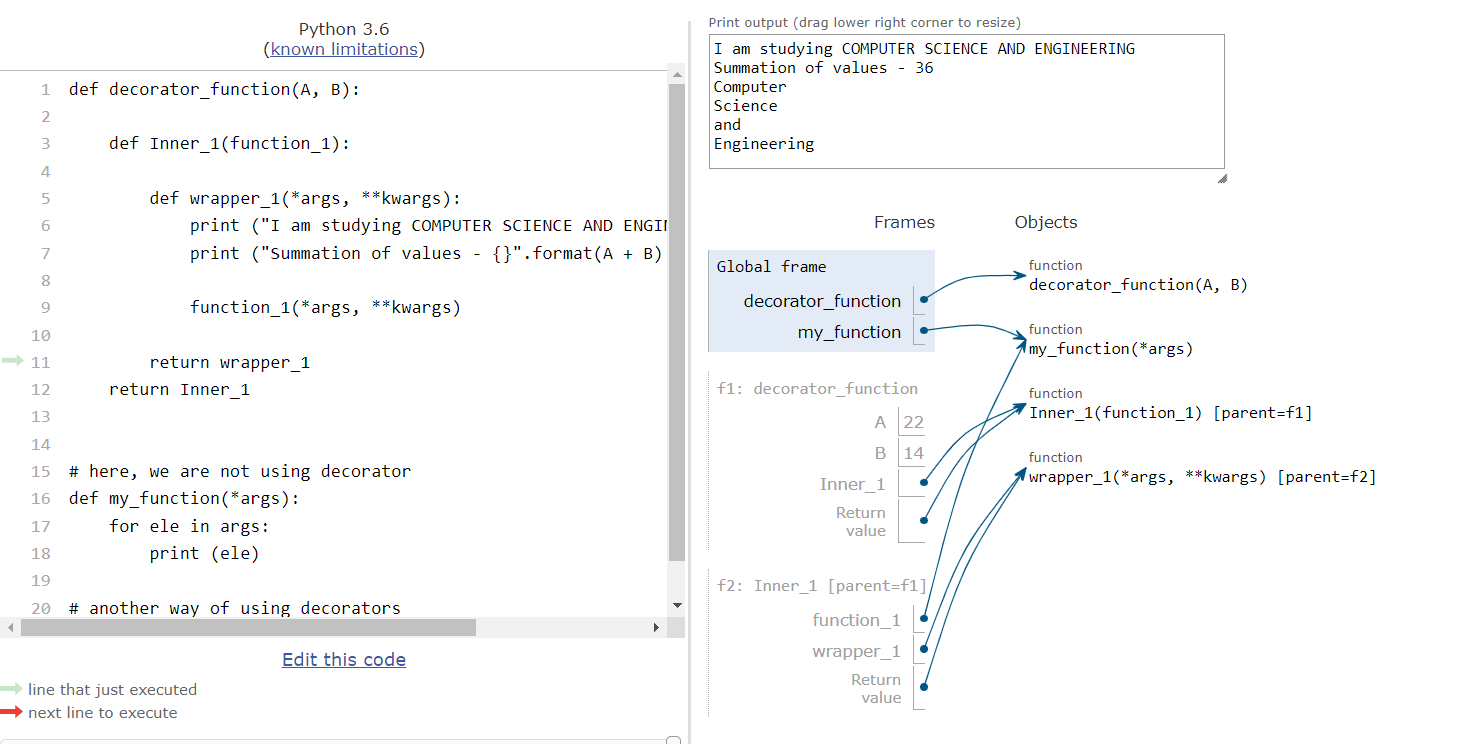
Science

and

Engineering

The above example also shows that the enclosed inner function can access the parameters of the outer function.

### Visual Representation of the Code Execution:



**Example 3:**

1. def deco\_decorator(dataType, message\_1, message\_2):
2. def decorator\_1(function\_1):
3. print (message\_1)
4. def wrapper\_1(\*args, \*\*kwargs):
5. print(message\_2)
6. **if** all([type(arg) == dataType **for** arg in args]):
7. **return** function\_1(\*args, \*\*kwargs)
8. **return** "Invalid Input"
9. **return** wrapper\_1
10. **return** decorator\_1

13. @deco\_decorator(str, "Decorator for 'string\_Join'", "stringJoin process started ...")
14. def string\_Join(\*args):
15. st1 = ''
16. **for** K in args:
17. st1 += K
18. **return** st1

21. @deco\_decorator(**int**, "Decorator for 'summation\_1'\n", "summation process started ...")
22. def summation\_1(\*args):
23. summ1 = 0
24. **for** arg in args:
25. summ1 += arg
26. **return** summ1

29. print (string\_Join("I ", 'am ', "studying ", 'Computer ', "Science ", "and ", "Engineering"))
30. print ()
31. print ("The sum is equal to: ", summation\_1(22, 12, 48, 133, 627, 181, 219))

**Output:**

Decorator for 'string\_Join'

Decorator for 'summation\_1'

stringJoin process started ...

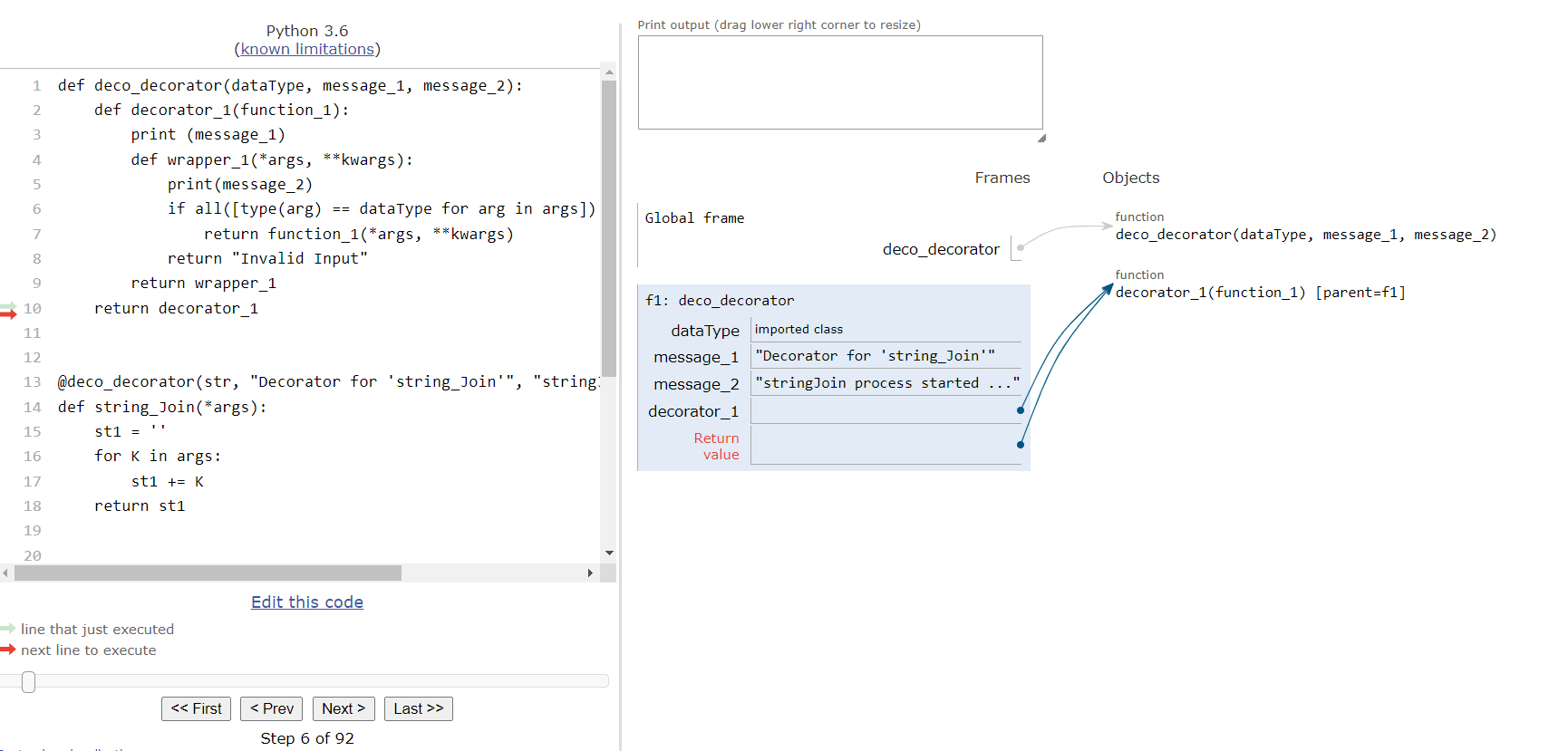
I am studying Computer Science and Engineering

summation process started ...

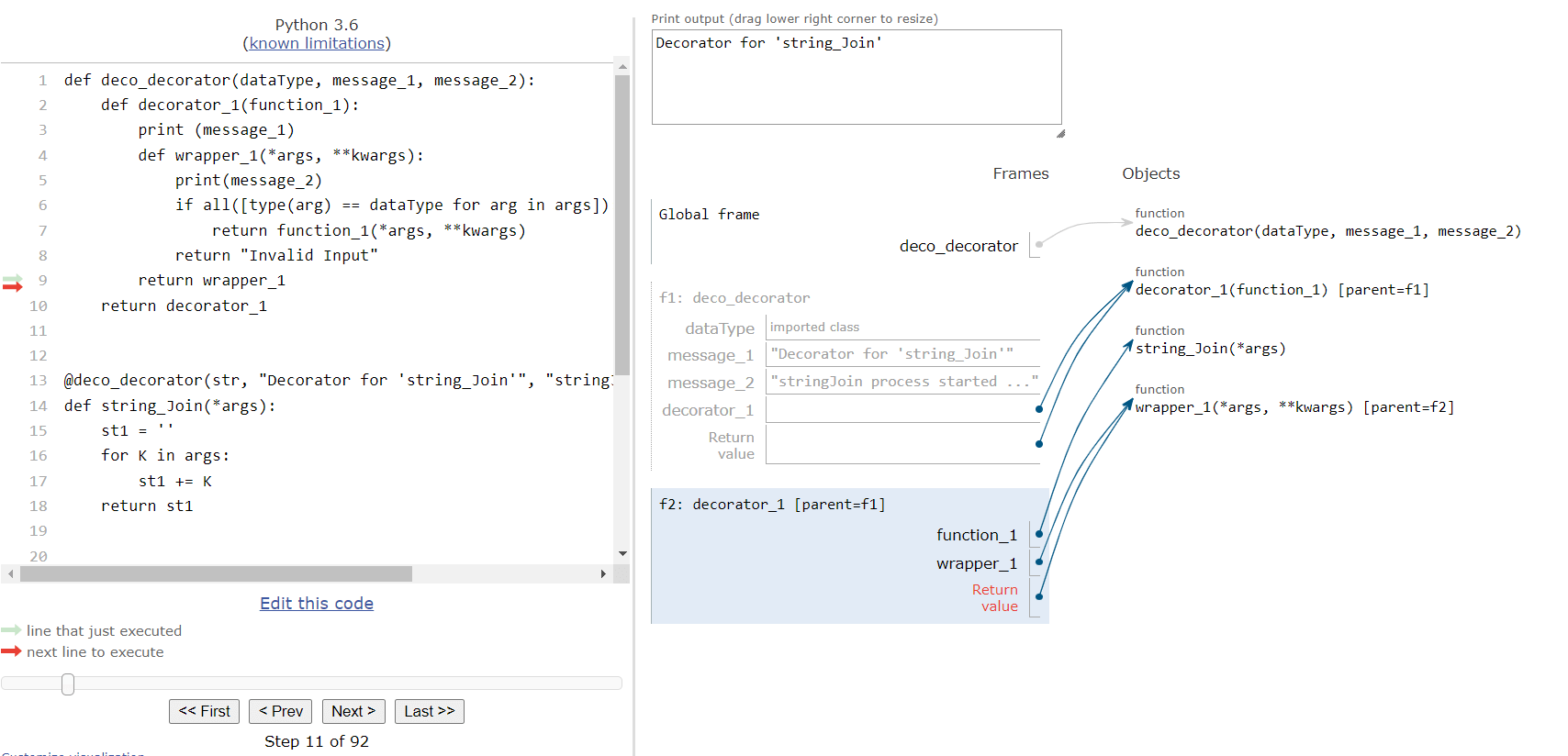
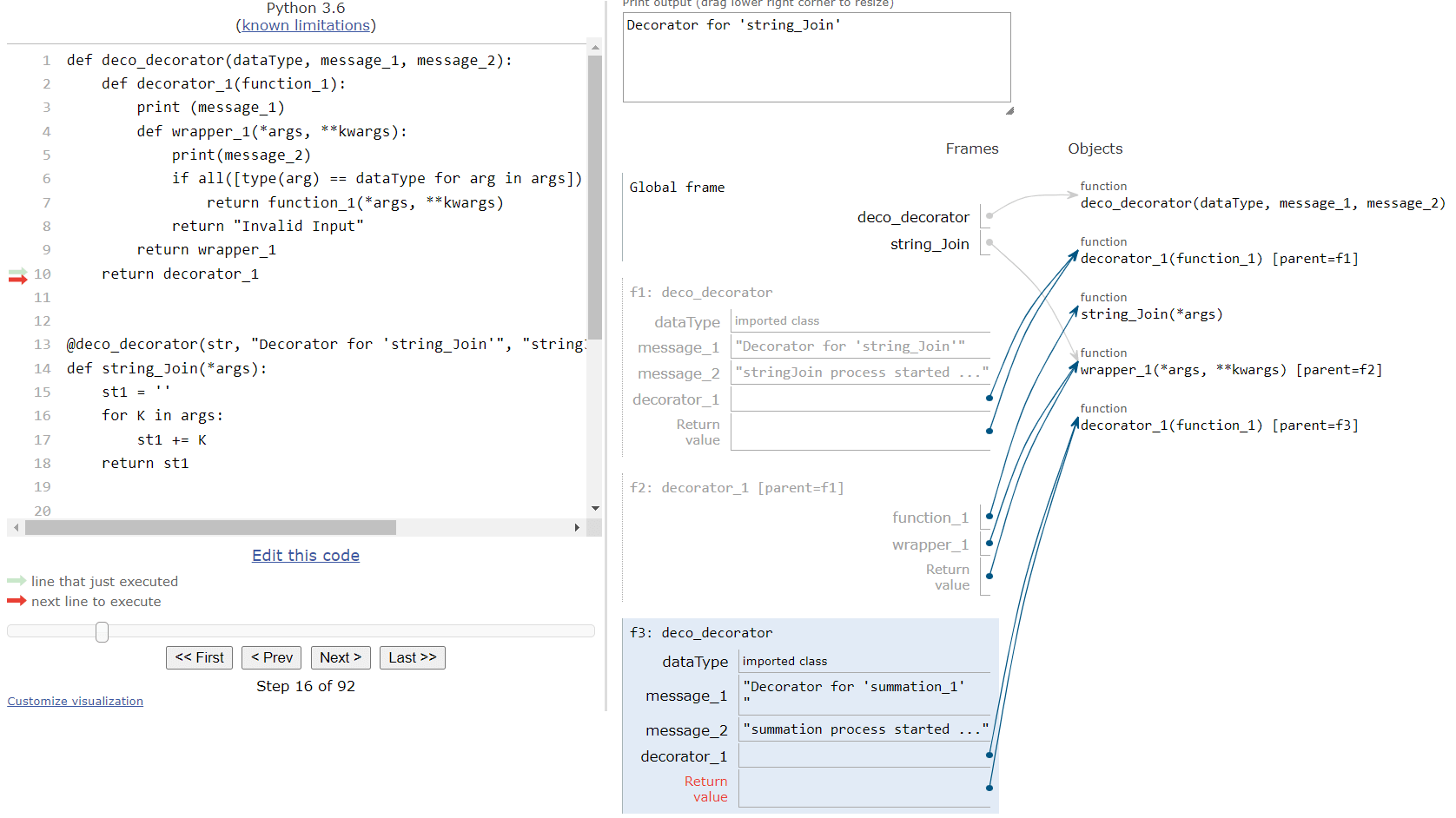
The sum is equal to: 1242

