**Data Types :-**

Basically data type represents the type of value and determines how the value can be used in a python program. Since all the data values are encapsulated in relevant object classes. Everything in Python, is simply an object and every object has an identity, a type and a value.

The data stored in memory can be of many types.

**4 For example:-**

**A** person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

**Following are the standard or built-in data type of Python:**

* [**Numeric**](https://www.geeksforgeeks.org/python-data-types/#numeric)
* [**Sequence Type**](https://www.geeksforgeeks.org/python-data-types/#Sequence)
* [**Boolean**](https://www.geeksforgeeks.org/python-data-types/#boolean)
* [**Set**](https://www.geeksforgeeks.org/python-data-types/#set)
* [**Dictionary**](https://www.geeksforgeeks.org/python-data-types/#dictionary)



**1) Numeric :-**

In Python, numeric data type represent the data which has numeric value. Numeric value can be integer, floating number or even complex numbers. These values are defined as int, float and complex class in Python.

* Integers – This value is represented by int class. It contains positive or negative whole numbers (without fraction or decimal). In Python there is no limit to how long an integer value can be.
* Float – This value is represented by float class. It is a real number with floating point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
* Complex Numbers – Complex number is represented by complex class. It is specified as (real part) + (imaginary part)j. For example – 2+3j

Example:-

a = 5

print("Type of a: ", type(a))

b = 5.0

print("\nType of b: ", type(b))

c = 2 + 4j

print("\nType of c: ", type(c))

# Sequence Type

In Python, sequence is the ordered collection of similar or different data types. Sequences allows to store multiple values in an organized and efficient fashion. There are several sequence types in Python –

* [String](https://www.geeksforgeeks.org/python-data-types/#string)
* [List](https://www.geeksforgeeks.org/python-data-types/#list)
* [Tuple](https://www.geeksforgeeks.org/python-data-types/#tuple)

## 1) String

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:] ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (\*) is the repetition operator.

**For example −**

str = 'Hello World!'

print str

print str[0]

print str[2:5]

print str[2:]

print str \* 2

print str + "TEST"

**Output:-**

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

**2) Python Lists:-**

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (\*) is the repetition operator.

**For example −**

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tinylist = [123, 'john']

print list

print list[0]

print list[1:3]

print list[2:]

print tinylist \* 2

print list + tinylist

**Output:-**

['abcd', 786, 2.23, 'john', 70.200000000000003]

abcd

[786, 2.23]

[2.23, 'john', 70.200000000000003]

[123, 'john', 123, 'john']

['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']

**3) Python Tuples:-**

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.The main **differences between lists and tuples are:** Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of as **read-only** lists.

**For example −**

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

tinytuple = (123, 'john')

print tuple

print tuple[0]

print tuple[1:3]

print tuple[2:]

print tinytuple \* 2

print tuple + tinytuple

**Output:-**

('abcd', 786, 2.23, 'john', 70.200000000000003)

abcd

(786, 2.23)

(2.23, 'john', 70.200000000000003)

(123, 'john', 123, 'john')

('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tuple[2] = 1000

list[2] = 1000

**Boolean**

Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false. It is denoted by the class bool.

**Note**– True and False with capital ‘T’ and ‘F’ are valid booleans otherwise python will throw an error.

**Example :-**

|  |
| --- |
| print(type(True))  print(type(False))  print(type(true)) |

# Set

In Python, [Set](https://www.geeksforgeeks.org/python-sets/) is an unordered collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

#### Creating Sets

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by ‘comma’. Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

**Example:-**

set1 = set()

print("Intial blank Set: ")

print(set1)

set1 = set("GeeksForGeeks")

print("\nSet with the use of String: ")

print(set1)

set1 = set(["Geeks", "For", "Geeks"])

print("\nSet with the use of List: ")

print(set1)

set1 = set([1, 2, 'Geeks', 4, 'For', 6, 'Geeks'])

print("\nSet with the use of Mixed Values")

print(set1)

#### Accessing elements of Sets

Set items cannot be accessed by referring to an index, since sets are unordered the items has no index. But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

**Example:-**

set1 = set(["Geeks", "For", "Geeks"])

print("\nInitial set")

print(set1)

print("\nElements of set: ")

for i in set1:

    print(i, end =" ")

print("Geeks" in set1)

**Python Dictionary:-**

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

**For example −**

dict = {}

dict['one'] = "This is one"

dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print dict['one']

print dict[2]

print tinydict

print tinydict.keys()

print tinydict.values()

**Output:-**

This is one

This is two

{'dept': 'sales', 'code': 6734, 'name': 'john'}

['dept', 'code', 'name']

['sales', 6734, 'john']

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

# Python Literals

Literals can be defined as a data that is given in a variable or constant.

