Overview

Features, History and Version of Python, Applications of Python, Installation Steps, Input/output operations, Data Types, Variables, Keywords, Identifiers, Literals, Operators, Single line comments, Multiline comments, Type Conversions

Features:

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

History and Version of Python:

the important stages in the history of Python −

Python 0.9.0

Python's first published version is 0.9. It was released in February 1991. It consisted of support for core object-oriented programming principles.

Python 1.0

In January 1994, version 1.0 was released, armed with functional programming tools, features like support for complex numbers etc.

Python 2.0

Next major version − Python 2.0 was launched in October 2000. Many new features such as list comprehension, garbage collection and Unicode support were included with it.

Python 3.0

Python 3.0, a completely revamped version of Python was released in December 2008. The primary objective of this revamp was to remove a lot of discrepancies that had crept in Python 2.x versions. Python 3 was backported to Python 2.6. It also included a utility named as **python2to3** to facilitate automatic translation of Python 2 code to Python 3.

EOL for Python 2.x

Even after the release of Python 3, Python Software Foundation continued to support the Python 2 branch with incremental micro versions till 2019. However, it decided to discontinue the support by the end of year 2020, at which time Python 2.7.17 was the last version in the branch.

Current Version

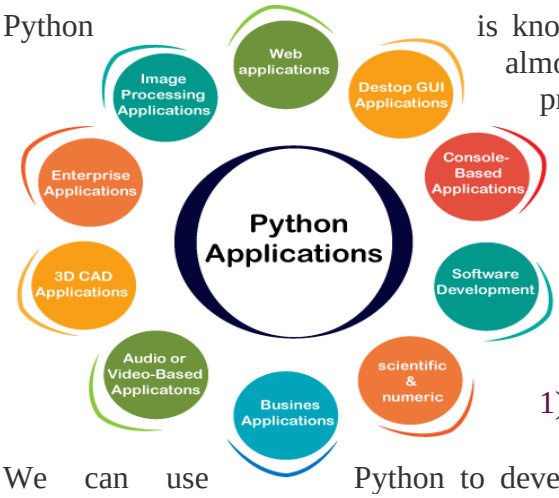
Meanwhile, more and more features have been incorporated into Python's 3.x branch. As of date, Python **3.11.2** is the current stable version, released in February 2023.

What's New in Python 3.11?

One of the most important features of Python's version 3.11 is the significant improvement in speed. According to Python's official documentation, this version is faster than the previous version (3.10) by up to 60%. It also states that the standard benchmark suite shows a 25% faster execution rate.

- Python 3.11 has a better exception messaging. Instead of generating a long traceback on the occurrence of an exception, we now get the exact expression causing the error.
- As per the recommendations of PEP 678, the **add_note()** method is added to the BaseException class. You can call this method inside the except clause and pass a custom error message.
- It also adds the **cbroot()** function in the **maths** module. It returns the cube root of a given number.
- A new module **tomllib** is added in the standard library. TOML (Tom's Obvious Minimal Language) can be parsed with tomlib module function.

Python Applications



Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

Here, we are specifying application areas where Python can be applied.

1) Web Applications

We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser, etc. One of Python web-framework named Django is used on **Instagram**. Python provides many useful frameworks, and these are given below:

Django and Pyramid framework(Use for heavy applications)

- Flask and Bottle (Micro-framework)
- Plone and Django CMS (Advance Content management)

2) Desktop GUI Applications

The GUI stands for the Graphical User Interface, which provides a smooth interaction to any application. Python provides a **Tk GUI library** to develop a user interface. Some popular GUI libraries are given below.

- Tkinter or Tk
- wxWidgetM
- Kivy (used for writing multitouch applications)
- PyQt or Pyside

3) Console-based Application

Console-based applications run from the command-line or shell. These applications are computer program which are used commands to execute. This kind of application was more popular in the old generation of computers. Python can develop this kind of application very effectively. It is famous for having REPL, which means **the Read-Eval-Print Loop** that makes it the most suitable language for the command-line applications.

Python provides many free library or module which helps to build the command-line apps. The necessary **IO** libraries are used to read and write. It helps to parse argument and create console help text out-of-the-box. There are also advance libraries that can develop independent console apps.

4) Software Development

Python is useful for the software development process. It works as a support language and can be used to build control and management, testing, etc.

- **SCons** is used to build control.
- **Buildbot** and **Apache** Gumps are used for automated continuous compilation and testing.
- **Round** or **Trac** for bug tracking and project management.

5) Scientific and Numeric

This is the era of Artificial intelligence where the machine can perform the task the same as the human. Python language is the most suitable language for Artificial intelligence or machine learning. It consists of many scientific and mathematical libraries, which makes easy to solve complex calculations.

Implementing machine learning algorithms require complex mathematical calculation. Python has many libraries for scientific and numeric such as Numpy, Pandas, Scipy, Scikit-learn, etc. If you have some basic knowledge of Python, you need to import libraries on the top of the code. Few popular frameworks of machine libraries are given below.

- SciPy
- Scikit-learn
- NumPy
- Pandas
- Matplotlib

6) Business Applications

Business Applications differ from standard applications. E-commerce and ERP are an example of a business application. This kind of application requires extensively, scalability and readability, and Python provides all these features.

Oddo is an example of the all-in-one Python-based application which offers a range of business applications. Python provides a **Tryton** platform which is used to develop the business application.

7) Audio or Video-based Applications

Python is flexible to perform multiple tasks and can be used to create multimedia applications. Some multimedia applications which are made by using Python are **TimPlayer, cplay,** etc. The few multimedia libraries are given below.

- Gstreamer
- Pyglet
- QT Phonon

8) 3D CAD Applications

The CAD (Computer-aided design) is used to design engineering related architecture. It is used to develop the 3D representation of a part of a system. Python can create a 3D CAD application by using the following functionalities.

- Fandango (Popular)
- CAMVOX
- HeeksCNC
- AnyCAD
- RCAM

9) Enterprise Applications

Python can be used to create applications that can be used within an Enterprise or an Organization. Some real-time applications are OpenERP, Tryton, Picalo, etc.

10) Image Processing Application

Python contains many libraries that are used to work with the image. The image can be manipulated according to our requirements. Some libraries of image processing are given below.

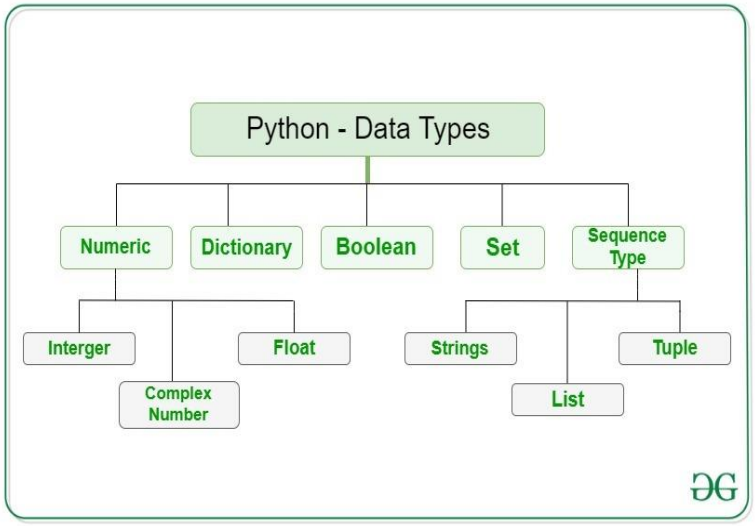
- OpenCV
- Pillow
- SimpleITK

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. The following are the standard or built-in data types in Python:

- **Numeric**
- **Sequence Type**
- **Boolean**
- **Set**
- **Dictionary**
- **Binary**

Types(memoryview, bytearray, bytes)



What is Python type() Function?

