Subject Name Solutions

4351108 – Summer 2024

Semester 1 Study Material

Detailed Solutions and Explanations

Question 1(a) [3 marks]

Explain for loop working in Python.

Solution

For loop repeats code block for each item in sequence like list, tuple, or string.

Syntax Table:

Component	Syntax	Example
Basic	`for variable in sequence:`	`for i in [1,2,3]:`
Range	`for i in range(n):`	`for i in range(5):`
String	`for char in string:`	`for c in "hello":`

Diagram:

```
Start {-{-} Check if items left in sequence}
         |
         v
    Execute loop body
         |
         v
    Move to next item {-{-} Check if items left}
         |                      |
         v                      v
    Items left? {-{-}{-}{-}No{-}{-}{-}{-} End}
         |
        Yes
         |
         v
    Back to Execute loop body
```

- **Iteration**: Loop variable gets each value from sequence one by one
- **Automatic**: Python handles moving to next item automatically
- **Flexible**: Works with lists, strings, tuples, ranges

Mnemonic

"For Each Item, Execute Block"

Question 1(b) [4 marks]

Explain working of if-elif-else in Python.

Solution

Multi-way decision structure that checks multiple conditions in sequence.

Structure Table:

Statement	Purpose	Syntax
if	First condition	`if condition1:`
elif	Alternative conditions	`elif condition2:`
else	Default case	`else:`

Flow Diagram:

```
    Start
      |
      v
 Check if condition
      |
   True/ {False}
     /    {}
    v      v
Execute   Check elif
if block   condition
    |         |
    v      True/ {False}
   End       /    {}
            v      v
       Execute   Check next elif
       elif      or else
       block        |
          |         v
          v      Execute
         End     else block
                    |
                    v
                   End
```

- **Sequential**: Checks conditions top to bottom
- **Exclusive**: Only one block executes
- **Optional**: elif and else are optional

Mnemonic

"If This, Else If That, Else Default"

Question 1(c) [7 marks]

Explain structure of a Python Program.

Solution

Python program has organized structure with specific components in logical order.
Program Structure Table:

Component	Purpose	Example
Comments	Documentation	`# This is comment`
Import	External modules	`import math`
Constants	Fixed values	`PI = 3.14`
Functions	Reusable code	`def function_name():`
Classes	Objects blueprint	`class ClassName:`
Main code	Program execution	`if __name__ == "__main__":`

Program Architecture:

```
         Comments
       \# Documentation


             v


        Import Section
       import modules


             v


        Constants \&
         Variables


             v


       Function
       Definitions


             v


       Class
       Definitions


             v


       Main Program
         Execution
```

- **Modular**: Each section has specific purpose
- **Readable**: Clear organization helps understanding
- **Maintainable**: Easy to modify and debug
- **Standard**: Follows Python conventions

Simple Example:

```
\# Program to calculate area
import math

PI = 3.14159

def calculate\_area(radius):
    return PI * radius * radius

\# Main execution
radius = float(input("Enter radius: "))
area = calculate\_area(radius)
print(f"Area = \{area\}")
```

Mnemonic

"Comment, Import, Constant, Function, Class, Main"

Question 1(c OR) [7 marks]

Explain features of Python Programming Language.

Solution

Python has unique characteristics that make it popular for beginners and professionals.

Python Features Table:

Feature	Description	Benefit
Simple	Easy syntax	Quick learning
Interpreted	No compilation	Fast development
Object-Oriented	Classes and objects	Code reusability
Open Source	Free to use	No licensing cost
Cross-Platform	Runs everywhere	High portability

Feature Categories:

```
         Python Features




    v         v         v
Language   Technical  Community
Features   Features   Features


    v         v         v
{- Simple   {-} Interpreted {-} Open Source}
{- Readable {-} Portable    {-} Large Library}
{- Dynamic  {-} Extensible  {-} Active Support}
```

- **Beginner-Friendly**: Simple syntax like English language
- **Versatile**: Used for web, AI, data science, automation
- **Rich Libraries**: Huge collection of pre-built modules
- **Dynamic Typing**: No need to declare variable types
- **Interactive**: Can test code line by line in interpreter
- **High-Level**: Handles memory management automatically

Code Example:

```
\# Simple Python syntax
name = "Python"
print(f"Hello, \{name\}!")
```

Mnemonic

"Simple, Interpreted, Object-Oriented, Open, Cross-platform"

Question 2(a) [3 marks]

Explain any 3 operations done on Strings.

Solution

String operations manipulate and process text data in various ways.

