PYTHON Assinment-1

Theory

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

Python is a simple, high-level, interpreted programming language widely used in web development, data science, artificial intelligence, automation, and more.

2.History and evolution of Python

Python was created by Guido van Rossum in 1991. It has evolved through versions (Python 2 → Python 3) with improvements in performance and features.

3.Advantages of using Python over other programming languages.

Python is easy to learn, supports multiple programming paradigms, has extensive libraries, strong community support, and is widely used in AI, data science, and web development.

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

Python can be installed directly from python.org, or with distributions like **Anaconda**. IDEs like **PyCharm** and **VS Code** are popular for development.

6.Writing and executing your first Python program.

A basic program is written in .py files and executed using the Python interpreter. Example: printing "Hello, World!".

7. Understanding Python’s PEP 8 guidelines

PEP 8 is the **official style guide for Python code**, created to improve the readability and consistency of Python code. Following PEP 8 makes code easier for you and others to read, maintain, and debug.

PEP 8 Key Points

1. **Indentation & Line Length**
    - 4 spaces per indent, max 79 characters per line.
2. **Blank Lines**
    - 2 blank lines before top-level functions/classes, 1 inside class methods.
3. **Imports**
    - One import per line, order: standard → third-party → local.
4. **Whitespace**
    - Avoid extra spaces: x = 1, y = x + 2.
5. **Naming Conventions**
    - Variables/functions: lower_case_with_underscores
    - Classes: CapitalizedWords
    - Constants: UPPERCASE_WITH_UNDERSCORES
6. **Comments & Docstrings**
    - Explain *why*, not *what*.
    - Docstrings for modules, classes, and functions.
7. **Programming Best Practices**
    - Use is/is not for None, avoid bare except, use with for file handling.
8. **Strings**
    - Single or double quotes, but be consistent.
9. **Tools**
    - Use flake8, black, or pylint to check PEP 8 compliance.

8. Indentation, comments, and naming conventions in Python

1. Indentation

- Use **4 spaces per level** (never tabs).
- Keeps code blocks clear and readable.
- Maintain **consistent indentation** throughout.
- Limit **line length to 79 characters**.

---

2. Comments

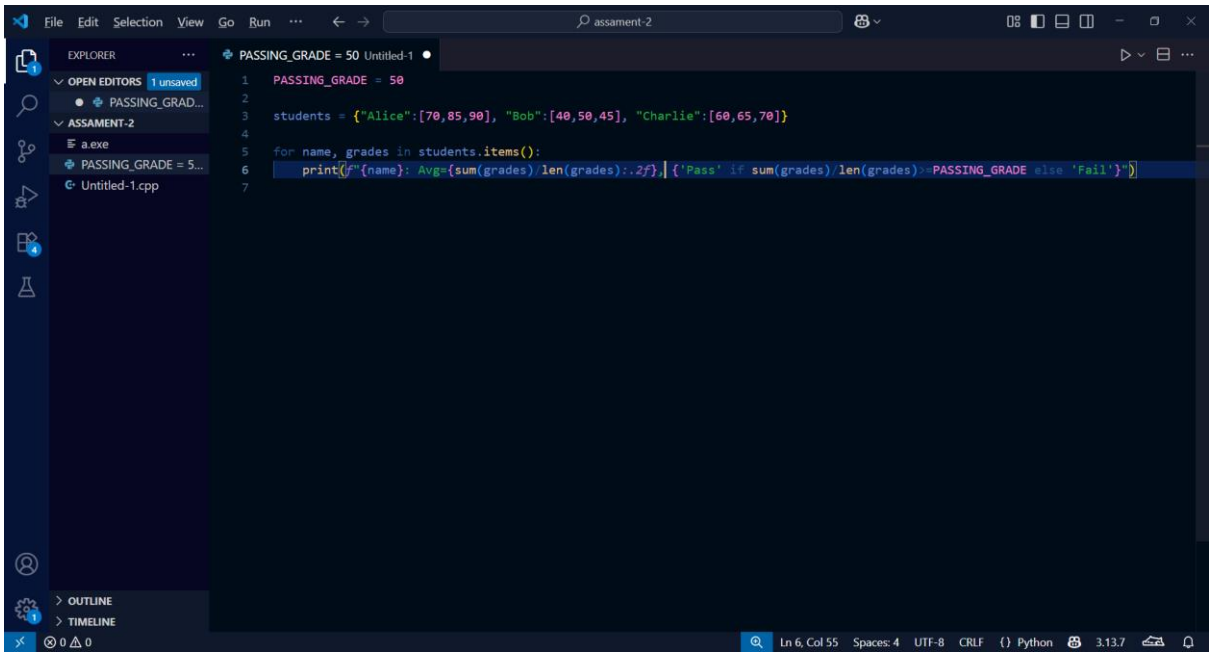
- Explain **why** the code exists, not what it does.
- **Inline comments**: use sparingly.
- **Block comments**: aligned with code, start with #.
- **Docstrings**: use triple quotes for modules, classes, and functions.

