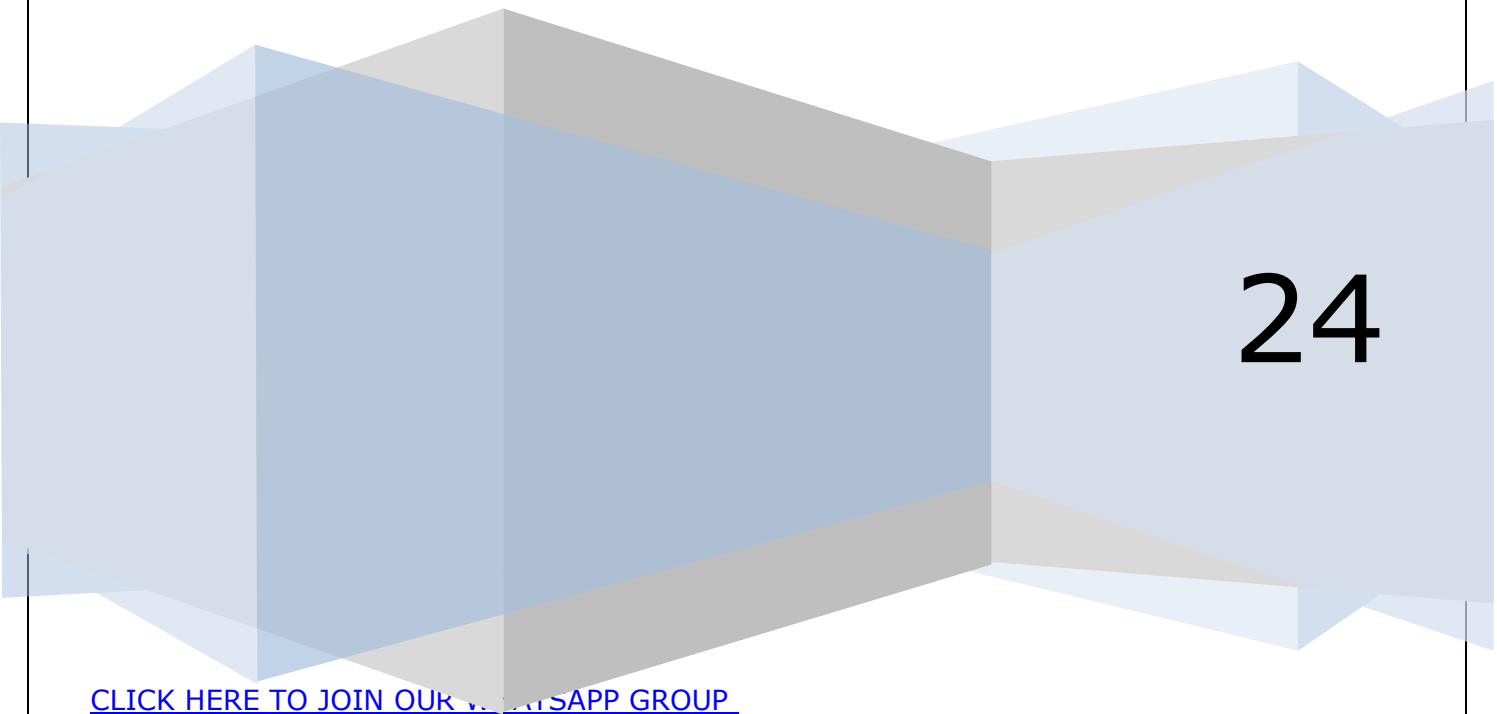
Content (click link)

Python Notes

Created by GPA HOSTELER

BINODPC



Files

What is a file?

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tiedwith the file are freed.

Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

How to open a file?

Python has a built-in function open()to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

File Handling

The key function for working with files in Python is the open()function.

The open() function takes two parameters; ***filename,*** and ***mode.***

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exis

"w" - Write - Opens a file for writing, truncating (remove) the file first.

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

'+'	Open a file for updating (reading and writing)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt","r")
```

The code above is the same as:

```
f = open(" demofile.txt","t"))
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

demofile.txt

Hello! Welcome to demofile.txt
This fileis for testing purposes.
Good Luck!

To open the file, use the built-in open() function.

The open() function returns a file object, which has a read() method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")
print(f.read())
```

Read Only Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the readline() method:

Example

Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

By calling readline() two times, you can read the two first lines:

Example

Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

Run example »

By looping through the lines of the file, you can read the whole file, line by line:

Example

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files

It is a good practice to always close the file when you are done with it.

Example

Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Run example »

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:f = open(
"demofile2.txt", "r") print(f.read())
```

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:

f = open("demofile3.txt", "r")

print(f.read())
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist "a" - Append - will create a file if the specified file does not exist "w" - Write - will create a file if the specified file does not exist

Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Delete a File

To delete a file, you must import the OS module, and run its os.remove() function:

Example

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the `os.rmdir()` method:

Example

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.

How to read files in Python?

To read a file in Python, we must open the file in reading mode.

There are various methods available for this purpose. We can use the `read(size)` method to read in `size` number of data. If `size` parameter is not specified, it reads and returns up to the end of the file.

```
>>> f = open("test.txt",'r',encoding = 'utf-8')
```

```
>>> f.readlines()
['This is my first file\n', 'This file\n', 'contains three lines\n']
```

Errors and Exceptions in Python

Errors are problems in a program that causes the program to stop its execution. On the other hand, exceptions are raised when some internal events change the program's normal flow.

Table of Content

Exception: matlab ek error jo program chalate waqt hoti hai. Python mein, jab error aati hai to program crash nahi hota balki aap use handle kar sakte ho **try** aur **except blocks** ka use karke, taaki aap decide kar sako ki error hone par kya karna hai.

Syntax Errors in Python

When the proper syntax of the language is not followed then a syntax error is thrown.
Example: It returns a syntax error message because after the if statement a colon: is missing. We can fix this by writing the correct syntax.
Python3

```
# initialize the amount variable
amount = 10000

# check that You are eligible to
#  purchase Dsa Self Paced or not
if(amount>2999)
   print("You are eligible to purchase Dsa Self Paced")
```

Output:

```
  File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
                  ^
SyntaxError: invalid syntax
```

Example 2: When indentation is not correct.
Python

```
if(a<3):
print("gfg")
```

Output

```
  File "/home/959e778cc1b15563df98d2e1e26f92e6.py", line 2
    print("gfg")
        ^
IndentationError: expected an indented block
```

Python Logical Errors (Exception)

A logical error in Python, or in any programming language, is a type of bug that occurs when a program runs without crashing (achanak se band hona) but produces incorrect or unintended (unexpected) results. Logical errors are mistakes in the program's logic that lead to incorrect behavior or output, despite the syntax being correct.

Characteristics of Logical Errors

1. **No Syntax Error**: The code runs successfully without any syntax errors.
2. **Unexpected Output**: The program produces output that is different from what is expected.
3. **Difficult to Detect**: Logical errors can be subtle and are often harder to identify and fix compared to syntax errors because the program appears to run correctly.
4. **Varied Causes**: They can arise from incorrect assumptions, faulty logic, improper use of operators, or incorrect sequence of instructions.

Example of a Logical Error

Consider a simple example where we want to compute the average of a list of numbers:

Python

```
numbers = [10, 20, 30, 40, 50]
total = 0

# Calculate the sum of the numbers
for number in numbers:
    total += number

# Calculate the average (this has a logical error)
average = total / len(numbers) - 1

print("The average is:", average)
```

Analysis

- **Expected Output**: The average of the numbers [10, 20, 30, 40, 50] should be 30.
- **Actual Output**: The program will output The average is: 29.0.

Cause of Logical Error

The logical error is in the calculation of the average:

```
average = total / len(numbers) - 1
```

Instead, it should be:

```
average = total / len(numbers)
```

The incorrect logic here is the subtraction of 1, which results in a wrong average calculation.

Common Builtin Exceptions

Some of the common built-in exceptions are other than above mention exceptions are:

Exception	Description
IndexError	When the wrong index of a list is retrieved.
AssertionError	It occurs when the assert statement fails
AttributeError	It occurs when an attribute assignment is failed.
ImportError	It occurs when an imported module is not found.
KeyError	It occurs when the key of the dictionary is not found.
NameError	It occurs when the variable is not defined.
MemoryError	It occurs when a program runs out of memory.
TypeError	It occurs when a function and operation are applied in an incorrect type.

Note: For more information, refer to Built-in Exceptions in Python

Error Handling

When an error and an exception are raised then we handle that with the help of the Handling method.

Handling Exceptions with Try/Except/Finally

We can handle errors by the Try/Except/Finally method. we write unsafe code in the try, fall back code in except and final code in finally block.

Python

```
# put unsafe operation in try block
try:
    print("code start")

    # unsafe operation perform
    print(1 / 0)

# if error occur the it goes in except block
except:
    print("an error occurs")

# final code in finally block
finally:
    print("gautam_kumar_mahto")
```

Output:

```
code start
an error occurs
Gautam_kumar_mhato
```

Raising exceptions for a predefined condition

When we want to code for the limitation of certain conditions then we can raise an exception.

Python3

```
# try for unsafe code
try:
    amount = 1999
    if amount < 2999:

        # raise the ValueError
        raise ValueError("please add money in your account")
    else:
        print("You are eligible to purchase DSA Self Paced course")

# if false then raise the value error
except ValueError as e:
        print(e)
```

Output:

please add money in your account

Click & go th first page

CLICK HERE TO JOIN OUR WHATSAPP GROUP

Modules and packages

we will see the difference between Python's Module, Package, and Library. We will also see some examples of each to things more clear.

What is Module in Python?

The module is a simple Python file that contains collections of functions and global variables and with having a .py extension file. It is an executable file and to organize all the modules we have the concept called Package in Python.

Examples of modules

1. Datetime
2. Regex
3. Random etc.

Example: Save the code in a file called demo_module.py

- Python3

```
def myModule(name):

    print("This is My Module : "+ name)
```

Import module named demo_module and call the myModule function inside it.

- Python3

```
import demo_module
```

```
demo_module.myModule("Math")
```

Output:

```
This is My Module : Math
```

What is Package in Python?

The package is a simple directory having collections of modules. This directory contains Python modules and also having __init__.py file by which the interpreter interprets it as a Package. The package is simply a namespace. The package also contains sub-packages inside it.

Examples of Packages:

1. Numpy
2. Pandas

Example:

```
Student(Package)

| __init__.py (Constructor)

| details.py (Module)

| marks.py (Module)

| collegeDetails.py (Module)
```

What is Library in Python

The **library** is having a collection of related functionality of codes that allows you to perform many tasks without writing your code. **It is a reusable chunk of code that we can use by importing it into our program**, we can just use it by importing that library and calling the method of that library with a period(.). However, it is often assumed that while a package is a collection of modules, a library is a collection of packages.

Examples of Libraries:

1. Matplotlib
2. Pytorch
3. Pygame

4. Seaborn etc.

Example:

Importing pandas library and call read_csv method using an alias of pandas i.e. pd.

- Python3

```
import pandas as pd

df = pd.read_csv("file_name.csv")
```

Click & go th first page

CLICK HERE TO JOIN OUR WHATSAPP GROUP

Numpy

NumPy stands for **Numerical Python,** is an open-source Python library that provides support for large, multi-dimensional arrays and matrices.
It also have a collection of high-level mathematical functions to operate on arrays.
It was created by Travis Oliphant in 2005.

Table of Content

What is NumPy?

NumPy is a general-purpose array-processing package.
It provides a high-performance multidimensional array object and tools for working with these arrays.
It is the fundamental package for scientific computing with Python. It is open-source software.

Features of NumPy

NumPy has various features which make them popular over lists.
Some of these important features include:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy in Python can also be used as an efficient multi-dimensional container of generic data.
Arbitrary data types can be defined using Numpy which allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Install Python NumPy

Numpy can be installed for **Mac** and **Linux** users via the following pip command:

```
pip install numpy
```

Windows does not have any package manager analogous to that in Linux or Mac. Please download the pre-built Windows installer for NumPy from here (according to your system configuration and Python version). And then install the packages manually.

Note: *All the examples discussed below will not run on an* ***online IDE.***

Arrays in NumPy

NumPy's main object is the homogeneous multidimensional array.

- It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In NumPy, dimensions are called *axes*. The number of axes is *rank*.
- NumPy's array class is called **ndarray**. It is also known by the alias **array**.

Example:

In this example, we are creating a two-dimensional array that has the **rank** of 2 as it has 2 **axes**.