String Operations Table:

Operation	Method	Example	Result
Concatenation	`+`	`"Hello" + "World"`	`"HelloWorld"`
Length	`len()`	`len("Python")`	`6`

Uppercase	`.upper()`	`"hello".upper()`	`"HELLO"`

Operation Examples:

```
text = "Python"
\# 1. Concatenation
result1 = text + " Programming"
\# 2. Find length
result2 = len(text)
\# 3. Convert to uppercase
result3 = text.upper()
```

- **Concatenation**: Joins two or more strings together
- **Length**: Counts total characters in string
- **Case Conversion**: Changes letter cases (upper/lower)

Mnemonic

"Combine, Count, Convert"

Question 2(b) [4 marks]

Develop a Python program to convert temperature from Fahrenheit to Celsius unit using eq: C=(F-32)/1.8

Solution

Program converts temperature using mathematical formula with user input.

Algorithm Table:

Step	Action	Code
1	Get input	`fahrenheit = float(input())`
2	Apply formula	`celsius = (fahrenheit - 32) / 1.8`
3	Display result	`print(f"Celsius: {celsius}")`

Complete Program:

```
\# Temperature conversion program
fahrenheit = float(input("Enter temperature in Fahrenheit: "))
celsius = (fahrenheit {-} 32) / 1.8
print(f"Temperature in Celsius: \{celsius:.2f\}")
```

Test Cases:

- Input: $32^{\rightarrow}Output : 0.00$
- Input: $100^{\rightarrow}Output : 37.78$
- **User Input**: Gets Fahrenheit temperature from user
- **Formula Application**: Uses given conversion equation
- **Formatted Output**: Shows result with decimal places

Mnemonic

"Input, Calculate, Output"

Question 2(c) [7 marks]

Explain in detail working of list data types in Python.

Solution

List is ordered, mutable collection that stores multiple items in single variable.

List Characteristics Table:

Property	Description	Example
Ordered	Items have position	`[1, 2, 3]`
Mutable	Can be changed	`list[0] = 10`
Indexed	Access by position	`list[0]`
Mixed Types	Different data types	`[1, "hello", 3.14]`

List Operations Diagram:

```
List: [10, 20, 30, 40]
       |   |   |   |
Index: 0   1   2   3

Operations:

   Access          Modify
  list[0]        list[0]=50



      v                 v
    "10"          [50, 20, 30, 40]
```

Common List Methods:

Method	Purpose	Example
append()	Add item at end	`list.append(5)`
insert()	Add at position	`list.insert(1, 15)`
remove()	Delete item	`list.remove(20)`
pop()	Remove last item	`list.pop()`
len()	Get length	`len(list)`

Example Code:

```
\# Creating and using lists
numbers = [1, 2, 3, 4, 5]
numbers.append(6)         \# Add 6 at end
numbers.insert(0, 0)      \# Add 0 at beginning
print(numbers[2])         \# Access 3rd element
numbers.remove(3)         \# Remove value 3
```

- **Dynamic Size**: Can grow or shrink during execution
- **Zero Indexing**: First element at index 0
- **Slicing**: Can extract portions using [start:end]
- **Nested Lists**: Can contain other lists

Mnemonic

"Ordered, Mutable, Indexed, Mixed"

Question 2(a OR) [3 marks]

Explain String formatting in Python.

Solution

String formatting creates formatted strings by inserting values into templates.
Formatting Methods Table:

Method	Syntax	Example
f-strings	`f"text {variable}"`	`f"Hello {name}"`
format()	`"text {}".format(value)`	`"Age: {}".format(25)`
% operator	`"text %s" % value`	`"Name: %s" % "John"`

Example Usage:

```
name = "Alice"
age = 25
\# f{-string formatting}
message = f"Hello \{name\}, you are \{age\} years old"
```

- **Placeholder**: {} marks where values go
- **Dynamic**: Values inserted at runtime
- **Readable**: Makes code cleaner than concatenation

Mnemonic

"Format, Insert, Display"

Question 2(b OR) [4 marks]

Develop a Python program to identify whether the scanned number is even or odd and print an appropriate message.

Solution

Program checks if number is divisible by 2 to determine even or odd.
Logic Table:

Condition	Result	Message
number % 2 == 0	Even	"Number is even"
number % 2 != 0	Odd	"Number is odd"

Complete Program:

```
\# Even/Odd checker program
number = int(input("Enter a number: "))
if number \% 2 == 0:
    print(f"\{number\} is even")
else:
    print(f"\{number\} is odd")
```

Test Cases:

- Input: 4 → $Output: “4 is even''$
- Input: 7 → $Output: “7 is odd''$
- **Modulo Operator**: % gives remainder after division
- **Conditional Logic**: if-else determines result
- **User Feedback**: Clear message about result

Mnemonic

"Input, Check Remainder, Display Result"

Question 2(c OR) [7 marks]

Explain in detail working of Set data types in Python.

