UGCSA203: PYTHON PROGRAMMING (BCA)

Unit-1

Introduction:

Python is a widely used programming language that offers several unique features and advantages compared to languages like Java and C++. Our Python tutorial thoroughly explains Python basics and advanced concepts, starting with installation, conditional statements, loops, built-in data structures, Object-Oriented Programming, Generators, Exception Handling, Python RegEx, and many other concepts. This tutorial is designed for beginners and working professionals.

In the late 1980s, Guido van Rossum dreamed of developing Python. The first version of Python 0.9.0 was released in 1991. Since its release, Python started gaining popularity. According to reports, Python is now the most popular programming language among developers because of its high demands in the tech realm.

What is Python?

Python is a general-purpose, dynamically typed, high-level, compiled and interpreted, garbage-collected, and purely object-oriented programming language that supports procedural, object-oriented, and functional programming.

Features of Python:

- Easy to use and Read - Python's syntax is clear and easy to read, making it an ideal language for both beginners and experienced programmers. This simplicity can lead to faster development and reduce the chances of errors.
- Dynamically Typed - The data types of variables are determined during run-time. We do not need to specify the data type of a variable during writing codes.
- High-level - High-level language means human readable code.

- Compiled and Interpreted - Python code first gets compiled into bytecode, and then interpreted line by line. When we download the Python in our system form org we download the default implement of Python known as CPython. CPython is considered to be Complied and Interpreted both.
- Garbage Collected - Memory allocation and de-allocation are automatically managed. Programmers do not specifically need to manage the memory.
- Purely Object-Oriented - It refers to everything as an object, including numbers and strings.
- Cross-platform Compatibility - Python can be easily installed on Windows, macOS, and various Linux distributions, allowing developers to create software that runs across different operating systems.
- Rich Standard Library - Python comes with several standard libraries that provide ready-to-use modules and functions for various tasks, ranging from web development and data manipulation to machine learning and networking.
- Open Source - Python is an open-source, cost-free programming language. It is utilized in several sectors and disciplines as a result.

Python Basic Syntax

There is no use of curly braces or semicolons in Python programming language. It is an English-like language. But Python uses indentation to define a block of code. Indentation is nothing but adding whitespace before the statement when it is needed.

For example -

```
def func():
      statement 1
      statement 2
      …………………
      …………………
```

```
    Statement N
```

In the above example, the statements that are the same level to the right belong to the function. Generally, we can use four whitespaces to define indentation.

Instead of Semicolon as used in other languages, Python ends its statements with a Newline character.

Python is a case-sensitive language, which means that uppercase and lowercase letters are treated differently. For example, 'name' and 'Name' are two different variables in Python.

In Python, comments can be added using the '#' symbol. Any text written after the '#' symbol is considered a comment and is ignored by the interpreter. This trick is useful for adding notes to the code or temporarily disabling a code block. It also helps in understanding the code better by some other developers.

'If', 'otherwise', 'for', 'while', 'try', 'except', and 'finally' are a few reserved keywords in Python that cannot be used as variable names. These terms are used in the language for particular reasons and have fixed meanings. If you use these keywords, your code may include errors, or the interpreter may reject them as potential new Variables.

History of Python

- Python was created by Guido van Rossum. In the late 1980s, Guido van Rossum, a Dutch programmer, began working on Python while at the Centrum Wiskunde & Informatica (CWI) in the Netherlands. He wanted to create a successor to the ABC programming language that would be easy to read and efficient.
- In February 1991, the first public version of Python, version 0.9.0, was released. This marked the official birth of Python as an open-source project. The language was named after the British comedy series "Monty Python's Flying Circus".

- Python development has gone through several stages. In January 1994, Python 1.0 was released as a usable and stable programming language. This version included many of the features that are still present in Python today.
- From the 1990s to the 2000s, Python gained popularity for its simplicity, readability, and versatility. In October 2000, Python 2.0 was released. Python 2.0 introduced list comprehensions, garbage collection, and support for Unicode.
- In December 2008, Python 3.0 was released. Python 3.0 introduced several backward-incompatible changes to improve code readability and maintainability.
- Throughout 2010s, Python's popularity increased, particularly in fields like data science, machine learning, and web development. Its rich ecosystem of libraries and frameworks made it a favourite among developers.
- The Python Software Foundation (PSF) was established in 2001 to promote, protect, and advance the Python programming language and its community.

Why learn Python?

Python provides many useful features to the programmer. These features make it the most popular and widely used language.