The first axis(dimension) is of length 2, i.e., the number of rows, and the second axis(dimension) is of length 3, i.e., the number of columns. The overall shape of the array can be represented as (2, 3)

Python

```
import numpy as np

# Creating array object
arr = np.array( [[ 1, 2, 3],
                 [ 4, 2, 5]] )

# Printing type of arr object
print("Array is of type: ", type(arr))

# Printing array dimensions (axes)
print("No. of dimensions: ", arr.ndim)

# Printing shape of array
print("Shape of array: ", arr.shape)

# Printing size (total number of elements) of array
print("Size of array: ", arr.size)

# Printing type of elements in array
print("Array stores elements of type: ", arr.dtype)
```

Output:

```
Array is of type: <class 'numpy.ndarray'>
No. of dimensions: 2
Shape of array: (2, 3)
```

```
Size of array: 6
Array stores elements of type: int64
```

NumPy Array Creation

There are various ways of Numpy array creation in Python. They are as follows:

1. Create NumPy Array with List and Tuple

You can create an array from a regular Python list or tuple using the array() function. The type of the resulting array is deduced from the type of the elements in the sequences. Let's see this implementation:

Python

```
import numpy as np

# Creating array from list with type float
a = np.array([[1, 2, 4], [5, 8, 7]], dtype = 'float')
print ("Array created using passed list:\n", a)

# Creating array from tuple
b = np.array((1 , 3, 2))
print ("\nArray created using passed tuple:\n", b)
```

Output:

```
Array created using passed list:
 [[1. 2. 4.]
 [5. 8. 7.]]
Array created using passed tuple:
 [1 3 2]
```

2. Create Array of Fixed Size

Often, the element is of an array is originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with **initial placeholder content**.

This minimize the necessity of growing arrays, an expensive operation. **For example:** np.zeros, np.ones, np.full, np.empty, etc.

To create sequences of numbers, NumPy provides a function analogous to the range that returns arrays instead of lists.

Python

```
# Creating a 3X4 array with all zeros
c = np.zeros((3, 4))
print ("An array initialized with all zeros:\n", c)

# Create a constant value array of complex type
d = np.full((3, 3), 6, dtype = 'complex')
print ("An array initialized with all 6s."
        "Array type is complex:\n", d)

# Create an array with random values
e = np.random.random((2, 2))
print ("A random array:\n", e)
```

Output:

```
An array initialized with all zeros:
[[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]]
An array initialized with all 6s.Array type is complex:
[[6.+0.j 6.+0.j 6.+0.j]
[6.+0.j 6.+0.j 6.+0.j]
[6.+0.j 6.+0.j 6.+0.j]]
A random array:
[[0.15471821 0.47506745]
[0.03637972 0.15772238]]
```

3. Create Using arange() Function

arange(): This function returns evenly spaced values within a given interval. **Step** size is specified.

Python

```
# Create a sequence of integers
# from 0 to 30 with steps of 5
f = np.arange(0, 30, 5)
print ("A sequential array with steps of 5:\n", f)
```

Output:

```
A sequential array with steps of 5:
[ 0 5 10 15 20 25]
```

5. Create Using linspace() Function

linspace(): It returns evenly spaced values within a given interval.

Python

```
# Create a sequence of 10 values in range 0 to 5
g = np.linspace(0, 5, 10)
print ("A sequential array with 10 values between"
                    "0 and 5:\n", g)
```

Output:

```
A sequential array with 10 values between0 and 5:
[0. 0.55555556 1.11111111 1.66666667 2.22222222 2.77777778
 3.33333333 3.88888889 4.44444444 5. ]
```

6. Reshaping Array using Reshape Method

Reshaping array: We can use **reshape** method to reshape an array.

Consider an array with shape (a1, a2, a3, …, aN). We can reshape and convert it into another array with shape (b1, b2, b3, …, bM). The only required condition is a1 x a2 x a3 … x aN = b1 x b2 x b3 … x bM. (i.e. the original size of the array remains unchanged.)

Python

```
# Reshaping 3X4 array to 2X2X3 array
arr = np.array([[1, 2, 3, 4],
                [5, 2, 4, 2],
                [1, 2, 0, 1]])
```

```
newarr = arr.reshape(2, 2, 3)

print ("Original array:\n", arr)
print("---------------")
print ("Reshaped array:\n", newarr)
```

Output:

```
Original                                                array:
 [[1                  2                  3                  4]
 [5                  2                  4                  2]
 [1                  2                  0                  1]]
---------------
Reshaped                                                array:
 [[[1                          2                          3]
  [4                          5                          2]]
 [[4                          2                          1]
  [2 0 1]]]
```

7. Flatten Array

Flatten array: We can use **flatten** method to get a copy of the array collapsed into **one dimension**.

It accepts *order* argument. The default value is ????' (for row-major order). Use ????' for column-major order.

Python

```
# Flatten array
arr = np.array([[1, 2, 3], [4, 5, 6]])
flat_arr = arr.flatten()

print ("Original array:\n", arr)
print ("Fattened array:\n", flat_arr)
```

Output:

```
Original                                                array:
 [[1                          2                          3]
 [4                          5                          6]]
Fattened                                                array:
 [1 2 3 4 5 6]
```

Note: *The type of array can be explicitly defined while creating the array.*

NumPy Array Indexing

Knowing the basics of NumPy array indexing is important for analyzing and manipulating the array object. NumPy in Python offers many ways to do array indexing.

- **Slicing:** Just like lists in Python, NumPy arrays can be sliced. As arrays can be multidimensional, you need to specify a slice for each dimension of the array.
- **Integer array indexing:** In this method, lists are passed for indexing for each dimension. One-to-one mapping of corresponding elements is done to construct a new arbitrary array.

- **Boolean array indexing:** This method is used when we want to pick elements from the array which satisfy some condition.

Python

```
# Python program to demonstrate
# indexing in numpy
import numpy as np

# An exemplar array
arr = np.array([[-1, 2, 0, 4],
                [4, -0.5, 6, 0],
                [2.6, 0, 7, 8],
                [3, -7, 4, 2.0]])

# Slicing array
temp = arr[:2, ::2]
print ("Array with first 2 rows and alternate"
                    "columns(0 and 2):\n", temp)

# Integer array indexing example
temp = arr[[0, 1, 2, 3], [3, 2, 1, 0]]
print ("\nElements at indices (0, 3), (1, 2), (2, 1),"
                                "(3, 0):\n", temp)

# boolean array indexing example
cond = arr > 0 # cond is a boolean array
temp = arr[cond]
print ("\nElements greater than 0:\n", temp)
```

Output:

```
Array with first 2 rows and alternatecolumns(0 and 2):
 [[-1.  0.]
 [ 4.  6.]]
Elements at indices (0, 3), (1, 2), (2, 1),(3, 0):
 [ 4.  6.  0.  3.]
Elements greater than 0:
 [ 2.  4.  4.  6.  2.6 7.  8.  3.  4.  2. ]
```

NumPy Basic Operations

The Plethora of built-in arithmetic functions is provided in Python NumPy.

1. Operations on a single NumPy array

We can use overloaded arithmetic operators to do element-wise operations on the array to create a new array. In the case of +=, -=, *= operators, the existing array is modified.

Python

```
# Python program to demonstrate
# basic operations on single array
import numpy as np
```

```
a = np.array([1, 2, 5, 3])

# add 1 to every element
print ("Adding 1 to every element:", a+1)

# subtract 3 from each element
print ("Subtracting 3 from each element:", a-3)

# multiply each element by 10
print ("Multiplying each element by 10:", a*10)

# square each element
print ("Squaring each element:", a**2)

# modify existing array
a *= 2
print ("Doubled each element of original array:", a)

# transpose of array
a = np.array([[1, 2, 3], [3, 4, 5], [9, 6, 0]])

print ("\nOriginal array:\n", a)
print ("Transpose of array:\n", a.T)
```

Output:

```
Adding 1 to every element: [2 3 6 4]
Subtracting 3 from each element: [-2 -1 2 0]
Multiplying each element by 10: [10 20 50 30]
Squaring each element: [ 1 4 25 9]
Doubled each element of original array: [ 2 4 10 6]
Original array:
 [[1 2 3]
 [3 4 5]
 [9 6 0]]
Transpose of array:
 [[1 3 9]
 [2 4 6]
 [3 5 0]]
```

NumPy – Unary Operators

Many unary operations are provided as a method of **ndarray** class. This includes sum, min, max, etc. These functions can also be applied row-wise or column-wise by setting an axis parameter.

Python

```
# Python program to demonstrate
# unary operators in numpy
```

```
import numpy as np

arr = np.array([[1, 5, 6],
                [4, 7, 2],
                [3, 1, 9]])

# maximum element of array
print ("Largest element is:", arr.max())
print ("Row-wise maximum elements:",
                    arr.max(axis = 1))

# minimum element of array
print ("Column-wise minimum elements:",
                       arr.min(axis = 0))

# sum of array elements
print ("Sum of all array elements:",
                          arr.sum())

# cumulative sum along each row
print ("Cumulative sum along each row:\n",
                    arr.cumsum(axis = 1))
```

Output:

```
Largest element is: 9
Row-wise maximum elements: [6 7 9]
Column-wise minimum elements: [1 1 2]
Sum of all array elements: 38
Cumulative sum along each row:
[[ 1  6 12]
 [ 4 11 13]
 [ 3  4 13]]
```

NumPy – Binary Operators

These operations apply to the array elementwise and a new array is created. You can use all basic arithmetic operators like +, -, /, etc. In the case of +=, -=, = operators, the existing array is modified.

Python

```
# Python program to demonstrate
# binary operators in Numpy
import numpy as np

a = np.array([[1, 2],
              [3, 4]])
b = np.array([[4, 3],
              [2, 1]])
```

```
# add arrays
print ("Array sum:\n", a + b)

# multiply arrays (elementwise multiplication)
print ("Array multiplication:\n", a*b)

# matrix multiplication
print ("Matrix multiplication:\n", a.dot(b))
```

Output:

```
Array sum:
[[5 5]
 [5 5]]
Array multiplication:
[[4 6]
 [6 4]]
Matrix multiplication:
[[ 8  5]
 [20 13]]
```

Also Read: Numpy Binary Operations

NymPy's ufuncs

NumPy provides familiar mathematical functions such as sin, cos, exp, etc. These functions also operate elementwise on an array, producing an array as output.

Note: All the operations we did above using overloaded operators can be done using ufuncs like np.add, np.subtract, np.multiply, np.divide, np.sum, etc.

Click & go th first page

Panda

Pandas Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called *index*. Labels need not be unique but must be a hashable type. The object supports both integer and label-based indexing and provides a host of methods for performing operations involving the index.

To create Series with any of the methods make sure to import pandas library.

Creating an empty Series: Series() function of Pandas is used to create a series. A basic series, which can be created is an Empty Series.

- Python3

```
# import pandas as pd

import pandas as pd

 # Creating empty series

ser = pd.Series()
```

```
print(ser)
```

Output :

```
Series([], dtype: float64)
```

By default, the data type of Series is float.

Creating a series from array: In order to create a series from NumPy array, we have to import numpy module and have to use array() function.

- Python3

```
# import pandas as pd

import pandas as pd

 # import numpy as np

import numpy as np

# simple array

data = np.array(['g', 'e', 'e', 'k', 's'])

 ser = pd.Series(data)

print(ser)
```

Output:

```
0    g
1    e
2    e
3    k
4    s
dtype: object
```

By default, the index of the series starts from 0 till the length of series -1.

Creating a series from array with an index: In order to create a series by explicitly proving index instead of the default, we have to provide a list of elements to the index parameter with the same number of elements as it is an array.

- Python3

```
# import pandas as pd

import pandas as pd



# import numpy as np

import numpy as np



# simple array

data = np.array(['g', 'e', 'e', 'k', 's'])



# providing an index

ser = pd.Series(data, index=[10, 11, 12, 13, 14])
```

```
print(ser)
```

Output:

```
10    g
11    e
12    e
13    k
14    s
dtype: object
```

Creating a series from Lists: In order to create a series from list, we have to first create a list after that we can create a series from list.

- Python3

```
import pandas as pd



# a simple list

list = ['g', 'e', 'e', 'k', 's']



# create series form a list

ser = pd.Series(list)

print(ser)
```

Output :

```
0    g
1    e
2    e
3    k
4    s
dtype: object
```

Creating a series from Dictionary: In order to create a series from the dictionary, we have to first create a dictionary after that we can make a series using dictionary. Dictionary keys are used to construct indexes of Series.

- Python3

```
import pandas as pd



# a simple dictionary

dict = {'Geeks': 10,

        'for': 20,

        'geeks': 30}



# create series from dictionary

ser = pd.Series(dict)



print(ser)
```

Output:

```
Geeks    10
for      20
geeks    30
dtype: int64
```

Creating a series from Scalar value: In order to create a series from scalar value, an index must be provided. The scalar value will be repeated to match the length of the index.

- Python3

```
import pandas as pd

import numpy as np

# giving a scalar value with index

ser = pd.Series(10, index=[0, 1, 2, 3, 4, 5])

print(ser)
```

Output:

```
0    10
1    10
2    10
3    10
4    10
5    10
dtype: int64
```

Creating a series using NumPy functions : In order to create a series using numpy function, we can use different function of numpy like numpy.linspace(), numpy.random.radn().