To define the values of various data types and check their data types we **use the type() function.** Consider the following examples.

This code assigns variable **'x'** different values of various data types in Python. It covers **string**, **integer**, **float**, **complex**, **list**, **tuple**, **range**, **dictionary**, **set**, **frozenset**, **boolean**, **bytes**, **byte array**, **memoryview**, and the special value **'None'** successively. Each assignment replaces the previous value, making **'x'** take on the data type and value of the most recent assignment.

- Python

```
x = "Hello World"
x = 50
x = 60.5
x = 3j
x = ["geeks", "for", "geeks"]
x = ("geeks", "for", "geeks")
x = range(10)
x = {"name": "Suraj", "age": 24}
x = {"geeks", "for", "geeks"}
x = frozenset({"geeks", "for", "geeks"})
x = True
x = b"Geeks"
x = bytearray(4)
x = memoryview(bytes(6))
x = None
```

Numeric Data Types in Python

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number. These values are defined as Python int, Python float, and Python complex classes in Python.

- **Integers** – This value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.
- **Float** – This value is represented by the float class. It is a real number with a 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** – A complex number is represented by a complex class. It is specified as *(real part) + (imaginary part)j*. For example – 2+3j

Note – type() function is used to determine the type of data type.

Example: This code demonstrates how to determine the data type of variables in Python using the **type() function**. It prints the data types of three variables**: a (integer)**, **b (float)**, and **c (complex)**. The output shows the respective data types for each variable.

- Python

```
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))
```

Output:

Type of a: <class 'int'>
Type of b: <class 'float'>
Type of c: <class 'complex'>

Sequence Data Type in Python

The sequence Data Type in Python is the ordered collection of similar or different data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence types in Python –

- Python String
- Python List
- Python Tuple

String Data Type

Strings in Python are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote, or triple-quote. In Python there is no character data type, a character is a string of length one. It is represented by str class.

Creating String

Strings in Python can be created using single quotes, double quotes, or even triple quotes.

Example: This Python code showcases various string creation methods. It uses single quotes, double quotes, and triple quotes to create strings with different content and includes a multiline string. The code also demonstrates printing the strings and checking their data types.

- Python

```
String1 = 'Welcome to the Geeks World'
print("String with the use of Single Quotes: ")
print(String1)
String1 = "I'm a Geek"
print("\nString with the use of Double Quotes: ")
print(String1)
print(type(String1))
String1 = '''I'm a Geek and I live in a world of "Geeks"'''
print("\nString with the use of Triple Quotes: ")
print(String1)
print(type(String1))

String1 = '''Geeks
        For
        Life'''
print("\nCreating a multiline String: ")
print(String1)
```

Output:

```
String with the use of Single Quotes:
Welcome to the Geeks World
String with the use of Double Quotes:
I'm a Geek
<class 'str'>
String with the use of Triple Quotes:
I'm a Geek and I live in a world of "Geeks"
<class 'str'>
Creating a multiline String:
Geeks
        For
        Life
```

Accessing elements of String

In Python programming, individual characters of a String can be accessed by using the method of Indexing. Negative Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

Example: This Python code demonstrates how to work with a string named '**String1′**. It initializes the string with **"GeeksForGeeks"** and prints it. It then showcases how to access the first character (**"G"**) using an index of 0 and the last character (**"s"**) using a negative index of -1.

- Python

```
String1 = "GeeksForGeeks"
print("Initial String: ")
print(String1)
print("\nFirst character of String is: ")
print(String1[0])
print("\nLast character of String is: ")
print(String1[-1])
```

Output:
Initial String:
GeeksForGeeks
First character of String is:
G
Last character of String is:
s

Note – To know more about strings, refer to Python String.

List Data Type

Lists are just like arrays, declared in other languages which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type.

Creating a List in Python

Lists in Python can be created by just placing the sequence inside the square brackets[].

Example: This Python code demonstrates list creation and manipulation. It starts with an empty list and prints it. It creates a list containing a single string element and prints it. It creates a list with multiple string elements and prints selected elements from the list. It creates a multi-dimensional list (a list of lists) and prints it. The code showcases various ways to work with lists, including single and multi-dimensional lists.

- Python

```
List = []
print("Initial blank List: ")
print(List)
List = ['GeeksForGeeks']
print("\nList with the use of String: ")
print(List)
List = ["Geeks", "For", "Geeks"]
print("\nList containing multiple values: ")
print(List[0])
print(List[2])
List = [['Geeks', 'For'], ['Geeks']]
print("\nMulti-Dimensional List: ")
```

```
print(List)
```

Output:
Initial blank List:
[]
List with the use of String:
['GeeksForGeeks']
List containing multiple values:
Geeks
Geeks
Multi-Dimensional List:
[['Geeks', 'For'], ['Geeks']]

Python Access List Items
In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

- Python

```
List = ["Geeks", "For", "Geeks"]
print("Accessing element from the list")
print(List[0])
print(List[2])
print("Accessing element using negative indexing")
print(List[-1])
print(List[-3])
```

Output:
Accessing element from the list
Geeks
Geeks
Accessing element using negative indexing
Geeks
Geeks

Note – To know more about Lists, refer to Python List.
Tuple Data Type
Just like a list, a tuple is also an ordered collection of Python objects. The only difference between a tuple and a list is that tuples are immutable i.e. tuples cannot be modified after it is created. It is represented by a tuple class.
Creating a Tuple in Python
In Python, tuples are created by placing a sequence of values separated by a 'comma' with or without the use of parentheses for grouping the data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, lists, etc.). **Note:** Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing **'comma'** to make it a tuple.

Example: This Python code demonstrates different methods of creating and working with tuples. It starts with an empty tuple and prints it. It creates a tuple containing string elements and prints it. It converts a list into a tuple and prints the result. It creates a tuple from a string using the **tuple()** function. It forms a tuple with nested tuples and displays the result.

- Python

```
Tuple1 = ()
print("Initial empty Tuple: ")
print(Tuple1)
Tuple1 = ('Geeks', 'For')
print("\nTuple with the use of String: ")
print(Tuple1)
list1 = [1, 2, 4, 5, 6]
print("\nTuple using List: ")
print(tuple(list1))
Tuple1 = tuple('Geeks')
print("\nTuple with the use of function: ")
print(Tuple1)
Tuple1 = (0, 1, 2, 3)
Tuple2 = ('python', 'geek')
Tuple3 = (Tuple1, Tuple2)
print("\nTuple with nested tuples: ")
print(Tuple3)
```

Output:

```
Initial empty Tuple:
()
Tuple with the use of String:
('Geeks', 'For')
Tuple using List:
(1, 2, 4, 5, 6)
Tuple with the use of function:
('G', 'e', 'e', 'k', 's')
Tuple with nested tuples:
((0, 1, 2, 3), ('python', 'geek'))
```

Note – The creation of a Python tuple without the use of parentheses is known as Tuple Packing.

Access Tuple Items

In order to access the tuple items refer to the index number. Use the index operator [ ] to access an item in a tuple. The index must be an integer. Nested tuples are accessed using nested indexing.

