MODULE : 1 PYTHON FUNDAMENTALS

❖ Introduction to Python Theory:

1. Introduction to Python and its Features (simple, high-level, interpreted language).

- Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.
- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language

Simple :

- Python's syntax is clean and easy to understand, resembling plain English. This reduces the learning curve for new programmers and promotes clarity and maintainability in code.

high-level **:**

- 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.

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.

2. History and evolution of Python.

History :

- Python is a widely used general-purpose, high-level programming language. It was initially designed by **Guido van Rossum** in **1991** and developed by Python Software Foundation. It was mainly developed to emphasize code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Evolution :

- The language was finally released in 1991. When it was released, it used a lot fewer codes to express the concepts, when we compare it with java, c++ & c Its design philosophy was quite good too. Its main objective is to provide code readability and advanced developer productivity.

3. Advantages of using Python over other programming languages.

- Python is a widely used programming language that is utilized extensively in various disciplines such as web development, data analysis, artificial intelligence, and scientific computing.

1.The simplicity of Usage and Learning :

- Python has an easy-to-understand syntax, and the language has a sizable and engaged user base that contributes to a wide range of libraries and tools.

2.Versatility

- Python is highly versatile. The language is used in a wide range of applications, including web development, data analysis, and scientific computing, and there are libraries and frameworks available for just about any task you can think of.

3.Performance :

- Python is generally considered to be quite fast, and it has a number of features that make it well-suited for high-performance computing tasks.

4. Data Analysis and Scientific Computing :

- One of the key advantages of Python is its strong support for data analysis and scientific computing. The language has a number of libraries and frameworks that are specifically designed for these tasks, such as NumPy, Pandas, and SciPy.

5. Large and Active Community

- Another advantage of Python is that it has a large and active community of users. This means that there is a wealth of resources and support available for developers who are working with the language.

6. Flexibility in Deployment

- Python is also highly flexible when it comes to deployment. It applies to creating desktop, online, and even mobile applications.

7. Object-Oriented Programming

- Python is an object-oriented programming language (OOP). Python has a number of features that support OOP, including classes, inheritance, and polymorphism, which make it a good choice for projects that use this programming paradigm.

8. Functional Programming :

- Python also supports functional programming, which is a benefit. The foundation of the programming paradigm known as "functional programming" is the idea of "functions," which are self-contained units of code that carry out particular tasks.

9. Testing and Debugging

- Python is also known for its strong support for testing and debugging. The language has a number of libraries and frameworks that make it easy to write and run automated tests, as well as to identify and fix errors in your code

10 Integration with Other Languages and Systems

- Python has a number of libraries and frameworks that make it easy to call code written in other languages, such as C and C ++, as well as to interact with other systems and technologies, such as databases and web servers.

4. Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).

1.Download Python:

- Visit the official Python website: https://www.python.org.
- Navigate to the Downloads section and select the appropriate version for your operating system (Windows, macOS, or Linux).

2.Install Python:

- - Run the installer and follow the on-screen instructions.
- - Ensure that the checkbox "Add Python to PATH" is selected during installation. This allows you to use Python from the command line.

3.Verify Installation :

- - Open a terminal or command prompt and type:
  python –version

Setting Up Development Environments

1.Download VS Code :

- Visit the official website: https://code.visualstudio.com.

2.Install Python Extension :

- Open VS Code and go to the Extensions Marketplace.
- Search for and install the Python extension by Microsoft.

4. Writing and executing your first Python program.

- print("Hello, World!")
- output :- Hellow World!

1.Programming Style

1. Understanding Python's PEP 8 guidelines.

- PEP 8 is a document that provides guidelines and best practices on how to write Python code in a readable and consistent manner. It is the official style guide for Python code and is widely followed by Python developers.

- key sections of PEP 8:

1. Indentation
2. Line Length
3. Blank Lines
4. Imports
5. Whitespace in Expressions and Statements
6. Comments
7. Naming Conventions
8. Docstrings
9. Programming Practices
10. Version Compatibility

2.Indentation, comments, and naming conventions in Python.

1.Indentation:

- Use **4 spaces per indentation level**. This is to make the code structure clear and easy to read.
- **Never use tabs**; always use spaces. This avoids conflicts between different editors and IDEs.

