Assignment 1

1. Introduction to Python :-

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

- Python is a programming language that turned into created between the 12 months of 1985 to 1990 by means of Dutch developer Guido van Rossum.
- Python is an interpreted, item-oriented, high-stage language with dynamic semantics. Python is a garbage-amassed and dynamic-typed programming language.
- It gives garbage-amassed and dynamic-typed programming language.
- It gives sturdy guide for integration with different languages and tools, comes with a huge fashionable library, and may be used for scripting, graphical consumer interfaces, internet packages, recreation improvement, and plenty

3. History and evolution of Python.:-

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.

4. Advantages of using Python over other programming languages :-

Python stands out due to its simplicity, readability, extensive libraries, cross-platform compatibility, large community support, and ease of learning, making it a preferred choice for various applications compared to other programming languages, especially for beginners and data science projects; its syntax is closer to natural language, allowing for faster development and prototyping

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

1. Install Python:
   - Go to python.org and download the latest stable version for your operating system.
   - Run the installer and follow the prompts.

2. Choose an IDE:
   - Anaconda:

     A comprehensive distribution that includes Python, the conda package manager, and many scientific computing libraries.
   - PyCharm:

     A dedicated Python IDE with powerful features for debugging, code analysis, and refactoring.
   - Visual Studio Code (VS Code):

     A versatile code editor that requires a separate Python extension to work with Python.

3. Set up your development environment:
   - Anaconda:

     Launch Anaconda Navigator.

Create a new conda environment with specific Python version if needed using the command conda create -n my_env python=3.x.
Activate the environment to start working.

- PyCharm:

Open PyCharm and create a new project.
Select the desired Python interpreter (either a system-wide Python installation or a conda environment).

- VS Code:

Install the "Python" extension from the marketplace.
Open the command palette (Ctrl+Shift+P) and select "Python: Select Interpreter" to choose your Python environment.

6. Writing and executing your first Python program:-

- Now that you've installed Python, you're ready to run your first Python code!
- Open up Terminal if you are in Mac OS or Command Prompt if you are in Windows. (If you haven't used Terminal very much yet, feel free to refer to this chapter in our Learn the Command Line on Terminal course for a refresher on opening the program.)
- Type in python, and you should see information about Python come up with a >>> signaling where to type in your code.
- Next, type in print("hello, world!") and press enter; see what it returns

2.Programming Style:

Understanding Python's PEP 8 guidelines:-

1. **Indentation:**
   Use 4 spaces per indentation level for consistent and readable nesting. Avoid mixing tabs and spaces, as it causes errors**.**
2. **Line Length:**
   Keep lines of code limited to 79 characters for readability, especially when viewing on smaller screens or side-by-side with other code. Comments and docstrings should not exceed 72 characters to ensure clarity in documentation.
3. **Imports:**
   - Import one module per line to maintain clarity and avoid confusion.
   - Organize imports into three sections, separated by blank lines:
     1. Standard library imports.
     2. Third-party library imports.
     3. Local project-specific imports.
   - Prefer absolute imports over relative imports for better maintainability.
4. **Naming:**
   - Use descriptive snake_case for variables and functions to convey their purpose clearly.
   - For classes, use CamelCase to distinguish them as objects or constructs.

- Define constants in ALL_CAPS to indicate that their values should not change.

5. **Whitespace:**
    - Avoid extra spaces within parentheses, brackets, or braces.
    - Use spaces around operators (e.g., +, -, =) for better readability.
    - Avoid trailing whitespace at the end of lines.
7. **Comments:**
   Write meaningful comments to explain why the code exists or behaves a certain way, rather than describing what the code does (which should be clear from the code itself). Use inline comments sparingly and only when necessary.
8. **Docstrings:**
   Use triple quotes for module, class, and function docstrings. Include a concise summary of the purpose and, if necessary, details about parameters and return values.
9. **Exceptions:**
   Handle exceptions explicitly to provide meaningful feedback to users. Use specific exception types instead of generic ones and include helpful error messages to aid debugging.

Indentation, comments, and naming conventions in Python:-

1. Indentation

- Use 4 spaces per indentation level for consistency and readability.
- Never mix tabs and spaces; always prefer spaces.
- Proper indentation is essential in Python as it defines code blocks (e.g., in loops, conditionals, and functions).