Python support the following literals:

**I. String literals:**

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes for a String.

**Eg:**

"Aman" , '12345'

**Types of Strings:**

There are two types of Strings supported in Python:

a).Single line String-

Strings that are terminated within a single line are known as Single line Strings.

**Eg:**

text1='hello'

b).Multi line String-

A piece of text that is spread along multiple lines is known as Multiple line String.

There are two ways to create Multiline Strings:

**1). Adding black slash at the end of each line.**

**Eg:**

text1='hello \

user'

print text1

**2).Using triple quotation marks:-**

**Eg:**

str2='''''welcome

to

SSSIT'''

print str2

**Output:-**

welcome

to

SSSIT

**II.Numeric literals:**

Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

|  |  |  |  |
| --- | --- | --- | --- |
| **Int(signed integers)** | **Long(long integers)** | **float(floating point)** | **Complex(complex)** |
| Numbers( can be both positive and negative) with no fractional part.eg: 100 | Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L | Real numbers with both integer and fractional part eg: -26.2 | In the form of a+bj where a forms the real part and b forms the imaginary part of complex number. eg: 3.14j |

**III. Boolean literals:**

A Boolean literal can have any of the two values: True or False.

**IV. Special literals.**

Python contains one special literal i.e., None.

None is used to specify to that field that is not created. It is also used for end of lists in Python.

**Example:**

val1=10

val2=None

print val1

**print** val2

**V.Literal Collections.**

Collections such as tuples, lists and Dictionary are used in Python.

**List:**

* List contain items of different data types. Lists are mutable i.e., modifiable.
* The values stored in List are separated by commas(,) and enclosed within a square brackets([]). We can store different type of data in a List.
* Value stored in a List can be retrieved using the slice operator([] and [:]).
* The plus sign (+) is the list concatenation and asterisk(\*) is the repetition operator.

**Example:-**

 list=['aman',678,20.4,'saurav']

list1=[456,'rahul']

print  list

print   list[1:3]

print   list+list1

print   list1\*2

**Constants**

A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later.Non technically, you can think of constant as a bag to store some books and those books cannot be replaced once placed inside the bag.

**Assigning value to a constant in Python**

constants are usually declared and assigned on a module. Here, the module means a new file containing variables, functions etc which is imported to main file. Inside the module, constants are written in all capital letters and underscores separating the words.

**Example 3: Declaring and assigning value to a constant**

constant.py

PI = 3.14

GRAVITY = 9.8

**main.py**

import constant

print(constant.PI)

print(constant.GRAVITY)

**Type conversion:-**

Python defines type conversion functions to directly convert one data type to another which is useful in day-to-day and competitive programming. This article is aimed at providing information about certain conversion functions.There are two types of Type Conversion in Python:

**1)Implicit Type Conversion**

**2)Explicit Type Conversion**

**1)Implicit Type Conversion**

In Implicit type conversion of data types in Python, the Python interpreter automatically converts one data type to another without any user involvement

**Example:-**

x = 10

print("x is of type:",type(x))

y = 10.6

print("y is of type:",type(y))

z = x + y

print(z)

print("z is of type:",type(z))

**Output:-**

x is of type: <class 'int'>

y is of type: <class 'float'>

20.6

z is of type: <class 'float'>

**2) Explicit Type Conversion:-**

In Explicit Type Conversion in Python, the data type is manually changed by the user as per their requirement. With explicit type conversion, there is a risk of data loss since we are forcing an expression to be changed in some specific data type.

**1. int(a,base)** :

This function converts**any data type to integer**. ‘Base’

specifies the**base in which string is** if data type is string.

**2. float()** :

This function is used to convert **any data type to a floating point number**

**Example:-**

s = "10010"

c = int(s,2)

print ("After converting to integer base 2 : ", end="")

print (c)

 e = float(s)

print ("After converting to float : ", end="")

print (e)

**Output:**

After converting to integer base 2 : 18

After converting to float : 10010.0

**3. ord() :**

This function is used to convert a **character to integer.**

**4. hex() :**

This function is to convert integer to hexadecimal string.