### Visual Representation of the Process:

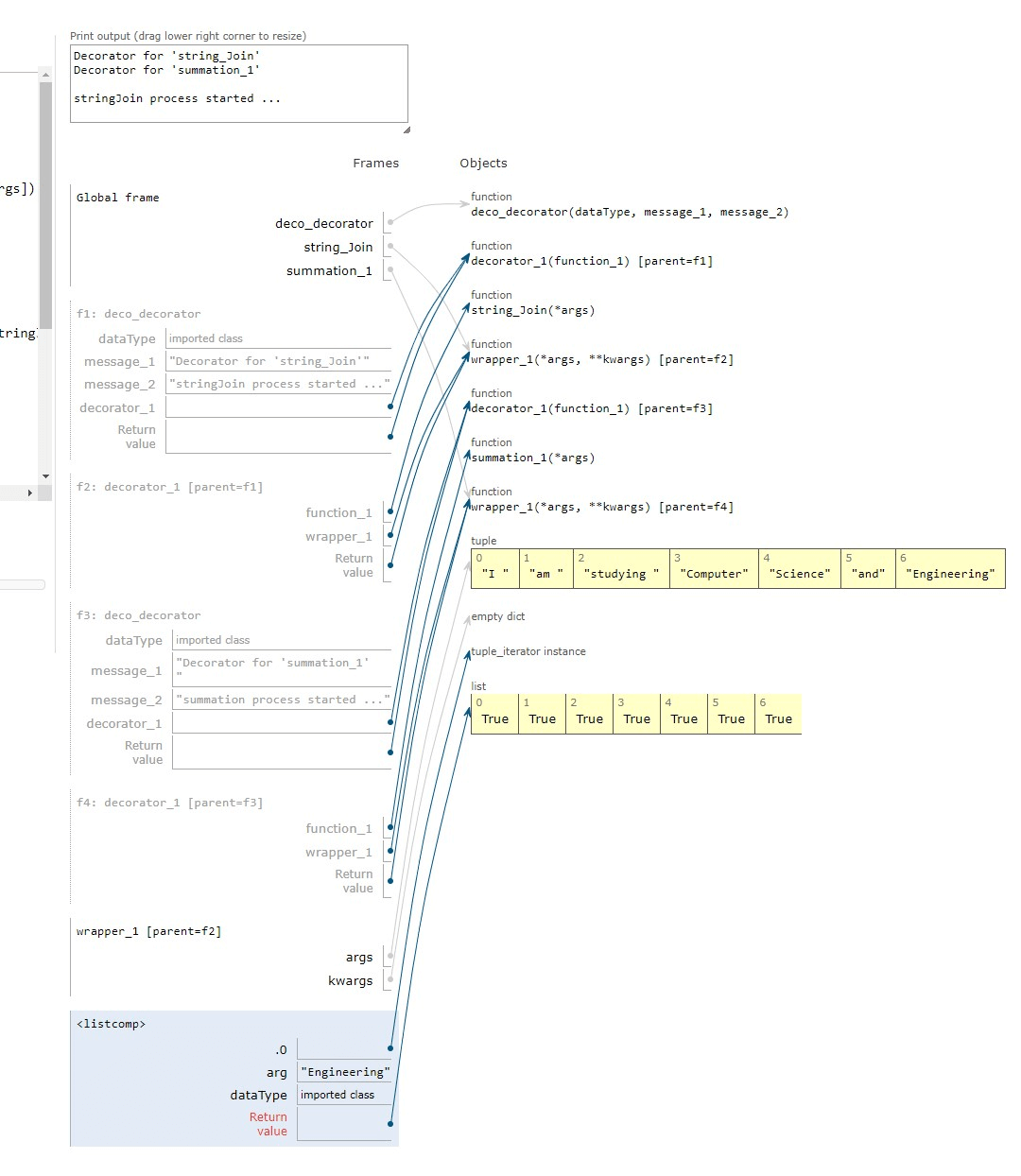
**Returning decorator\_1:**



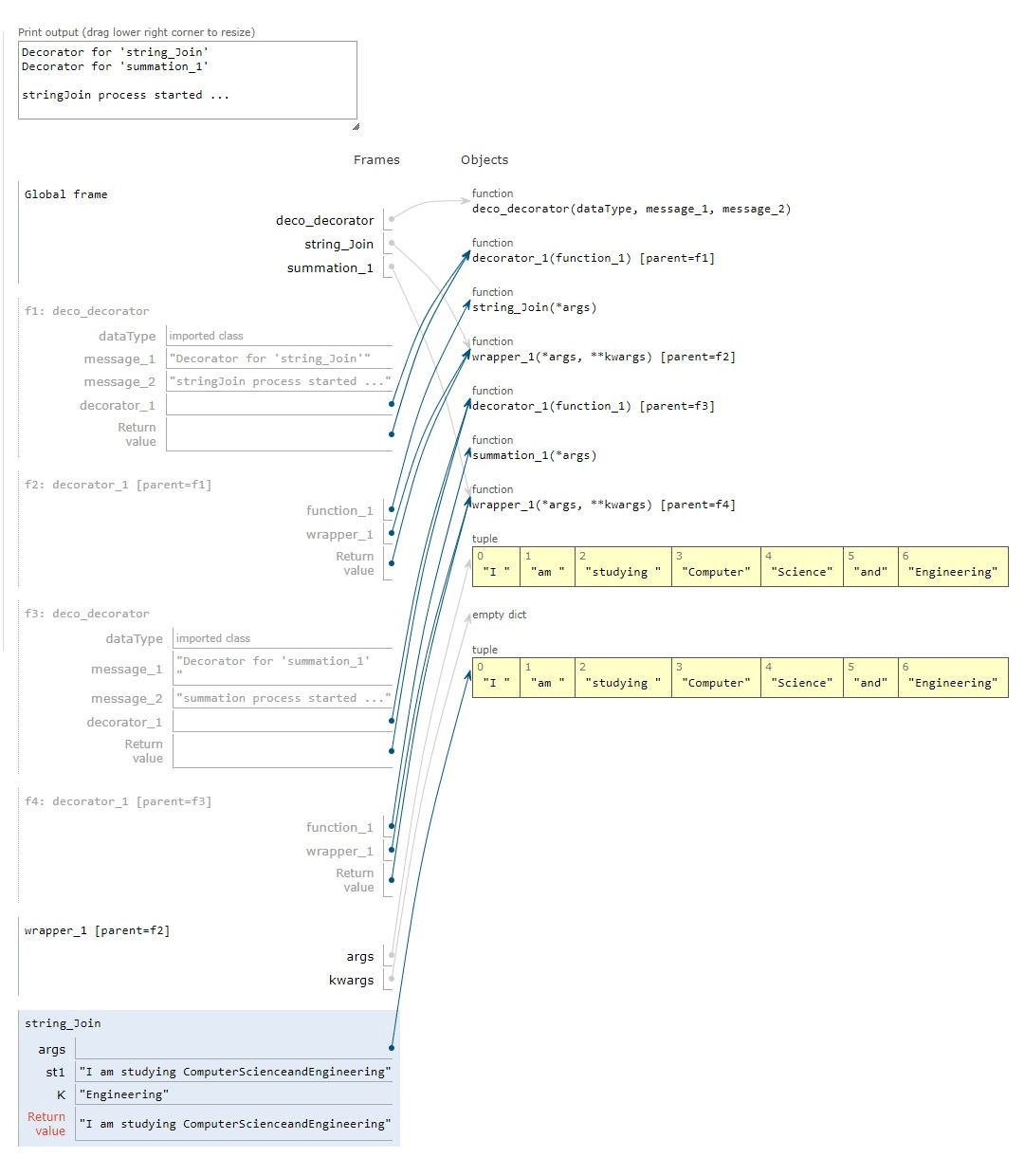
**Returning wrapper\_1:**

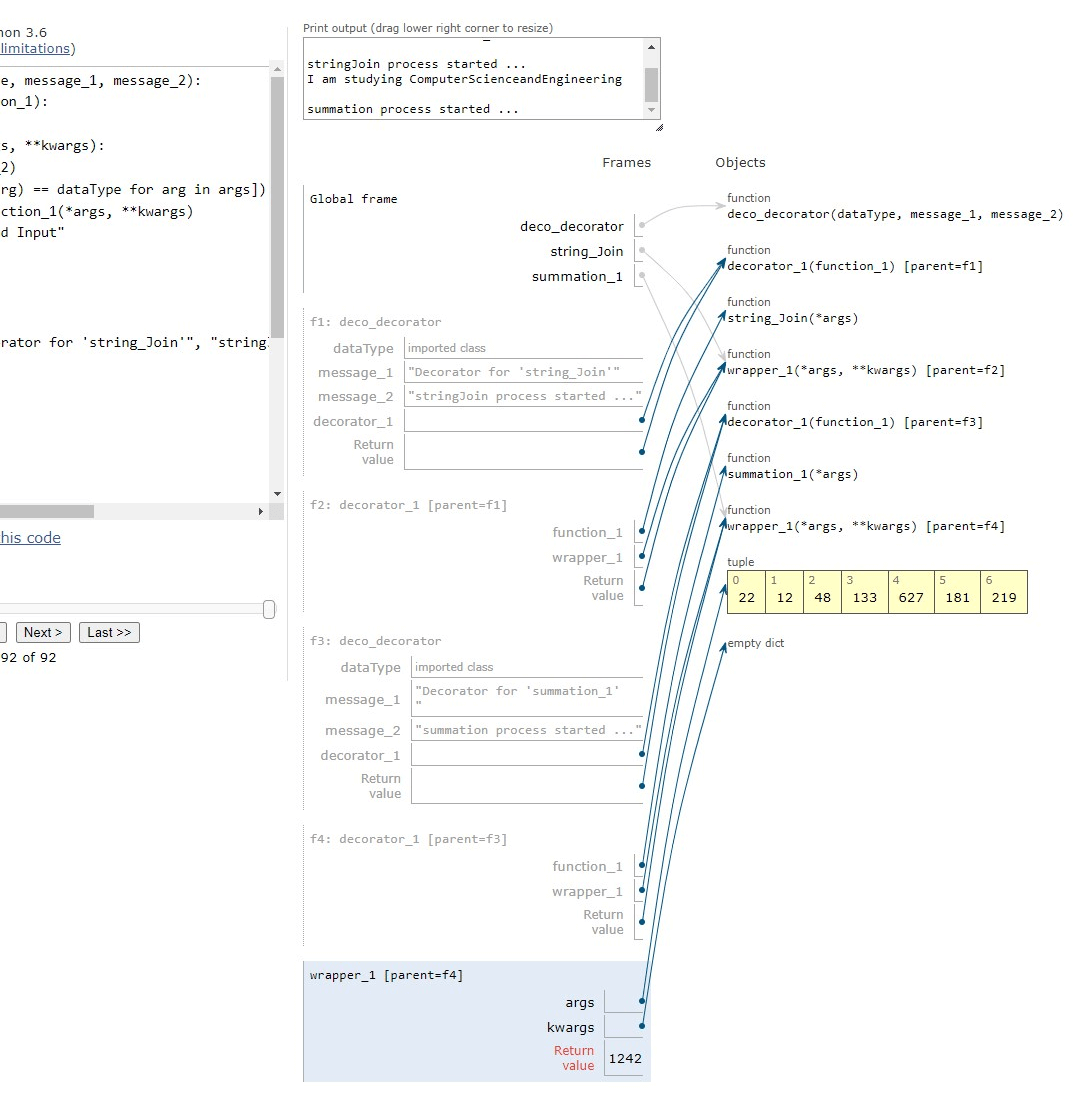
**Executing message\_1:**



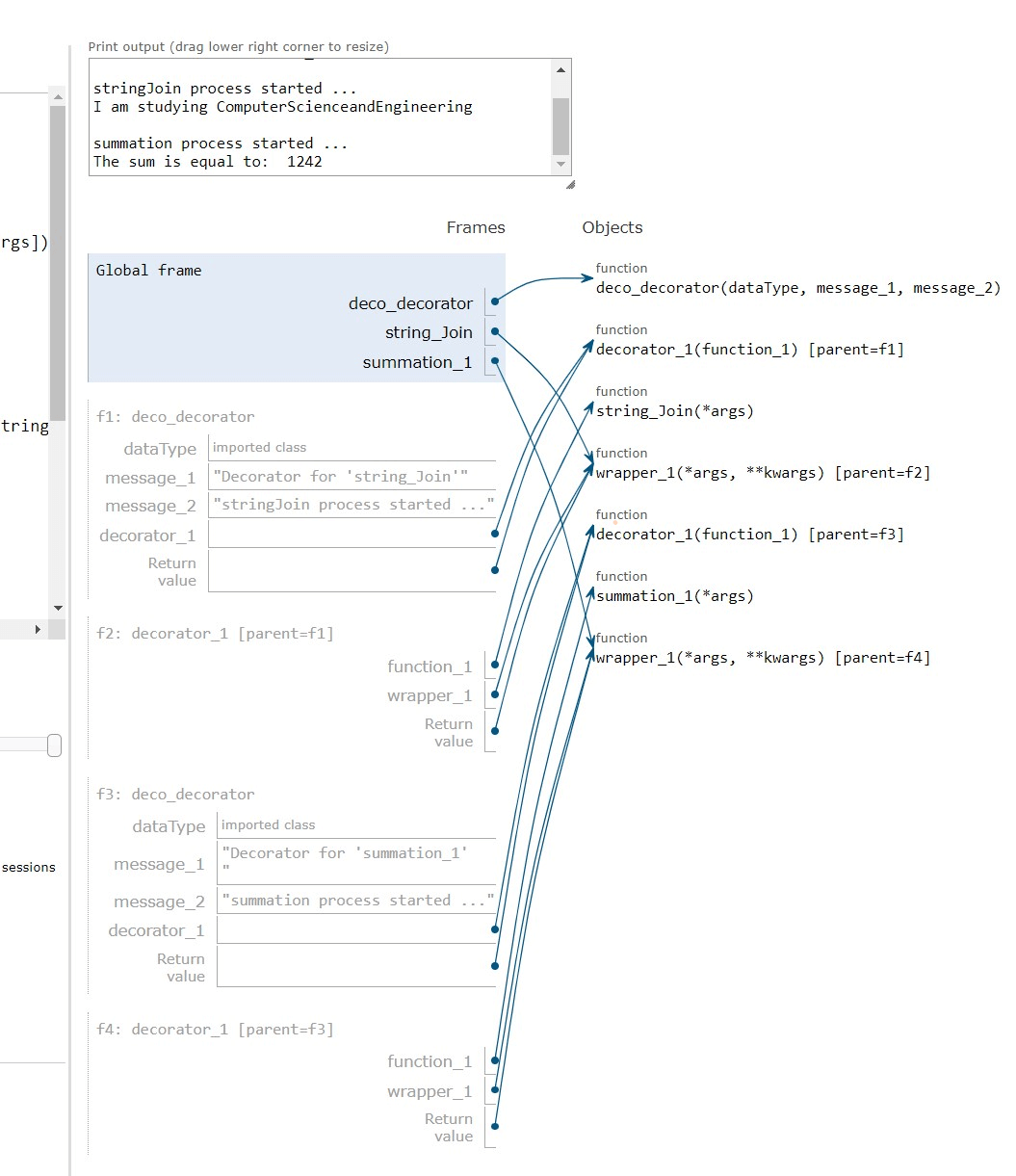
**Executing String\_Join:**



**Executing summation\_1:**



**Final Output Execution:**



## Conclusion

In this tutorial, we have discussed how to execute the function using a decorator with parameters. We have also explained examples using a visual representation of inner functions and Outer functions of the parameters.

