Subject Name Solutions

4351108 – Winter 2024

Semester 1 Study Material

Detailed Solutions and Explanations

Question 1(a) [3 marks]

List out features of python programming language.

Solution

Feature	Description
Simple & Easy	Clean, readable syntax
Free & Open Source	No cost, community driven
Cross-platform	Runs on Windows, Linux, Mac
Interpreted	No compilation needed
Object-Oriented	Supports classes and objects
Large Libraries	Rich standard library

Mnemonic

"Simple Free Cross Interpreted Object Large"

Question 1(b) [4 marks]

Write applications of python programming language.

Solution

Application Area	Examples
Web Development	Django, Flask frameworks
Data Science	NumPy, Pandas, Matplotlib
Machine Learning	TensorFlow, Scikit-learn
Desktop GUI	Tkinter, PyQt applications
Game Development	Pygame library
Automation	Scripting and testing

Mnemonic

"Web Data Machine Desktop Game Auto"

Question 1(c) [7 marks]

Explain various datatypes in python.

Solution

Data Type	Example	Description
int	`x = 5`	Whole numbers
float	`y = 3.14`	Decimal numbers
str	`name = "John"`	Text data
bool	`flag = True`	True/False values
list	`[1, 2, 3]`	Ordered, mutable
tuple	`(1, 2, 3)`	Ordered, immutable
dict	`{"a": 1}`	Key-value pairs
set	`{1, 2, 3}`	Unique elements

Code Example:

```
\# Numeric types
age = 25          \# int
price = 99.99     \# float

\# Text type
name = "Python"   \# str

\# Boolean type
is\_valid = True   \# bool

\# Collection types
numbers = [1, 2, 3]         \# list
coordinates = (10, 20)      \# tuple
student = \{"name": "John"\ }\# dict
unique\_ids = \{1, 2, 3\     }\# set
```

Mnemonic

"Integer Float String Boolean List Tuple Dict Set"

Question 1(c OR) [7 marks]

Explain arithmetic, assignment, and identity operators with example.

Solution

Arithmetic Operators:

Operator	Operation	Example
`+`	Addition	`5 + 3 = 8`
`-`	Subtraction	`5 - 3 = 2`
`*`	Multiplication	`5 * 3 = 15`
`/`	Division	`10 / 3 = 3.33`
`//`	Floor Division	`10 // 3 = 3`
`%`	Modulus	`10 % 3 = 1`
`**`	Exponent	`2 ** 3 = 8`

Assignment Operators:

Operator	Example	Equivalent
`=`	`x = 5`	Assign value
`+=`	`x += 3`	`x = x + 3`
`-=`	`x -= 2`	`x = x - 2`
`*=`	`x *= 4`	`x = x * 4`

Identity Operators:

Operator	Purpose	Example
`is`	Same object	`x is y`
`is not`	Different object	`x is not y`

Code Example:

```
\# Arithmetic
a = 10 + 5    \# 15
b = 10 // 3   \# 3

\# Assignment
x = 5
x += 3        \# x becomes 8

\# Identity
list1 = [1, 2, 3]
list2 = [1, 2, 3]
print(list1 is list2)      \# False
print(list1 is not list2)  \# True
```

Mnemonic

"Add Assign Identity"

Question 2(a) [3 marks]

Which of the following identifier names are invalid? **(i) Total Marks (ii)Total_Marks (iii)total-Marks (iv) Hundred$ (v) _Percentage (vi) True**

Solution

Identifier	Valid/Invalid	Reason
Total Marks	**Invalid**	Contains space
Total_Marks	Valid	Underscore allowed
total-Marks	**Invalid**	Hyphen not allowed
Hundred$	**Invalid**	$ symbol not allowed
_Percentage	Valid	Can start with underscore
True	**Invalid**	Reserved keyword

Invalid identifiers: Total Marks, total-Marks, Hundred$, True

Mnemonic

"Space Hyphen Dollar Keyword = Invalid"

Question 2(b) [4 marks]

Write a program to find a maximum number among the given three numbers.

