Question 1(a) [3 marks]

List any 6 applications of Python programming language.

Answer:

Table of Python Applications:

Application Area	Description
Web Development	Django, Flask frameworks
Data Science	Analysis and visualization
Machine Learning	AI model development
Desktop Applications	GUI using Tkinter, PyQt
Game Development	Pygame library
Automation	Scripting and testing

Mnemonic: "Web Data Machine Desktop Game Auto"

Question 1(b) [4 marks]

List any 8 features of Python programming language.

Answer:

Table of Python Features:

Feature	Description
Simple Syntax	Easy to read and write
Interpreted	No compilation needed
Object-Oriented	Supports OOP concepts
Dynamic Typing	Variables don't need type declaration
Cross-Platform	Runs on multiple OS
Large Libraries	Rich standard library
Open Source	Free to use and modify
Interactive	REPL environment

Mnemonic: "Simple Interpreted Object Dynamic Cross Large Open Interactive"

Question 1(c) [7 marks]

Explain working of for and while loops in Python.

Answer:

For Loop:

- **Iteration**: Repeats over sequences (lists, strings, ranges)
- **Syntax**: `for variable in sequence:`
- **Automatic**: Handles iteration automatically

While Loop:

- **Condition-based**: Continues while condition is true
- **Manual control**: Programmer controls iteration
- **Risk**: Can create infinite loops if condition never becomes false

Diagram:

```
    Start
      │
   Initialize
      │
    Condition? ----No----> End
      │Yes
    Execute
      │
   Update
      │
    (loop back)
```

Code Example:

```python
# For loop
for i in range(5):
    print(i)

# While loop
i = 0
while i < 5:
    print(i)
    i += 1
```

Mnemonic: "For Automatic, While Manual"

Question 1(c OR) [7 marks]

Explain working of break continue and pass statements in Python.

Answer:

Break Statement:

- **Exit**: Terminates the entire loop
- **Usage**: When specific condition is met
- **Effect**: Control moves to next statement after loop

Continue Statement:

- **Skip**: Skips current iteration only
- **Usage**: Skip specific values in iteration
- **Effect**: Moves to next iteration

Pass Statement:

- **Placeholder**: Does nothing, syntactic placeholder
- **Usage**: When syntax requires statement but no action needed
- **Effect**: No operation performed

Code Examples:

```
# Break
for i in range(10):
    if i == 5:
        break
    print(i)  # prints 0,1,2,3,4

# Continue
for i in range(5):
    if i == 2:
        continue
    print(i)  # prints 0,1,3,4

# Pass
if True:
    pass  # placeholder
```

Mnemonic: "Break Exits, Continue Skips, Pass Waits"

Question 2(a) [3 marks]

Develop a Python program to increment each element of list by one.

Answer:

Code:

```python
# Method 1 - Using for loop
numbers = [1, 2, 3, 4, 5]
for i in range(len(numbers)):
    numbers[i] += 1
print(numbers)

# Method 2 - List comprehension
numbers = [1, 2, 3, 4, 5]
result = [x + 1 for x in numbers]
print(result)
```

Mnemonic: "Loop Index or Comprehension"

Question 2(b) [4 marks]

Develop a Python program to read three numbers from the user and find the average of the numbers.

Answer:

Code:

```python
# Input three numbers
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
num3 = float(input("Enter third number: "))

# Calculate average
average = (num1 + num2 + num3) / 3

# Display result
print(f"Average is: {average}")
```

Key Points:

- **Input**: Use `float()` for decimal numbers
- **Formula**: Sum divided by count
- **Output**: Use f-string for formatting

Mnemonic: "Input Float, Sum Divide, Format Output"

Question 2(c) [7 marks]

Explain Python's list data type in detail.