2. Comments :

- Comments should be used to explain **why** something is done, not **what** is done (which should be evident from the code itself).
- Use **block comments** to explain larger sections of code. Block comments should be indented to the same level as the code they describe.
- Use **inline comments** sparingly and only to clarify code that is complex or not immediately clear:

python
x = x + 1 # Increment x by 1

3. Naming Conventions :

- **Function names**: Use lowercase with words separated by underscores, i.e., function_name.
- **Variable names**: Similar to function names, use lowercase and underscores, i.e., variable_name.
- **Class names**: Use the CapWords (CamelCase) convention, i.e., ClassName.
- **Constants**: Use uppercase letters with words separated by underscores, i.e., CONSTANT_NAME.
- **Module names**: Should be short, all lowercase, and use underscores if necessary, i.e., module_name.
- **Package names**: Should be short and lowercase without underscores.

3. Writing readable and maintainable code.

- First, let's define what we mean by "maintainability" and "readability." Code maintainability refers to the ease with which a developer can make changes to the codebase without introducing new bugs. Readability, on the other hand, refers to the ease with which a developer can understand the codebase.

2.Core Python Concepts :

1. Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

1.Interger :

- Whole numbers, positive or negative, without any decimal point.

- Example : 3,0,42

2.Floats **:**

- Numbers that have a decimal point, used for representing real numbers. They can also be in scientific notation.
- **Examples**: 3.14, -0.001

3.String :

- A sequence of characters, used to represent text. Strings can be defined using single quotes, double quotes, or triple quotes for multi-line strings.
- **Examples**: "Hello World!", 'Python', """This is a python."""

4.Lists :

- An ordered collection of items, which can be of different data types. Lists are mutable, meaning you can change, add, or remove items.
- **Syntax**: Square brackets [] are used to define a list.
- **Examples**: [1, 2, 3], ['apple', 'banana', 'cherry'], [1, 'two', 3.0]

5.Tuples :

- Similar to lists, but tuples are immutable, meaning once created, you cannot change their elements. They are often used for fixed collections of items. their
- **Syntax**: Parentheses () are used to define a tuple.
- **Examples**: (1, 2, 3), ('apple', 'banana', 'cherry'), (1, 'two', 3.0)

6.Dictionaries :

- A collection of key-value pairs. Keys must be unique and immutable, allowing for efficient data retrieval.
- **Syntax**: Defined using curly braces {} with keys and values separated by colons :.
- **Examples**: {'name': 'Alice', 'age': 30}, {'brand': 'Ford', 'model': 'Mustang'}

6.Sets :

- An unordered collection of unique items. Sets are used to store unique values and support mathematical set operations like unions and intersections.
- **Syntax**: Defined using curly braces {} or the set() function.
- **Examples**: {1, 2, 3}, {'apple', 'banana', 'cherry'}, set([1, 2, 2, 3]) (this will result in {1, 2, 3})

2. Python variables and memory allocation.

➢ **Memory allocation** in Python refers to the process of reserving memory to store objects (like integers, strings, lists, etc.) when they are created during program execution. Python uses an automatic memory management system to handle this process.

- ➢ **Variable :**
- ➢ In Python, a **variable** is a symbolic name that refers to an object stored in memory. When you assign a value to a variable, you are binding the variable name to an object in memory. Variables in Python are not containers for values, but references (or pointers) to objects in memory.

3. Python operators: arithmetic, comparison, logical, bitwise.

1.Arithmetic operator：

- Python Arithmetic operator 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//

2.Comparison operator :

- Python operator can be used with various data types, including numbers, strings, boolean and more. In Python, comparison operators are used to compare the values of two operands (elements being compared). When comparing strings, the comparison is based on the alphabetical order of their characters (lexicographic order).

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

3.Logical Operator :

- Python logical operator 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	A and B
or	Logical OR: True if either of the operands is true	A or B
not	Logical NOT: True if the operand is false	A not B

3.Bitwise operators :

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

Operator	Description	Syntax
&	Bitwise AND	A & B
|	Bitwise OR	A | B
~	Bitwise NOT	A ~ B

4. Conditional Statements :

1.Introduction to conditional statements: if, else, elif

- There are situations in real life when we need to do some specific task and based on some specific conditions, we decide what we should do next. Similarly, there comes a situation in programming where a specific task is to be performed if a specific condition is True.