---

3. Naming Conventions

- **Variables & Functions:** lowercase with underscores.
- **Classes:** CapitalizedWords.
- **Constants:** UPPERCASE_WITH_UNDERSCORES.
- **Modules & Packages:** lowercase.
- Be **descriptive but concise**.

9. Writing readable and maintainable code



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

1. Integers (int)

- Whole numbers, positive or negative, without decimals.
- Example values: 10, -5, 0
- **Use:** counting, indexing, arithmetic operations.

---

2. Floats (float)

- Numbers with decimal points.
- Example values: 3.14, -0.5, 2.0
- **Use:** precise calculations, measurements, averages.

---

3. Strings (str)

- Text enclosed in single (') or double (") quotes.
- Example values: "Hello", 'Python'
- **Use:** storing names, messages, text processing.

---

4. Lists (list)

- Ordered, **mutable** collections of items.
- Items can be of **different types**.
- Example: [1, 2, 3], ["apple", "banana"]
- **Use:** sequences where you might **add, remove, or change items**.

---

5. Tuples (tuple)

- Ordered, **immutable** collections of items.
- Items can be of **different types**.
- Example: (1, 2, 3), ("a", "b")
- **Use:** fixed collections, like coordinates or multiple return values.

---

6. Dictionaries (dict)

- Unordered, **mutable** collections of **key-value pairs**.
- Keys must be **unique and immutable**.
- Example: {"name": "Alice", "age": 25}
- **Use:** mapping relationships, like storing user info.

---

7. Sets (set)

- Unordered collections of **unique items**.
- Mutable (items can be added/removed).
- Example: {1, 2, 3}, {"apple", "banana"}

- **Use:** remove duplicates, perform mathematical set operations.

11.Python variables and memory allocation.

What is a Variable?

- A **variable** is a **name that refers to a value stored in memory**.
- Variables are used to **store, modify, and retrieve data**.
- In Python, you **don't need to declare the type explicitly**—it is determined automatically (dynamic typing).

Memory Allocation

- Python variables **store references to objects** in memory, not the actual data itself.
- Every object has:
    1. **Type** (int, str, list, etc.)
    2. **Value** (the data it holds)
    3. **Memory address** (where it is stored)
- **Immutable objects** (int, float, string, tuple) cannot be changed after creation.
    - Changing a value creates a **new object in memory**.
- **Mutable objects** (list, dict, set) can be **modified in place** without creating a new object.

Dynamic Typing

- You can **reassign variables to different types** at runtime.

Variable Naming Rules

- Must start with a **letter or underscore**.

- Can contain **letters, digits, and underscores**.
- Case-sensitive (age ≠ Age).
- Avoid using Python **keywords**.

Memory Management in Python

- Python uses **automatic memory management** via **garbage collection**.
- Objects that are **no longer referenced** are automatically removed from memory.
- This reduces **memory leaks** and makes programming easier.

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

1. Arithmetic Operators

Used for **mathematical operations**.

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Floor division
%	Modulus (remainder)
**	Exponentiation

---

2. Comparison Operators

Used to **compare values**, returning **True or False**.

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater or equal
<=	Less or equal

---

3. Logical Operators

Used to **combine conditional statements**.

Operator	Meaning
and	True if **both conditions** are True
or	True if **at least one condition** is True
not	Reverses the logical state

---

4. Bitwise Operators

Operate on **binary representations of integers**.

Operator	Meaning
&	AND
`	`
^	XOR
~	NOT (inverts bits)
<<	Left shift

Operator	Meaning
>>	Right shift

13. Introduction to conditionalstatements: if, else, elif

- Conditional statements allow your program to **execute code only if certain conditions are true**.
- They help **control the flow** of a program.