Answer:

List Characteristics:

- **Ordered**: Elements maintain sequence
- **Mutable**: Can be modified after creation

- **Heterogeneous**: Can store different data types
- **Indexed**: Access elements using index (0-based)

List Operations Table:

Operation	Syntax	Description
Creation	`list = [1,2,3]`	Create new list
Access	`list[0]`	Get element by index
Append	`list.append(4)`	Add element at end
Insert	`list.insert(1,5)`	Add at specific position
Remove	`list.remove(2)`	Remove first occurrence
Pop	`list.pop()`	Remove and return last
Slice	`list[1:3]`	Get sublist

Code Example:

```
# List creation and operations
fruits = ['apple', 'banana', 'orange']
fruits.append('mango')
fruits.insert(1, 'grape')
print(fruits[0])  # apple
print(len(fruits))  # 5
```

Mnemonic: "Ordered Mutable Heterogeneous Indexed"

Question 2(a OR) [3 marks]

Develop a Python program to find sum of all elements in a list using for loop.

Answer:

Code:

```python
# Method 1 - Traditional for loop
numbers = [10, 20, 30, 40, 50]
total = 0
for num in numbers:
    total += num
print(f"Sum is: {total}")

# Method 2 - Using range and index
numbers = [10, 20, 30, 40, 50]
total = 0
for i in range(len(numbers)):
    total += numbers[i]
print(f"Sum is: {total}")
```

Mnemonic: "Initialize Zero, Loop Add, Print Total"

Question 2(b OR) [4 marks]

Develop a Python program to get input from user for principal, rate and no of years then calculate and display simple interest from that.

Answer:

Code:

```python
# Get input from user
principal = float(input("Enter principal amount: "))
rate = float(input("Enter rate of interest: "))
time = float(input("Enter time in years: "))

# Calculate simple interest
simple_interest = (principal * rate * time) / 100

# Display results
print(f"Principal: {principal}")
print(f"Rate: {rate}%")
print(f"Time: {time} years")
print(f"Simple Interest: {simple_interest}")
print(f"Total Amount: {principal + simple_interest}")
```

Formula:

- **Simple Interest** = (P × R × T) / 100
- **Total Amount** = Principal + Simple Interest

Mnemonic: "Principal Rate Time, Multiply Divide Hundred"

Question 2(c OR) [7 marks]

Explain Python's tuple data type in detail.

Answer:

Tuple Characteristics:

- **Ordered**: Elements maintain sequence
- **Immutable**: Cannot be modified after creation
- **Heterogeneous**: Can store different data types
- **Indexed**: Access using index (0-based)

Tuple Operations Table:

Operation	Syntax	Description
Creation	`tuple = (1,2,3)`	Create new tuple
Access	`tuple[0]`	Get element by index
Count	`tuple.count(2)`	Count occurrences
Index	`tuple.index(3)`	Find first index
Slice	`tuple[1:3]`	Get sub-tuple
Length	`len(tuple)`	Get tuple size
Concatenate	`tuple1 + tuple2`	Join tuples

Code Example:

```
# Tuple creation and operations
coordinates = (10, 20, 30)
print(coordinates[0])  # 10
print(len(coordinates))  # 3
x, y, z = coordinates  # tuple unpacking
new_tuple = coordinates + (40, 50)
```

Key Differences from List:

- **Immutable**: Cannot change elements
- **Performance**: Faster than lists
- **Usage**: For fixed data collections

Mnemonic: "Ordered Immutable Heterogeneous Indexed"

Question 3(a) [3 marks]

Explain any 3 random module methods.

Answer:

Random Module Methods Table:

Method	Syntax	Description
random()	`random.random()`	Float between 0.0 to 1.0
randint()	`random.randint(1,10)`	Integer between given range
choice()	`random.choice(list)`	Random element from sequence

Code Example:

```
import random

# Generate random float
print(random.random())  # 0.7234567

# Generate random integer
print(random.randint(1, 10))  # 7

# Choose random element
colors = ['red', 'blue', 'green']
print(random.choice(colors))  # blue
```

Mnemonic: "Random Float, Randint Integer, Choice Select"

Question 3(b) [4 marks]

Develop a Python program that asks the user for a string and prints out the location of each 'a' in the string.

