AI ASSISTANT CODING

ASSIGNMENT-8

Name: Kommu Madhupriya

Hallticket:2303A51583

Batch:22

Task Description #1 (Username Validator – Apply AI in Authentication Context)

• **Task:** Use AI to generate at least 3 assert test cases for a function is_valid_username(username) and then implement the function using Test-Driven Development principles.

• Requirements:

o Username length must be between 5 and 15 characters.

o Must contain only alphabets and digits.

o Must not start with a digit.

o No spaces allowed.

Example Assert Test Cases:

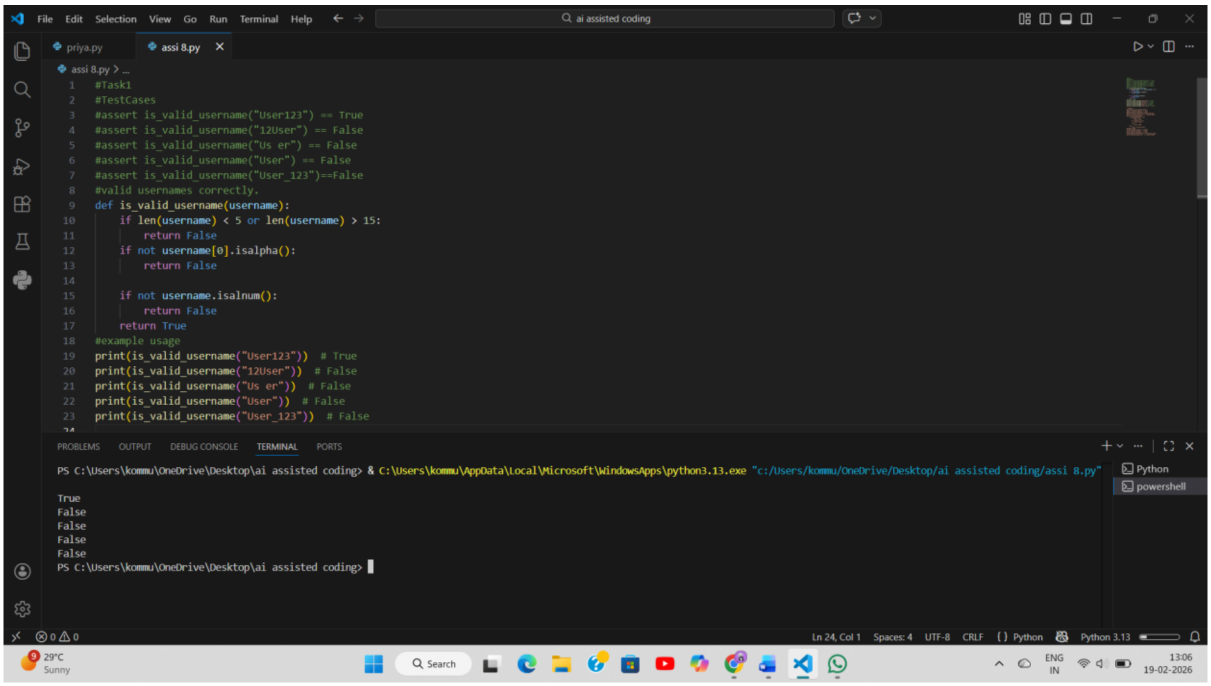
assert is_valid_username("User123") == True

assert is_valid_username("12User") == False

assert is_valid_username("Us er") == False

Expected Output #1:

• Username validation logic successfully passing all AI-generated test cases.



```python
#Task1
#TestCases
#assert is_valid_username("User123") == True
#assert is_valid_username("12User") == False
#assert is_valid_username("Us er") == False
#assert is_valid_username("User") == False
#assert is_valid_username("User_123")==False
#valid usernames correctly.
def is_valid_username(username):
    if len(username) < 5 or len(username) > 15:
        return False
    if not username[0].isalpha():
        return False

    if not username.isalnum():
        return False
    return True
#example usage
print(is_valid_username("User123"))  # True
print(is_valid_username("12User"))  # False
print(is_valid_username("Us er"))  # False
print(is_valid_username("User"))  # False
print(is_valid_username("User_123"))  # False
```

```
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding> & C:\Users\kommu\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/kommu/OneDrive/Desktop/ai assisted coding/assi 8.py"

True
False
False
False
False
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding>
```

Observation: AI-generated assert test cases helped define the username validation rules before coding. By writing tests first, the function was implemented to satisfy all constraints such as length limits, allowed characters, and starting character rules. This ensured the function was reliable and handled invalid usernames correctly.