Solution

Set is unordered collection of unique items with no duplicate values allowed.

Set Characteristics Table:

Property	Description	Example
Unordered	No fixed position	`{1, 3, 2}`
Unique	No duplicates	`{1, 2, 3}`
Mutable	Can be modified	`set.add(4)`
Iterable	Can loop through	`for item in set:`

Set Operations Diagram:

```
    Set A: \{1, 2, 3\    Set B: \{3, 4, 5\}}
         {                    /}
          {                  /}
           v                v


          Set Operations

      Union: \{1, 2, 3, 4, 5\       }
      Intersection: \{3\            }
      Difference: \{1, 2\           }
      Symmetric Diff: \{1,2,4,5\    }
```

Set Methods Table:

Method	Purpose	Example
add()	Add single item	`set.add(6)`
update()	Add multiple items	`set.update([7, 8])`
remove()	Delete item	`set.remove(3)`
union()	Combine sets	`set1.union(set2)`
intersection()	Common items	`set1.intersection(set2)`

Example Code:

```
\# Creating and using sets
fruits = \{"apple", "banana", "orange"\}
fruits.add("mango")              \# Add single item
fruits.update(["grape", "kiwi"]) \# Add multiple
fruits.remove("banana")          \# Remove item
print(len(fruits))               \# Count items
```

- **Automatic Deduplication**: Removes duplicate values automatically
- **Fast Membership**: Quick checking if item exists
- **Mathematical Operations**: Union, intersection, difference
- **No Indexing**: Cannot access items by position

Mnemonic

"Unique, Unordered, Mutable, Mathematical"

Question 3(a) [3 marks]

Explain working of any 3 methods of math module.

Solution

Math module provides mathematical functions for complex calculations.
Math Methods Table:

Method	Purpose	Example	Result
math.sqrt()	Square root	`math.sqrt(16)`	`4.0`
math.pow()	Power calculation	`math.pow(2, 3)`	`8.0`
math.ceil()	Round up	`math.ceil(4.3)`	`5`

Usage Example:

```
import math
number = 16
result1 = math.sqrt(number)  \# Square root
result2 = math.pow(2, 4)     \# 2 to power 4
result3 = math.ceil(7.2)     \# Round up to 8
```

- **Precision**: More accurate than basic operators
- **Import Required**: Must import math module first
- **Return Values**: Usually return float numbers

Mnemonic

"Square root, Power, Ceiling"

Question 3(b) [4 marks]

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

Solution

Program iterates through list and accumulates sum of all elements.
Algorithm Table:

Step	Action	Code
1	Initialize sum	`total = 0`
2	Loop through list	`for element in list:`
3	Add to sum	`total += element`
4	Display result	`print(total)`

Complete Program:

```
\# Sum of list elements
numbers = [10, 20, 30, 40, 50]
total = 0
for element in numbers:
    total += element
print(f"Sum of all elements: \{total\}")
```

Test Case:

- Input: [1, 2, 3, 4, 5] $\rightarrow Output : 15$
- **Accumulator**: Variable stores running total
- **Iteration**: Loop visits each element once
- **Addition**: Adds each element to running sum

Mnemonic

"Initialize, Loop, Add, Display"

Question 3(c) [7 marks]

Develop a Python program to check if two lists are having similar length. If yes then merge them and create a dictionary from them.

Solution

Program compares list lengths and creates dictionary if they match.

Logic Flow Table:

Step	Condition	Action
1	Check lengths	`len(list1) == len(list2)`
2	If equal	Merge and create dictionary
3	If not equal	Display error message

Process Diagram:

```
List1: [a, b, c]     List2: [1, 2, 3]
   |                      |
   v                      v
 len(List1) == len(List2) ?
         |
    Yes / { No}
       /   {}
      v     v
Create Dict  Error
\{a:1, b:2,   Message
 c:3\}
```

Complete Program:

```
\# Merge lists into dictionary
list1 = [{name}, {age}, {city}]
list2 = [{John}, 25, {Mumbai}]

if len(list1) == len(list2):
    \# Create dictionary using zip
    result\_dict = dict(zip(list1, list2))
    print("Dictionary created:", result\_dict)
else:
    print("Lists have different lengths, cannot merge")
```

Expected Output:

```
Dictionary created: {'name': 'John', 'age': 25, 'city': 'Mumbai'}
```

- **Length Comparison**: Ensures lists can be paired properly
- **zip() Function**: Pairs elements from both lists
- **dict() Constructor**: Creates dictionary from paired elements
- **Error Handling**: Prevents incorrect pairing

Alternative Method:

```
\# Manual dictionary creation
result\_dict = \{\}
for i in range(len(list1)):
    result\_dict[list1[i]] = list2[i]
```

Mnemonic

"Check Length, Zip, Create Dictionary"

Question 3(a OR) [3 marks]

Explain working of any 3 methods of statistics module.

