UNIT 1

BASICS OF PYTHON

WHAT IS PYTHON?

- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. It combines the features of C and Java.
- Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together.
- Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance.
- Python supports modules and packages, which encourages program modularity and code reuse.
- The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.
- Python began in late 1989 by Guido van Rossum, then at CWI (Centrum voor Wiskunde en Informatica, the National Research Institute for Mathematics and Computer Science) in the Netherlands. It was eventually released for public distribution in early 1991.

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

WHY PYTHON?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

FEATURES OF PYTHON

1. Free and Open Source

- Python language is freely available at the official website and you can download it. Download Python Since it is open-source, this means that source code is also available to the public. So you can download it, use it as well as share it.

2. Easy to code

- Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, Javascript, Java, etc. It is very easy to code in the Python language and anybody can learn Python basics in a few hours or days. It is also a developer-friendly language.

3. Easy to Read

- As you will see, learning Python is quite simple. As was already established, Python's syntax is really straightforward. The code block is defined by the indentations rather than by semicolons or brackets.

4. Object-Oriented Language

- One of the key features of Python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, object encapsulation, etc.

5. GUI Programming Support

- Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in Python. PyQt5 is the most popular option for creating graphical apps with Python.

6. High-Level Language

- Python is a high-level language. When we write programs in Python, we do not need to remember the system architecture, nor do we need to manage the memory.

7. Large Community Support

- Python has gained popularity over the years. Many questions are constantly answered by the enormous StackOverflow community. These websites have already provided answers to many questions about Python, so Python users can consult them as needed.

8. Easy to Debug

- Excellent information for mistake tracing. You will be able to quickly identify and correct the majority of your program's issues once you understand how to interpret Python's error traces. Simply by glancing at the code, you can determine what it is designed to perform.

9. Python is a Portable language

- Python language is also a portable language. For example, if we have Python code for Windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.

10. Python is an Integrated language

- Python is also an Integrated language because we can easily integrate Python with other languages like C, C++, etc.

11. Interpreted Language:

- Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile Python code this makes it easier to debug our code. The source code of Python is converted into an immediate form called **bytecode**.

12. Large Standard Library

- Python has a large standard library that provides a rich set of modules and functions so you do not have to write your own code for every single thing. There are many libraries present in Python such as regular expressions, unit-testing, web browsers, etc.

13. Dynamically Typed Language

- Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.

14. Frontend and backend development

- With a new project py script, you can run and write Python codes in HTML with the help of some simple tags <py-script>, <py-env>, etc. This will help you do frontend development work in Python like javascript. Backend is the strong forte of Python it's extensively used for this work cause of its frameworks like Django and Flask.

15. Allocating Memory Dynamically

- In Python, the variable data type does not need to be specified. The memory is automatically allocated to a variable at runtime when it is given a value. Developers do not need to write int y = 18 if the integer value 15 is set to y. You may just type y=18.

WHAT IS IDLE AND ITS BASICS

- IDLE stands for Integrated Development and Learning Environment.
- Every Python installation comes with an **Integrated Development and Learning Environment**, which you'll see shortened to IDLE or even IDE.
- These are a class of applications that help you write code more efficiently.
- While there are many IDEs for you to choose from, Python IDLE is very bare-bones, which makes it the perfect tool for a beginning programmer.
- Python IDLE comes included in Python installations on Windows and Mac.
- IDLE provides
  - a text editor with syntax highlighting, auto completion, and smart indentation
  - a shell with syntax highlighting
  - an integrated debugger

- There are two ways to run a program using the Python interpreter: a) Interactive mode b) Script mode
- (A) Interactive Mode :: In the interactive mode, we can type a Python statement on the >>> prompt directly. As soon as we press enter, the interpreter executes the statement and displays the result(s) Working in the interactive mode is convenient for testing a single line code for instant execution. But in the interactive mode, we cannot save the statements for future use and we have to retype the statements to run them again.
- (B) Script Mode :: In the script mode, we can write a Python program in a file, save it and then use the interpreter to execute the program from the file. Such program files have a .py extension and they are also known as scripts. Usually, beginners learn Python in interactive mode, but for programs having more than a few lines, we should always save the code in files for future use. Python scripts can be created using any editor. Python has a built-in editor called IDLE which can be used to create programs.

