Python Programming

Unit: 1

Introduction to Python: Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Python Variables: Python Variable is containers that store values. Python is not "statically typed". We do not need to declare variables before using them or declare their type. A variable is created the moment we first assign a value to it. A Python variable is a name given to a memory location. It is the basic unit of storage in a program.

An Example of a Variable in Python is a representational name that serves as a pointer to an object. Once an object is assigned to a variable, it can be referred to by that name. In layman's terms, we can say that Variable in Python is containers that store values.

Example:
Var = "Ram"
print(Var)
Output: Ram

Notes:

- The value stored in a variable can be changed during program execution.
- A Variables in Python is only a name given to a memory location, all the operations done on the variable effects that memory location.

Rules for Python variables:

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

Python basic Operators: In Python programming, Operators in general are used to perform operations on values and variables. These are standard symbols used for the purpose of logical and arithmetic operations.

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

Types of Operators in Python

1. Arithmetic Operators
2. Comparison Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators

6. Identity Operators and Membership Operators

1. Arithmetic Operators in Python:

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

Operator	Description	Syntax
+	Addition: adds two operands	x + y
–	Subtraction: subtracts two operands	x – y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when the first operand is divided by the second	x % y
**	Power: Returns first raised to power second	x ** y

Example of Arithmetic Operators in Python

Division Operators

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

1. Float division
2. Floor division

1. Float division

The quotient returned by this operator is always a float number, no matter if two numbers are integers. For example:

```
# python program to demonstrate the use of "/"

print(5/5)

print(10/2)

print(-10/2)

print(20.0/2)
```

Output:

```
1.0

5.0

-5.0

10.0
```

2. Integer division (Floor division)

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

```
# python program to demonstrate the use of "//"

print(10//3)

print (-5//2)

print (5.0//2)

print (-5.0//2)
```

Output:

3

-3

2.0

-3.0

Precedence of Arithmetic Operators in Python

The precedence of Arithmetic Operators in python is as follows:

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

6. S – Subtraction

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

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

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

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

```
# Examples of Arithmetic Operator

a = 9

b = 4

# Addition of numbers

add = a + b



# Subtraction of numbers

sub = a – b



# Multiplication of number
```

```
mul = a * b

# Modulo of both number

mod = a % b



# Power

p = a ** b



# print results

print(add)

print(sub)

print(mul)

print(mod)

print(p)
```

Output:

13

5

36

1

6561

2. Comparison Operators in Python

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

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

= is an assignment operator and == comparison operator.

Precedence of Comparison Operators in Python

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

Example of Comparison Operators in Python

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

```
# Examples of Relational Operators

a = 13

b = 33



# a > b is False

print(a > b)



# a < b is True

print(a < b)



# a == b is False

print(a == b)



# a != b is True

print(a != b)
```

```
# a >= b is False

print(a >= b)



# a <= b is True

print(a <= b)
```

Output

False

True

False

True

False

True

3. Logical Operators in Python

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

Operator	Description	Syntax
And	Logical AND: True if both the operands are true	x and y
Or	Logical OR: True if either of the operands is true	x or y

Operator	Description	Syntax
Not	Logical NOT: True if the operand is false	not x

Precedence of Logical Operators in Python

The precedence of Logical Operators in python is as follows:

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

Example of Logical Operators in Python

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

```
# Examples of Logical Operator

a = True

b = False



# Print a and b is False

print(a and b)



# Print a or b is True
```

```
print(a or b)



# Print not a is False

print(not a)
```

Output

False

True

False

4. Bitwise Operators in Python

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

Operator	Description	Syntax
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y

Operator	Description	Syntax
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Precedence of Bitwise Operators in Python

The precedence of Bitwise Operators in python is as follows:

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

Bitwise Operators in Python

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

```
# Examples of Bitwise operators

a = 10

b = 4



# Print bitwise AND operation

print(a & b)
```

```
# Print bitwise OR operation

print(a | b)


# Print bitwise NOT operation

print(~a)


# print bitwise XOR operation

print(a ^ b)


# print bitwise right shift operation

print(a >> 2)


# print bitwise left shift operation

print(a << 2)
```

