AI ASSISTANT CODING ASSIGNMENT -8.5

NAME : MOGILI KEERTHI

ROLL NO : 2303A51427

BATCH NO : 21

Lab 8: Test-Driven Development with AI – Generating and Working with Test Cases

Lab Objectives:

• To introduce students to test-driven development (TDD) using AI code generation tools.

• To enable the generation of test cases before writing code implementations.

• To reinforce the importance of testing, validation, and error handling.

• To encourage writing clean and reliable code based on AI- generated test expectations.

Lab Outcomes (LOs):

After completing this lab, students will be able to:

• Use AI tools to write test cases for Python functions and classes.

• Implement functions based on test cases in a test-first development style.

• Analyze the completeness and coverage of AI-generated tests.

• Compare AI-generated and manually written test cases for quality and logic

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

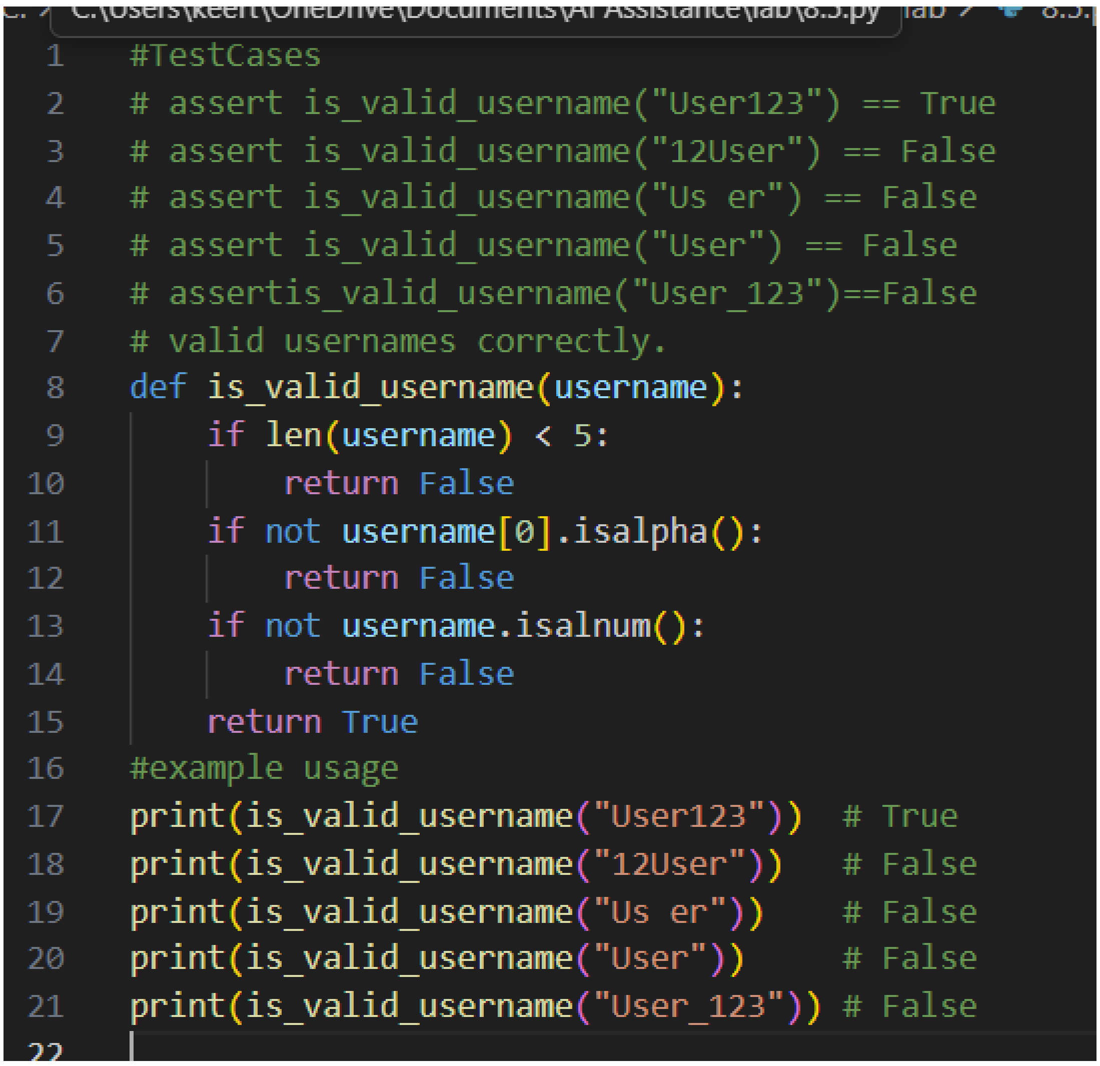
assert is_valid_username("12User") == False