PYTHON IDENTIFIERS

- A Python identifier is a name used to identify a variable, function, class, module or other object.
- An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
- Python does not allow punctuation characters such as @, $ and % within identifiers.
- Python is a case sensitive programming language. Thus, Manpower and manpower are two different identifiers in Python.
- Following are the identifier naming convention for Python:
  - Class names start with an uppercase letter and all other identifiers with a lowercase letter.
  - Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private.
  - Starting an identifier with two leading underscores indicates a strongly private identifier.
  - If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

PYTHON RESERVED WORD

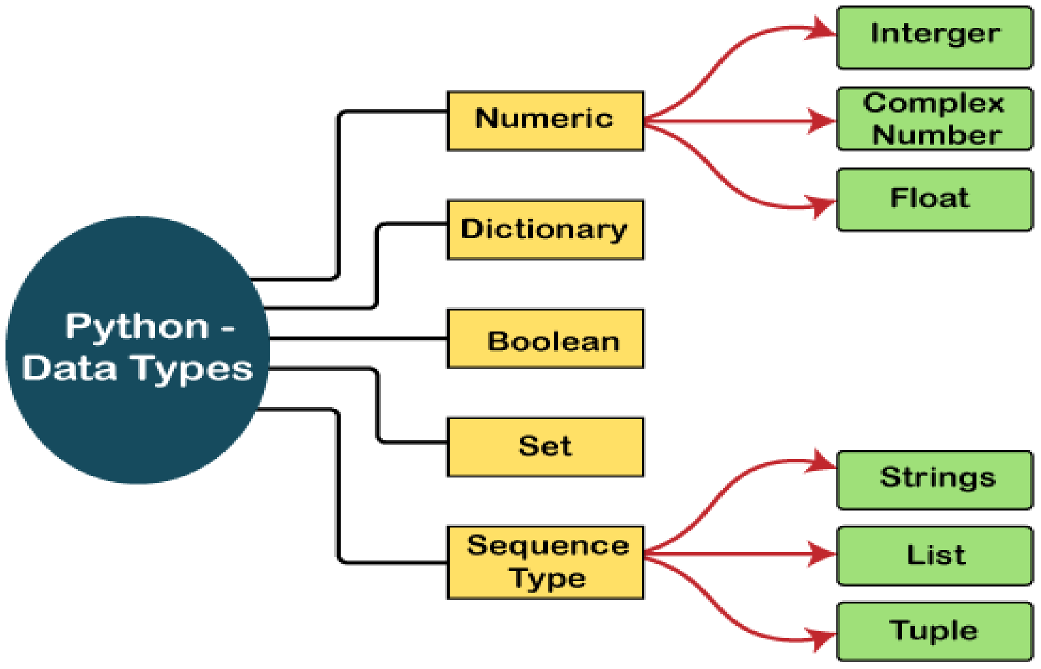
- The following list shows the reserved words in Python.
- These reserved words may not be used as constant or variable or any other identifier names.
- All the Python keywords contain lowercase letters only :
  - And
  - exec
  - not
  - assert
  - finally
  - or
  - break
  - for
  - pass
  - class
  - from

- print
- continue
- global
- raise
- def
- if
- return
- del
- import
- try

- elif
- in
- while
- else
- is
- with
- except
- lambda
- yield

PYTHON BUILT-IN DATA TYPES

- A data type represents the type of data stored into a variable or memory. The data types which are already available in Python language are called Built-in datatypes.
- The built in datatypes are of five types:
  - None type
  - Numeric type
  - Sequence
  - Sets
  - Mappings



THE NONE TYPE

- In python, the 'None' datatype represents an object that does not contain any value.
- Maximum of only one 'none' object is provided in Python.
- One of the use of 'None' is that it is used inside a function as a default value of the arguments. When calling the function, if no value is passed, then the default value will be taken as 'None'.