Answer:

Code:

```
# Get string from user
text = input("Enter a string: ")

# Find all positions of 'a'
positions = []
for i in range(len(text)):
    if text[i].lower() == 'a':
        positions.append(i)

# Display results
if positions:
    print(f"Letter 'a' found at positions: {positions}")
else:
    print("Letter 'a' not found in the string")

# Alternative method using enumerate
text = input("Enter a string: ")
for index, char in enumerate(text):
    if char.lower() == 'a':
        print(f"'a' found at position {index}")
```

Key Points:

- **Case-insensitive**: Use `.lower()` to find both 'a' and 'A'
- **Index tracking**: Use range or enumerate
- **Output format**: Clear position indication

Mnemonic: "Loop Index Check Append Print"

Question 3(c) [7 marks]

Explain Python's string data type in detail.

Answer:

String Characteristics:

- **Immutable**: Cannot be changed after creation
- **Sequence**: Ordered collection of characters
- **Indexed**: Access characters using index
- **Unicode**: Supports all languages and symbols

String Methods Table:

Method	Example	Description
upper()	`"hello".upper()`	Convert to uppercase
lower()	`"HELLO".lower()`	Convert to lowercase
strip()	`" hello ".strip()`	Remove whitespace
split()	`"a,b,c".split(",")`	Split into list
replace()	`"hello".replace("l","x")`	Replace substring
find()	`"hello".find("e")`	Find substring index
join()	`",".join(["a","b"])`	Join list elements

String Operations:

```
# String creation
name = "Python Programming"

# String indexing and slicing
print(name[0])      # P
print(name[0:6])    # Python
print(name[-1])     # g

# String formatting
age = 25
message = f"I am {age} years old"
```

Key Features:

- **Concatenation**: Using + operator
- **Repetition**: Using * operator
- **Membership**: Using 'in' operator
- **Formatting**: f-strings, .format(), % formatting

Mnemonic: "Immutable Sequence Indexed Unicode"

Question 3(a OR) [3 marks]

Explain any 3 math module methods.

Answer:

Math Module Methods Table:

Method	Syntax	Description
sqrt()	`math.sqrt(16)`	Square root calculation
pow()	`math.pow(2,3)`	Power calculation
ceil()	`math.ceil(4.3)`	Round up to integer

Code Example:

```
import math

# Square root
print(math.sqrt(25))    # 5.0

# Power
print(math.pow(2, 3))   # 8.0

# Ceiling
print(math.ceil(4.2))   # 5
```

Mnemonic: "Square Root, Power Up, Ceiling Round"

Question 3(b OR) [4 marks]

Develop a Python program to get a string from the user and count total no. of Vowels present in that string.

Answer:

Code:

```python
# Get string from user
text = input("Enter a string: ")

# Define vowels
vowels = "aeiouAEIOU"

# Count vowels
vowel_count = 0
for char in text:
    if char in vowels:
        vowel_count += 1

# Display result
print(f"Total vowels in '{text}': {vowel_count}")

# Alternative method using list comprehension
text = input("Enter a string: ")
vowels = "aeiouAEIOU"
count = sum(1 for char in text if char in vowels)
print(f"Total vowels: {count}")
```

Key Points:

- **Vowel definition**: Include both cases
- **Loop through**: Each character in string
- **Count logic**: Check membership and increment

Mnemonic: "Define Vowels, Loop Check, Count Increment"

Question 3(c OR) [7 marks]

Explain Python's set data type in detail.

Answer:

Set Characteristics:

- **Unordered**: No fixed sequence of elements
- **Mutable**: Can add/remove elements
- **Unique**: No duplicate elements allowed

- **Iterable**: Can loop through elements

Set Operations Table:

Operation	Syntax	Description
Creation	`set = {1,2,3}`	Create new set
Add	`set.add(4)`	Add single element
Remove	`set.remove(2)`	Remove element (error if not found)
Discard	`set.discard(2)`	Remove element (no error)
Union	`set1 | set2`	Combine sets
Intersection	`set1 & set2`	Common elements
Difference	`set1 - set2`	Elements in set1 only

Set Mathematical Operations:

```python
# Set creation
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

# Set operations
print(A | B)     # Union: {1,2,3,4,5,6}
print(A & B)     # Intersection: {3,4}
print(A - B)     # Difference: {1,2}
print(A ^ B)     # Symmetric difference: {1,2,5,6}
```

Key Uses:

- **Remove duplicates**: From lists
- **Mathematical operations**: Union, intersection
- **Membership testing**: Fast lookup

Mnemonic: "Unordered Mutable Unique Iterable"

Question 4(a) [3 marks]

What is the class in Python. How is it different from an object?

