

Lab 6: AI-Based Code Completion – Classes, Loops, and Conditionals

Week 3 – Wednesday

Name: P.Sathvika

Hall Ticket No: 2303A51497

Task 1: Classes – Student Class

Prompt Used:

"Generate a Python Student class with name, roll number, branch, and a method to display details."

class Student:

```
    def __init__(self, name, roll_no, branch):
```

```
        self.name = name
```

```
        self.roll_no = roll_no
```

```
        self.branch = branch
```

```
    def display_details(self):
```

```
        print("Name:", self.name)
```

```
        print("Roll Number:", self.roll_no)
```

```
        print("Branch:", self.branch)
```

```
s1 = Student("Eshwar", "2303A51495", "CSE")
```

```
s1.display_details()
```

Output:

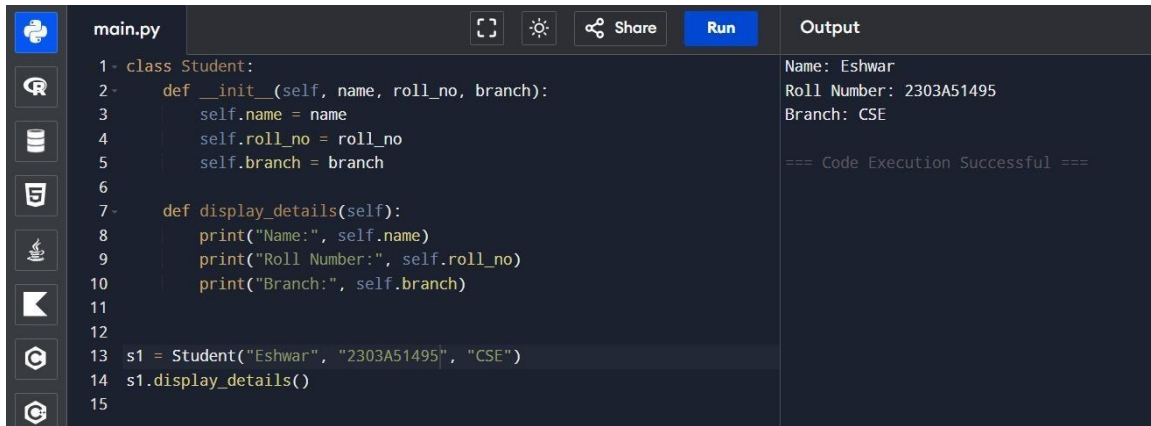
Name: Eshwar

Roll Number: 2303A51495

Branch: CSE

Analysis:

The AI-generated class is well-structured, readable, and correctly uses a constructor and instance method.



The screenshot shows a Python IDE interface. On the left, there is a sidebar with icons for file explorer, search, and other tools. The main area displays a Python file named 'main.py' with the following code:

```
1 class Student:
2     def __init__(self, name, roll_no, branch):
3         self.name = name
4         self.roll_no = roll_no
5         self.branch = branch
6
7     def display_details(self):
8         print("Name:", self.name)
9         print("Roll Number:", self.roll_no)
10        print("Branch:", self.branch)
11
12
13 s1 = Student("Eshwar", "2303A51495", "CSE")
14 s1.display_details()
15
```

On the right, the 'Output' pane shows the results of running the code:

```
Name: Eshwar
Roll Number: 2303A51495
Branch: CSE

=== Code Execution Successful ===
```

Task 2: Loops – Multiples of a Number

Prompt Used:

"Generate Python code to print first 10 multiples of a number using loops."

Using for loop:

```
def multiples_for(n):
    for i in range(1, 11):
        print(n * i)

multiples_for(5)
```

Using while loop:

```
def multiples_while(n):
    i = 1
    while i <= 10:
        print(n * i)
        i += 1

multiples_while(5)
```

Output:

5 10 15 20 25 30 35 40 45 50

```
main.py
1 def multiples_for(n):
2     for i in range(1, 11):
3         print(n * i)
4
5 multiples_for(5)
6
```

Output

5
10
15
20
25
30
35
40
45
50

Analysis:

The for loop is concise and readable, while the while loop provides explicit control over iteration.

Task 3: Conditional Statements – Age Classification

Prompt Used:

"Generate Python code to classify age using if-elif-else."

```
def classify_age(age):
    if age < 13: return
    "Child"
    elif age < 20: return
    "Teenager"
    elif age < 60:
    return "Adult"
else: return "Senior"
print(classify_age(18))
```

Output:

Teenager

```
main.py
1 def classify_age(age):
2     if age < 13:
3         return "Child"
4     elif age < 20:
5         return "Teenager"
6     elif age < 60:
7         return "Adult"
8     else:
9         return "Senior"
10
11 print(classify_age(18))
12 print(classify_age(35))
13 print(classify_age(70))
14
```

Output

Teenager
Adult
Senior

=== Code Execution Successful ===

Explanation:

The conditions are checked in sequence. The first matching condition determines the age group.

