Introduction

Unit-1

Introduction

Python is a high-level, general-purpose, and very popular programming language.

Python is a widely used general-purpose, high level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation.

It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

Python is also known as a general-purpose programming language, as it is used in the domains given below:

- Web Development
- Software Development
- Game Development
- AI & ML
- Data Analytics

Every programming language serves some purpose or use-case according to a domain, and Python is no exception.

An introduction to Python programming can help us understand the purpose of the language.

For example, Python is widely used in data science, machine learning, and artificial intelligence due to its simplicity and powerful libraries.

Similarly, JavaScript is the most popular language among web developers as it gives the developer the power to handle applications via different frameworks like React, Vue, Angular, which are used to build beautiful user interfaces.

Similarly, they have pros and cons at the same time. so if we consider python it is general-purpose which means it is widely used in every domain the reason is it's very simple to understand, scalable because of which the speed of development is so fast.

Now you get the idea why besides learning python it doesn't require any programming background so that's why it's popular amongst developers as well.

Python has simpler syntax similar to the English language and also the syntax allows developers to write programs with fewer lines of code. Since it is open-source there are many libraries available that make developers' jobs easy ultimately results in high productivity. They can easily focus on business logic and Its demanding skills in the digital era where information is available in large data sets.

Key Features of Python:

Readable and Expressive Syntax: Python's syntax is designed to be clear and concise, resembling pseudo-code in many cases. Its use of whitespace for code indentation enhances readability, making it easier to understand and maintain code.

Dynamic Typing and Strong Typing: Python is dynamically typed, meaning you don't need to declare the data type of variables explicitly. It also enforces strong typing, ensuring type safety and preventing common errors.

High-level Language: Python abstracts away low-level details, allowing developers to focus on solving problems rather than managing memory or system resources. This high-level nature makes Python suitable for rapid development and prototyping.

Interpreted and Interactive: Python is an interpreted language, which means that code is executed line by line, without the need for compilation. This enables interactive development through tools like the Python interpreter and Jupyter notebooks.

Multi-paradigm: Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming. This flexibility allows developers to choose the most suitable approach for their projects.

Large Standard Library: Python comes with a comprehensive standard library that provides support for many common tasks and functionalities. This includes modules for file I/O, networking, data manipulation, and more, reducing the need for third-party dependencies.

Rich Ecosystem: Python has a vast ecosystem of third-party libraries and frameworks contributed by its active community. These libraries cover various domains such as web development (Django, Flask), data science (NumPy, pandas), machine learning (scikit-learn, TensorFlow), and more.

Cross-platform Compatibility: Python code is highly portable and can run on different operating systems, including Windows, macOS, and Linux. This cross-platform compatibility ensures that Python applications behave consistently across different environments.

Scalability and Performance: While Python is known for its simplicity and ease of use, it's also highly scalable. Performance-critical parts of Python applications can be optimized using techniques such as code profiling, caching, and leveraging compiled extensions.

Open Source and Community-driven: Python is open source, meaning its source code is freely available and can be modified and redistributed. This fosters a collaborative development environment, where developers worldwide contribute to the language's evolution and improvement.

We will see this in lab

- Installing and Running Python using Interactive Shell and Console
- Using IDLE and IDE
- Installing Third Party Libraries
- Working with Virtual Environment

Simple Program

```
print("Hello, World!")
```

To run program just go to terminal and type python3 and filename

Eg: python3 hello.py

Writing Comments

Single-line Comments: Single-line comments begin with the ‘#’ symbol and continue until the end of the line. They're commonly used for short, inline explanations.

```
# This is a single-line comment

print("Hello, World!")  # This comment explains the purpose of the print statement
```

Multi-line Comments:

Python doesn't have a specific syntax for multi-line comments like some other programming languages, but you can use a series of single-line comments to achieve a similar effect.

Alternatively, you can enclose multi-line comments within triple quotes (''' or """). While these triple-quoted strings are technically not comments, they're often used for documentation and can serve the same purpose.