- Python3

```
# import pandas and numpy

import pandas as pd

import numpy as np



# series with numpy linspace()

ser1 = pd.Series(np.linspace(3, 33, 3))

print(ser1)



# series with numpy linspace()

ser2 = pd.Series(np.linspace(1, 100, 10))

print(& quot

      \n"       , ser2)
```

Output:

```
0     3.0
1    18.0
2    33.0
dtype: float64

 0      1.0
1     12.0
2     23.0
3     34.0
4     45.0
5     56.0
6     67.0
7     78.0
8     89.0
9    100.0
dtype: float64
```

Creating a Series using range function:

- Python3

```
# code

import pandas as pd

ser=pd.Series(range(10))

print(ser)
```

Output:

0 0

1 1

2 2

3 3

4 4

5 5

6 6

7 7

```
8    8
9    9
dtype: int64
```

Creating a Series using for loop and list comprehension:

- Python3

```
import pandas as pd

ser=pd.Series(range(1,20,3), index=[x for x in 'abcdefg'])

print(ser)
```

Output:

```
a     1
b     4
c     7
d    10
e    13
f    16
g    19
dtype: int64
```

Creating a Series using mathematical expressions:

- Python3

```
import pandas as pd
```

```
import numpy as np

ser=np.arange(10,15)

serobj=pd.Series(data=ser*5,index=ser)

print(serobj)
```

Output:

```
10    50
11    55
12    60
13    65
14    70
dtype: int32
```

Click & go th first page

CLICK HERE TO JOIN OUR WHATSAPP GROUP

Function

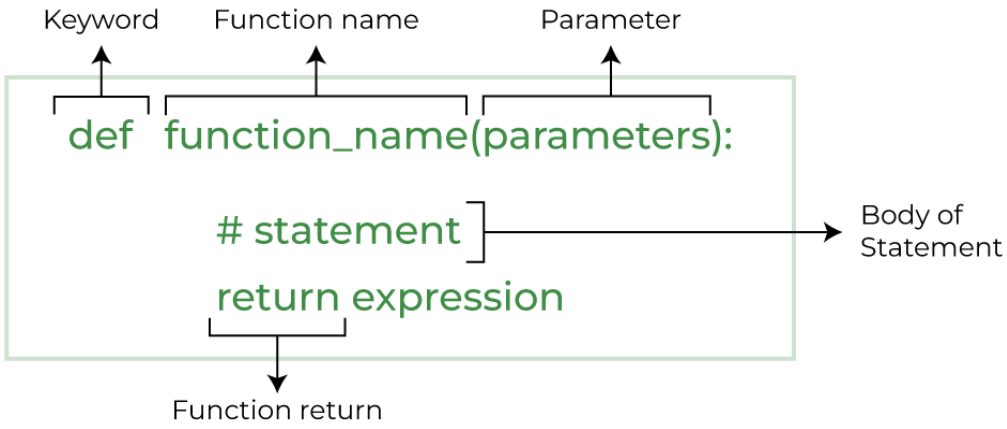
Python Functions is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.
Some .

Benefits of Using Functions

- Increase Code Readability
- Increase Code Reusability

Python Function Declaration

The syntax to declare a function is:



Syntax of Python Function Declaration

Types of Functions in Python

Below are the different types of functions in Python:

- **Built-in library function:** These are Standard functions in Python that are available to use.
- **User-defined function:** We can create our own functions based on our requirements.

Creating a Function in Python

We can define a function in Python, using the **def** keyword. We can add any type of functionalities and properties to it as we require. By the following example, we

can understand how to write a function in Python. In this way we can create Python function definition by using def keyword.

Python

```
# A simple Python function
def fun():
    print("Welcome to GFG")
```

Calling a Function in Python

After creating a function in Python we can call it by using the name of the functions Python followed by parenthesis containing parameters of that particular function. Below is the example for calling def function Python.

Python

```
# A simple Python function
def fun():
    print("Welcome to GFG")


# Driver code to call a function
fun()
```

Output:

```
Welcome to GFG
```

Python Function with Parameters

If you have experience in C/C++ or Java then you must be thinking about the *return type* of the function and *data type* of arguments. That is possible in Python as well (specifically for Python 3.5 and above).

Python Function Syntax with Parameters

```
def function_name(parameter: data_type) -> return_type:
    """Docstring"""
    # body of the function
    return expression
```

The following example uses arguments and parameters that you will learn later in this article so you can come back to it again if not understood.

Python

```
def add(num1: int, num2: int) -> int:
    """Add two numbers"""
    num3 = num1 + num2

    return num3

# Driver code
num1, num2 = 5, 15
ans = add(num1, num2)
print(f"The addition of {num1} and {num2} results {ans}.")
```

Output:

```
The addition of 5 and 15 results 20.
```

Note: *The following examples are defined using syntax 1, try to convert them in syntax 2 for practice.*

Python

```
# some more functions
def is_prime(n):
    if n in [2, 3]:
        return True
    if (n == 1) or (n % 2 == 0):
        return False
    r = 3
    while r * r <= n:
        if n % r == 0:
            return False
        r += 2
    return True
print(is_prime(78), is_prime(79))
```

Output:

```
False True
```

Python Function Arguments

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.
In this example, we will create a simple function in Python to check whether the number passed as an argument to the function is even or odd.

Python

```
# A simple Python function to check
# whether x is even or odd
def evenOdd(x):
    if (x % 2 == 0):
        print("even")
    else:
        print("odd")


# Driver code to call the function
evenOdd(2)
evenOdd(3)
```

Output:

```
even
odd
```

Types of Python Function Arguments

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following function argument types in Python:

- **Default argument**
- **Keyword arguments (named arguments)**
- **Positional arguments**

- **Arbitrary arguments** (variable-length arguments *args and **kwargs)

Let's discuss each type in detail.

Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments to write functions in Python.

Python

```
# Python program to demonstrate
# default arguments
def myFun(x, y=50):
    print("x: ", x)
    print("y: ", y)


# Driver code (We call myFun() with only
# argument)
myFun(10)
```

Output:

```
x:                                                    10
y:  50
```

Like C++ default arguments, any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

Keyword Arguments

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

Python

```
# Python program to demonstrate Keyword Arguments
def student(firstname, lastname):
    print(firstname, lastname)


# Keyword arguments
student(firstname='Geeks', lastname='Practice')
student(lastname='Practice', firstname='Geeks')
```

Output:

```
Geeks                                                 Practice
Geeks Practice
```

Positional Arguments

We used the Position argument during the function call so that the first argument (or value) is assigned to name and the second argument (or value) is assigned to

age. By changing the position, or if you forget the order of the positions, the values can be used in the wrong places, as shown in the Case-2 example below, where 27 is assigned to the name and Suraj is assigned to the age.

Python

```
def nameAge(name, age):
    print("Hi, I am", name)
    print("My age is ", age)


# You will get correct output because
# argument is given in order
print("Case-1:")
nameAge("Suraj", 27)
# You will get incorrect output because
# argument is not in order
print("\nCase-2:")
nameAge(27, "Suraj")
```

Output:

```
Case-1:
Hi, I am Suraj
My age is 27
Case-2:
Hi, I am 27
My age is  Suraj
```

Arbitrary Keyword Arguments

In Python Arbitrary Keyword Arguments, *args, and **kwargs can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- *args in Python (Non-Keyword Arguments)
- **kwargs in Python (Keyword Arguments)

Example 1: Variable length non-keywords argument

Python

```
# Python program to illustrate
# *args for variable number of arguments
def myFun(*argv):
    for arg in argv:
        print(arg)


myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Output:

```
Hello
Welcome
```

```
to
GeeksforGeeks
```

Example 2: Variable length keyword arguments

Python

```
# Python program to illustrate
# *kwargs for variable number of keyword arguments


def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))


# Driver code
myFun(first='Geeks', mid='for', last='Geeks')
```

Output:

```
first == Geeks
mid == for
last == Geeks
```

Docstring

The first string after the function is called the Document string or Docstring in short. This is used to describe the functionality of the function. The use of docstring in functions is optional but it is considered a good practice.

The below syntax can be used to print out the docstring of a function.

Syntax: print(function_name.__doc__)

Example: Adding Docstring to the function

Python

```
# A simple Python function to check
# whether x is even or odd


def evenOdd(x):
    """Function to check if the number is even or odd"""

    if (x % 2 == 0):
        print("even")
    else:
        print("odd")


# Driver code to call the function
print(evenOdd.__doc__)
```

Output:

```
Function to check if the number is even or odd
```

Python Function within Functions

A function that is defined inside another function is known as the **inner function** or **nested function**. Nested functions can access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function.

Python

```
# Python program to
# demonstrate accessing of
# variables of nested functions

def f1():
    s = 'I love GeeksforGeeks'

    def f2():
        print(s)

    f2()

# Driver's code
f1()
```

Output:

```
I love GeeksforGeeks
```

Anonymous Functions in Python

In Python, an anonymous function means that a function is without a name. As we already know the def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.

Python

```
# Python code to illustrate the cube of a number
# using lambda function
def cube(x): return x*x*x

cube_v2 = lambda x : x*x*x

print(cube(7))
print(cube_v2(7))
```

Output:

```
343
343
```

Recursive Functions in Python

Recursion in Python refers to when a function calls itself. There are many instances when you have to build a recursive function to solve **Mathematical and Recursive Problems.**

Using a recursive function should be done with caution, as a recursive function can become like a non-terminating loop. It is better to check your exit statement while creating a recursive function.

Python

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(4))
```

Output

```
24
```

Here we have created a recursive function to calculate the factorial of the number. You can see the end statement for this function is when n is equal to 0.

Return Statement in Python Function

The function return statement is used to exit from a function and go back to the function caller and return the specified value or data item to the caller. **The syntax for the return statement is:**

```
return [expression_list]
```

The return statement can consist of a variable, an expression, or a constant which is returned at the end of the function execution. If none of the above is present with the return statement a None object is returned.

Example: Python Function Return Statement

Python

```
def square_value(num):
    """This function returns the square
    value of the entered number"""
    return num**2


print(square_value(2))
print(square_value(-4))
```

Output:

```
4
16
```

Pass by Reference and Pass by Value

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function Python, a new reference to the object is created. Parameter passing in Python is the same as reference passing in Java.

Python

```
# Here x is a new reference to same list lst
```

```
def myFun(x):
    x[0] = 20


# Driver Code (Note that lst is modified
# after function call.
lst = [10, 11, 12, 13, 14, 15]
myFun(lst)
print(lst)
```

Output:

```
[20, 11, 12, 13, 14, 15]
```

When we pass a reference and change the received reference to something else, the connection between the passed and received parameters is broken. For example, consider the below program as follows:

Python

```
def myFun(x):

    # After below line link of x with previous
    # object gets broken. A new object is assigned
    # to x.
    x = [20, 30, 40]


# Driver Code (Note that lst is not modified
# after function call.
lst = [10, 11, 12, 13, 14, 15]
myFun(lst)
print(lst)
```

Output:

```
[10, 11, 12, 13, 14, 15]
```

Another example demonstrates that the reference link is broken if we assign a new value (inside the function).

Python

```
def myFun(x):

    # After below line link of x with previous
    # object gets broken. A new object is assigned
    # to x.
    x = 20


# Driver Code (Note that x is not modified
# after function call.
x = 10
myFun(x)
print(x)
```

Output:

```
10
```

Exercise: Try to guess the output of the following code.

Python

```
def swap(x, y):
    temp = x
    x = y
    y = temp

# Driver code
x = 2
y = 3
swap(x, y)
print(x)
print(y)
```

Output:

```
2
3
```

Quick Links

- Quiz on Python Functions
- Difference between Method and Function in Python
- First Class functions in Python
- Recent articles on Python Functions.

FAQs- Python Functions

What is function in Python?

Python function is a block of code, that runs only when it is called. It is programmed to return the specific task. You can pass values in functions called parameters. It helps in performing repetitive tasks.

What are the 4 types of Functions in Python?

The main types of functions in Python are:

- *Built-in function*
- *User-defined function*
- *Lambda functions*
- *Recursive functions*

H**ow to Write a Function in Python**?

*To write a function in Python you can use the def keyword and then write the function name. You can provide the function code after using **':'**. Basic syntax to define a function is:*

def function_name():

#statement

What are the parameters of a function in Python?

Parameters in Python are the variables that take the values passed as arguments when calling the functions. A function can have any number of parameters. You can also set default value to a parameter in Python.

What is Python main function?

The Python main function refers to the entry point of a Python program. It is often defined using the if __name__ == "__main__": construct to ensure that certain code is only executed when the script is run directly, not when it is imported as a module.

