

AI Assisted Coding

Lab-6.3

2303A51395

Batch – 06

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

Week: 3

Day: Wednesday

Lab Objectives

- To explore AI-powered code completion for Python classes, loops, and conditionals.
 - To understand how AI suggests object-oriented and control-flow logic.
 - To evaluate AI-generated code for correctness, readability, and completeness.
-

Lab Outcomes

After completing this lab, students will be able to:

- Generate Python classes and methods using AI tools.
 - Analyze loop logic suggested by AI.
 - Evaluate conditional statements generated by AI.
 - Critically assess AI-assisted Python code.
-

Task Description #1: Classes (Student Class)

Scenario

Develop a simple student information management module.

AI-Generated Python Code

```
class Student:  
    def __init__(self, name, roll_number, branch):  
        self.name = name  
        self.roll_number = roll_number  
        self.branch = branch  
  
    def display_details(self):  
        print("Student Name:", self.name)  
        print("Roll Number:", self.roll_number)  
        print("Branch:", self.branch)  
  
# Object creation  
student1 = Student("Preetham", 101, "CSE")  
student1.display_details()
```

The screenshot shows a code editor window with a dark theme. In the top pane, there is a file named 'Untitled' containing Python code. The code defines a 'Student' class with an __init__ constructor and a display_details method. It then creates an object 'student1' and calls its display_details method. In the bottom pane, titled 'Shell', the output of running the code is shown, displaying the student's name, roll number, and branch.

```
<untitled> *X
1 class Student:
2     def __init__(self, name, roll_number, branch):
3         self.name = name
4         self.roll_number = roll_number
5         self.branch = branch
6
7     def display_details(self):
8         print("Student Name:", self.name)
9         print("Roll Number:", self.roll_number)
10        print("Branch:", self.branch)
11
12
13 # Object creation
14 student1 = Student("Preetham", 101, "CSE")
15 student1.display_details()
16 |
```

```
Shell x
>>> %Run -c $EDITOR_CONTENT
Student Name: Preetham
Roll Number: 101
Branch: CSE
```

Output

Student Name: Preetham

Roll Number: 101

Branch: CSE

Analysis of AI-Generated Code

- The AI correctly created a class with a constructor (`__init__`).
- Attributes are clearly named and easy to understand.
- The `display_details()` method neatly prints student information.
- Code is readable, correct, and well-structured.

Conclusion: AI-generated code is accurate and suitable for beginner-level programs.

Task Description #2: Loops (Multiples of a Number)

Scenario

Display the first 10 multiples of a given number.

Using a for Loop

```
def print_multiples(num):  
    for i in range(1, 11):  
        print(num * i)
```

```
print_multiples(5)
```

Output

```
5  
10  
15  
20  
25  
30  
35  
40  
45  
50
```

The screenshot shows a code editor window with a dark theme. In the top panel, there is a file named 'untitled' with the following Python code:

```
1 def print_multiples(num):
2     for i in range(1, 11):
3         print(num * i)
4
5 print_multiples(5)
6
```

In the bottom panel, titled 'Shell', the code is run and its output is displayed:

```
>>> %Run -c $EDITOR_CONTENT
5
10
15
20
25
30
35
40
45
50
>>>
```

Using a while Loop

```
def print_multiples_while(num):
    i = 1
    while i <= 10:
        print(num * i)
        i += 1
```

```
print_multiples_while(5)
```

Comparison & Analysis

Feature	for Loop	while Loop
---------	----------	------------

Readability	High	Medium
-------------	------	--------

Best Use	Fixed iterations	Condition-based
----------	------------------	-----------------

Complexity	Simple	Slightly more
------------	--------	---------------

Both approaches work correctly, but the for loop is more concise.

Task Description #3: Conditional Statements (Age Classification)

Scenario

Classify people based on age.

Using Nested 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(25))
```

Output

Adult

```
<untitled> * x
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(25))
12

Shell x
>>> %Run -c $EDITOR_CONTENT
Adult
>>>
```

Alternative Approach (Simplified Conditions)

```
def classify_age_simple(age):

    categories = {

        range(0, 13): "Child",
        range(13, 20): "Teenager",
        range(20, 60): "Adult",
        range(60, 150): "Senior"