2. Comments

- Purpose: Explain the "why" behind the code, not just "what" it does.
- **Types:**
- Inline Comments: Place on the same line as the code. Use sparingly and only when necessary.
- Block Comments: Explain complex code or logic, placed above the relevant code block.
- Start comments with a capital letter and ensure proper punctuation for clarity.
- Avoid redundant comments that merely restate the code.

3. Naming Conventions

- Variables and Functions: Use descriptive names in snake_case (e.g., calculate_sum).
- Classes: Use CamelCase (e.g., MyClass).
- Constants: Use ALL_CAPS with underscores separating words (e.g., MAX_SIZE).

- Private Members: Prefix names with an underscore (e.g., _private_variable).
- Be consistent and choose meaningful names that reflect the purpose of the variable, function, or class.

Writing readable and maintainable code.:-

1.Follow PEP 8 Guidelines:
Adhere to Python's style guide for consistent code formatting, including proper indentation, naming conventions, and line lengths.

2.Use Meaningful Names:
Choose descriptive and self-explanatory names for variables, functions, and classes. Avoid abbreviations or vague names.

3.Write Modular Code:
Break large tasks into smaller, reusable functions or methods. Each function should perform a single, clear task.

4.Use Comments Wisely:

- Explain why something is done, not just what the code does.
- Avoid over-commenting or stating the obvious.

5.Keep Functions Short:
Limit functions to 20–30 lines, focusing on a single responsibility. This improves readability and makes debugging easier.

3.Core Python Concepts

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

1. Integers (int)

- Description: Whole numbers, positive or negative, with no decimal point.
- Examples: -5, 0, 42.
- Use Case: Counting, indexing, or calculations where fractional values aren't needed.

2. Floats (float)

- Description: Numbers with a decimal point, representing real numbers.
- Examples: 3.14, -0.001, 1.0.
- Use Case: Precise measurements, scientific calculations, or any data requiring fractional values.

3. Strings (str)

- Description: Text data enclosed in single ('), double ("), or triple quotes ('''/""").
- Examples: 'Hello', "Python", '''Multiline text'''.
- Use Case: Storing and manipulating text, such as names, messages, or file paths.

4. Lists (list)

- Description: Ordered, mutable collections that can hold items of any data type.
- Examples: [1, 2, 3], ['a', 'b', 'c'], [1, 'Python', 3.14].
- Use Case: Flexible sequences for storing items that may change over time.

5. Tuples (tuple)

- Description: Ordered, immutable collections.
- Examples: (1, 2, 3), ('a', 'b', 'c'), (1, 'Python', 3.14).
- Use Case: Fixed sequences where data should remain constant (e.g., coordinates).

6. Dictionaries (dict)

- Description: Key-value pairs for mapping relationships.
- Examples: {'name': 'Alice', 'age': 25}, {'a': 1, 'b': 2}.
- Use Case: Associative arrays for quick lookups or organizing related data**.**

7. Sets (set)

- Description: Unordered collections of unique elements.
- Examples: {1, 2, 3}, {'a', 'b', 'c'}.
- Use Case: Ensuring uniqueness, set operations (union, intersection, etc.).

Python variables and memory allocation.:-

1. Variables in Python

- Definition: A variable is a name that refers to a value stored in memory.
- Dynamic Typing: Python variables don't require an explicit declaration of type. The type is inferred when a value is assigned:

Naming Rules:

- Must start with a letter or underscore (_).
- Cannot start with a digit.
- Only contain alphanumeric characters and underscores.
- Case-sensitive (myVar ≠ myvar).

2. Memory Allocation in Python

- Python uses references to manage variables and memory.

- When a variable is assigned a value:
- Python creates an object in memory to represent the value.
- The variable is a reference (or pointer) to this object.

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

1. Arithmetic Operators:

- +: Addition
- -: Subtraction
- *: Multiplication
- /: Division
- //: Floor Division (returns the quotient without the remainder)
- %: Modulus (returns the remainder)
- `: `:** Exponentiation (raises a number to a power)

2. Comparison Operators:

- ==: Equal to
- !=: Not equal to
- >: Greater than
- <: Less than
- >=: Greater than or equal to
- <=: Less than or equal to

3. Logical Operators:

- and: Returns True if both statements are true
- or: Returns True if one of the statements is true
- not: Reverses the result, returns False if the result is true

4. Bitwise Operators:

- &: Bitwise AND
- |: Bitwise OR
- ^: Bitwise XOR
- ~: Bitwise NOT
- <<: Left shift
- >>: Right shift

4.Conditional Statements

Introduction to Conditional Statements in Python:-

1. if Statement

- Executes a block of code if the condition is true.