Solution

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

\# Find maximum using if{-elif{-}else}
if num1 {=} num2 and num1 {=} num3:
    maximum = num1
elif num2 {=} num1 and num2 {=} num3:
    maximum = num2
else:
    maximum = num3

\# Display result
print(f"Maximum number is: \{maximum\}")
```

Alternative using max() function:

```
num1, num2, num3 = map(float, input("Enter 3 numbers: ").split())
maximum = max(num1, num2, num3)
print(f"Maximum: \{maximum\}")
```

Mnemonic

"Input Compare Display"

Question 2(c) [7 marks]

Explain dictionaries in Python. Write statements to add, modify, and delete elements in a dictionary.

Solution

Dictionary is a collection of key-value pairs that is ordered, changeable, and does not allow duplicate keys.
Operations Table:

Operation	Syntax	Example
Create	`dict_name = {}`	`student = {}`
Add	`dict[key] = value`	`student['name'] = 'John'`
Modify	`dict[key] = new_value`	`student['name'] = 'Jane'`
Delete	`del dict[key]`	`del student['name']`
Access	`dict[key]`	`print(student['name'])`

Code Example:

```
\# Create empty dictionary
student = \{\}

\# Add elements
student[{name}] = {John}
student[{age}] = 20
student[{grade}] = {A}

\# Modify element
student[{age}] = 21

\# Delete element
del student[{grade}]

\# Display dictionary
print(student)  \# Output: \{{name: John, age: 21\}}

\# Other methods
student.pop({age})            \# Remove and return value
student.update(\{{city}: {Mumbai}\)  }\# Add multiple items
```

Dictionary Properties:

- **Ordered**: Maintains insertion order (Python 3.7+)
- **Changeable**: Can modify after creation
- **No Duplicates**: Keys must be unique

Mnemonic

"Key-Value Ordered Changeable Unique"

Question 2(a OR) [3 marks]

Write a program to display the following pattern.

Solution

```
\# Pattern program
for i in range(1, 6):
    for j in range(1, i + 1):
        print(j, end=" ")
    print()  \# New line after each row
```

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Mnemonic

"Outer Row Inner Column Print"

Question 2(b OR) [4 marks]

Write a program to find the sum of digits of an integer number, input by the user.

Solution

```
\# Input number from user
number = int(input("Enter a number: "))
original\_number = number
sum\_digits = 0

\# Extract and sum digits
while number {} 0:
    digit = number \% 10    \# Get last digit
    sum\_digits += digit    \# Add to sum
    number = number // 10  \# Remove last digit

\# Display result
print(f"Sum of digits of \{original\_number\} is: \{sum\_digits\}")
```

Alternative Method:

```
number = input("Enter number: ")
sum\_digits = sum(int(digit) for digit in number)
print(f"Sum of digits: \{sum\_digits\}")
```

Mnemonic

"Input Extract Sum Display"

Question 2(c OR) [7 marks]

Explain slicing and concatenation operation on list.

Solution

List Slicing: Extracting portion of list using `[start:stop:step]` syntax.
Slicing Syntax Table:

Syntax	Description	Example
`list[start:stop]`	Elements from start to stop-1	`nums[1:4]`
`list[:stop]`	From beginning to stop-1	`nums[:3]`
`list[start:]`	From start to end	`nums[2:]`
`list[::step]`	All elements with step	`nums[::2]`
`list[::-1]`	Reverse list	`nums[::-1]`

Concatenation: Joining two or more lists using `+` operator or `extend()` method.
Code Example:

```
\# Create lists
list1 = [1, 2, 3, 4, 5]
list2 = [6, 7, 8]

\# Slicing operations
print(list1[1:4])    \# [2, 3, 4]
print(list1[:3])     \# [1, 2, 3]
print(list1[2:])     \# [3, 4, 5]
print(list1[::2])    \# [1, 3, 5]
print(list1[::{-}1])   \# [5, 4, 3, 2, 1]

\# Concatenation operations
result1 = list1 + list2           \# [1, 2, 3, 4, 5, 6, 7, 8]
list1.extend(list2)               \# Adds list2 to list1
combined = [*list1, *list2]       \# Using unpacking operator
```

Key Points:

- **Slicing**: Creates new list without modifying original
- **Concatenation**: Combines lists into single list
- **Negative indexing**: `list[-1]` gives last element

Mnemonic

"Slice Extract Concat Join"

Question 3(a) [3 marks]

Define a list in Python. Write name of the function used to add an element to the end of a list.

Solution

List Definition: A **list** is an ordered collection of items that is changeable and allows duplicate values.
Properties Table:

Property	Description
Ordered	Items have defined order
Changeable	Can modify after creation
Duplicates	Allows duplicate values
Indexed	Items accessed by index

Function to add element: `append()`
Example:

```
\# Create list
fruits = [{apple}, {banana}]

