Name : **Dhruvit Savaliya**

Module - Python Fundamentals

1. Introduction to Python Theory:

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

  - Python is a versatile and powerful programming language that has gained immense popularity across various domains such as web development, data analysis, artificial intelligence, scientific computing, and more.

  - Features of Python

  - Simple and Easy to Learn
    Python's syntax is clean and straightforward, resembling natural language.
    This simplicity reduces the learning curve and makes it accessible to beginners.

- High-Level Language
  As a high-level language, Python abstracts many complex programming details, allowing developers to focus on the logic and functionality of their code rather than on intricate system-level details.

- Interpreted Language
  Python is an interpreted language, meaning that the code is executed line by line at runtime.

- Dynamically Typed
  The type of a variable is determined at runtime, which provides flexibility and reduces boilerplate code.

- Platform Independent
  Python code is portable and can run on multiple platforms (Windows, macOS, Linux) with minimal or no modification.

History and evolution of Python.

- Python is a versatile and powerful programming language that has gained immense popularity across various domains such as web development, data analysis, artificial intelligence, scientific computing, and more.
- Created by Guido van Rossum and first released in 1991, Python emphasizes code readability and simplicity, making it an excellent choice for beginners and experienced developers alike.

- Late 1980s: Creation
  Guido van Rossum began developing Python at CWI in the Netherlands.
  Inspired by the ABC language but aimed to add more features.

- 1991: Python 0.9.0
  First release in February 1991.
  Included functions, exception handling, and core data types.

- 1994: Python 1.0
  Official release in January 1994.
  Added modules, classes, and basic tools like map() and filter().

- 2000: Python 2.0
  Released in October 2000.
  Introduced list comprehensions and garbage collection.
  Widely used but had design issues.

- 2008: Python 3.0
  Released in December 2008.
  Not backward compatible with Python 2.x.
  Improved Unicode support, cleaner syntax, and modernized libraries.

Advantages of using Python over other programming languages.

- Simple and Readable Syntax
  Python’s syntax is clean and easy to understand, making it beginner-friendly and efficient for experienced developers.

- Versatility

Python supports multiple paradigms and can be used for web development, data analysis, AI, automation, and more.

- Extensive Standard Library
  Python offers a wide range of built-in modules and functions for tasks like file handling, networking, and web scraping, reducing the need for additional libraries.

- Cross-Platform Compatibility
  Python code can run on various operating systems (Windows, macOS, Linux) with minimal changes.

- Integration and Extensibility
  Python integrates well with other languages and tools, making it suitable for diverse applications.

- Dynamic Typing
  Variables don’t require explicit type declarations, simplifying development and reducing boilerplate code.

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

- Download and Install VS Code
  Visit VS Code’s website and download the installer.

- Install Python Extension
  Open VS Code, go to the Extensions Marketplace (Ctrl+Shift+X), and search for Python. Install the extension by Microsoft.

- Set Up Python Environment
  Click Select Interpreter in the Command Palette (Ctrl+Shift+P) and choose your Python installation.

- Verifying the Setup
  Open your chosen IDE or environment.
  Create a new Python file (test.py) with the following code:

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

  Run the file. If you see Hello, Python! in the output, your setup is complete.

- Writing and executing your first Python program.

- Open VS Code and create a new file named hello.py.

- Write the same program:

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

- Use the Run button in the top right or press Ctrl+F5 to execute the file.

  Output will appear in the terminal.

2. Programming Style

- Understanding Python's PEP 8 guidelines.

- Code Layout

  Maximum Line Length: Limit lines to 79 characters. For longer blocks, like docstrings or comments, use 72 characters.

  Blank Lines:

  Use 2 blank lines to separate top-level functions or classes.

  Use 1 blank line to separate methods within a class.

- Imports

  Place all imports at the top of the file.

  Follow this order:

  Standard library imports

  Related third-party imports

  Local application-specific imports

- Naming Conventions

  Variables, Functions, and Methods: Use snake_case.

  ```
  def calculate_total():
  total_amount = 100
  ```

  Class Names: Use CamelCase.

  ```
  class MyClass:
  pass
  ```

- Whitespace Usage

Avoid extra spaces inside parentheses, brackets, or braces.
Use a single space around operators and after commas.