The next way is by using string literals but not assigning them to any variables. If you do not assign a string literal to a variable, the Python interpreter ignores it. Use this to your advantage to write multi-line comments. You can either use a single (‘’) quotation or double (“”) quotation.

1. """

   This is a multi-line comment.

   It spans multiple lines.

   """

2. '''

   Another way to write a multi-line comment is using triple single quotes.

   '''

3. # You can also use multiple single-line comments to achieve the same effect:

   # This is a multi-line comment.

   # It spans multiple lines.

Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.
- In Python, indentation plays a crucial role in defining the structure and logic of your code. Unlike many other programming languages that use curly braces {} or keywords like begin and end to denote blocks of code, Python uses indentation to indicate the beginning and end of blocks

Here are some key points about Python indentation:

- Whitespace Matters: In Python, the amount of whitespace (spaces or tabs) at the beginning of a line is significant. It determines which statements are part of the same block of code.
- Consistent Indentation: It's essential to use consistent indentation throughout your code. Typically, four spaces are used for each level of indentation. Mixing spaces and tabs for indentation is not recommended and can lead to errors.
- Indentation Levels: Blocks of code with the same level of indentation are considered part of the same code block. When you decrease the indentation level, you're ending the current block. Python uses indentation to determine the scope of statements in loops, conditionals, and function definitions.
- Colon to Indicate Blocks: In Python, a colon (:) is used to indicate the beginning of an indented code block. For example, after an if statement or a loop declaration, you use a colon followed by an indented block of code.

Example

```
if 5 > 2:
  print("Five is greater than two!")
```

Note: Python will give you an error if you skip the indentation:

```
if 5 > 2:
print("Five is greater than two!")
```
-> Syntax Error

Tokens

- In Python, tokens are the basic building blocks of a program.
- They are the smallest units of a Python program, and the Python interpreter recognizes them as meaningful elements.
- Here are the main types of tokens in Python:
    - Identifiers
    - Keywords
    - Literals
    - Operators
    - Delimiters

Identifiers

Identifiers are names given to entities in a Python program, such as variables, functions, classes, modules, etc.

- Rules for identifiers:
  - Must start with a letter (a-z, A-Z) or an underscore (_).
  - Can be followed by letters, digits (0-9), or underscores.
  - Cannot be a Python keyword.
  - Case-sensitive (myVar is different from MyVar).

Examples of valid identifiers: my_variable, myFunction, PI, _private_variable

Keywords

- Keywords are reserved words that have special meaning in Python and cannot be used as identifiers.
- Examples of keywords: if, else, for, while, def, class, import, True, False, None, etc.
- You can get a list of all keywords in Python by using the keyword module:

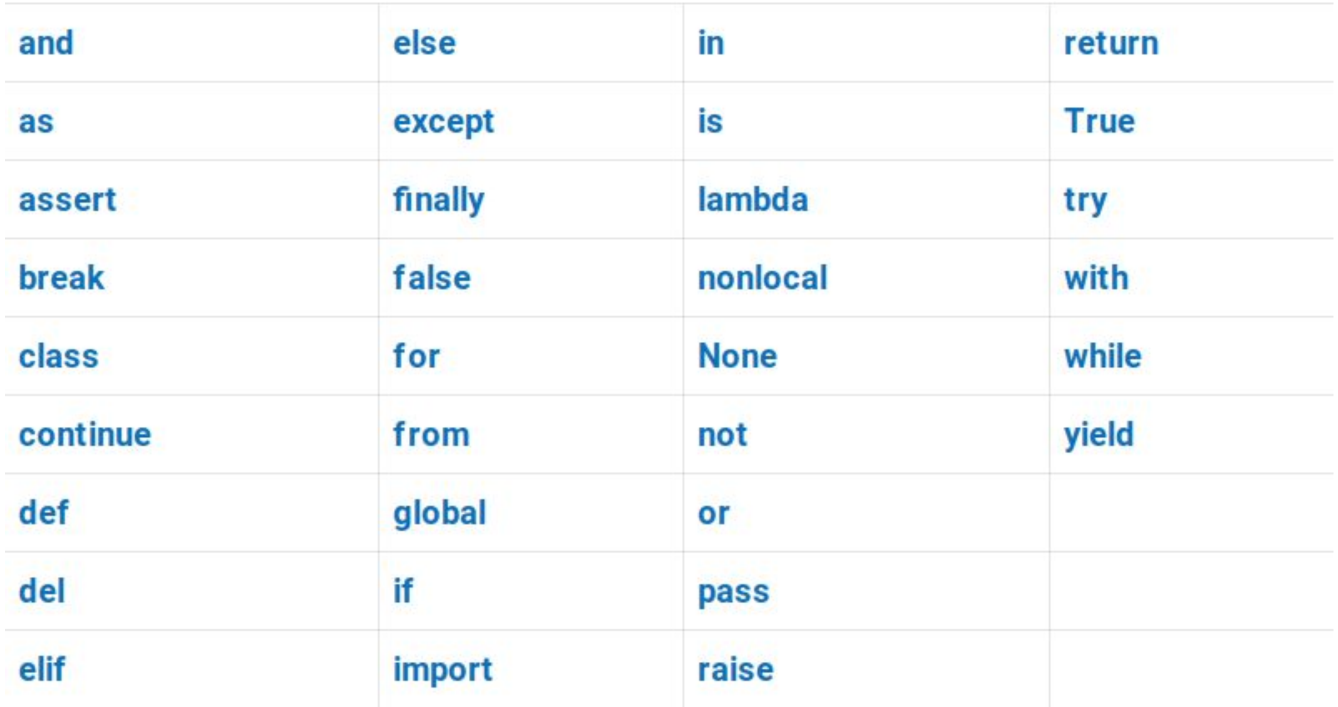
```
import keyword

print(keyword.kwlist)
```

Note:

Python keywords are different from Python’s built-in functions and types. The built-in functions and types are also always available, but they aren’t as restrictive as the keywords in their usage.

An example of something you can’t do with Python keywords is assign something to them. If you try, then you’ll get a SyntaxError. You won’t get a SyntaxError if you try to assign something to a built-in function or type, but it still isn’t a good idea.

As of Python 3.8, there are thirty-five keywords in Python.



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

Literals

Literals are literal constants used in Python programs. They represent fixed values.

Examples of literals:

- 42 (integer literal),
- 3.14 (floating-point literal),
- 'hello' (string literal),
- True (boolean literal),
- None (NoneType literal), etc.

Types of Literals in Python

- String literals
- Character literal
- Numeric literals
- Boolean literals
- Literal Collections
- Special literals

Python String Literals

A string is literal and can be created by writing a text(a group of Characters) surrounded by a single(”), double(“), or triple quotes.

We can write multi-line strings or display them in the desired way by using triple quotes.

```
# in single quote

        s = ‘NCCS’

# in double quotes

        t = "NCCS"

# multi-line String

        m = “””National

        College of

        Computer studies”””
```

Python Character literal

It is also a type of Python string literal where a single character is surrounded by single or double quotes.

```
# character literal in single quote

    v = 'n'


# character literal in double quotes

    w = "a"
```

Python Numeric literal

They are immutable and there are three types of numeric literal:

- Integer
- Float
- Complex

```
# integer literal
# Binary Literals
a = 0b10100
# Decimal Literal
b = 50
# Octal Literal
c = 0o320
# Hexadecimal Literal
d = 0x12b
```

```
# Float Literal

e = 24.8

f = 45.0
```

Complex

In Python, complex literals represent complex numbers. A complex number consists of a real part and an imaginary part, both of which are floating-point numbers. The imaginary part is indicated by the suffix j or J.

```
# Real part = 3, Imaginary part = 2

z1 = 3 + 2j

# Real part = -1.5, Imaginary part = 4.2

z2 = -1.5 + 4.2j

# Real part = 0, Imaginary part = -7

z3 = 0 - 7j

# Real part = 0.0, Imaginary part = 1.0

z4 = 1j
```