---

if Statement

- Executes a block of code **only if the condition is True**.

```
if condition:
    # code runs if condition is True
```

---

else Statement

- Provides a block of code to execute **if the condition is False**.

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

---

elif Statement

- Stands for **“else if”**.
- Checks **multiple conditions in sequence**.
- Only the **first True condition** runs.

```
if condition1:
    # runs if condition1 is True
```

```
elif condition2:
    # runs if condition2 is True and condition1 is False
else:
    # runs if none of the above conditions are True
```

14. Nested if-else conditions

1. What is a Nested if-else?

- A **nested if-else** is an if or else statement **inside another if or else block**.
- Allows you to **check multiple levels of conditions**.

---

2. Syntax

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

---

3. Key Points

- Indentation is crucial to **define nested blocks**.
- Can have multiple levels of nesting, but **keep it readable**.
- Each if can have its own elif and else.

Check if a student passed:

1. If score ≥ 50 → Check grade level:
    - ≥ 90 → Excellent
    - ≥ 70 → Good
    - else → Pass
2. Else → Fail

15. Introduction to for and while loops

What are Loops?

- Loops allow you to **repeat a block of code multiple times**.
- Helps **avoid writing repetitive code**.

---

for Loop

- Used to **iterate over a sequence** (like a list, tuple, string, or range).
- Executes the block for **each item in the sequence**.

Key Points:

- Iterates over items one by one.
- Works well when the **number of iterations is known**.

---

while Loop

- Repeats a block of code **as long as a condition is True**.
- Executes until the **condition becomes False**.

Key Points:

- Works well when the **number of iterations is not known in advance**.
- Can lead to **infinite loops** if the condition never becomes False.

Common Loop Keywords

Keyword	Purpose
break	Exit the loop immediately
continue	Skip the current iteration and continue
else	Executes a block if loop ends normally (no break)

16. How loops work in Python.

Concept of Loops

- A loop **repeats a block of code** multiple times.
- Each cycle of repetition is called an **iteration**.
- Python supports two main loops: **for** and **while**.

The for Loop

- Iterates **over a sequence** (list, tuple, string, range, etc.).
- On each iteration, the loop variable takes the **next item** from the sequence.
- Loop stops when the sequence is exhausted.

Works best when the **number of iterations is known**.

The while Loop

- Repeats as long as a **condition is True**.
- On each iteration, the condition is checked:

- If **True** → run the loop body.
- If **False** → exit the loop.

Works best when the **number of iterations is unknown**.

Loop Control Flow

- break → exits the loop immediately.
- continue → skips the current iteration, goes to next.
- else with loops → runs if the loop finishes normally (no break).

Behind the Scenes

- Python loops use **iterators**:
  - A for loop calls iter() on the sequence to get an iterator.
  - Then it repeatedly calls next() to fetch items until StopIteration is raised.
- A while loop just checks a **boolean condition** each time.

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

Lists

- for loop iterates over each element.
- Common use: process items in order.

Tuples

- Work like lists in loops.
- Often used when values are **fixed** (like coordinates).

Sets

- Loops go through items, but **order is not guaranteed**.
- Useful for unique elements.

Dictionaries

- Looping can be done in different ways:
  - Over **keys** (default).
  - Over **values** (.values()).
  - Over **key-value pairs** (.items()).

Strings

- A string is a collection of characters.
- Looping goes through each character one by one.

While Loop with Collections

- Often used with an **index** to access items.
- Example: traverse a list by index until the end.

18. Understanding how generators work in Python

What is a Generator?

- A **generator** is a special type of iterator in Python.
- It allows you to **generate values one at a time** instead of storing them all in memory.
- Very useful for working with **large data** or **infinite sequences**.

How Generators Work

- Created using:
    1. **Generator functions** (use yield instead of return).
    2. **Generator expressions** (like list comprehensions, but with () instead of []).
- When a generator function is called:
    - It doesn't run immediately.
    - It returns a **generator object**.
    - Each time you call next(), the function runs until it hits yield.
    - The value is returned, but the function's state is **saved**.
    - On the next next(), execution continues right after the last yield.

Key Features

- **Lazy evaluation** → values are produced only when needed.
- **Memory efficient** → doesn't store the entire sequence.
- **Can be infinite** → like generating numbers forever.