Solution

Statistics module provides functions for statistical calculations on numeric data.
Statistics Methods Table:

Method	Purpose	Example	Result
statistics.mean()	Average value	`mean([1,2,3,4,5])`	`3.0`
statistics.median()	Middle value	`median([1,2,3,4,5])`	`3`
statistics.mode()	Most frequent	`mode([1,1,2,3])`	`1`

Usage Example:

```
import statistics
data = [10, 20, 30, 40, 50]
avg = statistics.mean(data)      \# Calculate average
mid = statistics.median(data)    \# Find middle value
```

- **Data Analysis**: Helps understand data patterns
- **Built-in Functions**: No need to write complex formulas
- **Accurate Results**: Handles edge cases properly

Mnemonic

"Mean, Median, Mode"

Question 3(c OR) [7 marks]

Develop a Python program to count the number of times a character appears in a given string using a dictionary.

Solution

Program creates dictionary where keys are characters and values are their counts.
Character Counting Algorithm:

Step	Action	Code
1	Initialize dictionary	`char_count = {}`
2	Loop through string	`for char in string:`
3	Count occurrences	`char_count[char] = char_count.get(char, 0) + 1`
4	Display results	`print(char_count)`

Counting Process:

```
    String: "hello"
         |
         v
    Loop through each character
         |

        h      e    l    l    o

         v
    Dictionary: \{{h:1, e:1, l:2, o:1\}}
```

Complete Program:

```
\# Character frequency counter
text = input("Enter a string: ")
char\_count = \{\}

for char in text:
    if char in char\_count:
        char\_count[char] += 1
    else:
        char\_count[char] = 1

print("Character frequencies:")
for char, count in char\_count.items():
    print(f"{}\{char\}{: }\{count\}")
```

Alternative Method (More Pythonic):

```
\# Using get() method
text = "programming"
char\_count = \{\}

for char in text:
    char\_count[char] = char\_count.get(char, 0) + 1

print(char\_count)
```

Example Output:

```
Input: "hello"
Output: {'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

- **Dictionary Keys**: Each unique character becomes a key
- **Dictionary Values**: Count of character occurrences
- **get() Method**: Returns 0 if key doesn't exist, avoiding errors
- **Iteration**: Processes each character in string once

Mnemonic

"Loop, Check, Count, Store"

Question 4(a) [3 marks]

Explain working of Python class and objects with example.

Solution

Class is blueprint for creating objects. Objects are instances of classes.
Class-Object Relationship:

Concept	Purpose	Example
Class	Template/Blueprint	`class Car:`
Object	Instance of class	`my_car = Car()`
Attributes	Data in class	`self.color = "red"`
Methods	Functions in class	`def start(self):`

Class Structure:

```
         Class: Car

       Attributes:
       {- color           }
       {- model           }

       Methods:
       {- start()         }
       {- stop()          }


            v
    Object: my\_car = Car()
```

Example Code:

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

    def display(self):    \# Method
        print(f"Name: \{self.name\}, Age: \{self.age\}")

\# Creating objects
student1 = Student("Alice", 20)
student1.display()
```

- **Encapsulation**: Groups related data and functions together
- **Reusability**: One class can create multiple objects
- **Organization**: Better code structure and maintenance

Mnemonic

"Class Blueprint, Object Instance"

Question 4(b) [4 marks]

Develop a Python program to print all odd numbers in a list.

Solution

Program filters list elements and displays only odd numbers.
Odd Number Check Table:

Number	number % 2	Result
1	1	Odd
2	0	Even
3	1	Odd
4	0	Even

Complete Program:

```
\# Print odd numbers from list
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print("Odd numbers in the list:")
for number in numbers:
    if number \% 2 != 0:
        print(number, end=" ")
```

Alternative Methods:

```
\# Method 2: List comprehension
odd\_numbers = [num for num in numbers if num \% 2 != 0]
print(odd\_numbers)

\# Method 3: Using filter
odd\_numbers = list(filter(lambda x: x \% 2 != 0, numbers))
print(odd\_numbers)
```

Expected Output:

```
Odd numbers in the list:
1 3 5 7 9
```

- **Modulo Operation**: % operator finds remainder
- **Condition Check**: If remainder is not 0, number is odd
- **Loop Iteration**: Checks each number in list

Mnemonic

"Loop, Check Remainder, Print Odd"

Question 4(c) [7 marks]

Explain working of user defined functions in Python.

