PLC2 SEE Important Questions

Module 1

1. Explain the different standard data types used in Python programming language with examples.

Python provides a variety of built-in data types that help store different kinds of values efficiently. These data types are categorized as follows:

A) Numeric Types

Python has three numeric types:

Integer (`int`) → Stores whole numbers

```
a = 10

print(type(a))  # Output: <class 'int'>
```

1. **Explanation:** Here, `a` is an integer variable. The `type()` function confirms it belongs to the `int` class.

Floating Point (`float`) → Stores decimal numbers

```
b = 10.5

print(type(b))  # Output: <class 'float'>
```

2. **Explanation:** The variable `b` stores a floating-point number (decimal value).

Complex Numbers (`complex`) → Stores numbers with real and imaginary parts

```
c = 2 + 3j

print(type(c))  # Output: <class 'complex'>
```

3. **Explanation:** Python supports complex numbers, where `2` is the real part and `3j` is the imaginary part.

B) Sequence Types

String (`str`) → A collection of characters

```
text = "Python"

print(type(text))  # Output: <class 'str'>
```

1. **Explanation:** Strings in Python are enclosed in single (`' '`) or double (`" "`) quotes.

List (`list`) → Ordered, mutable collection

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

print(type(fruits))  # Output: <class 'list'>
```

2. **Explanation:** Lists allow storage of multiple values in a single variable, supporting different data types.

Tuple (`tuple`) → Ordered, immutable collection

```
numbers = (1, 2, 3)

print(type(numbers))  # Output: <class 'tuple'>
```

3. **Explanation:** Tuples are similar to lists but cannot be changed after creation.

C) Set Types

Set (`set`) → Unordered, unique values

```
unique_numbers = {1, 2, 3, 4, 4}

print(unique_numbers)  # Output: {1, 2, 3, 4}

print(type(unique_numbers))  # Output: <class 'set'>
```

1. **Explanation:** Sets automatically remove duplicate values and do not maintain order.

D) Mapping Type

Dictionary (`dict`) → Stores key-value pairs
student = {"name": "Alice", "age": 20}

print(type(student)) # Output: <class 'dict'>

1. **Explanation:** A dictionary allows fast access to values using unique keys.

---

2. Illustrate the usage of `str()`, `int()`, and `float()` functions with suitable examples.

Python provides type conversion functions to convert one data type into another.

1) `str()` - Converts to string

x = 100

print(str(x)) # Output: '100'

Explanation: The integer `100` is converted to a string `'100'`, which can now be used in string operations.

---

2) `int()` - Converts to integer

y = "50"

print(int(y)) # Output: 50

Explanation: The string `"50"` is converted to an integer `50`.

---

3) `float()` - Converts to floating-point number

z = "20.5"

```
print(float(z))  # Output: 20.5
```

Explanation: The string `"20.5"` is converted into a float `20.5`.

---

3. Illustrate string concatenation and replication with an example for each.

String Concatenation (+ operator)

Combines two or more strings.

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

Explanation: The + operator joins two strings together.

String Replication (* operator)

Repeats a string multiple times.

```
word = "Python "
print(word * 3)  # Output: Python Python Python
```

Explanation: The * operator repeats a string the specified number of times.

---

4. Differentiate between `break` and `continue` statements in Python.

Statement	Functionality	Example
`break`	Terminates the loop completely	`for i in range(5): if i == 3: break`
`continue`	Skips the current iteration and continues	`for i in range(5): if i == 3: continue`

Example:

```
for i in range(5):
  if i == 3:
    break
  print(i)
# Output: 0 1 2
```

Explanation: The `break` statement exits the loop when `i` is `3`.

```
for i in range(5):
  if i == 3:
    continue
  print(i)
# Output: 0 1 2 4
```

Explanation: The `continue` statement skips `i = 3` but allows the loop to continue.

---

5. Explain the `if/else` statement with its general form and flowchart.

Syntax of `if-else`:

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

Example:

```
num = 10
if num > 0:
  print("Positive")
else:
  print("Negative")
# Output: Positive
```

Explanation: The condition `num > 0` is `True`, so "Positive" is printed.

---

6. Explain the precedence of mathematical operators in Python.

✔️ **Operator Precedence Order:**
1️⃣ **Parentheses `( )`**
2️⃣ **Exponentiation `**`**
3️⃣ **Multiplication `*`, Division `/`, Floor Division `//`, Modulus `%`**
4️⃣ **Addition `+`, Subtraction `-`**

Example:

result = 10 + 2 * 3 - 8 / 2 ** 2

print(result) # Output: 14.0

Explanation:

1. `2 ** 2` → `4`
2. `8 / 4` → `2.0`
3. `2 * 3` → `6`
4. `10 + 6 - 2.0` → `14.0`

Here's the **continuation (Q7 to Q10) with full explanations and example programs** in an **exam-ready format** for **Google Docs**.

---

7. Explain import statements in Python with syntax and examples.

The `import` statement allows us to use functions from external Python modules.

1) Importing an entire module

import math

print(math.sqrt(16)) # Output: 4.0

Explanation: The `math` module is imported, and `math.sqrt(16)` calculates the square root.

---

2) Importing specific functions

from math import sqrt

print(sqrt(25)) # Output: 5.0

Explanation: The `sqrt()` function is directly imported, so we don't need to use `math.sqrt()`.

---

3) Importing with an alias

```
import numpy as np

array = np.array([1, 2, 3])

print(array)
```

Explanation: The `numpy` module is imported as `np`, making it easier to reference.

---

8. Describe local and global scope in Python with examples.

✔ **Local variables** are defined inside a function and cannot be accessed outside of it.
✔ **Global variables** are defined outside a function and can be accessed anywhere.

1) Example of a local variable

```
def local_example():
  x = 10  # Local variable
  print(x)

local_example()

# Output: 10
```

Explanation: The variable `x` is only accessible inside the `local_example()` function.

---

2) Example of a global variable

```
x = 10  # Global variable

def global_example():
  print(x)

global_example()
```

```
# Output: 10
```

Explanation: Since `x` is declared outside, it is accessible inside the function.

---

3) Using `global` keyword

```
x = 10

def modify():
    global x
    x = 20

modify()

print(x)  # Output: 20
```

Explanation: The `global` keyword allows modification of the global variable inside the function.

---

9. Illustrate with examples how to define and call functions in Python.

✔ **Functions** in Python help in code reusability and modularity.

1) Defining and calling a function

```
def greet():
    print("Hello, World!")

greet()

# Output: Hello, World!
```

Explanation: The function `greet()` is defined and then called, which prints the message.

---

2) Function with parameters

```
def add(a, b):
  return a + b
print(add(5, 3))  # Output: 8
```

Explanation: The function `add(a, b)` takes two numbers and returns their sum.

---

3) Function with default parameter