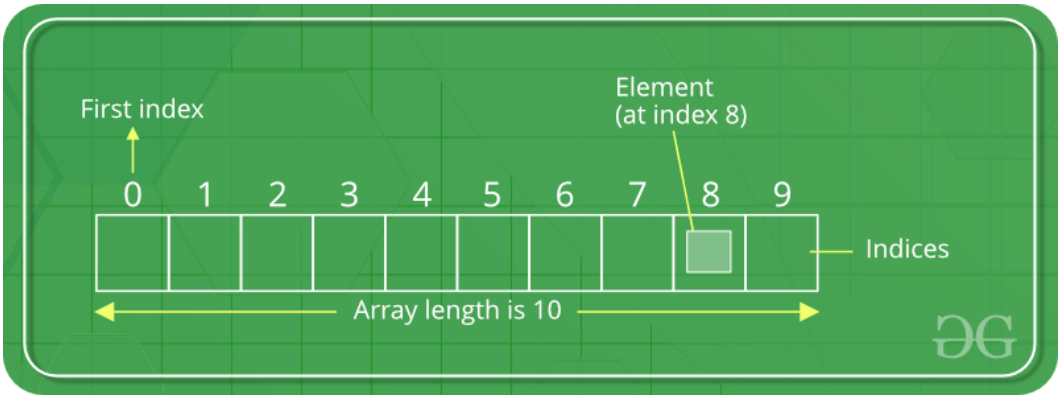
Click & go th first page

CLICK HERE TO JOIN OUR WHATSAPP GROUP

Array

What is an Array in Python?

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



Create an Array in Python

Array in Python can be created by importing an array module. **array(*data_type, value_list*)** is used to create array in Python with data type and value list specified in its arguments.
In below code Python create array : one of **integers** and one of **doubles**. It then prints the contents of each array to the console.
Python

```
import array as arr
a = arr.array('i', [1, 2, 3])
print("The new created array is : ", end=" ")
for i in range(0, 3):
    print(a[i], end=" ")
print()
b = arr.array('d', [2.5, 3.2, 3.3])
print("\nThe new created array is : ", end=" ")
for i in range(0, 3):
    print(b[i], end=" ")
```

Output

```
The new created array is :  1 2 3
```

```
The new created array is :  2.5 3.2 3.3
```

Complexities for Creation of Arrays:
TimeComplexity: O(1)
Auxiliary Space: O(n)
Some of the data types are mentioned below which will help in create array in Python 3.8
of different data types.

Type Code	C Type	Python Type	Minimum Size In Bytes
'b'	signed cahar	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Now we will see how to use array in Python 3.8 with example.

Adding Elements to a Array

Elements can be added to the Python Array by using built-in insert() function. Insert is used to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. append() is also used to add the value mentioned in its arguments at the end of the Python array.
3.2, 3.3] is created and printed before and after appending the double **4.4** to the array.
Python

```
import array as arr
```

```
a = arr.array('i', [1, 2, 3])
print("Array before insertion : ", end=" ")
for i in range(0, 3):
    print(a[i], end=" ")
print()
a.insert(1, 4)
print("Array after insertion : ", end=" ")
for i in (a):
    print(i, end=" ")
print()
b = arr.array('d', [2.5, 3.2, 3.3])
print("Array before insertion : ", end=" ")
for i in range(0, 3):
    print(b[i], end=" ")
print()
b.append(4.4)
print("Array after insertion : ", end=" ")
for i in (b):
    print(i, end=" ")
print()
```

Output

```
Array before insertion :  1 2 3

Array after insertion :  1 4 2 3

Array before insertion :  2.5 3.2 3.3

Array after insertion :  2.5 3.2 3.3 4.4
```

Complexities for Adding elements to the Arrays

Time Complexity: O(1)/O(n) (O(1) – for inserting elements at the end of the array, O(n) – for inserting elements at the beginning of the array and to the full array

Auxiliary Space: O(1)

Accessing Elements from the Array

In order to access the array items refer to the index number. Use the index operator [ ] to access an item in a array in Python. The index must be an integer.

Below, code shows first how to Python import array and use of indexing to access elements in arrays. The a[0] expression accesses the first element of the array a, which is 1. The a[3] expression accesses the fourth element of the array a, which is 4. Similarly, the b[1] expression accesses the second element of the array b, which is 3.2, and the b[2] expression accesses the third element of the array b, which is 3.3.

Python

```
import array as arr
```

```
a = arr.array('i', [1, 2, 3, 4, 5, 6])
print("Access element is: ", a[0])
print("Access element is: ", a[3])
b = arr.array('d', [2.5, 3.2, 3.3])
print("Access element is: ", b[1])
print("Access element is: ", b[2])
```

Output

```
Access element is:  1

Access element is:  4

Access element is:  3.2

Access element is:  3.3
```

Complexities for accessing elements in the Arrays

Time **Complexity:** O(1)

Auxiliary Space: O(1)

Removing Elements from the Array

Elements can be removed from the Python array by using built-in remove() function but an Error arises if element doesn't exist in the set. Remove() method only removes one element at a time, to remove range of elements, iterator is used. pop() function can also be used to remove and return an element from the array, but by default it removes only the last element of the array, to remove element from a specific position of the array, index of the element is passed as an argument to the pop() method.

Note – Remove method in List will only remove the first occurrence of the searched element.

Below, code shows how to Python import array, how to create, print, remove elements from, and access elements of an array in Python. It imports the array module, which is used to work with arrays. It creates an array of integers in and Python print arrays or prints the original array. It then removes an element from the array and prints the modified array. Finally, it removes all occurrences of a specific element from the array and prints the updated array

Python

```
import array
arr = array.array('i', [1, 2, 3, 1, 5])
print("The new created array is : ", end="")
for i in range(0, 5):
    print(arr[i], end=" ")
```

```
print("\r")
print("The popped element is : ", end="")
print(arr.pop(2))
print("The array after popping is : ", end="")
for i in range(0, 4):
    print(arr[i], end=" ")

print("\r")
arr.remove(1)
print("The array after removing is : ", end="")
for i in range(0, 3):
    print(arr[i], end=" ")
```

Output

```
The new created array is : 1 2 3 1 5

The popped element is : 3

The array after popping is : 1 2 1 5

The array after removing is : 2 1 5
```

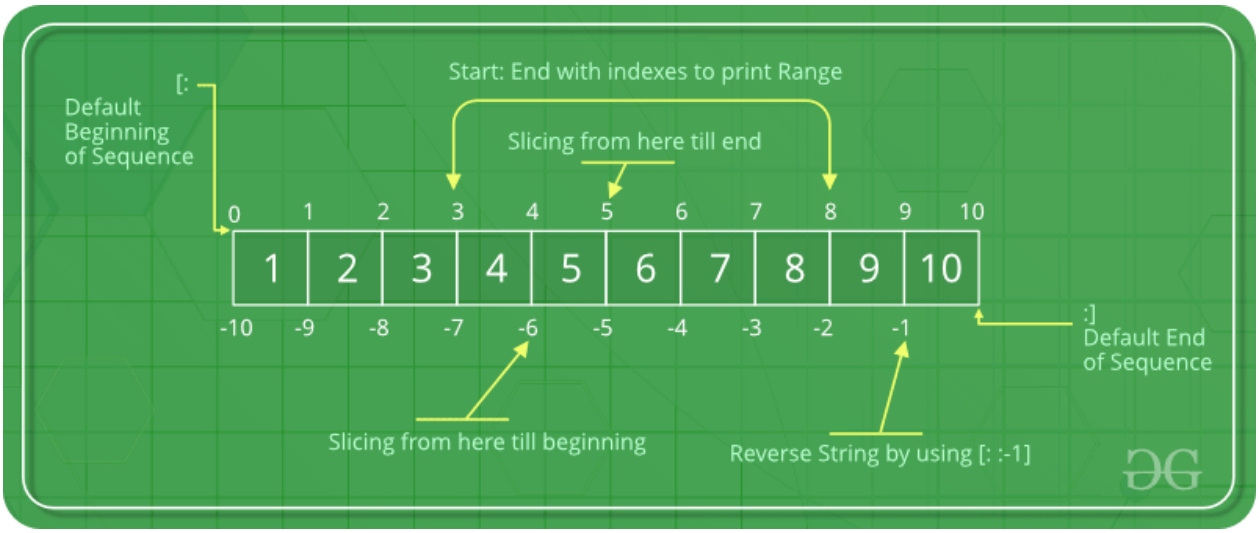
Complexities for Removing elements in the Arrays

Time Complexity: O(1)/O(n) (O(1) – for removing elements at the end of the array, O(n) – for removing elements at the beginning of the Python create array and to the full array

Auxiliary Space: O(1)

Slicing of an Array

In Python array, there are multiple ways to print the whole array with all the elements, but to print a specific range of elements from the array, we use Slice operation. Slice operation is performed on array with the use of colon(:). To print elements from beginning to a range use [:Index], to print elements from end use [:-Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print whole List with the use of slicing operation, use [:]. Further, to print whole array in reverse order, use [::-1].



This code employs slicing to extract elements or subarrays from an array. It starts with an initial array of integers and creates an array from the list. The code slices the array to extract elements from index 3 to 8, from index 5 to the end, and the entire array and In below code Python print array as The sliced arrays are then printed to demonstrate the slicing operations.

Python

```
import array as arr
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

a = arr.array('i', l)
print("Initial Array: ")
for i in (a):
    print(i, end=" ")
Sliced_array = a[3:8]
print("\nSlicing elements in a range 3-8: ")
print(Sliced_array)
Sliced_array = a[5:]
print("\nElements sliced from 5th "
      "element till the end: ")
print(Sliced_array)
Sliced_array = a[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_array)
```

Output

```
Initial Array:

1 2 3 4 5 6 7 8 9 10

Slicing elements in a range 3-8:
```

```
array('i', [4, 5, 6, 7, 8])


Elements sliced from 5th element till the end:

array('i', [6, 7, 8, 9, 10])


Printing all elements...
```

Searching Element in an Array

In order to search an element in the array we use a python in-built index() method. This function returns the index of the first occurrence of value mentioned in arguments.

Example: The code demonstrates how to create array in Python, print its elements, and find the indices of specific elements. It imports the array module, creates an array of integers, prints the array using a for loop, and then uses the index() method to find the indices of the first occurrences of the integers 2 and 1.

Python

```
import array
arr = array.array('i', [1, 2, 3, 1, 2, 5])
print("The new created array is : ", end="")
for i in range(0, 6):
    print(arr[i], end=" ")

print("\r")
print("The index of 1st occurrence of 2 is : ", end="")
print(arr.index(2))
print("The index of 1st occurrence of 1 is : ", end="")
print(arr.index(1))
```

Output

```
The new created array is : 1 2 3 1 2 5

The index of 1st occurrence of 2 is : 1

The index of 1st occurrence of 1 is : 0
```

Complexities for searching elements in the Arrays

Time Complexity: O(n)

Auxiliary Space: O(1)

Updating Elements in a Array

In order to update an element in the array we simply reassign a new value to the desired index we want to update.

Example: This code illustrates the functionality of modifying elements within an array using indexing. It imports the array module, creates an array of integers, and prints the initial array. Then, it modifies two elements of the array at specific indexes and prints the updated array. This serves to demonstrate how indexing allows for dynamic manipulation of array contents.

Python

```
import array
arr = array.array('i', [1, 2, 3, 1, 2, 5])
print("Array before updation : ", end="")
for i in range(0, 6):
    print(arr[i], end=" ")

print("\r")
arr[2] = 6
print("Array after updation : ", end="")
for i in range(0, 6):
    print(arr[i], end=" ")
print()
arr[4] = 8
print("Array after updation : ", end="")
for i in range(0, 6):
    print(arr[i], end=" ")
```

Output

```
Array before updation : 1 2 3 1 2 5

Array after updation : 1 2 6 1 2 5

Array after updation : 1 2 6 1 8 5
```

Complexities for updating elements in the Arrays

Time **Complexity:** O(1)

Auxiliary Space: O(1)

Reversing Elements in a Array

In order to reverse elements of an array we need to simply use reverse method.

Example: The presented code demonstrates the functionality of reversing the order of elements within an array using the **reverse()** method. It imports the array module, creates an array of integers, prints the original array, reverses the order of elements using reverse(), and then prints the reversed array. This effectively illustrates the ability to modify the arrangement of elements in an array.

Python

```
import array
my_array = array.array('i', [1, 2, 3, 4, 5])
print("Original array:", *my_array)
my_array.reverse()
print("Reversed array:", *my_array)
```

Output

```
Original array: 1 2 3 4 5

Reversed array: 5 4 3 2 1
```

Complexities for reversing elements in the Arrays:

Time **Complexity:** O(n)

Auxiliary Space: O(1)

Extend Element from Array

In the article, we will cover the python list **extend()** and try to understand the **Python list extend().**

What is extend element from array?

In Python, an array is used to store multiple values or elements of the same datatype in a single variable. The **extend()** function is simply used to attach an item from iterable to the end of the array. In simpler terms, this method is used to add an array of values to the end of a given or existing array.