- Easy to use and Learn: Python has a simple and easy-to-understand syntax, unlike traditional languages like C, C++, Java, etc., making it easy for beginners to learn.
- Expressive Language: It allows programmers to express complex concepts in just a few lines of code or reduces Developer's Time.
- Interpreted Language: Python does not require compilation, allowing rapid development and testing. It uses Interpreter instead of Compiler.
- Object-Oriented Language: It supports object-oriented programming, making writing reusable and modular code easy.
- Open-Source Language: Python is open-source and free to use, distribute and modify.

- Extensible: Python can be extended with modules written in C, C++, or other languages.
- Learn Standard Library: Python's standard library contains many modules and functions that can be used for various tasks, such as string manipulation, web programming, and more.
- GUI Programming Support: Python provides several GUI frameworks, such as Tkinter and PyQt, allowing developers to create desktop applications easily.
- Integrated: Python can easily integrate with other languages and technologies, such as C/C++, Java, and . NET.
- Embeddable: Python code can be embedded into other applications as a scripting language.
- Dynamic Memory Allocation: Python automatically manages memory allocation, making it easier for developers to write complex programs without worrying about memory management.
- Wide Range of Libraries and Frameworks: Python has a vast collection of libraries and frameworks, such as NumPy, Pandas, Django, and Flask, that can be used to solve a wide range of problems.
- Versatility: Python is a universal language in various domains such as web development, machine learning, data analysis, scientific computing, and more.
- Large Community: Python has a vast and active community of developers contributing to its development and offering support. This makes it easy for beginners to get help and learn from experienced developers.
- Career Opportunities: Python is a highly popular language in the job market. Learning Python can open up several career opportunities in data science, artificial intelligence, web development, and more.
- High Demand: With the growing demand for automation and digital transformation, the need for Python developers is rising. Many industries seek skilled Python developers to help build their digital infrastructure.

- Increased Productivity: Python has a simple syntax and powerful libraries that can help developers write code faster and more efficiently. This can increase productivity and save time for developers and organizations.
- Big Data and Machine Learning: Python has become the go-to language for big data and machine learning. Python has become popular among data scientists and machine learning engineers with libraries like NumPy, Pandas, Scikit-learn, TensorFlow, and more.

Where is Python used?

Python is a general-purpose, popular programming language, and it is used in almost every technical field. The various areas of Python use are given below.

- Data Science: Data Science is a vast field, and Python is an important language for this field because of its simplicity, ease of use, and availability of powerful data analysis and visualization libraries like NumPy, Pandas, and Matplotlib.
- Desktop Applications: PyQt and Tkinter are useful libraries that can be used in GUI - Graphical User Interface-based Desktop Applications. There are better languages for this field, but it can be used with other languages for making Applications.
- Console-based Applications: Python is also commonly used to create command-line or console-based applications because of its ease of use and support for advanced features such as input/output redirection and piping.
- Mobile Applications: While Python is not commonly used for creating mobile applications, it can still be combined with frameworks like Kivy or BeeWare to create cross-platform mobile applications.
- Software Development: Python is considered one of the best software-making languages. Python is easily compatible with both from Small Scale to Large Scale software.

- Artificial Intelligence: AI is an emerging Technology, and Python is a perfect language for artificial intelligence and machine learning because of the availability of powerful libraries such as TensorFlow, Keras, and PyTorch.
- Web Applications: Python is commonly used in web development on the backend with frameworks like Django and Flask and on the front end with tools like JavaScript HTML and CSS.
- Enterprise Applications: Python can be used to develop large-scale enterprise applications with features such as distributed computing, networking, and parallel processing.
- 3D CAD Applications: Python can be used for 3D computer-aided design (CAD) applications through libraries such as Blender.
- Machine Learning: Python is widely used for machine learning due to its simplicity, ease of use, and availability of powerful machine learning libraries.
- Computer Vision or Image Processing Applications: Python can be used for computer vision and image processing applications through powerful libraries such as OpenCV and Scikit-image.
- Speech Recognition: Python can be used for speech recognition applications through libraries such as SpeechRecognition and PyAudio.
- Scientific computing: Libraries like NumPy, SciPy, and Pandas provide advanced numerical computing capabilities for tasks like data analysis, machine learning, and more.
- Education: Python's easy-to-learn syntax and availability of many resources make it an ideal language for teaching programming to beginners.
- Testing: Python is used for writing automated tests, providing frameworks like unit tests and pytest that help write test cases and generate reports.
- Gaming: Python has libraries like Pygame, which provide a platform for developing games using Python.
- IoT: Python is used in IoT for developing scripts and applications for devices like Raspberry Pi, Arduino, and others.