\# Add element to end
fruits.append({orange})
print(fruits)  \# [{apple, banana, orange]}
```

Mnemonic

"List Append End"

Question 3(b) [4 marks]

Define a tuple in Python. Write statement to access last element of a tuple.

Solution

Tuple Definition: A **tuple** is an ordered collection of items that is unchangeable and allows duplicate values.
Properties Table:

Property	Description
Ordered	Items have defined order
Unchangeable	Cannot modify after creation
Duplicates	Allows duplicate values
Indexed	Items accessed by index

Accessing Last Element:

```
\# Method 1: Using negative index
my\_tuple = (10, 20, 30, 40, 50)
last\_element = my\_tuple[{-}1]
print(last\_element)  \# Output: 50

\# Method 2: Using length
last\_element = my\_tuple[len(my\_tuple) {-} 1]
print(last\_element)  \# Output: 50
```

Mnemonic

"Tuple Unchangeable Negative Index"

Question 3(c) [7 marks]

Write statements for following set operations: create empty set, add an element to a set, remove an element from set, Union of two sets, Intersection of two sets, Difference between two sets and symmetric difference between two sets.

Solution

Set Operations Table:

Operation	Method	Operator	Example
Create Empty	`set()`	-	`s = set()`
Add Element	`add()`	-	`s.add(5)`
Remove Element	`remove()`	-	`s.remove(5)`
Union	`union()`	`|`	`A.union(B)` or `A | B`
Intersection	`intersection()`	`&`	`A.intersection(B)` or `A & B`
Difference	`difference()`	-	`A.difference(B)` or `A - B`
Symmetric Diff	`symmetric_difference()`		`A.symmetric_difference(B)` or `A ^ B`

Code Example:

```
\# Create empty set
my\_set = set()

\# Add elements
my\_set.add(10)
my\_set.add(20)

\# Remove element
my\_set.remove(10)

\# Create two sets for operations
A = \{1, 2, 3, 4\}
B = \{3, 4, 5, 6\}

\# Union (all unique elements)
union\_result = A.union(B)        \# \{1, 2, 3, 4, 5, 6\}

\# Intersection (common elements)
intersection\_result = A.intersection(B)  \# \{3, 4\}

\# Difference (A {- B)}
difference\_result = A.difference(B)      \# \{1, 2\}

\# Symmetric difference (elements in A or B, but not both)
sym\_diff\_result = A.symmetric\_difference(B)  \# \{1, 2, 5, 6\}
```

Mnemonic

"Create Add Remove Union Intersect Differ Symmetric"

Question 3(a OR) [3 marks]

Define a string in Python. Using example illustrate (i) How to create a string. (ii) Accessing individual characters using indexing.

Solution

String Definition: A **string** is a sequence of characters enclosed in quotes (single or double).
(i) Creating String:

```
\# Single quotes
name = {Python}

\# Double quotes
message = "Hello World"

\# Triple quotes (multiline)
text = """This is a
multiline string"""
```

(ii) Accessing Characters:

```
word = "PYTHON"
print(word[0])    \# P (first character)
print(word[2])    \# T (third character)
print(word[{-}1])   \# N (last character)
print(word[{-}2])   \# O (second last)
```

Mnemonic

"String Quotes Index Access"

Question 3(b OR) [4 marks]

Explain list traversing using for loop and while loop.

Solution

List Traversing means visiting each element of list one by one.
For Loop Traversing:

```
numbers = [10, 20, 30, 40, 50]

\# Method 1: Direct iteration
for num in numbers:
    print(num)

\# Method 2: Using index
for i in range(len(numbers)):
    print(f"Index \{i\}: \{numbers[i]\}")
```

While Loop Traversing:

```
numbers = [10, 20, 30, 40, 50]
i = 0

while i {} len(numbers):
    print(f"Element at index \{i\}: \{numbers[i]\}")
    i += 1
```

Comparison Table:

Loop Type	Advantage	Use Case
For Loop	Simpler syntax	When number of iterations known
While Loop	More control	When condition-based iteration needed

Mnemonic

"For Simple While Control"

Question 3(c OR) [7 marks]

Write a program to create a dictionary with the roll number, name, and marks of n students and display the names of students who have scored marks above 75.

Solution

```
\# Input number of students
n = int(input("Enter number of students: "))