Python Boolean literal

There are only two Boolean literals in Python. They are true and false. In Python, True represents the value as 1, and False represents the value as 0. In this example ‘a‘ is True and ‘b‘ is False because 1 is equal to True.

```
a = (1 == True)

b = (1 == False)

c = True + 3

d = False + 7

print("a is", a)

print("b is", b)

print("c:", c)

print("d:", d)
```

Python Special literal

Python contains one special literal (None). ‘None’ is used to define a null variable. If ‘None’ is compared with anything else other than a ‘None’, it will return false.

eg:

```
water_remain = None

print(water_remain)
```

Python literal collections

Python provides four different types of literal collections:

- List literals
- Tuple literals
- Dict literals
- Set literals

List literal

```
number = [1, 2, 3, 4, 5]

name = [NCCS, PRIME, KAKASHI, 1]

print(number)

print(name)
```

Tuple literal

A tuple is a collection of different data-type. It is enclosed by the parentheses ‘()‘ and each element is separated by the comma(,). It is immutable.

eg:

```
even_number = (2, 4, 6, 8)

odd_number = (1, 3, 5, 7)


print(even_number)

print(odd_number)
```

Dictionary literal

The dictionary stores the data in the key-value pair. It is enclosed by curly braces ‘{}‘ and each pair is separated by the commas(,). We can store different types of data in a dictionary. Dictionaries are mutable.

eg:

```
alphabets = {'a': 'apple', 'b': 'ball', 'c': 'cat'}

information = {'name': 'amit', 'age': 20, 'ID': 20}


print(alphabets)

print(information)
```

Set literal

In Python, a set literal represents a set, which is an unordered collection of unique elements. Set literals are created using curly braces {} with comma-separated elements inside.

Each element in a set must be unique. If you try to add a duplicate element to a set, it will be ignored.

Here's how you can create set literals:

```
vowels = {'a', 'e', 'i', 'o', 'u'}

fruits = {"apple", "banana", "cherry"}


print(vowels)

print(fruits)
```

Python Variables

In Python, a variable is a name that refers to a value stored in the computer's memory.

Variables are used to store and manipulate data within a program.

Unlike some other programming languages, Python is dynamically typed, meaning you don't need to declare the type of a variable before assigning a value to it.

```python
# Assigning values to variables

name = "Alice"

age = 30

height = 5.9

is_student = True

# Accessing variable values

print("Name:", name)

print("Age:", age)

print("Height:", height)

print("Is Student:", is_student)
```

Note: Python is a dynamically typed language, so the type of a variable is determined dynamically based on the type of the value it refers to. You can reassign variables to values of different types without any issues.

Example

```
# Define a variable 'x' and assign an integer value

x = 5

print("x:", x, "Type:", type(x))  # Output: x: 5 Type: <class 'int'>

# Reassign 'x' to a string value

x = "hello"

print("x:", x, "Type:", type(x))  # Output: x: hello Type: <class 'str'>

# Reassign 'x' to a boolean value

x = True

print("x:", x, "Type:", type(x))  # Output: x: True Type: <class 'bool'>

# Reassign 'x' to a floating-point value

x = 3.14

print("x:", x, "Type:", type(x))  # Output: x: 3.14 Type: <class 'float'>
```

Python Assign Values to Multiple Variables

Also, Python allows assigning a single value to several variables simultaneously with “=” operators.

For example:

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

Assigning different values to multiple variables

Python allows adding different values in a single line with “,” operators.

eg:

```
a, b, c = 1, 20.2, "GeeksforGeeks"

print(a)
print(b)
print(c)
```

Can We Use the Same Name for Different Types?

If we use the same name, the variable starts referring to a new value and type.

eg:

```
a = 10

a = "GeeksforGeeks"


print(a)
```

How does + operator work with variables?