The code creates a tuple named '**tuple1**' with five elements: **1, 2, 3, 4, and 5**. Then it prints the first, last, and third last elements of the tuple using indexing.

- Python

```
tuple1 = tuple([1, 2, 3, 4, 5])
print("First element of tuple")
print(tuple1[0])
print("\nLast element of tuple")
print(tuple1[-1])


print("\nThird last element of tuple")
print(tuple1[-3])
```

Output:
First element of tuple
1
Last element of tuple
5
Third last element of tuple
3

Note – To know more about tuples, refer to Python Tuples.

Boolean Data Type in Python

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). However non-Boolean objects can be evaluated in a 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: The first two lines will print the type of the boolean values True and False, which is **<class 'bool'>.** The third line will cause an error, because true is not a valid keyword in Python. Python is case-sensitive, which means it distinguishes between uppercase and lowercase letters. You need to capitalize the first letter of true to make it a boolean value.

- Python

```
print(type(True))
print(type(False))


print(type(true))
```

Output:
<class 'bool'>
<class 'bool'>

Traceback (most recent call last):
  File "/home/7e8862763fb66153d70824099d4f5fb7.py", line 8, in
    print(type(true))

NameError: name 'true' is not defined

Set Data Type in Python

In Python, a Set is an unordered collection of data types 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.

Create a Set in Python

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 a **'comma'.** The 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: The code is an example of how to create sets using different types of values, such as **strings**, **lists**, and mixed values

- Python

```
set1 = set()
print("Initial 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)
```

Output:

Initial blank Set:
set()
Set with the use of String:
{'F', 'o', 'G', 's', 'r', 'k', 'e'}
Set with the use of List:
{'Geeks', 'For'}
Set with the use of Mixed Values
{1, 2, 4, 6, 'Geeks', 'For'}

Access Set Items

Set items cannot be accessed by referring to an index, since sets are unordered the items have 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 the keyword.

Example: This Python code creates a set named **set1** with the values **"Geeks"**, **"For"** and **"Geeks"**. The code then prints the initial set, the elements of the set in a loop, and checks if the value **"Geeks"** is in the set using the '**in**' operator

- Python

```
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)
```

Output:
Initial set:
{'Geeks', 'For'}
Elements of set:
Geeks For
True

Note – To know more about sets, refer to Python Sets.

Dictionary Data Type in Python

A dictionary in Python is an unordered collection of data values, used to store data values like a map, unlike other Data Types that hold only a single value as an element, a Dictionary holds a key: value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon : , whereas each key is separated by a 'comma'.

Create a Dictionary in Python

In Python, a Dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable. The dictionary can also be created by the built-in function **dict().** An empty dictionary can be created by just placing it in curly braces{}. **Note** – Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.

Example: This code creates and prints a variety of dictionaries. The first dictionary is empty. The second dictionary has integer keys and string values. The third dictionary has mixed keys, with one string key and one integer key. The fourth dictionary is created using the **dict()** function, and the fifth dictionary is created using the **[(key, value)]** syntax

- Python

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

```
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 Integer Keys:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
Dictionary with the use of Mixed Keys:
{1: [1, 2, 3, 4], 'Name': 'Geeks'}
Dictionary with the use of dict():
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
Dictionary with each item as a pair:
{1: 'Geeks', 2: 'For'}
```

Accessing Key-value in Dictionary

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets. There is also a method called **get()** that will also help in accessing the element from a dictionary.

Example: The code in Python is used to access elements in a dictionary. Here's what it does, It creates a dictionary Dict with keys and values as {**1: 'Geeks', 'name': 'For', 3: 'Geeks'}**. It prints the value of the element with the key **'name'**, which is **'For'**. It prints the value of the element with the key 3, which is **'Geeks'**.

- Python

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

Output:

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

Python Data Type Exercise Questions

Below are two exercise questions on Python Data Types. We have covered list operation and tuple operation in these exercise questions. For more exercises on Python data types visit the page mentioned below.

Q1. Code to implement basic list operations

- Python

```
fruits = ["apple", "banana", "orange"]
print(fruits)
fruits.append("grape")
print(fruits)
fruits.remove("orange")
print(fruits)
```

Output
['apple', 'banana', 'orange']
['apple', 'banana', 'orange', 'grape']
['apple', 'banana', 'grape']

Q2. Code to implement basic tuple operation

- Python

```
coordinates = (3, 5)
print(coordinates)
print("X-coordinate:", coordinates[0])
print("Y-coordinate:", coordinates[1])
```

Output
(3, 5)
X-coordinate: 3
Y-coordinate: 5

Python Variables

A variable is the name given to a memory location. A value-holding Python variable is also known as an identifier.

Since Python is an infer language that is smart enough to determine the type of a variable, we do not need to specify its type in Python.

Variable names must begin with a letter or an underscore, but they can be a group of both letters and digits.

The name of the variable should be written in lowercase. Both Rahul and rahul are distinct variables.

Identifier Naming

Identifiers are things like variables. An Identifier is utilized to recognize the literals utilized in the program. The standards to name an identifier are given underneath.

- The variable's first character must be an underscore or alphabet (_).

- Every one of the characters with the exception of the main person might be a letter set of lower-case(a-z), capitalized (A-Z), highlight, or digit (0-9).
- White space and special characters (!, @, #, %, etc.) are not allowed in the identifier name. ^, &, *).
- Identifier name should not be like any watchword characterized in the language.
- Names of identifiers are case-sensitive; for instance, my name, and MyName isn't something very similar.
- Examples of valid identifiers: a123, _n, n_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.

Declaring Variable and Assigning Values

- Python doesn't tie us to pronounce a variable prior to involving it in the application. It permits us to make a variable at the necessary time.
- In Python, we don't have to explicitly declare variables. The variable is declared automatically whenever a value is added to it.
- The equal (=) operator is utilized to assign worth to a variable.

Object References

When we declare a variable, it is necessary to comprehend how the Python interpreter works. Compared to a lot of other programming languages, the procedure for dealing with variables is a little different.

Python is the exceptionally object-arranged programming language; Because of this, every data item is a part of a particular class. Think about the accompanying model.

```
1. print("John")
```

Output:

```
John
```

The Python object makes a integer object and shows it to the control center. We have created a string object in the print statement above. Make use of the built-in type() function in Python to determine its type.

```
1. type("John")
```

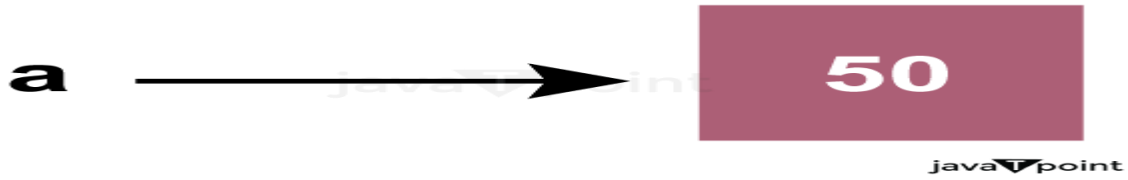
Output:

```
<class 'str'>
```

In Python, factors are an symbolic name that is a reference or pointer to an item. The factors are utilized to indicate objects by that name.

Let's understand the following example

```
1. a = 50
```



In the above image, the variable **a** refers to an integer object.

