

LAB ASSIGNMENT 6.3

Subject : AI Assistant Coding

Hall Ticket No : 2303A52465

Name : M.Avinash

Batch : 50

Date : 4/2/26

Task-1 :

Prompt : Generate a simple python code for student information management module with details like name, roll no and branch with user input with brief analysis of the generated code.

```
# Roll-No : 2303A52465
# Generate a simple python code for student information management module with details like name,roll no and branch with user input with brief analysis of the

class Student:
    def __init__(self, name, roll_no, branch):
        self.name = name
        self.roll_no = roll_no
        self.branch = branch

    def display_info(self):
        print(f"Name: {self.name}, Roll No: {self.roll_no}, Branch: {self.branch}")

def main():
    students = []
    n = int(input("Enter number of students: "))
    for i in range(n):
        name = input(f"Enter name of student {i+1}: ")
        roll_no = input(f"Enter roll no of student {i+1}: ")
        branch = input(f"Enter branch of student {i+1}: ")
        students.append(Student(name, roll_no, branch))

    print("\nStudent Information:")
    for student in students:
        student.display_info()

if __name__ == "__main__":
    main()

analysis = """
1. The code defines a 'Student' class with an initializer to set the name, roll number, and branch of a student.
2. The 'display_info' method in the 'Student' class prints the student's details in a formatted string.
3. The 'main' function handles user input to create multiple 'Student' objects based on the number of students specified by the user.
4. It stores the created 'Student' objects in a list and then iterates through the list to display each student's information.
5. The use of a class encapsulates student-related data and behavior, promoting code organization and reusability.
6. The program is interactive, allowing users to input data dynamically, making it flexible for different numbers of students.
"""

print(analysis)
```

Output :

```
Student Information:
Name: Avinash, Roll No: 2303A52465, Branch: CSE
Name: Charan, Roll No: 2303A52367, Branch: ECE
Name: Vineeth, Roll No: 2303A52447, Branch: AIML
```

1. The code defines a 'Student' class with an initializer to set the name, roll number, and branch of a student.
2. The 'display_info' method in the 'Student' class prints the student's details in a formatted string.
3. The 'main' function handles user input to create multiple 'Student' objects based on the number of students specified by the user.
4. It stores the created 'Student' objects in a list and then iterates through the list to display each student's information.
5. The use of a class encapsulates student-related data and behavior, promoting code organization and reusability.
6. The program is interactive, allowing users to input data dynamically, making it flexible for different numbers of students.

Justification : This task explains the creation of a simple student information management module in Python. The code is structured to allow easy input and display of student details, demonstrating basic concepts of object-oriented programming such as classes and methods.

Task-2 :

Prompt : Generate a python code to print first 10 multiples of a given number using different loops and give comparison analysis of the generated codes in tabular format.

```
# Roll-No : 2303A52465
# Generate a python code to print first 10 multiples of a given number using different loops and give comparison analysis of the generated codes in tabular for
def print_multiples_for_loop(number):
    print("Using For Loop:")
    for i in range(1, 11):
        print(f"{number * i}", end="\t")
    print()
def print_multiples_while_loop(number):
    print("Using While Loop:")
    i = 1
    while i <= 10:
        print(f"{number * i}", end="\t")
        i += 1
    print()
def main():
    number = int(input("Enter a number: "))
    print_multiples_for_loop(number)
    print_multiples_while_loop(number)
if __name__ == "__main__":
    main()

# Comparison Analysis of Different Looping Approaches:
analysis = """
| Aspect | For Loop | While Loop |
|-----|-----|-----|
| Structure | Fixed number of iterations | Condition-based iteration |
| Readability | More concise for known iterations | Slightly more verbose |
| Control | Easier to control with range() | Requires manual incrementing |
| Use Case | Best for iterating over sequences | Best for unknown iterations |
| Performance | Generally faster for fixed loops | Slightly slower due to condition |
"""
print(analysis)
```

Output :

Using For Loop:
5 10 15 20 25 30 35 40 45 50

Using While Loop:
5 10 15 20 25 30 35 40 45 50

Aspect	For Loop	While Loop
Structure	Fixed number of iterations	Condition-based iteration
Readability	More concise for known iterations	Slightly more verbose
Control	Easier to control with range()	Requires manual incrementing
Use Case	Best for iterating over sequences	Best for unknown iterations
Performance	Generally faster for fixed loops	Slightly slower due to condition

Justification : This task explains the implementation of two different looping constructs in Python to achieve the same goal of printing multiples of a number. The comparison analysis highlights the strengths and weaknesses of each approach, providing insights into when to use each type of loop effectively. This is useful for understanding control flow in programming and making informed decisions about code structure.