- Comments
  Use comments sparingly but meaningfully.
  Begin comments with a # and a single space

- Docstrings
  Use triple double-quotes for docstrings in functions, classes, and modules.

- Avoid Trailing Whitespace
  Ensure there are no unnecessary spaces at the end of lines.

- Logical Comparisons
  Use is for singletons like None.

Indentation, comments, and naming conventions in Python.

- Indentation
  Indentation is critical in Python as it defines the structure of the code.
  Unlike many other languages, Python uses indentation instead of braces {} to define blocks of code.

  Indentation Rules:
    Use 4 spaces per indentation level (PEP 8 standard).
    Avoid mixing spaces and tabs to prevent errors.
    All statements within a block must have the same indentation level.

- Comments
  Comments explain the purpose of code and improve readability.

  Single-Line Comments:
    Begin with a # and a single space.
  Multi-Line Comments:
    Use multiple # lines for longer comments.
  Docstrings:
    Use triple double-quotes """ for documenting functions, classes, or modules.

- Naming Conventions

Naming conventions help create readable and consistent code.

Variables and Functions:

Use snake_case (lowercase letters with underscores between words).

```
total_price = 100
def calculate_sum(a, b):
return a + b
```

Constants:

Use ALL_CAPS for constants.

```
MAX_LIMIT = 100
```

Classes:

Use CamelCase with the first letter of each word capitalized.

```
class MyClass:
pass
```

- Writing readable and maintainable code.

- Here's a simpler version of the program that calculates the area of a circle and a rectangle:

```
import math

def circle_area(radius):
   """Calculates the area of a circle."""
   return math.pi * radius ** 2

def rectangle_area(width, height):
   """Calculates the area of a rectangle."""
   return width * height

# Example usage
print("Circle area:", circle_area(5))
print("Rectangle area:", rectangle_area(4, 7))
```

3. Core Python Concepts

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

➢ Integers (int)

Whole numbers without a fractional part.

```
x = 5
y = -3
z = 1000
```

➢ Floats (float)

Numbers with a decimal point, representing real numbers.

```
pi = 3.14
temp = -0.5
distance = 100.75
```

➢ Strings (str)

A sequence of characters enclosed in single (') or double (") quotes.

```
Str1 = 'Hello, World!'
```

➢ Lists (list)

Ordered, mutable (changeable) collection of elements that can be of different data types.

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "apple", 3.14, True]
```

➢ Tuples (tuple)

Ordered, immutable (unchangeable) collection of elements, like lists, but cannot be modified after creation.

```
point = (3, 4)
colors = ("red", "green", "blue")
```

➢ Dictionaries (dict)

Unordered collection of key-value pairs, where each key must be unique.

```
person = {"name": "Alice", "age": 25, "city": "New York"}
```

➢ Sets (set)

Unordered collection of unique elements. Sets do not allow duplicates.

```
fruits = {"apple", "banana", "cherry"}
numbers = {1, 2, 3, 4, 5}
```

- Python variables and memory allocation.

➢ Variables in Python

A variable in Python is essentially a name (identifier) used to reference data stored in memory.

Python variables are dynamically typed, meaning you do not need to declare a variable with its type explicitly; Python determines the type based on the value assigned.

```
x = 10  # 'x' is a variable referencing an integer object with value 10.
name = "Alice"  # 'name' references a string object.
```

➢ Memory Allocation for Variables

Python manages memory using object-oriented memory management, where every value is treated as an object.

Here’s how memory allocation works:

Object Creation: When you assign a value to a variable, Python creates an object in memory to store the value.

Variable Binding: The variable acts as a reference (or pointer) to that object.

Memory Address: Every object has a unique memory address, which you can inspect using the id() function.