1. if Statement :

- The if statement is used to test a condition. If the condition evaluates to True, the block of code inside the if statement is executed.
- Syntax :

```
if condition:
    #statements
```

2. else Statement :

- The else statement is used when the if condition is False. It provides an alternative block of code to execute if the condition is not met.
- Syntax :

```
if condition:
    # if condition is True
else:
    # if condition is False
```

2. elif Statement :

- elif stands for "else if" and is used when you want to check multiple conditions. If the first if condition is False, Python will check the elif conditions in order. If any of the elif conditions are True, the associated block of code is executed. You can have multiple elif statements in a single if-else block.
- Syntax:

```
If condition :
    #if condition1 is true
elif condition:
    # elif condition2 is true
Else :
    # if none of the conditions are True
```

2. Nested if-else conditions.

- **Nested if-else conditions** are when you have an if-else statement inside another if-else statement. This allows you to create more complex decision structures by checking additional conditions after the initial condition is evaluated.

- **Syntax :**

```
if condition1:
   if condition2:
      # if both condition1 and condition2 are True
   else:
      # if condition1 is True and condition2 is False
 else:
     # if condition1 is False
```

5. Looping (For, While)

1. Introduction to for and while loops.

- In programming, **loops** are used to repeat a block of code multiple times.

1.for loop:

- A for loop is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each item in that sequence. It is often used when you know in advance how many times you want to loop.
- Syntax :

```
for item in sequence:
    # statements
```

1.while loop :

- A while loop repeatedly executes a block of code as long as a given condition is True. This loop is used when you don't know how many times you need to repeat the code but want to continue as long as the condition is met.
- Syntax :

```
While condition:
    # statements
```

2. How loops work in Python.

- **for loop**: Best used for iterating over a sequence of elements (like a list, string, or range). You know the number of iterations in advance.
- **while loop**: Best used when you want to repeat a block of code as long as a condition is True. You may not know the number of iterations beforehand.

- Both loops are essential in Python, and choosing the right type of loop depends on the problem you're trying to solve. Properly managing the flow with break, continue, and avoiding infinite loops will make your code more efficient and maintainable

3. Using loops with collections (lists, tuples, etc.).

- Iterating Over Collections: The most common use of for loops is to iterate over collections like lists, tuples, dictionaries, etc., accessing each element individually.

1.lists :

- A **list** is an ordered collection of elements. You can use a loop to access and process each element in a list.

2.tuples :

- A **tuple** is similar to a list but is immutable (cannot be changed). You can use a loop to iterate over the elements in a tuple, just like a list.

3.dictinaries :

- A **dictionary** is a collection of key-value pairs. You can use loops to access the keys, values, or both in a dictionary.

4.sets :

- A **set** is an unordered collection of unique elements. You can iterate through a set using a for loop, but the order of elements is not guaranteed.

6. Generators and Iterators

1. Understanding how generators work in Python.

- **Generators** in Python are special functions that allow you to iterate over a sequence of values, but instead of storing all values in memory, they generate each value on the fly when needed. They are defined using the yield keyword.

How Do They Work?

1. **Generator Function**: A function that uses yield to produce values one at a time.
2. **Stateful**: When yield is called, the function "pauses" and remembers its state. On the next iteration, it resumes where it left off.
3. **Lazy Evaluation**: Values are computed only when requested, saving memory

2. Difference between yield and retur.

No	YIELD	RETURN
1	Yield is generally used to convert a regular Python function into a generator.	Return is generally used for the end of the execution and “returns” the result to the caller statement.
2	It replace the return of a function to suspend its execution without destroying local variables.	It exits from a function and handing back a value to its caller.
3	It is used when the generator returns an intermediate result to the caller.	It is used when a function is ready to send a value.
4	Code written after yield statement execute in next function call.	while, code written after return statement wont execute.
5	It can run multiple times.	It only runs single time.