Task-3 :

Prompt : Generate a python code for to classify age groups using nested if-elif-else conditional statements and also for alternative conditional structures (simplified conditions and dictionary-based logic) and give a comparison analysis of them in a tabular form detailed.

```
# Roll No : 2303W02405
# Generate a python code for to classify age groups using nested if-elif-else conditional statements and also for alternative conditional structures (simplified conditions and dictionary-based logic) and give a comparison analysis of them in a tabular form detailed.

def classify_age_nested(age):
    if age < 0:
        return "Invalid age"
    elif age <= 12:
        return "Child"
    elif age <= 19:
        return "Teenager"
    elif age <= 59:
        return "Adult"
    else:
        return "Senior"

def classify_age_dict(age):
    age_groups = {
        range(0, 13): "Child",
        range(13, 20): "Teenager",
        range(20, 60): "Adult",
        range(60, 150): "Senior"
    }
    for age_range, group in age_groups.items():
        if age in age_range:
            return group
    return "Invalid age"

def classify_age_simplified(age):
    if age < 0:
        return "Invalid age"
    if age <= 12:
        return "Child"
    if age <= 19:
        return "Teenager"
    if age <= 59:
        return "Adult"
    return "Senior"

def main():
    age = int(input("Enter age: "))
    print("Nested If-Else Classification:", classify_age_nested(age))
    print("Dictionary-Based Classification:", classify_age_dict(age))
    print("Simplified Conditions Classification:", classify_age_simplified(age))

if __name__ == "__main__":
    main()

# Comparison Analysis of Different Conditional Structures:
analysis = """
| Aspect | Nested If-Elif-Else | Dictionary-Based Logic | Simplified Conditions |
|-----|-----|-----|-----|
| Structure | Hierarchical and clear | Uses data structures for mapping | Linear and straightforward |
| Readability | Easy to read for small conditions | Can be less intuitive for beginners | Very clear for simple checks |
| Maintainability | Can become complex with many conditions | Easy to update by modifying the dictionary | Easy to maintain for simple logic |
| Performance | Slightly slower due to multiple checks | Efficient for lookups | Fast for simple conditions |
| Use Case | Best for a small number of conditions | Best for many discrete conditions | Best for simple, linear checks |
"""
```

Output :

```
Nested If-Else Classification: Teenager
Dictionary-Based Classification: Teenager
Simplified Conditions Classification: Teenager

| Aspect | Nested If-Elif-Else | Dictionary-Based Logic | Simplified Conditions |
|-----|-----|-----|-----|
| Structure | Hierarchical and clear | Uses data structures for mapping | Linear and straightforward |
| Readability | Easy to read for small conditions | Can be less intuitive for beginners | Very clear for simple checks |
| Maintainability | Can become complex with many conditions | Easy to update by modifying the dictionary | Easy to maintain for simple logic |
| Performance | Slightly slower due to multiple checks | Efficient for lookups | Fast for simple conditions |
| Use Case | Best for a small number of conditions | Best for many discrete conditions | Best for simple, linear checks |
```

Justification : This task explains the implementation of different programming constructs in Python, including classes for data management, loops for iteration, and conditional statements for decision-making. The comparison analyses provide insights into the strengths and weaknesses of each approach, helping users understand when to use each construct effectively. This is essential for writing efficient, readable, and maintainable code in various programming scenarios.

Task-4 :

Prompt : Generate a python code to calculate sum of first n natural numbers using a for loop, and suggest an alternative method using while loop or a mathematical formula and give a comparison analysis of the generated codes in tabular format.

```
# Roll-No : 2303AS2465
# Generate a python code to calculate sum of first n natural numbers using a for loop, and suggest an alternative method using while loop or a mathematical formula
# and give a comparison analysis of the generated codes in tabular format.
def sum_natural_numbers_for_loop(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
def sum_natural_numbers_while_loop(n):
    total = 0
    i = 1
    while i <= n:
        total += i
        i += 1
    return total
def sum_natural_numbers_formula(n):
    return n * (n + 1) // 2
def main():
    n = int(input("Enter a natural number n: "))
    print("Input Number:", n)
    print("Sum using For Loop:", sum_natural_numbers_for_loop(n))
    print("Sum using While Loop:", sum_natural_numbers_while_loop(n))
    print("Sum using Mathematical Formula:", sum_natural_numbers_formula(n))
if __name__ == "__main__":
    main()
# Comparison Analysis of Different Methods to Calculate Sum of First n Natural Numbers:
analysis = """
| Aspect | For Loop | While Loop | Mathematical Formula |
|-----|-----|-----|-----|
| Structure | Iterative with fixed range | Iterative with condition | Direct calculation |
| Readability | Clear for small n | Slightly more verbose | Very concise and clear |
| Maintainability | Easy to understand and modify | Easy to understand and modify | Minimal maintenance required |
| Performance | O(n) time complexity | O(n) time complexity | O(1) time complexity |
| Use Case | Best for small to moderate n | Best for small to moderate n | Best for large n |
"""
print(analysis)
```