Syntax of list extend()

The syntax of the **extend()** method:

```
list.extend(iterable)
```

Here, all the element of iterable are added to the end of list1

Example 1:

The provided code demonstrates the capability of extending an array to include additional elements. It imports the array module using an alias, creates an array of integers, prints the array before extension, extends the array using the **extend()** method, and finally prints the extended array. This concisely illustrates the ability to add elements to an existing array structure

Python

```
import array as arr
a = arr.array('i', [1, 2, 3,4,5])
print("The before array extend  : ", end =" ")
for i in range (0, 5):

    print (a[i], end =" ")

print()
a.extend([6,7,8,9,10])
print("\nThe array after extend :",end=" ")
```

```
for i in range(0,10):

    print(a[i],end=" ")

print()
```

Output

```
The before array extend  :  1 2 3 4 5



The array after extend : 1 2 3 4 5 6 7 8 9 10
```

CLICK HERE TO JOIN OUR WHATSAPP GROUP

Dictionary

A Python dictionary is a data structure that stores the value in **key:value pairs**.
Example:
Python dictionaries are essential for efficient data mapping and manipulation in programming. To deepen your understanding of dictionaries and explore advanced techniques in data handling, consider enrolling in our **Complete Machine Learning & Data Science Program** . This course covers everything from basic dictionary operations to advanced data processing methods, empowering you to become proficient in Python programming and data analysis.
Python

```
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print(Dict)
```

Output:

```
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

Python Dictionary Syntax

dict_var = {key1 : value1, key2 : value2, ...}

What is a Dictionary in Python?

Dictionaries in Python is a data structure, used to store values in key:value format. This makes it different from lists, tuples, and arrays as in a dictionary each key has an associated value.
Note: *As of Python version 3.7, dictionaries are ordered and can not contain duplicate keys.*

How to Create a Dictionary

In Python , a dictionary can be created by placing a sequence of elements within curly **{}** braces, separated by a ‘comma’.
The dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value** .
Values in a dictionary can be of any data type and can be duplicated, whereas keys can’t be repeated and must be *immutable* .
Note – Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.
The code demonstrates creating dictionaries with different types of keys. The first dictionary uses integer keys, and the second dictionary uses a mix of string and integer keys with corresponding values. This showcases the flexibility of Python dictionaries in handling various data types as keys.

Python

```
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

Output

```
Dictionary with the use of Integer Keys:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
Dictionary with the use of Mixed Keys:
{'Name': 'Geeks', 1: [1, 2, 3, 4]}
```

Dictionary Example

A dictionary can also be created by the built-in function **dict().** An empty dictionary can be created by just placing **curly braces{}.**

Different Ways to Create a Python Dictionary

The code demonstrates different ways to create dictionaries in Python. It first creates an empty dictionary, and then shows how to create dictionaries using the **dict()** constructor with key-value pairs specified within curly braces and as a list of tuples.

Python

```
Dict = {}
print("Empty Dictionary: ")
print(Dict)

Dict = dict({1: 'Geeks', 2: 'For', 3: 'Geeks'})
print("\nDictionary with the use of dict(): ")
print(Dict)

Dict = dict([(1, 'Geeks'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}
Dictionary with the use of dict():
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
Dictionary with each item as a pair:
{1: 'Geeks', 2: 'For'}
```

Complexities for Creating a Dictionary:

Adding Elements to a Dictionary

The addition of elements can be done in multiple ways. One value at a time can be added to a Dictionary by defining value along with the key e.g. Dict[Key] = 'Value'.
Updating an existing value in a Dictionary can be done by using the built-in **update()** method. Nested key values can also be added to an existing Dictionary.
Note- While adding a value, if the key-value already exists, the value gets updated otherwise a new Key with the value is added to the Dictionary.
Example: Add Items to a Python Dictionary with Different DataTypes
The code starts with an empty dictionary and then adds key-value pairs to it. It demonstrates adding elements with various data types, updating a key's value, and even nesting dictionaries within the main dictionary. The code shows how to manipulate dictionaries in Python.
Python

```
Dict = {}
print("Empty Dictionary: ")
print(Dict)
Dict[0] = 'Geeks'
Dict[2] = 'For'
Dict[3] = 1
print("\nDictionary after adding 3 elements: ")
print(Dict)

Dict['Value_set'] = 2, 3, 4
print("\nDictionary after adding 3 elements: ")
print(Dict)

Dict[2] = 'Welcome'
print("\nUpdated key value: ")
print(Dict)
Dict[5] = {'Nested': {'1': 'Life', '2': 'Geeks'}}
print("\nAdding a Nested Key: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}
Dictionary after adding 3 elements:
{0: 'Geeks', 2: 'For', 3: 1}
Dictionary after adding 3 elements:
{0: 'Geeks', 2: 'For', 3: 1, 'Value_set': (2, 3, 4)}
Updated key value:
{0: 'Geeks', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4)}
Adding a Nested Key:
{0: 'Geeks', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4), 5:
```

```
{'Nested': {'1': 'Life', '2': 'Geeks'}}
```

Complexities for Adding Elements in a Dictionary:

- **Time complexity:** O(1)/O(n)
- **Space complexity:** O(1)

Accessing Elements of a Dictionary

To access the items of a dictionary refer to its key name. Key can be used inside square brackets.

Access a Value in Python Dictionary

The code demonstrates how to access elements in a dictionary using keys. It accesses and prints the values associated with the keys ‘name’ and 1, showcasing that keys can be of different data types (string and integer).

Python

```
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}
print("Accessing a element using key:")
print(Dict['name'])
print("Accessing a element using key:")
print(Dict[1])
```

Output:

```
Accessing a element using key:
For
Accessing a element using key:
Geeks
```

There is also a method called **get()** that will also help in accessing the element from a dictionary. This method accepts key as argument and returns the value.

Complexities for Accessing elements in a Dictionary:

- **Time complexity:** O(1)
- **Space complexity:** O(1)

Access a Value in Dictionary using get() in Python

The code demonstrates accessing a dictionary element using the **get()** method

```
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

print("Accessing a element using get:")
print(Dict.get(3))
```

Output:

```
Accessing a element using get:
Geeks
```

Accessing an Element of a Nested Dictionary

To access the value of any key in the nested dictionary, use indexing [] syntax.

```
Dict = {'Dict1': {1: 'Geeks'},
       'Dict2': {'Name': 'For'}}

print(Dict['Dict1'])
print(Dict['Dict1'][1])
print(Dict['Dict2']['Name'])
```

Output:

```
{1:                                                    'Geeks'}
Geeks
For
```

Deleting Elements using 'del' Keyword

The items of the dictionary can be deleted by using the del keyword as given below.

```
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

print("Dictionary =")
print(Dict)
del(Dict[1])
print("Data after deletion Dictionary=")
print(Dict)
```

Output

```
Dictionary ={1: 'Geeks', 'name': 'For', 3: 'Geeks'}
Data after deletion Dictionary={'name': 'For', 3: 'Geeks'}
```

Dictionary Methods

Here is a list of in-built dictionary functions with their description.

Method	Description
dict.clear()	Remove all the elements from the dictionary

Method	Description
dict.copy()	Returns a copy of the dictionary
dict.get(key, default = “None”)	Returns the value of specified key
dict.items()	Returns a list containing a tuple for each key value pair
dict.keys()	Returns a list containing dictionary’s keys
dict.update(dict2)	Updates dictionary with specified key-value pairs
dict.values()	Returns a list of all the values of dictionary
pop()	Remove the element with specified key
popItem()	Removes the last inserted key-value pair
dict.setdefault(key,default= “None”)	set the key to the default value if the key is not

Method	Description
	specified in the dictionary
dict.has_key(key)	returns true if the dictionary contains the specified key.

-Click go to First page

Cl

List

In Python, a list is a collection data type that is ordered, mutable, and allows duplicate elements. Lists are one of the most commonly used data structures and provide a versatile way to work with sequences of elements. Here is a detailed explanation of lists in Python:

Creating a List

You can create a list by placing elements inside square brackets [], separated by commas.

```
# Creating a list with different types of elements
my_list = [1, 2, 3, "apple", "banana", 4.5, True]
print(my_list)
```

Accessing Elements

You can access elements in a list using indexing. Python uses zero-based indexing, meaning the first element has an index of 0

```
# Accessing the first element
print(my_list[0])  # Output: 1
```

```
# Accessing the fourth element
print(my_list[3])  # Output: apple
```

```
# Accessing the last element
print(my_list[-1])  # Output: True
```

Slicing a List

You can retrieve a subset of a list by using slicing. Slicing is done using the colon : operator.

```
# Slicing from index 1 to 4 (excluding index 4)
print(my_list[1:4])  # Output: [2, 3, 'apple']
```

```
# Slicing from the beginning to index 3 (excluding index 3)
print(my_list[:3])  # Output: [1, 2, 3]
```

```
# Slicing from index 2 to the end
print(my_list[2:])  # Output: [3, 'apple', 'banana', 4.5, True]
```

Modifying Elements

Since lists are mutable, you can change their elements.

```
# Changing the second element
my_list[1] = "orange"
print(my_list)  # Output: [1, 'orange', 3, 'apple', 'banana', 4.5, True]
```

Adding Elements

You can add elements to a list using methods like append(), insert(), and extend().

```
# Appending an element to the end of the list
my_list.append("grape")
print(my_list)  # Output: [1, 'orange', 3, 'apple', 'banana', 4.5, True, 'grape']

# Inserting an element at a specific index
my_list.insert(2, "cherry")
print(my_list)  # Output: [1, 'orange', 'cherry', 3, 'apple', 'banana', 4.5, True, 'grape']

# Extending the list with another list
my_list.extend([7, 8, 9])
print(my_list)  # Output: [1, 'orange', 'cherry', 3, 'apple', 'banana', 4.5, True, 'grape', 7,
8, 9]
```

Removing Elements

You can remove elements from a list using methods like remove(), pop(), and del.

```
# Removing a specific element by value
my_list.remove("banana")
print(my_list)  # Output: [1, 'orange', 'cherry', 3, 'apple', 4.5, True, 'grape', 7, 8, 9]

# Removing an element by index
my_list.pop(3)
print(my_list)  # Output: [1, 'orange', 'cherry', 'apple', 4.5, True, 'grape', 7, 8, 9]

# Removing the last element
my_list.pop()
print(my_list)  # Output: [1, 'orange', 'cherry', 'apple', 4.5, True, 'grape', 7, 8]

# Deleting an element by index using `del`
del my_list[2]
print(my_list)  # Output: [1, 'orange', 'apple', 4.5, True, 'grape', 7, 8]
```

List Comprehensions

List comprehensions provide a concise way to create lists. They consist of brackets containing an expression followed by a for clause.

python

Copy code

```
# Creating a list of squares from 0 to 9
squares = [x**2 for x in range(10)]
print(squares)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Common List Methods

- append(x): Add an item to the end of the list.
- extend(iterable): Extend the list by appending all the items from the iterable.
- insert(i, x): Insert an item at a given position.
- remove(x): Remove the first item from the list whose value is equal to x.
- pop([i]): Remove the item at the given position in the list and return it.
- clear(): Remove all items from the list.
- index(x[, start[, end]]): Return the index in the list of the first item whose value is equal to x.
- count(x): Return the number of times x appears in the list.
- sort(key=None, reverse=False): Sort the items of the list in place.
- reverse(): Reverse the elements of the list in place.
- copy(): Return a shallow copy of the list.

Lists are powerful and versatile, making them a fundamental part of Python programming.

Difference Between List and Array in Python

List	Array
Can consist of elements belonging to different data types	Only consists of elements belonging to the same data type
No need to explicitly import a module for the declaration	Need to explicitly import the **array** module for declaration
Cannot directly handle arithmetic operations	Can directly handle arithmetic operations
Preferred for a shorter sequence of data items	Preferred for a longer sequence of data items

List	Array
Greater flexibility allows easy modification (addition, deletion) of data	Less flexibility since addition, and deletion has to be done element-wise
The entire list can be printed without any explicit looping	A loop has to be formed to print or access the components of the array
Consume larger memory for easy addition of elements	Comparatively more compact in memory size
Nested lists can be of variable size	Nested arrays has to be of same size.
Can perform direct operations using functions like: count() – for counting a particular element in the list sort() – sort the complete list max() – gives maximum of the list min() – gives minimum of the list sum() – gives sum of all the elements in list for integer list index() – gives first index of the element specified append() – adds the element to the end of the list remove() – removes the element specified No need to import anything to use these functions. and many more…	Need to import proper modules to perform these operations.
Example: my_list = [1, 2, 3, 4]	**Example:** import array arr = array.array('i', [1, 2, 3])

-Go to first page

CLICK HERE TO JOIN OUR WHATSAPP GROUP

Control flow loop

Python While Loop

Until a specified criterion is true, a block of statements will be continuously executed in a Python while loop. And the line in the program that follows the loop is run when the condition changes to false.

Syntax of Python While

```
while expression:
    statement(s)
```

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

- Python3

```
# prints Hello Geek 3 Times

count = 0

while (count < 3):

    count = count+1

    print("Hello Geek")
```

Output:

```
Hello Geek

Hello Geek

Hello Geek
```

See this for an example where a while loop is used for iterators. As mentioned in the article, it is not recommended to use a while loop for iterators in python.

Python for Loop

In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is a "**for in**" loop which is similar to for each loop in other languages.

Syntax of Python for Loop

```
for iterator_var in sequence:
    statements(s)
```

It can be used to iterate over iterators and a range.

- Python3

```
# Iterating over a list

print("List Iteration")

l = ["geeks", "for", "geeks"]

for i in l:

    print(i)



# Iterating over a tuple (immutable)

print("\nTuple Iteration")

t = ("geeks", "for", "geeks")

for i in t:

    print(i)
```

```
# Iterating over a String

print("\nString Iteration")

s = "Geeks"

for i in s :

    print(i)



# Iterating over dictionary

print("\nDictionary Iteration")

d = dict()

d['xyz'] = 123

d['abc'] = 345

for i in d :

    print("%s %d" %(i, d[i]))
```

Output:

```
List Iteration
geeks
for
geeks


Tuple Iteration
geeks
```

```
for
geeks


String Iteration
G
e
e
k
s


Dictionary Iteration
xyz  123
abc  345
```

Time complexity: O(n), where n is the number of elements in the iterable (list, tuple, string, or dictionary).
Auxiliary space: O(1), as the space used by the program does not depend on the size of the iterable.
We can use a for-in loop for user-defined iterators. See this for example.

Python Nested Loops

Python programming language allows using one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax of Python Nested for Loop

The syntax for a nested for loop statement in Python programming language is as follows:

```
for iterator_var in sequence:
    for iterator_var in sequence:
        statements(s)
        statements(s)
```

Syntax of Python Nested while Loop

The syntax for a nested while loop statement in Python programming language is as follows:

```
while expression:
   while expression:
      statement(s)
      statement(s)
```

A final note on loop nesting is that we can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

- Python3

```
from __future__ import print_function

for i in range(1, 5):
    for j in range(i):
        print(i, end=' ')
    print()
```

Output:

```
1
2 2
3 3 3
4 4 4 4
```

Python Loop Control Statements

Loop control statements change execution from their normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

Python Continue

It returns the control to the beginning of the loop.

- Python3

```
# Prints all letters except 'a' and 'u'
for letter in 'gautam':
    if letter == 'a' or letter == 'u':
        continue
    print('Current Letter :', letter)
```

Output:

```
Current Letter : g
Current Letter : t
Current Letter : m
```

Python Break

It brings control out of the loop.

- Python3

```
for letter in 'geeksforgeeks':

    # break the loop as soon it sees 'e'
    # or 's'
    if letter == 'e' or letter == 's':
        break
```

```
print('Current Letter :', letter)
```

Output:

```
Current Letter : e
```

Python Pass

We use pass statements to write empty loops. Pass is also used for empty control statements, functions, and classes.

- Python3

```
# An empty loop

for letter in 'geeksforgeeks':

    pass

print('Last Letter :', letter)
```

Output:

```
Last Letter : s
```

Click to go in front page

CLICK HERE TO JOIN OUR WHATSAPP GROUP

Control Flow : Conditional block

In both real life and programming, decision-making is crucial. We often face situations where we need to make choices, and based on those choices, we determine our next actions. Similarly, in programming, we encounter scenarios where we must make decisions to control the flow of our code.
Conditional statements in Python play a key role in determining the direction of program execution. Among these, If-Else statements are fundamental, providing a way to execute different blocks of code based on specific conditions. As the name suggests, If-Else statements offer two paths, allowing for different outcomes depending on the condition evaluated.

Types of Control Flow in Python

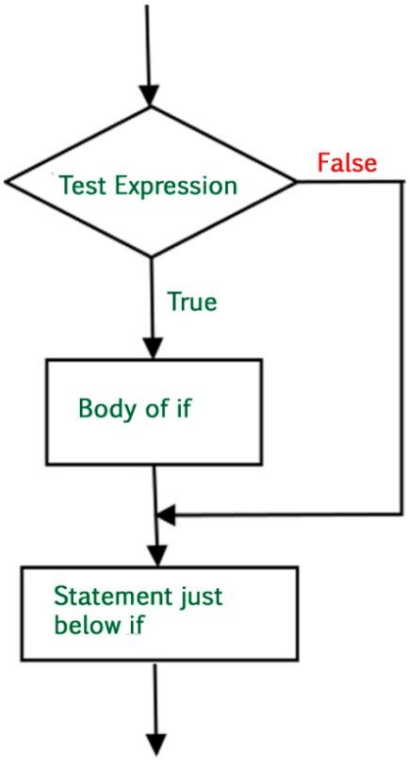
- Python If Statement
- Python If Else Statement
- Python Nested If Statement
- Python Elif
- Ternary Statement | Short Hand If Else Statement

Python If Statement

The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not.

Flowchart of If Statement

Let’s look at the flow of code in the Python If statements.



Flowchart of Python if statement

Syntax of If Statement in Python

Here, the condition after evaluation will be either true or false. if the statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not.

```
#if syntax Python

if condition:
  # Statements to execute if
  # condition is true
```

As we know, Python uses indentation to identify a block. So the block under the Python if statements will be identified as shown in the below example:

```
if condition:
  statement1
statement2
# Here if the condition is true, if block
# will consider only statement1 to be inside
# its block.
```

Example of Python if Statement

As the condition present in the if statements in Python is false. So, the block below the if statement is executed.

Python

```
# python program to illustrate If statement
i = 10
```

```
if (i > 15):
    print("10 is less than 15")
print("I am Not in if")
```

Output:

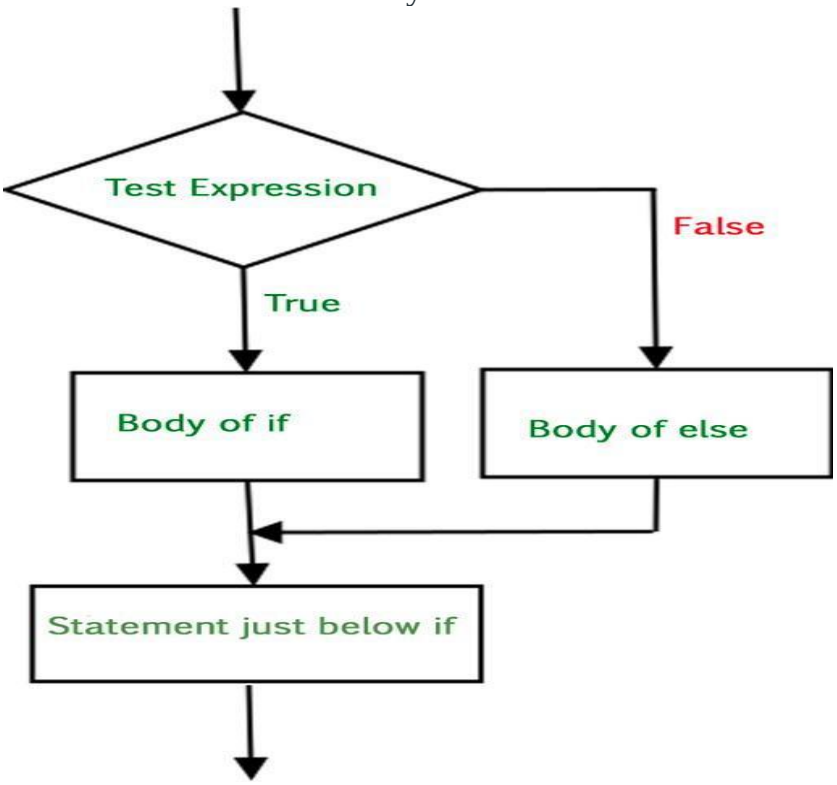
```
I am Not in if
```

Python If Else Statement

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But if we want to do something else if the condition is false, we can use the else statement with the if statement Python to execute a block of code when the Python if condition is false.

Flowchart of If Else Statement

Let's look at the flow of code in an if else Python statement.



Syntax of If Else in Python

```
if(condition):
    #Executesthisblockif
    #conditionistrue
else:
    #Executesthisblockif
    # condition is false
```

Example of Python If Else Statement

The block of code following the else if in Python, the statement is executed as the condition present in the if statement is false after calling the statement which is not in the block(without spaces).

Python

```
# python program to illustrate else if in Python statement
#!/usr/bin/python

i = 20
if (i < 15):
    print("i is smaller than 15")
    print("i'm in if Block")
else:
    print("i is greater than 15")
    print("i'm in else Block")
print("i'm not in if and not in else Block")
```

Output:

```
iisgreaterthan15
i'minelseBlock
i'm not in if and not in else Block
```

If Else in Python using List Comprehension

In this example, we are using an Python else if statement in a list comprehension with the condition that if the element of the list is odd then its digit sum will be stored else not.

Python

```
# Explicit function
def digitSum(n):
    dsum = 0
    for ele in str(n):
        dsum += int(ele)
    return dsum


# Initializing list
List = [367, 111, 562, 945, 6726, 873]

# Using the function on odd elements of the list
newList = [digitSum(i) for i in List if i & 1]

# Displaying new list
print(newList)
```

Output :

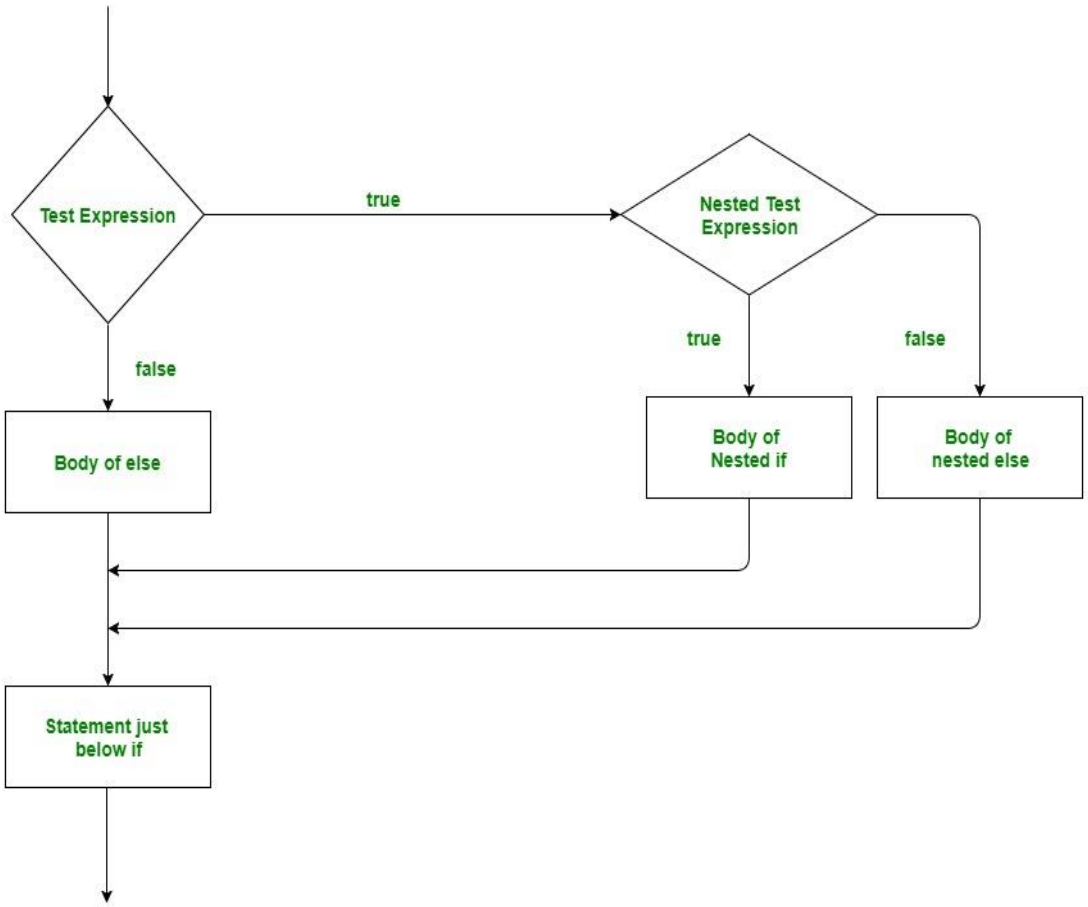
```
[16, 3, 18, 18]
```

Python Nested If Statement

A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement.

Yes, Python allows us to nest if statements within if statements. i.e., we can place an if statement inside another if statement.

Flowchart of Python Nested if Statement



Flowchart of Python Nested if statement

Syntax:

```
  if(condition1):
  #Executeswhencondition1istrue
  if(condition2):
    #Executeswhencondition2istrue
  #ifBlockisendhere
# if Block is end here
```

Example of Python Nested If Statement

In this example, we are showing nested if conditions in the code, All the If condition in Python will be executed one by one.

Python