```
a = 10
b = 10
print(id(a))  # Memory address of the integer object 10
print(id(b))  # Same address, as Python reuses immutable objects
```

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

➢ Arithmetic Operators

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 – 3	2
*	Multiplication	5 * 3	15
/	Division	5 / 2	2.5
//	Floor Division	5 // 2	2
%	Modulus	5 % 2	1
**	Exponentiation	5 ** 2	25

➢ Comparison (Relational) Operators

Operator	Description	Example	Result
==	Equal to	5 == 3	False
!=	Not equal to	5 != 3	True
>	Greater than	5 > 3	True
<	Less than	5 < 3	False
>=	Greater than or equal	5 >= 3	True
<=	Less than or equal	5 <= 3	False

➢ Logical Operators

Operator	Description	Example	Result
And	Logical AND (True if both True)	True and False	False
Or	Logical OR (True if at least one is True)	True or False	True
not	Logical NOT (Negates the value)	not True	False

➢ Bitwise Operators

Operator	Description	Example	Result
&	Bitwise AND	5 & 3	0001 (1)
^	Bitwise XOR	5 ^ 3	0110 (6)
~	Bitwise NOT	~5	1010 (-6)
<<	Left shift (multiply by 2^n)	5 << 1	1010 (10)
>>	Right shift (divide by 2^n)	5 >> 1	0010 (2)

4. Conditional Statements

- **Introduction to conditional statements: if, else, elif.**

➢ Conditional statements in Python allow the program to make decisions and execute different blocks of code based on specific conditions.

➢ These conditions evaluate to either True or False.

➢ if Statement

The if statement checks a condition. If the condition evaluates to True, the code inside the if block is executed.

```
x = 10
if x > 5:
    print("x is greater than 5")  # Output: x is greater than 5
```

- if-else Statement

  The if-else structure allows you to specify an alternative block of code to execute when the condition evaluates to False.

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is less than or equal to 5")  # Output: x is less than or equal to 5
```

- if-elif-else Statement

  The elif (short for else if) allows checking multiple conditions. If one of the conditions evaluates to True, the corresponding block is executed, and the rest are ignored.

```
x = 7
if x > 10:
    print("x is greater than 10")
elif x > 5:
    print("x is greater than 5 but less than or equal to 10")  # Output: x is greater than
5 but less than or equal to 10
else:
    print("x is less than or equal to 5")
```

Nested if-else conditions.

- A nested if-else condition is a conditional statement placed inside another if or else block.
- This allows the program to evaluate additional conditions based on previous ones. It is useful for handling complex decision-making processes.

- Syntax

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

- How It Works

  The outer if condition is evaluated first.
  If the outer if is True, the inner if (or nested else) is evaluated.
  If the outer if is False, the outer else block is executed, skipping the nested conditions.

5. Looping (For, While)

- Introduction to for and while loops.

- Loops are used in programming to repeat a block of code multiple times.
- Python provides two primary types of loops: for and while.

- The for Loop

  The for loop is used to iterate over a sequence (like a list, tuple, string, or range) or other iterable objects.
  It executes the block of code once for each element in the sequence.

  Syntax:

  ```
  for variable in iterable:
   # Code block to execute for each element
  ```

- The while Loop

  The while loop is used when you want to repeat a block of code as long as a condition is true.

  Syntax:

  ```
  while condition:
  # Code block to execute while condition is True
  ```

- How loops work in Python.

- Loops in Python allow for repeated execution of a block of code.

- The for Loop

  The for loop iterates over an iterable, such as a list, tuple, string, dictionary, or a generator.

  Python uses an iterator protocol to fetch elements one at a time.

  How it Works:

    The loop retrieves the first element from the iterable.
    Executes the code block with the current element.
    Proceeds to the next element until all elements are exhausted.

- The while Loop

  The while loop executes based on a condition. The loop will continue as long as the condition evaluates to True.

  How it Works:

    The condition is evaluated before each iteration.
    If the condition is True, the code block is executed.
    After the block, the condition is checked again.

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

- Iterating Over Lists

  Lists are ordered, mutable collections of elements. You can use for loops or while loops to iterate over them.

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

- Iterating Over Tuples

  Tuples are immutable collections. The iteration process is similar to lists.

```
coordinates = (10, 20, 30)
for coord in coordinates:
   print(coord)
```

- Iterating Over Sets

Sets are unordered collections of unique elements. Iteration is done directly over elements, but the order is not guaranteed.

```
unique_numbers = {1, 2, 3, 4}
for num in unique_numbers:
    print(num)
```

6. Generators and Iterators

- **Understanding how generators work in Python.**

➢ Generators are a special type of iterable in Python that allow you to produce values lazily—one at a time and only as needed.