# String to int in Python

In this tutorial, we will learn the ways of converting a string to integer in Python-

Let us have a look at an example before proceeding-

1. a='Learning Python is fun'
2. b= 20
3. #Displaying the type of a and b
4. print(type(a))
5. print(type(b))

**Output:**

<class 'str'>

<class 'int'>

In the above example, we have declared the variables 'a' and 'b' with a string and an integer value respectively.

We can verify their data types using **type().**

The question that arises here is why do we need to convert a string to an integer.

The following program illustrates the same-

1. value\_a = "100"
2. value\_b = "26"
3. res = value\_a \* value\_b
4. print("The multiplication of val\_a and val\_b gives: ",res)

**Output:**

res = value\_a \* value\_b

TypeError: can't multiply sequence by non-int of type 'str'

Since it generates this type of error, this is the reason that we must convert the string values to integers so that we can easily proceed with the operations.

It's time to have a look at the first program that demonstrates converting a string to an integer.

1. a = '7'
2. print(type(a))
3. #using **int**()
4. conv\_a=**int**(a)
5. print(type(conv\_a))
6. conv\_a = conv\_a+10
7. print(conv\_a)
8. print(type(conv\_a))

**Output:**

<class 'str'>

<class 'int'>

17

<class 'int'>

**Explanation:**

Let's see the explanation of the above program-

1. The first step is to declare the variable 'a' with a string value.
2. After this, we have checked its data type using **type().**
3. For converting the string to an integer, we have used **int()** and then checked its type.
4. Now we have operated on the variable 'a' by adding 10 to it.
5. Finally, the resultant value is displayed in the output.

### Approach-2

In the next example, we will go for an indirect approach of converting a string to an integer.

The following program shows how it can be done-

1. value\_a = "100"
2. value\_b = "26"
3. print(type(value\_a))
4. print(type(value\_b))
5. #converting to **float**
6. value\_a=**float**(value\_a)
7. #converting to **int**
8. value\_b=**int**(value\_b)
9. res\_sum=value\_a+value\_b
10. print("The sum of value\_a and value\_b is ",res\_sum)

**Output:**

<class 'str'>

<class 'str'>

The sum of value\_a and value\_b is 126.0

**Explanation:**

Let us understand what we have done in the above program-

1. The first step is to declare the two variables 'value\_a' and 'value\_b' with a string value.
2. After this, we have checked their data type using **type().**
3. For converting the string to an integer, we have used **float()** to convert the string to float value.
4. In the next step, we will convert the string value of 'value\_b' to an integer.
5. Now we have added 'value\_a' and 'value\_b' and printed their sum.
6. Finally, the resultant value is displayed in the output.

### Approach-3:

In the last program, we will discuss one more scenario of converting string to int in Python.

Here we will see how we can convert a number present as a string value to base 10 when it is on different bases.

The following program illustrates the same-

1. num\_value = '234'
2. # printing the value of num\_value
3. print('The value of num\_value is :', num\_value)
4. #converting 234 to base 10 assuming it is in base 10
5. print('The value of num\_value from base 10 to base 10 is:', **int**(num\_value))
6. #converting 234 to base 10 assuming it is in base 8
7. print('The value of num\_value from base 8 to base 10 is :', **int**(num\_value, base=8))
8. #converting 234 to base 10 assuming it is in base 6
9. print('The value of num\_value base 6 to base 10 is :', **int**(num\_value, base=6))

**Output:**

The value of num\_value is: 234

The value of num\_value from base 10 to base 10 is: 234

The value of num\_value from base 8 to base 10 is: 156

The value of num\_value base 6 to base 10 is: 94

**Explanation:**

It's time to have a glance at the explanation of the above program.

1. In the first step, we have declared the value of the variable.
2. Since the output will always be in base 10, we have provided the different base values inside **int().**
3. The base values we have taken here are 10, 8, and 6.
4. On executing the program, the expected output is displayed.

## Conclusion

In this tutorial, we learned the different ways of converting a string to an int value.

# Set to list in Python

In this article, we will discuss how we can convert a set to a list in Python.

Before that let's have a quick revision of lists and sets.

**List -** It is a sequence of elements enclosed in square brackets where each element is separated with a comma.

**Syntax of a list is-**

1. a = [1,2,4.5,'Python','Java']

We can print the list and check its type using-

1. print(a)
2. print(type(a))

#### NOTE: The list is mutable which means we can change its elements.

**Set -** It is an unordered collection of elements that contains all the unique values enclosed within curly brackets.

**Syntax of a set is-**

1. b = {1,2,4.5,'Python','Java'}

We can print the set and check its type using-

1. print(b)
2. print(type(b))

The different approaches of converting a set to a string that we will use are-

1. Using **list()**
2. Using **sorted()**
3. Using **\*set**
4. Using **for loop**
5. Using **frozenset**

### Using list()

In the first method, we will use **list()** to convert the set.

The following program shows how it can be done-

1. #declaring a set
2. subjects={'C','C++','Java','Python','HTML'}
3. #using list()
4. res=list(subjects)
5. print(res)

**Output:**

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

**Explanation:**

Let us understand what we have done in the above program-

1. The first thing that we have done here is to declare the set that consists of different subject names.
2. After this, we have used **list()** function in which we passed the set 'subjects'.
3. On executing the program, the desired output is displayed.

### Using sorted()

The second approach is to use the **sorted()** function to convert a set to a list.

The program below illustrates the same-

1. #defining a function
2. def convert\_set(set):
3. **return** sorted(set)
5. subjects={'C','C++','Java','Python','HTML'}
6. res = set(subjects)
7. print(convert\_set(res))

**Output:**

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

**Explanation:**

Let us understand what we have done in the above program-

1. The first thing that we have done here is, we created a function that takes a set as its parameter and returns the expected output.
2. After this, we have declared the variable of a set type that consists of different subject names.
3. The next step was to pass our set in the function 'convert\_set'.
4. On executing the program, the desired output is displayed.

### Using \*set

In the third method, we will use the **\*set** to convert a set to a list in Python.

The **\*set** unpacks the set inside a list.

The following program shows how it can be done-

1. #defining a function
2. def convert\_set(set):
3. **return** [\*set, ]
5. res = set({'C','C++','Java','Python','HTML'})
6. print(convert\_set(res))

**Explanation:**

Let us understand what we have done in the above program-

1. The first thing that we have done here is, we created a function that takes a set as its parameter and returns the expected output.
2. After this, we have passed the value of the set that consists of different subject names inside the **set().**
3. The next step was to pass our set in the function 'convert\_set'.
4. On executing the program, the desired output is displayed.

**Output**

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

### Using for loop

In the fourth method, we will use **for** loop to convert a set to a list in Python.

The program below illustrates the same-

1. #using **for** loop
2. subjects = set({'C','C++','Java','Python','HTML'})
4. res = []
6. **for** i in subjects:
7. res.append(i)

**Output:**

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

Let us understand what we have done in the above program-

1. The first thing that we have done here is to declare the set that consists of different subject names.
2. After this, we have declared an empty list res.
3. We have used for loop here, that took each element from the set and added it to the list.
4. On executing the program, the desired output is displayed.

### Using frozenset

Finally, in the last method, we will use **frozenset** to convert a set to a list in Python.

The difference between a set and a frozenset is that a set is mutable whereas a frozenset is immutable.

The following program shows how it can be done-

1. subjects = frozenset({'C','C++','Java','Python','HTML'})
3. res = list(subjects)
5. print(res)

**Output:**

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

**Explanation:**

Let us understand what we have done in the above program-

1. The first thing that we have done here is to declare the **frozenset** that consists of different subject names.
2. After this, we have used **list()** in which we passed the set 'subjects'.
3. On executing the program, the desired output is displayed.

## Conclusion

In this tutorial, we came across the different approaches of converting a set to a list in Python.

Python Program to Find Number of Days Between Two Given Dates

In this tutorial, we will discuss how to write a Python program to find the number of days between two given numbers.

Suppose we have given two dates our expected output would be:

**Example:**

1. Input: Date\_1 = 12/10/2021, Date\_2 = 31/08/2022
2. Output: Number of Days between the given Dates are: 323 days
3. Input: Date\_1 = 10/09/2023, Date\_2 = 04/02/2025
4. Output: Number of Days between the given Dates are: 323 days: 513 days

Method 1: Naïve Approach

In this approach, the naïve solution will start from date\_1, and it will keep counting the number of days until it reaches date\_2. This solution will require more than **O(1)** times. It is a simple solution for counting the total number of days before date\_1, which means it will count total days from 00/00/0000 to date\_1, then it will count the total number of days before date\_2. At last, it will return the difference between the two counts in the form of total days between the two given dates.

**Example:**

1. # First, we will create a **class** **for** dates
2. **class** date\_n:
3. def \_\_init\_\_(self, day, month, year):
4. self.day = day
5. self.month = month
6. self.year = year

9. # For storng number of days in all months from
10. # January to December.
11. month\_Days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
13. # This function will count the number of leap years from 00/00/0000 to the #given date

16. def count\_Leap\_Years(day):
18. years = day.year
20. # Now, it will check **if** the current year should be considered **for** the count          # of leap years or not.
21. **if** (day.month <= 2):
22. years -= 1
24. # The condition **for** an year is a leap year: **if** te year is a multiple of 4, and a            # multiple of 400 but not a multiple of 100.
25. **return** **int**(years / 4) - **int**(years / 100) + **int**(years / 400)

28. # This function will **return** number of days between two given dates
29. def get\_difference(date\_1, date\_2):
31. # Now, it will count total number of days before first date "date\_1"
33. # Then, it will initialize the count by using years and day
34. n\_1 = date\_1.year \* 365 + date\_1.day
36. # then, it will add days **for** months in the given date
37. **for** K in range(0, date\_1.month - 1):
38. n\_1 += month\_Days[K]
40. # As every leap year is of 366 days, we will add
41. # a day **for** every leap year
42. n\_1 += count\_Leap\_Years(date\_1)
44. # SIMILARLY, it will count total number of days before second date "date\_2"
46. n\_2 = date\_2.year \* 365 + date\_2.day
47. **for** K in range(0, date\_2.month - 1):
48. n\_2 += month\_Days[K]
49. n\_2 += count\_Leap\_Years(date\_2)
51. # Then, it will **return** the difference between two counts
52. **return** (n\_2 - n\_1)

55. # Driver program
56. date\_1 = date\_n(12, 10, 2021)
57. date\_2 = date\_n(30, 8, 2022)
59. print ("Number of Days between the given Dates are: ", get\_difference(date\_1, date\_2), "days")

**Output:**

Number of Days between the given Dates are: 322 days

Method 2: By using Python datetime module

In this method, we will see how we can use a built\_in function of Python "datetime", which can help the users in solving various date-time related problems. For finding the difference between two dates, we can input the two dates in a date type format and subtract them, and this will result in output as the number of days between the two given dates.

**Example:**

1. from datetime **import** date as date\_n
3. def number\_of\_days(date\_1, date\_2):
4. **return** (date\_2 - date\_1).days
6. # Driver program
7. date\_1 = date\_n(2023, 9, 10)
8. date\_2 = date\_n(2025, 2, 4)
9. print ("Number of Days between the given Dates are: ", number\_of\_days(date\_1, date\_2), "days")

**Output:**

Number of Days between the given Dates are: 513 days

Conclusion

In this tutorial, we have discussed two different methods of how to write a python code to find the total number of days between the two given dates.

# Convert string to dictionary in Python

We have worked on different problems based on strings and dictionaries. In this tutorial, we will see how we can convert a string to a dictionary in Python.

Before that, let's have a quick recall of strings and dictionaries.

Strings are defined as a sequence of characters and are denoted using single or double inverted commas.

**For example-**

1. flower = 'Rose'
2. sub = 'Python'
3. name = 'James'

We can check the data type of the above variables using **type().**

Dictionaries are defined as a data structure in Python that uses key-value pairs which are enclosed in curly braces.

We can access the values present in the dictionary with the help of respective keys.

The example of a dictionary is-

1. Subj = {'subj1': 'Computer Science', 'subj2': 'Physics', 'subj3': 'Chemistry', 'subj4': 'Mathematics'}

Now let's list out the methods that can convert a string to a dictionary.

1. Using **loads()**
2. Using **literal\_eval()**
3. Using Generator Expressions

It's time to discuss each one of them in detail-

### Using json.loads()

The following program shows how we can convert a string to a dictionary using json.loads()

1. #using json()
2. **import** json
3. #initialising the string
4. string\_1 = '{"subj1":"Computer Science","subj2":"Physics","subj3":"Chemistry","subj4":"Mathematics"}'
5. print("String\_1 is ",string\_1)
6. #using json.loads()
7. res\_dict=json.loads(string\_1)
8. #printing converted dictionary
9. print("The resultant dictionary is ",res\_dict)

**Output:**

String\_1 is {"subj1":"Computer Science","subj2":"Physics","subj3":"Chemistry","subj4":"Mathematics"}

The resultant dictionary is {'subj1': 'Computer Science', 'subj2': 'Physics', 'subj3': 'Chemistry', 'subj4': 'Mathematics'}

**Explanation:**

Let's understand what we have done in the above program-

1. In the first step, we have imported the json module.
2. After this, we have initialized the string that we would like to convert.
3. Now we have simply passed 'string\_1' as a parameter in **loads().**
4. Finally, in the last step, we have displayed the resultant dictionary.

### Using ast.literal\_eval()

Now we will see how **ast.literal\_eval** can help us to meet our objective.

The following program illustrates the same-

1. #convert string to dictionary
2. #using ast()
3. **import** ast
4. #initialising the string
5. string\_1 = '{"subj1":"Computer Science","subj2":"Physics","subj3":"Chemistry","subj4":"Mathematics"}'
6. print("String\_1 is ",string\_1)
7. #using ast.literal\_eval
8. res\_dict=ast.literal\_eval(string\_1)
9. #printing converted dictionary
10. print("The resultant dictionary is ",res\_dict)

**Output:**

String\_1 is {"subj1":"Computer Science","subj2":"Physics","subj3":"Chemistry","subj4":"Mathematics"}

The resultant dictionary is {'subj1': 'Computer Science', 'subj2': 'Physics', 'subj3': 'Chemistry', 'subj4': 'Mathematics'}

**Explanation:**

Let's understand what we have done in the above program-

1. In the first step, we have imported the ast module.
2. After this, we have initialized the string that we would like to convert.
3. Now we have simply passed 'string\_1' as a parameter in **literal\_eval().**
4. Finally, in the last step, we have displayed the resultant dictionary.

## Using Generator Expressions

Finally, in the last example we will discuss how generator expressions can be used.

Let's study the given program carefully.

1. #convert string to dictionary
2. #using generator expressions
3. #initialising the string
4. string\_1 = "subj1 - 10 , subj2 - 20, subj3 - 25, subj4 - 14"
5. print("String\_1 is ",string\_1)
6. #using strip() and split()
7. res\_dict = dict((a.strip(), **int**(b.strip()))
8. **for** a, b in (element.split('-')
9. **for** element in string\_1.split(', ')))
10. #printing converted dictionary
11. print("The resultant dictionary is: ", res\_dict)
12. print(type(res\_dict))

**Output:**

String\_1 is subj1 - 10 , subj2 - 20, subj3 - 25, subj4 - 14

The resultant dictionary is: {'subj1': 10, 'subj2': 20, 'subj3': 25, 'subj4': 14}

<class 'dict'>

It's time to check the explanation of this approach-

1. In the first step, we have declared a string that has values paired with a hyphen, and each pair is separated with a comma. This information is important since it will act as a great tool in obtaining the desired output.
2. Further, we have used **strip()** and **split()** in the for loop so that we get the dictionary in the usual format.
3. Finally, we have printed the dictionary we created and verified its type using **type().**

## Conclusion

In this tutorial, we explored the conversion methods of string to the dictionary.