Difference from Functions

- return → ends the function immediately.
- yield → pauses the function, remembers state, and continues later.

Common Uses

- Reading large files line by line.
- Generating infinite sequences (e.g., Fibonacci numbers).
- Processing data streams.

19. Difference between yield and return.

Feature	return	yield
Function type	Normal function	Generator function
Behavior	Ends function immediately	Pauses and resumes later
Values produced	One final value	Multiple values (one at a time)
State of function	Lost after returning	Saved between calls
Memory usage	Stores all results in memory	Generates results lazily (efficient)
Use case	Single result needed	Sequence of results, large data, streams

20. Understanding iterators and creating custom iterators

What is an Iterator?

- An **iterator** is an object that allows you to **traverse (loop) through elements** one by one.
- Uses two special methods:
    - **__iter__()** → returns the iterator object itself.
    - **__next__()** → returns the next value; raises **StopIteration** when no more items are left.

Iterables vs Iterators

- **Iterable** → an object you can loop over (e.g., list, tuple, string).

- **Iterator** → the actual object that produces elements one at a time.
- You can get an iterator from any iterable using **iter()**.

Creating a Custom Iterator

To build your own iterator:

1. Define a class.
2. Implement __iter__() (returns the iterator).
3. Implement __next__() (defines how to get the next item).

Key Points

- Iterators are **memory efficient** (don't store all items at once).
- Work well with **large or infinite sequences**.
- Python loops (for) automatically call __iter__() and __next__() behind the scenes.

21. Defining and calling functions in Python.

1. What is a function?

A function is a reusable block of code that performs a specific task. It helps avoid repetition and makes programs easier to manage.

---

2. Defining a function

- Use the def keyword followed by the function name and parentheses.
- You can include **parameters** inside the parentheses (optional).
- The function block contains statements that define what the function does.
- You can optionally use a return statement to send a value back.

Syntax:

```
def function_name(parameters):
    statements
    return value  # optional
```

---

3. Calling a function

- A function is **called** by writing its name followed by parentheses.
- If the function requires parameters, pass the values inside the parentheses.
- If the function returns a value, you can store it in a variable.

---

4. Key points

1. Functions must be defined **before** calling them.
2. Parameters are optional.
3. return is optional; if omitted, the function returns None.
4. Functions improve code readability and reusability.

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

1. Positional Arguments

- Arguments are **passed to a function in the order** in which the parameters are defined.
- The first argument goes to the first parameter, the second to the second, and so on.
- Order matters; switching the order may lead to different behavior.

---

2. Keyword Arguments

- Arguments are passed using the **parameter names**.

- Order does not matter because each argument explicitly specifies which parameter it belongs to.
- Makes the function call more readable.

---

3. Default Arguments

- Parameters can have **default values**.
- If a value is not provided for that parameter during the function call, the **default value is used**.
- If a value is provided, it **overrides the default**.

23. Scope of variables in Python

Variable **scope** determines **where in the program a variable can be accessed or modified**.

---

1. Types of variable scope

a) Local Scope

- Variables defined **inside a function**.
- Accessible **only within that function**.
- Created when the function is called and destroyed when the function ends.

---

b) Global Scope

- Variables defined **outside any function**.
- Accessible **anywhere in the program**, including inside functions (but **modifying them inside a function requires global keyword**).

---

c) Enclosing (Nonlocal) Scope

- Applies to **nested functions**.

- Inner functions can access variables of the **enclosing outer function**.
- To modify the outer function's variable inside the inner function, use the nonlocal keyword.

---

d) Built-in Scope

- Python has **built-in names and functions** (like len(), print(), etc.).
- Accessible **anywhere in the program** without defining them.

---

2. LEGB Rule

Python resolves variable names using the **LEGB rule**:

1. **L – Local:** Inside the current function.
2. **E – Enclosing:** Variables in enclosing functions (for nested functions).
3. **G – Global:** Variables at the module level.
4. **B – Built-in:** Python's predefined names and functions.