Solution

User-defined functions are custom functions created by programmers to perform specific tasks.
Function Components Table:

Component	Purpose	Syntax
def keyword	Function declaration	`def function_name():`
Parameters	Input values	`def func(param1, param2):`
Body	Function code	Indented statements
return	Output value	`return value`

Function Structure:

```
    def function\_name(parameters):

                            Input values
                   Function identifier
            Keyword to define function

    Function Body (indented)

          v

       Local variables
       Processing logic
       Calculations


          v
    return result (optional)
```

Types of Functions:

Type	Description	Example
No parameters	Takes no input	`def greet():`
With parameters	Takes input	`def add(a, b):`
Return value	Gives output	`return a + b`
No return	Performs action	`print("Hello")`

Example Functions:

```
\# Function with no parameters
def greet():
    print("Hello, World!")

\# Function with parameters and return value
def calculate\_area(length, width):
    area = length * width
    return area

\# Function with default parameters
def introduce(name, age=18):
    print(f"My name is \{name\} and I am \{age\} years old")

\# Using functions
greet()
result = calculate\_area(5, 3)
print(f"Area: \{result\}")
introduce("Alice", 25)
introduce("Bob")  \# Uses default age
```

Function Benefits:

- **Reusability**: Write once, use multiple times
- **Modularity**: Break complex problems into smaller parts
- **Maintainability**: Easy to update and debug
- **Readability**: Makes code more organized and understandable
- **Testing**: Can test individual functions separately

Variable Scope:

- **Local Variables**: Exist only inside function
- **Global Variables**: Accessible throughout program
- **Parameters**: Act as local variables

Mnemonic

“Define, Parameters, Body, Return”

Question 4(a OR) [3 marks]

Explain working constructors in Python.

Solution

Constructor is special method that initializes objects when they are created.
Constructor Details Table:

Aspect	Description	Syntax
Method name	Always `__init__`	`def __init__(self):`
Purpose	Initialize object	Set initial values
Automatic call	Called during object creation	`obj = Class()`
Parameters	Can accept arguments	`def __init__(self, param):`

Constructor Example:

```
class Student:
    def \_\_init\_\_(self, name, age):
        self.name = name
        self.age = age
        print("Student object created")

\# Object creation automatically calls constructor
student1 = Student("Alice", 20)
```

- **Automatic Execution**: Runs immediately when object is created
- **Initialization**: Sets up object’s initial state
- **self Parameter**: Refers to current object being created

Mnemonic

“Initialize, Automatic, Self”

Question 4(b OR) [4 marks]

Develop a Python program to find smallest number in a list without using min function.

Solution

Program manually compares all elements to find the smallest value.
Finding Minimum Algorithm:

Step	Action	Code
1	Assume first is smallest	`smallest = list[0]`
2	Compare with others	`for num in list[1:]:`
3	Update if smaller found	`if num < smallest:`
4	Display result	`print(smallest)`

Complete Program:

```
\# Find smallest number without min()
numbers = [45, 23, 67, 12, 89, 5, 34]

smallest = numbers[0]  \# Assume first is smallest

for i in range(1, len(numbers)):
    if numbers[i] {} smallest:
        smallest = numbers[i]

print(f"Smallest number: \{smallest\}")
```

Alternative Method:

```
\# Using for loop with list elements
numbers = [45, 23, 67, 12, 89, 5, 34]
smallest = numbers[0]

for num in numbers[1:]:
    if num {} smallest:
        smallest = num

print(f"Smallest number: \{smallest\}")
```

Expected Output:

```
Smallest number: 5
```

- **Comparison Logic**: Compare each element with current smallest
- **Update Strategy**: Replace smallest when smaller number found
- **Linear Search**: Check all elements once

Mnemonic

"Assume, Compare, Update, Display"

Question 4(c OR) [7 marks]

Explain working of user defined Modules in Python.

Solution

User-defined modules are custom Python files containing functions, classes, and variables that can be imported and used in other programs.