**5. oct() :**

This function is to convert integer to octal string.

**Example:-**

s = '4'

c = ord(s)

print ("After converting character to integer : ",end="")

print (c)

c = hex(56)

print ("After converting 56 to hexadecimal string : ",end="")

print (c)

c = oct(56)

print ("After converting 56 to octal string : ",end="")

print (c)

**Output:**

After converting character to integer : 52

After converting 56 to hexadecimal string : 0x38

After converting 56 to octal string : 0o70

**6. tuple() :**

This function is used to **convert to a tuple**.

**7. set() :**

This function returns the **type after converting to set**.

**8. list() :**

This function is used to convert **any data type to a list type**.

|  |
| --- |
| **Example:-**  # initializing string  s = 'geeks'   c = tuple(s)  print ("After converting string to tuple : ",end="")  print (c)  c = set(s)  print ("After converting string to set : ",end="")  print (c)   c = list(s)  print ("After converting string to list : ",end="")  print (c) |

**Output:**

After converting string to tuple : ('g', 'e', 'e', 'k', 's')

After converting string to set : {'k', 'e', 's', 'g'}

After converting string to list : ['g', 'e', 'e', 'k', 's']

**9. dict() :**

This function is used to **convert a tuple of order (key,value) into a dictionary**.

**10. str() :**

Used to **convert integer into a string.**

**11. complex(real,imag) :**

This function**converts real numbers to complex(real,imag) number.**

|  |
| --- |
| **Example:-**  # initializing integers  a = 1  b = 2  tup = (('a', 1) ,('f', 2), ('g', 3))   c = complex(1,2)  print ("After converting integer to complex number : ",end="")  print (c)   c = str(a)  print ("After converting integer to string : ",end="")  print (c)   c = dict(tup)  print ("After converting tuple to dictionary : ",end="")  print (c) |

**Output:-**

After converting integer to complex number : (1+2j)

After converting integer to string : 1

After converting tuple to dictionary : {'a': 1, 'f': 2, 'g': 3}

**Python Comments:-**

**Comments in Python** are the lines in the code that are ignored by the interpreter during the execution of the program. Comments enhance the readability of the code and help the programmers to understand the code very carefully. There are three types of comments in Python –

* Single **line Comments**
* **Multiline Comments**
* Docstring Comments

### Example:

# sample comment

name = "geeksforgeeks"

print(name)

**Comments are generally used for the following purposes:**

* Code Readability
* Explanation of the code or Metadata of the project
* Prevent execution of code
* To include resources

## Types of Comments in Python

There are three main kinds of comments in Python. They are:

## 1)Single-Line****Comments****

Python single-line comment starts with the hashtag symbol (#) with no white spaces and lasts till the end of the line. If the comment exceeds one line then put a hashtag on the next line and continue the comment. Python’s single-line comments are proved useful for supplying short explanations for variables, function declarations, and expressions. See the following code snippet demonstrating single line comment:

**Example:**

# Print “GeeksforGeeks !” to console

print("GeeksforGeeks")

## ****2)Multi-Line Comments****

Python does not provide the option for [multiline comments](https://www.geeksforgeeks.org/multiline-comments-in-python/). However, there are different ways through which we can write multiline comments.

### Using Multiple Hashtags (#)

We can multiple hashtags (#) to write multiline comments in Python. Each and every line will be considered as a single-line comment.

### ****Example: Multiline comments using multiple****hashtags****(#)****

# Python program to demonstrate

# multiline comments

print("Multiline comments")

### Using String Literals

Python ignores the string literals that are not assigned to a variable so we can use these string literals as a comment**.**

### ****Example****

'This will be ignored by Python'

On executing the above code we can see that there will not be any output so we use the strings with triple quotes(“””) as multiline comments.

### Example 2: Multiline comments using string literals

""" Python program to demonstrate

 multiline comments"""

print("Multiline comments")

## ****3) Python Docstring****

[**Python docstring**](https://www.geeksforgeeks.org/python-docstrings/) is the string literals with triple quotes that are appeared right after the function.It is used to associate documentation that has been written with Python modules, functions, classes, and methods. It is added right below the functions, modules, or classes to describe what they do. In Python, the docstring is then made available via the \_\_doc\_\_ attribute.