# Operator Module in Python

In this tutorial, we will learn about the operator module in Python and its various functions. We will use these functions of the operator module in a Python program to demonstrate their work.

## Python Operator Module

As the name suggests, the operator module performs various operations and operates two input numbers in a Python program. The Python operator module is one of the inbuilt modules in Python, and it provides us with a lot of functions such as add(x, y), floordiv(x, y) etc., which we can use to perform various mathematical, relational, logical and bitwise operations on two input numbers. We will learn about some of these functions of the operator module in this section and will use each of them in a Python program to understand their functioning. We have categorised the operator functions into categories according to their functioning.

### Mathematical Operation Functions:

Here, we will look at some of the major operator functions we can use to perform mathematical operations such as addition, subtraction, division etc., on two input given values. Let's have a look at the following functions where we understand them in brief by using them in a program:

**1. add(x, y):** We can use add(x, y) function of the operator module to add two input given numbers, whereas x and y are two input values.

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

Operator module performs **"x + y"** operation in this function.

**Syntax:**

1. operator.add(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Adding both input numbers
7. addResult = operator.add(x, y)
8. # Print result
9. print("Addition of input numbers given by you is: ", addResult)

**Output:**

Enter first integer number: 234

Enter second integer number: 729

Addition of input numbers given by you is: 963

**Explanation:** After importing the operator module in the program, we added the two user input numbers by using the add() function and giving these input numbers as arguments. Then, we printed the addition of two numbers as a result of the program in the output.

**2. sub(x, y):** As the name suggests, the sub() function of operation is used to perform subtraction operation, and we can subtract two numbers by giving them as an argument inside the sub() function.

Operator module performs **"x - y"** operation in this function.

**Syntax:**

1. operator.sub(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Subtracting both input numbers
7. subResult = operator.sub(x, y)
8. # Print result
9. print("Subtraction of input numbers given by you is: ", subResult)

**Output:**

Enter first integer number: 727

Enter second integer number: 344

Subtraction of input numbers given by you is: 383

**Explanation:** We have subtracted the user input numbers by giving them as arguments inside the sub() function of the operation module and printed the subtraction result in the output.

**3. mul(x, y):** We can use the mul(x, y) function to get the multiplication of two input given numbers, whereas x and y are two input values.

Operator module performs **"x \* y"** operation in this function.

**Syntax:**

1. operator.mul(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Multiply both input numbers
7. mulResult = operator.mul(x, y)
8. # Print result
9. print("Multiplication result of numbers given by you is: ", mulResult)

**Output:**

Enter first integer number: 27

Enter second integer number: 23

Multiplication result of numbers given by you is: 621

**Explanation:** We have multiplied the user input numbers by giving them as arguments inside the mul() function of the operation module and printed the result.

**4. truediv(x, y):** The truediv() function of the operator module is used to get the exact division value or result of the two numbers, i.e., x and y, which is given as arguments in it.

Operator module performs **"x / y"** operation in this function.

**Syntax:**

1. operator.truediv(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Divide both input numbers
7. truedivResult = operator.truediv(x, y)
8. # Print result
9. print("True division result of numbers given by you is: ", truedivResult)

**Output:**

Enter first integer number: 25

Enter second integer number: 6

True division result of numbers given by you is: 4.166666666666667

**Explanation:** We divided the user input numbers by giving them as arguments inside the truediv() function to get the exact true division result and printed it.

**5. floordiv(x, y):** As the function's name suggests, the floordiv() function is also used to divide two numbers, but it performs floor division on them and returns the floored value, i.e., greatest small integer.

Operator module performs **"x // y"** operation in this function.

**Syntax:**

1. operator.floordiv(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Perform floor division
7. floordivResult = operator.floordiv(x, y)
8. # Print result
9. print("Floor division result of numbers given by you is: ", floordivResult)

**Output:**

Enter first integer number: 25

Enter second integer number: 6

Floor division result of numbers given by you is: 4

**Explanation:** We have performed the floor division on the user input numbers by giving them as arguments inside the floordiv() function to get the floored value as the division result and printed it.

**6. mod(x, y):** The mod(x, y) is used to get the modulus of two numbers by giving them as arguments inside the function.

Operator module performs **"x % y"** operation in this function.

**Syntax:**

1. operator.mod(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Perform modulus operation
7. modResult = operator.mod(x, y)
8. # Print modulus result
9. print("Result of modulus operation on numbers given by you is: ", modResult)

**Output:**

Enter first integer number: 17

Enter second integer number: 13

Result of modulus operation on numbers given by you is: 4

**Explanation:** We performed the modulus operation on the user input numbers by giving them as arguments inside the mod() function and printed the modulus result.

**7. pow(x, y):** To get the power value of a function, we use the pow(x, y) function where y is treated as the power of x.

Operator module performs **"x \*\* y"** operation in this function.

**Syntax:**

1. operator.pow(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter an integer number: "))
5. y = **int**(input("Enter power value for the integer number: "))
6. # Perform power value operation
7. powResult = operator.pow(x, y)
8. # Print exponentiation result
9. print("Exponentiation result of power value of number given by you is: ", powResult)

**Output:**

Enter an integer number: 5

Enter power value for the integer number you gave: 5

Exponentiation result of power value of number given by you is: 3125

### Relational Operation Functions:

We will look at some more functions from the operator module, but these functions belong to the relational operation category. With these functions, we can establish a relationship between any two given input numbers such as smaller one, larger one, equal etc. Look at the following operator functions from relational category with the use of each of them in a Python program:

**8. lt(x, y):** We can use this function to compare if the first number (x) given in the argument is smaller than the second input number, i.e., y. lt(x, y) will return 'true' in the output if x is smaller than y; otherwise, it will return false as a result.

Operator module performs **"a < b"** operation in this function.

**Syntax:**

1. operator.lt(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Comparing both input numbers
7. fResult = operator.lt(x, y)
8. # Print result
9. print("Is first number given by you is smaller than the second number: ", fResult)

**Output:**

Enter first integer number: 24

Enter second integer number: 26

Is first number given by you is smaller than the second number: True

**Explanation:** After taking both numbers as user input, we have compared them and checked if the first number given by the user is smaller than the second number or not. Then, we printed the comparison result as true or false in the output statement.

**9. le(x, y):** We can use the le(x, y) function of the operator module to establish a relationship between x & y and to check if x (first number) is smaller or equal to y (second number) or not.

Operator module performs **"a <= b**" operation in this function.

**Syntax:**

1. operator.le(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Comparing both input numbers
7. fResult = operator.le(x, y)
8. # Print result
9. print("Is first number given by you is smaller or equal to the second number: ", fResult)

**Output:**

Enter first integer number: 23

Enter second integer number: 19

Is first number given by you is smaller or equal to the second number: False

**10. gt(x, y):** The gt(x, y) is used to compare the two input numbers and check if the first number in the argument (x) is greater than the second number (y) or not, and it also results in the form of True or False only.

Operator module performs **"a > b"** operation in this function.

**Syntax:**

1. operator.gt(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Comparing both input numbers
7. fResult = operator.gt(x, y)
8. # Print result
9. print("Is first number given by you is greater than the second number: ", fResult)

**Output:**

Enter first integer number: 38

Enter second integer number: 49

Is first number given by you is greater than the second number: False

**11. ge(x, y):** We can use the ge(x, y) function of the operator module to establish a relationship between x & y and to check if x (first number) is greater or equal to y (second number) or not.

Operator module performs **"a => b"** operation in this function.

**Syntax:**

1. operator.ge(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Comparing both input numbers
7. fResult = operator.ge(x, y)
8. # Print result
9. print("Is first number given by you is greater or equal to the second number: ", fResult)

**Output:**

Enter first integer number: 23

Enter second integer number: 21

Is first number given by you is greater or equal to the second number: True

**12. eq(x, y):** The eq(x, y) is used to establish a relationship between the two input numbers and check if the first number in the argument (x) is equal to the second number (y), and it also results in the form of True or False only.

Operator module performs **"a = b"** operation in this function.

**Syntax:**

1. operator.eq(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Comparing both input numbers
7. fResult = operator.eq(x, y)
8. # Print result
9. print("Is both input numbers given by you are equal: ", fResult)

**Output:**

Enter first integer number: 24

Enter second integer number: 26

Is both input numbers given by you are equal: False

**13. ne(x, y):** The ne(x, y) function works exactly opposite to that of the eq(x, y) function, i.e., it checks if both the numbers are given in the argument are not equal, and then it yields result in the form of true and false.

Operator module performs **"a != b**" operation in this function.

**Syntax:**

1. operator.ne(x, y)

**Example:**

1. # Import operator module
2. **import** operator
3. # Take two input numbers from user
4. x = **int**(input("Enter first integer number: "))
5. y = **int**(input("Enter second integer number: "))
6. # Comparing both input numbers
7. fResult = operator.ne(x, y)
8. # Print result
9. print("Is both input numbers given by you are not equal: ", fResult)

**Output:**

Enter first integer number: 7

Enter second integer number: 9

Is both input numbers given by you are not equal: True

## Conclusion

# Defaultdict in Python

The dictionary is an unordered collection of data values in Python used for storing data values such as maps. The dictionary holds key-value pairs instead of holding a single value as an element like other data types. The key implemented in the dictionary must be unique and immutable. That is, the Python tuple can be a key, but the Python list cannot be a key in the dictionary. We can create a dictionary by placing a sequence of elements inside the curly brackets {}, a comma "," can separate the values.

### Example 1:

1. Dict\_1 = {1: 'A', 2: 'B', 3: 'C', 4: 'D'}
2. print ("Dictionary: ")
3. print (Dict\_1)
4. print ("key pair 1: ", Dict\_1[1])
5. print ("key pair 3: ", Dict\_1[3])

**Output:**

Dictionary:

{1: 'A', 2: 'B', 3: 'C', 4: 'D'}

key pair 1: A

key pair 3: C

But if we try to print the 5th key value then, we will get the error because **"Dict\_1"** does not contain the 5th key value.

### Example 2:

1. Dict\_1 = {1: 'A', 2: 'B', 3: 'C', 4: 'D'}
2. print ("Dictionary: ")
3. print (Dict\_1)
4. print ("key pair 5: ", Dict\_1[5])

**Output:**

Dictionary:

{1: 'A', 2: 'B', 3: 'C', 4: 'D'}

---------------------------------------------------------------------------

KeyError Traceback (most recent call last)

in

2 print ("Dictionary: ")

3 print (Dict\_1)

----> 4 print ("key pair 5: ", Dict\_1[5])

KeyError: 5

Whenever the **keyError** is raised, it may become a problem for the users. We can overcome this error by using another dictionary of Python, which is like a container known as **Defaultdict**. The users can find this dictionary inside the **'collections'** module.

## defaultdict

The defaultdict is a dictionary of Python, which is like a container present inside the 'collections' module. It is a sub-class of the dictionary class which is used for returning the dictionary-like object. Both defaultdict and dictionary have the same functionality, except defaultdict never raise any KeyError as it provides a default value for the Key, which does not exist in the dictionary created by the user.

**Syntax:**

1. defaultdict(default\_factory)

**Parameters:**

* **default\_factory:** The default\_factory() function returns the default value set by the user for the dictionary defined by them. If this argument is absent, then the dictionary will raise the KeyError.

**Example:**

1. from collections **import** defaultdict as DD
2. # Function **for** returning a **default** values **for** the
3. # keys which are not present in the dictionary
4. def default\_value():
5. **return** "This key is not present"
7. # Now, we will define the dict
8. dict\_1 = DD(default\_value)
9. dict\_1["ABC"] = 1
10. dict\_1["DEF"] = 2
11. dict\_1["GHI"] = 3
12. dict\_1["JKL"] = 4
13. print ("Dictionary: ")
14. print (dict\_1)
15. print ("key pair 1: ", dict\_1["ABC"])
16. print ("key pair 3: ", dict\_1["GHI"])
17. print ("key pair 5: ", dict\_1["MNO"])

**Output:**

Dictionary:

defaultdict(, {'ABC': 1, 'DEF': 2, 'GHI': 3, 'JKL': 4})

key pair 1: 1

key pair 3: 3

key pair 5: This key is not present

### Inner Working of defaultdict

When we use defaultdict, we get an additional writable instance variable and one method in addition to the standard dictionary operations. The writable instance variable is the default\_factory parameter and **\_\_missing\_\_** is the method.

* **default\_factory:** The default\_factory() function returns the default value set by the user for the dictionary defined by them.

**Example:**

1. from collections **import** defaultdict as DD
2. dict\_1 = DD(lambda: "This key is not present")
3. dict\_1["ABC"] = 1
4. dict\_1["DEF"] = 2
5. dict\_1["GHI"] = 3
6. dict\_1["JKL"] = 4
7. print ("Dictionary: ")
8. print (dict\_1)
9. print ("key value 1: ", dict\_1["ABC"])
10. print ("key value 3: ", dict\_1["GHI"])
11. print ("key value 5: ", dict\_1["MNO"])

**Output:**

Dictionary:

defaultdict( at 0x0000019EFC4B58B0>, {'ABC': 1, 'DEF': 2, 'GHI': 3, 'JKL': 4})

key value 1: 1

key value 3: 3

key value 5: This key is not present

* **\_\_missing\_\_():** The \_\_missing\_\_() function is used for providing the default value to the dictionary. The \_\_missing\_\_() function takes default\_factory as an argument, and if the argument is set to None, a KeyError will raise; otherwise, it will provide a default value for the given key. This method is essentially called by the **\_\_getitem\_\_()** function of the dict class when the requested key is not found. The \_\_getitem\_\_() function raise or return the value which is present in the \_\_missing\_\_() function.

**Example:**

1. from collections **import** defaultdict as DD
2. dict\_1 = DD(lambda: "This key is not present")
3. dict\_1["ABC"] = 1
4. dict\_1["DEF"] = 2
5. dict\_1["GHI"] = 3
6. dict\_1["JKL"] = 4
7. print ("Dictionary: ")
8. print (dict\_1)
9. print ("key value 1: ", dict\_1.\_\_missing\_\_('ABC'))
10. print ("key value 4: ", dict\_1["JKL"])
11. print ("key value 5: ", dict\_1.\_\_missing\_\_('MNO'))

**Output:**

Dictionary:

defaultdict( at 0x0000019EFC4B5670>, {'ABC': 1, 'DEF': 2, 'GHI': 3, 'JKL': 4})

key value 1: This key is not present

key value 4: 4

key value 5: This key is not present

### How to Use "List" as default\_factory

We can pass a list class as the default\_factory argument, and it will create a defaultdict with the values that are set in list format.

**Example:**

1. from collections **import** defaultdict as DD
2. # Defining a dictionary
3. dict\_1 = DD(list)
5. **for** k in range(7, 12):
6. dict\_1[k].append(k)
8. print ("Dictionary with values as list:")
9. print (dict\_1)

**Output:**

Dictionary with values as list:

defaultdict(<class 'list'>, {7: [7], 8: [8], 9: [9], 10: [10], 11: [11]})

### How to Use "int" as default\_factory

We can pass the int class as the default\_factory argument, and it will create a defaultdict with the default value set as zero.