Answer:

Class vs Object Comparison:

Aspect	Class	Object
Definition	Blueprint or template	Instance of class
Memory	No memory allocated	Memory allocated
Existence	Logical entity	Physical entity
Creation	Using class keyword	Using class constructor

Example:

```
# Class definition (blueprint)
class Car:
    def __init__(self, brand):
        self.brand = brand

# Object creation (instances)
car1 = Car("Toyota")  # Object 1
car2 = Car("Honda")   # Object 2
```

Key Points:

- **Class**: Template defining properties and methods
- **Object**: Actual instance with specific values
- **Relationship**: One class, multiple objects

Mnemonic: "Class Blueprint, Object Instance"

Question 4(b) [4 marks]

Explain any four methods of dictionary data type of Python.

Answer:

Dictionary Methods Table:

Method	Syntax	Description
keys()	`dict.keys()`	Get all keys
values()	`dict.values()`	Get all values
items()	`dict.items()`	Get key-value pairs
get()	`dict.get('key')`	Get value safely

Code Example:

```python
student = {'name': 'John', 'age': 20, 'grade': 'A'}

# Dictionary methods
print(student.keys())    # dict_keys(['name', 'age', 'grade'])
print(student.values())  # dict_values(['John', 20, 'A'])
print(student.items())   # dict_items([('name', 'John'), ...])
print(student.get('name'))  # John
```

Mnemonic: "Keys Values Items Get"

Question 4(c) [7 marks]

Develop a Python program that defines a user-defined module for performing some tasks. Import this module and use its functions.

Answer:

Module Creation (math_operations.py):

```python
# math_operations.py
def add(a, b):
    """Add two numbers"""
    return a + b

def multiply(a, b):
    """Multiply two numbers"""
    return a * b

def factorial(n):
    """Calculate factorial"""
    if n <= 1:
        return 1
    return n * factorial(n - 1)

PI = 3.14159

def circle_area(radius):
    """Calculate circle area"""
    return PI * radius * radius
```

Main Program (main.py):

```python
# Import entire module
import math_operations

# Use module functions
result1 = math_operations.add(5, 3)
result2 = math_operations.multiply(4, 6)
result3 = math_operations.factorial(5)
area = math_operations.circle_area(5)
```

```python
print(f"Addition: {result1}")
print(f"Multiplication: {result2}")
print(f"Factorial: {result3}")
print(f"Circle Area: {area}")

# Import specific functions
from math_operations import add, multiply
print(f"Direct call: {add(10, 20)}")
```

Key Points:

- **Module creation**: Separate .py file with functions
- **Import methods**: import module or from module import function
- **Usage**: Access using module.function() or direct function()

Mnemonic: "Create Import Use"

Question 4(a OR) [3 marks]

Define types of methods available in Python classes.

Answer:

Types of Methods Table:

Method Type	Syntax	Description
Instance Method	`def method(self):`	Access instance variables
Class Method	`@classmethod def method(cls):`	Access class variables
Static Method	`@staticmethod def method():`	Independent of class/instance

Example:

```python
class MyClass:
    class_var = "Class Variable"

    def instance_method(self):  # Instance method
        return "Instance method"

    @classmethod
    def class_method(cls):      # Class method
        return cls.class_var

    @staticmethod
    def static_method():        # Static method
        return "Static method"
```

Mnemonic: "Instance Self, Class Cls, Static None"

Question 4(b OR) [4 marks]

Explain any four methods of string data type of Python.

Answer:

String Methods Table:

Method	Syntax	Description
startswith()	`str.startswith('pre')`	Check if starts with substring
endswith()	`str.endswith('suf')`	Check if ends with substring
isdigit()	`str.isdigit()`	Check if all digits
count()	`str.count('sub')`	Count substring occurrences

Code Example:

```
text = "Hello World 123"

# String methods
print(text.startswith('Hello'))  # True
print(text.endswith('123'))      # True
print('123'.isdigit())           # True
print(text.count('l'))           # 3
```

Mnemonic: "Start End Digit Count"

Question 4(c OR) [7 marks]

Develop a Python program to find factorial of a number using recursive user defined function.