```
def power(base, exponent=2):
  return base ** exponent
print(power(3))  # Output: 9
```

Explanation: If the second argument is not provided, it defaults to 2 (square).

---

10. Distinguish between `if-else`, `if-elif-else`, `for`, and `while` control statements. Also, explain `for` loop with `range()`.

✔ **Comparison Table**

Statement	Description	Example
`if-else`	Executes a block of code based on a condition	`if x > 10: print("Yes") else: print("No")`

`if-elif-else`	Used for multiple conditions	`if x > 10: print("Big") elif x > 5: print("Medium") else: print("Small")`
`for` loop	Iterates over a sequence	`for i in range(5): print(i)`
`while` loop	Repeats while a condition is true	`while x < 10: print(x); x += 1`

Example of `if-elif-else`

num = 15

if num > 20:

  print("Greater than 20")

elif num > 10:

  print("Between 10 and 20")

else:

  print("Less than 10")

Output: Between 10 and 20

Explanation: The correct condition (`num > 10`) is met, so that block executes.

Example of a `for` loop

for i in range(5):

  print(i)

Output: 0 1 2 3 4

Explanation: The `for` loop iterates from `0` to `4`, as `range(5)` generates numbers from `0` to `4`.

Example of `while` loop

```
x = 1

while x <= 5:

  print(x)

  x += 1

# Output: 1 2 3 4 5
```

Explanation: The loop continues until `x` exceeds `5`.

How `for` loop works with `range()` function?

The `range()` function generates a sequence of numbers.

✔ **Example of `range(start, stop, step)`**

```
for i in range(2, 10, 2):

  print(i)
```

Output:

2

4

6

8

✔ **Explanation:**
1️⃣ `start = 2` → Starts from `2`
2️⃣ `stop = 10` → Stops at `10` (not included)
3️⃣ `step = 2` → Increments by `2`

Programs - Module 1

1. Write a program to read the name and age of a person. Display whether the person is a senior citizen or not using `if` and `elif` statements.

```
# Function to check if a person is a senior citizen
def check_senior_citizen(name, age):
    if age ⩾ 60:
        print(name, "is a senior citizen.")
    else:
        print(name, "is not a senior citizen.")

# Taking user input
name = input("Enter your name: ")
age = int(input("Enter your age: "))

# Function call
check_senior_citizen(name, age)
```

Explanation:

✔ The function `check_senior_citizen()` determines if a person is a senior citizen.
✔ It takes `name` and `age` as arguments and prints the result.
✔ **Function is called** after taking input.

---

2. What is exception handling? Write a Python program to handle zero division error.

```
# Function to perform division and handle exceptions
def divide_numbers():
```

```
    try:
        num = int(input("Enter numerator: "))
        den = int(input("Enter denominator: "))
        result = num / den
        print("Result:", result)
    except ZeroDivisionError:
        print("Error! Division by zero is not allowed.")

# Function call
divide_numbers()
```

✔ **Exception handling ensures** that dividing by zero does not crash the program.
✔ **`try-except` block** catches errors and provides a friendly message.

---

3. Write Python programs for the following:

i) Convert degrees Fahrenheit to Celsius

```
# Function to convert Fahrenheit to Celsius
def fahrenheit_to_celsius(fahrenheit):
    return (fahrenheit - 32) * 5/9

# Taking input
fahrenheit = float(input("Enter temperature in Fahrenheit: "))
print("Temperature in Celsius:", fahrenheit_to_celsius(fahrenheit))
```

✔ Uses a **function** for better modularity.

---

ii) Find the factorial of a number

```
# Function to calculate factorial
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Taking input and calling function
```

```
num = int(input("Enter a number: "))
print("Factorial of", num, "is", factorial(num))
```

✔ The function **`factorial(n)`** calculates and returns the factorial of n.

---

4. What are nested loops? Write a Python program to display all permutations of a three-letter word.

```
# Function to generate permutations
def generate_permutations(word):
    for i in word:
        for j in word:
            for k in word:
                if i ≠ j and j ≠ k and i ≠ k:
                    print(i+j+k)

# Taking input
word = input("Enter a 3-letter word: ")
generate_permutations(word)
```

✔ **Nested loops** allow iteration over all character combinations.

---

5. Develop a Python program that prints numbers from 1 to 10 using a `for` loop and a `while` loop.

i) Using a `for` loop

```
# Function to print numbers using for loop
def print_numbers_for():
    for i in range(1, 11):
        print(i)

print_numbers_for()
```

---

ii) Using a `while` loop

```
# Function to print numbers using while loop
def print_numbers_while():
    num = 1
    while num ⩽ 10:
        print(num)
        num += 1

print_numbers_while()
```

✔ Functions **improve reusability** and keep the main code clean.

---

6. Write a program for the following:

i) Add n numbers accepted from the user

```
# Function to calculate sum of n numbers
def sum_of_numbers(n):
    total = 0
    for i in range(n):
        num = float(input("Enter a number: "))
        total += num
    return total

n = int(input("Enter how many numbers: "))
print("Total sum:", sum_of_numbers(n))
```

✔ The function **accepts n numbers** and **returns their sum**.

---

ii) Find the max of two numbers using a function

```
# Function to find maximum of two numbers
def max_of_two(a, b):
    return max(a, b)

num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

print("Maximum is:", max_of_two(num1, num2))
```

✔ Uses the **built-in `max()` function** for efficiency.

---

7. Write a program to realize Rock, Paper, and Scissors game with a count of wins, losses, and draws.

```
import random

# Function to play Rock-Paper-Scissors
def rock_paper_scissors():
    choices = ["rock", "paper", "scissors"]
    wins, losses, draws = 0, 0, 0

    while True:
        user_choice = input("Enter rock, paper, or scissors (or
'quit' to stop): ").lower()

        if user_choice == "quit":
            break

        if user_choice not in choices:
            print("Invalid choice! Try again.")
            continue

        computer_choice = random.choice(choices)
        print("Computer chose:", computer_choice)

        if user_choice == computer_choice:
            print("It's a draw!")
            draws += 1
        elif (user_choice == "rock" and computer_choice ==
"scissors") or \
             (user_choice == "scissors" and computer_choice ==
"paper") or \
             (user_choice == "paper" and computer_choice == "rock"):
            print("You win!")
            wins += 1
        else:
            print("You lose!")
```

```
            losses += 1

    print("\nFinal Score:")
    print("Wins:", wins, "Losses:", losses, "Draws:", draws)

# Function call
rock_paper_scissors()
```

✔ The function `rock_paper_scissors()` handles all logic, making it clean and reusable.

---

8. Write Python programs for the following:

i) Convert temperature from Celsius to Fahrenheit

Hint: C=(F−32)×59C = \frac{(F-32) \times 5}{9}

```
# Function to convert Celsius to Fahrenheit
def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32

# Taking input
celsius = float(input("Enter temperature in Celsius: "))
print("Temperature in Fahrenheit:", celsius_to_fahrenheit(celsius))
```

✔ **Formula used:**

F=(C×95)+32F = (C \times \frac{9}{5}) + 32

✔ **Explanation:**

- The function `celsius_to_fahrenheit()` converts Celsius to Fahrenheit.
- It takes `celsius` as input, applies the formula, and returns the Fahrenheit temperature.

---

ii) Print "Welcome to SIT" Exactly Five Times

(1) Using a `for` loop:

```
# Function to print message 5 times using for loop
def print_message_for():
    for i in range(5):
```

```
        print("Welcome to SIT")

print_message_for()
```

(2) Using a `while` loop:

```
# Function to print message 5 times using while loop
def print_message_while():
    count = 0
    while count < 5:
        print("Welcome to SIT")
        count += 1

print_message_while()
```

✔ **Explanation:**

- The function **`print_message_for()`** uses a **`for` loop** to print the message 5 times.
- The function **`print_message_while()`** uses a **`while` loop** to print the message 5 times.

---

iii) Add `n` numbers accepted from the user and print the result

```
# Function to calculate the sum of n numbers
def sum_of_n_numbers(n):
    total = 0
    for i in range(n):
        num = float(input("Enter a number: "))
        total += num
    return total

# Taking input and calling function
n = int(input("Enter how many numbers: "))
print("Total sum:", sum_of_n_numbers(n))
```

✔ **Explanation:**

- The function **`sum_of_n_numbers(n)`** takes `n` numbers as input, adds them, and returns the sum.
- A `for` loop is used to accept numbers one by one and accumulate the sum.

---

9. Number Guessing Game

```
import random

# Function to guess the number
def guess_number():
    target = random.randint(1, 100)  # Random number between 1 and
100
    guesses = 0
    guess = 0

    print("Guess a number between 1 and 100!")

    while guess ≠ target:
        guess = int(input("Enter your guess: "))
        guesses += 1

        if guess < target:
            print("Too low! Try again.")
        elif guess > target:
            print("Too high! Try again.")
        else:
            print(f"Congratulations! You guessed the correct number
in {guesses} attempts.")

# Function call
guess_number()
```

✔ **Explanation:**

- The program generates a **random number** between **1 and 100**.
- The user keeps **guessing** until they get the correct answer.
- The program **counts the number of attempts** and prints the result.

✔ **Sample Output:**

Guess a number between 1 and 100!
Enter your guess: 30
Too low! Try again.
Enter your guess: 50
Too high! Try again.
Enter your guess: 40
Congratulations! You guessed the correct number in 3 attempts.

---

10. Magic 8-Ball Program using `if` statements and `randint()`

```
import random

# Function to generate Magic 8-ball response
def magic_8_ball():
    responses = [
        "Yes, definitely!",
        "It is certain.",
        "Without a doubt.",
        "Ask again later.",
        "Better not tell you now.",
        "Cannot predict now.",
        "Don't count on it.",
        "My sources say no.",
        "Outlook not so good.",
        "Very doubtful."
    ]

    input("Ask the Magic 8-ball a yes/no question: ")  # User asks a
question
    print("Thinking...\n")

    # Randomly select a response
    print("Magic 8-Ball says:", random.choice(responses))

# Function call
magic_8_ball()
```

✔ **Explanation:**

- The function `magic_8_ball()` randomly selects a response from a **list of 10 possible answers**.
- The user **asks a question**, and the program **randomly picks a response**.

✔ **Sample Output:**

Ask the Magic 8-ball a yes/no question: Will I pass my exams?
Thinking...

Magic 8-Ball says: It is certain.

Module 2

Lists

1. Explain the working of any four methods associated with the list data type.

Lists in Python have several built-in methods. Here are four commonly used ones:

(i) append() - Adds an element to the end of a list.

```
numbers = [1, 2, 3]
numbers.append(4)
print(numbers)  # Output: [1, 2, 3, 4]
```

(ii) insert() - Inserts an element at a specified index.

```
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "mango")
print(fruits)  # Output: ['apple', 'mango', 'banana', 'cherry']
```

(iii) remove() - Removes the first occurrence of a value.

```
colors = ["red", "blue", "green"]
colors.remove("blue")
print(colors)  # Output: ['red', 'green']
```

(iv) pop() - Removes and returns an element at a specified index (default is last element).

```
letters = ["a", "b", "c"]
last_letter = letters.pop()
print(letters)  # Output: ['a', 'b']
print(last_letter)  # Output: 'c'
```

---

2. What are mutable and immutable data types? Give examples.

- **Mutable Data Types**: Can be changed after creation.
    - Example: **Lists, Dictionaries, Sets**

```
my_list = [1, 2, 3]
my_list[0] = 100  # List is modified
print(my_list)  # Output: [100, 2, 3]
```

- **Immutable Data Types**: Cannot be changed after creation.
    - Example: **Tuples, Strings, Integers, Booleans**

```
my_tuple = (1, 2, 3)
my_tuple[0] = 100  # This will cause an error
```

---

3. Difference between Lists and Tuples with examples.

Feature	List	Tuple
Mutability	Mutable	Immutable
Syntax	[ ]	( )
Performance	Slower (modifiable)	Faster (fixed)
Methods	More built-in methods	Fewer methods

Example:

```
# List (Mutable)
my_list = [1, 2, 3]
my_list.append(4)  # Works fine
print(my_list)  # Output: [1, 2, 3, 4]

# Tuple (Immutable)
my_tuple = (1, 2, 3)
my_tuple.append(4)  # Causes an error
```

---

4. Explain how to assign and copy data in Python lists.

- **Assignment (`=`)**: Both lists reference the same memory location.
- **Shallow Copy (`copy()` or `[:]`)**: Creates a new list but still references sub-elements.
- **Deep Copy (`copy.deepcopy()`)**: Copies entire structure, including sub-elements.

```
import copy

# Assignment (same reference)
list1 = [1, 2, 3]
list2 = list1
list2.append(4)
print(list1)  # Output: [1, 2, 3, 4]  (Both lists are modified)

# Shallow Copy
list1 = [1, 2, [3, 4]]
list2 = list1.copy()  # or list2 = list1[:]
list2[2].append(5)
print(list1)  # Output: [1, 2, [3, 4, 5]]  (Sub-list is affected)

# Deep Copy
list1 = [1, 2, [3, 4]]
list2 = copy.deepcopy(list1)
list2[2].append(5)
print(list1)  # Output: [1, 2, [3, 4]]  (Deep copy prevents modification)
```

---

5. Explain the augmented assignment operators with examples.

Augmented operators **modify a variable in place**.

Operator	Example	Equivalent to
`+=`	`x += 5`	`x = x + 5`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 2`	`x = x * 2`
`/=`	`x /= 4`	`x = x / 4`

Example:

```
x = 10
x += 5  # Equivalent to x = x + 5
```

```
print(x)  # Output: 15
```

---

6. Difference between lists and tuples, and unsupported tuple functions.

Feature	List	Tuple
Mutability	Mutable	Immutable
Methods	`append()`, `remove()`, `sort()`	Only `count()` and `index()`
Memory Usage	More	Less (optimized)

Functions **not supported** by tuples:

- `append()`
- `remove()`
- `sort()`
- `reverse()`

Example:

```
my_tuple = (1, 2, 3)
my_tuple.append(4)  # Error: Tuples do not support append()
```

---

7. List operations with examples:

```
import random

# Indexing
nums = [10, 20, 30, 40]
print(nums[2])  # Output: 30

# Slicing
print(nums[1:3])  # Output: [20, 30]

# Random choice
print(random.choice(nums))  # Output: Randomly selects an element

# Shuffle
random.shuffle(nums)
```

```
print(nums)  # Output: Randomly shuffled list

# Append and Remove
nums.append(50)
print(nums)  # Output: [shuffled elements + 50]
nums.remove(20)
print(nums)  # Removes 20 from list

# Sorting
nums.sort()
print(nums)  # Output: Sorted list
```

---

8. Explain the Following List Methods with Illustrative Example Code

Python provides several built-in methods to manipulate lists. Below are three important list methods with explanations and example code:

---

i) `append()` Method

Definition:

The `append()` method **adds an element to the end of the list**. It modifies the original list and does not return a new list.

Syntax:

```
list.append(element)
```

Example Code:

```
# Creating a list
numbers = [1, 2, 3]

# Appending an element
numbers.append(4)

print(numbers)  # Output: [1, 2, 3, 4]
```

Explanation:

- The list starts as `[1, 2, 3]`.
- `append(4)` adds the number `4` to the **end** of the list.
- The final output is `[1, 2, 3, 4]`.

---

ii) `insert()` Method

Definition:

The `insert()` method **adds an element at a specific position (index) in the list**.

Syntax:

list.insert(index, element)

- `index`: The position where the element should be inserted.
- `element`: The item to be inserted.

Example Code:

```
# Creating a list
fruits = ["apple", "banana", "cherry"]

# Inserting an element at index 1
fruits.insert(1, "orange")

print(fruits)  # Output: ['apple', 'orange', 'banana', 'cherry']
```

Explanation:

- The list starts as `["apple", "banana", "cherry"]`.
- `insert(1, "orange")` inserts `"orange"` at index **1**.
- `"banana"` and `"cherry"` shift to the right.
- The final output is `['apple', 'orange', 'banana', 'cherry']`.

---

iii) `remove()` Method

Definition:

The `remove()` method **removes the first occurrence** of a specified element from the list.

Syntax:

list.remove(element)

Example Code:

```
# Creating a list
numbers = [10, 20, 30, 40, 20]

# Removing the first occurrence of 20
numbers.remove(20)

print(numbers)  # Output: [10, 30, 40, 20]
```

Explanation:

- The list starts as `[10, 20, 30, 40, 20]`.
- `remove(20)` removes the **first occurrence** of `20` (not both).
- The final output is `[10, 30, 40, 20]`.

Summary Table

Method	Purpose	Example
`append()`	Adds an element to the end of the list	`list.append(5)`
`insert()`	Adds an element at a specific index	`list.insert(1, "orange")`
`remove()`	Removes the first occurrence of an element	`list.remove(10)`

Tuples, Dictionaries & Structuring Data

1. Differentiate between Lists, Tuples, and Dictionaries.

Feature	List ([ ])	Tuple (( ))	Dictionary ({key: value})
Mutability	Mutable (modifiable)	Immutable (fixed)	Mutable (modifiable)
Indexing	Supports indexing	Supports indexing	Accessed by keys
Order	Ordered	Ordered	Ordered (Python 3.7+)
Duplicates	Allows duplicates	Allows duplicates	Keys must be unique
Performance	Slower (modifiable)	Faster (fixed)	Slightly slower (key-value lookup)
Usage	When a collection of items is needed	When a fixed set of values is required	When key-value pairs are required

Example Code

```
# List
my_list = [1, 2, 3, 4]

# Tuple
my_tuple = (1, 2, 3, 4)

# Dictionary
my_dict = {"name": "Alice", "age": 25}
```

---

2. What are mutable and immutable data types? Give examples.

Mutable Data Types (Can be changed after creation)

- **Examples:** Lists, Dictionaries, Sets

Example Code:

```
 my_list = [1, 2, 3]
my_list.append(4)  # List is modified
print(my_list)  # Output: [1, 2, 3, 4]
```

-

Immutable Data Types (Cannot be changed after creation)

- **Examples:** Tuples, Strings, Integers, Booleans

Example Code:
```
 my_tuple = (1, 2, 3)
my_tuple[0] = 100  # Error: Tuple does not support item assignment
```

-

---

3. Explain the syntax of the tuple data type with an example. How do you convert lists to tuples and vice versa?

Tuple Syntax:

- Defined using **parentheses** `()`.
- **Immutable** (Cannot be changed after creation).

Example of Tuple Creation:
```
# Creating a tuple
my_tuple = (10, 20, 30, "Python")
print(my_tuple)  # Output: (10, 20, 30, 'Python')
```

Conversion Between Lists and Tuples:

Convert List to Tuple:
```
my_list = [1, 2, 3]
my_tuple = tuple(my_list)
print(my_tuple)  # Output: (1, 2, 3)
```

Convert Tuple to List:
```
my_tuple = (4, 5, 6)
my_list = list(my_tuple)
print(my_list)  # Output: [4, 5, 6]
```

---

4. Discuss the functions that are not supported by tuples.

Unlike lists, **tuples do not support modification**. The following functions are **not available** for tuples:

Unsupported Functions	Reason
`append()`	Cannot add elements to a tuple
`remove()`	Cannot remove elements from a tuple
`insert()`	Cannot insert elements
`pop()`	Cannot remove an element
`sort()`	Cannot sort elements

Example (Unsupported Operation on Tuples)

```
my_tuple = (1, 2, 3)
my_tuple.append(4)  # Error: AttributeError: 'tuple' object has no attribute 'append'
```

---

5. Discuss four built-in functions of the dictionary data type.

Dictionaries provide several built-in functions for accessing and manipulating data.

Function	Description	Example
`keys()`	Returns all keys in the dictionary	`my_dict.keys()`
`values()`	Returns all values	`my_dict.values()`
`items()`	Returns key-value pairs as tuples	`my_dict.items()`
`get()`	Fetches value for a key without error	`my_dict.get("age")`

Example Code:

```
# Creating a dictionary
my_dict = {"name": "Alice", "age": 25, "city": "Bangalore"}

# keys()
print(my_dict.keys())  # Output: dict_keys(['name', 'age', 'city'])
```

```
# values()
print(my_dict.values())  # Output: dict_values(['Alice', 25, 'Bangalore'])

# items()
print(my_dict.items())  # Output: dict_items([('name', 'Alice'), ('age', 25), ('city', 'Bangalore')])

# get()
print(my_dict.get("name"))  # Output: Alice
```

---

6. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.keys()`?

If a dictionary is stored in `spam`, the two expressions behave slightly differently:

Expression	Meaning
`'cat' in spam`	Checks if `'cat'` exists as a key in `spam`.
`'cat' in spam.keys()`	Explicitly checks if `'cat'` is in the dictionary's keys (same result as above, but more explicit).

Example:

```
spam = {'cat': 5, 'dog': 7}

print('cat' in spam)  # Output: True
print('cat' in spam.keys())  # Output: True
```

✔ **Both expressions return `True` because "cat" is a key in `spam`.**
✔ The first method is **shorter** and **preferred** in Pythonic code.

Programs - Module 2

1. Write a program to read N numbers from the console and create a list. Calculate and display mean, variance, and standard deviation of these numbers.

Code:

```
import math

# Read numbers from user
n = int(input("Enter the number of elements: "))
numbers = [float(input(f"Enter number {i+1}: ")) for i in range(n)]

# Calculate Mean
mean = sum(numbers) / n

# Calculate Variance
variance = sum((x - mean) ** 2 for x in numbers) / n

# Calculate Standard Deviation
std_dev = math.sqrt(variance)

# Display Results
print(f"Mean: {mean}")
print(f"Variance: {variance}")
print(f"Standard Deviation: {std_dev}")
```

Explanation:

- **Mean**: Sum of all numbers divided by count.
- **Variance**: Measures how much the numbers deviate from the mean.
- **Standard Deviation**: Square root of variance (shows data spread).

---

2. Show the output of the following code:

```
def main():
    d = {"red": 4, "blue": 1, "green": 14, "yellow": 2}
    print(d["red"])  # Output: 4
```

```
    print(list(d.keys()))  # Output: ['red', 'blue', 'green',
'yellow']
    print(list(d.values()))  # Output: [4, 1, 14, 2]
    print("blue" in d)  # Output: True
    print("purple" in d)  # Output: False
    d["blue"] += 10
    print(d["blue"])  # Output: 11

main()
```

Explanation:

- The dictionary stores color counts.
- `d["red"]` retrieves **4**.
- `list(d.keys())` gives `['red', 'blue', 'green', 'yellow']`.
- `list(d.values())` gives `[4, 1, 14, 2]`.
- `"blue" in d` checks if "blue" is a key (**True**).
- `"purple" in d` checks if "purple" exists (**False**).
- `d["blue"] += 10` updates **blue** count to **11**.

---

3. Develop a program to generate the Fibonacci series.

Code:

```
def fibonacci(n):
    fib_series = [0, 1]
    for i in range(2, n):
        fib_series.append(fib_series[i-1] + fib_series[i-2])
    return fib_series

n = int(input("Enter the number of terms: "))
print(f"Fibonacci Series: {fibonacci(n)}")
```

Explanation:

- Starts with `[0, 1]`.
- Each term is sum of the **previous two**.
- Uses a loop to generate `n` terms.

---

4. Develop a Python program that stores friends' birthdays using a dictionary and updates it whenever a new birthday is encountered.

Code:

```
birthdays = {}

while True:
    name = input("Enter friend's name (or 'exit' to stop): ")
    if name.lower() == 'exit':
        break
    if name in birthdays:
        print(f"{name}'s birthday: {birthdays[name]}")
    else:
        dob = input(f"Enter birthday for {name} (DD-MM-YYYY): ")
        birthdays[name] = dob
        print("Birthday added!")

print("Final Birthday Dictionary:", birthdays)
```

Explanation:

- Uses a **dictionary** to store birthdays.
- Checks if a name exists; if not, it adds a new entry.

---

5. Write a Magic 8-Ball program using the list data type.

Code:

```
import random

responses = [
    "Yes, definitely!", "No, not now.", "Ask again later.",
    "Absolutely!", "Very doubtful.", "It is certain.", "Better not
tell you now."
]

input("Ask the Magic 8 Ball a Yes/No question: ")
print("Magic 8 Ball says:", random.choice(responses))
```

Explanation:

- Uses `random.choice()` to pick a response from the list.
- Simulates a **Magic 8-Ball** with pre-defined answers.

---

6. Predict the output of the following code:

```
mylist = [1, 4, 7, 9, 2, 6, 9, 2, 10, 5, 3]

for i in range(1, 8):
    mylist[i] = mylist[i-1]

print(mylist)
```

Output:

[1, 1, 1, 1, 1, 1, 1, 1, 10, 5, 3]

Explanation:

- The loop replaces **each element** in range `1 to 7` with the previous element.
- The first element (`mylist[0] = 1`) remains unchanged.
- The values from `mylist[1]` to `mylist[7]` become `1`.

Module 3 String Manipulation

1. Explain the usage of the following built-in string methods.

i) `join()` and `split()`

- `join()`: Joins elements of a list into a single string.
- `split()`: Splits a string into a list of words.

Example Code:

```
# join() Example
words = ["Python", "is", "awesome"]
joined_string = " ".join(words)
print(joined_string)  # Output: "Python is awesome"

# split() Example
sentence = "Learning Python is fun"
split_list = sentence.split()
print(split_list)  # Output: ['Learning', 'Python', 'is', 'fun']
```

Explanation:

- `join()` **concatenates** a list into a string using a separator.
- `split()` **divides** a string into a list using spaces or a specific delimiter.

---

ii) `ljust()`, `rjust()`, `center()`

- Used for **text alignment** in strings.

Example Code:

```
text = "Python"

# Left justify
print(text.ljust(10, '-'))  # Output: "Python----"

# Right justify
print(text.rjust(10, '-'))  # Output: "----Python"

# Center align
print(text.center(10, '-'))  # Output: "--Python--"
```

Explanation:

- `ljust(width, char)`: Aligns text to the left.
- `rjust(width, char)`: Aligns text to the right.
- `center(width, char)`: Aligns text to the center.

---

iii) `strip()`, `rstrip()`, and `lstrip()`

- Used to **remove whitespaces** or specific characters.

Example Code:

```
text = "  Hello Python  "

print(text.strip())   # Output: "Hello Python"
print(text.rstrip())  # Output: "  Hello Python"
print(text.lstrip())  # Output: "Hello Python  "
```

Explanation:

- `strip()`: Removes spaces from **both** ends.
- `rstrip()`: Removes spaces from the **right**.
- `lstrip()`: Removes spaces from the **left**.

---

iv) `startswith()` and `endswith()`

- Used to check if a string **starts or ends** with a specific substring.

Example Code:

```
text = "Hello, Python!"

print(text.startswith("Hello"))  # Output: True
print(text.endswith("Python!"))  # Output: True
```

Explanation:

- `startswith(substring)`: Checks if the string **begins** with `substring`.
- `endswith(substring)`: Checks if the string **ends** with `substring`.

---

v) `upper()` and `lower()`

- Used to **change case** of text.

Example Code:

```
text = "Python Programming"

print(text.upper())  # Output: "PYTHON PROGRAMMING"
print(text.lower())  # Output: "python programming"
```

Explanation:

- `upper()`: Converts text to **uppercase**.
- `lower()`: Converts text to **lowercase**.

---

2. Explain splitting strings with the `partition()` method with an example.

- The `partition()` method **splits a string into three parts**:
    1. **Before the separator**
    2. **Separator itself**
    3. **After the separator**

Example Code:

```
text = "Python is amazing"

result = text.partition("is")
print(result)  # Output: ('Python ', 'is', ' amazing')
```

Explanation:

- `partition("is")` method **splits** the string at `"is"` into three parts:
    - **Before "is"** → `"Python "`
    - **Separator** → `"is"`
    - **After "is"** → `" amazing"`

---

3. Briefly explain the function of the `pyperclip` module in Python with suitable code.

- The `pyperclip` module allows **copying and pasting text to/from the clipboard**.

Example Code:

```
import pyperclip

# Copy text to clipboard
pyperclip.copy("Hello, Python!")

# Paste text from clipboard
print(pyperclip.paste())  # Output: "Hello, Python!"
```

Explanation:

- `pyperclip.copy(text)`: Copies `text` to the clipboard.
- `pyperclip.paste()`: Retrieves the copied text from the clipboard.

---

4. Distinguish between different steps involved in the project "Adding Bullets to Wiki Markup".

Steps in Adding Bullets to Wiki Markup

This project is used to **add bullets (`*`) at the beginning of each line** in a text block.

Steps:

1. **Take Input:**
    - Accepts a **multi-line string** from the user.
2. **Split the Text:**
    - The `split("\n")` method is used to break the text into **individual lines**.
3. **Modify Each Line:**
    - Each line is prefixed with `"* "` to create a bullet point.
4. **Join and Display Output:**
    - The modified lines are combined using `join("\n")` and printed.

Example Code:

```
text = """Python is fun
Coding is interesting
AI is the future"""

# Split text into lines
lines = text.split("\n")

# Add bullets
bulleted_lines = ["* " + line for line in lines]

# Join the lines back
result = "\n".join(bulleted_lines)

print(result)
```

Output:

```
* Python is fun
* Coding is interesting
```

* AI is the future

Explanation:

- The input text is split into multiple lines.
- `"* "` is added before each line to create a bullet point.
- The modified lines are joined and displayed as a formatted list.

Programs - Module 3

1. Develop a Python program that repeatedly asks users for their age and a password until they provide valid input (Use `isX()` string method).

Program:

```
while True:
    age = input("Enter your age: ")

    # Check if age contains only digits
    if age.isdigit():
        age = int(age)
        break
    else:
        print("Invalid input! Age should be a number.")

while True:
    password = input("Enter a password: ")

    # Check if password is alphanumeric
    if password.isalnum():
        print("Valid password and age entered successfully!")
        break
    else:
        print("Invalid password! It should contain only letters and
numbers.")
```

Explanation:

- The program **repeatedly asks** for valid inputs.
- `isdigit()` ensures that the age contains **only numbers**.
- `isalnum()` ensures that the password **contains only letters and numbers**.

---

2. What do the following expressions evaluate to?

Expression	Evaluated Output	Explanation
`'Hello, world!'[1]`	`'e'`	Retrieves the character at index **1** (0-based indexing).
`'Hello, world!'[0:5]`	`'Hello'`	Extracts substring from index **0 to 4** (5 is excluded).
`'Hello, world!'[:5]`	`'Hello'`	Extracts substring from the **start** to index 4.
`'Hello, world!'[3:]`	`'lo, world!'`	Extracts substring from index **3 to the end**.

---

3. Develop a program that counts the number of occurrences of each letter in a string.

Program:

```
def count_letters(text):
    letter_count = {}

    for char in text:
        if char.isalpha():  # Consider only letters
            char = char.lower()  # Convert to lowercase
            letter_count[char] = letter_count.get(char, 0) + 1

    return letter_count

# Input from the user
text = input("Enter a string: ")

# Count letters
result = count_letters(text)
```

```
# Display the result
print("Letter frequency in the string:")
for letter, count in result.items():
    print(f"{letter}: {count}")
```

Explanation:

- The program **iterates over each character** in the string.
- It **ignores non-alphabetic characters** using `isalpha()`.
- It converts the character to **lowercase** for uniform counting.
- It stores the count of each letter in a **dictionary**.
- Finally, the letter frequency is **printed**.

Sample Output:

Enter a string: Python Programming
Letter frequency in the string:
p: 2
y: 1
t: 1
h: 1
o: 2
n: 2
r: 2
g: 2
a: 1
m: 2
i: 1

Here's a well-structured document with detailed answers that you can use for Google Docs:

Module 4 - Reading, Writing and Organising Files

1. Differentiate between Absolute and Relative Path Names. Give Examples.

Absolute Path:

- An absolute path specifies the complete location of a file or folder from the root directory.
- It remains the same regardless of the current working directory.

Example:

- Windows: `C:\Users\Chethan\Documents\file.txt`
- Linux/Mac: `/home/chethan/Documents/file.txt`

Relative Path:

- A relative path specifies the location of a file or folder concerning the current working directory (CWD).
- It changes depending on the current working directory.

Example:

- If the CWD is `C:\Users\Chethan`, the relative path for `file.txt` inside `Documents` would be `Documents\file.txt`.

---

2. Define the 'Absolute Path' of a File. When 'cwd' is set to C:\bacon, write the absolute path and relative path for other folders.

Absolute Path:

- The absolute path is the full path to a file, starting from the root directory (`C:\` in Windows or `/` in Linux/Mac).

Examples when CWD is `C:\bacon`

- **Absolute Path for `toast.txt` inside `eggs` folder:**
  - `C:\bacon\eggs\toast.txt`
- **Relative Path for the same file (`toast.txt`) from `C:\bacon`:**
  - `eggs\toast.txt`

---

3. Illustrate the Different Methods of File Operations Supported by the Python `shutil` Module.

The `shutil` module provides high-level file operations such as copying, moving, and deleting files.

Common Methods in `shutil`

Method	Description	Example
`shutil.copy(src, dest)`	Copies a file from `src` to `dest`	`shutil.copy("file.txt", "backup.txt")`
`shutil.copy2(src, dest)`	Copies a file with metadata	`shutil.copy2("file.txt", "backup.txt")`
`shutil.copytree(src, dest)`	Copies an entire directory	`shutil.copytree("folder", "backup_folder")`
`shutil.move(src, dest)`	Moves a file or directory	`shutil.move("file.txt", "D:\\backup")`
`shutil.rmtree(path)`	Deletes a directory and its contents	`shutil.rmtree("old_folder")`

---

4. Explain the Different Steps Involved in Backing Up a Folder into a ZIP File.

Steps to Backup a Folder into a ZIP File using `shutil`

Import the required module
import shutil

1.

Specify the source folder and ZIP file location
source_folder = "my_data"
zip_file_name = "backup"

2.

Create a ZIP backup using `shutil.make_archive()`
shutil.make_archive(zip_file_name, 'zip', source_folder)

3.
4. **Output**: This will create `backup.zip` containing all files from `my_data`.

---

5. Explain How to Save Variables Using the `shelve` Module with an Example Code.

Steps to Save and Load Variables with `shelve`

Import the `shelve` module

```
import shelve
```

1.

Saving Variables

```
with shelve.open("data_store") as db:
    db["name"] = "Chethan"
    db["age"] = 18
```

2.

Loading Variables