Output

```
0
```

14

-11

14

2

40

5. Assignment Operators in Python

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

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

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

Assignment Operators in Python

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

```
# Examples of Assignment Operators

a = 10



# Assign value

b = a

print(b)



# Add and assign value

b += a

print(b)



# Subtract and assign value

b -= a

print(b)
```

```
# multiply and assign

b *= a

print(b)



# bitwise lishift operator

b <<= a

print(b)
```

Output

10

20

10

100

102400

Identity Operators in Python

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

is True if the operands are identical

is not True if the operands are not identical

Example Identity Operators in Python

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

```
a = 10

b = 20

c = a



print(a is not b)

print(a is c)
```

Output

```
True
True
```

Membership Operators in Python

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

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

Examples of Membership Operators in Python

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

```
# Python program to illustrate

# not 'in' operator
```

```
x = 24

y = 20

list = [10, 20, 30, 40, 50]



if (x not in list):

    print("x is NOT present in given list")

else:

    print("x is present in given list")



if (y in list):

    print("y is present in given list")

else:

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

Output

```
x is NOT present in given list
y is present in given list
```

Ternary Operator in Python

in Python, Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being true or false. It was added to Python in version 2.5.
It simply allows testing a condition in a **single line** replacing the multiline if-else making the code compact.
Syntax : *[on_true] if [expression] else [on_false]*

Examples of Ternary Operator in Python

Here is a simple example of Ternary Operator in Python.

```
# Program to demonstrate conditional operator

a, b = 10, 20



# Copy value of a in min if a < b else copy b

min = a if a < b else b



print(min)
```

Output:
10

Precedence and Associativity of Operators in Python

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

Operator Precedence in Python

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

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

```
# Examples of Operator Precedence



# Precedence of '+' & '*'

expr = 10 + 20 * 30

print(expr)



# Precedence of 'or' & 'and'

name = "Alex"

age = 0



if name == "Alex" or name == "John" and age >= 2:

    print("Hello! Welcome.")

else:
```

```
print("Good Bye!!")
```

Output

610

Hello! Welcome.

Operator Associativity in Python

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

The following code shows how Operator Associativity in Python works:

```
# Examples of Operator Associativity



# Left-right associativity

# 100 / 10 * 10 is calculated as

# (100 / 10) * 10 and not

# as 100 / (10 * 10)

print(100 / 10 * 10)
```

```
# Left-right associativity

# 5 - 2 + 3 is calculated as

# (5 - 2) + 3 and not

# as 5 - (2 + 3)

print(5 - 2 + 3)



# left-right associativity

print(5 - (2 + 3))



# right-left associativity

# 2 ** 3 ** 2 is calculated as

# 2 ** (3 ** 2) and not

# as (2 ** 3) ** 2

print(2 ** 3 ** 2)
```

Output

```
100.0
6
```

0

512

Understanding python blocks: A block in Python refers to a piece of code that performs a specific task. It can contain one or more statements and is defined by its indentation. Blocks are used to group statements together and provide structure to a program.

A block in Python is a group of one or more statements that perform a specific task. Blocks are defined by their indentation, which provides structure to the program. Indentation in Python is important as it defines the scope of a block and helps to keep the code organized.

For example, consider the following code:

```
if x > 0:
    print("x is positive")
    x = x + 1
```

In this code, the block is defined by the indentation of the two statements under the if clause. The block starts with the line print("x is positive") and ends with the line x = x + 1. The statements within the block will only be executed if the condition specified in the if clause is met.

Similarly, consider the following code:

```
for i in range(10):
    print(i)
```

In this code, the block is defined by the indentation of the statement under the for clause. The block starts with the line print(i) and is executed once for each iteration of the loop.

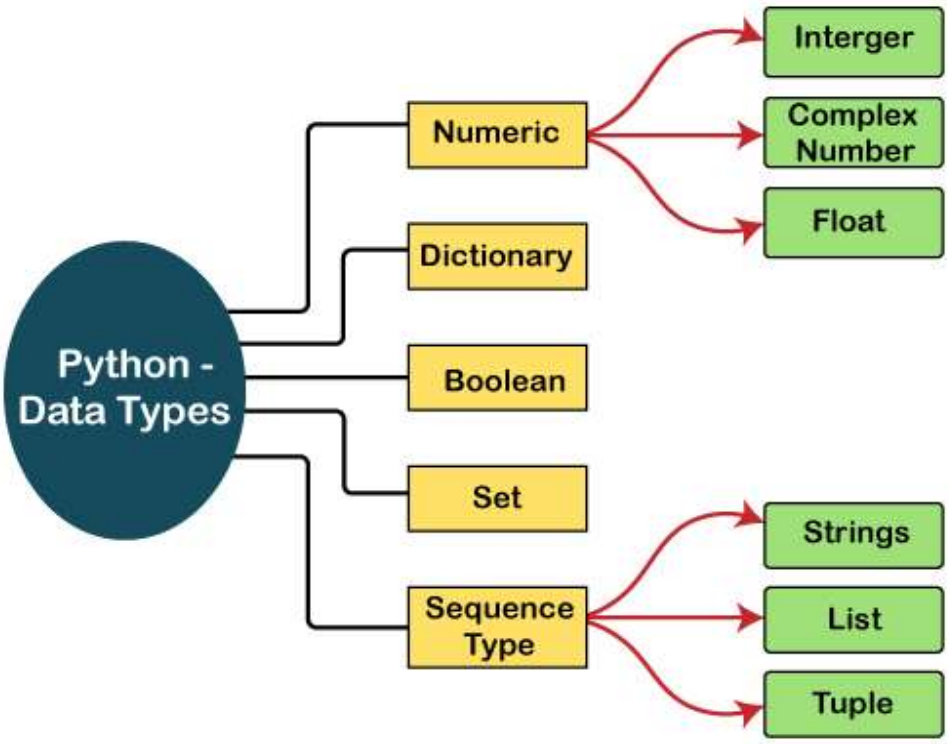
It's important to note that blocks in Python can be nested, meaning that a block can contain one or more blocks within it. This allows for the creation of complex programs that can perform a variety of tasks.

Blocks in Python are essential to understand as they provide structure and organization to programs. By grouping related statements together, blocks make code easier to read and maintain.

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

The following is a list of the Python-defined data types.

1. Numbers
2. Sequence Type
3. Boolean
4. Set
5. Dictionary



Numbers

Numeric values are stored in numbers. The whole number, float, and complex qualities have a place with a Python Numbers datatype. Python offers the type() function to determine a variable's data type. The instance () capability is utilized to check whether an item has a place with a specific class.

When a number is assigned to a variable, Python generates Number objects. For instance,

```
a = 5
print("The type of a", type(a))

b = 40.5
print("The type of b", type(b))

c = 1+3j
print("The type of c", type(c))
```

9. print(" c is a complex number", isinstance(1+3j,complex))

Output:

```
The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
c is complex number: True
```

Python supports three kinds of numerical data.

- **Int:** Whole number worth can be any length, like numbers 10, 2, 29, -20, - 150, and so on. An integer can be any length you want in Python. Its worth has a place with int.
- **Float:** Float stores drifting point numbers like 1.9, 9.902, 15.2, etc. It can be accurate to within 15 decimal places.
- **Complex:** An intricate number contains an arranged pair, i.e., x + iy, where x and y signify the genuine and non-existent parts separately. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

Sequence Type
String

- The sequence of characters in the quotation marks can be used to describe the string. A string can be defined in Python using single, double, or triple quotes.
- String dealing with Python is a direct undertaking since Python gives worked-in capabilities and administrators to perform tasks in the string.
- When dealing with strings, the operation "hello"+" python" returns "hello python," and the operator + is used to combine two strings.
- Because the operation "Python" *2 returns "Python," the operator * is referred to as a repetition operator.
- The Python string is demonstrated in the following example.

Example - 1

1. str = "string using double quotes"

```
print(str)
s = """A multiline
string"'
print(s)
```

Output:

```
string using double quotes
A multiline
string
```

Look at the following illustration of string handling.

Example - 2

```
str1 = 'hello world' #string str1
str2 = ' how are you' #string str2
print (str1[0:2]) #printing first two character using slice operator
print (str1[4]) #printing 4th character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

Output:

```
he
o
hello worldhello world
hello world how are you
```

List

Lists in Python are like arrays in C, but lists can contain data of different types. The things put away in the rundown are isolated with a comma (,) and encased inside square sections [].