➢ They are particularly useful when working with large datasets or streams of data where creating and storing the entire dataset in memory would be inefficient.

➢ What Are Generators?

Generators are iterators that do not store their contents in memory. Instead, they generate values on the fly.
They are defined using either a generator function or a generator expression.

➢ Generator Functions

A generator function is a function that uses the yield statement to produce a series of values one at a time.

How It Works:

When called, a generator function returns a generator object but does not execute the function body.
Each time next() is called on the generator, the function runs until it hits a yield statement.
The yield statement pauses the function, saving its state, and returns the value to the caller.
The function resumes execution from where it was paused.

- **Difference between yield and return.**

yield	return
Sequence of values (one at a time).	Final result (single output).
Retains function state for resumption.	Ends function execution and loses state.
Memory-efficient (lazy evaluation).	Can be memory-intensive.
Ideal for iterating large/infinite data.	Suitable for returning complete results.
Ideal for large datasets or infinite data.	Best for small or manageable data sizes.

- **Understanding iterators and creating custom iterators.**

➢ What Is an Iterator?

An iterator is an object that allows you to traverse through all the elements of a collection, such as a list or a tuple, one by one, without needing to know the underlying structure.

Key Components of an Iterator:

__iter__(): This method returns the iterator object itself. It's called when the iteration is initialized, typically by calling iter() on the object.

__next__(): This method returns the next item in the sequence. When there are no more items, it raises the StopIteration exception, signaling that the iteration is complete.

➢ Creating Custom Iterators

You can create custom iterators by defining a class that implements both __iter__() and __next__() methods.

Steps to Create a Custom Iterator:

Define the __iter__() method: This method should return the iterator object itself.

Define the __next__() method: This method should return the next item and raise StopIteration when there are no more items.

7. Functions and Methods

- **Defining and calling functions in Python.**

➢ Defining a Function

A function is defined using the def keyword, followed by the function name and parentheses.
Optionally, you can include parameters inside the parentheses.

```
def function_name(parameters):
    """Optional docstring: explains the function."""
    # Body of the function
    return result  # Optional
```

function_name: The name you choose for the function.
parameters: Inputs to the function (optional).
return: Used to return a result (optional).

- Calling a Function

  Use the function name followed by parentheses.
  Pass arguments if the function requires them.

```
def multiply(a, b):
    return a * b

result = multiply(6, 7)  # Call with arguments
print(result)  # Output: 42
```

Function arguments (positional, keyword, default).

- Positional Arguments

  Positional arguments are the most basic type of arguments.
  Their values are assigned to parameters based on their position in the function call.

```
def greet(name, message):
    """Greets a person with a message."""
    print(f"{message}, {name}!")

# Calling the function
greet("Alice", "Hello")  # Positional arguments
```

- Keyword Arguments

  Keyword arguments are explicitly assigned by the parameter name during a function call, making the order irrelevant.

```
def greet(name, message):
    """Greets a person with a message."""
    print(f"{message}, {name}!")

# Calling the function
greet(message="Hi", name="Bob")  # Keyword arguments
```

- Default Arguments

  Default arguments allow you to specify a default value for a parameter.
  If the caller doesn’t provide a value, the default is used.

```
def greet(name, message="Hello"):
    """Greets a person with a default message."""
    print(f"{message}, {name}!")

# Calling the function
greet("Charlie")  # Only `name` is provided
greet("Diana", "Good morning")  # Both `name` and `message` are provided
```

Scope of variables in Python.

- Local Scope

  Variables declared inside a function are local to that function.
  They can only be accessed within the function where they are defined.

```
def greet():
    message = "Hello, World!"  # Local variable
    print(message)

greet()
# print(message)  # NameError: name 'message' is not defined
```

- Global Scope

  Variables declared outside of any function or block have a global scope.
  They can be accessed from any part of the program.

```
greeting = "Hello"  # Global variable
```

```
def greet():
    print(greeting)  # Accessing the global variable

greet()
print(greeting)  # Global variable can be accessed outside the function
```

- Enclosing Scope (Nonlocal Scope)

  Enclosing scope applies to nested functions.
  Variables in an outer (enclosing) function are accessible to inner functions.
  To modify them in the inner function, use the nonlocal keyword.

```
def outer():
    message = "Outer scope"

    def inner():
        nonlocal message
        message = "Modified in inner scope"
        print("Inner:", message)

    inner()
    print("Outer:", message)

outer()
```

Built-in methods for strings, lists, etc.