\# Create empty dictionary
students = \{\}
```

```
\# Input student data
for i in range(n):
    print(f"{n}Enter details for student \{i + 1\}:")
    roll\_no = int(input("Roll number: "))
    name = input("Name: ")
    marks = float(input("Marks: "))

    \# Store in dictionary
    students[roll\_no] = \{
        {name}: name,
        {marks}: marks
    \}

\# Display students with marks above 75
print("{n}Students with marks above 75:")
print("{-"} * 30)

high\_performers = []
for roll\_no, data in students.items():
    if data[{marks}] {} 75:
        high\_performers.append(data[{name}])
        print(f"Name: \{data[{name}]\}, Marks: \{data[{marks}]\}")

if not high\_performers:
    print("No student scored above 75 marks")
else:
    print(f"{n}Total high performers: \{len(high\_performers)\}")
```

Sample Output:

```
Enter number of students: 2

Enter details for student 1:
Roll number: 101
Name: John
Marks: 80

Enter details for student 2:
Roll number: 102
Name: Alice
Marks: 70

Students with marks above 75:
------------------------------
Name: John, Marks: 80.0

Total high performers: 1
```

Mnemonic

"Input Store Filter Display"

Question 4(a) [3 marks]

Write any three functions available in random module. Write syntax and example of each function.

Solution

Random Module Functions:

Function	Syntax	Purpose	Example
random()	`random.random()`	Random float 0.0 to 1.0	`0.7534`
randint()	`random.randint(a, b)`	Random integer a to b	`randint(1, 10)`
choice()	`random.choice(seq)`	Random element from sequence	`choice(['a', 'b', 'c'])`

Code Example:

```
import random

\# random() {- generates float between 0.0 and 1.0}
num = random.random()
print(num)  \# Example: 0.7234567

\# randint() {- generates integer between given range}
dice = random.randint(1, 6)
print(dice)  \# Example: 4

\# choice() {- selects random element from sequence}
colors = [{red}, {blue}, {green}]
selected = random.choice(colors)
print(selected)  \# Example: {blue}
```

Mnemonic

"Random Randint Choice"

Question 4(b) [4 marks]

Write the advantages of functions.

Solution

Function Advantages:

Advantage	Description
Code Reusability	Write once, use multiple times
Modularity	Break large program into smaller parts
Easy Debugging	Isolate and fix errors easily
Readability	Makes code more organized and clear
Maintainability	Easy to update and modify
Avoid Repetition	Reduces duplicate code

Example:

```
\# Without function (repetitive)
num1 = 5
square1 = num1 * num1
print(square1)

num2 = 8
square2 = num2 * num2
print(square2)

\# With function (reusable)
def calculate\_square(num):
    return num * num

print(calculate\_square(5))  \# 25
print(calculate\_square(8))  \# 64
```

Mnemonic

"Reuse Modular Debug Read Maintain Avoid"

Question 4(c) [7 marks]

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

Solution

```
\# Input 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}:  \# Check for both {a and A}
        positions.append(i)

\# Display results
if positions:
    print(f"Character {a found at positions: }\{positions\}")
    print("Detailed locations:")
    for pos in positions:
        print(f"Position \{pos\}: {}\{text[pos]\}{"})
else:
    print("Character {a not found in the string"})

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

Sample Output:

```
Enter a string: Python Programming

Character 'a' found at positions: [12]
Detailed locations:
Position 12: 'a'
```

```
Alternative approach:
'a' found at position 12
```

Enhanced Version:

```
text = input("Enter a string: ")
count = 0

print(f"Searching for {a in: }\{text\}{"})
print("{-"} * 30)

for i, char in enumerate(text):
    if char.lower() == {a}:
        count += 1
        print(f"Found {a at index }\{i\} (character: {}\{char\}{)"})

print(f"{n}Total occurrences of {a: }\{count\}")
```

Mnemonic

"Input Loop Check Store Display"

Question 4(a OR) [3 marks]

Explain local and global variables.

Solution

Variable Scope Types:

Variable Type	Scope	Access	Example
Local	Inside function only	Within function	`def func(): x = 5`
Global	Entire program	Anywhere in program	`x = 5` (outside function)

Code Example:

```
\# Global variable
global\_var = "I am global"

def my\_function():
    \# Local variable
    local\_var = "I am local"
    print(global\_var)  \# Can access global
    print(local\_var)   \# Can access local

my\_function()
print(global\_var)      \# Can access global
\# print(local\_var)     \# Error {- cannot access local}
```

Global Keyword:

```
counter = 0  \# Global variable

def increment():
    global counter  \# Declare as global to modify
    counter += 1

increment()
print(counter)  \# Output: 1
```

Mnemonic

"Local Inside Global Everywhere"

Question 4(b OR) [4 marks]

Explain creation and use of user defined function with example.