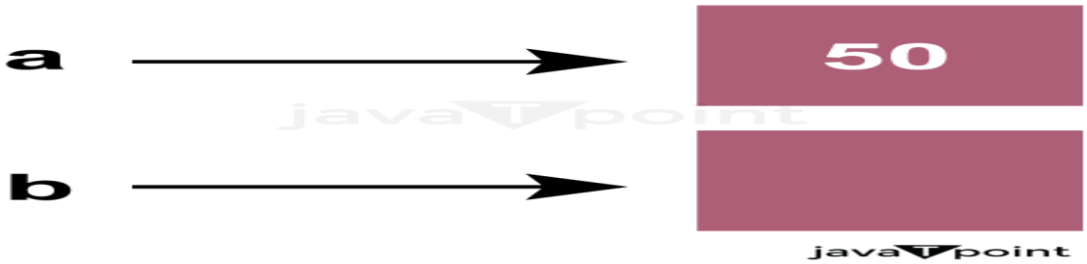
Suppose we assign the integer value 50 to a new variable b.

1. a = 50
2. b = a

The variable b refers to the same object that a points to because Python does not create another object.

Let's assign the new value to b. Now both variables will refer to the different objects.

1. a = 50
2. b =100



Python manages memory efficiently if we assign the same variable to two different values.

Object Identity

Every object created in Python has a unique identifier. Python gives the dependable that no two items will have a similar identifier. The object identifier is identified using the built-in id() function. consider about the accompanying model.

```
1. a = 50
2. b = a
3. print(id(a))
4. print(id(b))
5. # Reassigned variable a
6. a = 500
7. print(id(a))
```

Output:

```
140734982691168
140734982691168
2822056960944
```

We assigned the b = a, an and b both highlight a similar item. The id() function that we used to check returned the same number. We reassign a to 500; The new object identifier was then mentioned.

Variable Names

The process for declaring the valid variable has already been discussed. Variable names can be any length can have capitalized, lowercase (start to finish, a to z), the digit (0-9), and highlight character(_). Take a look at the names of valid variables in the following example.

```
1. name = "Devansh"
2. age = 20
3. marks = 80.50
4.
5. print(name)
6. print(age)
7. print(marks)
```

Output:

```
Devansh
20
80.5
```

Consider the following valid variables name.

```
1. name = "A"
2. Name = "B"
3. naMe = "C"
4. NAME = "D"
5. n_a_m_e = "E"
6. _name = "F"
7. name_ = "G"
8. _name_ = "H"
9. na56me = "I"
10.
11. print(name,Name,naMe,NAME,n_a_m_e, NAME, n_a_m_e, _name, name_,_name, na56me)
```

Output:

```
A B C D E D E F G F I
```

We have declared a few valid variable names in the preceding example, such as name, _name_, and so on. However, this is not recommended because it may cause confusion when we attempt to read code. To make the code easier to read, the name of the variable ought to be descriptive.

The multi-word keywords can be created by the following method.

- **Camel Case -** In the camel case, each word or abbreviation in the middle of begins with a capital letter. There is no intervention of whitespace. For example - nameOfStudent, valueOfVaraible, etc.
- **Pascal Case -** It is the same as the Camel Case, but here the first word is also capital. For example - NameOfStudent, etc.
- **Snake Case -** In the snake case, Words are separated by the underscore. For example - name_of_student, etc.

Multiple Assignment

Multiple assignments, also known as assigning values to multiple variables in a single statement, is a feature of Python.

We can apply different tasks in two ways, either by relegating a solitary worth to various factors or doling out numerous qualities to different factors. Take a look at the following example.

1. Assigning single value to multiple variables

Eg:

```
x=y=z=50
print(x)
print(y)
print(z)
```

Output:

```
50
50
50
```

2. Assigning multiple values to multiple variables:

Eg:

```
a,b,c=5,10,15
print a
print b
print c
```

Output:

```
5
10
```

```
15
```

The values will be assigned in the order in which variables appear.

Python Variable Types

There are two types of variables in Python - Local variable and Global variable. Let's understand the following variables.

Local Variable

The variables that are declared within the function and have scope within the function are known as local variables. Let's examine the following illustration.

Example -

```
1. # Declaring a function
2. def add():
3.     # Defining local variables. They has scope only within a function
4.     a = 20
5.     b = 30
6.     c = a + b
7.     print("The sum is:", c)
8.
9. # Calling a function
10. add()
```

Output:

```
The sum is: 50
```

Explanation:

We declared the function add() and assigned a few variables to it in the code above. These factors will be alluded to as the neighborhood factors which have scope just inside the capability. We get the error that follows if we attempt to use them outside of the function.

```
1. add()
2. # Accessing local variable outside the function
3. print(a)
```

Output:

```
The sum is: 50
   print(a)
NameError: name 'a' is not defined
```

We tried to use local variable outside their scope; it threw the **NameError.**

Global Variables

Global variables can be utilized all through the program, and its extension is in the whole program. Global variables can be used inside or outside the function.

By default, a variable declared outside of the function serves as the global variable. Python gives the worldwide catchphrase to utilize worldwide variable inside the capability. The function treats it as a local variable if we don't use the global keyword. Let's examine the following illustration.

Example -

```
# Declare a variable and initialize it
x = 101

# Global variable in function
def mainFunction():
    # printing a global variable
    global x
    print(x)
    # modifying a global variable
    x = 'Welcome To Javatpoint'
    print(x)

mainFunction()
print(x)
```

Output:

```
101
Welcome To Javatpoint
Welcome To Javatpoint
```

Explanation:

In the above code, we declare a global variable x and give out a value to it. We then created a function and used the global keyword to access the declared variable within the function. We can now alter its value. After that, we gave the variable x a new string value and then called the function and printed x, which displayed the new value.

=

Delete a variable

We can delete the variable using the **del** keyword. The syntax is given below.

Syntax -

```
del <variable_name>
```

In the following example, we create a variable x and assign value to it. We deleted variable x, and print it, we get the error **"variable x is not defined"**. The variable x will no longer use in future.

Example -

```
# Assigning a value to x
x = 6
print(x)
# deleting a variable.
del x
print(x)
```

Output:

```
6
Traceback (most recent call last):
  File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/multiprocessing.py", line 389, in
    print(x)
NameError: name 'x' is not defined
```

Maximum Possible Value of an Integer in Python

Python, to the other programming languages, does not support long int or float data types. It uses the int data type to handle all integer values. The query arises here. In Python, what is the maximum value that the variable can hold? Take a look at the following example.

Example -

```
# A Python program to display that we can store
# large numbers in Python

a = 1000000000000000000000000000000000000000000000000
a = a + 1
print(type(a))
print (a)
```

Output:

```
<class 'int'>
10000000000000000000000000000000000000000000001
```

As we can find in the above model, we assigned a large whole number worth to variable x and really look at its sort. It printed class <int> not long int. As a result, the number of bits is not limited, and we are free to use all of our memory.

There is no special data type for storing larger numbers in Python.

Print Single and Numerous Factors in Python

We can print numerous factors inside the single print explanation. The examples of single and multiple printing values are provided below.

Example - 1 (Printing Single Variable)

```
# printing single value
a = 5
print(a)
print((a))
```

Output:

```
5
5
```

Example - 2 (Printing Multiple Variables)

```
a = 5
b = 6
# printing multiple variables
print(a,b)
# separate the variables by the comma
Print(1, 2, 3, 4, 5, 6, 7, 8)
```

Output:

```
5 6
1 2 3 4 5 6 7 8
```

Basic Fundamentals:

This section contains the fundamentals of Python, such as:

i)Tokens and their types.

ii) Comments

a)Tokens:

- The tokens can be defined as a punctuator mark, reserved words, and each word in a statement.
- The token is the smallest unit inside the given program.