- String Methods

  str.lower(): Converts to lowercase.
  str.upper(): Converts to uppercase.
  str.capitalize(): Capitalizes the first letter.
  str.title(): Capitalizes the first letter of each word.
  str.strip(): Removes leading and trailing spaces (or specified characters).
  str.replace(old, new): Replaces all occurrences of old with new.

  str.startswith(sub): Returns True if the string starts with sub.
  str.endswith(sub): Returns True if the string ends with sub.
  str.isalpha(): Returns True if all characters are alphabetic.
  str.isdigit(): Returns True if all characters are digits.
  str.isspace(): Returns True if all characters are whitespace.

- List Methods

  list.append(x): Adds an item to the end.

  list.insert(i, x): Inserts an item at position i.

  list.pop([i]): Removes and returns the item at position i (last item if not specified).

  list.remove(x): Removes the first occurrence of x.

  list.clear(): Removes all elements.

  list.index(x): Returns the index of the first occurrence of x.

  list.count(x): Counts occurrences of x.

  len(lst): Returns the number of elements in the list.

  list.sort(): Sorts the list in ascending order (in-place).

  list.reverse(): Reverses the order of the list (in-place).

  sorted(lst): Returns a new sorted list.

8. Control Statements (Break, Continue, Pass)

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

- break Statement

  The break statement is used to exit a loop prematurely.
  When break is executed, the loop stops immediately, and control moves to the statement following the loop.

```
for i in range(5):
   if i == 3:
      break
   print(i)
print("Loop ended")
```

- continue Statement

  The continue statement is used to skip the rest of the code in the current iteration of the loop and proceed to the next iteration.

```
for i in range(5):
   if i == 3:
      continue
   print(i)
```

- pass Statement

  The pass statement is a placeholder.
  It does nothing and allows the loop or function to run without errors when no action is specified.

```
for i in range(5):
  if i == 3:
    pass  # Placeholder for future logic
  print(i)
```

9. String Manipulation

- **Understanding how to access and manipulate strings.**

- Accessing Strings

- Indexing

  Strings can be accessed using index numbers.
  The index starts from 0 for the first character and goes up to n-1 for the last character.
  Negative indices can be used to access characters from the end.

```
s = "Python"
print(s[0])  # 'P' (First character)
print(s[-1]) # 'n' (Last character)
```

- Slicing

  Slicing allows you to extract a substring using the syntax:
  string[start:end:step]
    start: Index to begin slicing (inclusive).
    end: Index to stop slicing (exclusive).
    step: Interval between characters.

```
s = "Python"
print(s[1:4])   # 'yth' (Characters at indices 1, 2, 3)
print(s[:3])    # 'Pyt' (First 3 characters)
print(s[::2])   # 'Pto' (Every 2nd character)
print(s[::-1])  # 'nohtyP' (Reversed string)
```

- Manipulating Strings

- Concatenation

  Combine two or more strings using the + operator.

```
s1 = "Hello"
s2 = "World"
print(s1 + " " + s2)  # 'Hello World'
```

- Repetition

  Repeat a string multiple times using the * operator.

```
s = "Python"
print(s * 3)  # 'PythonPythonPython'
```

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

- Concatenation

  Concatenation is the process of combining two or more strings using the + operator.

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)  # Output: "Hello World"
```

- Repetition

  Repetition involves repeating a string multiple times using the * operator.

```
str1 = "Hi! "
result = str1 * 3
print(result)  # Output: "Hi! Hi! Hi! "
```

- String Methods

  str.lower(): Converts to lowercase.
  str.upper(): Converts to uppercase.
  str.capitalize(): Capitalizes the first letter.
  str.title(): Capitalizes the first letter of each word.
  str.strip(): Removes leading and trailing spaces (or specified characters).

str.replace(old, new): Replaces all occurrences of old with new.
str.startswith(sub): Returns True if the string starts with sub.
str.endswith(sub): Returns True if the string ends with sub.
str.isalpha(): Returns True if all characters are alphabetic.
str.isdigit(): Returns True if all characters are digits.
str.isspace(): Returns True if all characters are whitespace.

• String slicing

- String slicing in Python allows you to extract parts of a string (substrings) using a specific syntax:

```
string[start:end:step]
```

start: The index where the slice starts (inclusive). Default is 0.
end: The index where the slice ends (exclusive). Default is the length of the string.
step: The interval between each character in the slice. Default is 1.

- Simple Slicing

```
s = "Python"

print(s[0:3])  # 'Pyt' (characters at indices 0, 1, 2)
print(s[2:5])  # 'tho' (characters at indices 2, 3, 4)
print(s[:4])   # 'Pyth' (start defaults to 0)
print(s[3:])   # 'hon' (end defaults to length of the string)
print(s[:])    # 'Python' (full string)
```

- Negative Indices

```
s = "Python"

print(s[-4:-1])  # 'tho' (characters from -4 to -2)
print(s[-3:])    # 'hon' (last three characters)
print(s[:-3])    # 'Pyt' (all except the last three characters)
```

- Skipping Characters (Step)

```
s = "Python"

print(s[::2])   # 'Pto' (every second character)
print(s[1::2])  # 'yhn' (every second character starting from index 1)
print(s[::-1])  # 'nohtyP' (reversed string)
print(s[::-2])  # 'nhy' (every second character in reverse order)
```

- Slicing Patterns
  1. Extract Substring
     ```
     s = "Programming"
     substring = s[0:6]
     print(substring)  # 'Progra'
     ```
  2. Reverse a String
     ```
     s = "Python"
     reversed_s = s[::-1]
     print(reversed_s)  # 'nohtyP'
     ```

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

• How functional programming works in Python.

- Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state or mutable data.

- Pure Functions

  Functions that always produce the same output for the same input and have no side effects.
  They don't modify external variables or depend on external state.

```
# Pure function
def add(a, b):
   return a + b

# Impure function (depends on external variable)
x = 10
def add_impure(a):
   return a + x
```

- First-Class Functions

  Functions in Python are first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables.

```
def square(x):
    return x * x

# Assign to a variable
func = square
print(func(5))  # 25

# Pass as an argument
def apply_function(f, value):
    return f(value)

print(apply_function(square, 6))  # 36
```

- Immutability

  Functional programming emphasizes immutability, where data is not modified but rather new data is created.
  Python does not enforce immutability but supports immutable data types like tuples and strings.

```
# Instead of modifying a list:
nums = [1, 2, 3]
new_nums = [x + 1 for x in nums]  # Creates a new list
print(new_nums)  # [2, 3, 4]
```

- Lambda Functions

  Anonymous, single-expression functions.
  Commonly used in map(), filter(), and reduce().

```
# Traditional function
def add(x, y):
    return x + y

# Lambda function
add_lambda = lambda x, y: x + y
print(add_lambda(2, 3))  # 5
```

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

- map()

  Applies a function to each element in an iterable and returns a map object (which can be converted to a list, tuple, etc.).

  Syntax:

  ```
  map(function, iterable)
  ```

- filter()

  Filters elements from an iterable based on a condition defined in a function.
  It returns a filter object containing elements where the condition is True.

  Syntax:

  ```
  filter(function, iterable)
  ```

- reduce()

  Aggregates elements in an iterable into a single value using a function.
  Unlike map() and filter(), reduce() is not a built-in function and must be imported from the functools module.

  Syntax:

  ```
  from functools import reduce
  reduce(function, iterable, initializer=None)
  ```

• Introduction to closures and decorators.

- Closures

  A closure is a function that remembers its lexical scope, even when the function is called outside that scope.
  In simpler terms, closures occur when a nested function references variables from its enclosing function, and the outer function has finished execution.
  The inner function "remembers" the environment in which it was created, including the values of the variables.

- Decorators

  A decorator is a function that takes another function as an argument, extends or alters its behavior, and then returns a new function.
  Decorators allow you to modify or enhance the behavior of functions or methods without changing their actual code.

Decorators are commonly used for:

Logging function calls
Timing functions
Adding access control or authentication

How Decorators Work:

A decorator is defined as a function that accepts a function as an argument.
It defines a new function that wraps (or enhances) the original function.
The new function is returned, and when the decorated function is called, the new behavior is applied.