Solution

Function Creation Syntax:

```
def function\_name(parameters):
    """Optional docstring"""
    \# Function body
    return value  \# Optional
```

Function Components:

Component	Purpose	Example
def	Keyword to define function	`def`
function_name	Name of function	`calculate_area`
parameters	Input values	`(length, width)`
return	Output value	`return result`

Example:

```
\# Function definition
def greet\_user(name, age):
    """Function to greet user with name and age"""
    message = f"Hello \{name\}! You are \{age\} years old."
    return message

\# Function call
user\_name = "John"
user\_age = 25
greeting = greet\_user(user\_name, user\_age)
print(greeting)  \# Output: Hello John! You are 25 years old.

\# Function with default parameter
def calculate\_power(base, exponent=2):
    return base ** exponent

print(calculate\_power(5))     \# 25 (using default exponent=2)
print(calculate\_power(5, 3))  \# 125 (using exponent=3)
```

Mnemonic

"Define Call Return Parameter"

Question 4(c OR) [7 marks]

Write a program to create a user defined function calcFact() to calculate and display the factorial of a number passed as an argument.

Solution

```
def calcFact(number):
    """
    Function to calculate factorial of a number
    Input: number (integer)
    Output: factorial (integer)
    """
    if number {} 0:
        return "Factorial is not defined for negative numbers"
    elif number == 0 or number == 1:
        return 1
    else:
        factorial = 1
        for i in range(2, number + 1):
            factorial *= i
        return factorial

\# Main program
try:
    \# Input from user
    num = int(input("Enter a number: "))

    \# Call function
    result = calcFact(num)

    \# Display result
    if isinstance(result, str):
        print(result)
    else:
```

```
        print(f"Factorial of \{num\} is: \{result\}")

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

\# Test with multiple values
print("{n}Testing with different values:")
test\_values = [0, 1, 5, 10, {-}3]
for val in test\_values:
    result = calcFact(val)
    print(f"calcFact(\{val\}) = \{result\}")
```

Recursive Version:

```
def calcFactRecursive(n):
    """Recursive function to calculate factorial"""
    if n {} 0:
        return "Undefined for negative numbers"
elif

n == 0 or

n == 1:

        return 1
    else:
        return n * calcFactRecursive(n {-} 1)

\# Example usage
number = int(input("Enter number: "))
result = calcFactRecursive(number)
print(f"Factorial: \{result\}")
```

Sample Output:

```
Enter a number: 5
Factorial of 5 is: 120

Testing with different values:
calcFact(0) = 1
calcFact(1) = 1
calcFact(5) = 120
calcFact(10) = 3628800
calcFact(-3) = Factorial is not defined for negative numbers
```

Mnemonic

"Define Check Loop Multiply Return"

Question 5(a) [3 marks]

Give difference between class and object.

Solution

Class vs Object Comparison:

Aspect	Class	Object
Definition	Blueprint/template	Instance of class

Memory	No memory allocated	Memory allocated
Creation	Defined using `class` keyword	Created using class name
Attributes	Defined but not initialized	Have actual values
Example	`class Car:`	`my_car = Car()`

Code Example:

```
\# Class definition (blueprint)
class Student:
    def \_\_init\_\_(self, name, age):
        self.name = name
        self.age = age

\# Object creation (instances)
student1 = Student("John", 20)  \# Object 1
student2 = Student("Alice", 19) \# Object 2

print(student1.name)  \# John
print(student2.name)  \# Alice
```

Mnemonic

"Class Blueprint Object Instance"

Question 5(b) [4 marks]

State the purpose of a constructor in a class.

Solution

Constructor Purpose:

Purpose	Description
Initialize Objects	Set initial values to attributes
Automatic Execution	Called automatically when object created
Memory Setup	Allocate memory for object attributes
Default Values	Provide default values to attributes

Types of Constructors:

Type	Description	Example
Default	No parameters	`def __init__(self):`
Parameterized	Takes parameters	`def __init__(self, name):`

Example:

```
class Rectangle:
    def \_\_init\_\_(self, length=0, width=0):  \# Constructor
        self.length = length  \# Initialize attribute
        self.width = width    \# Initialize attribute
        print("Rectangle object created!")

    def area(self):
        return self.length * self.width

\# Object creation {- constructor called automatically}
rect1 = Rectangle(10, 5)  \# Output: Rectangle object created!
rect2 = Rectangle()       \# Uses default values

print(rect1.area())       \# 50
print(rect2.area())       \# 0
```

Mnemonic

"Initialize Automatic Memory Default"

Question 5(c) [7 marks]

Write a program to create a class "Student" with attributes such as name, roll number, and marks. Implement method to display student information. Create object of the student class and show how to use method.