- Networking: Python is used in networking for developing scripts and applications for network automation, monitoring, and management.
- DevOps: Python is widely used in DevOps for automation and scripting of infrastructure management, configuration management, and deployment processes.
- Finance: Python has libraries like Pandas, Scikit-learn, and Stats models for financial modeling and analysis.
- Audio and Music: Python has libraries like Pyaudio, which is used for audio processing, synthesis, and analysis, and Music21, which is used for music analysis and generation.
- Writing scripts: Python is used for writing utility scripts to automate tasks like file operations, web scraping, and data processing.
- Python Popular Frameworks and Libraries
- Python has wide range of libraries and frameworks widely used in various fields such as machine learning, artificial intelligence, web applications, etc.

We define some popular frameworks and libraries of Python as follows.

- Web development (Server-side) - Django Flask, Pyramid, CherryPy
- GUIs based applications - Tkinter, PyGTK, PyQt, PyJs, etc.
- Machine Learning - TensorFlow, PyTorch, Scikit-learn, Matplotlib, Scipy, etc.
- Mathematics - NumPy, Pandas, etc.
- BeautifulSoup: a library for web scraping and parsing HTML and XML
- Requests: a library for making HTTP requests
- SQLAlchemy: a library for working with SQL databases
- Kivy: a framework for building multi-touch applications
- Pygame: a library for game development
- Pytest: a testing framework for Python Django
- REST framework: a toolkit for building RESTful APIs
- FastAPI: a modern, fast web framework for building APIs

- Streamlit: a library for building interactive web apps for machine learning and data science
- NLTK: a library for natural language processing

Python print() Function

Python print() function is used to display output to the console or terminal. It allows us to display text, variables and other data in a human readable format.

Syntax:

```
print(object(s), sep=separator, end=end, file=file, flush=flush)
```

It takes one or more arguments separated by comma(,) and adds a 'newline' at the end by default.

Parameters:

object(s) - As many as you want data to display, will first converted into string and printed to the console.

sep - Separates the objects by a separator passed, default value = " ".

end - Ends a line with a newline character

file - a file object with write method, default value = sys.stdout

Example:

```
# Displaying a string

print("Hello, World!")


# Displaying multiple values

name = "Aman"

age = 21

print("Name:", name, "Age:", age)
```

```
  # Printing variables and literals
x = 5
y = 7
print("x =", x, "y =", y, "Sum =", x + y)
  # Printing with formatting
percentage = 85.75
print("Score: {:.2f}%".format(percentage))
```

Output:

```
Hello, World!
Name: Aman Age: 21
X = 5 y = 7 Sum = 12
Score: 85.75%
```

In this example, the print statement is used to print string, integer, and float values in a human readable format.

The print statement can be used for debugging, logging and to provide information to the user.

Python Data Types

Every value has a datatype, and variables can hold values. Python is a powerfully composed language; consequently, we don't have to characterize the sort of variable while announcing it. The interpreter binds the value implicitly to its type.

```
a = 5
```

We did not specify the type of the variable a, which has the value five from an integer. The Python interpreter will automatically interpret the variable as an integer.

We can verify the type of the program-used variable thanks to Python. The type() function in Python returns the type of the passed variable.

Consider the following illustration when defining and verifying the values of various data types.

```
a=10

b="Hi Python"

c = 10.5

print(type(a))

print(type(b))

print(type(c))
```

Output:

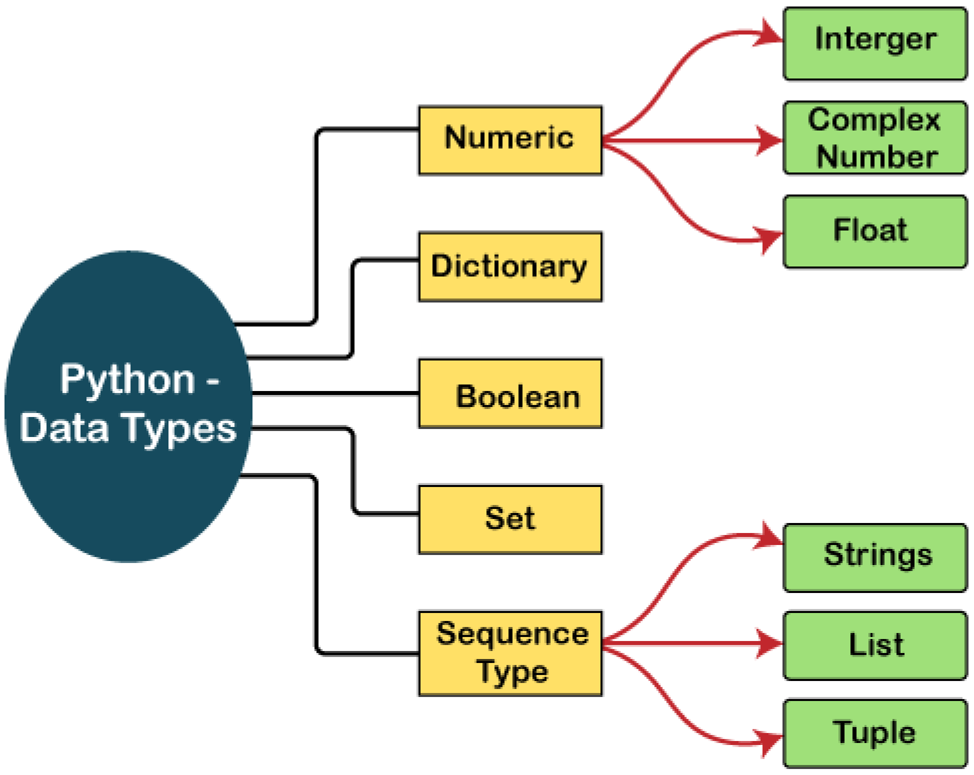
```
<type 'int'>

<type 'str'>

<type 'float'>
```

Standard data types

A variable can contain a variety of values. On the other hand, a person's id must be stored as an integer, while their name must be stored as a string.

The storage method for each of the standard data types that Python provides is specified by Python. The following is a list of the Python-defined data types.

- Numbers
- Sequence Type
- Boolean
- Set
- Dictionary



The data types will be briefly discussed in this tutorial section. We will talk about every single one of them exhaustively later in this instructional exercise.

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,

1. a = 5
2. print("The type of a", type(a))
3. b = 40.5
4. print("The type of b", type(b))

5. c = 1+3j
6. print("The type of c", type(c))
7. 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.

```
str = "string using double quotes"

print(str)

s = """A multiline

string"'

print(s)
```

Output:

```
string using double quotes

A multiline
```

String

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 (*).

Example:

```
list1  = [1, "hi", "Python", 2]
```

```
#Checking type of given list

print(type(list1))

#Printing the list1

print (list1)

# List slicing

print (list1[3:])

# List slicing

print (list1[0:2])

# List Concatenation using + operator

print (list1 + list1)

# List repetation using * operator

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:

```
tup  = ("hi", "Python", 2)

# Checking type of tup

print (type(tup))

#Printing the tuple

print (tup)

 # Tuple slicing

print (tup[1:])

print (tup[0:1])

# Tuple concatenation using + operator

print (tup + tup)

# Tuple repatation using * operator

print (tup * 3)
```

```
# Adding value to tup. It will throw an error.

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.

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}
```

```
# Printing dictionary

print (d)

# Accesing value using keys

print("1st name is "+d[1])

print("2nd name is "+ d[4])

print (d.keys())

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.

```
# Python program to check the boolean type

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

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.

```
# creating Empty set

set1 = set()

set2 = {'James', 2, 3,'Python'}

#Printing Set value

print(set2)

# Adding element to the set

set2.add(10)
```

```
print(set2)

#Removing element from the set

set2.remove(2)

print(set2)
```

Output:

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

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

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

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:

```
a = 32    # Initialize the value of a
b = 6      # Initialize the value of b
print('Addition of two numbers:',a+b)
print('Subtraction of two numbers:',a-b)
print('Multiplication of two numbers:',a*b)
print('Division of two numbers:',a/b)
print('Reminder of two numbers:',a%b)
print('Exponent of two numbers:',a**b)
print('Floor division of two numbers:',a//b)
```

Output –

```
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.

```
a = 32      # Initialize the value of a
b = 6       # Initialize the value of b
print('Two numbers are equal or not:',a==b)
print('Two numbers are not equal or not:',a!=b)
print('a is less than or equal to b:',a<=b)
print('a is greater than or equal to b:',a>=b)
print('a is greater b:',a>b)
print('a is less than b:',a<b)
```

Output:
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

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.

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.