Syntax:

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

2. else Statement

- Executes a block of code when the if condition is false.
- **Syntax**:

```
if condition:
    # Code to execute if condition is true
else:
    # Code to execute if condition is false
```

3. elif Statement (Short for "else if")

- Checks multiple conditions, executing the first one that is true.

Syntax:

```
if condition1:
    # Code to execute if condition1 is true
elif condition2:
    # Code to execute if condition2 is true
else:
    # Code to execute if none of the above conditions are true
```

Nested if-else conditions.:-

A nested if-else statement is when an if or else block contains another if-else block. This allows you to check multiple conditions in a hierarchical manner.

Syntax

```
if condition1:
    if condition2:
```

```
        # Code to execute if both condition1 and condition2 are true
    else:
        # Code to execute if condition1 is true and condition2 is false
else:
    # Code to execute if condition1 is false
```

5.Looping (For, While):-

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 the sequence.

Syntax:

```
for item in sequence:
    # Code to execute for each item
```

2. while Loop

A while loop repeatedly executes a block of code as long as the given condition is true.

Syntax:

```
while condition:
    # Code to execute as long as the condition is true
```

How loops work in Python.:-

Loops in Python are used to repeat a block of code multiple times. The two primary types of loops are the for loop and the while loop. Both types have distinct ways of executing, but they all work through the concept of iteration — repeating actions based on certain conditions.

1. for Loop: Iterating Over a Sequence

A for loop in Python iterates over a sequence (like a list, tuple, string, or range) and executes the block of code once for each element in the sequence.

How it Works:

The for loop picks each item from the sequence one by one.

The loop executes the code block for each item and moves to the next until all items are processed.

Example:

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

for fruit in fruits:
    print(fruit)
```

2. while Loop: Repeating as Long as a Condition is True

A while loop repeats a block of code as long as the condition (an expression) evaluates to True. When the condition becomes False, the loop exits.

How it Works:

- The while loop checks the condition before each iteration.
- If the condition is True, the loop executes the code block.
- If the condition is False, the loop terminates, and the program moves on to the next statement after the loop.

Example:

```
count = 1

while count <= 5:
    print(count)
    count += 1
```

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

1. Looping through Lists

A list is an ordered collection of items that can be modified. You can loop through a list using a for loop to access each element.

Example:

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

for fruit in fruits:
    print(fruit)
```

2. Looping through Tuples

Tuples are similar to lists but are immutable (cannot be modified). You can loop through a tuple in the same way as a list.

Example:

```
colors = ("red", "green", "blue")

for color in colors:
    print(color)
```

3. Looping through Strings

Strings are sequences of characters. You can loop through each character in a string using a for loop.

Example:

```
word = "hello"

for char in word:
    print(char)
```

6.Generators and Iterators

Understanding how generators work in Python:-

Generators in Python are a powerful and memory-efficient way to create iterators. Instead of returning a complete collection all at once, a generator yields items one by one, which can be processed lazily. This is especially useful when working with large datasets, as it doesn't require loading everything into memory at once.

Difference between yield and return.:-

1. Return:

- Purpose: return is used to send a value from a function back to the caller and terminate the function.
- Behavior: When return is executed, the function stops running, and the value is returned to the caller. The function does not retain its state and cannot continue after returning.
- Use Case: return is used in standard functions that need to give back a single result and terminate.

Example of return:

```
def add(a, b):
    return a + b

result = add(3, 4)
print(result)  # Output: 7
```

2. Yield:

- **Purpose**: yield is used in generator functions to return a value, but unlike return, it allows the function to pause and resume execution.
- **Behavior**: When yield is executed, the function's state (variables, execution point) is saved. The function pauses and returns the value to the caller. The function can later be resumed, continuing from where it left off, when the next item is requested (using next() or a loop).
- **Use Case**: yield is used when you need to generate a sequence of values lazily (on demand), especially when working with large datasets or infinite sequences.

Example of yield:

```
python
Copy code
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

counter = count_up_to(3)
for num in counter:
    print(num)
```

Understanding iterators and creating custom iterators.:-

An iterator in Python is any object that implements two methods:

1. __iter__(): This method returns the iterator object itself. It is used to initialize the iterator.
2. __next__(): This method returns the next item in the sequence. When there are no more items to return, it raises a StopIteration exception to signal the end of the iteration.