**Example:**

1. from collections **import** defaultdict as DD

4. # Defining the dict
5. dict\_1 = DD(**int**)
7. J = [1, 2, 3, 4, 2, 4, 1, 2]
9. # Now, we will iterate through the list "J"
10. # **for** keeping the count
11. **for** k in J:
13. # As, The **default** value is 0
14. # so we **do** not need to
15. # enter the key first
16. dict\_1[k] += 1
18. print(dict\_1)

**Output:**

defaultdict(<class 'int'>, {1: 2, 2: 3, 3: 1, 4: 2})

## Conclusion

In this tutorial, we have discussed defaultdict in Python and how we can perform different operations on defaultdict by using the default\_factory parameter.

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-

[](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.

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**

[](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 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**

[](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 Program to Display Calendar of Given Year

In this tutorial, we will learn how to display the calendar of any month of any year using Python.

In the code below, we will import the "calendar" module. It has an inbuilt "month()" function, which takes the year and month of which the user wants to display the calendar.

**Example:**

1. **import** calendar
3. year = **int**(input ("Please enter the Year: ")) # Here, it will take the year
4. month = **int**(input ("Please enter the month: "))    # Here, it will take the month
6. # Now, we will display the calendar
7. Print ("The Calendar of: ", calendar.month(year, month))

<

**Output:**

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

**Case -1**

Please enter the Year: 1919

Please enter the month: 01

The Calendar of: January 1919

Mo Tu We Th Fr Sa Su

1 2 3 4 5

6 7 8 9 10 11 12

13 14 15 16 17 18 19

20 21 22 23 24 25 26

27 28 29 30 31

**Case -2**

Please enter the Year: 2022

Please enter the month: 10

The Calendar of: October 2022

Mo Tu We Th Fr Sa Su

1 2

3 4 5 6 7 8 9

10 11 12 13 14 15 16

17 18 19 20 21 22 23

24 25 26 27 28 29 30

31

Conclusion

In this tutorial, we have discussed how a user can display the calendar in Python.

# Difference Between Class-based views and Function-Based Views

Django is one of a most popular framework for the web development. The reason of its popularity is that it provides the many built-in features that save lots of time of a developer. If you are beginners then it will take some time to understand the flow to the project. But once you get some experience then you can easily understand the execution of features.

Django is based on the [MVT](https://www.javatpoint.com/django-mvt) pattern means Model View Template. We mainly use the two-types of view - function-Based views and class based views. If you are beginners, you must be aware from the function based views (FBV).

When [Django](https://www.javatpoint.com/django-tutorial) was introduced, it only supported the function-based view but later Django added the concept of the class-based views. The class-based views reduce the redundancy of the code. It also supports the DRY concept of the Django.

In this tutorial, we will learn in-depth how the FBV and CBC are different from each other. We can perform the same task using the function-based views and class-based views. But both approaches are different.

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

## Requirements of Django Views

Below are the points that we should remember about the Django views.

* The first positional argument of the view should be
* It should return the **HttpResponse** object, or it should raise an exception.
* Django views are callable. We can use both methods either function-based or class-based. While using CBVs we inherit the method **as\_view()** that uses the **dispatch()** method to call the method appropriate depending on the HTTP requests.

Let's have a detailed explanation of way of creating views.

## Function-based Views

Function-based views are simple to use and beginners can easy understand them. It helps to understand the core concept of the Django fundamentals. FBV provides the advantages to understand the django concept from scratch.

Django project usually have the CRUD operations, so we need to implement same code for the multiple times unnecessarily and that's why the Django class-based views come into the scenario. The class based-views are created to solving the code redundancy problems. Let's understand the following pros and cons of the FBV.

### Advantages of Function Based Views

The advantages of FBV are given below.

* Easy to implement, read and understand
* Explicit code flow
* Decorator can be implemented easily
* Good for one-off or specialized functionality
* Helps to understand the core concept of the Django.

### Disadvantages of Function Based Views

The cons of FBV are given below.

* Code Redundancy is the biggest concern of FBV.
* Condition branching will be used handle HTTP request.
* Hard to extend the code.

Let's understand the following implementation of FBV.

**Code**

1. def  create\_view(request, pk):
2. template\_name = 'form.html'
3. form\_class = MyForm
5. form = form\_class
7. **if** request.method == 'POST':
8. form = form\_class(request.POST)
9. **if** form.is\_valid():
10. form.save()
11. **return** HttpResponseRedirect(reverse('list-view'))
13. **return** render(request, template\_name, {'form': form})

## Class Based Views

Class-based views are advance way to create views in Python. It is implemented in the project as Python objects instead of functions. It is not a substitute of FBV, but they provide advantages over the function based views. It reduces the code repetition and takes cares of the basic operations such as deleting and adding item.

It is slightly hard to get the concept of class-based views for beginners. You should go through the documentation, and you will have study properly. If you have a clear idea about the function based views, you can move to the class based views.

Let's understand the following pros and cons of the views in Python.

### Advantages of Class Based Views

Below are the advantages of the class based views.

* One of the biggest advantages of CBV is inheritance. CBV allows us to inherit another class and can be modified for the different use cases.
* It supports the DRY principle. It prevents the repetition of the code. Code reusability is possible in the class based views.
* It comes with the built-in generic class-based views.
* The class based views provides the proper code structuring. We can use the different class instance method (instead of conditional branching statement inside function-based-views) to generate the HTTP requests.

### Disadvantages of Class Based Views

Below are the disadvantages of the class based views.

* Hard to understand the complex to implement.
* Implicit code flow.
* Extra import or method override required in view decorators.

Let's see the implementation of the class based view.

1. **class** MyCreateView(View):
2. template\_name = 'form.html'
3. form\_class = MyForm
5. def get(self, request, \*args, \*\*kwargs):
6. form = self.form\_class
7. **return** render(request, template\_name, {'form': form})
9. def post(self, request, \*args, \*\*kwargs):
10. form = self.form\_class(request.POST)
11. **if** form.is\_valid():
12. form.save()
13. **return** HttpResonseRedirect(reverse('list-view'))
14. **else**:
15. **return** render(request, self.template\_name, {'form': form})

We call the as\_view() method to serve the request to the user. The as\_view() method calls the dispatch() method to determine which class method needs to executed, depending on the HTTP request. We can implement it in following way.

1. urlpatterns = [
2. url(**new**/', MyCreateView.as\_view(), name='original-create-view'),
3. url(new\_two/', MyCreateView.as\_view(), name='modified-create-view')
4. ]

When we use the Django generic class-based views, we can over-write the helper method like get\_form\_class and get\_template\_names. We can also add some custom logic at these points instead of just overriding the class attributes.

ModelFormMixin is one of the best examples. The **form\_valid** method is overridden with the updated value stored in self.object form\_valid method is overridden.

## Django Generic Class-Based View

With the help of generic class based view, we can perform some important task such as creating a new object, list views, pagination, form handling, archive views, delete view, etc.

We can implement this by importing the **django.views.generic.** Generic class-based views are excellent way to perform some essential tasks. It speeds up the development process.

Django provides a set of mixins, and generic class-based views. With the help of these tools we can solve the most common tasks in web development.

It saves us to writing same code again and again. In the below example, we can modify **MyCreateView** to inherit from **django.views.generic.CreateView**.

1. from django.views.generic **import** CreateView
2. **class** FirstCreateView(CreateView):
3. model = MyModel
4. form\_class = MyForm

As we can observe that, it takes very less code in comparison of previous views. The django.views.generic.CreateView comes with the lots of built-in functionality and shortcuts. Let's discuss few more details.

By default template should reside in /<modelname>/<modelname>\_form.html. We can modify it by setting the class attributes **template\_name** and **template\_name\_sufix**.

* In the CreateView class, we need to specify the model name and form\_class\_attributes.
* We also need to specify the success\_url that will redirect to that mentioned page after successful form submission. It can be done using **get\_absoute\_url().**
* We can specify the fields' class attributes on the view. Below is the example of form fields.

1. from django **import** forms
2. from . models **import** MyModel
3. **class** MyModelForm(forms.ModelForm):
4. **class** Meta:
5. model = MyModel
6. fields = ['name', 'description']

## Conclusion

This is a hot debate among the developers that which views are best to use Class-based or function-based views? We have discussed the use, cons, and pros of both types of the views. It totally depends the project requirement and what you are comfortable for. In some cases, class-based view performs well and in some case function based is better.

# sizeof in Python

When we are writing large scripts or code of many lines, memory management should be our utmost priority. Therefore, we should have good knowledge of handling memory efficiently in addition to good programming knowledge. We have many functions given in Python to get the size in memory of a particular object present in the program, and one of such functions is \_\_sizeof\_\_(). In this tutorial, we will learn about \_\_sizeof\_\_() function and its working inside a Python program.

## Python \_\_sizeof\_\_() function

The \_\_sizeof\_\_() function in Python doesn't exactly tell us the size of the object. It doesn't return the size of a generator object as Python cannot tell us beforehand that how much size of a generator is. Still, in actuality, it returns the internal size for a particular object (in bytes) occupying the memory.

To understand this, let us look at the following example program with an endless generator object.

**Example 1:** Look at the following Python program:

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

1. # A **default** function with endless generator object in it
2. def endlessGenerator():
3. # A counting variable to initialize the generator
4. counting = 0
5. # Using **while** loop to create an endless generator
6. **while** True:
7. yield counting
8. counting += 1 # Creating infinite loop
9. # Printing memory size of a generator object
10. print("Internal memory size of endless generator object: ", endlessGenerator.\_\_sizeof\_\_())

**Output**

Internal memory size of endless generator object: 120

**Explanation:**

We have used a default function, i.e., endlessGenerator(), to create an endless generator object in the program. In the function, we have initialized a variable, i.e., counting = 0. We have used a while loop on the counting variable without giving a breakpoint on the loop. By creating an infinite loop in the function, we have made the default function as an endless generator object. Finally, we have printed the internal memory size of the endless generator object by using the \_\_sizeof\_\_() function.

Now, we can clearly understand the functioning of \_\_sizeof\_\_() function. As the endless generator object in the above program doesn't have any end or breakpoint, Python can't tell us the size of the generator beforehand. But at the same time, we can check the internal memory size allotted to the generator object by \_\_sizeof\_\_() function as it must be occupying some internal memory in Python.

Let's look at one more example where we use the \_\_sizeof\_\_() function to get the internal memory size without any overhead.

### Example 2:

1. # Define an empty list in the program
2. emptyList = []
3. # Printing size of empty list
4. print("Internal memory size of an empty list: ", emptyList.\_\_sizeof\_\_())
5. # Define some lists with elements
6. a = [24]
7. b = [24, 26, 31, 6]
8. c = [1, 2, 6, 5, 415, 9, 23, 29]
9. d = [4, 5, 12, 3, 2, 9, 20, 40, 32, 64]
10. # Printing internal memory size of lists
11. print("Memory size of first list: ", a.\_\_sizeof\_\_())
12. print("Memory size of second list: ", b.\_\_sizeof\_\_())
13. print("Memory size of third list: ", c.\_\_sizeof\_\_())
14. print("Memory size of fourth list: ", d.\_\_sizeof\_\_())

**Output**

Internal memory size of an empty list: 40

Memory size of first list: 48

Memory size of second list: 104

Memory size of third list: 104

Memory size of fourth list: 136

**Explanation:**

Using the \_\_sizeof\_\_() function, we can clearly see that the internal memory size of an empty list is 40 bytes and every element present in the list adds 8 bytes size to the total memory size of the list.

# Python sorted reverse

We have **sorted()** function in Python, and we can use it to sort strings given in input. But what if we have to do reverse sorting of the input string. Can we use this sorted() function to sort in reverse? The answer is Yes. In this tutorial, we will learn how we can use the sorted() function to do reverse sorting of the input string.

## The sorted() function for reverse sorting of string

In Python, strings are immutable data types, and that's why it becomes very difficult for us to do reverse sorting of strings. But, with the help of the **sorted()** function, we can easily perform the task of reverse sorting of strings. For doing reverse sorting with the **sorted()** function, we can use the following methods:

1. With the help of sorted() + reduce() + lambda
2. With the help of join() + sorted() + reverse key

In both the methods mentioned above, we will go through an example program to understand the implementation of these methods.

#### Method 1: With the help of sorted() + reduce() + lambda:

We can perform the reverse sorting of strings by using a combination of reduce() and lambda function with sorted() function. In this method, after operating the input string in reverse order, we will join the resultant reverse sorted characters list using the lambda function.

#### Note: We can use this function in Python 2.x versions only. This is because reduce() function is removed from the Python 3.x versions and will throw an error if we run on Python having 3.x versions.

**Example:**



**Output**

Give an input string for reverse sorting: JAVATPOINT

The input string after reverse sorting is: VTTPONJIAA

### Method 2: With the help of join() + sorted() + reverse key:

By using the combination of join() function and reverse key with sorted() function, we can do the task of reverse sorting very easily in two steps. In the first step of operation, we will get the list of reverse sorted characters, and in the second step, we join these characters to get the reverse sorted string as a result.

**Example:**



**Output**

Give an input string for reverse sorting: JAVATPOINT

The input string after reverse sorting is: VTTPONJIAA

# Different Methods in Python for Swapping Two Numbers without using third variable

In this tutorial, we will discuss different methods used for swapping two variables (n1 and n2) without using a third variable in Python programs.

**Example:**

1. P: 112
2. Q: 211
3. After swapping P and Q:
4. P: 211
5. Q: 112

### Method 1: By using inbuilt method

The inbuilt method can work with any data type values such as string, float, it. This method is very easy to use.

1. Left, Right = Right, Left

**Example:**

1. P = JavaTpoint
2. Q = Tutorial
4. print ("Variables Value Before Swapping: ")
5. print ("Value of P: ", P)
6. print ("Value of Q: ", Q)
8. # Method to swap 'P' and 'Q'
9. P, Q = Q, P
11. print ("Variables Value After Swapping: ")
12. print ("Value of P: ", P)
13. print ("Value of Q: ", Q)

**Output:**

Variables Value Before Swapping:

Value of P: JavaTpoint

Value of Q: Tutorial

Variables Value After Swapping:

Value of P: Tutorial

Value of Q: JavaTpoint

### Method 2: By using Bitwise XOR operator

The Bitwise XOR method only works with integers, and it works faster as it uses bit operation that is for same value result = 0 and for different value result = 1.

1. P ^= Q
2. Q ^= P
3. P ^= Q

**Example:**

1. P = 5 # P = 0101
2. Q = 10 # Q = 1010
4. print ("Variables Value Before Swapping: ")
5. print ("Value of P: ", P)
6. print ("Value of Q: ", Q)
8. # Method to swap 'P' and 'Q'
9. P ^= Q # P = 1111, Q = 1010
10. Q ^= P # Q = 0101, P = 1111
11. P ^= Q # P = 1010, Q = 0101
13. print ("Variables Value After Swapping: ")
14. print ("Value of P: ", P)
15. print ("Value of Q: ", Q)

**Output:**

Variables Value Before Swapping:

Value of P: 5

Value of Q: 10

Variables Value After Swapping:

Value of P: 10

Value of Q: 5

### Method 3: By using Addition and Subtraction Operators

This method can be used only for numerical values.

1. P = P + Q
2. Q = P - Q
3. P = P - Q

**Example:**

1. P = 112
2. Q = 211
4. print ("Variables Value Before Swapping: ")
5. print ("Value of P: ", P)
6. print ("Value of Q: ", Q)
8. # Method to swap 'P' and 'Q'
9. P = P + Q # P = 323, Q = 211
10. Q = P - Q # P = 323, Q = 112
11. P = P - Q # P = 211, Q = 112
13. print ("Variables Value After Swapping: ")
14. print ("Value of P: ", P)
15. print ("Value of Q: ", Q)

**Output:**

Variables Value Before Swapping:

Value of P: 112

Value of Q: 211

Variables Value After Swapping:

Value of P: 112

Value of Q: 211

### Method 4: By using Multiplication and Division Operators

This method can be only used for numerical value except 0.