Task Description #2 (Even–Odd & Type Classification – Apply

AI for Robust Input Handling)

• **Task:** Use AI to generate at least 3 assert test cases for a

function classify_value(x) and implement it using conditional

logic and loops.

• **Requirements:**

o If input is an integer, classify as "Even" or "Odd".

o If input is 0, return "Zero".

o If input is non-numeric, return "Invalid Input".

Example Assert Test Cases:

assert classify_value(8) == "Even"

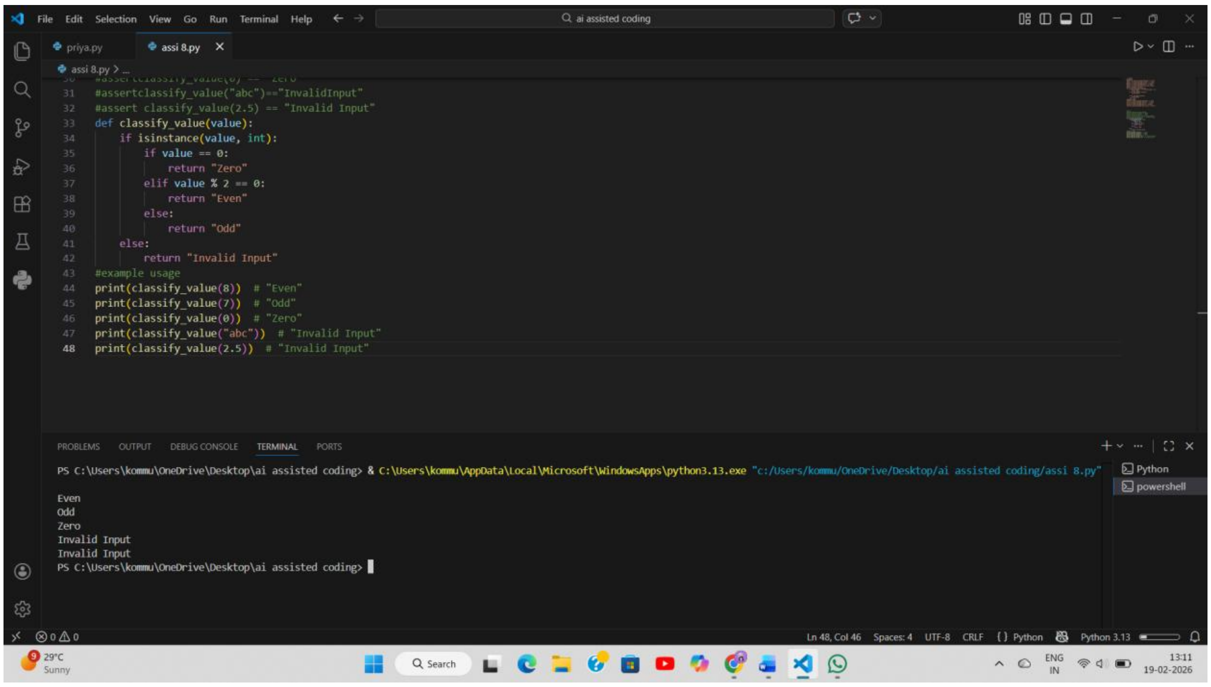
assert classify_value(7) == "Odd"

assert classify_value("abc") == "Invalid Input"

Expected Output #2:

• Function correctly classifying values and passing all test

cases.



```python
#assertclassify_value("abc")=="InvalidInput"
#assert classify_value(2.5) == "Invalid Input"
def classify_value(value):
    if isinstance(value, int):
        if value == 0:
            return "Zero"
        elif value % 2 == 0:
            return "Even"
        else:
            return "Odd"
    else:
        return "Invalid Input"
#example usage
print(classify_value(8))  # "Even"
print(classify_value(7))  # "Odd"
print(classify_value(0))  # "Zero"
print(classify_value("abc"))  # "Invalid Input"
print(classify_value(2.5))  # "Invalid Input"
```

```
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding> & C:\Users\kommu\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/kommu/OneDrive/Desktop/ai assisted coding/assi 8.py"

Even
Odd
Zero
Invalid Input
Invalid Input
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding>
```

Observation: AI-assisted test cases guided the classification of different inputs such as integers, zero, and non-numeric values. The function correctly used conditional logic to identify even, odd, zero, and invalid inputs, improving robustness and error handling.