NUMERIC TYPES

- The numeric types represent numbers. There are three sub types:
  - int
  - float
  - complex
- int Datatype::
  - The int represents an integer number. An integer number is a number without any decimal point or fraction part. For example, 200, -50, 0, 9888758 etc.
  - Eg. A=57
- float Datatype ::
  - The flaot datatype represents floating point numbers. A floating point number is a number that contains a decimal point. For example, 0.5, -3.547, 290.08 etc.
  - Eg. Num = 55.6789
  - Eg. X=22.55e3 (e represents exponentiation)

NUMERIC TYPES

- Complex Datatype ::
  - A complex number is a number that is written in the form of a+bj or a + bJ.
  - Here ‘a’ represents the real part of the number and ‘b’ represents the imaginary part of the number. The suffix ‘j’ or ‘J’ after ‘b’ indicates the square root values of -1.
  - For example 3+5j, -1-5.5J etc.
  - For example :: c1=-1-5.5J
- Representing Binary, Octal and Hexadecimal numbers
  - Octal -> 0o17
  - Binary -> 0B1110010
  - Hexadecimal -> 0X1c2

SEQUENCES IN PYTHON

- Generally, a sequence represents a group of elements or items. For example, a group of integer numbers will form a sequence.
- There are six types of sequences in Python:
  - String
  - Bytes
  - Bytearray
  - List
  - Tuple
  - Range

STRING DATATYPE

- A string is represented by a group of characters.
- Strings are enclosed in single quotes or double quotes. Both are valid.
- For example :: str1=‘Hello World’

  str2 = “Hello World”
- We can also write string inside ‘’“‘’ (Triple single or double quotes) to span a group of lines including spaces.
- For Example :: str3=‘“ Hello

  My first Python Trial

  to program”’
- The triple quotes are useful to embed da string inside another string as shown below :

  Str4 = ‘“This is ‘core python’ book example”’

STRING DATATYPE

- Python allows for either pairs of single or double quotes.
- Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.
- For example: str = 'Hello World!'
  - print (str) # Prints complete string
  - print (str[0]) # Prints first character of the string
  - print (str[2:5]) # Prints characters starting from 3rd to 5th
  - print (str[2:]) # Prints string starting from 3rd character
  - print (str * 2) # Prints string two times
  - print (str + "TEST") # Prints concatenated strin

LIST DATATYPE

- Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]).
- To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.
- The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.
- The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator

- For example:
- list1 = ['abcd', 786 , 2.23, 'john', 70.2]
- tinylist = [123, 'john']
  - print (list1) # Prints complete list
  - print (list1[0]) # Prints first element of the list
  - print (list1[1:3]) # Prints elements starting from 2nd till 3rd
  - print (list1[2:]) # Prints elements starting from 3rd element
  - print (tinylist * 2) # Prints list two times
  - print (list1 + tinylist) # Prints concatenated list

TUPLE

- A tuple is another sequence data type that is similar to the list.
- A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
- The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated.
- Tuples can be thought of as read-only lists.
- For example:
- tup1 = ('abcd', 786 , 2.23, 'john', 70.2)
- tinytuple = (123, 'john')
  - print (tup1) # Prints complete list
  - print (tup1[0]) # Prints first element of the list
  - print (tup1[1:3]) # Prints elements starting from 2nd till 3rd
  - print (tup1[2:]) # Prints elements starting from 3rd element
  - print (tinytuple * 2) # Prints list two times
  - print (tup1 + tinytuple) # Prints concatenated list

DICTIONARY

- Python's dictionaries are kind of hash table type.
- They work like associative arrays or hashes found in Perl and consist of key-value pairs.
- A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.
- Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).
- Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered

- For example:
- dict = {} #empty dictionary
- dict['one'] = "This is one"
- dict[2] = "This is two"
- tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
- print (dict['one']) # Prints value for 'one' key
- print (dict[2]) # Prints value for 2 key
- print (tinydict) # Prints complete dictionary
- print (tinydict.keys()) # Prints all the keys
- print (tinydict.values()) # Prints all the values

SET

- In Python, Set is an unordered collection of data type that is iterable, mutable and has no duplicate elements.
- The order of elements in a set is undefined though it may consist of various elements.
- Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by ‘comma’.
- Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.
- For Example :
  - set1 = set() #blank set
  - set2 = set("PythonForKing") #Use of string
  - set3 = set(["Python", "For", "King"]) #with list
  - set4 = set([1, 2, 'Python', 4, 'For', 6, 'King']) #mixed type of values