The Python plus operator + provides a convenient way to add a value if it is a number and concatenate if it is a string.

If a variable is already created it assigns the new value back to the same variable.

eg:

```
a = 10

b = 20

print(a+b)


a = "NCCS"

b = "College"

print(a+b)
```

Can we use + for different Datatypes also?

No use for different types would produce an error.

```
a = 10

b = "Geeks"

print(a+b)
```

Output:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Types of variable in python

Based on Scope:

- Global Variables
- Local Variables

Based on Mutability:

- Mutable Variables
- Immutable Variables

Based on Lifetime:

- Instance Variables
- Class Variables

Based on Accessibility:

- Public Variables
- Private Variables

Global Variables

Global variables are variables that are defined outside of any function and can be accessed from anywhere within the program, including inside functions.

```
# Global variable

global_var = 10

# Function accessing global variable

def my_function():

    print("Inside the function - Global variable:", global_var)

# Call the function

my_function()

# Accessing global variable outside the function

print("Outside the function - Global variable:", global_var)
```

In this example, global_var is a global variable. It is defined outside any function and can be accessed both inside and outside the my_function() function. When we call my_function(), it prints the value of global_var from within the function. Similarly, we can also access and print the value of global_var outside the function.

Local Variables

Local variables are variables that are defined within a function and can only be accessed from within that function.

```
def my_function():

    # Local variable

    local_var = 20

    print("Inside the function - Local variable:", local_var)


# Call the function

my_function()
```

Python Constants

A Python Constant is a variable whose value cannot be changed throughout the program.

Note – Unlike other programming languages, Python does not contain any constants. Instead, Python provides us a Capitalized naming convention method. Any variable written in the Upper case is considered as a Constant in Python.

Rules to be followed while declaring a Constant

- Python Constants and variable names should contain a combination of lowercase (a-z) or capital (A-Z) characters, numbers (0-9), or an underscore (_).
- When using a Constant name, always use UPPERCASE, For example, CONSTANT = 50.
- The Constant names should not begin with digits.
- Except for underscore(_), no additional special character (!, #, ^, @, $) is utilized when declaring a constant.
- We should come up with a catchy name for the python constants. VALUE, for example, makes more sense than V. It simplifies the coding process.

Assigning Values to Constants

Constants are typically declared and assigned in a module in Python. In this case, the module is a new file containing variables, functions, and so on that is imported into the main file. Constants are written in all capital letters with underscores separating the words within the module.

We usually create a separate file for declaring constants. We then use this file to import the constant module in the main.py file from the other file.

```python
# Define constants (usually written in uppercase)

PI = 3.14159

GRAVITY = 9.81

SPEED_OF_LIGHT = 299792458

# Use constants in calculations or wherever their values are needed

radius = 5

area = PI * radius**2

print("Area of the circle with radius", radius, "is", area)

# Attempting to modify a constant (will result in an error)

# PI = 3.14  # Uncommenting this line will raise an error
```

Another example

```
# create a separate constant.py file
PI = 3.14
GRAVITY = 9.8


# main.py file
import constant as const
print('Value of PI:', cons.PI)
print('Value of Gravitational force:', cons.GRAVITY)
```

id() function

In Python, every data element or object stored in the memory is been referenced by a numeric value.

These numeric values are useful in order to distinguish them from other values. This is referred to as the identity of an object.

The python id() function is used to return a unique identification value of the object stored in the memory.

Definition:

The Python id() function accepts a single parameter and returns a unique identity (integer value) of that object.

eg:

```
# Python program to illustrate id() function

a = 10

b = 11

c = 130.56

text = 'Hello'


print('ID of a =', id(a))

print('ID of b =', id(b))

print('ID of c =', id(c))

print('ID of text =', id(text))
```

More Examples on id()

An instance where ID’s are unidentical

```
# Python program to illustrate id() function

a = 10

b = 14

text1 = 'Hello'

text2 = 'Hi'


print(id(a) == id(b))

print(id(text1) == id(text2))
```

Output –

```
False

False
```

An instance where ID’s are identical

There are few cases when the Python id() function assigns the exact same identification number to multiple objects.