How Iterators Work:

3. Iteration is the process of accessing elements of a collection (like a list, tuple, or dictionary) one at a time.
4. Iterator Protocol: An object is considered an iterator if it implements both __iter__() and __next__() methods. These methods allow objects to be used in a for loop or with next().

7.Functions and Methods

Defining and calling functions in Python.:-

1. Defining Functions:

To define a function in Python, use the def keyword, followed by the function name, parentheses (which may contain parameters), and a colon (:). The code block that makes up the function is indented.

Syntax:

```
def function_name(parameters):
    # function body
    # optional return statement
```

- Parameters: Variables passed into the function to work with.
- Return: A function can return a value to the caller using the return statement, though it's optional.

2. Calling Functions:

To call a function, simply use its name followed by parentheses. If the function accepts parameters, pass the required values inside the parentheses.

Syntax:

```
function_name(arguments)
```

Function arguments (positional, keyword, default):-

1. Positional Arguments

Positional arguments are the most common type of arguments. The values passed in the function call must match the order of parameters defined in the function.

Explanation:

- The first argument passed corresponds to the first parameter.
- The second argument corresponds to the second parameter, and so on.

Example:

```
def add(a, b):
    return a + b

result = add(3, 5)  # The values 3 and 5 are passed as positional arguments
print(result)  # Output: 8
```

2. Keyword Arguments

Keyword arguments are passed to functions by explicitly specifying the parameter name, making the function call more readable and allowing you to pass arguments in any order.

Explanation:

- You specify the parameter name and the value when calling the function.
- The order of the arguments does not matter because they are matched by name.

Example:

```
def greet(name, greeting):
    print(f"{greeting}, {name}!")

greet(name="Alice", greeting="Hello")  # Output: Hello, Alice!
greet(greeting="Hi", name="Bob")      # Output: Hi, Bob!
```

3. Default Arguments

Default arguments allow you to define a function with parameters that have default values. If no value is provided for those parameters when calling the function, the default value is used.

Explanation:

- Default values are assigned in the function definition.
- If an argument is not passed during the function call, the default value is used.
- Default arguments must appear after non-default arguments in the function signature.

Example:

```
def greet(name="Guest", greeting="Hello"):
    print(f"{greeting}, {name}!")

greet()           # Output: Hello, Guest!
greet("Alice")    # Output: Hello, Alice!
greet("Bob", "Hi")  # Output: Hi, Bob!
```

Scope of variables in Python.:-

In Python, scope refers to the region of a program where a variable is accessible. The scope of a variable determines its visibility and lifetime. There are four main types of variable scopes in Python

1.Local Scope

2.Global Scope

1. Local Scope

A variable has local scope if it is defined inside a function. It is only accessible within that function and ceases to exist after the function completes.

Example:

```
def my_function():
    a = 10  # Local variable
    print(a)  # This will work as 'a' is defined in this function

my_function()  # Output: 10
# print(a)  # This will cause an error because 'a' is not defined outside the function
```

2. Global Scope

A variable has global scope if it is defined outside of any function or class. It is accessible anywhere in the program, both inside and outside functions.

Example:

```
x = 20  # Global variable

def my_function():
    print(x)  # Access global variable inside function

my_function()  # Output: 20
print(x)  # Output: 20
```

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

1.String Methods

- upper(): Converts all characters in the string to uppercase.
- lower(): Converts all characters in the string to lowercase.
- strip(): Removes leading and trailing whitespace.
- replace(old, new): Replaces occurrences of old with new.
- split(delimiter): Splits the string into a list using the specified delimiter.
- find(substring): Returns the index of the first occurrence of substring or -1 if not found.
- count(substring): Returns the number of occurrences of substring.
- join(iterable): Joins elements of an iterable into a single string with the calling string as a separator.

2.List Methods

- append(element): Adds an element to the end of the list.
- extend(iterable): Extends the list by appending elements from another iterable.
- insert(index, element): Inserts an element at a specified index.
- remove(element): Removes the first occurrence of the specified element.
- pop(index): Removes and returns the element at a specified index.
- sort(): Sorts the list in ascending order.
- reverse(): Reverses the order of elements in the list.
- index(element): Returns the index of the first occurrence of an element.
- count(element): Returns the number of occurrences of an element in the list.