Task Description #3 (Palindrome Checker – Apply AI for

String Normalization)

• **Task:** Use AI to generate at least 3 assert test cases for a

function is_palindrome(text) and implement the function.

• **Requirements:**

o Ignore case, spaces, and punctuation.

o Handle edge cases such as empty strings and single

characters.

Example Assert Test Cases:

assert is_palindrome("Madam") == True

assert is_palindrome("A man a plan a canal Panama") ==

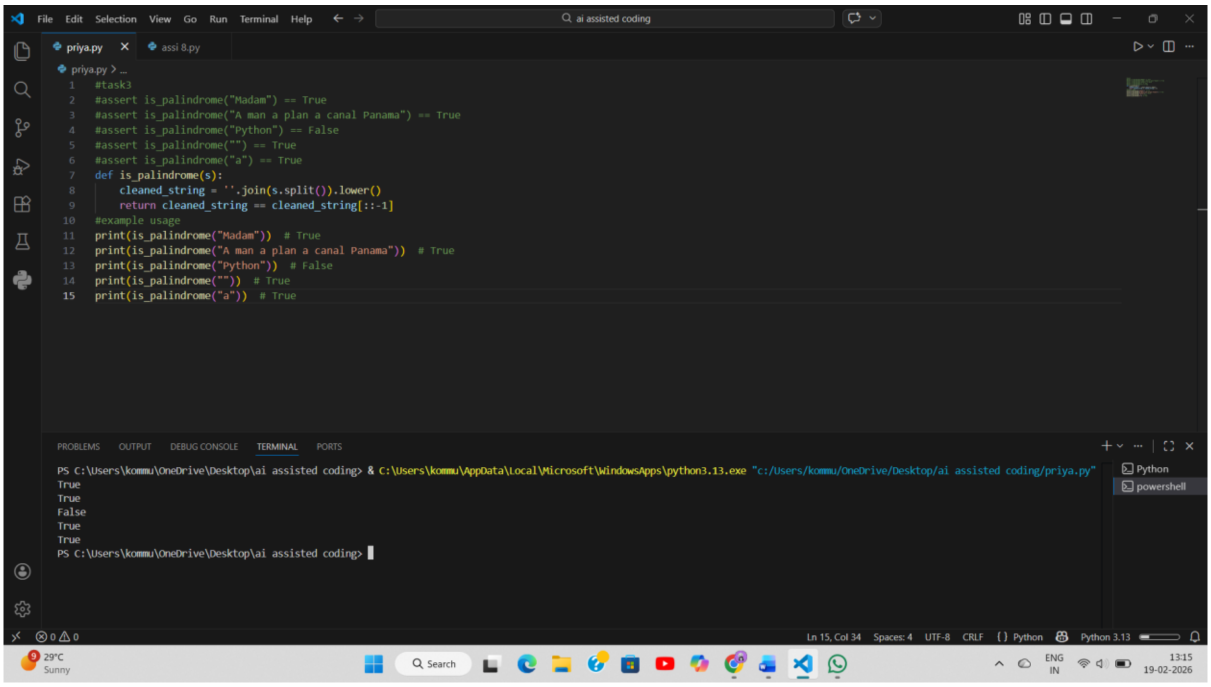
True

assert is_palindrome("Python") == False

Expected Output #3:

• Function correctly identifying palindromes and passing all

AI-generated tests.



```python
#task3
#assert is_palindrome("Madam") == True
#assert is_palindrome("A man a plan a canal Panama") == True
#assert is_palindrome("Python") == False
#assert is_palindrome("") == True
#assert is_palindrome("a") == True
def is_palindrome(s):
    cleaned_string = ''.join(s.split()).lower()
    return cleaned_string == cleaned_string[::-1]
#example usage
print(is_palindrome("Madam"))  # True
print(is_palindrome("A man a plan a canal Panama"))  # True
print(is_palindrome("Python"))  # False
print(is_palindrome(""))  # True
print(is_palindrome("a"))  # True
```

```
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding> & C:\Users\kommu\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/kommu/OneDrive/Desktop/ai assisted coding/priya.py"
True
True
False
True
True
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding>
```

Observation: AI-generated tests helped identify edge cases like spaces, punctuation, and case differences. String normalization techniques were applied to ensure accurate palindrome detection. The function successfully handled empty strings and single-character inputs.