ACCESSING ELEMENTS OF SETS

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

```
set1 = set(["Goal", "For", "Life"])
print(set1)
print("\nElements of set: ")
for i in set1:
    print(i, end =" ")
print("Goal" in set1)
```

INDENTATION AND BLOCK STRUCTURING

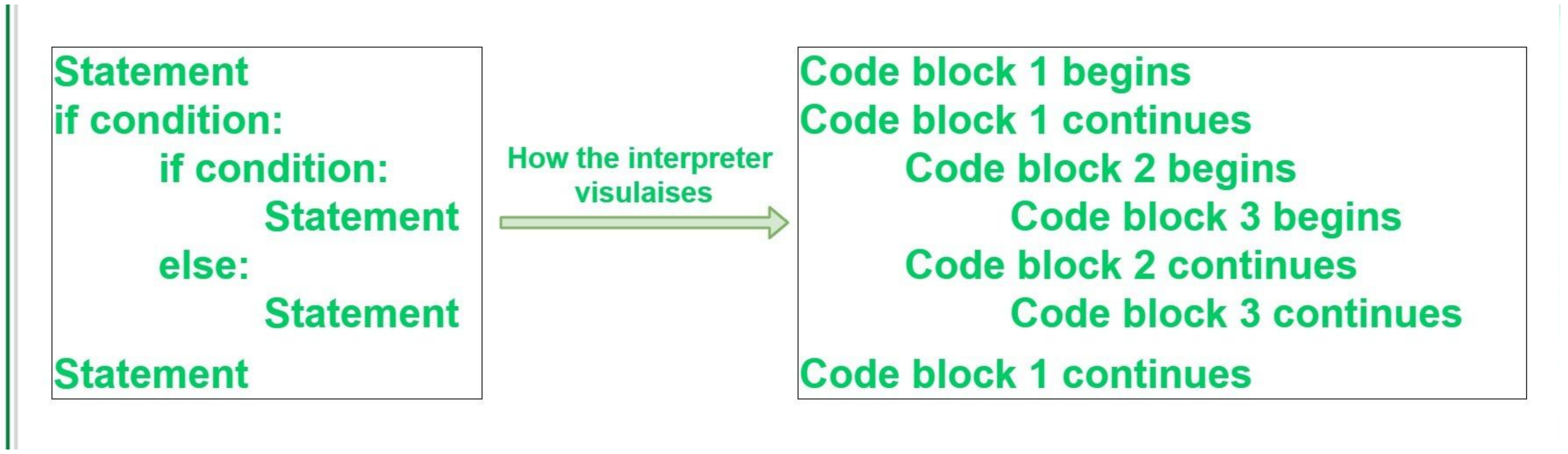
- One of the first caveats programmers encounter when learning Python is the fact that there are no braces ({ }) to indicate blocks of code for class and function definitions or flow control.
- Blocks of code are denoted by line indentation, which is rigidly enforced.
- Python uses indentation to express the block structure of a program. Unlike other languages, Python does not use braces or begin/end delimiters around blocks of statements: indentation is the only way to indicate such blocks.
- Each logical line in a Python program is indented by the whitespace on its left.
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Both blocks in this example are fine:

```
if True:
        print ("True")
else:
        print ("False")
```

- The second block in following example will generate an error:

```
if True:
    print ("Answer")
    print ("True")
else:
        print ("Answer")
      print ("False")
```

- Thus, in Python all the continuous lines indented with similar number of spaces would form a block.



```
Statement
if condition:
        if condition:
                Statement
        else:
                Statement
Statement
```

```
Code block 1 begins
Code block 1 continues
        Code block 2 begins
                Code block 3 begins
        Code block 2 continues
                Code block 3 continues
Code block 1 continues
```

COMMENTS

- A hash sign (#) that is not inside a string literal begins a comment.
- All characters after the # and up to the physical line end are part of the comment and the Python interpreter ignores them.
- #!/usr/bin/python
- # First comment
- print "Hello, Python!"; # second comment

VARIABLES AND ASSIGNMENTS

- Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.
- Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory.
- Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.
- Python variables do not have to be explicitly declared to reserve memory space. The declaration happens automatically when you assign a value to a variable.
- The equal sign (=) is used to assign values to variables.
- The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.
- For example:
  - counter = 100 # An integer assignment
  - miles = 1000.0 # A floating point
  - name = "John" # A string

- In Python, variable names can contain uppercase and lowercase letters, digits (but they cannot start with a digit), and the special character _.
- Python variable names are case-sensitive e.g., Var1 and var1 are different names.
- There are a small number of reserved words (sometimes called keywords) in Python that have built-in meanings and cannot be used as variable names.
- Python allows multiple assignment. For Example the statement :

```
x, y = 2, 3
```

binds x to 2 and y to 3.