Solution

```
class Student:
    def \_\_init\_\_(self, name, roll\_number, marks):
        """Constructor to initialize student attributes"""
        self.name = name
        self.roll\_number = roll\_number
        self.marks = marks

    def display\_info(self):
        """Method to display student information"""
        print("{-"} * 30)
        print("STUDENT INFORMATION")
        print("{-"} * 30)
        print(f"Name: \{self.name\}")
        print(f"Roll Number: \{self.roll\_number\}")
        print(f"Marks: \{self.marks\}")
        print("{-"} * 30)

    def calculate\_grade(self):
        """Method to calculate grade based on marks"""
        if self.marks {=} 90:
            return {A+}
        elif self.marks {=} 80:
            return {A}
```

```
        elif self.marks {=} 70:
            return {B}
        elif self.marks {=} 60:
            return {C}
        else:
            return {F}

    def display\_grade(self):
        """Method to display grade"""
        grade = self.calculate\_grade()
        print(f"Grade: \{grade\}")

\# Creating objects of Student class
print("Creating Student Objects:")
student1 = Student("John Doe", 101, 85)
student2 = Student("Alice Smith", 102, 92)
student3 = Student("Bob Johnson", 103, 78)

\# Using methods to display information
print("{n}=== Student 1 Details ===")
student1.display\_info()
student1.display\_grade()

print("{n}=== Student 2 Details ===")
student2.display\_info()
student2.display\_grade()

print("{n}=== Student 3 Details ===")
student3.display\_info()
student3.display\_grade()

\# Accessing attributes directly
print(f"{n}Direct access {- Student 1 name: }\{student1.name\}")
print(f"Direct access {- Student 2 marks: }\{student2.marks\}")
```

Sample Output:

```
Creating Student Objects:

=== Student 1 Details ===
------------------------------
STUDENT INFORMATION
------------------------------
Name: John Doe
Roll Number: 101
Marks: 85
------------------------------
Grade: A

=== Student 2 Details ===
------------------------------
STUDENT INFORMATION
------------------------------
Name: Alice Smith
Roll Number: 102
Marks: 92
------------------------------
Grade: A+
```

Class Components:

- **Attributes**: name, roll_number, marks
- **Constructor**: `__init__()` method

- **Methods**: display_info(), calculate_grade(), display_grade()
- **Objects**: student1, student2, student3

Mnemonic

"Class Attributes Constructor Methods Objects"

Question 5(a OR) [3 marks]

State the purpose of encapsulation.

Solution

Encapsulation Purpose:

Purpose	Description
Data Hiding	Hide internal implementation details
Data Protection	Protect data from unauthorized access
Controlled Access	Provide controlled access through methods
Code Security	Prevent accidental modification of data
Modularity	Keep related data and methods together

Implementation Example:

```
class BankAccount:
    def \_\_init\_\_(self, balance):
        self.\_\_balance = balance  \# Private attribute

    def get\_balance(self):       \# Getter method
        return self.\_\_balance

    def deposit(self, amount):   \# Controlled access
        if amount {} 0:
            self.\_\_balance += amount

account = BankAccount(1000)
print(account.get\_balance())     \# 1000
\# print(account.\_\_balance)       \# Error {- cannot access directly}
```

Benefits:

- **Security**: Data cannot be accessed directly
- **Maintenance**: Easy to modify internal implementation
- **Validation**: Can add validation in getter/setter methods

Mnemonic

"Hide Protect Control Secure Modular"

Question 5(b OR) [4 marks]

Explain multilevel inheritance.

Solution

Multilevel Inheritance is when a class inherits from another class, which in turn inherits from another class, forming a chain.