To gain access to the list's data, we can use slice [:] operators. Like how they worked with strings, the list is handled by the concatenation operator (+) and the repetition operator (*).

Look at the following example.

Example:

```
1. list1  = [1, "hi", "Python", 2]
2. #Checking type of given list
3. print(type(list1))
4.
5. #Printing the list1
6. print (list1)
7.
8. # List slicing
9. print (list1[3:])
10.
11.         # List slicing
12.         print (list1[0:2])
13.
14.         # List Concatenation using + operator
15.         print (list1 + list1)
16.
17.         # List repetation using * operator
18.         print (list1 * 3)
```

Output:

```
[1, 'hi', 'Python', 2]
[2]
[1, 'hi']
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
```

```
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
```

Tuple

In many ways, a tuple is like a list. Tuples, like lists, also contain a collection of items from various data types. A parenthetical space () separates the tuple's components from one another.

Because we cannot alter the size or value of the items in a tuple, it is a read-only data structure.

Let's look at a straightforward tuple in action.

Example:

```
1. tup  = ("hi", "Python", 2)
2. # Checking type of tup
3. print (type(tup))
4.
5. #Printing the tuple
6. print (tup)
7.
8. # Tuple slicing
9. print (tup[1:])
10.        print (tup[0:1])
11.
12.        # Tuple concatenation using + operator
13.        print (tup + tup)
14.
15.        # Tuple repatation using * operator
16.        print (tup * 3)
17.
18.        # Adding value to tup. It will throw an error.
19.        t[2] = "hi"
```

Output:

```
<class 'tuple'>
('hi', 'Python', 2)
('Python', 2)
('hi',)
('hi', 'Python', 2, 'hi', 'Python', 2)
('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)

Traceback (most recent call last):
  File "main.py", line 14, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

Dictionary

A dictionary is a key-value pair set arranged in any order. It stores a specific value for each key, like an associative array or a hash table. Value is any Python object, while the key can hold any primitive data type.

The comma (,) and the curly braces are used to separate the items in the dictionary.

Look at the following example.

```
1. d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}
2.
3. # Printing dictionary
4. print (d)
5.
6. # Accesing value using keys
7. print("1st name is "+d[1])
8. print("2nd name is "+ d[4])
9.
10.         print (d.keys())
```

11. `print (d.values())`

Output:

```
1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
dict_keys([1, 2, 3, 4])
dict_values(['Jimmy', 'Alex', 'john', 'mike'])
```

Boolean

True and False are the two default values for the Boolean type. These qualities are utilized to decide the given assertion valid or misleading. The class book indicates this. False can be represented by the 0 or the letter "F," while true can be represented by any value that is not zero.

Look at the following example.

```
1. # Python program to check the boolean type
2. print(type(True))
3. print(type(False))
4. print(false)
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

Set

The data type's unordered collection is Python Set. It is iterable, mutable(can change after creation), and has remarkable components. The elements of a set have no set order; It might return the element's altered sequence. Either a sequence of elements is passed through the curly braces and separated by a comma to create the set or the built-in function set() is used to create the set. It can contain different kinds of values.

Look at the following example.

```
1. # Creating Empty set
2. set1 = set()
3.
4. set2 = {'James', 2, 3,'Python'}
5.
6. #Printing Set value
7. print(set2)
8.
9. # Adding element to the set
10.
11.         set2.add(10)
12.         print(set2)
13.
14.         #Removing element from the set
15.         set2.remove(2)
16.         print(set2)
```

Output:

```
{3, 'Python', 'James', 2}
{'Python', 'James', 3, 2, 10}
{'Python', 'James', 3, 10}
```