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

1. String Methods

Python provides many built-in methods to manipulate strings:

- **Case conversion:** upper(), lower(), capitalize(), title(), swapcase()
- **Searching:** find(), index(), startswith(), endswith()
- **Modification:** replace(), strip(), lstrip(), rstrip()
- **Splitting & Joining:** split(), rsplit(), join()
- **Checking content:** isalnum(), isalpha(), isdigit(), isspace(), islower(), isupper()

---

2. List Methods

Lists have built-in methods for adding, removing, and manipulating elements:

- **Adding elements:** append(), extend(), insert()
- **Removing elements:** remove(), pop(), clear()
- **Searching & counting:** index(), count()
- **Sorting & reversing:** sort(), reverse()
- **Copying:** copy()

---

3. Tuple Methods

Tuples are immutable, so only a few methods are available:

- count() – Counts occurrences of a value
- index() – Returns the index of a value

---

4. Set Methods

Sets support methods for adding, removing, and performing set operations:

- **Adding/removing elements:** add(), remove(), discard(), pop(), clear()
- **Set operations:** union(), intersection(), difference(), symmetric_difference()
- **Checking membership:** issubset(), issuperset(), isdisjoint()

---

5. Dictionary Methods

Dictionaries store key-value pairs and have methods to manage them:

- **Access & modification:** get(), keys(), values(), items(), update(), pop(), popitem(), clear()

- **Checking existence:** setdefault()

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

break

- The break statement **terminates the loop immediately**, regardless of the loop condition.
- The program **exits the loop** and continues execution with the first statement **after the loop**.
- Useful when you want to stop looping once a certain condition is met.

---

2. continue

- The continue statement **skips the rest of the code in the current iteration** and moves to the **next iteration of the loop**.
- The loop itself **does not terminate**; only the current iteration is skipped.
- Useful when you want to **ignore certain cases** but continue looping.

---

3. pass

- The pass statement **does nothing**; it is a placeholder.
- Often used **when a statement is syntactically required** but no action needs to be performed.
- Commonly used for **empty functions, classes, or loops** during development.

26. Understanding how to access and manipulate strings

1. Accessing Characters in a String

- Strings are **sequences of characters**, so each character has an **index**.
- **Indexing starts at 0** for the first character.
- You can also use **negative indexing** to access characters from the end (-1 is the last character).

---

2. Slicing Strings

- You can extract a **substring** using slicing:
- string[start:end:step]
  - start → starting index (inclusive)
  - end → ending index (exclusive)
  - step → number of characters to skip (optional)
- Omitting start or end defaults to the beginning or end of the string.

---

3. String Concatenation

- Use the + operator to **join two or more strings** together.
- You can also use the * operator to **repeat a string multiple times**.

---

4. String Methods for Manipulation

- **Changing case:** upper(), lower(), capitalize(), title(), swapcase()
- **Trimming spaces:** strip(), lstrip(), rstrip()
- **Replacing content:** replace()
- **Splitting & joining:** split(), join()
- **Checking content:** isalnum(), isalpha(), isdigit(), isspace()

---

5. Iterating Over Strings

- Strings are **iterable**, so you can loop through each character using a for loop.

---

6. Accessing Characters Dynamically

- You can **use variables as indices** to access characters.
- Useful in loops or functions for **dynamic string processing**.

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

1. Concatenation

- **Joining two or more strings** together using the + operator.
- Creates a **new string**; original strings remain unchanged.

---

2. Repetition

- **Repeating a string multiple times** using the * operator.
- The result is a **new string** consisting of the original string repeated the specified number of times.

---

3. String Methods

Python provides several built-in methods to manipulate strings:

- **Changing case:**
  - upper() → converts all characters to uppercase
  - lower() → converts all characters to lowercase
  - capitalize() → makes the first character uppercase
  - title() → capitalizes the first letter of each word
  - swapcase() → switches uppercase to lowercase and vice versa
- **Trimming spaces:**

    - strip() → removes spaces from both ends
    - lstrip() → removes spaces from the left
    - rstrip() → removes spaces from the right
- **Replacing content:**
    - replace() → replaces a specified substring with another
- **Splitting & joining:**
    - split() → splits a string into a list of substrings
    - join() → joins elements of a list into a string
- **Checking content:**
    - isalnum() → checks if all characters are alphanumeric
    - isalpha() → checks if all characters are alphabetic
    - isdigit() → checks if all characters are digits
    - isspace() → checks if all characters are whitespace

28. String slicing.

1. What is String Slicing?

- **Slicing** is a way to **extract a portion (substring) of a string**.
- It uses **indices** to select characters.
- The original string **remains unchanged**.