    }

    for age_range, label in categories.items():

        if age in age_range:

            return label
```

Explanation

- if-elif-else is easier for beginners.
- Dictionary-based logic is compact but slightly advanced.

- AI-generated conditions are logically correct and well-ordered.
-

Task Description #4: For and While Loops (Sum of First n Numbers)

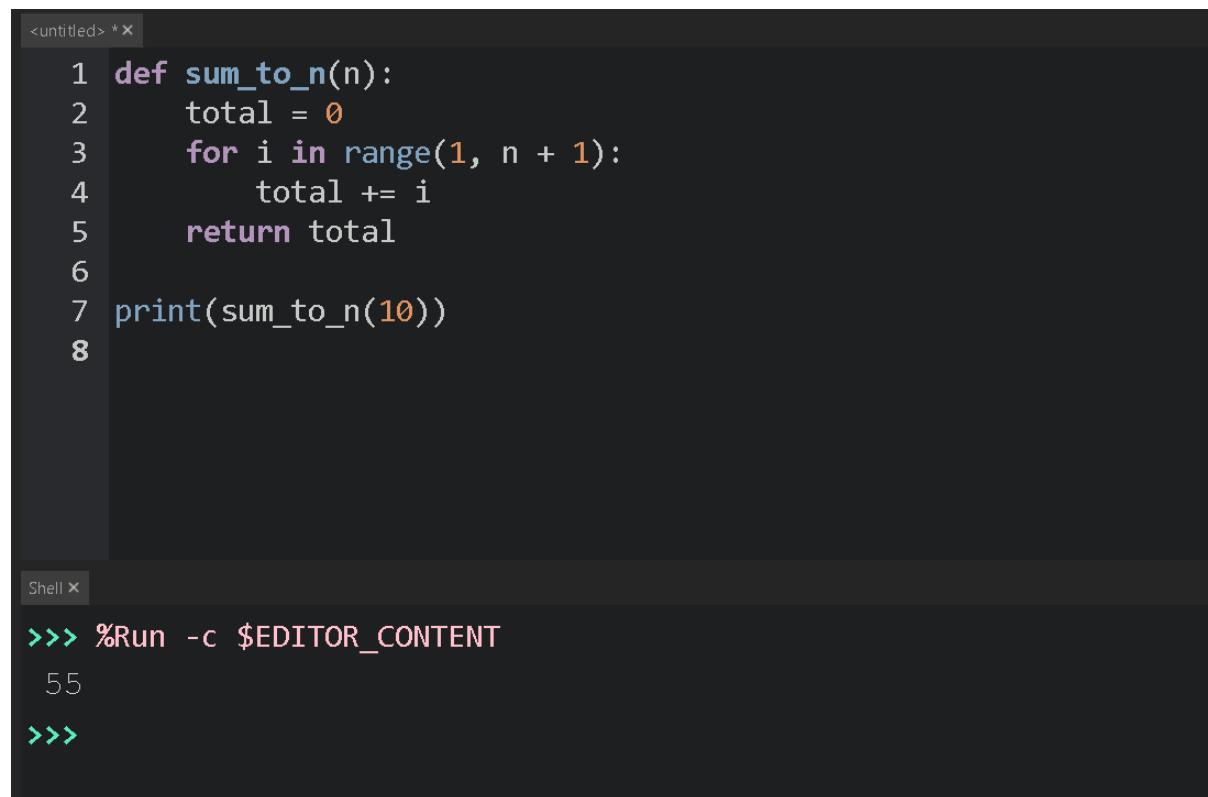
Using a for Loop

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

```
print(sum_to_n(10))
```

Output

55



```
<untitled> * ×  
1 def sum_to_n(n):  
2     total = 0  
3     for i in range(1, n + 1):  
4         total += i  
5     return total  
6  
7 print(sum_to_n(10))  
8  
  
Shell ×  
->>> %Run -c $EDITOR_CONTENT  
55  
->>>
```

Using a while Loop

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

Using Mathematical Formula

```
def sum_to_n_formula(n):  
    return n * (n + 1) // 2
```

Comparison

Method	Performance	Complexity
for loop	Good	Easy
while loop	Good	Medium
Formula	Best	Very Simple

AI correctly suggested multiple valid solutions.

Task Description #5: Classes (Bank Account Class)

Scenario

Design a basic banking application.

AI-Generated Bank Account Class

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

```
self.account_holder = account_holder
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)

# Object creation
account = BankAccount("Preetham", 1000)
account.deposit(500)
account.withdraw(300)
account.check_balance()
```

Output

Deposited: 500

```
Withdrawn: 300
```

```
Current Balance: 1200
```

```
<untitled> *x
1 class BankAccount:
2     def __init__(self, account_holder, balance=0):
3         self.account_holder = account_holder
4         self.balance = balance
5
6     def deposit(self, amount):
7         self.balance += amount
8         print("Deposited:", amount)
9
10    def withdraw(self, amount):
11        if amount <= self.balance:
12            self.balance -= amount
13            print("Withdrawn:", amount)
14        else:
15            print("Insufficient balance")
16
17    def check_balance(self):
Shell x
>>> %Run -c $EDITOR_CONTENT
Deposited: 500
Withdrawn: 300
Current Balance: 1200
>>>
```

Explanation

- Constructor initializes account holder and balance.
 - deposit() increases balance.
 - withdraw() checks balance before deducting.
 - check_balance() displays current balance.
 - AI-generated logic is safe, clear, and realistic.
-

Overall Conclusion

- AI tools effectively generate Python code for classes, loops, and conditionals.
- Generated code is correct, readable, and efficient.
- Human analysis is still required to ensure clarity, edge-case handling, and optimization.