Output :

```
Input Number: 20
Sum using For Loop: 210
Sum using While Loop: 210
Sum using Mathematical Formula: 210

| Aspect | For Loop | While Loop | Mathematical Formula |
|-----|-----|-----|-----|
| Structure | Iterative with fixed range | Iterative with condition | Direct calculation |
| Readability | Clear for small n | Slightly more verbose | Very concise and clear |
| Maintainability | Easy to understand and modify | Easy to understand and modify | Minimal maintenance required |
| Performance | O(n) time complexity | O(n) time complexity | O(1) time complexity |
| Use Case | Best for small to moderate n | Best for small to moderate n | Best for large n |
```

Justification : This task explains about generating Python code snippets for various programming tasks, along with detailed comparative analyses. It demonstrates different programming constructs such as classes, loops, and conditional statements, highlighting their use cases, performance, and maintainability. This approach aids in understanding the strengths and weaknesses of each method, fostering better coding practices.

Task-5 :

Prompt : Generate a python code for a simple banking application with methods such as deposit, withdraw, and check balance, with meaningful comments analyzing the generated code and explain clearly in points or tabular form

```
# Roll No : 2303AS2465
Generate a python code for a simple banking application with methods such as deposit, withdraw, and check balance, with meaningful comments analyzing the generated code and explain clearly in points or tabular form

# Simple Banking Application
class BankAccount:
    def __init__(self, account_holder, initial_balance=0):
        self.account_holder = account_holder
        self.balance = initial_balance
        self.transactions = []

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            self.transactions.append(("Deposited: (amount)"))
            print(f"Deposited (amount). New balance: {self.balance}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount <= 0:
            print("Withdrawal amount must be positive.")
        elif amount > self.balance:
            print("Insufficient funds.")
        else:
            self.balance -= amount
            self.transactions.append(("Withdraw: (amount)"))
            print(f"Withdraw (amount). New balance: {self.balance}")

    def check_balance(self):
        print(f"Account Holder: {self.account_holder}, Balance: {self.balance}")

    def show_transactions(self):
        print("Transaction History:")
        for transaction in self.transactions:
            print(transaction)

def main():
    account = BankAccount("John Doe", 1000)
    account.check_balance()
    account.deposit(500)
    account.withdraw(200)
    account.withdraw(1500) # Should show insufficient funds
    account.show_transactions()

if __name__ == "__main__":
    main()

# Analysis of the Banking Application
analysis = """
| Aspect | Description |
|-----|-----|
| Class Structure | Encapsulates account data and operations in BankAccount class |
| Initialization | Sets account holder and initial balance |
| Deposit Method | Validates positive amount, updates balance, logs transaction |
| Withdraw Method | Checks for positive amount and sufficient funds before withdrawal |
| Check Balance Method | Displays account holder name and current balance |
| Transaction History | Maintains a list of all transactions for record keeping |
| Error Handling | Validates inputs and prevents invalid operations |
| Extensibility | Easy to add features like interest calculation, transfers, etc. |
"""
print(analysis)
```

Output :

```
Account Holder: John Doe, Balance: 1000
Deposited 500. New balance: 1500
Withdraw 200. New balance: 1300
Insufficient funds.
Transaction History:
Deposited: 500
Withdraw: 200

| Aspect | Description |
|-----|-----|
| Class Structure | Encapsulates account data and operations in BankAccount class |
| Initialization | Sets account holder and initial balance |
| Deposit Method | Validates positive amount, updates balance, logs transaction |
| Withdraw Method | Checks for positive amount and sufficient funds before withdrawal |
| Check Balance Method | Displays account holder name and current balance |
| Transaction History | Maintains a list of all transactions for record keeping |
| Error Handling | Validates inputs and prevents invalid operations |
| Extensibility | Easy to add features like interest calculation, transfers, etc. |
```

Justification : This task explains the implementation of a simple banking application using object-oriented programming principles. It demonstrates how to manage account operations such as deposits, withdrawals, and balance checks while maintaining a transaction history. The analysis provides insights into the design choices and functionality of the application.