- Python also allows you to assign a single value to several variables simultaneously. For example:

```
a = b = c = 1
```

GETTING INPUT FROM USER

- Developers often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input.
- **input ()** function first takes the input from the user and converts it into a string. The type of the returned object always will be <class 'str'>.
- It does not evaluate the expression it just returns the complete statement as String.
- For example, Python provides a built-in function called input which takes the input from the user.
- When the input function is called it stops the program and waits for the user's input.
- When the user presses enter, the program resumes and returns what the user typed.
- For Example : name = input('What is your name?\n')

- **How the input function works in Python :**
  - When input() function executes program flow will be stopped until the user has given input.
  - The text or message displayed on the output screen to ask a user to enter an input value is optional i.e. the prompt, which will be printed on the screen is optional.
  - Whatever you enter as input, the input function converts it into a string. if you enter an integer value still input() function converts it into a string.
  - You need to explicitly convert it into an integer in your code using typecasting.
  - For Example ::
    - num = int(input ("Enter number :"))
    - print(num)
    - name1 = input("Enter name : ")
    - print(name1)

    - # Printing type of input value
    - print ("type of number", type(num))
    - print ("type of name", type(name1))

BUILT-IN OPERATORS

- Operators are used to perform operations on variables and values.
- Python language supports the following types of operators.
  - Arithmetic Operators
  - Comparison (i.e., Relational) Operators
  - Assignment Operators
  - Logical Operators
  - Bitwise Operators
  - Membership Operators
  - Identity Operators

ARITHMETIC OPERATORS

- Python Arithmetic operators are used to perform basic mathematical operations like **addition, subtraction, multiplication**, and **division**.

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

COMPARISON OPERATOR

- 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

ASSIGNMENT OPERATOR

- Python Assignment_operators are used to assign values to the variables.

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 operand	a%=b a=a%b
//=	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

LOGICAL OPERATORS

- 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
not	Logical NOT: True if the operand is false	not x

BITWISE OPERATORS

- 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
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

MEMBERSHIP OPERATORS

- 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

- Example :

```
x = 20
list1 = [10, 20, 30, 40, 50]

if (x in list1):
    print("x is NOT present in given list")
else:
    print("x is present in given list")
```

IDENTITY OPERATORS

- Identity operators compare the memory locations of two objects. There are two Identity operators explained below:

Operator	**Description**	**Example**
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

PYTHON DECISION MAKING STATEMENTS

- Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.
- Python control flow statements are as follows:
  1. The if statement
  2. The if-else statement
  3. The nested-if statement
  4. The if-elif-else ladder

THE IF STATEMENT

- The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not.
- **Syntax:**

```
if condition:
    # Statements to execute if condition is true
```

- Here, the condition after evaluation will be either true or false. if the statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not.
- As we know, python uses indentation to identify a block. So the block under an if statement will be identified as shown in the below example:

```
if condition:
    statement1
statement2
# Here if the condition is true, if block  will consider only statement1 to be inside its block.
```

IF-ELSE STATEMENT

- The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't.
- But if we want to do something else if the condition is false, we can use the else statement with the if statement to execute a block of code when the if condition is false.
- **Syntax of Python If-Else:**

```
if (condition):
    # Executes this block if condition is true
else:
    # Executes this block if condition is false
```

NESTED-IF STATEMENT

- A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement.
- Yes, Python allows us to nest if statements within if statements. i.e., we can place an if statement inside another if statement.
- **Syntax:**

```
if (condition1):
        # Executes when condition1 is true
        if (condition2):
                # Executes when condition2 is true
        # if Block is end here
# if Block is end here
```

IF-ELIF-ELSE LADDER

- Here, a user can decide among multiple options. The if statements are executed from the top down.
- As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final "else" statement will be executed.
- **Syntax:**

```
if (condition):
      statement
elif (condition):
      statement
.
.
else:
      statement
```

SHORT HAND IF AND IF...ELSE

- If you have only one statement to execute, you can put it on the same line as the if statement.
- Example
- One line if statement:

```
if a > b: print("a is greater than b")
```

- If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:
- Example
- One line if else statement:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

LOOPS

- There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times.
- Python programming language provides two types of loops – **For loop** and **While loop** to handle looping requirements.