3. Understanding iterators and creating custom iterators.

- An iterator in Python is an object that holds a sequence of values and provide sequential traversal through a collection of items such as lists, tuples and dictionaries. . The Python iterators object is initialized using the **iter()** method. It uses the **next()** method for iteration.
- **__iter__(): __**iter__() method initializes and returns the iterator object itself.
- **__next__(): **the __next__() method retrieves the next available item, throwing a StopIteration exception when no more items are available.

What is a Custom Iterator?

- A **custom iterator** is an iterator that you define yourself by creating a class that implements the __iter__() and __next__() methods. This is useful when you need to customize the way iteration works over your objects or data.

How to Create a Custom Iterator

1. **Create a class**: This class should implement the __iter__() and __next__() methods.
   - __iter__() should return the iterator object (which is typically self).
   - __next__() should return the next value and raise StopIteration when there are no more values to iterate.

7. Functions and Methods

1. Defining and calling functions in Python.

- We can define a function in Python, using the **def** keyword. We can add any type of functionalities and properties to it as we require. By the following example, we can understand how to write a function in Python. In this way we can create Python function definition by using def keyword.

- Syntax:
  def function name:
  # print statements

Calling function :

- After creating a function in Python we can call it by using the name of the functions Python followed by parenthesis containing parameters of that particular function. Below is the example for calling def function Python.
- Syntax :
  def fun():
  print statements
  fun()

2. Function arguments (positional, keyword, default).

- Python provides different ways of passing the arguments during the function call from which we will explore **keyword-only argument** means passing the argument by using the parameter names during the function call.

1. Positional Arguments

- Positional arguments are the most straightforward type of arguments. They are passed to a function in the exact order in which the parameters are defined.

2. Keyword Arguments

- Keyword arguments are passed by explicitly naming the parameter in the function call. You specify the parameter name followed by the value you want to assign to it.

3. Default Arguments

- Default arguments allow you to specify a default value for a parameter in the function definition. If no argument is provided for that parameter when the function is called, the default value is used.

3. Scope of variables in Python.

- The location where we can find a variable and also access it if required is called the **scope of a variable**.

Python Local variable :

- **Local variables are those that are initialized within a function and are unique to that function. It cannot be accessed outside of the function. Let's look at how to make a local variable.**

Python Global variables

- Global variables are the ones that are defined and declared outside any function and are not specified to any function. They can be used by any part of the program.

4. Built-in methods for strings, lists, etc

- In Python, both strings and lists are objects, and they come with built-in methods that can be used to perform common operations. Below are some of the built-in methods for strings, lists, and other common objects.

String Methods :

- Strings in Python are sequences of characters, and they come with various methods for manipulation.
- Example :

  Name = "khushbu kindarkhediya

List Methods :

- Lists in Python are ordered collections of items and have a variety of methods to modify and manipulate the list.
- Example:

  lst = [1, 2, 3]

lst.append(4)

Dictionary Methods :

- Dictionaries are key-value pairs, and they come with various built-in methods for manipulation.

Set Methods :

- Sets are unordered collections of unique elements.

8. Control Statements (Break, Continue, Pass)

1. Understanding the role of break, continue, and pass in Python loops.

- Using loops in Python automates and repeats the tasks in an efficient manner. But sometimes, there may arise a condition where you want to exit the loop completely, skip an iteration or ignore that condition. These can be done by **loop control statements**.

Break statement:

- The break statement in Python is used to exit or “break” out of a loop (either a for or while loop) prematurely, before the loop has iterated through all its items or reached its condition. When the break statement is executed, the program immediately exits the loop, and the control moves to the next line of code after the loop.

Continue statement :

- Python continue statement is a loop control statement that forces to execute the next iteration of the loop while skipping the rest of the code inside the loop for the current iteration only, i.e. when the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped for the current iteration and the next iteration of the loop will begin.

Pass statement :

- Pass keyword in a function is used when we define a function but don’t want to implement its logic immediately. It allows the function to be syntactically valid, even though it doesn’t perform any actions yet.

9. String Manipulation

1. Understanding how to access and manipulate strings

- A string is a sequence of characters. Python treats anything inside quotes as a string. This includes letters, numbers, and symbols. Python has no character data type so single character is a string of length 1.