There are following tokens in Python:

- Keywords.
- Identifiers.
- Literals.
- Operators.

Keywords in Python
Python Keywords are some predefined and reserved words in Python that have special meanings. Keywords are used to define the syntax of the coding. The keyword cannot be used as an identifier,

function, or variable name. All the keywords in Python are written in lowercase except True and False. There are 35 keywords in Python 3.11.
In Python, there is an inbuilt keyword module that provides an iskeyword() function that can be used to check whether a given string is a valid keyword or not. Furthermore, we can check the name of the keywords in Python by using the kwlist attribute of the keyword module.
0 seconds of 17 secondsVolume 0%

Rules for Keywords in Python

- Python keywords cannot be used as identifiers.
- All the keywords in Python should be in lowercase except True and False.

List of Python Keywords

Keywords	Description
and	This is a logical operator which returns true if both the operands are true else returns false.
or	This is also a logical operator which returns true if anyone operand is true else returns false.
not	This is again a logical operator it returns True if the operand is false else returns false.
if	This is used to make a conditional statement.
elif	Elif is a condition statement used with an if statement. The elif statement is executed if the previous conditions were not true.
else	Else is used with if and elif conditional statements. The else block is executed if the given condition is not true.
for	This is used to create a loop.
while	This keyword is used to create a while loop.
break	This is used to terminate the loop.
as	This is used to create an alternative.
def	It helps us to define functions.
lambda	It is used to define the anonymous function.
pass	This is a null statement which means it will do nothing.
return	It will return a value and exit the function.
True	This is a boolean value.

Keywords	Description
False	This is also a boolean value.
try	It makes a try-except statement.
with	The with keyword is used to simplify exception handling.
assert	This function is used for debugging purposes. Usually used to check the correctness of code
class	It helps us to define a class.
continue	It continues to the next iteration of a loop
del	It deletes a reference to an object.
except	Used with exceptions, what to do when an exception occurs
finally	Finally is used with exceptions, a block of code that will be executed no matter if there is an exception or not.
from	It is used to import specific parts of any module.
global	This declares a global variable.
import	This is used to import a module.
in	It’s used to check whether a value is present in a list, range, tuple, etc.
is	This is used to check if the two variables are equal or not.
none	This is a special constant used to denote a null value or avoid. It’s important to remember, 0, any empty container(e.g empty list) do not compute to None
nonlocal	It’s declared a non-local variable.
raise	This raises an exception.
yield	It ends a function and returns a generator.
async	It is used to create asynchronous coroutine.

Keywords	Description
await	It releases the flow of control back to the event loop.

The following code allows you to view the complete list of Python's keywords.
This code imports the "keyword" module in Python and then prints a list of all the keywords in Python using the "kwlist" attribute of the "keyword" module. The "kwlist" attribute is a list of strings, where each string represents a keyword in Python. By printing this list, we can see all the keywords that are reserved in Python and cannot be used as identifiers.

- Python3

```
# code
import keyword

print(keyword.kwlist)
```

Output
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is',...

Identifiers in Python
Identifier is a user-defined name given to a variable, function, class, module, etc. The identifier is a combination of character digits and an underscore. They are case-sensitive i.e., 'num' and 'Num' and 'NUM' are three different identifiers in python. It is a good programming practice to give meaningful names to identifiers to make the code understandable.
We can also use the Python string isidentifier() method to check whether a string is a valid identifier or not.
Rules for Naming Python Identifiers

- It cannot be a reserved python keyword.
- It should not contain white space.
- It can be a combination of A-Z, a-z, 0-9, or underscore.
- It should start with an alphabet character or an underscore (_).
- It should not contain any special character other than an underscore (_).

Examples of Python Identifiers
Valid identifiers:

- *var1*
- *_var1*
- *_1_var*
- *var_1*

Invalid Identifiers

- *!var1*
- *1var*
- *1_var*
- *var#1*
- *var 1*

Python Keywords and Identifiers Examples
Example 1: Example of and, or, not, True, False keywords.

- Python

```
print("example of True, False, and, or, not keywords")

#  compare two operands using and operator
print(True and True)

# compare two operands using or operator
print(True or False)

# use of not operator
print(not False)
```

Output
example of True, False, and, or, not keywords

True

True

True

Example 2: Example of a break, continue keywords and identifier.

- Python

```
# execute for loop
for i in range(1, 11):

    # print the value of i
    print(i)

    # check the value of i is less than 5
    # if i lessthan 5 then continue loop
    if i < 5:
        continue

    # if i greater than 5 then break loop
    else:
        break
```

Output
1

2

3

4

5

Example 3: example of for, in, if, elif, and else keywords.

- Python

```
# run for loop
for t in range(1, 5):
  # print one of t ==1
    if t == 1:
        print('One')
   # print two if t ==2
    elif t == 2:
        print('Two')
    else:
        print('else block execute')
```

Output
One

Two

else block execute

else block execute

Example 4: Example of def, if, and else keywords.

- Python

```
# define GFG() function using def keyword
def GFG():
    i=20
    # check i is odd or not
    # using if and else keyword
    if(i % 2 == 0):
        print("given number is even")
    else:
        print("given number is odd")

# call GFG() function
GFG()
```

Output
given number is even

Example 5: Example of try, except, raise.

- Python

```
def fun(num):
```

```
    try:
        r = 1.0/num
    except:
        print('Exception raises')
        return
    return r

print(fun(10))
print(fun(0))
```

Output

0.1

Exception raises

None

Example 6: Example of a lambda keyword.

- Python

```
# define a anonymous using lambda keyword
# this lambda function increment the value of b
a = lambda b: b+1

# run a for loop
for i in range(1, 6):
    print(a(i))
```

Output

2

3

4

5

6

Example 7: use of return keyword.

- Python

```
# define a function
def fun():
  # declare a variable
    a = 5
    # return the value of a
    return a
```

```
# call fun method and store
# it's return value in a variable
t = fun()
# print the value of t
print(t)
```

Output
5

Example 8: use of a del keyword.

- Python

```
# create a list
l = ['a', 'b', 'c', 'd', 'e']

# print list before using del keyword
print(l)

del l[2]

# print list after using del keyword
print(l)
```

Output
['a', 'b', 'c', 'd', 'e']

['a', 'b', 'd', 'e']

Example 9: use of global keyword.

- Python

```
# declare a variable
gvar = 10

# create a function
def fun1():
  # print the value of gvar
    print(gvar)

# declare fun2()
def fun2():
  # declare global value gvar
    global gvar
    gvar = 100
```

```
# call fun1()
fun1()

# call fun2()
fun2()
```

Output
10

Example 10: example of yield keyword.

- Python

```
def Generator():
    for i in range(6):
        yield i+1

t = Generator()
for i in t:
    print(i)
```

Output
1

2

3

4

5

6

Example 11: Use of assert keyword.