WHILE LOOP

- In Python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes false, the line immediately after the loop in the program is executed.
- **While Loop Syntax:**

```
while expression:
    statement(s)
```

- The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.
- All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.
- **Example:**

```
count = 0
while (count < 3):
        count = count + 1
        print("Hello Geek")
```

- A loop becomes infinite loop if a condition never becomes false. You must use caution when using while loops because of the possibility that this condition never resolves to a false value. This results in a loop that never ends. Such a loop is called an infinite loop.
- An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.
- For Example:

```
var = 1
while var == 1 : # This constructs an infinite loop
    num = input("Enter a number :")
    print ("You entered: ", num)
print "Good bye!"
```

- Above example will go in an infinite loop and you would need to use CTRL+C to come out of the program.

THE ELSE STATEMENT USED WITH LOOPS

- Python supports to have an else statement associated with a loop statement.
- If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.
- If the else statement is used with a while loop, the else statement is executed when the condition becomes false.
- The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

```
count = 0
while count < 5:
        print (count, " is less than 5")
        count = count + 1
else:
        print (count, " is not less than 5")
```

FOR LOOP

- The for loop in Python has the ability to iterate over the items of any sequence, such as a list or a string.
- Syntax:

```
        for iterating_var in sequence:
                statements(s)
```

- If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable iterating_var. Next, the statements block is executed.
- Each item in the list is assigned to iterating_var, and the statement(s) block is executed until the entire sequence is exhausted.

- Example:

```
for letter in 'Python':   # First Example
        print ('Current Letter :', letter)

fruits = ['banana', 'apple', 'mango']
for fruit in fruits:     # Second Example
        print ('Current fruit :', fruit)
print "Good bye!"
```

THE RANGE() FUNCTION

- To loop through a set of code a specified number of times, we can use the range() function,
- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6):
  print(x)
```

  - Note that range(6) is not the values of 0 to 6, but the values 0 to 5.
- The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

```
for x in range(2, 6):
  print(x)
```

- The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, **3**):

```
for x in range(2, 30, 3):
  print(x)
```

ELSE IN FOR LOOP

- The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

-
  ```
  for x in range(6):
          print(x)
  else:
          print("Finally finished!")
  ```
- Note: The else block will NOT be executed if the loop is stopped by a break statement.

NESTED LOOP

- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop":

-
  ```
  adj = ["red", "big", "tasty"]
  fruits = ["apple", "banana", "cherry"]

  for x in adj:
    for y in fruits:
      print(x, y)
  ```

LOOP CONTROL STATEMENT

- Loop control statements change execution from its normal sequence.
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Python supports the following control statements.
- break statement -> Terminates the loop statement and transfers execution to the statement immediately following the loop.
- continue statement -> Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
- pass statement -> The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

BREAK STATEMENT

- The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.
- The most common use for break is when some external condition is triggered requiring a hasty exit from a loop.
- The break statement can be used in both while and for loops.
- If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.
- Syntax:

```
break
```

- Example:

```
for letter in 'Python': # First Example
        if letter == 'h':
                break
        print ('Current Letter :', letter)

var = 10 # Second Example
while var > 0:
        print ('Current variable value :', var)
        var = var -1
        if var == 5:
                break
print ("Good bye!")
```

CONTINUE STATEMENT

- The continue statement in Python returns the control to the beginning of the while loop.
- The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.
- The continue statement can be used in both while and for loops.
- Syntax:

```
continue
```

- Example:

```
for letter in 'Python':
    if letter == 'h':
        continue
    print ('Current Letter :', letter)
```

PASS STATEMENT

- The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.
- The pass statement is a null operation; nothing happens when it executes.
- The pass is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

```
for letter in 'Python':
    if letter == 'h':
        pass
        print ('This is pass block')
    print ('Current Letter :', letter)
print ("Good bye!")
```

STRINGS

- Strings in python are surrounded by either single quotation marks, or double quotation marks.
- To get the length of a string, use the len() function.