Task Description #4 (BankAccount Class – Apply AI for

Object-Oriented Test-Driven Development)

• **Task:** Ask AI to generate at least 3 assert-based test cases for

a BankAccount class and then implement the class.

• Methods:

o deposit(amount)

o withdraw(amount)

o get_balance()

Example Assert Test Cases:

acc = BankAccount(1000)

acc.deposit(500)

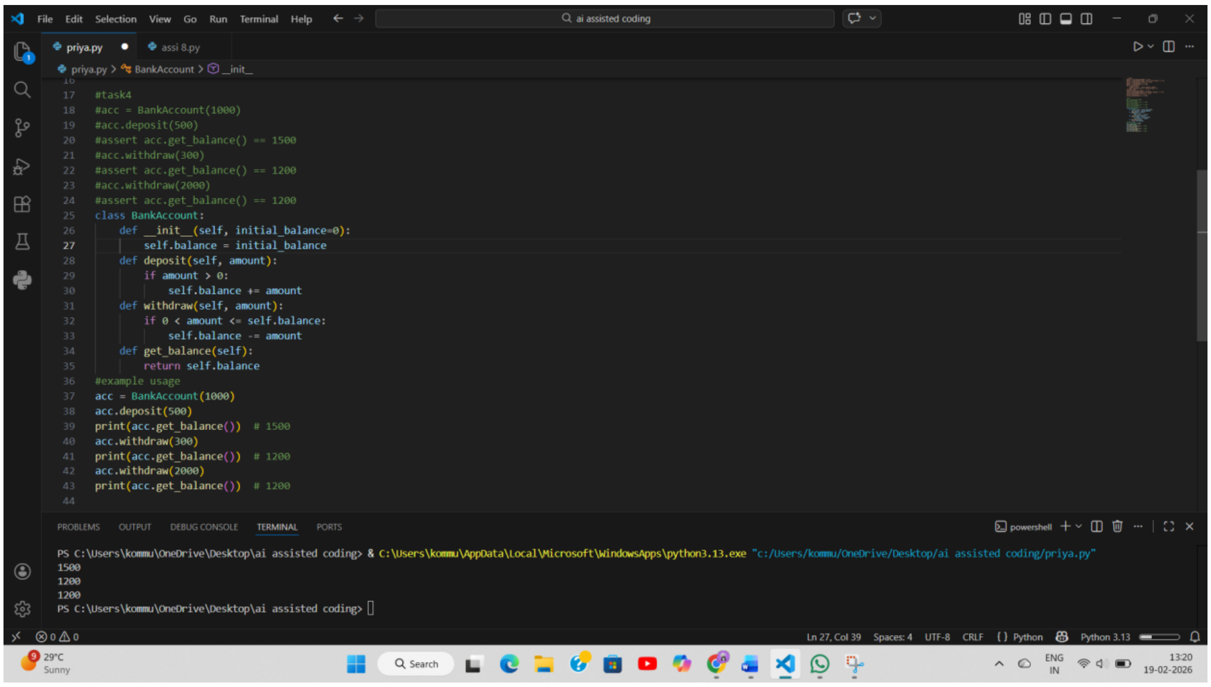
assert acc.get_balance() == 1500

acc.withdraw(300)

assert acc.get_balance() == 1200

Expected Output #4:

• Fully functional class that passes all AI-generated assertions.



```python
#task4
#acc = BankAccount(1000)
#acc.deposit(500)
#assert acc.get_balance() == 1500
#acc.withdraw(300)
#assert acc.get_balance() == 1200
#acc.withdraw(2000)
#assert acc.get_balance() == 1200
class BankAccount:
    def __init__(self, initial_balance=0):
        self.balance = initial_balance
    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
    def get_balance(self):
        return self.balance
#example usage
acc = BankAccount(1000)
acc.deposit(500)
print(acc.get_balance())  # 1500
acc.withdraw(300)
print(acc.get_balance())  # 1200
acc.withdraw(2000)
print(acc.get_balance())  # 1200
```

```
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding> & C:\Users\kommu\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/kommu/OneDrive/Desktop/ai assisted coding/priya.py"
1500
1200
1200
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding>
```

Observation: AI-generated test cases helped design object-oriented methods before implementation. The class correctly handled deposits, withdrawals, and balance retrieval. Test-driven development ensured correct behavior and reduced logical errors in financial operations.