- Python3

```
def sumOfMoney(money):
    assert len(money) != 0,"List is empty."
    return sum(money)

money = []
print("sum of money:",sumOfMoney(money))
```

Output:
AssertionError: List is empty.
Example 12: Use of pass keyword

- Python3

```
class GFG:
    pass
g = GFG
```

Output

Example 13: Use of finally keyword

- Python3

```
def divide(a, b):
    try:
        c = a/b
        print("Inside try block")
    except:
        print("Inside Exception block")
    finally:
        print("Inside finally block")
divide(3,2)
divide(3,0)
```

Output

Inside try block

Inside finally block

Inside Exception block

Inside finally block

Example 14: Use of import keyword

- Python3

```
import math
print("factorial of 5 is :", math.factorial(5))
```

Output

factorial of 5 is : 120

Example 15: Use of is keyword

- Python3

```
x = 10
y = 20
z = x
print(x is z)
```

```
print(x is y)
```

Output
True

False

Example 16: Use of from keyword

- Python3

```
from math import gcd
print("gcd of 345 and 675 is : ", gcd(345, 675))
```

Output
gcd of 345 and 675 is : 15

Example 17: Use of async and await keyword

- Python3

```
# code
import asyncio

async def factorial(n):
    if n == 0:
        return 1
    return n * await factorial(n - 1)

def main():
    result = asyncio.run(factorial(5))
    print(result)

if __name__ == "__main__":
    main()
```

Output
120

Python Literals

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

Python supports the following literals:

1. String literals:

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

Example:

```
1. "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.

Example:

```
1. text1='hello'
```

b) Multi-line String - A piece of text that is written in multiple lines is known as multiple lines string.

There are two ways to create multiline strings:

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

Example:

```
1. text1='hello\
2. user'
3. print(text1)
```

```
'hellouser'
```

2) Using triple quotation marks:-

Example:

```
1. str2="""welcome
2. to
3. SSSIT"""
4. 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 the complex number. eg: 3.14j

Example - Numeric Literals

```
x = 0b10100 #Binary Literals
y = 100 #Decimal Literal
z = 0o215 #Octal Literal
u = 0x12d #Hexadecimal Literal

#Float Literal
float_1 = 100.5
float_2 = 1.5e2

#Complex Literal
a = 5+3.14j

print(x, y, z, u)
print(float_1, float_2)
print(a, a.imag, a.real)
```

Output:

```
20 100 141 301
100.5 150.0
(5+3.14j) 3.14 5.0
```

III. Boolean literals:

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

Example - Boolean Literals

```
x = (1 == True)
y = (2 == False)
z = (3 == True)
a = True + 10
b = False + 10
```

```
6.
7. print("x is", x)
8. print("y is", y)
9. print("z is", z)
10. print("a:", a)
11. print("b:", b)
```

Output:

```
x is True
y is False
z is False
a: 11
b: 10
```

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 the end of lists in Python.

Example - Special Literals

```
1. val1=10
2. val2=None
3. print(val1)
4. print(val2)
```

Output:

```
10
None
```

V. Literal Collections.

Python provides the four types of literal collection such as List literals, Tuple literals, Dict literals, and Set literals.

List:

- List contains items of different data types. Lists are mutable i.e., modifiable.
- The values stored in List are separated by comma(,) and enclosed within square brackets([]). We can store different types of data in a List.

Example - List literals

```
1. list=['John',678,20.4,'Peter']
2. list1=[456,'Andrew']
```

3. print(list)
4. print(list + list1)

Output:

```
['John', 678, 20.4, 'Peter']
['John', 678, 20.4, 'Peter', 456, 'Andrew']
```

Dictionary:

- Python dictionary stores the data in the key-value pair.
- It is enclosed by curly-braces {} and each pair is separated by the commas(,).

Example

1. dict = {'name': 'Pater', 'Age':18,'Roll_nu':101}
2. print(dict)

Output:

```
{'name': 'Pater', 'Age': 18, 'Roll_nu': 101}
```

Tuple:

- Python tuple is a collection of different data-type. It is immutable which means it cannot be modified after creation.
- It is enclosed by the parentheses () and each element is separated by the comma(,).

Example

1. tup = (10,20,"Dev",[2,3,4])
2. print(tup)

Output:

```
(10, 20, 'Dev', [2, 3, 4])
```

Set:

- Python set is the collection of the unordered dataset.
- It is enclosed by the {} and each element is separated by the comma(,).

Example: - Set Literals

1. set = {'apple','grapes','guava','papaya'}
2. print(set)

Output:

{'guava', 'apple', 'papaya', 'grapes'}

Python Operators

Introduction:

In this article, we are discussing Python Operators. The operator is a symbol that performs a specific operation between two operands, according to one definition. Operators serve as the foundation upon which logic is constructed in a program in a particular programming language. In every programming language, some operators perform several tasks. Same as other languages, Python also has some operators, and these are given below -

- Arithmetic operators
- Comparison operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators
- Arithmetic Operators

Arithmetic Operators

Arithmetic operators used between two operands for a particular operation. There are many arithmetic operators. It includes the exponent (**) operator as well as the + (addition), - (subtraction), * (multiplication), / (divide), % (reminder), and // (floor division) operators.

Consider the following table for a detailed explanation of arithmetic operators.

Operator	Description
+ (Addition)	It is used to add two operands. For example, if a = 10, b = 10 => a+b = 20
- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if a = 20, b = 5 => a - b = 15
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0
*** (Multiplication)**	It is used to multiply one operand with the other. For example, if a = 20, b = 4 => a * b = 80
% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0

**** (Exponent)**	As it calculates the first operand's power to the second operand, it is an exponent operator.
// (Floor division)	It provides the quotient's floor value, which is obtained by dividing the two operands.

Program Code:

Backward Skip 10sPlay VideoForward Skip 10s

Now we give code examples of arithmetic operators in Python. The code is given below -

```
1. a = 32    # Initialize the value of a
2. b = 6     # Initialize the value of b
3. print('Addition of two numbers:',a+b)
4. print('Subtraction of two numbers:',a-b)
5. print('Multiplication of two numbers:',a*b)
6. print('Division of two numbers:',a/b)
7. print('Reminder of two numbers:',a%b)
8. print('Exponent of two numbers:',a**b)
9. print('Floor division of two numbers:',a//b)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Addition of two numbers: 38
Subtraction of two numbers: 26
Multiplication of two numbers: 192
Division of two numbers: 5.333333333333333
Reminder of two numbers: 2
Exponent of two numbers: 1073741824
Floor division of two numbers: 5
```

Comparison operator

Comparison operators mainly use for comparison purposes. Comparison operators compare the values of the two operands and return a true or false Boolean value in accordance. The example of comparison operators are ==, !=, <=, >=, >, <. In the below table, we explain the works of the operators.

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal, then the condition becomes true.

<=	The condition is met if the first operand is smaller than or equal to the second operand.
>=	The condition is met if the first operand is greater than or equal to the second operand.
>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

Program Code:

Now we give code examples of Comparison operators in Python. The code is given below -


```
1. a = 32      # Initialize the value of a
2. b = 6       # Initialize the value of b
3. print('Two numbers are equal or not:',a==b)
4. print('Two numbers are not equal or not:',a!=b)
5. print('a is less than or equal to b:',a<=b)
6. print('a is greater than or equal to b:',a>=b)
7. print('a is greater b:',a>b)
8. print('a is less than b:',a<b)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Two numbers are equal or not: False
Two numbers are not equal or not: True
a is less than or equal to b: False
a is greater than or equal to b: True
a is greater b: True
a is less than b: False
```

Assignment Operators

Operator	Description
=	It assigns the value of the right expression to the left operand.