```
# python program to illustrate nested If statement

i = 10
if (i == 10):
```

```
# First if statement
if (i < 15):
    print("i is smaller than 15")

# Nested - if statement
# Will only be executed if statement above
# it is true
if (i < 12):
    print("i is smaller than 12 too")
else:
    print("i is greater than 15")
```

Output:

```
i is smaller than 15
i is smaller than 12 too
```

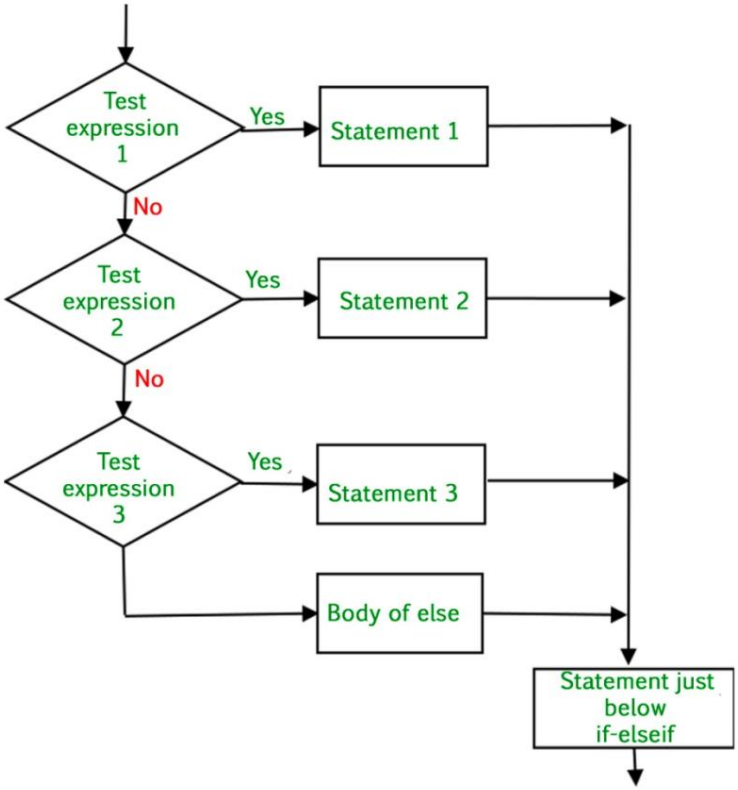
Python Elif

Here, a user can decide among multiple options. The if statements are executed from the top down.

As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final "else" statement will be executed.

Flowchart of Elif Statement in Python

Let's look at the flow of control in if-elif-else ladder:



Flowchart of if-elif-else ladder

Syntax:

```
if(condition):
    statement
elif(condition):
    statement
.
.
else:
    statement
```

Example of Python if-elif-else ladder

In the example, we are showing single if in Python, multiple elif conditions, and single else condition.

Python

```
# Python program to illustrate if-elif-else ladder
#!/usr/bin/python

i = 25
if (i == 10):
    print("i is 10")
elif (i == 15):
    print("i is 15")
elif (i == 20):
    print("i is 20")
else:
    print("i is not present")
```

Output:

```
i is not present
```

Ternary Statement | Short Hand If Else Statement

Whenever there is only a single statement to be executed inside the if block then shorthand if can be used. The statement can be put on the same line as the if statement.

Example of Python If shorthand

In the given example, we have a condition that if the number is less than 15, then further code will be executed.

```
if condition: statement
```

Python

```
# Python program to illustrate short hand if
i = 10
if i < 15: print("i is less than 15")
```

Output

```
i is less than 15
```

Example of Short Hand If Else Statements

This can be used to write the if-else statements in a single line where only one statement is needed in both the if and else blocks.

Syntax: statement_when_True **if** condition **else** statement_when_False

In the given example, we are printing True if the number is 15, or else it will print False.

Python

```
# Python program to illustrate short hand if-else
i = 10
print(True) if i < 15 else print(False)
```

Output:

```
True
```

Similar Reads:

- **Python3 – if , if..else, Nested if, if-elif statements**
- **Using Else Conditional Statement With For loop in Python**
- **How to use if, else & elif in Python Lambda Functions**

Python If Else Statements – Conditional Statements

What is the conditional statement of if-else?

The if-else statement in Python is used to control the flow of the program based on a condition. It has the following syntax:

```
if..condition:
    #Execute..this..block..if..conditions..True
else:
    # Execute this block if condition is False
```

For example:

```
x=10
if..x>5:
    print("x..is..greater..than..5")
else:
    print("x is not greater than 5")
```

How many else statements can a single if condition have in Python?

A single if condition can have at most one else statement. However, you can have multiple elif (else if) statements to check additional conditions if needed:

```
x=10
if..x>15:
    print("x..is..greater..than..15")
elif..x>5:
    print("x..is..greater..than..5..but..not..greater..than..15")
else:
    print("x is 5 or less")
```

What are the different types of control statements in Python?

In Python, control statements are used to alter the flow of execution based on specific conditions or looping requirements. The main types of control statements are:

- ***Conditional statements**: if, else, elif*
- ***Looping statements**: for, while*
- ***Control flow statements**: break, continue, pass, return*

What are the two types of control statements?

The two primary types of control statements in Python are:

- ***Conditional statements**: Used to execute code based on certain conditions (if, else, elif).*
- ***Looping statements**: Used to execute code repeatedly until a condition is met (for, while).*

Are control statements and conditional statements the same?

No, control statements and conditional statements are not exactly the same.

- ***Conditional statements** (if, else, elif) specifically deal with checking conditions and executing code based on whether those conditions are True or False.*
- ***Control statements** encompass a broader category that includes both conditional statements (if, else, elif) and looping statements (for, while), as well as other statements (break, continue, pass, return) that control the flow of execution in a program.*

CLICK HERE TO JOIN OUR WHATSAPP GROUP

Basic I/O operators

In Python programming, Operators in general are used to perform operations on values and variables. These are standard symbols used for logical and arithmetic operations. In this article, we will look into different types of **Python operators.**

- OPERATORS: These are the special symbols. Eg- + , * , /, etc.
- OPERAND: It is the value on which the operator is applied.

Types of Operators in Python

1. Arithmetic Operators
2. Comparison Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Identity Operators and Membership Operators

Arithmetic Operators in Python

Python Arithmetic operators are used to perform basic mathematical operations like **addition, subtraction, multiplication**, and **division**.
In Python 3.x the result of division is a floating-point while in Python 2.x division of 2 integers was an integer. To obtain an integer result in Python 3.x floored (// integer) is used.

Operator	Description	Syntax
+	Addition: adds two operands	x + y
–	Subtraction: subtracts two operands	x – y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the	x / y

Operator	Description	Syntax
	second	
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when the first operand is divided by the second	x % y
**	Power: Returns first raised to power second	x ** y

Example of Arithmetic Operators in Python

Division Operators

In Python programming language **Division Operators** allow you to divide two numbers and return a quotient, i.e., the first number or number at the left is divided by the second number or number at the right and returns the quotient.
There are two types of division operators:

1. Float division
2. Floor division

Float division

The quotient returned by this operator is always a float number, no matter if two numbers are integers. For example:
Example: The code performs division operations and prints the results. It demonstrates that both integer and floating-point divisions return accurate results. For example, **'10/2'** results in **'5.0'**, and **'-10/2'** results in **'-5.0'**.
Python

```
print(5/5)
print(10/2)
print(-10/2)
print(20.0/2)
```

Output:

```
1.0
5.0
-5.0
10.0
```

Integer division(Floor division)

The quotient returned by this operator is dependent on the argument being passed. If any of the numbers is float, it returns output in float. It is also known as Floor division because, if any number is negative, then the output will be floored. For example:

Example: The code demonstrates integer (floor) division operations using the **// in Python operators**. It provides results as follows: **'10//3'** equals **'3'**, **'-5//2'** equals **'-3'**, '**5.0//2'** equals **'2.0'**, and **'-5.0//2'** equals **'-3.0'**. Integer division returns the largest integer less than or equal to the division result.

Pythons

```
print(10//3)
print (-5//2)
print (5.0//2)
print (-5.0//2)
```

Output:

```
3
-3
2.0
-3.0
```

Precedence of Arithmetic Operators in Python

1. P – Parentheses
2. E – Exponentiation
3. M – Multiplication (Multiplication and division have the same precedence)
4. D – Division
5. A – Addition (Addition and subtraction have the same precedence)
6. S – Subtraction

The modulus of Python operators helps us extract the last digit/s of a number. For example:

- x % 10 -> yields the last digit
- x % 100 -> yield last two digits

Arithmetic Operators With Addition, Subtraction, Multiplication, Modulo and Power

Here is an example showing how different Arithmetic Operators in Python work:

Example: The code performs basic arithmetic operations with the values of **'a'** and **'b'**. It adds **('+')**, subtracts **('-')**, multiplies **('*')**, computes the

remainder **(‘%’)**, and raises a to the power of **‘b (**)’**. The results of these operations are printed.

Python

```
a = 9
b = 4
add = a + b

sub = a - b

mul = a * b

mod = a % b

p = a ** b
print(add)
print(sub)
print(mul)
print(mod)
print(p)
```

Output:

```
13
5
36
1
6561
```

Comparison of Python Operators

In Python Comparison of Relational operators compares the values. It either returns **True** or **False** according to the condition.

Operator	Description	Syntax
>	Greater than: True if the left operand is greater than the right	x > y
<	Less than: True if the left operand is less than the right	x < y
==	Equal to: True if both	x == y

Operator	Description	Syntax
	operands are equal	
!=	Not equal to – True if operands are not equal	x != y
>=	Greater than or equal to True if the left operand is greater than or equal to the right	x >= y
<=	Less than or equal to True if the left operand is less than or equal to the right	x <= y

= is an assignment operator and == comparison operator.

Precedence of Comparison Operators in Python

In Python, the comparison operators have lower precedence than the arithmetic operators. All the operators within comparison operators have the same precedence order.

Example of Comparison Operators in Python

Let's see an example of Comparison Operators in Python.

Example: The code compares the values of **'a'** and **'b'** using various comparison Python operators and prints the results. It checks if **'a'** is greater than, less than, equal to, not equal to, greater than, or equal to, and less than or equal to **'b'**.

Python

```
a = 13
b = 33

print(a > b)
print(a < b)
print(a == b)
print(a != b)
print(a >= b)
print(a <= b)
```

Output

```
False

True
```

```
False
True
False
True
```

Logical Operators in Python

Python Logical operators perform **Logical AND**, **Logical OR**, and **Logical NOT** operations. It is used to combine conditional statements.

Operator	Description	Syntax
And	Logical AND: True if both the operands are true	x and y
Or	Logical OR: True if either of the operands is true	x or y
Not	Logical NOT: True if the operand is false	not x

Precedence of Logical Operators in Python

The precedence of Logical Operators in Python is as follows:

1. Logical not
2. logical and
3. logical or

Example of Logical Operators in Python

The following code shows how to implement Logical Operators in Python:

Example: The code performs logical operations with Boolean values. It checks if both **'a'** and **'b'** are true (**'and'**), if at least one of them is true (**'or'**), and negates the value of **'a'** using **'not'**. The results are printed accordingly.

Python

```
a = True
b = False
print(a and b)
print(a or b)
```

```
print(not a)
```

Output

```
False

True

False
```

Bitwise Operators in Python

Python Bitwise operators act on bits and perform bit-by-bit operations. These are used to operate on binary numbers.

Operator	Description	Syntax
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Precedence of Bitwise Operators in Python

The precedence of Bitwise Operators in Python is as follows:

1. Bitwise NOT
2. Bitwise Shift
3. Bitwise AND
4. Bitwise XOR
5. Bitwise OR

Bitwise Operators in Python

Here is an example showing how Bitwise Operators in Python work:

Example: The code demonstrates various bitwise operations with the values of **'a'** and **'b'**. It performs bitwise **AND (&)**, **OR (|)**, **NOT (~)**, **XOR (^)**, **right shift (>>)**, and **left shift (<<)** operations and prints the results. These operations manipulate the binary representations of the numbers.

Python

```
a = 10
b = 4
print(a & b)
print(a | b)
print(~a)
print(a ^ b)
print(a >> 2)
print(a << 2)
```

Output

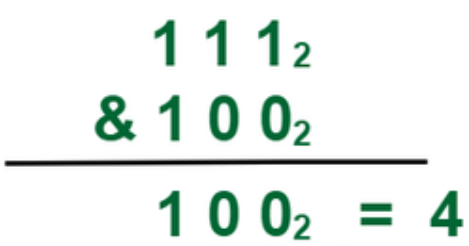
```
0