**Example:**

def multiply(a, b):

    """Multiplies the value of a and b"""

    return a\*b

# Print the docstring of multiply function

print(multiply.\_\_doc\_\_)

**Python Identifiers:-**

All the variables, class, object, functions, lists, dictionaries etc. in Python are together termed as Python Identifiers. Identifiers are the basis of any Python program. Almost every Python Code uses some or other identifiers.

**Rules for using Python Identifiers:**

An identifier name should not be a keyword.

An identifier name can begin with a letter or an underscore only.

An identifier name can contain both numbers and letters along with underscores (A-z, 0-9, and \_ ).

An identifier name in Python is case-sensitive i.e, sum and Sum are two different identifier.

**Python Naming Conventions**

**1. General**

Avoid using names that are too general or too wordy. Strike a good balance between the two.

Bad:data\_structure,my\_list,info\_map,dictionary\_for\_the\_purpose\_of\_storing\_data\_representing\_word\_definitions Good: user\_profile, menu\_options, word\_definitions

Don’t be a jackass and name things “O”, “l”, or “I”

When using Camel Case names, capitalize all letters of an abbreviation (e.g. HTTPServer)

**2. Packages**

Package names should be all lower case When multiple words are needed, an underscore should separate them It is usually preferable to stick to 1 word names

**3. Modules**

Module names should be all lower case When multiple words are needed, an underscore should separate them It is usually preferable to stick to 1 word names

**4. Classes**

Class names should follow the UpperCase CamelCase convention Python’s built-in classes, however are typically lowercase words Exception classes should end in “Error”

**5. Global (module-level) Variables**

Global variables should be all lowercase Words in a global variable name should be separated by an underscore

**6. Instance Variables**

Instance variable names should be all lower case Words in an instance variable name should be separated by an underscore Non-public instance variables should begin with a single underscore

If an instance name needs to be mangled, two underscores may begin its name

**7. Methods**

Method names should be all lower case Words in an method name should be separated by an underscore Non-public method should begin with a single underscore

If a method name needs to be mangled, two underscores may begin its name

**8. Method Arguments**

Instance methods should have their first argument named ‘self’. Class methods should have their first argument named ‘cls’

**9. Functions**

Function names should be all lower case Words in a function name should be separated by an underscore

**10. Constants**

Constant names must be fully capitalized Words in a constant name should be separated by an underscore

**Input-output Statements:-**

Python has two functions designed for accepting data directly from the user:

* raw\_input()
* input()

There are also very simple ways of reading a file and, for stricter control over input, reading from stdin if necessary.

**1)raw\_input()**:-

raw\_input is used to read text (strings) from the user. (ended with a newline), and simply returns the string. It can also take an argument, which is displayed as a prompt before the user enters the data

**Example:-**

**Print**  raw\_input('What is your name? ')

prints out

What is your name? <user input data here>

**Example:** in order to assign the user's name, i.e. string data, to a variable "x" you would type

x = raw\_input('What is your name?')

Once the user inputs his name, e.g. Simon, you can call it as x

**print** 'Your name is ' + x

prints out

Your name is Simon

**2) input():-**

input() uses raw\_input to read a string of data, and then attempts to evaluate it as if it were a Python program, and then returns the value that results. So entering

[1,2,3]would return a list containing those numbers, just as if it were assigned directly in the Python script. input is used to read integers.

**Example:-**

str = input("Enter your input: ");

print "Received input is : ", str

**Output:-**

We use the print() function to output data to the standard output device (screen).

We can also [output data to a file](https://www.programiz.com/python-programming/file-operation)

print('This sentence is output to the screen')

# Output: This sentence is output to the screen

a = 5

print('The value of a is', a)

In the second print() statement, we can notice that a space was added between the [string](https://www.programiz.com/python-programming/string) and the value of variable a.This is by default, but we can change it.

The actual **syntax** of the print() function is

print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Here, objects is the value(s) to be printed.

The sep separator is used between the values. It defaults into a space character.

After all values are printed, end is printed. It defaults into a new line.

The file is the object where the values are printed and its default value issys.stdout (screen)

**Example:-**

print(1,2,3,4)

print(1,2,3,4,sep='\*')

print(1,2,3,4,sep='#',end='&')

**output:-**

1 2 3 4

1\*2\*3\*4

1#2#3#4&

### Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

>>> x = 5; y = 10

>>> print('The value of x is {} and y is {}'.format(x,y))

The value of x is 5 and y is 10

Here the curly braces {} are used as placeholders. We can specify the order in which it is printed by using numbers (tuple index).