+=	By multiplying the value of the right operand by the value of the left operand, the left operand receives a changed value. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

Using the assignment operators, the right expression's value is assigned to the left operand. There are some examples of assignment operators like =, +=, -=, *=, %=, **=, //=. In the below table, we explain the works of the operators.

Program Code:

Now we give code examples of Assignment operators in Python. The code is given below -

```
1. a = 32        # Initialize the value of a
2. b = 6         # Initialize the value of b
3. print('a=b:', a==b)
4. print('a+=b:', a+b)
5. print('a-=b:', a-b)
6. print('a*=b:', a*b)
7. print('a%=b:', a%b)
8. print('a**=b:', a**b)
9. print('a//=b:', a//b)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
a=b: False
a+=b: 38
a-=b: 26
a*=b: 192
```

```
a%=b: 2
a**=b: 1073741824
a//=b: 5
```

Bitwise Operators

The two operands' values are processed bit by bit by the bitwise operators. The examples of Bitwise operators are bitwise OR (|), bitwise AND (&), bitwise XOR (^), negation (~), Left shift (<<), and Right shift (>>). Consider the case below.

For example,

1. **if** a = 7
2. b = 6
3. then, binary (a) = 0111
4. binary (b) = 0110
5.
6. hence, a & b = 0011
7. a | b = 0111
8. a ^ b = 0100
9. ~ a = 1000
10. Let, Binary of x = 0101
11. Binary of y = 1000
12. Bitwise OR = 1101
13. 8 4 2 1
14. 1 1 0 1 = 8 + 4 + 1 = 13
15.
16. Bitwise AND = 0000
17. 0000 = 0
18.
19. Bitwise XOR = 1101
20. 8 4 2 1
21. 1 1 0 1 = 8 + 4 + 1 = 13
22. Negation of x = ~x = (-x) - 1 = (-5) - 1 = -6
23. ~x = -6

In the below table, we are explaining the works of the bitwise operators.

Operator	Description
& (binary and)	A 1 is copied to the result if both bits in two operands at the same location are 1. If not, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1.

^ (binary xor)	If the two bits are different, the outcome bit will be 1, else it will be 0.
~ (negation)	The operand's bits are calculated as their negations, so if one bit is 0, the next bit will be 1, and vice versa.
<< (left shift)	The number of bits in the right operand is multiplied by the leftward shift of the value of the left operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

Program Code:

Now we give code examples of Bitwise operators in Python. The code is given below -

```
a = 5        # initialize the value of a
b = 6        # initialize the value of b
print('a&b:', a&b)
print('a|b:', a|b)
print('a^b:', a^b)
print('~a:', ~a)
print('a<<b:', a<<b)
print('a>>b:', a>>b)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
a&b: 4
a|b: 7
a^b: 3
~a: -6
a<>b: 0
```

Logical Operators

Operator	Description
and	The condition will also be true if the expression is true. If the two expressions a and b are the same, then a and b must both be true.

or	The condition will be true if one of the phrases is true. If a and b are the two expressions, then an or b must be true if and is true and b is false.
not	If an expression **a** is true, then not (a) will be false and vice versa.

The assessment of expressions to make decisions typically uses logical operators. The examples of logical operators are and, or, and not. In the case of logical AND, if the first one is 0, it does not depend upon the second one. In the case of logical OR, if the first one is 1, it does not depend on the second one. Python supports the following logical operators. In the below table, we explain the works of the logical operators.

Program Code:

Now we give code examples of arithmetic operators in Python. The code is given below -

```
1. a = 5          # initialize the value of a
2. print(Is this statement true?:',a > 3 and a < 5)
3. print('Any one statement is true?:',a > 3 or a < 5)
4. print('Each statement is true then return False and vice-versa:',(not(a > 3 and a < 5)))
```

Output:

Now we give code examples of Bitwise operators in Python. The code is given below -

```
Is this statement true?: False
Any one statement is true?: True
Each statement is true then return False and vice-versa: True
```

Membership Operators

The membership of a value inside a Python data structure can be verified using Python membership operators. The result is true if the value is in the data structure; otherwise, it returns false.

Operator	Description
in	If the first operand cannot be found in the second operand, it is evaluated to be true (list, tuple or dictionary).
not in	If the first operand is not present in the second operand, the evaluation is true (list, tuple or dictionary).

Program Code:

Now we give code examples of Membership operators in Python. The code is given below -

```
1. x = ["Rose", "Lotus"]
2. print(' Is value Present?', "Rose" in x)
3. print(' Is value not Present?', "Riya" not in x)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Is value Present? True
Is value not Present? True
```

Identity Operators

Operator	Description
is	If the references on both sides point to the same object, it is determined to be true.
is not	If the references on both sides do not point at the same object, it is determined to be true.

Program Code:

Now we give code examples of Identity operators in Python. The code is given below -

```
a = ["Rose", "Lotus"]
b = ["Rose", "Lotus"]
c = a
print(a is c)
print(a is not c)
print(a is b)
print(a is not b)
print(a == b)
print(a != b)
```

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
True
False
False
True
True
False
```

Operator Precedence

The order in which the operators are examined is crucial to understand since it tells us which operator needs to be considered first. Below is a list of the Python operators' precedence tables.

Operator	Description
**	Overall other operators employed in the expression, the exponent operator is given precedence.
~ + -	the minus, unary plus, and negation.
* / % //	the division of the floor, the modules, the division, and the multiplication.
+ -	Binary plus, and minus
>> <<	Left shift. and right shift
&	Binary and.
^ |	Binary xor, and or
<= < > >=	Comparison operators (less than, less than equal to, greater than, greater then equal to).
<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Python Comments

We'll study how to write comments in our program in this article. We'll also learn about single-line comments, multi-line comments, documentation strings, and other Python comments.

Introduction to Python Comments

We may wish to describe the code we develop. We might wish to take notes of why a section of script functions, for instance. We leverage the remarks to accomplish this. Formulas, procedures, and sophisticated business logic are typically explained with comments. The Python interpreter overlooks the remarks and solely interprets the script when running a program. Single-line comments, multi-line comments, and documentation strings are the 3 types of comments in Python.

Advantages of Using Comments

Our code is more comprehensible when we use comments in it. It assists us in recalling why specific sections of code were created by making the program more understandable.

Aside from that, we can leverage comments to overlook specific code while evaluating other code sections. This simple technique stops some lines from running or creates a fast pseudo-code for the program.

Below are some of the most common uses for comments:

- Readability of the Code
- Restrict code execution
- Provide an overview of the program or project metadata
- To add resources to the code

Types of Comments in Python

In Python, there are 3 types of comments. They are described below:

Single-Line Comments

Single-line remarks in Python have shown to be effective for providing quick descriptions for parameters, function definitions, and expressions. A single-line comment of Python is the one that has a hashtag # at the beginning of it and continues until the finish of the line. If the comment continues to the next line, add a hashtag to the subsequent line and resume the conversation. Consider the accompanying code snippet, which shows how to use a single line comment:

Code

```
# This code is to show an example of a single-line comment
print( 'This statement does not have a hashtag before it' )
```

Output:

```
This statement does not have a hashtag before it
```

The following is the comment:

```
# This code is to show an example of a single-line comment
```

The Python compiler ignores this line.

Everything following the # is omitted. As a result, we may put the program mentioned above in one line as follows:

Code