3.Dictionary Methods

- get(key): Returns the value for the specified key, or None if the key doesn't exist.
- keys(): Returns a view object containing all the dictionary keys.
- values(): Returns a view object containing all the dictionary values.
- items(): Returns a view object containing all the dictionary key-value pairs.
- update(dictionary): Updates the dictionary with key-value pairs from another dictionary.
- pop(key): Removes and returns the value associated with the specified key.

4.Set Methods

- add(element): Adds an element to the set.
- remove(element): Removes the specified element. Raises an error if the element is not found.
- discard(element): Removes the specified element, but doesn't raise an error if not found.
- union(set): Returns a set containing all elements from both sets.
- intersection(set): Returns a set containing only the common elements between sets.
- difference(set): Returns a set of elements in the first set but not in the second set.

5.Tuple Methods

- count(element): Returns the number of occurrences of the specified element.
- index(element): Returns the index of the first occurrence of the specified element.

8.Control Statements (Break, Continue, Pass):-

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

1. break Statement

- Purpose: Exits the loop entirely when a certain condition is met, regardless of whether the loop has completed all its iterations.
- Use case: When you need to terminate the loop prematurely based on a condition.

2. continue Statement

- Purpose: Skips the current iteration and moves directly to the next iteration of the loop.
- Use case: When you want to skip certain iterations of a loop based on a condition but continue looping.

3. pass Statement

- Purpose: Does nothing and is used as a placeholder in situations where a statement is required syntactically but no action is desired.
- Use case: When you need to create an empty loop or function for future implementation.

9.String Manipulation

Understanding how to access and manipulate strings:-

Accessing Strings

- Indexing: You can access individual characters of a string using zero-based indexing.
- Positive indexing: string[index] (where index starts from 0).
- Negative indexing: string[-index] (where -1 represents the last character).
- Slicing: You can extract a substring using slicing syntax: string[start:end].
- start: Starting index (inclusive).
- end: Ending index (exclusive).
- Example: string[1:5] extracts characters from index 1 to 4.

Manipulating Strings

- Concatenation: You can join two or more strings using the + operator.
- Repetition: You can repeat a string using the * operator.

Common String Methods

- upper(): Converts all characters to uppercase.
- lower(): Converts all characters to lowercase.
- capitalize(): Capitalizes the first letter of the string.
- replace(old, new): Replaces all occurrences of old with new.
- strip(): Removes leading and trailing whitespace.
- split(delimiter): Splits the string into a list of substrings based on the delimiter.
- find(substring): Returns the index of the first occurrence of the substring, or -1 if not found.

String Formatting

- f-strings: A modern and concise way to embed expressions inside string literals using curly braces {}.
- Example: name = "Alice"; greeting = f"Hello, {name}!".
- format(): Allows positional and keyword-based formatting.
- Example: "Hello, {}!".format(name).

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

1. Concatenation

- **Purpose**: Joining two or more strings together.
- **Syntax**: string1 + string2
    - Example: "Hello" + " " + "World" results in "Hello World".

2. Repetition

- **Purpose**: Repeating a string multiple times.
- **Syntax**: string * n, where n is the number of times to repeat.
    - Example: "Hello" * 3 results in "HelloHelloHello".

3. String Methods

- **upper()**: Converts all characters in a string to uppercase.
    - Example: "hello".upper() results in "HELLO".
- **lower()**: Converts all characters in a string to lowercase.
    - Example: "HELLO".lower() results in "hello".
- **capitalize()**: Capitalizes the first letter of the string.
    - Example: "hello".capitalize() results in "Hello".
- **title()**: Capitalizes the first letter of each word in the string.
    - Example: "hello world".title() results in "Hello World".
- **strip()**: Removes leading and trailing whitespace from the string.
    - Example: " hello ".strip() results in "hello".
- **replace(old, new)**: Replaces all occurrences of old with new in the string.
    - Example: "hello world".replace("world", "Python") results in "hello Python".
- **split(delimiter)**: Splits the string into a list using the specified delimiter.
    - Example: "apple,banana,orange".split(",") results in ['apple', 'banana', 'orange'].