---

2. Syntax

string[start:end:step]

- **start** → starting index (inclusive)
- **end** → ending index (exclusive)
- **step** → number of characters to skip (optional; default is 1)

---

3. Key Points

1. **Indexing starts at 0** for the first character.
2. **Negative indices** count from the end of the string (-1 is the last character).
3. Omitting **start** → defaults to the beginning of the string.
4. Omitting **end** → defaults to the end of the string.
5. Omitting **step** → defaults to 1.
6. Using a **negative step** → slices the string in **reverse**.

---

4. Benefits

- Allows **extracting substrings**, **reversing strings**, and **skipping characters** efficiently.
- Works without modifying the original string.

29. How functional programming works in Python.

1. What is Functional Programming?

- Functional programming (FP) is a **programming paradigm** where computation is treated as the **evaluation of mathematical functions**.
- Focuses on **what to do**, not **how to do it**.
- Emphasizes **immutability** and **no side effects**.

---

2. Key Concepts in Functional Programming

1. **First-class functions**
    - Functions are treated as **first-class citizens**, meaning they can be **assigned to variables, passed as arguments, or returned from other functions**.
2. **Pure functions**
    - A pure function always **returns the same output for the same input** and **does not modify external state**.

3. **Higher-order functions**
    - Functions that **accept other functions as arguments** or **return functions**.
4. **Immutability**
    - Data is **not modified**; instead, new data structures are returned.
5. **Function composition**
    - Combining simple functions to build more complex functionality.

---

3. Functional Programming Tools in Python

- **map()** → applies a function to all items in an iterable
- **filter()** → filters elements of an iterable based on a function that returns True or False
- **reduce()** (from functools) → reduces an iterable to a single value using a binary function
- **lambda** → creates small anonymous (unnamed) functions
- **list comprehensions** → a functional-style way to create new lists

---

4. Benefits

- Encourages **clean, concise, and readable code**
- Reduces **side effects**, making code more predictable
- Makes it easier to **parallelize operations**

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

1. map()

- The map() function **applies a given function to each item in an iterable** (like a list, tuple, etc.).
- Returns a **map object**, which can be converted to a list or other iterable.
- Useful when you want to **transform each element** of a dataset.

Key Points:

1. Takes **two arguments**: the function and the iterable.
2. The function can be **named or a lambda**.
3. Original data remains **unchanged**.

---

2. filter()

- The filter() function **selects elements from an iterable based on a condition**.
- Returns a **filter object**, which can be converted to a list or other iterable.
- Useful when you want to **extract only the elements that satisfy a condition**.

Key Points:

1. Takes **two arguments**: a function that returns True or False, and the iterable.
2. Only elements for which the function returns **True** are included.

---

3. reduce()

- The reduce() function (from functools) **applies a function cumulatively to the items of an iterable**, reducing it to a **single value**.
- Useful for **aggregation operations**, like computing the sum, product, or any cumulative result.

Key Points:

1. Takes **two arguments**: a binary function and the iterable.
2. The function takes **two arguments**: the accumulated value and the next item.
3. Returns a **single value**.

---

4. Summary

Function	Purpose	Returns
map()	Transform each element of an iterable	Map object
filter()	Select elements based on a condition	Filter object
reduce()	Combine elements cumulatively into a single value	Single value

31. Introduction to closures and decorators.

1. Closures

Definition

- A **closure** is a **function object that remembers values from its enclosing scope**, even if that scope has finished execution.
- In simpler terms, a closure allows a **nested function** to access variables from the **outer function**, even after the outer function has returned.

Key Points

1. Involves a **nested function**.
2. The nested function **remembers variables** from the outer function.
3. Useful for creating **data hiding**, **encapsulation**, and **factory functions**.

---

2. Decorators

Definition

- A **decorator** is a **function that takes another function as input and returns a new function**, usually extending or modifying its behavior.
- Decorators are often used to **add functionality** to functions or methods **without modifying their code**.

Key Points

1. A decorator **wraps a function** and can run code **before or after** the original function.
2. Can be applied using the **@decorator_name** syntax above a function.
3. Common uses: logging, authentication, caching, timing, and validation.

Relationship

- Decorators often use **closures** internally to **retain access to the original function and its environment**.
- Essentially, **every decorator is a closure**, but not every closure is a decorator.