This happens when the objects, like integers, floats, strings, tuples are immutable.

In Python, two variables referring to immutable objects (such as integers, strings, or tuples) may have identical IDs if their values are the same and fall within a certain range.

This behavior is known as "object interning" and is an optimization technique used by Python to improve memory efficiency.

The id() function works on the principle of caching which in turn works only on the objects that are immutable. This helps Python preserve memory.

Eg:

```
# Two variables with the same value
a = 10
b = 10

# Checking IDs
print("ID of 'a':", id(a))
print("ID of 'b':", id(b))
```

Output:

```
ID of 'a': 140716155852992

ID of 'b': 140716155852992
```

```
# Python program to illustrate id() function

my_list1 = [1, 2, 3, 4]

my_list2 = [1, 2, 3, 4]


my_dict1 = {1: 'Hi', 2: 'Hello'}

my_dict2 = {1: 'Hi', 2: 'Hello'}


print(id(my_list1))

print(id(my_list2))

print(id(my_dict2))

print(id(my_dict2))
```

Output:

```
140117770739136

140117770107456

140117770935360

140117770935360
```

Operators

In Python, operators are special symbols or keywords that carry out operations on values and python variables.

They serve as a basis for expressions, which are used to modify data and execute computations. Python contains several operators, each with its unique purpose.

In the example below, we use the + operator to add together two values:

```
print(10 + 5)
```

Types of Python Operators

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic Operators

Mathematical operations including addition, subtraction, multiplication, and division are commonly carried out using Python arithmetic operators.

They are compatible with integers, variables, and expressions.

In addition to the standard arithmetic operators, there are operators for modulus, exponentiation, and floor division.

Operator	Name	Example
+	Addition	10 + 20 = 30
-	Subtraction	20 – 10 = 10
*	Multiplication	10 * 20 = 200
/	Division	20 / 10 = 2
%	Modulus	22 % 10 = 2
**	Exponent	4**2 = 16
//	Floor Division	9//2 = 4

example

```
a = 21
b = 10
# Addition
print ("a + b : ", a + b)
# Subtraction
print ("a - b : ", a - b)
# Multiplication
print ("a * b : ", a * b)
# Division
print ("a / b : ", a / b)
# Modulus
print ("a % b : ", a % b)
# Exponent
print ("a ** b : ", a ** b)
# Floor Division
print ("a // b : ", a // b)
```

Output

```
a + b : 31

a - b : 11

a * b : 210

a / b : 2.1

a % b : 1

a ** b : 16679880978201

a // b : 2
```

Comparison Operators

To compare two values, Python comparison operators are needed.

Based on the comparison, they produce a Boolean value (True or False).

Operator	Name	Example
==	Equal	4 == 5 is not true.
!=	Not Equal	4 != 5 is true.
>	Greater Than	4 > 5 is not true
<	Less Than	4 < 5 is true
>=	Greater than or Equal to	4 >= 5 is not true.
<=	Less than or Equal to	4 <= 5 is true.

```
a = 4
b = 5
# Equal
print ("a == b : ", a == b)
# Not Equal
print ("a != b : ", a != b)
# Greater Than
print ("a > b : ", a > b)
# Less Than
print ("a < b : ", a < b)
# Greater Than or Equal to
print ("a >= b : ", a >= b)
# Less Than or Equal to
print ("a <= b : ", a <= b)
```

Output

```
a == b : False

a != b : True

a > b : False

a < b : True

a >= b : False

a <= b : True
```

Assignment Operators

Python assignment operators are used to assign values to variables in Python.

The single equal symbol (=) is the most fundamental assignment operator.

It assigns the value on the operator's right side to the variable on the operator's left side.