String slicing.:-

String slicing allows you to extract substrings from a string using a specified range of indices. The general syntax for slicing is:

string[start:end:step]

Where:

- **start**: The index where the slice starts (inclusive). Defaults to 0 if not specified.
- **end**: The index where the slice ends (exclusive). Defaults to the length of the string if not specified.
- **step**: The step size (or stride), which determines the interval between indices. Defaults to 1 if not specified.

Examples of String Slicing

1. **Basic slicing**:
    - string[start:end]: Extracts characters from start to end-1.
    - Example: "Hello"[1:4] results in "ell".
2. **Omitting start and end**:
    - If you omit start, it defaults to 0.
    - If you omit end, it defaults to the length of the string.
    - Example: "Hello"[:3] results in "Hel".
    - Example: "Hello"[2:] results in "llo".
3. **Using step**:
    - string[start:end:step]: Selects characters with the specified step.
    - Example: "Hello"[::2] results in "Hlo", skipping every second character.
4. **Negative indices**:
    - Negative indices count from the end of the string, with -1 being the last character.
    - Example: "Hello"[-3:-1] results in "ll".

Reversing a string:

- You can reverse a string using slicing with a negative step.
- Example: "Hello"[::-1] results in "olleH".

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

How functional programming works in Python. :-

Functional programming in Python is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. Python, while not a purely functional programming language, supports functional programming features

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

1. map() Function

- Purpose: The map() function applies a given function to all items in an iterable (such as a list or tuple) and returns an iterator that produces the results.
- **Syntax**: map(function, iterable)
- function: The function to apply to each item.
- iterable: The iterable whose items will be processed.
- **Example:** Applying a function to square each number in a list.

```
numbers = [1, 2, 3, 4]

squared_numbers = map(lambda x: x ** 2, numbers)

print(list(squared_numbers))  # Output: [1, 4, 9, 16]
```

- Use case: Transforming data, such as converting strings to integers or performing mathematical operations on a list.

2. filter() Function

- Purpose: The filter() function filters out elements from an iterable based on a condition defined in a function. It returns an iterator containing only the items that satisfy the condition.
- **Syntax:** filter(function, iterable)
  - **function**: The function that defines the condition. It should return True or False.
  - iterable: The iterable to filter.
  - **Example**: Filtering out even numbers from a list.
  - numbers = [1, 2, 3, 4, 5, 6]
  - even_numbers = filter(lambda x: x % 2 == 0, numbers)
  - print(list(even_numbers)) # Output: [2, 4, 6]
  - Use case: Filtering data based on specific criteria, such as extracting all positive numbers or filtering out invalid entries.

3. reduce() Function

- Purpose: The reduce() function (from the functools module) applies a rolling computation to pairs of items in an iterable, reducing them to a single result. It's typically used for cumulative operations, like summing or multiplying all elements.
- **Syntax:** reduce(function, iterable, initial)
- **function**: The function that applies the reduction (takes two arguments and returns a single value).
- iterable: The iterable whose items will be reduced.
- initial: An optional initial value that is used in the first computation. If not provided, the first element of the iterable is used.
- **Example**: Summing all numbers in a list.

```
from functools import reduce


numbers = [1, 2, 3, 4]

sum_result = reduce(lambda x, y: x + y, numbers)

print(sum_result)  # Output: 10
```

Introduction to closures and decorators.:-

1.Closures in Python

A **closure** in Python refers to a function that retains access to variables from its enclosing scope, even after the outer function has finished executing. This allows you to create functions with "memory" of their environment, enabling more flexible and reusable code.

How Closures Work:

1. **Enclosing Scope**: A closure occurs when a function is defined inside another function and references variables from the outer function.
2. **Retaining State**: The inner function retains access to the outer function's local variables after the outer function has returned, forming a closure.

Example of Closure:

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function
```

```
# Create a closure with x = 10
closure = outer_function(10)

# Call the closure with y = 5
print(closure(5))  # Output: 15
```

2.Decorators in Python

A **decorator** is a function that takes another function (or method) as an argument and extends or alters its behavior without modifying the original function. Decorators are used to modify the behavior of functions or methods in a clean and readable way.

How Decorators Work:

1. **Wrapping a Function**: A decorator is typically a function that wraps another function, adding functionality before, after, or around the original function.
2. **Syntax**: In Python, decorators are often applied using the @ syntax before the function definition.

Example of Decorator:

python

Copy code

```
def decorator_function(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

# Applying the decorator
@decorator_function
def say_hello():
    print("Hello!")

# Calling the decorated function
say_hello()
```