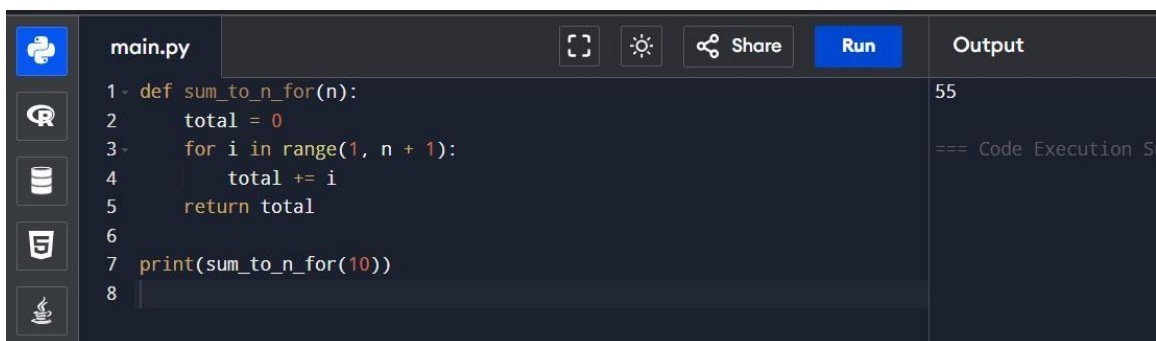
Task 4: For and While Loops – Sum of First n Numbers

Prompt Used:

"Generate Python code to find sum of first n natural numbers using loops."

Using for loop:

```
def sum_to_n_for(n):  
    total = 0  
    for i in range(1, n + 1):  
        total += i  
    return total  
  
print(sum_to_n_for(10))
```



The screenshot shows a code editor with a dark theme. On the left, there is a sidebar with icons for Python, R, SQL, and Jupyter. The main area displays a Python file named 'main.py' with the following code:

```
1 def sum_to_n_for(n):  
2     total = 0  
3     for i in range(1, n + 1):  
4         total += i  
5     return total  
6  
7 print(sum_to_n_for(10))  
8
```

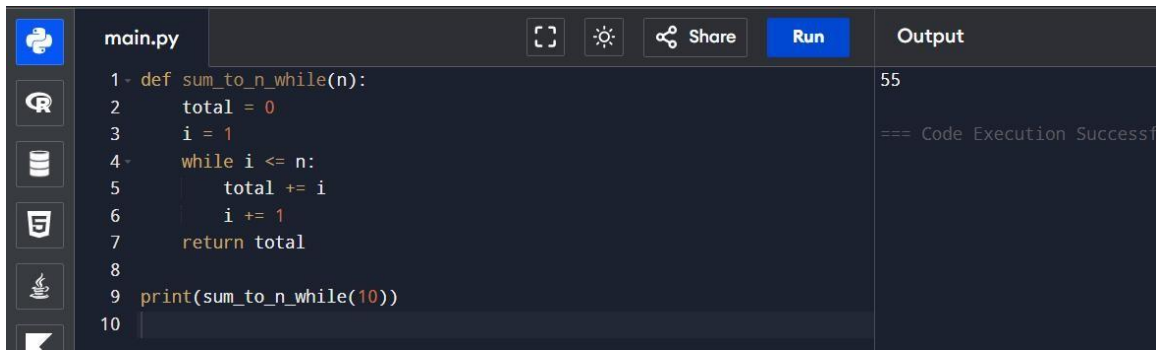
On the right, there is an 'Output' panel showing the result of the code execution: '55'. Above the code editor, there are buttons for 'Run', 'Share', and 'Output'.

Using while loop:

```
def sum_to_n_while(n):  
    total = 0  
    i = 1  
    while i <= n:  
        total += i  
        i += 1  
    return total  
  
print(sum_to_n_while(10))
```

Output:

55



The screenshot shows a Python IDE with a file named 'main.py'. The code defines a function 'sum_to_n_while(n)' that calculates the sum of integers from 1 to n using a while loop. The function is called with 'sum_to_n_while(10)', and the output is '55'. The IDE interface includes a left sidebar with icons for Python, a file explorer, a search bar, and a terminal. The top bar has buttons for 'Run', 'Share', and 'Output'.

```
1 def sum_to_n_while(n):
2     total = 0
3     i = 1
4     while i <= n:
5         total += i
6         i += 1
7     return total
8
9 print(sum_to_n_while(10))
10
```

Output: 55

=== Code Execution Successful ===

Analysis:

Both approaches are correct. Loop-based methods are simple, while mathematical formulas can be more efficient.