Program Code:

```
a = 32        # 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*=b:', a*b)
print('a%=b:', a%b)
print('a**=b:', a**b)
print('a//=b:', a//b)
```

Output –

```
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,

```
if a = 7
b = 6
then, binary (a) = 0111
binary (b) = 0110
hence, a & b = 0011
a | b = 0111
a ^ b = 0100
~ a = 1000
Let, Binary of x = 0101
Binary of y = 1000
Bitwise OR = 1101
8 4 2 1
1 1 0 1 = 8 + 4 + 1 = 13
Bitwise AND = 0000
0000 = 0
Bitwise XOR = 1101
8 4 2 1
1 1 0 1 = 8 + 4 + 1 = 13
Negation of x = ~x = (-x) - 1 = (-5) - 1 = -6
~x = -6
```

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)	

```
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 –

```
a&b: 4

a|b: 7

a^b: 3

~a: -6

a<>b: 0
```

Logical Operators

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.

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.

```
a = 5        # initialize the value of a

print(Is this statement true?:',a > 3 and a < 5)

print('Any one statement is true?:',a > 3 or a < 5)

print('Each statement is true then return False and vice-versa:',(not(a > 3 and a < 5)))
```

Output -

```
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).

```
x = ["Rose", "Lotus"]

print(' Is value Present?', "Rose" in x)

print(' Is value not Present?', "Riya" not in x)
```

Output:

```
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.

```
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:

```
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 Conditional Statements

Conditional statements help us to execute a particular block for a particular condition. In this tutorial, we will learn how to use conditional expression to execute a different block of statements. Python provides if and else keywords to set up logical conditions. The elif keyword is also used as a conditional statement.

Example code for if..else statement

```
x = 10

y = 5

if x > y:
    print("x is greater than y")
else:
    print("y is greater than or equal to x")
```

Output:

```
x is greater than y
```

In the above code, we have two variables, x, and y, with 10 and 5, respectively. Then we used an if..else statement to check if x is greater than y or vice versa. If the first condition is true, the statement "x is greater than y" is printed. If the first condition is false, the statement "y is greater than or equal to x" is printed instead.

The if keyword checks the condition is true and executes the code block inside it. The code inside the else block is executed if the condition is false. This way, the if..else statement helps us to execute different blocks of code based on a condition.

Python Loops

Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several times. For this purpose, the programming languages provide various loops capable of repeating some specific code several times. Consider the following tutorial to understand the statements in detail.

Python For Loop

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x, end=" ")
```

Output:

```
apple banana cherry
```

Python While Loop

```
i = 1
while i<5:
    print(i, end=" ")
    i += 1
```

Output:

```
1 2 3 4
```

In the above example code, we have demonstrated using two types of loops in Python - For loop and While loop.

The For loop is used to iterate over a sequence of items, such as a list, tuple, or string. In the example, we defined a list of fruits and used a for loop to print each fruit, but it can also be used to print a range of numbers.

The While loop repeats a code block if the specified condition is true. In the example, we have initialized a variable i to 1 and used a while loop to print the value of i until it becomes greater than or equal to 6. The i += 1 statement is used to increment the value of i in each iteration.

Python Data Structures

Python offers four built-in data structures: lists, tuples, sets, and dictionaries that allow us to store data in an efficient way. Below are the commonly used data structures in Python, along with example code:

1. Lists

Lists are ordered collections of data elements of different data types.

Lists are mutable meaning a list can be modified anytime.

Elements can be accessed using indices.

They are defined using square bracket '[]'.

Example:

```
# Create a list
fruits = ['apple', 'banana', 'cherry']
print("fuirts[1] =", fruits[1])
  # Modify list
fruits.append('orange')
print("fruits =", fruits)
num_list = [1, 2, 3, 4, 5]
# Calculate sum
sum_nums = sum(num_list)
print("sum_nums =", sum_nums)
```

Output:

```
fuirts[1] = banana
fruits = ['apple', 'banana', 'cherry', 'orange']
sum_nums = 15
```

2. Tuples

Tuples are also ordered collections of data elements of different data types, similar to Lists.

Elements can be accessed using indices.

Tuples are immutable meaning Tuples can't be modified once created.

They are defined using open bracket '()'.

Example:

```
# Create a tuple
```

```
point = (3, 4)
x, y = point
print("(x, y) =", x, y)
  # Create another tuple
tuple_ = ('apple', 'banana', 'cherry', 'orange')
print("Tuple =", tuple_)
```

Output:

```
(x, y) = 3 4
Tuple = ('apple', 'banana', 'cherry', 'orange')
```

3. Sets

Sets are unordered collections of immutable data elements of different data types.

Sets are mutable.

Elements can't be accessed using indices.

Sets do not contain duplicate elements.

They are defined using curly braces '{}'

Example:

```
# Create a set
set1 = {1, 2, 2, 1, 3, 4}
print("set1 =", set1)
  # Create another set
set2 = {'apple', 'banana', 'cherry', 'apple', 'orange'}
print("set2 =", set2)
```

Output:

```
set1 = {1, 2, 3, 4}

set2 = {'apple', 'cherry', 'orange', 'banana'}
```

4. Dictionaries

Dictionary are key-value pairs that allow you to associate values with unique keys.

They are defined using curly braces '{}' with key-value pairs separated by colons ':'.

Dictionaries are mutable.

Elements can be accessed using keys.

Example:

```
# Create a dictionary

person = {'name': 'Umesh', 'age': 25, 'city': 'Noida'}

print("person =", person)

print(person['name'])

  # Modify Dictionary

person['age'] = 27

print("person =", person)
```

Output:

```
person = {'name': 'Umesh', 'age': 25, 'city': 'Noida'}

Umesh

person = {'name': 'Umesh', 'age': 27, 'city': 'Noida'}
```

These are just a few examples of Python's built-in data structures. Each data structure has its own characteristics and use cases.

Python Functional Programming

This section of the Python tutorial defines some important tools related to functional programming, such as lambda and recursive functions. These functions are very efficient in accomplishing complex tasks. We define a few important functions, such as reduce, map, and filter. Python provides the functools module that includes various functional programming tools. Visit the following tutorial to learn more about functional programming.

Recent versions of Python have introduced features that make functional programming more concise and expressive. For example, the "walrus operator":= allows for inline variable assignment in expressions, which can be useful when working with nested function calls or list comprehensions.

Python Function

- Lambda Function - A lambda function is a small, anonymous function that can take any number of arguments but can only have one expression. Lambda functions are often used in functional programming to create functions "on the fly" without defining a named function.
- Recursive Function - A recursive function is a function that calls itself to solve a problem. Recursive functions are often used in functional programming to perform complex computations or to traverse complex data structures.
- Map Function - The map() function applies a given function to each item of an iterable and returns a new iterable with the results. The input iterable can be a list, tuple, or other.
- Filter Function - The filter() function returns an iterator from an iterable for which the function passed as the first argument returns True. It filters out the items from an iterable that do not meet the given condition.
- Reduce Function - The reduce() function applies a function of two arguments cumulatively to the items of an iterable from left to right to reduce it to a single value.

- functools Module - The functools module in Python provides higher-order functions that operate on other functions, such as partial() and reduce().
- Currying Function - A currying function is a function that takes multiple arguments and returns a sequence of functions that each take a single argument.
- Memoization Function - Memoization is a technique used in functional programming to cache the results of expensive function calls and return the cached Result when the same inputs occur again.
- Threading Function - Threading is a technique used in functional programming to run multiple tasks simultaneously to make the code more efficient and faster.

Python Modules

Python modules are the program files that contain Python code or functions. Python has two types of modules - User-defined modules and built-in modules. A module the user defines, or our Python code saved with .py extension, is treated as a user-define module.

Built-in modules are predefined modules of Python. To use the functionality of the modules, we need to import them into our current working program.

Python modules are essential to the language's ecosystem since they offer reusable code and functionality that can be imported into any Python program. Here are a few examples of several Python modules, along with a brief description of each:

1. Math: Gives users access to mathematical constants and pi and trigonometric functions.
2. Datetime: Provides classes for a simpler way of manipulating dates, times, and periods.
3. OS: Enables interaction with the base operating system, including administration of processes and file system activities.
4. Random: The random function offers tools for generating random integers and picking random items from a list.

5. JSON: JSON is a data structure that can be encoded and decoded and is frequently used in online APIs and data exchange. This module allows dealing with JSON.
6. Re: Supports regular expressions, a potent text-search and text-manipulation tool.
7. Collections: Provides alternative data structures such as sorted dictionaries, default dictionaries, and named tuples.
8. NumPy: NumPy is a core toolkit for scientific computing that supports numerical operations on arrays and matrices.
9. Pandas: It provides high-level data structures and operations for dealing with time series and other structured data types.
10. Requests: Offers a simple user interface for web APIs and performs HTTP requests.