```
with shelve.open("data_store") as db:
    print(db["name"])  # Output: Chethan
    print(db["age"])   # Output: 18
```

3.

---

6. Differentiate between `shutil.copy()` and `shutil.copytree()` Methods.

Method	Description	Example
`shutil.copy(src, dest)`	Copies a single file	`shutil.copy("file.txt", "backup.txt")`
`shutil.copytree(src, dest)`	Copies an entire directory with all files and subfolders	`shutil.copytree("source_folder", "backup_folder")`

---

7. Discuss All File Accessing Modes with Illustrated Examples and Write a Python Program to Count and Display Keywords from a Python Source File.

File Accessing Modes

Mode	Description
`'r'`	Read mode (file must exist)
`'w'`	Write mode (overwrites existing file)
`'a'`	Append mode (adds to existing file)
`'r+'`	Read & write mode
`'w+'`	Write & read mode (overwrites)
`'a+'`	Append & read mode

Python Program to Count Python Keywords in a File

```
import keyword

def count_keywords(filename):
    with open(filename, 'r') as f:
        content = f.read()
        words = content.split()
        keyword_count = sum(1 for word in words if word in keyword.kwlist)
        print(f"Number of Python keywords: {keyword_count}")

count_keywords("example.py")  # Replace with your file name
```

---

8. Differentiate between the `read()` and `readlines()` Methods.

Method	Description	Example
`read()`	Reads the entire file as a single string	`file.read()`
`readlines()`	Reads file line by line into a list	`file.readlines()`

Example:

```
with open("sample.txt", "r") as file:
    content = file.read()  # Reads entire file as a string
    lines = file.readlines()  # Reads file into a list of lines
```

---

9. Differentiate Between the Different Steps Involved in Generating Random Quiz Files.

Steps in Generating Random Quiz Files in Python

1. **Import required modules** (`random`, `os`)
2. **Define questions and answers**
3. **Randomly shuffle questions**
4. **Create multiple quiz files**
5. **Write questions and choices into each file**
6. **Generate an answer key**
7. **Save the files**

Example Code:

```
import random

questions = {
    "What is the capital of India?": "New Delhi",
    "What is 2 + 2?": "4",
    "What is the square root of 16?": "4"
}

quiz_file = open("quiz.txt", "w")
for question in random.sample(list(questions.keys()), len(questions)):
    quiz_file.write(f"{question}\n")
quiz_file.close()
```

---

10. List and Illustrate the Methods with Examples for Finding File Sizes and Folder Contents.

Finding File Size

```
import os
print(os.path.getsize("file.txt"))  # Returns size in bytes
```

Listing Folder Contents

```
print(os.listdir("."))  # Lists files and folders in the current directory
```

Using `os.walk()` to Get All Files in a Directory

```
for foldername, subfolders, filenames in os.walk("my_directory"):
```

```
print("Folder:", foldername)
print("Subfolders:", subfolders)
print("Files:", filenames)
```

This document is structured in a Google Docs-friendly format. You can copy-paste it into your document and adjust the styling if needed! 🚀

Here is a well-structured document with detailed explanations and Python programs in a Google Docs-friendly format:

Programs - Module 4

1. Develop a Program in Python to Create a File, Copy Its Content to Another File, and Count the Number of Characters and Lines.

Steps Involved:

1. Create a text file and write content into it.
2. Read the content from the file.
3. Copy the content to another file.
4. Count the number of characters and lines in the original file.

Python Program:

```
# Step 1: Create and write to a file
with open("source.txt", "w") as file:
    file.write("Hello, this is a test file.\nIt contains multiple
lines.\nPython is fun!")

# Step 2: Read from the file and copy content to another file
with open("source.txt", "r") as source_file:
    content = source_file.read()

    # Writing to the destination file
    with open("destination.txt", "w") as dest_file:
        dest_file.write(content)

# Step 3: Count the number of characters and lines
```

```
with open("source.txt", "r") as file:
    lines = file.readlines()
    num_lines = len(lines)
    num_chars = sum(len(line) for line in lines)

# Output results
print(f"Number of lines: {num_lines}")
print(f"Number of characters: {num_chars}")
```

Expected Output:

Number of lines: 3
Number of characters: 66

---

2. Write a Python Program to Read and Extract ZIP Files.

Steps Involved:

1. Import the `zipfile` module.
2. Open the ZIP file in read mode.
3. Extract all files to a specific directory.
4. Print the extracted files.

Python Program:

```
import zipfile

# Step 1: Open the ZIP file in read mode
with zipfile.ZipFile("sample.zip", "r") as zip_ref:
    # Step 2: Extract all files to a folder named 'extracted_files'
    zip_ref.extractall("extracted_files")

    # Step 3: Print extracted files
    print("Extracted files:", zip_ref.namelist())
```

Expected Output (If ZIP contains `file1.txt` and `file2.txt`):

Extracted files: ['file1.txt', 'file2.txt']

---

3. Develop a Program to Compress Files and Read ZIP Files in Python Using the `zipfile` Module.

Steps Involved:

1. Import the `zipfile` module.
2. Create a new ZIP file and add files to it.
3. Open and read the ZIP file to list its contents.

Python Program:

```
import zipfile

# Step 1: Create a ZIP file and add files to it
with zipfile.ZipFile("compressed_files.zip", "w") as zipf:
    zipf.write("source.txt")  # Add file to ZIP
    zipf.write("destination.txt")  # Add another file

# Step 2: Read the ZIP file and list its contents
with zipfile.ZipFile("compressed_files.zip", "r") as zipf:
    print("Files inside ZIP:", zipf.namelist())
```

Expected Output:

Files inside ZIP: ['source.txt', 'destination.txt']

Here is a well-structured and detailed document for your questions, formatted in a **Google Docs-friendly** way.

Module 5

1. When Do Objects Become Mutable? Explain This Concept with the Help of a `Rectangle` Class and Suitable Methods.

Mutability in Python:

- **Mutable objects** are those whose state or content can be changed after creation.
- Objects like lists, dictionaries, and user-defined class objects are mutable.
- **Immutable objects** (e.g., integers, strings, and tuples) cannot be changed after creation.

Example: `Rectangle` Class with Mutable Attributes

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length  # Mutable attribute
        self.width = width    # Mutable attribute

    def change_size(self, new_length, new_width):
        """Method to modify rectangle dimensions."""
        self.length = new_length
        self.width = new_width

    def area(self):
        """Method to calculate the area."""
        return self.length * self.width

# Creating an object
rect = Rectangle(10, 5)
print("Original Area:", rect.area())

# Modifying the object (Mutable)
rect.change_size(20, 10)
print("Updated Area:", rect.area())
```

Output:

Original Area: 50
Updated Area: 200

Here, the object `rect` is **mutable** because we can modify its attributes (`length` and `width`).

---

2. What Are Pure Functions? Explain in Detail with a Suitable Program.

Definition:

- A **pure function** does not modify global state or change the object's state.
- It **only depends on its input parameters** and returns a value without side effects.

Example of a Pure Function:

```
def add_numbers(a, b):
    return a + b  # No modification of external state

result = add_numbers(5, 10)
print("Sum:", result)
```

Example of an Impure Function (Modifies Global State):

```
total = 0

def add_to_total(value):
    global total  # Modifies global state
    total += value

add_to_total(5)
print("Total:", total)  # Changes state
```

Here, `add_numbers()` is **pure** while `add_to_total()` is **impure** because it modifies the `total` variable.