**Example:-**

print('I love {0} and {1}'.format('bread','butter'))

print('I love {1} and {0}'.format('bread','butter'))

**Output:**

I love bread and butter

I love butter and bread

I love bread and butter

I love butter and bread

I love bread and butter

I love butter and bread

I love bread and butter

I love butter and bread

**Operators:-**

Operators are the constructs which can manipulate the value of operands.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

**Types of Operator**

* Arithmetic Operators
* Comparison (Relational) Operators
* Assignment Operators
* Logical Or Boolean Operators
* Bitwise Operators
* Membership Operators
* Identity Operators
  1. **Python Arithmetic Operators**

Arithmetic operators are used to perform arithmetic operations between two operands. It includes + (addition), - (subtraction), \*(multiplication), /(divide), %(reminder), //(floor division), and exponent (\*\*) operators.

* Assume variable a holds 10 and variable b holds 20, then −

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| + Addition | Adds values on either side of the operator. | a + b = 30 |
| - Subtraction | Subtracts right hand operand from left hand operand. | a – b = -10 |
| \* Multiplication | Multiplies values on either side of the operator | a \* b = 200 |
| / Division | Divides left hand operand by right hand operand | b / a = 2 |
| % Modulus | Divides left hand operand by right hand operand and returns remainder | b % a = 0 |
| \*\* Exponent | Performs exponential (power) calculation on operators | a\*\*b =10 to the power 20 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) − | 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0 |

**Example:-**

x = 10

y = 20

print('x + y =',x+y)

print('x - y =',x-y)

print('x \* y =',x\*y)

print('x / y =',x/y)

print('x // y =',x//y)

print('x \*\* y =',x\*\*y)

**2) Python Comparison Operators**

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly. The comparison operators are described in the following table..

Assume variable a holds 10 and variable b holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | If the values of two operands are equal, then the condition becomes true. | (a == b) is not true. |
| != | If values of two operands are not equal, then condition becomes true. | (a != b) is true. |
| <> | If values of two operands are not equal, then condition becomes true. | (a <> b) is true. This is similar to != operator. |
| > | If the value of left operand is greater than the value of right operand, then condition becomes true. | (a > b) is not true. |
| < | If the value of left operand is less than the value of right operand, then condition becomes true. | (a < b) is true. |
| >= | If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. | (a >= b) is not true. |
| <= | If the value of left operand is less than or equal to the value of right operand, then condition becomes true. | (a <= b) is true. |

**Example:-**

x = 10

y = 20

print('x > y is',x>y)

print('x < y is',x<y)

print('x == y is',x==y)

print('x != y is',x!=y)

print('x >= y is',x>=y)

print('x <= y is',x<=y)

**3) Python Assignment Operators**

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

* Assume variable a holds 10 and variable b holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Assigns values from right side operands to left side operand | c = a + b assigns value of a + b into c |
| += Add AND | It adds right operand to the left operand and assign the result to left operand | c += a is equivalent to c = c + a |
| -= Subtract AND | It subtracts right operand from the left operand and assign the result to left operand | c -= a is equivalent to c = c - a |
| \*= Multiply AND | It multiplies right operand with the left operand and assign the result to left operand | c \*= a is equivalent to c = c \* a |
| /= Divide AND | It divides left operand with the right operand and assign the result to left operand | c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a |
| %= Modulus AND | It takes modulus using two operands and assign the result to left operand | c %= a is equivalent to c = c % a |
| \*\*= Exponent AND | Performs exponential (power) calculation on operators and assign value to the left operand | c \*\*= a is equivalent to c = c \*\* a |
| //= Floor Division | It performs floor division on operators and assign value to the left operand | c //= a is equivalent to c = c // a |

**Example:-**

a = 10

b = 20

c = 0

c = a + b

print "Line 1 - Value of c is ", c

c += a

print "Line 2 - Value of c is ", c

c \*= a

print "Line 3 - Value of c is ", c

c /= a

print "Line 4 - Value of c is ", c

c = 2

c %= a

print "Line 5 - Value of c is ", c

c \*\*= a

print "Line 6 - Value of c is ", c

c //= a

print "Line 7 - Value of c is ", c

**4)Python Bitwise Operators**

The bitwise operators perform bit by bit operation on the values of the two operands. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