Module Components Table:

Component	Purpose	Example
Functions	Reusable code blocks	`def calculate_area():`
Classes	Object blueprints	`class Shape:`
Variables	Shared data	`PI = 3.14159`
Constants	Fixed values	`MAX_SIZE = 100`

Module Creation Process:

```
    Step 1: Create .py file
         |
         v
    Step 2: Write functions/classes
         |
         v
    Step 3: Save file
         |
         v
    Step 4: Import in other programs
         |
         v
    Step 5: Use module functions
```

Example Module Creation:
File: math_operations.py

```
\# User{-defined module}
PI = 3.14159

def calculate\_circle\_area(radius):
    return PI * radius * radius

def calculate\_rectangle\_area(length, width):
    return length * width

class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a {-} b
```

Using the Module:
Import Methods Table:

Method	Syntax	Usage
Import entire module	`import math_operations`	`math_operations.calculate_circle_area(`
Import specific function	`from math_operations import calculate_circle_area`	`calculate_circle_area(5)`
Import with alias	`import math_operations as math_ops`	`math_ops.PI`
Import all	`from math_operations import *`	`calculate_circle_area(5)`

Main Program:

```
\# main.py {- Using the module}
import math\_operations

\# Using module functions
radius = 5
area = math\_operations.calculate\_circle\_area(radius)
print(f"Circle area: \{area\}")

\# Using module variables
print(f"PI value: \{math\_operations.PI\}")

\# Using module classes
calc = math\_operations.Calculator()
result = calc.add(10, 20)
print(f"Addition result: \{result\}")
```

Module Benefits:

- **Code Reusability**: Write once, use in multiple programs
- **Organization**: Keep related functions together
- **Namespace**: Avoid naming conflicts
- **Maintainability**: Easy to update and debug
- **Collaboration**: Share modules with other developers

Module Search Path:

1. Current directory
2. PYTHONPATH environment variable
3. Standard library directories
4. Site-packages directory

Best Practices:

- Use descriptive module names
- Include docstrings for documentation
- Keep related functionality together
- Avoid circular imports

Mnemonic

"Create File, Define Functions, Import, Use"

Question 5(a) [3 marks]

Explain single inheritance in Python with example.

Solution

Single inheritance is when one class inherits properties and methods from exactly one parent class.

Inheritance Structure Table:

Component	Role	Example
Parent Class	Base/Super class	`class Animal:`
Child Class	Derived/Sub class	`class Dog(Animal):`
Inheritance	`class Child(Parent):`	`class Dog(Animal):`

Inheritance Diagram:

```
    Parent Class: Animal

        Attributes:
        {- name               }
        {- age                }

        Methods:
        {- eat()              }
        {- sleep()            }


                inherits
              v
    Child Class: Dog

      Inherited:
        {- name, age          }
        {- eat(), sleep()     }

      Own Methods:
        {- bark()             }
```

Example Code:

```
\# Parent class
class Animal:
    def \_\_init\_\_(self, name):
        self.name = name

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

\# Child class inheriting from Animal
class Dog(Animal):
    def bark(self):
        print(f"\{self.name\} is barking")

\# Using inheritance
my\_dog = Dog("Buddy")
my\_dog.eat()    \# Inherited method
my\_dog.bark()   \# Own method
```

- **Code Reuse**: Child class gets parent's functionality automatically
- **Extension**: Child can add new methods and attributes
- **Is-a Relationship**: Dog is-a Animal

Mnemonic

"One Parent, One Child"

Question 5(b) [4 marks]

Explain concept of abstraction in Python with its advantages.

Solution

Abstraction hides complex implementation details and shows only essential features to the user.

Abstraction Concepts Table:

Concept	Description	Example
Abstract Class	Cannot be instantiated	`class Shape(ABC):`
Abstract Method	Must be implemented	`@abstractmethod`
Interface	Defines method structure	`def area(self):`

Abstraction Implementation:

```
from abc import ABC, abstractmethod

\# Abstract class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

\# Concrete class
class Rectangle(Shape):
    def \_\_init\_\_(self, length, width):
        self.length = length
        self.width = width

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

    def perimeter(self):
        return 2 * (self.length + self.width)
```

Advantages Table:

Advantage	Description	Benefit
Simplicity	Hide complex details	Easier to use
Security	Hide internal implementation	Data protection
Maintainability	Change implementation without affecting users	Flexible updates
Code Organization	Clear structure	Better design

- **Hide Complexity**: Users don't need to know internal workings
- **Consistent Interface**: All child classes follow same structure
- **Force Implementation**: Abstract methods must be defined in child classes

Mnemonic

"Hide Details, Show Interface"

Question 5(c) [7 marks]

Develop a Python program to demonstrate working of multiple and multi-level inheritances.