---

3. What Are Modifiers? Write a Suitable Code to Explain This Concept.

Definition:

- A **modifier** is a function that changes the internal state of an object.
- These functions modify instance variables instead of returning a new modified object.

Example:

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance  # Mutable attribute

    def deposit(self, amount):
        """Modifier function to change account balance."""
        self.balance += amount
```

```
    def withdraw(self, amount):
        """Modifier function to change account balance."""
        if amount <= self.balance:
            self.balance -= amount
        else:
            print("Insufficient funds!")

# Creating an object
account = BankAccount(1000)
account.deposit(500)
print("Balance after deposit:", account.balance)  # 1500

account.withdraw(200)
print("Balance after withdrawal:", account.balance)  # 1300
```

Here, `deposit()` and `withdraw()` **modify the object's state**, making them **modifiers**.

---

4. Demonstrate Polymorphism with a Function to Find a Histogram to Count the Number of Times Each Letter Appears in a Word and a Sentence.

Definition:

- **Polymorphism** allows different data types to use the same function name but behave differently.

Example: Histogram Counting Letters in a Word and a Sentence

```
def histogram(text):
    freq = {}  # Dictionary to store frequency
    for char in text:
        if char.isalpha():  # Ignore spaces and special characters
            freq[char] = freq.get(char, 0) + 1
    return freq

# Testing polymorphism
word_hist = histogram("hello")
sentence_hist = histogram("hello world")

print("Histogram for word:", word_hist)
print("Histogram for sentence:", sentence_hist)
```

Output:

Histogram for word: {'h': 1, 'e': 1, 'l': 2, 'o': 1}
Histogram for sentence: {'h': 1, 'e': 1, 'l': 3, 'o': 2, 'w': 1, 'r': 1, 'd': 1}

Here, the `histogram()` function works for both **words** and **sentences** (**polymorphism**).

---

5. Write a Function `draw_rect` That Takes a Turtle Object and a Rectangle and Uses the Turtle to Draw the Rectangle.

Example Code:

```
import turtle

class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

def draw_rect(t, rect):
    """Function to draw a rectangle using Turtle."""
    for _ in range(2):
        t.forward(rect.length)
        t.right(90)
        t.forward(rect.width)
        t.right(90)

# Create a Turtle object
t = turtle.Turtle()
rect = Rectangle(100, 50)

# Draw the rectangle
draw_rect(t, rect)
turtle.done()
```

This function uses **Turtle graphics** to draw a rectangle.

---

6. Differentiate Between `__init__` and `__str__` Methods with an Example.

`__init__` Method:

- **Constructor** that initializes an object when it is created.
- It is automatically called when an object is instantiated.

`__str__` Method:

- **String representation** of the object, which is returned when `print(object)` is called.

Example:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def __str__(self):
        return f"Car: {self.brand} {self.model}"

# Creating an object
my_car = Car("Toyota", "Corolla")
print(my_car)  # Calls __str__ method
```

Output:

```
Car: Toyota Corolla
```

- `__init__` initializes the object.
- `__str__` returns a readable string representation of the object.

---

7. Write a Definition for a Class Named `Circle` with Attributes `center` and `radius`, Where `center` is a `Point` Object and `radius` is a Number.

Example Code:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
class Circle:
    def __init__(self, center, radius):
        self.center = center  # Point object
        self.radius = radius  # Number

    def area(self):
        return 3.14159 * self.radius ** 2

# Creating objects
center_point = Point(5, 5)
my_circle = Circle(center_point, 10)

print(f"Circle Center: ({my_circle.center.x}, {my_circle.center.y})")
print(f"Circle Radius: {my_circle.radius}")
print(f"Circle Area: {my_circle.area()}")
```

Output:

```
Circle Center: (5, 5)
Circle Radius: 10
Circle Area: 314.159
```

Here, `center` is a **composition** of the `Point` class inside the `Circle` class.

Here is a well-structured, **Google Docs-friendly** document for your questions with detailed explanations and Python programs.

Programs - Module 5

1. Develop a Program That Uses Class Student Which Prompts the User to Enter Marks in Three Subjects and Calculates Total Marks, Percentage, and Displays the Score Details.

Steps Involved:

1. Create a `Student` class.
2. Use the `__init__()` method to initialize student details.

3. Use the `getmarks()` method to take input from the user.
4. Use the `display()` method to print total marks and percentage.

Python Program:

```
class Student:
    def __init__(self, name):
        """Initialize student name and marks."""
        self.name = name
        self.marks = []

    def getmarks(self):
        """Prompt the user to enter marks for 3 subjects."""
        for i in range(3):
            mark = float(input(f"Enter marks for subject {i + 1}:
"))
            self.marks.append(mark)

    def display(self):
        """Calculate and display total marks and percentage."""
        total = sum(self.marks)
        percentage = (total / 300) * 100
        print(f"\nStudent Name: {self.name}")
        print(f"Total Marks: {total}/300")
        print(f"Percentage: {percentage:.2f}%")

# Create a Student object and get user input
student_name = input("Enter student's name: ")
student = Student(student_name)
student.getmarks()
student.display()
```

Sample Output:

Enter student's name: Rahul
Enter marks for subject 1: 85
Enter marks for subject 2: 90
Enter marks for subject 3: 95

Student Name: Rahul
Total Marks: 270/300
Percentage: 90.00%

Here, we use **`__init__`**, **`getmarks()`**, and **`display()`** to manage student details.

---

2. Define a Class and Object. Develop a Python Program to Demonstrate the Concept of Classes and Objects.

Definition:

- **Class**: A blueprint for creating objects (e.g., a `Car` class defines attributes like `brand`, `model`).
- **Object**: An instance of a class that holds actual values (e.g., `Car("Toyota", "Corolla")`).

Example: Creating a Class and Object

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display(self):
        """Displays the car's details."""
        print(f"Car Brand: {self.brand}, Model: {self.model}")

# Creating objects
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")

# Calling methods
car1.display()
car2.display()
```

Output:

Car Brand: Toyota, Model: Corolla
Car Brand: Honda, Model: Civic

Here, `Car` is the **class**, and `car1`, `car2` are **objects**.

---

3. Explain Operator Overloading with a Python Programming Example.

Definition:

- **Operator overloading** allows us to define how operators like `+`, `-`, `*`, etc., behave for user-defined objects.
- In Python, this is done using **special methods** (e.g., `__add__()`, `__sub__()`).

Example: Overloading the `+` Operator for a `Vector` Class

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Overloading + operator to add two vectors."""
        return Vector(self.x + other.x, self.y + other.y)

    def display(self):
        """Display vector components."""
        print(f"Vector({self.x}, {self.y})")

# Creating vector objects
v1 = Vector(2, 3)
v2 = Vector(4, 5)

# Using overloaded + operator
v3 = v1 + v2  # Equivalent to v1.__add__(v2)

# Display results
v1.display()
v2.display()
v3.display()
```

Output:

Vector(2, 3)
Vector(4, 5)
Vector(6, 8)

Here, we **overload the + operator** to add two `Vector` objects.