```
a = "Hello, World!"
print(len(a))
```

- To check if a certain phrase or character is present or not in a string, we can use the keyword in or not in.

```
txt = "The best things in life are free!"
print("free" in txt)
```

SLICING A STRING

- You can return a range of characters by using the slice syntax.
- Specify the start index and the end index, separated by a colon, to return a part of the string.

```
b = "Hello, World!"
print(b[2:5])
```

- We can use negative indexes also to start the slice from the end of the string.

MODIFYING STRING

- The upper() method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

- The lower() method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

- The strip() method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!“
```

- The replace() method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

- The split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

LIST

- A list is a sequence of values (called elements) that can be any type.
- A list is a sequential collection of values, it is a data structure
- Each value has a location (an index). Indexes range from 0 to n-1 (where n is the length of the list) and from -1 to –n
- Lists are heterogeneous means values can be of any type (strings are homogeneous because their elements are characters)
- Here are some examples of lists •
  - [2,4,6,8,10]
  - [‘a’,’b’,’c’,’d’,’e’]
  - [‘hello’,’there’,’Bob’]
  - [‘bob’,23.0,145,[1,2]]
  - nums = [3,2,5,4.5,3.0,1000] •
  - names = [‘Bob’,’Sally’,’Tom’,’Harry] •
  - empty = []   #This is the empty list

LISTS ARE ORDERED AND MUTABLE

- The index starts at 0 just like with a string and goes up to n-1 if there are n elements in the list
- They also can use negative subscripts, -1 is the last element on the right, -n on the left end
- Unlike strings we can modify the individual elements of a list.

```
numbers =[7,-2,3,4,5,6.0]
numbers[3]=10
print (numbers)
```

- The in operator works here as well to check individual item is there in list or not.

```
3 in numbers
```

Output

```
[7, -2, 3, 10, 5, 6.0]
```

TRAVERSING A LIST

```
for val in numbers:
        print (val)
# Square each number in list
for i in range(len(numbers)):
        numbers[i]=numbers[i]**2
        print (numbers[i])
```

s = [[1,2],3.0,'Harry',[3,5,6,1,2]]

The length of the following list is 4.

Of course the length of s[3] is 5

OPERATIONS ON LIST

- + works like it does on strings, i.e. it concatenates

  [1,2,3]+ [4,5,6] becomes [1,2,3,4,5,6]
- and [1,2,3]*3 becomes [1,2,3,1,2,3,1,2,3]
- What does 10*[0] give you?

LIST SLICE

- t = [11,21,13,44,56,68]

  print t[2,4] # Remember: It doesn't include slot 4!!

  print t[3:]

  t[2:5]=[7,3,1] # You can update multiple elements

  print t

- t = [9, 41, 12, 3, 74, 15]

```
>>> t[1:3]
[41,12]
>>> t[:4]
[9, 41, 12, 3]
>>> t[3:]
[3, 74, 15]
>>> t[:]
[9, 41, 12, 3, 74, 15]
```

LIST METHODS

- Append : adds a new element to the end •

  ```
  t=[2,4,6]
  t.append(8)
  print t
  ```

- extend: adds a list to end of list •

  ```
  t=['a',b',c',d']
  t.extend(['w','x'])
  print t
  ```

- Sort(): sorts the list in place
- reverse()Reverses order of items in list.
- index(x) Returns integer index of first (leftmost) occurrence of x in list.
- count(x)Returns integer count of number of occurrences of x in list.

LIST BUILT-IN FUNCTIONS

- nums = [3, 41, 12, 9, 74, 15]
- >>> print(len(nums))

6

- >>> print(max(nums))

74

- >>> print(min(nums))

3

- >>> print(sum(nums))

154

- >>> print(sum(nums)/len(nums))

25.6

DELETING FROM LIST

t=[6,3,7,8,1,9]

- x=t.pop(3) #remove element in slot 3 and assign it to x
- del t[3] #does the same thing as pop(3) but returns nothing
- t.remove(8) #use this to remove an 8 from the list # it returns nothing
- del t # The del keyword can also delete the list completely.
- t. clear() #The clear() method empties all the items of the list.