There are following Bitwise operators supported by Python language

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & Binary AND | Operator copies a bit to the result if it exists in both operands | (a & b) (means 0000 1100) |
| | Binary OR | It copies a bit if it exists in either operand. | (a | b) = 61 (means 0011 1101) |
| ^ Binary XOR | It copies the bit if it is set in one operand but not both. | (a ^ b) = 49 (means 0011 0001) |
| ~ Binary Ones Complement | It is unary and has the effect of 'flipping' bits. | (~a ) = -61 (means 1100 0011 in 2's complement form due to a signed binary number. |
| << Binary Left Shift | The left operands value is moved left by the number of bits specified by the right operand. | a << 2 = 240 (means 1111 0000) |
| >> Binary Right Shift | The left operands value is moved right by the number of bits specified by the right operand. | a >> 2 = 15 (means 0000 1111) |

**Example:-**

a = 60 # 60 = 0011 1100

b = 13 # 13 = 0000 1101

c = 0

c = a & b; # 12 = 0000 1100

print "Line 1 - Value of c is ", c

c = a | b; # 61 = 0011 1101

print "Line 2 - Value of c is ", c

c = a ^ b; # 49 = 0011 0001

print "Line 3 - Value of c is ", c

c = ~a; # -61 = 1100 0011

print "Line 4 - Value of c is ", c

c = a << 2; # 240 = 1111 0000

print "Line 5 - Value of c is ", c

c = a >> 2; # 15 = 0000 1111

print "Line 6 - Value of c is ", c

**5) Python Logical Operators**

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| and | If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true. |
| or | If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true. |
| not | If an expression **a** is true, then not (a) will be false and vice versa. |

**Example:-**

x = True

y = False

print('x and y is',x and y)

print('x or y is',x or y)

print('not x is',not x)

**6)Python Membership Operators**

Python membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| in | Evaluates to true if it finds a variable in the specified sequence and false otherwise. | x in y, here in results in a 1 if x is a member of sequence y. |
| not in | Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. | x not in y, here not in results in a 1 if x is not a member of sequence y. |

**Example:-**

x = 'Hello world'

y = {1:'a',2:'b'}

print('H' in x)

print('hello' not in x)

print(1 in y)

print('a' in y)

**7)Python Identity Operators**

The identity operators are used to decide whether an element certain class or type.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| is | Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. | x is y, here **is** results in 1 if id(x) equals id(y). |
| is not | Evaluates to false if the variables on either side of the operator point to the same object and true otherwise. | x is not y, here **is not** results in 1 if id(x) is not equal to id(y). |

**Example:-**

x1 = 5

y1 = 5

x2 = 'Hello'

y2 = 'Hello'

x3 = [1,2,3]

y3 = [1,2,3]

print(x1 is not y1)

print(x2 is y2)

print(x3 is y3)

**Precedence Operator:-**

Python has well-defined rules for specifying the order in which the operators in an expression are evaluated when the expression has several operators. For example, multiplication and division have a higher precedence than addition and subtraction. Precedence rules can be overridden by explicit parentheses.

**The following table lists all operators from highest precedence to lowest.**

|  |  |
| --- | --- |
| Operator | Description |
| \*\* | Exponentiation (raise to the power) |
| ~ + - | Complement, unary plus and minus (method names for the last two are +@ and -@) |
| \* / % // | Multiply, divide, modulo and floor division |
| + - | Addition and subtraction |
| >> << | Right and left bitwise shift |
| & | Bitwise 'AND'td> |
| ^ | | Bitwise exclusive `OR' and regular `OR' |
| <= < > >= | Comparison operators |
| <> == != | Equality operators |
| = %= /= //= -= += \*= \*\*= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

Operator precedence affects how an expression is evaluated.

**For example**, x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has higher precedence than +, so it first multiplies 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

**Example**

a = 20

b = 10

c = 15

d = 5

e = 0

e = (a + b) \* c / d #( 30 \* 15 ) / 5

print "Value of (a + b) \* c / d is ", e

e = ((a + b) \* c) / d # (30 \* 15 ) / 5

print "Value of ((a + b) \* c) / d is ", e

e = (a + b) \* (c / d); # (30) \* (15/5)

print "Value of (a + b) \* (c / d) is ", e

e = a + (b \* c) / d; # 20 + (150/5)

print "Value of a + (b \* c) / d is ", e

**Output:-**

Value of (a + b) \* c / d is 90

Value of ((a + b) \* c) / d is 90

Value of (a + b) \* (c / d) is 90

Value of a + (b \* c) / d is 50

**Python Operator Associativity**

In the above table, you can confirm that some of the groups have many operators. It means that all operators in a group are at the same precedence level.