Task 5: Classes – Bank Account Class

Prompt Used:

"Generate a Python BankAccount class with deposit, withdraw, and check balance methods."

```
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print("Deposited:", amount)

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print("Withdrawn:", amount)
        else: print("Insufficient
            balance")

    def check_balance(self): print("Current
        Balance:", self.balance)
```

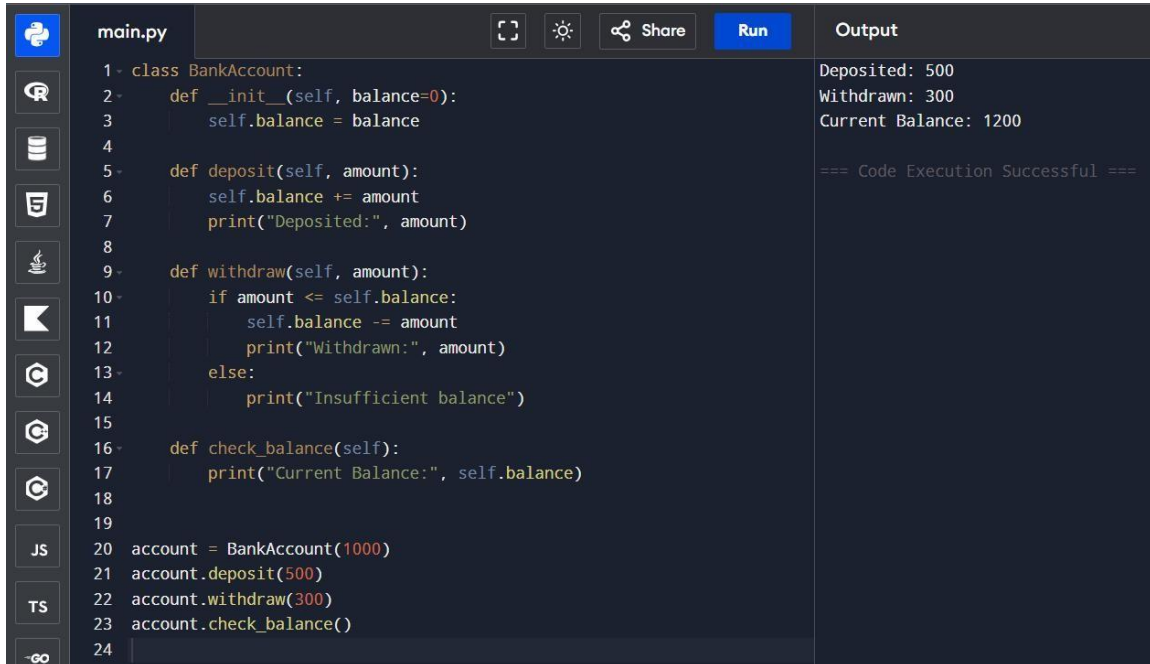
```
account = BankAccount(1000)
account.deposit(500)
account.withdraw(300)
account.check_balance() Output:
Deposited: 500
```

Withdrawn: 300

Current Balance: 1200

Explanation:

The class maintains account balance and updates it through deposit and withdraw methods.



The screenshot shows a code editor with a file named `main.py`. The code defines a `BankAccount` class with methods for deposit, withdraw, and check_balance. The main script creates an account with an initial balance of 1000, deposits 500, withdraws 300, and checks the balance. The output shows the results of these operations: 'Deposited: 500', 'Withdrawn: 300', and 'Current Balance: 1200'. The code execution is successful.

```
1 class BankAccount:
2     def __init__(self, balance=0):
3         self.balance = balance
4
5     def deposit(self, amount):
6         self.balance += amount
7         print("Deposited:", amount)
8
9     def withdraw(self, amount):
10        if amount <= self.balance:
11            self.balance -= amount
12            print("Withdrawn:", amount)
13        else:
14            print("Insufficient balance")
15
16    def check_balance(self):
17        print("Current Balance:", self.balance)
18
19
20 account = BankAccount(1000)
21 account.deposit(500)
22 account.withdraw(300)
23 account.check_balance()
24
```

Output:

```
Deposited: 500
Withdrawn: 300
Current Balance: 1200

=== Code Execution Successful ===
```

Overall Conclusion

This lab demonstrates how AI-assisted code completion helps in generating structured, readable, and correct Python programs. Human review is essential to ensure correctness and efficiency.