- **Accessing characters**: Use indexing (e.g., my_string[0]).
- **Slicing**: Use my_string[start:end] or slice() in JS.
- **Length**: Use len() in Python and .length in JavaScript.
- **Concatenation**: Use + for combining strings.
- **Replacing**: Use replace() in both Python and JavaScript.
- **Changing case**: Use .upper(), .lower() in Python; .toUpperCase(), .toLowerCase() in JavaScript.
- **Trimming**: Use .strip() in Python, .trim() in JavaScript.
- **Splitting**: Use .split() to break a string into parts.
- **Finding substrings**: Use .find() in Python, .indexOf() in JavaScript.

- **Joining**: Use .join() for concatenating parts of a list/array.

2. Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.)

- In Python, you can perform various operations on strings, including concatenation, repetition, and using built-in methods:

1. Concatenation

- Concatenation involves combining two or more strings into one.
- Example :

```
str1 = "Hello"

str2 = "World"

result = str1 + " " + str2

print(result)
```

2. Repetition

- Repetition allows you to repeat a string multiple times.
- Example

```
str1 = "Hi! "

result = str1 * 3

print(result)
```

3. String Methods

- Python provides several built-in methods to manipulate and analyze strings. Here are some common ones:
- upper()
- Example :

```
str1 = "hello"

result = str1.upper()

 print(result)
```

- lower()
- Example :

```
str1 = "HELLO"

result = str1.lower()

print(result)
```

3. String slicing.

- 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.
- Example :

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

10. Advanced Python (map(), reduce(), filter(), Closures and Decorators)

1. How functional programming works in Python.

- Functional programming in Python treats functions as first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables. It emphasizes immutability, pure functions, and declarative code. Key features include:
  - **First-Class Functions**: Functions can be passed around like any other object.
  - **Higher-Order Functions**: Functions that take other functions as input or return them.
  - **Lambda Functions**: Anonymous, small functions defined with lambda.
  - **Immutability**: Prefer immutable data (e.g., tuples) over mutable ones (e.g., lists).
  - **Map, Filter, Reduce**:
    - **map()**: Apply a function to each element.
    - **filter()**: Filter elements based on a condition.
    - **reduce()**: Accumulate values using a binary function.
- Example :

```
from functools import reduce

numbers = [1, 2, 3, 4, 5, 6]

result = reduce(lambda x, y: x + y, map(lambda x: x**2, filter(lambda x: x % 2 == 0, numbers)))

print(result)
```

2. Using map(), reduce(), and filter() functions for processing data.

- In Python, **map()**, **reduce()**, and **filter()** are higher-order functions that allow you to process data in a functional programming style.

1. map() Function

- The **map()** function applies a given function to each item of an iterable (list, tuple, etc.) and returns a map object (which can be converted into a list, tuple, etc.).
- **Syntax**: map(function, iterable)

2. filter() Function

- The **filter()** function filters elements from an iterable based on a function that returns a boolean value (True or False). It returns an iterator with the items for which the function returned True.
- **Syntax**: filter(function, iterable)

3. reduce() Function

- The **reduce()** function from the functools module applies a rolling computation to sequential pairs of values in an iterable. It reduces the iterable to a single cumulative value.
- **Syntax**: reduce(function, iterable, [initializer])

3. Introduction to closures and decorators.

Closures in Python

- A **closure** is a function that "remembers" its environment, meaning it can access variables from its enclosing scope even after that scope has finished executing. Closures are useful when you want to create a function that maintains state across multiple calls but without using global variables.

Key Points:

- A closure occurs when a nested function (inner function) refers to a variable from its enclosing scope (non-local scope).
- The inner function "remembers" the environment in which it was created.

Decorators in Python

- A **decorator** is a function that takes another function as an argument and extends or alters its behavior without modifying the function itself. Decorators are widely used to add functionality to existing code, often in the context of logging, access control, memoization, and more.

Key Points:

- Decorators are typically used with the @decorator_name syntax, but they can be applied manually by passing a function as an argument to another function.
- Decorators can be used to wrap functions or methods, effectively modifying their behavior.