AI ASSISTED CODING

ASSIGNMENT-6.4

M.KEERTHANA

2403A51259

BATCH-11

CSE-GEN

Task 1:

• Start a Python class named Student with attributes name, roll_number, and marks. Prompt GitHub Copilot to complete methods for displaying details and checking if marks are above average.

```python
class Student:
    def __init__(self, name, roll_number, marks):
        self.name = name
        self.roll_number = roll_number
        self.marks = marks  # marks should be a numeric value

    def display_details(self):
        print(f"Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Marks: {self.marks}")

    def has_passed(self, average_marks=40):
        """
        Checks if the student has passed.
        Default average_marks is 40. You can adjust as needed.
        """
        if self.marks > average_marks:
            print(f"{self.name} has passed.")
            return True
        else:
            print(f"{self.name} has not passed.")
            return False

# Example usage:
if __name__ == "__main__":
    student1 = Student("Alice", "101", 85)
    student1.display_details()
    student1.has_passed()  # Default average is 40
```

This class includes:

- `name`, `roll_number`, and `marks` attributes.
- `display_details()` method to show student data.
- `has_passed()` method to check if the student has marks above average (default set to 40, can be changed). Uses if-else logic.

Observation:

In this task, I explored how AI-assisted code completion can generate and improve Python class structures with methods, loops, and conditionals. By starting with a simple prompt for creating a `Student` class with attributes `name`, `roll_number`, and `marks`, the AI tool (such as GitHub Copilot) was able to suggest the complete implementation. The generated code not only included the constructor method `__init__` to initialize attributes but also methods like `display_details()` to print student information and `is_passed()` to check the pass or fail status using conditional statements.

Task 2:

• Write the first two lines of a for loop to iterate through a list of numbers. Use a comment prompt to let Copilot suggest how to calculate and print the square of even numbers only

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# for loop to go through numbers
for num in numbers:
    # AI: complete logic to print square of even numbers
    if num % 2 == 0:
        print(f"Square of {num} = {num ** 2}")
```

```
Square of 2 = 4
Square of 4 = 16
Square of 6 = 36
Square of 8 = 64
Square of 10 = 100
```

Observation

In this task, I observed how AI-assisted code completion can enhance loop structures with conditional logic. By writing the initial lines of the `for` loop and adding a guiding comment, the AI suggested the correct condition to filter even numbers and compute their squares. This demonstrated the usefulness of AI tools in quickly generating repetitive code while ensuring logic accuracy, reducing manual effort, and supporting error-free implementation.

Task 3:

• Create a class called BankAccount with attributes account_holder and balance. Use Copilot to complete methods for deposit(), withdraw(), and check for insufficient balance

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

    # Method to deposit money
    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"{amount} deposited. New balance: {self.balance}")
        else:
            print("Deposit amount must be positive.")

    # Method to withdraw money with insufficient balance check
    def withdraw(self, amount):
        if amount <= 0:
            print("Withdrawal amount must be positive.")
        elif amount > self.balance:
            print(f"Insufficient balance! Available: {self.balance}")
        else:
            self.balance -= amount
            print(f"{amount} withdrawn. New balance: {self.balance}")

    # Method to display balance
    def display_balance(self):
        print(f"Account Holder: {self.account_holder}, Balance: {self.balance}")


# Example usage
acc1 = BankAccount("Rahul", 5000)
acc1.display_balance()

acc1.deposit(2000)
acc1.withdraw(1000)
acc1.withdraw(7000)   # Trying to withdraw more than balance
```

```
Account Holder: Rahul, Balance: 5000
2000 deposited. New balance: 7000
1000 withdrawn. New balance: 6000
Insufficient balance! Available: 6000
```

Observation

In this task, I observed how AI-assisted code completion can generate a functional `BankAccount` class with essential methods. The AI provided complete implementations for `deposit()`, `withdraw()`, and balance validation using conditional checks. The code effectively prevented overdrawing by comparing the withdrawal amount with the available balance. This task highlighted the reliability of AI tools in suggesting practical banking logic while also emphasizing the importance of adding error handling and ensuring correctness in real-world applications.

Task 4:

• Define a list of student dictionaries with keys name and score. Ask Copilot to write a while loop to print the names of students who scored more than 75.

```
students = [
    {"name": "Alice", "score": 88},
    {"name": "Bob", "score": 72},
    {"name": "Charlie", "score": 94},
    {"name": "David", "score": 67},
    {"name": "Eve", "score": 81}
]

# AI: Generate a while loop to print names of students who scored more than 75
i = 0
while i < len(students):
    if students[i]["score"] > 75:
        print(students[i]["name"])
    i += 1
```

```
Alice
Charlie
Eve
```

Observation

In this task, I observed how AI-assisted code generation can simplify the creation of a `while` loop for list traversal. By defining a list of student dictionaries and prompting the AI, the loop was automatically completed to check scores and print names of students who scored above 75. This demonstrated the ability of AI tools to handle indexing, conditions, and iterations effectively, while also reducing manual effort in writing repetitive logic.

Task 5:

• Begin writing a class ShoppingCart with an empty items list. Prompt Copilot to generate methods to add_item, remove_item, and use a loop to calculate the total bill using conditional discounts.

```python
class ShoppingCart:
    def __init__(self):
        self.items = []  # List to hold cart items

    # AI: Generate methods to add_item and remove_item
    def add_item(self, item, price):
        self.items.append({'item': item, 'price': price})

    def remove_item(self, item):
        self.items = [i for i in self.items if i['item'] != item]

    # AI: Generate a loop to calculate the total bill with conditional discounts
    def calculate_total(self):
        total = 0
        for i in self.items:
            total += i['price']
        # Example discount logic: 10% off if total > 100
        if total > 100:
            total *= 0.9  # apply 10% discount
        return total

# Example usage:
if __name__ == "__main__":
    cart = ShoppingCart()
    cart.add_item("Book", 25)
    cart.add_item("Headphones", 80)
    cart.add_item("Pen", 5)
    print("Cart total:", cart.calculate_total())
    cart.remove_item("Pen")
    print("Cart total after removing Pen:", cart.calculate_total())
```

```
Cart total: 99.0
Cart total after removing Pen: 94.5
```

Observation:

In this task, I observed how AI-assisted code completion can generate a `ShoppingCart` class with useful methods such as `add_item`, `remove_item`, and total bill calculation with discount logic. By starting with an empty `items` list, the AI suggested loops and conditional statements to manage cart operations effectively. This highlighted how AI can quickly provide structured solutions for real-world scenarios like shopping systems, while also reminding us to validate discount conditions and logic for accuracy.