Solution

Program shows both inheritance types: multiple (multiple parents) and multi-level (chain of inheritance).
Inheritance Types Comparison:

Type	Structure	Example
Multiple	Child inherits from 2+ parents	`class C(A, B):`
Multi-level	Grandparent $\rightarrow Parent \rightarrow Child$	`class C(B):` where `class B(A):`

Inheritance Hierarchy:

```
Multiple Inheritance:
    Father    Mother
      {       /}
       {     /}
        Child

Multi{-level Inheritance:}
    Animal
      |
      v
    Mammal
      |
      v
     Dog
```

Complete Program:

```
\# Multi{-level Inheritance Demo}
print("=== Multi{-level Inheritance ==="})

class Animal:
    def \_\_init\_\_(self, name):
        self.name = name

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

class Mammal(Animal):  \# Inherits from Animal
    def breathe(self):
        print(f"\{self.name\} breathes air")

class Dog(Mammal):      \# Inherits from Mammal (which inherits from Animal)
    def bark(self):
        print(f"\{self.name\} can bark")

\# Using multi{-level inheritance}
my\_dog = Dog("Buddy")
my\_dog.eat()     \# From Animal (grandparent)
my\_dog.breathe() \# From Mammal (parent)
my\_dog.bark()    \# Own method

print("{n}=== Multiple Inheritance ===")

class Father:
    def father\_method(self):
        print("Method from Father class")

class Mother:
    def mother\_method(self):
        print("Method from Mother class")

class Child(Father, Mother):  \# Inherits from both Father and Mother
    def child\_method(self):
        print("Method from Child class")

\# Using multiple inheritance
child = Child()
child.father\_method()  \# From Father
child.mother\_method()  \# From Mother
child.child\_method()   \# Own method
```

```
\# Checking inheritance
print(f"{n}Child inherits from Father: \{issubclass(Child, Father)\}")
print(f"Child inherits from Mother: \{issubclass(Child, Mother)\}")
```

Expected Output:

```
=== Multi-level Inheritance ===
Buddy can eat
Buddy breathes air
Buddy can bark

=== Multiple Inheritance ===
Method from Father class
Method from Mother class
Method from Child class

Child inherits from Father: True
Child inherits from Mother: True
```

Key Differences:

Aspect	Multiple	Multi-level
Parents	2 or more direct parents	Single parent chain
Syntax	`class C(A, B):`	`class C(B):` where `B(A):`
Inheritance	Horizontal	Vertical
Complexity	Higher (diamond problem)	Lower

Method Resolution Order (MRO):

- **Multiple**: Python follows left-to-right order
- **Multi-level**: Goes up the inheritance chain

Mnemonic

"Multiple Parents, Multi-level Chain"

Question 5(a OR) [3 marks]

Explain working of 3 types of methods in Python.

Solution

Python classes have three types of methods based on how they access class data.

Method Types Table:

Method Type	Decorator	First Parameter	Purpose
Instance Method	None	`self`	Access instance data
Class Method	`@classmethod`	`cls`	Access class data
Static Method	`@staticmethod`	None	Utility functions

Example Code:

```
class Student:
    school\_name = "ABC School"  \# Class variable

    def \_\_init\_\_(self, name):
        self.name = name        \# Instance variable

    \# Instance method
    def display\_info(self):
        print(f"Student: \{self.name\}")

    \# Class method
    @classmethod
    def get\_school(cls):
        return cls.school\_name

    \# Static method
    @staticmethod
    def is\_adult(age):
        return age {=} 18

\# Usage
student = Student("Alice")
student.display\_info()           \# Instance method
print(Student.get\_school())      \# Class method
print(Student.is\_adult(20))      \# Static method
```

- **Instance Methods**: Work with object-specific data using `self`
- **Class Methods**: Work with class-wide data using `cls`
- **Static Methods**: Independent utility functions

Mnemonic

"Instance Self, Class Cls, Static None"

Question 5(b OR) [4 marks]

Explain polymorphism through inheritance in Python.

Solution

Polymorphism allows objects of different classes to be treated as objects of common base class, with each implementing methods differently.

Polymorphism Concept Table:

Aspect	Description	Example
Same Interface	Common method names	`area()` method
Different Implementation	Each class has own version	Rectangle vs Circle area
Runtime Decision	Method chosen during execution	Dynamic binding

Polymorphism Example:

```
\# Base class
class Shape:
    def area(self):
        pass

\# Different implementations
class Rectangle(Shape):
    def \_\_init\_\_(self, length, width):
        self.length = length
        self.width = width

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

class Circle(Shape):
    def \_\_init\_\_(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

\# Polymorphic behavior
shapes = [Rectangle(5, 3), Circle(4)]

for shape in shapes:
    print(f"Area: \{shape.area()\}")  \# Same method, different results
```

Benefits:

- **Flexibility**: Same code works with different object types
- **Extensibility**: Easy to add new classes without changing existing code
- **Maintainability**: Changes in one class don't affect others

Mnemonic

"Same Name, Different Behavior"

Question 5(c OR) [7 marks]

Develop a Python program to demonstrate working of hybrid inheritance.