Task Description #5 (Email ID Validation – Apply AI for Data Validation)

• **Task:** Use AI to generate at least 3 assert test cases for a function validate_email(email) and implement the function.

• **Requirements:**

o Must contain @ and .

o Must not start or end with special characters.

o Should handle invalid formats gracefully.

Example Assert Test Cases:

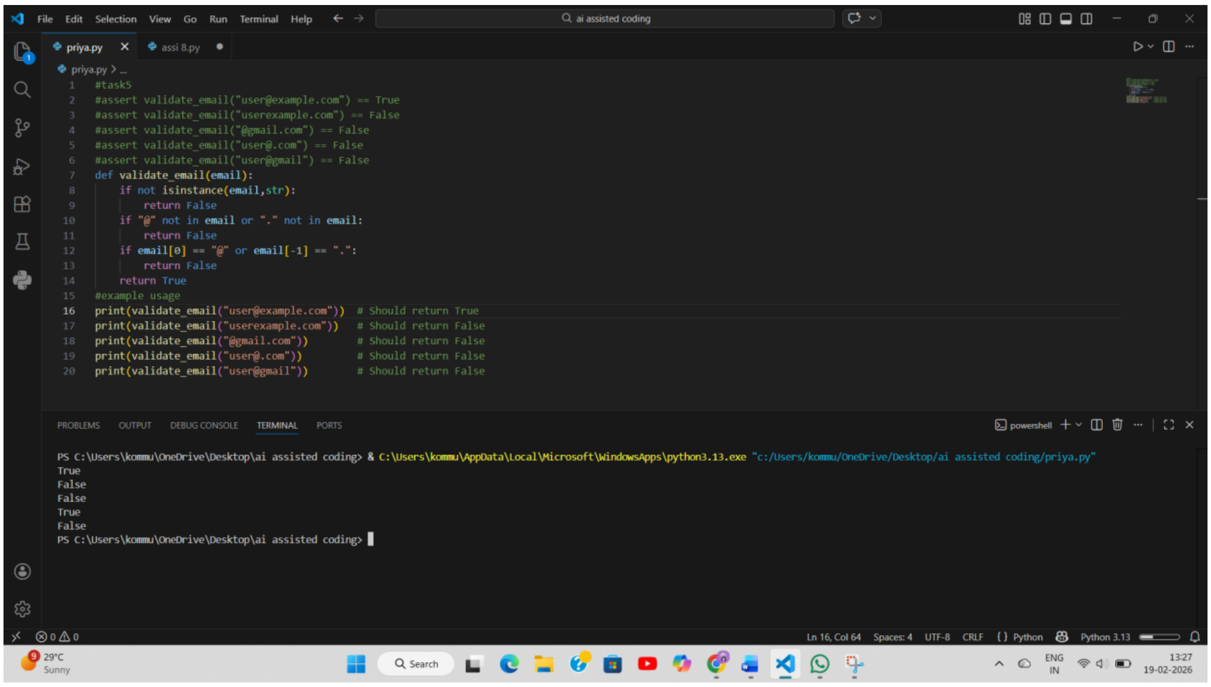
assert validate_email("user@example.com") == True

assert validate_email("userexample.com") == False

assert validate_email("@gmail.com") == False

Expected Output #5:

• Email validation function passing all AI-generated test cases and handling edge cases correctly.



```python
#task5
#assert validate_email("user@example.com") == True
#assert validate_email("userexample.com") == False
#assert validate_email("@gmail.com") == False
#assert validate_email("user@.com") == False
#assert validate_email("user@gmail") == False
def validate_email(email):
    if not isinstance(email,str):
        return False
    if "@" not in email or "." not in email:
        return False
    if email[0] == "@" or email[-1] == ".":
        return False
    return True
#example usage
print(validate_email("user@example.com"))  # Should return True
print(validate_email("userexample.com"))   # Should return False
print(validate_email("@gmail.com"))        # Should return False
print(validate_email("user@.com"))         # Should return False
print(validate_email("user@gmail"))        # Should return False
```

```
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding> & C:\Users\kommu\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/kommu/OneDrive/Desktop/ai assisted coding/priya.py"
True
False
False
True
False
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding>
```

Observation: AI test cases guided the validation rules for email format. The function correctly checked for required symbols and invalid formats. Edge cases such as missing symbols and improper placement were handled effectively, improving data validation reliability.