```
print( 'This is not a comment' ) # this code is to show an example of a single-line comment
```

Output:

```
This is not a comment
```

This program's output will be identical to the example above. The computer overlooks all content following #.

Multi-Line Comments

Python does not provide the facility for multi-line comments. However, there are indeed many ways to create multi-line comments.

With Multiple Hashtags (#)

In Python, we may use hashtags (#) multiple times to construct multiple lines of comments. Every line with a (#) before it will be regarded as a single-line comment.

Code

```
# it is a
# comment
# extending to multiple lines
```

In this case, each line is considered a comment, and they are all omitted.

Using String Literals

Because Python overlooks string expressions that aren't allocated to a variable, we can utilize them as comments.

Code

```
'it is a comment extending to multiple lines'
```

We can observe that on running this code, there will be no output; thus, we utilize the strings inside triple quotes(""") as multi-line comments.

Python Docstring

The strings enclosed in triple quotes that come immediately after the defined function are called Python docstring. It's designed to link documentation developed for Python modules, methods, classes, and functions together. It's placed just beneath the function, module, or class to explain what they perform. The docstring is then readily accessible in Python using the __doc__ attribute.

Code

```
# Code to show how we use docstrings in Python

def add(x, y):
    """This function adds the values of x and y"""
    return x + y

# Displaying the docstring of the add function
print( add.__doc__ )
```

Output:

```
This function adds the values of x and y
```

Type Conversion in Python

Introduction:

In this article, we are discussing type conversion in Python. It converts the Py-type data into another form of data. It is a conversion technique. Implicit type translation and explicit type converter are Python's two basic categories of type conversion procedures.

Python has type conversion routines that allow for the direct translation of one data type to another. This is very helpful for competitive programming as well as routine programming. This page aims to enlighten readers about specific conversion functions.

In Python, there are two kinds of type conversion, these are -

a. Explicit Type Conversion-The programmer must perform this task manually.

b. Implicit Type Conversion-By the Python program automatically.

Implicit Type Conversion

Implicit character data conversion is used in Python when a data type conversion occurs, whether during compilation or runtime. We do not need to manually change the file format into some other data type because Python performs the implicit character data conversion. Throughout the tutorial, we have seen numerous examples of this type of data type conversion. Without user input, the Programming language automatically changes one data type to another in an implicit shift of data types.

Program code 1:

Here we give an example of implicit type conversion in Python. In the below code, we initialize some values and find their data type in Python. So, the code is given below -

```
1. a = 15
2. print("Data type of a:",type(a))
3. b = 7.6
4. print("Data type of b:",type(b))
5. c = a + b
6. print("The value of c:", c)
7. print("Data type of c:",type(c))
8. d = "Priya"
9. print("Data type of d:",type(d))
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Data type of a: <class 'int'>
Data type of b: <class 'float'>
```

```
The value of c: 22.6
Data type of c: <class 'float'>
Data type of d: <class 'str'>
```

Explanation:

As we can see, while one variable, a, is only of integer type and the other, b, is of float type, the data type of "c" was automatically transformed to the "float" type. Due to type promotion, which enables operations by transforming raw data into a wider-sized type of data without any information loss, the float value is instead not turned into an integer. This is a straightforward instance of Python's Implicit type conversion.

The result variable was changed into the float dataset rather than the int data type since doing so would have required the compiler to eliminate the fractional element, which would have resulted in data loss. Python always translates smaller data types into larger data types to prevent data loss.

We may lose the decimal portion of the report if the variable percentage's data type is an integer. Python automatically converts percentage data to float type, which can store decimal values, to prevent this data loss.

Program code 2:

Here we give another example of implicit type conversion in Python. In the below code, we take user inputs and find their data type in Python. So, the code is given below -

```
1. x = input()
2. print("Data type of x:",type(x))
3. y = int(input())
4. print("Data type of y:",type(y))
5. z = float(input())
6. print("Data type of z:",type(z))
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Priya
Data type of x: <class 'str'>
23
Data type of y: <class 'int'>
0.5
Data type of z: <class 'float'>
```

Explanation:

In the above code, we take user input and find their data type. Here we take the value of x as a string, the value of y as an integer, and the value of z as a float. After taking the user input, they check their data type and print them.

Explicit Type Conversion:

Let us say we want to change a number from a higher data type to a lower data type. Implicit type conversion is ineffective in this situation, as we observed in the previous section. Explicit type conversion, commonly

referred to as type casting, now steps in to save the day. Using built-in Language functions like str() to convert to string form and int() to convert to integer type, a programmer can explicitly change the data form of an object by changing it manually.

The user can explicitly alter the data type in Python's Explicit Type Conversion according to their needs. Since we are compelled to change an expression into a certain data type when doing express type conversion, there is chance of data loss. Here are several examples of explicit type conversion.

a. The function int(a, base) transforms any data type to an integer. If the type of data is a string, "Base" defines the base wherein string is.

b. float(): You can turn any data type into a floating-point number with this function.

Program code:

Here we give another example of explicit type conversion in Python. In the below code, we initialize the values and find their data type in Python. So, the code is given below -

```
# Python code to show type conversion
# Initializing string with int() and float()
a = "10010"
# # outputting string to int base 2 conversion.
b = int(a,2)
print ("following the conversion to integer base 2: ", end="")
print (r)
# printing a float after converting a string
d = float(a)
print ("After converting to float : ", end="")
print (d)
```


Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
following the conversion to integer base 2: 18
After converting to float : 1010.0
```

c. The ord() method turns a character into an integer.

d. The hex() method turns an integer into a hexadecimal string.

e. The oct() method turns an integer into an octal string.

Program code:

Here we gave an example of finding the integer, hexadecimal, and octal values of given values. The code is given below -

```
# Python code to show type conversion
 # initialising integer
 # using ord(), hex(), and oct()
a = '4'
# printing and converting a character to an integer
b = ord(a)
print ("After converting character into integer : ",end="")
print (b)
# printing integer converting to hexadecimal string
b = hex(56)
print ("After converting 56 to hexadecimal string : ",end="")
print (b)
# printing the integer converting into octal string
b = oct(56)
print ("After converting 56 into octal string : ",end="")
print (b)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
After converting the character into integer : 52
After converting the 56 to hexadecimal string : 0x38
After converting the 56 into octal string : 0o70
```

f. The tuple() method is used to transform data into a tuple.

g. The set() function, which converts a type to a set, returns the set.

h. The list() function transforms any data type into a list type.

Program code:

Here we give an example of how to convert a given string into a list(), tuple(), and set(). The code is given below -

```
# Python code to show type conversion
 # Initializing string with tuple(), set(), and list()
x = 'javaTpoint'
# printing string into converting to tuple
y = tuple(x)
print ("After converting the string to a tuple: ",end="")
print (y)
# printing string the converting to set
y = set(x)
```

```
10. print ("After converting the string to a set: ",end="")
11. print (y)
12. # printing the string converting to list
13. y = list(x)
14. print ("After converting the string to the list: ",end="")
15. print (y)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
After converting the string to a tuple: ('j,' 'a', 'v,' 'a', 'T,' 'p,' 'o', 'i', 'n,' 't')
After converting the string to a set: {'n,' 'o', 'p,' 'j,' 'v,' 'T,' 'i', 'a', 't'}
After converting the string to the list: ['j,' 'a', 'v,' 'a', 'T,' 'p,' 'o', 'i', 'n,
```