Operator	Name	Example
=	Assignment Operator	a = 10
+=	Addition Assignment	a += 5 (Same as a = a + 5)
-=	Subtraction Assignment	a -= 5 (Same as a = a - 5)
*=	Multiplication Assignment	a *= 5 (Same as a = a * 5)
/=	Division Assignment	a /= 5 (Same as a = a / 5)
%=	Remainder Assignment	a %= 5 (Same as a = a % 5)
**=	Exponent Assignment	a **= 2 (Same as a = a ** 2)
//=	Floor Division Assignment	a //= 3 (Same as a = a // 3)

```
# Assignment Operator
a = 10
# Addition Assignment
a += 5
print ("a += 5 : ", a)
# Subtraction Assignment
a -= 5
print ("a -= 5 : ", a)
# Multiplication Assignment
a *= 5
print ("a *= 5 : ", a)
# Division Assignment
a /= 5
print ("a /= 5 : ",a)
# Remainder Assignment
a %= 3
print ("a %= 3 : ", a)
# Exponent Assignment
a **= 2
print ("a **= 2 : ", a)
# Floor Division Assignment
a //= 3
print ("a //= 3 : ", a)
```

Output

```
a += 5 : 105

a -= 5 : 100

a *= 5 : 500

a /= 5 : 100.0

a %= 3 : 1.0

a **= 2 : 1.0

a //= 3 : 0.0
```

Bitwise Operators

Python bitwise operators execute operations on individual bits of binary integers.

They work with integer binary representations, performing logical operations on each bit location.

Python includes various bitwise operators, such as AND (&), OR (|), NOT (), XOR (), left shift (), and right shift (>>).

Operator	Name	Example
&	Binary AND	Sets each bit to 1 if both bits are 1
|	Binary OR	Sets each bit to 1 if one of the two bits is 1
^	Binary XOR	Sets each bit to 1 if only one of two bits is 1
~	Binary Ones Complement	Inverts all the bits
~	Binary Ones Complement	Inverts all the bits
<<	Binary Left Shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Binary Right Shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Logical Operators

Python logical operators are used to compose Boolean expressions and evaluate their truth values.

They are required for the creation of conditional statements as well as for managing the flow of execution in programs.

Python has three basic logical operators: AND, OR, and NOT.

Operator	Description	Example
and Logical AND	If both of the operands are true then the condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands is non-zero then the condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand	Not(a and b) is false.

```
x = 5

y = 10

if x > 3 and y < 15:

  print("Both x and y are within the specified range")
```

The code assigns the values 5 and 10 to variables x and y. It determines whether x is larger than 3 and y is less than 15. If both conditions are met, it writes "Both x and y are within the specified range.”

Output:

```
Both x and y are within the specified range
```

Membership Operators

Python membership operators are used to determine whether or not a certain value occurs within a sequence.

They make it simple to determine the membership of elements in various data structures such as lists, tuples, sets, and strings.

Python has two primary membership operators: the in and not in operators.

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

```
fruits = ["apple", "banana", "cherry"]

if "banana" in fruits:

    print("Yes, banana is a fruit!")

else:

    print("No, banana is not a fruit!")
```

The code defines a list of fruits and tests to see if the word "banana" appears in the list. If it is, the message "Yes, banana is a fruit!" is displayed; otherwise, the message "No, banana is not a fruit!" is displayed.

Output

```
Yes, banana is a fruit!
```

Identity Operators

Python identity operators are used to compare two objects' memory addresses rather than their values.

If the two objects refer to the same memory address, they evaluate to True; otherwise, they evaluate to False.

Python includes two identity operators: the is and is not operators.

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise	x is y, here are results in 1 if id(x) equals id(y)
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise	x is not y, there are no results in 1 if id(x) is not equal to id(y).