1. P = P \* Q
2. Q = P / Q
3. P = P / Q

**Example:**

1. P = 11.2
2. Q = 21.1
4. print ("Variables Value Before Swapping: ")
5. print ("Value of P: ", P)
6. print ("Value of Q: ", Q)
8. # Method to swap 'P' and 'Q'
9. P = P \* Q # P = 236.32, Q = 21.1
10. Q = P / Q # P = 236.32, Q = 11.2
11. P = P / Q # P = 21.1, Q = 11.2
13. print ("Variables Value After Swapping: ")
14. print ("Value of P: ", P)
15. print ("Value of Q: ", Q)

**Output:**

Variables Value Before Swapping:

Value of P: 11.2

Value of Q: 21.1

Variables Value After Swapping:

Value of P: 21.1

Value of Q: 11.2

### Method 5: By using both Bitwise operators and Arithmetic operators

In this method, we will use both bitwise operators and arithmetic operators. This method only works with integers and not with float types.

**Example:**

1. P = 112
2. Q = 211
4. print ("Variables Value Before Swapping: ")
5. print ("Value of P: ", P)
6. print ("Value of Q: ", Q)
8. # Same as P = P + Q
9. P = (P & Q) + (P | Q) ;
11. # Same as Q = P - Q
12. Q = P + (~Q) + 1 ;
14. # Same as P = P - Q
15. P = P + (~Q) + 1 ;
17. print ("Variables Value After Swapping: ")
18. print ("Value of P: ", P)
19. print ("Value of Q: ", Q)

**Output:**

Variables Value Before Swapping:

Value of P: 112

Value of Q: 211

Variables Value After Swapping:

Value of P: 211

Value of Q: 112

## Conclusion

In this tutorial, we have discussed different methods used for swapping values of two variables without using the third variable.

# Difference between Class Variable and Instance

## Variable in Python

**Object-oriented programming** lets the developers use variables at the class level or the instance level. Variables are necessary symbols standing in for a value we are utilizing in a program.

Variables at the class level are known as **class variables**, whereas variables at the instance level are known as **instance variables**.

Whenever we expect that the variables are about to be consistent across instances, or whenever we have to initialize a variable, then that variable can be defined at the class level. Whenever we look forward to the variables that will alter significantly across instances, then that variable can be defined at the instance level.

Among various principles of software development is the principle of **DRY**, which is abbreviated for **Don't Repeat Yourself**. This principle is focused towards restrictive replication within the code, and object-oriented programming obeys the DRY principle since it decreases redundancy.

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

In the following tutorial, we will understand the class as well as instance variables in Object-Oriented Programming in the [Python programming language](https://www.javatpoint.com/python-tutorial). We will also discuss the fundamental differences between these two variables.

So, let's get begun.

## Understanding the Class Variables

Class Variables are declared inside the construction of class. Since these variables are owned by the class itself, they are shared by all class instances. They, therefore, will usually have the equivalent value for each instance unless we are utilizing the class variable in order to initialize a variable.

Class Variables are defined outside of all the methods by convention, classically placed right under the header class and before the method of constructor and other functions.

Let us consider the following syntax of a class variable.

**Syntax:**

1. # defining the **class**
2. **class** Class\_name:
3. # declaring the variable in the **class**
4. var = "xyz"

The "**var**" variable is assigned the "**xyz**" value in the above snippet of code.

We can define an object of the **Class\_name** class (we will call it "**myObj**") and print the variable with the help of the dot notation:

**Syntax:**

1. # defining the **class**
2. **class** Class\_name:
3. # declaring the variable in the **class**
4. var = "xyz"
5. # instantiating the **class**
6. myObj = Class\_name()

Let us consider the following example based on the concept of Class Variable.

**Example:**

1. # defining a **class**
2. **class** Animal:
3. # declaring the **class** variable
4. Terrestrial = "Lion"
5. # instantiating the **class**
6. my\_Animal = Animal()
7. # printing the values
8. print("Name of the Animal:", my\_Animal.Terrestrial)

**Output:**

Name of the Animal: Lion

**Explanation:**

In the above snippet of code, we have defined a class as "**Animal**" and declared the class variable. We have then instantiated the class with the **my\_Animal** object and printed the final value for the users. As a result, the program returns the value of the class variable.

Let us try adding multiple class variables to the class and print their values.

**Example:**

1. # defining a **class**
2. **class** Animal:
3. # declaring some **class** variables
4. Terrestrial = "Lion"
5. Location = "Jungle"
6. Type = "Carnivore"
7. Population = 20000
8. # instantiating the **class**
9. my\_Animal = Animal()
10. # printing the values
11. print("Name of the Animal:", my\_Animal.Terrestrial)
12. print("This Animal is found in:", my\_Animal.Location)
13. print("This Animal is a:", my\_Animal.Type)
14. print("Population of this Animal:", my\_Animal.Population, "approx.")

**Output:**

Name of the Animal: Lion

This Animal is found in: Jungle

This Animal is a: Carnivore

Population of this Animal: 20000 approx.

**Explanation:**

In the above snippet of code, we have defined a class and declared some variables to the class. We have then instantiated the class and printed the required output for the users. We can observe that these class variables can contain any data type available to us in Python. As in the above program, we have strings and an integer.

Moreover, we can also observe that the object of **myAnimal** is accessible to all the variables in the class and print them out when we execute the program.

Class variables enable us to define variables upon the construction of the class. These variables and their corresponding values are then accessible to every object of the class.

## Understanding the Instance Variables

Variables that are owned by the class instances are known as **instance variables.** This statement implies that for every instance or object of a class, the instance variables are unlike.

Different from class variables, instance variables are defined within the functions.

The syntax to use the instance variables is shown below.

**Syntax:**

1. # defining the **class**
2. **class** Class\_name:
3. # using the initializing function
4. def \_\_init\_\_(self, var1, var2):
5. self.var1 = var1
6. self.var2 = var2

In the above snippet of code, var1 and var2 are instance variables.

Let us consider an example based on Instance Variables

**Example:**

1. # defining the **class**
2. **class** Student:
3. # using the initializing function
4. def \_\_init\_\_(self, id, name, age):
5. self.id = id
6. self.name = name
7. self.age = age
9. #  instantiating the **class**
10. dBase = Student(102, "Sam", 13)
11. # printing the required values
12. print("Roll Number of the Student:", dBase.id)
13. print("Name of the Student:", dBase.name)
14. print("Age of the Student:", dBase.age)

**Output:**

Roll Number of the Student: 102

Name of the Student: Sam

Age of the Student: 13

**Explanation:**

In the above snippet of code, we have defined a **Student** class and defined some variables as **id, name**, and **age** passed as arguments within the constructor method. We have then instantiated the class and print the values of the instance variables for the users.

As a result, we will obtain a made-up of the values of the variables initialized for the Instance of **dBase**.

Instance variables, owned by the class objects, enable the developers to store different values in each instance assigned to those variables.

## Understanding the difference between the Class Variable and Instance Variable

Since we have understood the basic concepts of both the variables and how these variables are used in the class, let us understand how **class variable** differs from the **instance variable.** The major differences between these two variables are described in the tabular format shown below:

|  |  |  |
| --- | --- | --- |
| **S. No.** | **Class Variable** | **Instance Variable** |
| 1 | A class variable is a variable that defines a particular property or attribute for a class. | An instance variable is a variable whose value is specified to the Instance and shared among different instances. |
| 2 | We can share these variables between class and its subclasses. | We cannot share these variables between classes. However, they only fit in a particular class. |
| 3 | It generally supports a single shared value for every instance of class even if there is no instance object present in the class. | It generally stores memory for data required by the class. |
| 4 | It is usually defined whenever we begin the execution of the program. | It is usually defined whenever we create an instance of the class. |
| 5 | It generally recollects the values until the program ends. | It generally recollects the values as long as the object exists. |
| 6 | It has only one replica of the class variable, so it is shared between various class objects. | It has multiple replicas, so each object has its replica of the instance variable. |
| 7 | We can access these variables by calling with the class name. | We can access these variables directly by calling variable names within the class. |
| 8 | We have to declare these variables with the help of the static keyboard. | We have to declare these variables without utilizing the static keyword. |
| 9 | Whatever alterations we made to these variables via one object will be replicated in another object. | Whatever alterations we made to these variables via one object will not be replicated in another object. |

\_\_add\_\_ Method in Python

In this tutorial, we will discuss operator overloading, its advantages, and how we can overload the '+' operator.

Before discussing about the \_\_add\_\_ method, let us understand what operator overloading is.

Operator overloading enables us to create a definition of the existing operators so that we can use them for user-defined data types as well.

Consider the following examples-

**i) 10+2**

In the first example, the '+' operator is used to add two integers.

**ii) 'JavaTpoint' + 'Tutorials'**

In the second example, the '+' operator is used to concatenate two strings.

**iii) list1+list2**

In the third example, the '+' operator is used to combine the elements of two lists.

This makes it clear that based on the data type, the '+' operator works differently in different situations.

Advantages of Operator Overloading

1. The same operator can be used for different user-defined objects.
2. The syntactic support which can be observed in the case of built-in data types is provided for user-defined types as well.
3. This concept has an essential role in understanding scientific computations.
4. The program looks lucid and easy to comprehend.

It's time to deep dive into the world of applications where we will see how the operator '+' can be overloaded.

The function that we will use for the same is \_\_add\_\_.

In the first program, we will discuss how we can add two complex numbers.

**Example -**

1. class Complex:
2. def \_\_init\_\_(self):
3. self.real=0
4. self.imag=0
5. def decValue(self,real,imag):
6. self.real=real
7. self.imag=imag
8. def \_\_add\_\_(self,C):
9. temp=Complex()
10. temp.real=self.real+C.real
11. temp.imag=self.imag+C.imag
12. return temp
13. def display(self):
14. print("(",self.real, "+", self.imag,"i)")
15. C1=Complex()
16. C1.decValue(10,12)
17. C2=Complex()
18. C2.decValue(8,6)
19. C3=Complex()
20. C3=C1+C2
21. print("The result of the addition of two complex numbers is: ")
22. C3.display()

**Output:**

The result of the addition of two complex numbers is:

( 18 + 18 i)

**Explanation -**

In the above program, we have created a class named 'Complex' and defined some class methods. The first method uses **\_\_init\_\_** which is a class constructor, it is important to use this since it initializes the variables of the class object. Next, we can see the keyword 'self' here which refers to the object itself.

The next method, **decValue** is used to pass the values of real and imag.

Moving on to the next, we will add the two complex numbers where the real part of the first number will be added to the real part of the second number and the imaginary part of the first number will be added to the imaginary part of the second number.

Finally, the display method would display the desired output in the provided format.

1. class Student:
2. def \_\_init\_\_(self,name,marks):
3. self.name=name
4. self.marks=marks
5. def display(self):
6. print(self.name,self.marks)
7. def \_\_add\_\_(self,S):
8. temp=Student(S.name,[])
9. for i in range(len(self.marks)):
10. temp.marks.append(self.marks[i]+S.marks[i])
11. return temp
12. S1=Student("Aarav",[90,91,88])
13. S2=Student("Aarav",[88,86,87])
14. S1.display()
15. S2.display()
16. S3=Student("",[])
17. S3=S1+S2
18. S3.display()

**Output:**

Aarav [90, 91, 88]

Aarav [88, 86, 87]

Aarav [178, 177, 175]

**Explanation -**

In the above program, we have created a class named 'Student' and defined some class methods. The first method uses \_\_init\_\_ which is a class constructor, it is important to use this since it initializes the variables of the class object. Next, we can see the keyword 'self' here which refers to the object itself.

Moving on to the next, we will add the marks of students, for this we have used for loop to append the marks in the list.

Finally, the display method would display the desired output in the provided format.

So, in this article, we learned how we can use **\_\_add\_\_** in our Python program.

Double Underscores in Python

In the following tutorial, we will discuss the **Double Underscores** and its use in the Python programming language. But before we get to that, let us briefly discuss some aspects of the underscore as well.

Understanding the Python Underscore

The character **underscore (\_)** is not simple in Python. There are many languages that use the underscores to name the functions and variables in the snake case only; however, Python has a much more important use of it. Most probably, most of us might be familiar with the following syntax:

1. **for \_ in range(20)**
2. **\_\_init\_\_(self)**
3. **\_ = 10**

The **underscore (\_)** character conveys different meanings in different conditions.

**There are several uses of Underscore (\_) that are stated below:**

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

1. Using the Underscore in interpreter
2. Using the Underscore to ignore values
3. Using the Underscore in loops
4. Using the Underscore to separate digits of numbers
5. Using the Underscore for naming purpose

However, we will only be covering the naming conventions used with double underscores only.

These naming conventions are classified into two types:

1. Double-leading Underscore: **\_\_var**
2. Double-leading and trailing Underscore: **\_\_var\_\_**

Thus, let's get started.

Understanding Double-leading Underscore

**Double-leading underscore** is utilized for the **mangling of the name.**

The syntax for the Double-leading Underscore is shown below:

**Syntax:**

1. \_\_var

Double-leading underscore tells the interpreter of Python to rewrite the name of the attribute of subclasses in order to avoid any conflicts in naming.

**Name Mangling:** Python interpreter alters the name of the variable in a technique that is challenging to clatter during the inheritance of the class.

Let us consider an example based on this functionality.

**Example: 1**

1. # defining a class
2. class mySample():
3. # using the \_\_init\_\_() function
4. def \_\_init\_\_(self):
5. self.first = 10
6. self.\_second = 15
7. self.\_\_third = 20
9. # instantiating the class
10. myObj = mySample()
11. # printing the directory of the object
12. print(dir(myObj))

**Output:**

['\_\_class\_\_',

'\_\_delattr\_\_',

'\_\_dict\_\_',

'\_\_dir\_\_',

'\_\_doc\_\_',

'\_\_eq\_\_',

'\_\_format\_\_',

'\_\_ge\_\_',

'\_\_getattribute\_\_',

'\_\_gt\_\_',

'\_\_hash\_\_',

'\_\_init\_\_',

'\_\_init\_subclass\_\_',

'\_\_le\_\_',

'\_\_lt\_\_',

'\_\_module\_\_',

'\_\_ne\_\_',

'\_\_new\_\_',

'\_\_reduce\_\_',

'\_\_reduce\_ex\_\_',

'\_\_repr\_\_',

'\_\_setattr\_\_',

'\_\_sizeof\_\_',

'\_\_str\_\_',

'\_\_subclasshook\_\_',

'\_\_weakref\_\_',

'\_mySample\_\_third',

'\_second',

'first']

**Explanation:**

In the above snippet of code, we have defined a class as **mySample()** and used the initializing function to declare some values. We have then instantiated the class using the **myObj** object. At last, we have printed the directory of the object.

As a result, the above code block returns every attribute of the class object. Now, let us observe the variables in the list of attributes.

The variable **self.first** appears in the list without any alterations.

The variable **self.\_second** also appears in the list without any alterations.

However, we can observe some alterations in the case of the variable **self.\_\_third.**

If we observe the list of attributes, we will notice an attribute known as **\_mySample\_\_third.** This is the case of **name mangling.** It happens to avoid the overriding of the variable in subclasses.

Let us understand the working of overriding using another example, where we have created another class that inherits the functionalities of the **mySample** class.

**Example:**

1. # defining a class
2. class mySample():
3. # using the \_\_init\_\_() function
4. def \_\_init\_\_(self):
5. self.first = 10
6. self.\_second = 15
7. self.\_\_third = 20
8. # defining a child class
9. class AnotherClass(mySample):
10. def \_\_init\_\_(self):
11. super().\_\_init\_\_()
12. self.first = "Variable Overridden"
13. self.\_second = "Variable Overridden"
14. self.\_\_third = "Variable Overridden"
15. # instantiating the child class
16. myobj = AnotherClass()
17. # printing the values of variables
18. print(myobj.first)
19. print(myobj.\_second)
20. print(myobj.\_\_third)

**Output:**

Variable Overridden

Variable Overridden

Traceback (most recent call last):

File "D:\Python\ternarypy.py", line 24, in

print(myobj.\_\_third)

AttributeError: 'AnotherClass' object has no attribute '\_\_third'

**Explanation:**

In the above snippet of code, we have defined a class as **mySample()** and used the initializing function to declare some variables. We have then defined a child class for the **mySample()** class, where we used the **super()** function to inherit the variables from the parent class and overrode them. At last, we have instantiated the child class and print the values of overrode variables.