Solution

Hybrid inheritance combines multiple and multi-level inheritance in single program structure.

Hybrid Inheritance Structure:

```
        Animal (Base)
          |
          v
        Mammal
       /      {}
      v        v
     Dog      Cat
       {      /}
        {    /}
         v  v
      Hybrid Pet
```

Inheritance Types in Hybrid:

Level	Type	Classes
1	Single	Animal → $Mammal$
2	Multiple	Mammal → Dog, Cat
3	Multiple	Dog, Cat → Pet

Complete Program:

```
\# Hybrid Inheritance Demonstration

print("=== Hybrid Inheritance Demo ===")

\# Base class (Level 1)
class Animal:
    def \_\_init\_\_(self, name):
        self.name = name

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

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

\# Single inheritance (Level 2)
class Mammal(Animal):
    def breathe(self):
        print(f"\{self.name\} breathes air")

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

\# Multiple inheritance branches (Level 3)
class Dog(Mammal):
    def bark(self):
        print(f"\{self.name\} barks: Woof!")

    def loyalty(self):
        print(f"\{self.name\} is loyal to owner")

class Cat(Mammal):
    def meow(self):
        print(f"\{self.name\} meows: Meow!")

    def independence(self):
        print(f"\{self.name\} is independent")

\# Hybrid class {- Multiple inheritance (Level 4)}
class HybridPet(Dog, Cat):
    def \_\_init\_\_(self, name, breed):
        super().\_\_init\_\_(name)
        self.breed = breed

    def play(self):
        print(f"\{self.name\} loves to play")

    def show\_info(self):
        print(f"Name: \{self.name\}, Breed: \{self.breed\}")

\# Creating and using hybrid inheritance
print("{n}{-{-}{-} Creating Hybrid Pet {-}{-}{-}"})
pet = HybridPet("Buddy", "Labrador{-Persian Mix"})

print("{n}{-{-}{-} Methods from Animal (Great{-}grandparent) {-}{-}{-}"})
pet.eat()
pet.sleep()

print("{n}{-{-}{-} Methods from Mammal (Grandparent) {-}{-}{-}"})
pet.breathe()
pet.give\_birth()

print("{n}{-{-}{-} Methods from Dog (Parent 1) {-}{-}{-}"})
pet.bark()
pet.loyalty()

print("{n}{-{-}{-} Methods from Cat (Parent 2) {-}{-}{-}"})
pet.meow()
pet.independence()

print("{n}{-{-}{-} Own Methods {-}{-}{-}"})
pet.play()
pet.show\_info()

print("{n}{-{-}{-} Inheritance Chain {-}{-}{-}"})
print(f"MRO (Method Resolution Order): \{HybridPet.\_\_mro\_\_\}")

\# Checking inheritance relationships
print(f"{n}Is HybridPet subclass of Animal? \{issubclass(HybridPet, Animal)\}")
print(f"Is HybridPet subclass of Dog? \{issubclass(HybridPet, Dog)\}")
print(f"Is HybridPet subclass of Cat? \{issubclass(HybridPet, Cat)\}")
```

Expected Output:

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

--- Creating Hybrid Pet ---

--- Methods from Animal (Great-grandparent) ---
Buddy can eat
Buddy can sleep

--- Methods from Mammal (Grandparent) ---
Buddy breathes air
Buddy gives birth to babies

--- Methods from Dog (Parent 1) ---
Buddy barks: Woof!
Buddy is loyal to owner

--- Methods from Cat (Parent 2) ---
Buddy meows: Meow!
Buddy is independent

--- Own Methods ---
Buddy loves to play
Name: Buddy, Breed: Labrador-Persian Mix

--- Inheritance Chain ---
MRO (Method Resolution Order): (<class '__main__.HybridPet'>, <class '__main__.Dog'>, <class '__main__.

Is HybridPet subclass of Animal? True
Is HybridPet subclass of Dog? True
Is HybridPet subclass of Cat? True
```

Key Features of Hybrid Inheritance:

- **Complex Structure**: Combines different inheritance types
- **Method Resolution Order**: Python follows specific order for method lookup
- **Diamond Problem**: Handled automatically by Python's MRO
- **Flexibility**: Access to methods from multiple parent classes

Advantages:

- **Rich Functionality**: Inherits from multiple sources

- **Code Reuse**: Maximum utilization of existing code
- **Relationship Modeling**: Represents complex real-world relationships

Challenges:

- **Complexity**: Harder to understand and maintain
- **Name Conflicts**: Multiple parents may have same method names
- **Memory Usage**: Objects carry more overhead

Mnemonic

"Hybrid Combines All Types"