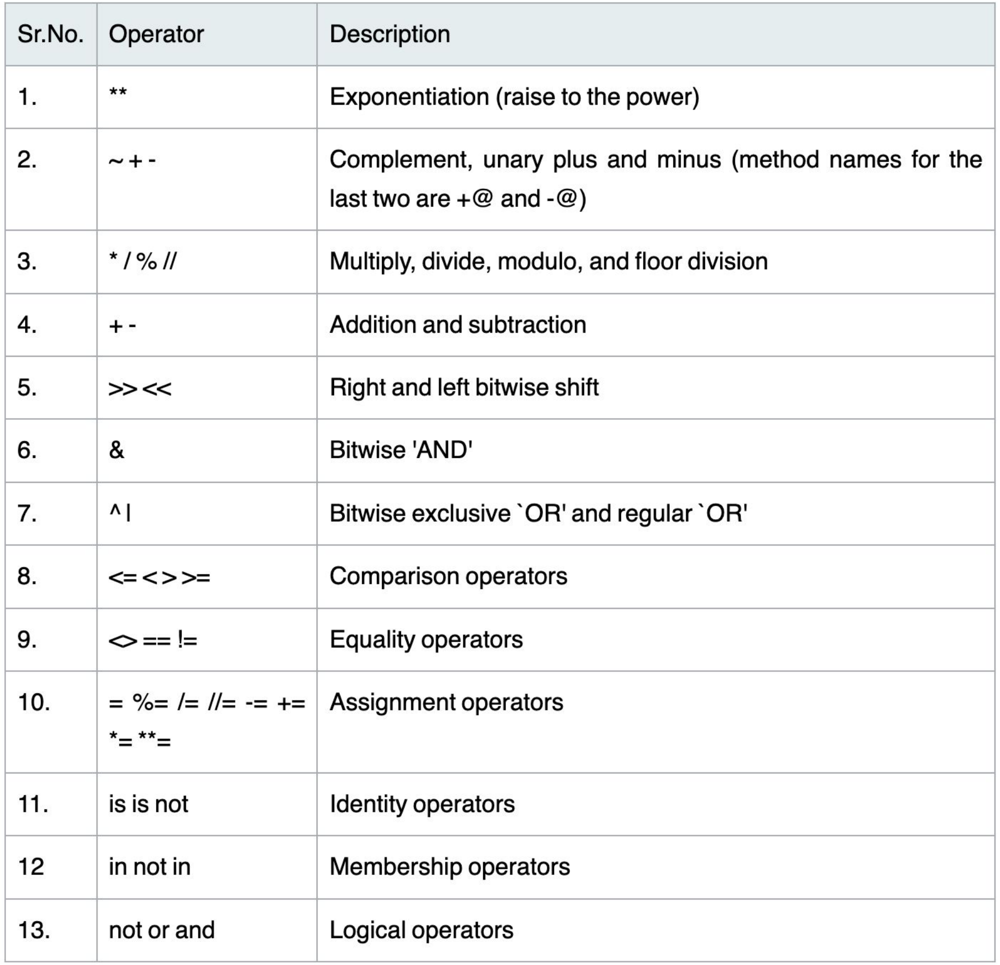
```
x = 10

y = 5

if x is y:

    print("x and y are the same object")

else:

    print("x and y are not the same object")
```

The code sets the variables x and y to 10 and 5, respectively. It then uses the is keyword to determine whether x and y relate to the same item in memory. If they are, it displays "x and y are the same object"; otherwise, it displays "x and y are not the same object."

Output

```
x and y are not the same object
```

Python Operators Precedence



Sr.No.	Operator	Description
1.	**	Exponentiation (raise to the power)
2.	~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
3.	* / % //	Multiply, divide, modulo, and floor division
4.	+ -	Addition and subtraction
5.	>> <<	Right and left bitwise shift
6.	&	Bitwise 'AND'
7.	^ |	Bitwise exclusive `OR' and regular `OR'
8.	<= < > >=	Comparison operators
9.	<> == !=	Equality operators
10.	= %= /= //= -= += *= **=	Assignment operators
11.	is is not	Identity operators
12	in not in	Membership operators
13.	not or and	Logical operators