14

-11

14

2

40
```

Bitwise AND Operator

The **Python Bitwise AND (&)** operator takes two equal-length bit patterns as parameters. The two-bit integers are compared. If the bits in the compared positions of the bit patterns are 1, then the resulting bit is 1. If not, it is 0.

Example: Take two bit values X and Y, where X = 7= (111)2 and Y = 4 = (100)2 . Take Bitwise and of both X & y

Note: Here, (111)2 represent binary number.



Python

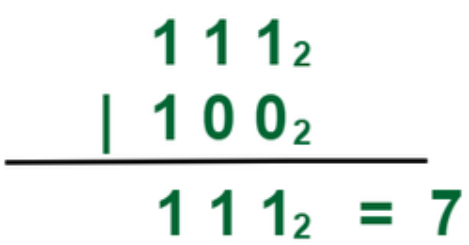
```
a = 10
b = 4

# Print bitwise AND operation
print("a & b =", a & b)
```

Output

```
a & b = 0
```

Bitwise OR Operator

The **Python Bitwise OR (|)** Operator takes two equivalent length bit designs as boundaries; if the two bits in the looked-at position are 0, the next bit is zero. If not, it is 1.
Example: Take two bit values X and Y, where X = 7= (111)2 and Y = 4 = (100)2 . Take Bitwise OR of both X, Y



Python

```
a = 10
b = 4

# Print bitwise OR operation
print("a | b =", a | b)
```

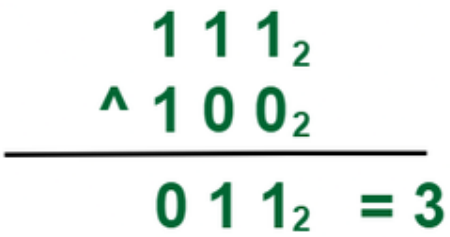
Output

```
a | b = 14
```

Bitwise XOR Operator

The **Python Bitwise XOR (^) Operator** also known as the exclusive OR operator, is used to perform the XOR operation on two operands. XOR stands for “exclusive or”, and it returns true if and only if exactly one of the operands is true. In the

context of bitwise operations, it compares corresponding bits of two operands. If the bits are different, it returns 1; otherwise, it returns 0.
Example: Take two bit values X and Y, where X = 7= (111)2 and Y = 4 = (100)2 . Take Bitwise and of both X & Y



Python

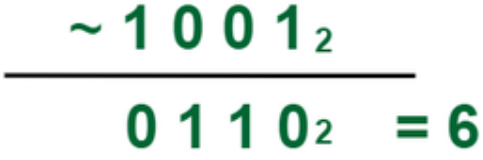
```
a = 10
b = 4

# print bitwise XOR operation
print("a ^ b =", a ^ b)
```

Output

```
a ^ b = 14
```

Bitwise NOT Operator

The preceding three bitwise operators are binary operators, necessitating two operands to function. However, unlike the others, this operator operates with only one operand.
The **Python Bitwise Not (~) Operator** works with a single value and returns its one's complement. This means it toggles all bits in the value, transforming 0 bits to 1 and 1 bits to 0, resulting in the one's complement of the binary number.
Example: Take two bit values X and Y, where X = 5= (101)2 . Take Bitwise NOT of X.



Python

```
a = 10
b = 4

# Print bitwise NOT operation
print("~a =", ~a)
```

Output

```
~a = -11
```

Bitwise Shift

These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two.

Python Bitwise Right Shift

Shifts the bits of the number to the right and fills 0 on voids left(fills 1 in the case of a negative number) as a result. Similar effect as of dividing the number with some power of two.

Example 1:

a = 10 = 0000 1010 (Binary)
a >> 1 = 0000 0101 = 5

tricks a/2

Example 2:

a = -10 = 1111 0110 (Binary)
a >> 1 = 1111 1011 = -5

Python

```
a = 10
b = -10

# print bitwise right shift operator
print("a >> 1 =", a >> 1)
print("b >> 1 =", b >> 1)
```

Output

```
a >> 1 = 5
b >> 1 = -5
```

Python Bitwise Left Shift

Shifts the bits of the number to the left and fills 0 on voids right as a result. Similar effect as of multiplying the number with some power of two.

Example 1:

a = 5 = 0000 0101 (Binary)
a << 1 = 0000 1010 = 10
a << 2 = 0001 0100 = 20

tricks a*2

Example 2:

b = -10 = 1111 0110 (Binary)
b << 1 = 1110 1100 = -20
b << 2 = 1101 1000 = -40

Python

```
a = 5
b = -10
```

```
# print bitwise left shift operator
print("a << 1 =", a << 1)
print("b << 1 =", b << 1)
```

Output:

```
a << 1 = 10
b << 1 = -20
```

Assignment Operators in Python

Python Assignment operators are used to assign values to the variables.

Operator	Description	Syntax
=	Assign the value of the right side of the expression to the left side operand	x = y + z
+=	Add AND: Add right-side operand with left-side operand and then assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a*=b a=a*b
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and	a%=b a=a%b

Operator	Description	Syntax
	right operands and assign the result to left operand	
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
**=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a**=b a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
|=	Performs Bitwise OR on operands and assign value to left operand	a|=b a=a|b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and	a <<= b a= a << b

Operator	Description	Syntax
	assign value to left operand	

Assignment Operators in Python

```
a = 10
b = a
print(b)
b += a
print(b)
b -= a
print(b)
b *= a
print(b)
b <<= a
print(b)
```

Output

```
10

20

10

100

102400
```

Identity Operators in Python

In Python, **is** and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

```
is..True..if..the..operands..are..identical
is..not..True..if..the..operands..are..not..identical
```

Example Identity Operators in Python

Let’s see an example of Identity Operators in Python.

Example: The code uses identity operators to compare variables in Python. It checks if **'a'** is not the same object as **'b'** (which is true because they have different values) and if **'a'** is the same object as **'c'** (which is true because **'c'** was assigned the value of **'a'**).

Python

```
a = 10
b = 20
c = a

print(a is not b)
print(a is c)
```

Output

```
True

True
```

Membership Operators in Python

In Python, **in** and **not in** are the membership operators that are used to test whether a value or variable is in a sequence.

in	True if value is found in the sequence
not in	True if value is not found in the sequence

Examples of Membership Operators in Python

The following code shows how to implement Membership Operators in Python:

Example: The code checks for the presence of values **'x'** and **'y'** in the list. It prints whether or not each value is present in the list. **'x'** is not in the list, and **'y'** is present, as indicated by the printed messages. The code uses the **'in'** and **'not in'** Python operators to perform these checks.

Python

```
x = 24
y = 20
list = [10, 20, 30, 40, 50]

if (x not in list):
    print("x is NOT present in given list")
else:
    print("x is present in given list")

if (y in list):
```

```
    print("y is present in given list")
else:
    print("y is NOT present in given list")
```

Output

```
x is NOT present in given list

y is present in given list
```

Ternary Operator in Python

in Python, Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being true or false. It was added to Python in version 2.5.

It simply allows testing a condition in a **single line** replacing the multiline if-else making the code compact.

Syntax : *[on_true] if [expression] else [on_false]*

Examples of Ternary Operator in Python

The code assigns values to variables **'a'** and **'b'** (10 and 20, respectively). It then uses a conditional assignment to determine the smaller of the two values and assigns it to the variable **'min'**. Finally, it prints the value of **'min'**, which is 10 in this case.

Python

```
a, b = 10, 20
min = a if a < b else b

print(min)
```

Output:

```
10
```

Precedence and Associativity of Operators in Python

In Python, Operator precedence and associativity determine the priorities of the operator.

Operator Precedence in Python

This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

Let's see an example of how Operator Precedence in Python works:

Example: The code first calculates and prints the value of the expression **10 + 20 * 30**, which is 610. Then, it checks a condition based on the values of the **'name'** and **'age'** variables. Since the name is "**Alex"** and the condition is satisfied using the or operator, it prints **"Hello! Welcome."**

Python
```
expr = 10 + 20 * 30
print(expr)
name = "Alex"
age = 0

if name == "Alex" or name == "John" and age >= 2:
    print("Hello! Welcome.")
else:
    print("Good Bye!!")
```

Output
```
610

Hello! Welcome.
```

Operator Associativity in Python

If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be Left to Right or from Right to Left.

The following code shows how Operator Associativity in Python works:

Example: The code showcases various mathematical operations. It calculates and prints the results of division and multiplication, addition and subtraction, subtraction within parentheses, and exponentiation. The code illustrates different mathematical calculations and their outcomes.

Python
```
print(100 / 10 * 10)
print(5 - 2 + 3)
print(5 - (2 + 3))
print(2 ** 3 ** 2)
```

Output
```
100.0

6

0

512
```

Click to go in front page

CLICK HERE TO JOIN OUR WHATSAPP GROUP

Fundamental Concepts

What is Python

Python's simplicity, readability, and versatility make it an excellent choice for beginners and experienced programmers alike. In this article, we've covered the basics of Python, from setting up your environment to writing your first program and understanding syntax, control flow, and functions. As you continue your journey with Python Basics, don't hesitate to explore its vast ecosystem of libraries, frameworks, and tools to unleash its full potential in various domains of programming.

Writing your first Python Program

Here we provided the latest Python 3 version compiler where you can edit and compile your written code directly with just one click of the RUN Button. So test yourself with Python's first exercises.

Python

```
print("Hello World! I Don't Give a Bug")
```

Output

```
Hello World! I Don't Give a Bug
```

Comments in Python

Comments in Python are the lines in the code that are ignored by the interpreter during the execution of the program. Also, Comments enhance the readability of the code and help the programmers to understand the code very carefully.

Python

```
# sample comment
# This is Python Comment
name = "geeksforgeeks"
print(name)
```

Output

```
geeksforgeeks
```

Keywords in Python

Keywords in Python are reserved words that can not be used as a variable name, function name, or any other identifier.

Keywords		
and	False	nonlocal
as	finally	not
assert	for	Or
break	from	pass
class	global	raise
continue	if	return
def	import	True
del	is	try
elif	in	while
else	lambda	with
except	None	yield

Python Variable

Python Variable is containers that store values. Python is not “statically typed”. An Example of a Variable in Python is a representational name that serves as a pointer to an object. Once an object is assigned to a variable, it can be referred to by that name.

Rules for Python variables

- A Python variable name must start with a letter or the underscore character.
- A Python variable name cannot start with a number.
- A Python variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable in Python names are case-sensitive (name, Name, and NAME are three different variables).
- The reserved words(keywords) in Python cannot be used to name the variable in Python.

Example

Python

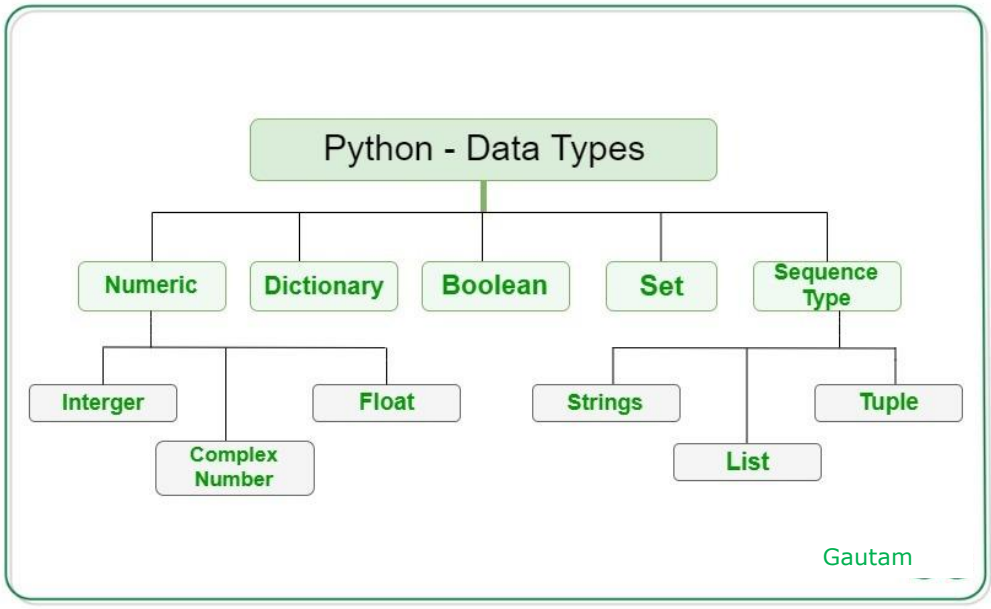
```
# An integer assignment
age = 45

# A floating point
salary = 1456.8

# A string
name = "John"

print(age)
print(salary)
print(name)
```

Output

```
45

1456.8

John
```

Python Data Types

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are classes and variables are instances (objects) of these classes.



Example: This code assigns variable **'x'** different values of various data types in Python.

Python

```
x = "Hello World" # string
x = 50  # integer
x = 60.5  # float
x = 3j  # complex
x = ["geeks", "for", "geeks"]  # list
x = ("geeks", "for", "geeks")  # tuple
x = {"name": "Gautam", "age":16} # dict
x = {"geeks", "for", "geeks"} # set
x = True  # bool
```

Click to go in front page

CLICK HERE TO JOIN OUR WHATSAPP GROUP