TUPLE

- Tuples are used to store multiple items in a single variable.
- Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
- A tuple is a collection which is ordered and **unchangeable**.
- Tuples are written with round brackets.

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.
- To determine how many items a tuple has, use the len() function:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

ACCESS TUPLE ITEMS

- You can access tuple items by referring to the index number, inside square brackets:
- thistuple = ("apple", "banana", "cherry")
  print(thistuple[1]) # PRINT SECOND ITEM
- Negative indexing means start from the end. -1 refers to the last item, -2 refers to the second last item etc.
- We can specify a range of indexes by specifying where to start and where to end the range.
- thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
  print(thistuple[2:5])
- To determine if a specified item is present in a tuple use the in keyword:
-
  ```
  thistuple = ("apple", "banana", "cherry")
  if "apple" in thistuple:
              print("Yes, 'apple' is in the fruits tuple")
  ```

UPDATE TUPLES

- Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.
- But there are some workarounds.
- Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.
- But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.
- Example::
  Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

- We are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:
- Example

  Create a new tuple with the value "orange", and add that tuple:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

REMOVING ITEMS FROM TUPLE

- Tuples are **unchangeable**, so we cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items for removing items also.
- Convert the tuple into a list, remove "apple", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

- The del keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

TUPLE METHODS

Method	Description
▪ count()	Returns the number of times a specified value occurs in a tuple
▪ index()	Searches the tuple for a specified value and returns the position of where it was found

DICTIONARIES IN PYTHON

- **A dictionary in Python** is a data structure that stores the value in value:key pairs.
- In Python, a dictionary can be created by placing a sequence of elements within curly **{}** braces, separated by a ‘comma’.
- The dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value**.
- Values in a dictionary can be of any data type and can be duplicated, whereas keys can’t be repeated and must be *immutable*.
- **Note –** Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.
- The code demonstrates creating dictionaries with different types of keys.
- **Example:**

```
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print(Dict)
```

ACCESSING ELEMENTS OF A DICTIONARY

- To access the items of a dictionary refer to its key name. Key can be used inside square brackets.
- The code demonstrates how to access elements in a dictionary using keys.
- It accesses and prints the values associated with the keys 'name' and 1, showcasing that keys can be of different data types (string and integer).

```
Dict1 = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}
print("Accessing a element using key:")
print(Dict1['name'])
```

- There is also a method called get() that will also help in accessing the element from a dictionary. This method accepts key as argument and returns the value.

```
print(Dict1.get(3))
```

LOOP THROUGH A DICTIONARY

- You can loop through a dictionary by using a for loop.
- When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.
- Print all key names in the dictionary, one by one:

```
for x in thisdict:
    print(x)
```

- Print all *values* in the dictionary, one by one:

```
for x in thisdict:
    print(thisdict[x])
```

- You can also use the values() method to return values of a dictionary:

```
for x in thisdict.values():
    print(x)
```

- You can use the keys() method to return the keys of a dictionary:

```
for x in thisdict.keys():
    print(x)
```

- Loop through both keys and values, by using the items() method:

```
for x, y in thisdict.items():
    print(x, y)
```

DELETING ELEMENTS USING ‘DEL’ KEYWORD

- The items of the dictionary can be deleted by using the del keyword as given below.

```
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}
print("Dictionary =")
print(Dict)
del(Dict[1])
print("Data after deletion Dictionary=")
print(Dict)
```

DICTIONARY METHODS

Method	**Description**
dic.clear()	Remove all the elements from the dictionary
dict.copy()	Returns a copy of the dictionary
dict.get(key, default = “None”)	Returns the value of specified key
dict.items()	Returns a list containing a tuple for each key value pair
dict.keys()	Returns a list containing dictionary’s keys
dict.update(dict2)	Updates dictionary with specified key-value pairs
dict.values()	Returns a list of all the values of dictionary
pop()	Remove the element with specified key
popItem()	Removes the last inserted key-value pair