Answer:

Code:

```
def factorial(n):
    """
    Calculate factorial using recursion
    Base case: factorial(0) = 1, factorial(1) = 1
    Recursive case: factorial(n) = n * factorial(n-1)
    """
    # Base case
    if n == 0 or n == 1:
        return 1

    # Recursive case
    else:
        return n * factorial(n - 1)
```

```
# Main program
try:
    num = int(input("Enter a number: "))

    if num < 0:
        print("Factorial not defined for negative numbers")
    else:
        result = factorial(num)
        print(f"Factorial of {num} is {result}")

except ValueError:
    print("Please enter a valid integer")

# Test cases
print(f"Factorial of 5: {factorial(5)}")  # 120
print(f"Factorial of 0: {factorial(0)}")  # 1
```

Recursion Flow:

```
factorial(5)
    |
5 * factorial(4)
        |
    4 * factorial(3)
            |
        3 * factorial(2)
                |
            2 * factorial(1)
                    |
                return 1

Result: 5 * 4 * 3 * 2 * 1 = 120
```

Key Points:

- **Base case**: Stops recursion (n=0 or n=1)
- **Recursive case**: Function calls itself
- **Error handling**: Check for negative input

Mnemonic: "Base Stop, Recursive Call, Error Check"

Question 5(a) [3 marks]

Develop a python program to Implement single inheritance.

Answer:

Code:

```
# Parent class
```

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound")

    def eat(self):
        print(f"{self.name} is eating")

# Child class inheriting from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call parent constructor
        self.breed = breed

    def bark(self):
        print(f"{self.name} is barking")

    def speak(self):  # Override parent method
        print(f"{self.name} says Woof!")

# Create objects and test
dog = Dog("Buddy", "Golden Retriever")
dog.speak()  # Buddy says Woof!
dog.eat()    # Buddy is eating (inherited)
dog.bark()   # Buddy is barking (own method)
```

Mnemonic: "Parent Child Inherit Override"

Question 5(b) [4 marks]

Explain the significance of constructors in Python classes.

Answer:

Constructor Significance:

Aspect	Description
Initialization	Automatically called when object is created
Setup	Initialize instance variables with values
Memory	Allocate memory for object attributes
Validation	Validate input parameters during creation

Constructor Types:

```python
class Student:
    # Default constructor
    def __init__(self):
        self.name = "Unknown"
        self.age = 0

    # Parameterized constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print(f"Student {name} created")

    # Constructor with default parameters
    def __init__(self, name="Unknown", age=0):
        self.name = name
        self.age = age
```

Key Benefits:

- **Automatic execution**: No need to call manually
- **Object state**: Ensures proper initialization
- **Code reusability**: Common setup code in one place

Mnemonic: "Initialize Setup Memory Validate"

Question 5(c) [7 marks]

Develop a Python program to demonstrate method overriding using inheritance.

Answer:

Code:

```
# Base class
class Shape:
    def __init__(self, name):
        self.name = name

    def area(self):
        print(f"Area calculation for {self.name}")
        return 0

    def display(self):
        print(f"This is a {self.name}")

# Derived class 1
class Rectangle(Shape):
    def __init__(self, length, width):
        super().__init__("Rectangle")
        self.length = length
        self.width = width

    # Override area method
```

```python
    def area(self):
        area_value = self.length * self.width
        print(f"Rectangle area: {area_value}")
        return area_value

# Derived class 2
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    # Override area method
    def area(self):
        area_value = 3.14 * self.radius * self.radius
        print(f"Circle area: {area_value}")
        return area_value

    # Override display method
    def display(self):
        super().display()  # Call parent method
        print(f"Radius: {self.radius}")

# Test method overriding
shapes = [
    Rectangle(5, 4),
    Circle(3),
    Shape("Generic Shape")
]

for shape in shapes:
    shape.display()
    shape.area()
    print("-" * 20)
```

Method Overriding Diagram:

```
    Shape (Base)
    |-- area()
    |-- display()
         |
    Rectangle    Circle
    |-- area()   |-- area()
                 |-- display()
```

Key Points:

- **Same method name**: In parent and child classes
- **Different implementation**: Child class provides specific logic
- **Runtime decision**: Correct method called based on object type
- **Super() usage**: Access parent class method

Mnemonic: "Same Name Different Logic Runtime Decision"

Question 5(a OR) [3 marks]

Explain concept of data encapsulation in Python.

Answer:

Data Encapsulation:

Aspect	Description
Definition	Bundling data and methods together
Access Control	Restrict direct access to internal data
Data Hiding	Internal implementation hidden from outside
Interface	Provide controlled access through methods

Implementation:

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):    # Public method
        if amount > 0:
            self.__balance += amount

    def get_balance(self):        # Public method
        return self.__balance

    def __validate(self):         # Private method
        return self.__balance >= 0

# Usage
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # 1500
# print(account.__balance)    # Error - cannot access private
```

Mnemonic: "Bundle Data Hide Interface"

Question 5(b OR) [4 marks]

Explain concept of abstract classes in Python.

Answer:

Abstract Classes:

Concept	Description
Definition	Class that cannot be instantiated directly
Abstract Methods	Methods declared but not implemented
Implementation	Subclasses must implement abstract methods
Purpose	Define common interface for related classes

Implementation using ABC:

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Abstract class
    @abstractmethod
    def make_sound(self):  # Abstract method
        pass

    def sleep(self):       # Concrete method
        print("Animal is sleeping")

class Dog(Animal):
    def make_sound(self):  # Must implement
        print("Woof!")

class Cat(Animal):
    def make_sound(self):  # Must implement
        print("Meow!")

# Usage
dog = Dog()
dog.make_sound()  # Woof!
# animal = Animal()  # Error - cannot instantiate
```

Key Features:

- **Cannot instantiate**: Abstract class cannot create objects
- **Force implementation**: Subclasses must implement abstract methods
- **Common interface**: Ensures consistent method signatures

Mnemonic: "Cannot Instantiate Force Implementation Common Interface"

Question 5(c OR) [7 marks]

Develop a python program to Implement multiple inheritance.

Answer:

Code:

```python
# First parent class
class Father:
    def __init__(self):
        self.father_name = "John"
        print("Father constructor called")

    def show_father(self):
        print(f"Father: {self.father_name}")

    def work(self):
        print("Father works as Engineer")

# Second parent class
class Mother:
    def __init__(self):
        self.mother_name = "Mary"
        print("Mother constructor called")

    def show_mother(self):
        print(f"Mother: {self.mother_name}")

    def work(self):
        print("Mother works as Doctor")

# Child class inheriting from both parents
class Child(Father, Mother):
    def __init__(self):
        Father.__init__(self)  # Call father's constructor
        Mother.__init__(self)  # Call mother's constructor
        self.child_name = "Alice"
        print("Child constructor called")

    def show_child(self):
        print(f"Child: {self.child_name}")

    def show_family(self):
        self.show_father()
        self.show_mother()
        self.show_child()

# Create child object and test
child = Child()
print("\nFamily Details:")
child.show_family()
print("\nMethod Resolution:")
child.work()  # Calls Father's work method (MRO)

# Check Method Resolution Order
print(f"\nMRO: {Child.__mro__}")
```

Multiple Inheritance Diagram:

```
    Father        Mother
    |              |
    |              |
    +------+-------+
           |
         Child
```

Key Points:

- **Multiple parents**: Child inherits from both Father and Mother
- **Method Resolution Order (MRO)**: Determines which method is called
- **Constructor calls**: Explicitly call parent constructors
- **Diamond problem**: Python handles with MRO

Output:

```
Father constructor called
Mother constructor called
Child constructor called

Family Details:
Father: John
Mother: Mary
Child: Alice

Method Resolution:
Father works as Engineer
```

Mnemonic: "Multiple Parents MRO Constructor Diamond"