And whenever two or more operators have the same precedence, then associativity defines the order of operations.

Hence, associativity is the order in which Python evaluates an expression containing multiple operators of the same precedence. Almost all operators except the exponent (\*\*) support the left-to-right associativity.

**For example**

the product (\*) and the modulus (%) have the same precedence. So, if both appear in an expression, then the left one will get evaluated first.

# Testing Left-right associativity

# Result: 1

print(4 \* 7 % 3)

# Testing left-right associativity

# Result: 0

print(2 \* (10 % 5))

As said earlier, the only operator which has right-to-left associativity in Python is the exponent (\*\*) operator.

**Example:-**

# Checking right-left associativity of \*\* exponent operator

# Output: 256

print(4 \*\* 2 \*\* 2)

# Checking the right-left associativity

# of \*\*

# Output: 256

print((4 \*\* 2) \*\* 2)

You might have observed that the ‘print(4 \*\* 2 \*\* 2)’ is similar to ‘(4 \*\* 2 \*\* 2).

**Command Line Arguments**:-

The arguments that are given after the name of the program in the command line shell of the operating system are known as Command Line Arguments. Python provides various ways of dealing with these types of arguments. The three most common are:

Using sys.argv

Using getopt module

Using argparse module

## Using sys.argv

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.  
One such variable is sys.argv which is a simple list structure. It’s main purpose are:

* It is a list of command line arguments.
* len(sys.argv) provides the number of command line arguments.
* sys.argv[0] is the name of the current Python script.

**Example:-**

import sys

n = len(sys.argv)

print("Total arguments passed:", n)

print("\nName of Python script:", sys.argv[0])

print("\nArguments passed:", end = " ")

for i in range(1, n):

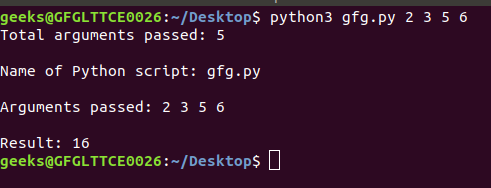
    print(sys.argv[i], end = " ")

Sum = 0

for i in range(1, n):

    Sum += int(sys.argv[i])

print("\n\nResult:", Sum)



**Using getopt module**

Python getopt module is similar to the getopt() function of C. Unlike sys module getopt module extends the separation of the input string by parameter validation. It allows both short, and long options including a value assignment. However, this module requires the use of the sys module to process input data properly. To use getopt module, it is required to remove the first element from the list of command-line arguments.

**Syntax:** getopt.getopt(args, options, [long\_options])

**Parameters:**

args: List of arguments to be passed.

options: String of option letters that the script want to recognize. Options that require an argument should be followed by a colon (:).

long\_options: List of string with the name of long options. Options that require arguments should be followed by an equal sign (=).

Return Type: Returns value consisting of two elements: the first is a list of (option, value) pairs. The second is the list of program arguments left after the option list was stripped.

**Example:-**

import getopt, sys

argumentList = sys.argv[1:]

options = "hmo:"

long\_options = ["Help", "My\_file", "Output="]

try:

    arguments, values = getopt.getopt(argumentList, options, long\_options)

    for currentArgument, currentValue in arguments:

        if currentArgument in ("-h", "--Help"):

            print ("Displaying Help")

        elif currentArgument in ("-m", "--My\_file"):

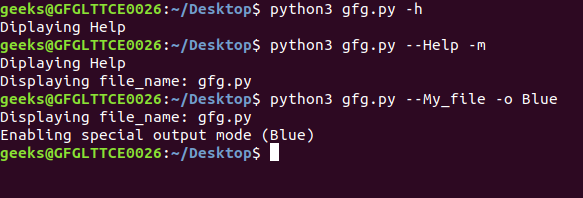
            print ("Displaying file\_name:", sys.argv[0])

        elif currentArgument in ("-o", "--Output"):

            print (("Enabling special output mode (% s)") % (currentValue))

except getopt.error as err:

    print (str(err))



**Using argparse module**

Using argparse module is a better option than the above two options as it provides a lot of options such as positional arguments, default value for arguments, help message, specifying data type of argument etc.

Note: As a default optional argument, it includes -h, along with its long version –help.

**Example:-**

import argparse

parser = argparse.ArgumentParser()

parser.parse\_args()