As a result, the messages of the first two variables are printed successfully; however, the program raised an exception in the case of the **"\_\_third"** variable. This happens due to the name mangling, which changed the **myObj.\_\_third to \_AnotherClass\_\_third.**

Let us consider another example in order to print that element with the help of a modified Attribute.

**Example: 2**

1. # defining a class
2. class mySample():
3. # using the \_\_init\_\_() function
4. def \_\_init\_\_(self):
5. self.first = 10
6. self.\_second = 15
7. self.\_\_third = 20
8. # defining a child class
9. class AnotherClass(mySample):
11. def \_\_init\_\_(self):
12. super().\_\_init\_\_()
13. self.first = "Variable Overridden"
14. self.\_second = "Variable Overridden"
15. self.\_\_third = "Variable Overridden"
16. # instantiating the child class
17. myobj = AnotherClass()
18. # printing the value(s) of variable(s)
19. print(myobj.\_AnotherClass\_\_third)

**Output:**

Variable Overridden

**Explanation:**

In the above snippet of code, we can observe that we have used the **"\_AnotherClass\_\_third"** variable instead of the **"\_\_third"** variable to access the variable's value.

We can access the Double-leading Underscore variables with the help of methods in the class. Let us consider an example based on this functionality.

**Example: 3**

1. # defining the class
2. class myClass:
3. # initializing function
4. def \_\_init\_\_(self):
5. self.\_\_myVar = "Welcome"
6. # defining another method to return the variable
7. def get\_Var(self):
8. return self.\_\_myVar
9. # instantiating the class
10. myObj = myClass()
11. # it returns the "Welcome" which is a \_\_var
12. print(myObj.get\_Var())
13. # here, an error is raised as stated before. It alters the variable's name
14. print(myObj.\_\_myVar)

**Output:**

Welcome

Traceback (most recent call last):

File "D:\Python\ternarypy.py", line 15, in

print(myObj.\_\_myVar)

AttributeError: 'myClass' object has no attribute '\_\_myVar'

**Explanation:**

In the above snippet of code, we defined a class and used the initializing function to declare a variable. We have then defined a method to return the value of the variable. At last, we have instantiated the class and print the value of the variable using both ways. As a result, the program returns the **"Welcome"** statement while printing the method. However, it also raised an exception for another method as it alters the name of the variable.

We can also utilize the Double-leading Underscore for the method names. Let us consider an example based on this functionality.

**Example: 4**

1. # defining a class
2. class myClass:
3. # defining a double-leading underscore function
4. def \_\_myfunction(self):
5. return "Welcome"
6. # defining a function to call the above function
7. def call\_function(self):
8. return self.\_\_myfunction()
9. # instantiating the class
10. myObj = myClass()
11. # printing the value within the function
12. print(myObj.call\_function())
13. # raised an error
14. print(myObj.\_\_myfunction())

**Output:**

Welcome

Traceback (most recent call last):

File "D:\Python\ternarypy.py", line 14, in

print(myObj.\_\_myfunction())

AttributeError: 'myClass' object has no attribute '\_\_myfunction'

**Explanation:**

In the above snippet of code, we have defined a function followed by the double-leading underscore within a class. We have then defined another function to call the from that function and printed the result for the user.

Now, let us understand another way of name mangling. First of all, we will declare a variable with the name **\_myClass\_\_myVar**, and we will attempt to access that variable using the Double-leading Underscore name.

Let us consider the following example:

**Example: 5**

1. # declaring a variable
2. \_myClass\_\_myVar = "Welcome"
3. # defining a class
4. class myClass:
5. # defining a function to return the declared variable
6. def call\_function(self):
7. return \_\_myVar
8. # instantiating the class
9. myObj = myClass()
10. # printing the value of the variable
11. print(myObj.call\_function())

**Output:**

Welcome

**Explanation:**

In the above snippet of code, we have declared a variable and defined a class. We have then defined a function to return the value of the declared variable. At last, we have instantiated the class and called the function to print the value of that variable.

Understanding Double-leading and trailing Underscore

In a Programming language like Python, we will discover various names which begin and end with the double underscore. These naming conventions are known as **Magic methods** or **Dunder methods.**

The syntax for the Double-leading and trailing Underscore is shown below:

**Syntax:**

1. \_\_var\_\_

Let us consider an example based on the magic methods.

**Example:**

1. # defining a class
2. class myClass:
3. # using a magic method
4. def \_\_init\_\_(self):
5. # using a magic method as variable name
6. self.\_\_myNum\_\_ = 10
7. # instantiating the class
8. myObj = myClass()
9. # printing the value of the variable
10. print(myObj.\_\_myNum\_\_)

**Output:**

10

**Explanation:**

In the above snippet of code, we have defined a class. We have defined a magic method as the **\_\_init\_\_()** function within the class, also known as initializing function. We have then declared the variable as **\_\_num\_\_**. At last, we have instantiated the class and print the value of the variable. As a result, this program works and yields a required output. However, it is not a good practice to use magic methods as the names of the variables as this will lead us to clashes. Thus, it is better to stay away from them.

# Difference between Yield and Return in Python

### Python Yield statement

The generators are defined by using the yield statement in Python. Generally, it converts a normal Python function into a generator.

The yield statement hauls the function and returns back the value to the function caller and restart from where it is left off. The yield statement can be called multiple times. While the return statement ends the execution of the function and returns the value back to the caller. The function returns nothing without it. In Python generators, the yield function is used for replacing the return function that sends back the value to the user without ruining local variables.

### What is Generator?

Python generators are used as a method for generating the iterators. Generators have automatically handled the task. A generator can be defined as the special function that returns an object of the generator to the caller.

In simple words, a generator is referred to the function that returns an iterator that we can iterate upon. In Python, it is very easy to create a generator and it uses the yield statement instead of the return statement.

**Example 1:**

1. # Python Code for using yield statement
2. def myfunction(a, b):
3. aadd = a + b
4. yield add
5. sub = a - b
6. yield sub
7. mul = a \* b
8. yield mul
9. div = a % b
10. yield div
11. # generator runs with for loop for getting all values
12. for value in myfunction(49,45):
13. print(value)

**Output:**

94

4

2205

1.08

**Example 2:**

1. # Python code for using the yield statement
2. def printoutput(String) :
3. for i in String:
4. if i == "a":
5. yield i
6. # string initialization
7. String = "Tutorial and examples"
8. ans = 0
9. print ("The number of 'a' in the string is : ", end = "" )
10. StringString = String.strip()
11. for j in printoutput(String):
12. ansans = ans + 1
13. print (ans)

**Output:**

The number of 'a' in the string is : 3

### Python Return Statement

The return statement is generally used for the execution ending and returns the value back to the caller. The return statement can return all types of values and it returns nothing when there is no expression passed to return statement. There can be multiple return statements in a function but only a single statement is called for any specified invocation of the function.

We can see a return statement is placed at the end of the block of the function for returning the final output of the execution of all statements inside the function. Also, it can be seen earlier in the block of the function for stopping the execution of all consequent statements in the block. The execution of the program at the caller is quickly restarted with a return statement. It returns none when no value is specified.

**Example 1:**

1. # Python code for
2. #demonstrating use of return statement
3. def myfunc(a, b):
4. aadd = a + b
5. sub = a - b
6. mul = a \* b
7. div = a % b
8. return(add, sub, mul, div)
9. # Getting return value and printing the output
10. output = myfunc(49,45)
11. print("Addition: ", output[0])
12. print("Subtraction: ", output[0])
13. print("Multiplication: ", output[0])
14. print("Division: ", output[0])

**Output:**

Addition: 94

Subtraction: 4

Multiplication: 2205

Division: 1.08

**Example 2:**

1. # Python code for
2. #demonstrating use of return statement
3. class check:
4. def \_\_init\_\_(self):
5. self.str = "Tutorial and examples"
6. self.x = "Kaushal"
7. # Object of test will be returned by this function
8. def fun():
9. return check()
10. # Driver code for checking the above method
11. t = fun()
12. print(t.str)
13. print(t.x)

**Output:**

Tutorial and examples

Kaushal

### Difference between yield and return Statement

|  |  |
| --- | --- |
| **Yield Statement** | **Return Statement** |
| Yield returns the generator object. | Returns the results to the caller. |
| It can run multiple times. | It runs only a single time. |
| Code can be executed in the next function call after the yield statement. | Code will not be executed after the return statement. |
| This statement can resume from where it is left off or paused. | The function call in this statement only runs the function from the beginning. |
| The yield statement hauls the function and returns back the value to the function caller. | The return statement takes exit from the execution and returns the value back to the caller. |

### Conclusion

In the above article, we have seen the differences in the return and yield statements. Also, we have understood the concepts of both statements and know how to use them in our Python programs.

# Python Important Tips and Tricks

For all the Python developers or any other language developers, it is always fascinating to know about the tricks and tips of the programming language that we are learning. As we all know that Python is one of the most preferred programming languages among developers. Therefore, in this tutorial, we bring the essential tips and tricks in Python that every Python developer should know.

## 10 Essential Tips and Tricks for Python

The tricks or tips of Python that we are going to discuss here will save us from writing many lines of code and will save a lot of time. These Python tips are tricks that will also help us to bring up our coding game and boost us while competing with other programmers.

Here is the list of such ten essential Python tips and tricks that every Python developer should know:

### 1. Creating a single string from elements of a list:

We can simply create a single string from all the elements of a given list by using "." with the **join() function** inside the print statement with the list variable. So, by this, we can easily get the single string data format from the multiple data elements given in list format.

Play Videox[[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack" \t "_blank)

**Example:**

1. # Given data elements in list format
2. GivenList = ["Hello", "Python", "Developers!", "Welcome", "to", "JavaTpoint"]
3. # printing single string using "." With join() function
4. print(" ".join(GivenList))

**Output:**

Hello Python Developers! Welcome to JavaTpoint

### 2. Python use of Enums:

In Python, we can simply use the Enums to check the number of occurrences of a variable inside the given function where it first occurred. We just have to use the word with the function name inside the print statement with the "." operator to print the number of the first occurrence of that variable inside the function.

**Example:**

1. # define a **class** **for** Enums
2. **class** EnumExample:
3. Hello, Python, Developers, Welcome, to, JavaTpoint, tutorial, of, Python = range(9)
4. # printing first number of occurrences of the given variable
5. print("Occurrence of JavaTpoint: ", EnumExample.JavaTpoint)
6. print("Occurrence of Hello: ", EnumExample.Hello)
7. print("Occurrence of Python: ", EnumExample.Python)
8. print("Occurrence of Welcome: ", EnumExample.Welcome)

**Output:**

Occurrence of JavaTpoint: 5

Occurrence of Hello: 0

Occurrence of Python: 8

Occurrence of Welcome: 3

### 3. Printing path of the imported module:

If we need to print the file directory or path for the Python modules we have imported in the programs, then we just have to use the module name inside the print statement simply, and the file directory will be printed in the output.

**Example:** Looking at the following Python program:

1. # Importing modules in the program
2. **import** socket
3. **import** numpy
4. **import** os
5. # Printing the file directory of module imported in program
6. print(socket)
7. print(numpy)
8. print(os)

**Output:**

<module 'socket' from 'C:\\Users\\Manish\\lib\\socket.py'>

<module 'numpy' from 'C:\\Users\\Manish\\lib\\site-packages\\numpy\\\_\_init\_\_.py'>

<module 'os' from 'C:\\Users\\Manish\\lib\\os.py'>

### 4. Printing the most occurred element of a list:

We have given a list with a number of elements in it, and there are many elements in it that are repeated more than one time. Now, if we want to print the number of elements that are occurred most in the list and it is the same as finding mode in statistics from the given data of numbers. We use max() and count function to get the result of the most occurred element.

**Example:**

1. # A list with the number of elements in it
2. GivenList = [24, 21, 27, 29, 17, 23, 29, 34, 67, 23, 21, 29, 19, 63, 29, 27, 35, 21, 29]
3. # Printing most occurred number or element in list
4. print("Most occurred element in the given list: ", max(set(GivenList), key = GivenList.count))

**Output:**

Most occurred element in the given list: 29

### 5. Printing given string number of times:

We can print a given string 'n' number of times in the output simply by using 'String Name \* n' syntax into the print statement. It will print the given string n number of times in continuation in output.

**Example:**

1. # Define a string and n number
2. GivenString = "Welcome to JavaTpoint, Python developers!"
3. n = 4
4. # Printing string multiple times
5. print("Given string for n number of times: ")
6. print(GivenString \* n)

**Output:**

Given string for n number of times:

Welcome to JavaTpoint, Python developers!Welcome to JavaTpoint, Python developers!Welcome to JavaTpoint, Python developers!Welcome to JavaTpoint, Python developers!

### 6. wapping of two variable numbers:

We can also do in-place swapping of two variables numbers so we can use their swapped values in the program.

**Example:** Look at the following Python program:

1. # Define two number variables
2. m = 24
3. n = 26
4. print("m before swapping: ", m)
5. print("n before swapping: ", n)
6. # In-place swapping variables
7. m, n = n, m
8. print("m after swapping: ", m)
9. print("n after swapping: ", n)

**Output:**

m before swapping: 24

n before swapping: 26

m after swapping: 26

n after swapping: 24

### 7. Using chain of comparison operators:

We can use the chain of comparison operator to compare the given variable number with multiple values in a single comparison.

**Example:**

1. # Defining a number variable
2. num = 31
3. # Chaining comparison operators on num variable
4. Result1 = 35 > num > 30
5. Result2 = 17 > num < 35
6. # Printing result of comparison
7. print(Result1)
8. print(Result2)

**Output:**

True

False

### 8. Reversing a given string:

Sometimes, we have a given string variable, and we may have to print or use the reverse order of that string. Therefore, we should know the easiest way to print the reverse format of a given string.

**Example:** Look at the following Python program:

1. # Define a string variable
2. GivenString = "Welcome to JavaTpoint Python Developers!"
3. print("Given String in program: ", GivenString)
4. # Printing reverse of string in the output
5. print("Reverse of Given string in program is: ", GivenString[::-1])

**Output:**

Given string in program: Welcome to JavaTpoint Python Developers!

Reverse of Given string in program is: !srepoleveD nohtyP tniopTavaJ to emocleW

### 9. eturning multiple values from a single function:

We can print the multiple values or elements from a given single function simply by using only one print statement. It will save a lot of time for us from writing multiple lines of code in the program.

**Example:**

1. # Define a **default** functions
2. def multival():
3. **return** 24, 25, 31, 43, 37, 29, 39, 23
4. # Defining multiple values from multival() function
5. j, k, l, m, n, o, p, q = multival()
6. # Printing multiple values in single statement
7. print(j, k, l, m, n, o, p, q)

**Output:**

24 25 31 43 37 29 39 23

### 10. Checking for anagram words:

Anagram words are the words in which all the letters are the same in two different words, but the letters of words are arranged in a different order within the word. We can check that if the given two words are pair of anagram words or not.