Structure Diagram:

```
    +{-{-}{-}{-}{-}{-}{-}{-}{-}{-}+}
    | GrandPa  |  (Base Class)
    +{-{-}{-}{-}{-}{-}{-}{-}{-}{-}+}
         \^{}
         |
    +{-{-}{-}{-}{-}{-}{-}{-}{-}{-}+}
    |  Parent  |  (Derived from GrandPa)
    +{-{-}{-}{-}{-}{-}{-}{-}{-}{-}+}
         \^{}
         |
    +{-{-}{-}{-}{-}{-}{-}{-}{-}{-}+}
    |  Child   |  (Derived from Parent)
    +{-{-}{-}{-}{-}{-}{-}{-}{-}{-}+}
```

Characteristics Table:

Level	Class	Inherits From	Access To
Level 1	GrandPa	None	Own methods
Level 2	Parent	GrandPa	GrandPa + Own methods
Level 3	Child	Parent	GrandPa + Parent + Own

Code Example:

```
\# Level 1 {- Base class}
class Vehicle:
    def \_\_init\_\_(self, brand):
        self.brand = brand

    def start(self):
        print(f"\{self.brand\} vehicle started")

\# Level 2 {- Inherits from Vehicle}
class Car(Vehicle):
    def \_\_init\_\_(self, brand, model):
        super().\_\_init\_\_(brand)
        self.model = model

    def drive(self):
        print(f"\{self.brand\} \{self.model\} is driving")

\# Level 3 {- Inherits from Car}
class SportsCar(Car):
    def \_\_init\_\_(self, brand, model, top\_speed):
        super().\_\_init\_\_(brand, model)
        self.top\_speed = top\_speed

    def race(self):
        print(f"\{self.brand\} \{self.model\} racing at \{self.top\_speed\} km/h")

\# Creating object and using methods
ferrari = SportsCar("Ferrari", "F8", 340)
ferrari.start()    \# From Vehicle class
ferrari.drive()    \# From Car class
ferrari.race()     \# From SportsCar class
```

Mnemonic

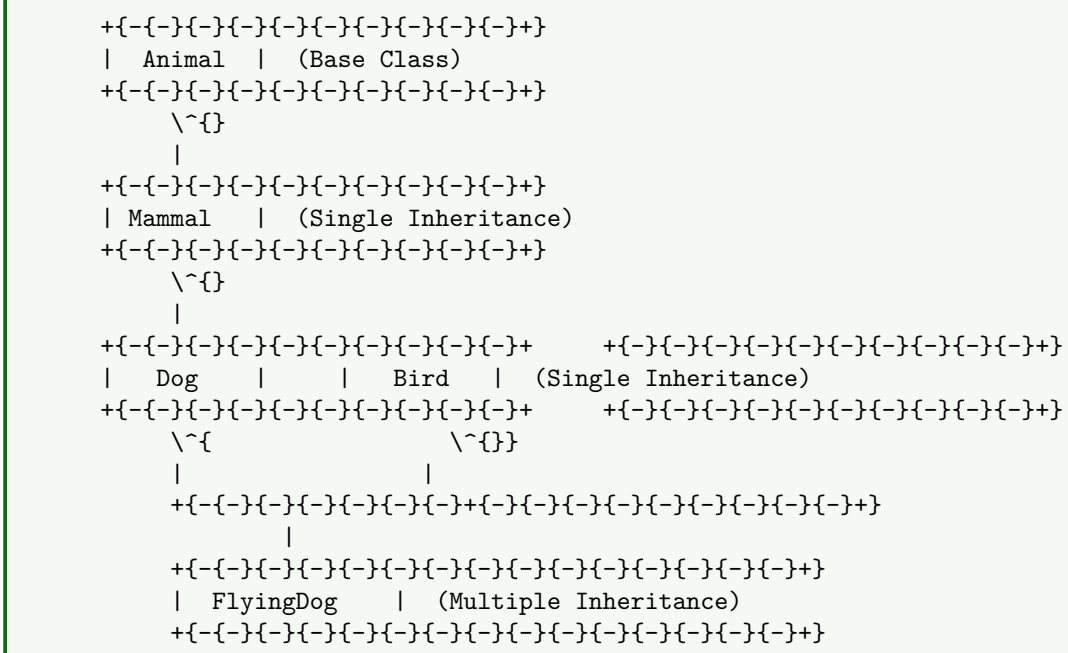
"Chain Inherit Level Access"

Question 5(c OR) [7 marks]

Write a Python program to demonstrate working of hybrid inheritance.

Solution

Hybrid Inheritance combines multiple types of inheritance (single, multiple, multilevel) in one program.
Structure Diagram:



Code Example:

```
\# Base class
class Animal:
    def \_\_init\_\_(self, name):
        self.name = name
        print(f"Animal \{self.name\} created")

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

    def sleep(self):
        print(f"\{self.name\} is sleeping")

\# Single inheritance from Animal
class Mammal(Animal):
    def \_\_init\_\_(self, name, fur\_color):
        super().\_\_init\_\_(name)
        self.fur\_color = fur\_color

    def give\_birth(self):
        print(f"\{self.name\} gives birth to live babies")

\# Single inheritance from Animal
class Bird(Animal):
    def \_\_init\_\_(self, name, wing\_span):
        super().\_\_init\_\_(name)
```

```
        self.wing\_span = wing\_span

    def fly(self):
        print(f"\{self.name\} is flying with \{self.wing\_span\}cm wings")

    def lay\_eggs(self):
        print(f"\{self.name\} lays eggs")

\# Single inheritance from Mammal
class Dog(Mammal):
    def \_\_init\_\_(self, name, fur\_color, breed):
        super().\_\_init\_\_(name, fur\_color)
        self.breed = breed

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

    def guard(self):
        print(f"\{self.name\} is guarding the house")

\# Multiple inheritance from Dog and Bird (Hybrid)
class FlyingDog(Dog, Bird):
    def \_\_init\_\_(self, name, fur\_color, breed, wing\_span):
        \# Initialize both parent classes
        Dog.\_\_init\_\_(self, name, fur\_color, breed)
        Bird.\_\_init\_\_(self, name, wing\_span)
        print(f"Magical \{self.name\} created with both mammal and bird features!")

    def fly\_and\_bark(self):
        print(f"\{self.name\} is flying and barking at the same time!")

    def show\_abilities(self):
        print(f"{n}\{self.name\}{s Abilities:"})
        print("{-"} * 25)
        self.eat()          \# From Animal
        self.sleep()        \# From Animal
        self.give\_birth()   \# From Mammal
        self.bark()         \# From Dog
        self.guard()        \# From Dog
        self.fly()          \# From Bird
        self.lay\_eggs()     \# From Bird
        self.fly\_and\_bark() \# Own method

\# Demonstration
print("=== Hybrid Inheritance Demo ==={n}")

\# Create objects
print("1. Creating regular dog:")
dog1 = Dog("Buddy", "Golden", "Retriever")
dog1.bark()
dog1.guard()

print("{n}2. Creating regular bird:")
bird1 = Bird("Eagle", 200)
bird1.fly()
bird1.lay\_eggs()

print("{n}3. Creating magical flying dog:")
flying\_dog = FlyingDog("Superdog", "Silver", "Husky", 150)
flying\_dog.show\_abilities()
```

```
\# Method Resolution Order
print(f"{n}Method Resolution Order for FlyingDog:")
for i, cls in enumerate(FlyingDog.\_\_mro\_\_):
    print(f"\{i+1\}. \{cls.\_\_name\_\_\}")
```

Sample Output:

```
=== Hybrid Inheritance Demo ===

1. Creating regular dog:
Animal Buddy created
Buddy the Retriever is barking
Buddy is guarding the house

2. Creating regular bird:
Animal Eagle created
Eagle is flying with 200cm wings
Eagle lays eggs

3. Creating magical flying dog:
Animal Superdog created
Animal Superdog created
Magical Superdog created with both mammal and bird features!

Superdog's Abilities:
-------------------------
Superdog is eating
Superdog is sleeping
Superdog gives birth to live babies
Superdog the Husky is barking
Superdog is guarding the house
Superdog is flying with 150cm wings
Superdog lays eggs
Superdog is flying and barking at the same time!
```

Inheritance Types in This Example:

1. **Single**: Mammal $\leftarrow$ $Animal, Bird \leftarrow Animal, Dog \leftarrow Mammal$
1. **Multiple**: FlyingDog $\leftarrow Dog + Bird$
1. **Multilevel**: FlyingDog $\leftarrow Dog \leftarrow Mammal \leftarrow Animal$
1. **Hybrid**: Combination of all above

Key Features:

- **Multiple Parent Classes**: FlyingDog inherits from both Dog and Bird
- **Method Resolution Order**: Python follows MRO to resolve method conflicts
- **Super() Usage**: Proper initialization of parent classes
- **Combined Functionality**: Access to methods from all parent classes

Mnemonic

"Hybrid Multiple Single Multilevel Combined"