We can perform this action of checking anagram words, by using the following two methods:

**a. Without importing an external module in program:**

Look at the following example Python program:

1. # A **default** function to check anagram word logic
2. def CheckAnagram(mkr1, mkr2):
3. **return** sorted(mkr1) == sorted(mkr2) # logic
4. # Checking anagram words with **default** function
5. print("Words are anagrams: ", CheckAnagram('Python', 'yPotnh'))
6. print("Words are anagrams: ", CheckAnagram('JavaTpoint', 'poijTtavaG'))

**Output:**

Words are anagrams: True

Words are anagrams: False

**b. By importing an external module in program:**

1. # Importing counter from collection module
2. from collections **import** Counter
3. # A **default** function to check anagram word logic
4. def CheckAnagram(mkr1, mkr2):
5. **return** Counter(mkr1) == Counter(mkr2)  # logic
6. # Checking anagram words with **default** function
7. print("Words are anagrams: ", CheckAnagram('Python', 'yPotnh'))
8. print("Words are anagrams: ", CheckAnagram('JavaTpoint', 'poijTtavaG'))

**Output:**

Words are anagrams: True

Words are anagrams: False

# Python program to find compound interest

In this tutorial, we will learn how to write a Python program to find compound interest for a given value. Before we proceed with writing the program, let's first under the basics of compound interest.

## Compound interest

The addition of interest in the given principal value for the deposit or loan is called **compound interest,** and it is also known as interest on interest. Compound interest is basically the result of reinvesting the gained interest on principal value, rather than taking or paying it out, which will result in the next period of interest; the paid interest will be added to the principal amount.

Let's compare compound interest and simple interest. We can see that, in simple interest, there is no compounding of interest on principal values, and gained interest remains the same on every cycle.

**Fact:** Compound interest is the standard method of interest calculation in banking, finance, and economics.

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

### The formula for compound interest:

The general mathematical formula for calculating compound interest on a given principal value, i.e., V, is given below:

**Total Amount** = P(1+r/100)tp

In the above formula, the variables we have used in it can be described as given below:

**Total amount** = Principal value + Compound interest gained

**P** = Principal value

**tp** = Time period for which Principal value is invested

**r** = Rate of interest

## Code Implementation

Till now, we have learned the basics of compound interest and how important it is in our daily life. We have also learned about the basic formula that we can use for calculating the Compound interest for a certain principal value.

Now, in this section, we will write a Python program to calculate the compound interest for a given certain value. For writing the required Python program, we have to follow certain steps, which are given below:

**Step 1:** We will take the principal amount as user input.

**Step 2:** Then, we ask the user to set the rate of interest and the time period for which the principal amount is invested.

**Step 3:** After taking all the three required variables from the user, we will use the compound interest formula, i.e., **"Total Amount = P(1+r/100)tp,"** on these variables in the program.

**Step 4:** We will store the result in a 'Result variable'.

**Step 5:** In the last, we will print the compound interest as a result in the output of the program.

Now, look at the following Python program to better understand the implementation of the above given steps:

**Example -**

1. # Using **default** function
2. def compound\_rate(PV, CRate, tp):
3. # Using CI formula
4. TotalAmount = PV \* (pow ((1 + CRate / 100), tp))
5. # Calculating CI
6. CInterest = TotalAmount - PV
7. # Printing CI as result in output
8. print("Total return value after completion of given time period: ", TotalAmount)
9. print("Compound interest gained on given amount is", CInterest)
11. # Taking principal amount value from the user
12. PV = **float**(input("Enter the principal amount: "))
13. CRate = **float**(input("Enter the rate for compound interest: ")) # taking interest rate value
14. tp = **float**(input("Enter the time period for which principal is invested: ")) # taking time period value
16. # Calling out CI function
17. compound\_rate(PV, CRate, tp)

**Output:**

Enter the principal amount: 600000

Enter the rate for compound interest: 2.7

Enter the time period for which principal is invested: 20

Total return value after completion of given time period: 1022257.0687807774

Compound interest gained on given amount is 422257.0687807774

**Explanation -**

After running the above program, we have given three required variables, i.e., **PV = 600000, Crate = 2.7**, and **tp = 20**; we got the total compound interest and total value (1022257.0687807774) that we will get on the given principal value and printed the result in the output. We can calculate the compound interest on any given amount for any rate of interest and for any time period using this program.

# Python %s - String Formatting

In this tutorial, we will learn about how we can implement and use %s in our Python program. We will also learn the uses of %s in the strings.

## %s in a string format in Python

Basically, the % symbol is used with a large variety of data having many data types and configurations in Python.

Talking about %s, it is specifically used to perform concatenation of two or more strings together in Python. The %s allow us to format or place a string or numerical value within a given string. In simple language, the %s in Python is used to incorporate a given string within another string. This operator automatically provides type conversion of a given value to string data type.

We put the % operator, where we have to specify the string. The numbers of values that we want to append into the given string should be equal to the numbers of %s operators that have been specified in the parentheses between the strings.

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

The following [Python](https://www.javatpoint.com/python-tutorial) code will explain the way by which we perform string formatting using %s operator:

**Example -**

1. # Define a string value
2. str = "JavaTpoint!"
3. # using %s to appending string
4. print("Hello Python developers! Welcome to, %s!" % str)

**Output:**

Hello Python developers! Welcome to, JavaTpoint!

As we can see in the above code, we have declared a string variable, i.e., str, and appended it inside the given string in parenthesis. We appended the str string value using the %s operator.

## Uses of %s operator in Python

In this section, we will discuss the highlighted uses of the %s operator. We will also learn the implementation of the %s operator by multiple approaches and how it is helpful to us.

We will discuss the following approaches of implementing %s:

1. Multiple %s in a string
2. Using %s for mapping string
3. Using %s for a list as a string
4. Order Dictionary with %s

### 1. Multiple %s in a string:

We can use the %s operator to append the given string variable inside a string by putting it where we want to add the value. Python will simply add the string variables where we have used the %s operator in the string. Let's go through an example to understand it.

**Example: Look at the following code:**

1. # Declaring multiple string values
2. mkr1 = "developers"
3. mkr2 = "JavaTpoint"
4. mkr3 = "day"
5. # Appending multiple string values inside a single string
6. print("Hello Python %s, Welcome to the %s! We hope you are having a good %s." % (mkr1, mkr2, mkr3))

**Output:**

Hello Python developers, Welcome to the JavaTpoint! We hope you are having a good day.

**Explanation -**

We have appended multiple string variables (mkr1, mkr2, mkr3) inside a single string and printed the string. We have appended these variables by simply using the %s operator multiple times within the string.

### 2. Using %s for mapping string:

We can also use the %s operator for mapping the string within the given program. We can map multiple string variables into a single string by simply using %s operators multiple times. However, when we want to add multiple string variables in a single string, we have to match the number of occurrences of %s in the string with the number of strings to replace it. We have to add that number of strings after the % operator.

**Example: 1**

1. # Declaring multiple string variables
2. mkr1 = "Hey"
3. mkr2 = "Python"
4. mkr3 = "Developers"
5. mkr4 = "Welcome"
6. mkr5 = "to"
7. mkr6 = "JavaTpoint"
8. # Mapping multiple string variables into a single string
9. ResultantStr = "%s %s %s %s %s %s" % (mkr1, mkr2, mkr3, mkr4, mkr5, mkr6)
10. # Printing result in output
11. print("Resultant mapped string using '%s' operator: ")
12. print(ResultantStr)

**Output:**

Resultant mapped string using '%s' operator:

Hey Python Developers Welcome to JavaTpoint

#### Note: But, if we don't give the same number of string variables after % operator, as the %s operator we have used, then the code will through a TypeError in the output like the following program:

**Example - 2**

1. # Declaring multiple string variables
2. mkr1 = "Hey"
3. mkr2 = "Python"
4. mkr3 = "Developers"
5. mkr4 = "Welcome"
6. mkr5 = "to"
7. mkr6 = "JavaTpoint"
8. # Not giving equal number of variables after % operator
9. ResultantStr = "%s %s %s %s %s %s" % (mkr1, mkr2, mkr4, mkr5, mkr6)
10. # Printing result in output
11. print("Resultant mapped string using '%s' operator: ")
12. print(ResultantStr)

**Output:**

Traceback (most recent call last):

File "C:\Users\Manish\Downloads\code.py", line 9, in

ResultantStr = "%s %s %s %s %s %s" % (mkr1, mkr2, mkr4, mkr5, mkr6)

TypeError: not enough arguments for format string

### 3. Using %s for a list as a string:

With the help of the %s operator, we can also map a given list data type variable with the string variables to form a single string. We have to follow the same set of instructions that we follow while mapping string variables. Let's go through an example to understand it.

**Example:**

1. # Declaring multiple string variables
2. mkr1 = "Hey"
3. mkr2 = "Python"
4. mkr3 = "Developers"
5. mkr4 = "Welcome"
6. mkr5 = "to"
7. mkr6 = "JavaTpoint"
8. # Declaring a list variable also
9. AList = ["Java", "Python", "C++", "HTML", "JavaScript"]
10. # Mapping string variables with list variable into a single string
11. ResultantStr = "%s %s %s %s %s %s, %s" % (mkr1, mkr2, mkr3, mkr4, mkr5, mkr6, AList)
12. # Printing result in output
13. print("Resultant mapped string with list variable in it: ")
14. print(ResultantStr)

**Output:**

Resultant mapped string with list variable in it:

Hey Python Developers Welcome to JavaTpoint, ['Java', 'Python', 'C++', 'HTML', 'JavaScript']

### 4. Order dictionary with %s:

We can define multiple string variables inside a dictionary data type variable. Then, we can call out these string variables by the order we want to, just by using the %s operator with the string variable name.

**Example:**

1. # Declaring a dict variable with multiple string variable in it
2. ADict = {'mkr1': 'at',
3. 'mkr2': 'JavaTpoint',
4. 'mkr3': 'Learning',
5. 'mkr4':'operator',
6. 'mkr5':'concept',
7. 'mkr6': '%s'}
8. # Mapping a string with string variables in dictionary
9. ResultantStr = "%(mkr3)s %(mkr6)s %(mkr4)s %(mkr5)s %(mkr1)s %(mkr2)s" % ADict
10. # Printing result in output
11. print("Resultant mapped string with ordered variable from dictionary: ")
12. print(ResultantStr)

**Output:**

Resultant mapped string with ordered variable from dictionary:

Learning %s operator concept at JavaTpoint

# Python super() Function

As we all know that, Python is an object-oriented programming language. Therefore, Python follows all the concepts of OOPs, and one of such concepts is inheritance.

While using the inheritance concept, we can refer to a parent class with the use of super() function inside the inherited or child class. The super() function we use in the child class returns a temporary created object of the superclass, that allow us to access all of its method present in the child class.

### Benefits of super() function:

Following are the benefits of using a super() function in child class:

* We don't need to remember the parent's class name while using the super() function. This is because we don't have to specify the name of the parent class to access the methods present in it.
* We can use the **super()** function with the single inheritance and multiple inheritances.
* The super() function in Python implements code reusability and modularity as there is no need for us to rewrite the whole function again and again.
* The super() function in Python is known as dynamical function, as we all know that Python is a dynamically typed programming language.

### Constraints of using super() function:

Following are three constraints that we must have to follow to use the super() function in a Python program:

* The arguments are given in the super() function and the arguments in the function that we have called should match.
* Every occurrence of the method that we are using should include the super() keyword after we use it.
* We have to specify the class and methods present in it, which are referred to by the super() function.

Now, as we know that, we can use the super() function in both types of inheritances in Python, i.e., single as well as multiple inheritances. Therefore, we will learn about using the super() function in both types of inheritance separately and with an example.

## Using super() function in single inheritance in Python

In this example, we will use animals as the reference for a [single inheritance](https://www.javatpoint.com/inheritance-in-python) example.

Cats, horses, cows, dogs, etc., all are part of class animalia. They all share some common characteristics also:

* They all are pet animals.
* They all have four legs and a tail.
* As part of class animalia, they all are mammals as well.

So, we can say that the class **cats**, class **horses,** and class **dogs** are the subclasses of the class animalia. This is an example of single inheritance because all the subclasses (class cats, class horses, and class dogs) are inherited from a single parent class only, i.e., class animalia. Now, look at the following program.

**Example -**

1. # Define parent **class** animalia
2. **class** Animalia:
4. # define construcors **for** parent animalia **class**
5. def \_\_init\_\_(self):
6. self.Legs = 4
7. self.adomestic = True
8. self.atail = True
9. self.amammals = True
11. # define mammal **class** as child **class**
12. def aMammal(self):
13. **if** self.amammals:
14. print("The given animal is a mammal type .")
16. # define domestic **class** as child **class**
17. def aDomestic(self):
18. **if** self.adomestic:
19. print("The given animal is a domestic animal type.")
20. # define dog **class**
21. **class** Dog(Animalia):
22. def \_\_init\_\_(self):
23. **super**().\_\_init\_\_() # using **super**() function to access **class** methods
25. def isMammal(self):
26. **super**().aMammal() # using mammal **class**
28. # define cat **class**
29. **class** Cat(Animalia):
30. def \_\_init\_\_(self):
31. **super**().\_\_init\_\_()
33. def isMammal(self):
34. **super**().aDomestic() # using domestic **class**
36. # define horse **class**
37. **class** Horse(Animalia):
38. def \_\_init\_\_(self):
39. **super**().\_\_init\_\_()
41. def TailandLegs(self): # using tail and legs **class**
42. **if** self.atail and self.Legs == 4:
43. print("The given animal has four legs and a tail")
45. # Taking the driver's code **for** defined classes
46. Tommy = Dog()
47. Tommy.aMammal()
48. Tom = Cat()
49. Tom.aDomestic()
50. Burno = Horse()
51. Burno.TailandLegs()

**Output:**

The given animal is a mammal type.

The given animal is a domestic animal type.

The given animal has four legs and a tail.

**Explanation:**

In the above code, we have defined animalia as parent class and inherited domestic, tail & legs, and mammal class from it. After that, we have defined the cat, horse, and dog class and used super function in it. With the help of super() function in these classes, we have accessed methods of animalia class in cat, horse & dog class.

## Using super() function in multiple inheritances in Python

In this example, we will use a parent class, i.e., mammal class. Then, we will inherit the 'Can Fly' and 'Can Swim' classes from the mammal class.

These classes will represent if a given mammal can fly or not and can swim or not. We will define an animal class after that, and it will inherit from both 'Can Fly' and 'Can Swim' class and return us that given animal have the defined characteristics or not.

So, we can see that the animal class we are using here is inherited from multiple base classes, and therefore, it is an example of multiple inheritances in Python. Now, look at the following program.

**Example**

1. # Define Mammal **class** as parent **class**
2. **class** aMammals():
4. def \_\_init\_\_(self, name):
5. print(name, "Is a mammal of animalia class")
7. # define can fly as child **class**
8. **class** FlyCapable(aMammals):
10. def \_\_init\_\_(self, FlyCapable\_name):
11. print(FlyCapable\_name, "is not capable of flying")
13. # Calling Parent **class** Constructor
14. **super**().\_\_init\_\_(FlyCapable\_name)
15. # define can swim as child **class**
16. **class** SwimCapable(aMammals):
18. def \_\_init\_\_(self, SwimCapable\_name):
20. print(SwimCapable\_name, "is not capable of swimming")
22. **super**().\_\_init\_\_(SwimCapable\_name)
24. # Inherit animalia **class** from both fly and swim **class**
25. **class** animalia(FlyCapable, SwimCapable):
27. def \_\_init\_\_(self, name):
29. **super**().\_\_init\_\_(name) # using **super**() function

32. # Taking driver Code **for** animalia **class**
33. Burno = animalia("Cat")

**Output:**

Cat is not capable of flying

Cat is not capable of swimming

Cat Is a mammal of animalia class

**Explanation:**

In the above code, we defined mammal as a parent class. After that, we inherited can fly & can swim classes from the mammal class. We used the methods of both can fly & can swim inside the animalia class with the help of super() function. The animalia class is inherited from both the can swim and can fly class.