assert is_valid_username("Us er") == False

Expected Output #1:

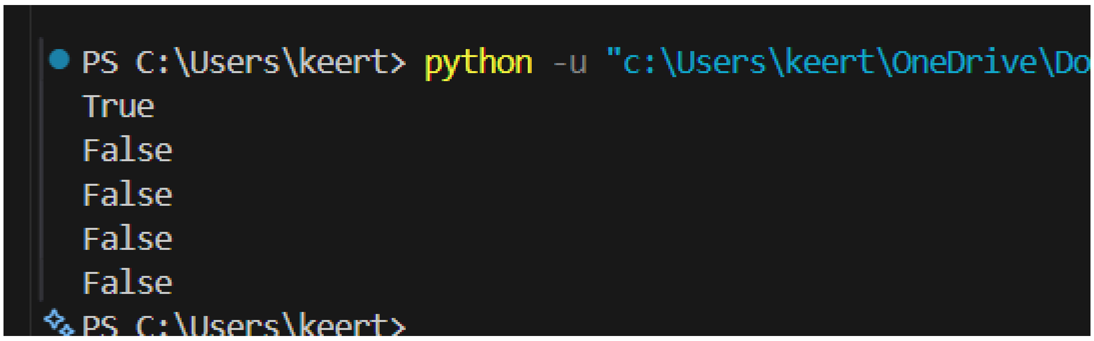
• Username validation logic successfully passing all AI-generated test cases.

Code:



```python
#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
# assertis_valid_username("User_123")==False
# valid usernames correctly.
def is_valid_username(username):
    if len(username) < 5:
        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
```

Output:



```
PS C:\Users\keert> python -u "c:\Users\keert\OneDrive\Do
True
False
False
False
False
PS C:\Users\keert>
```

Explanation:
The AI-generated assert test cases were useful in clearly defining the username validation rules before implementing the actual code. By designing the test cases first, the function was developed to meet all specified constraints, including length restrictions, permitted characters, and rules for the starting character. This test-first approach improved reliability and ensured that invalid usernames were properly identified and handled.

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:

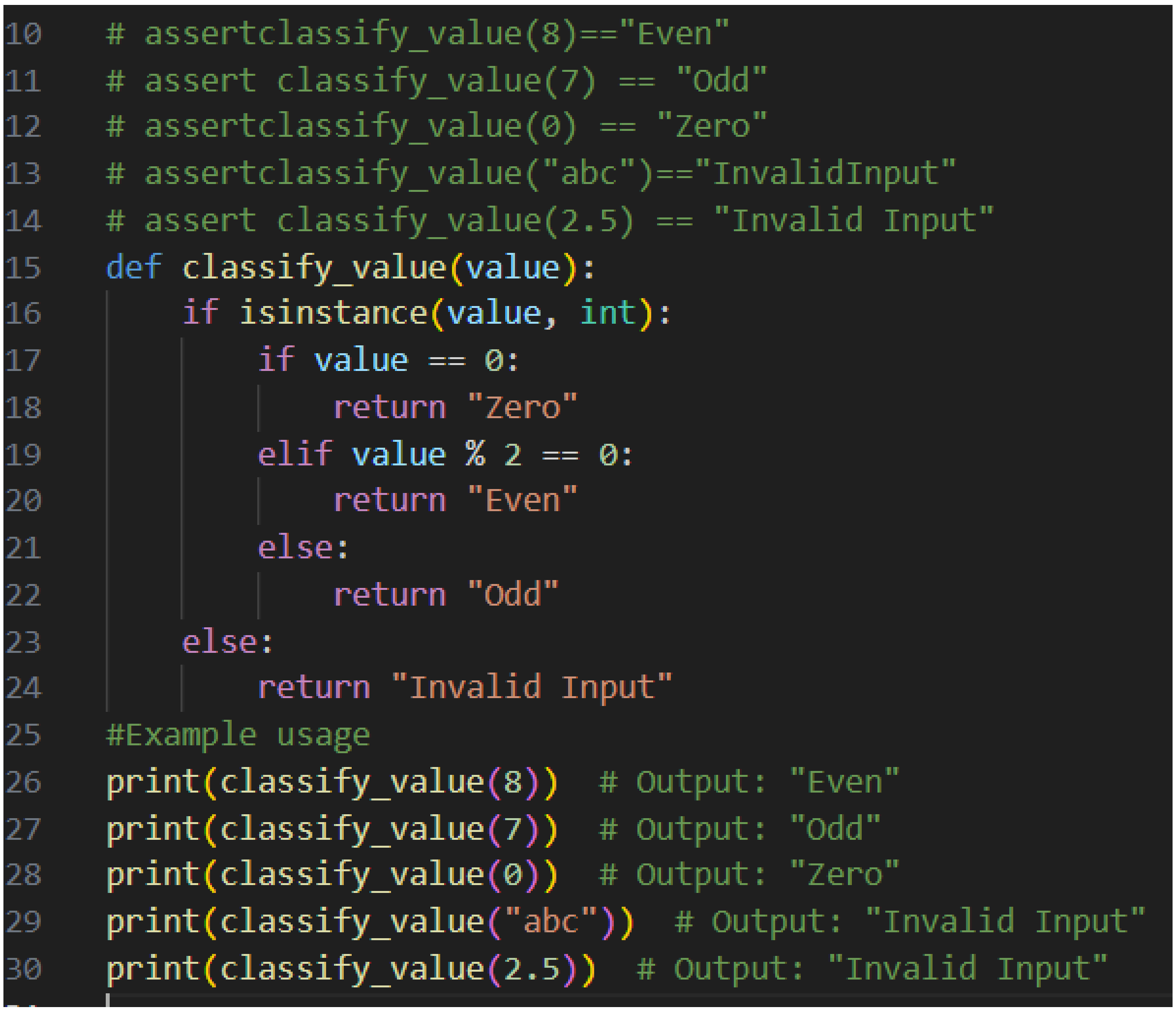
assert classify_value(8) == "Even"

assert classify_value(7) == "Odd"

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

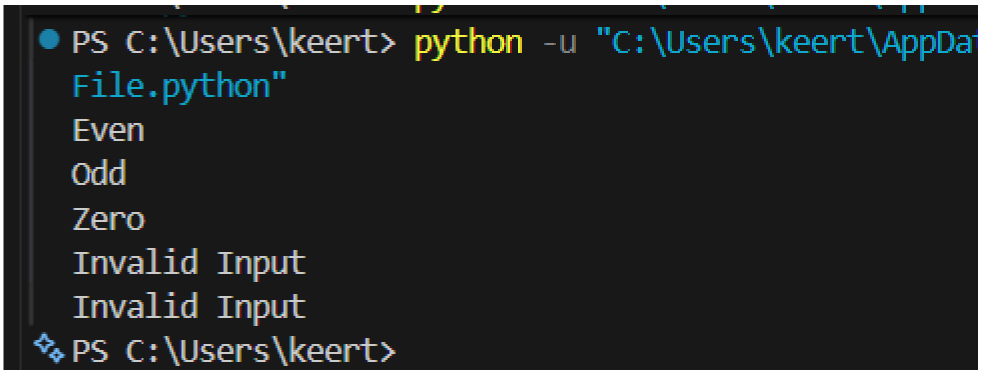
Expected Output #2:

• Function correctly classifying values and passing all test cases.



```python
# assertclassify_value(8)=="Even"
# assert classify_value(7) == "Odd"
# assertclassify_value(0) == "Zero"
# 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))  # Output: "Even"
print(classify_value(7))  # Output: "Odd"
print(classify_value(0))  # Output: "Zero"
print(classify_value("abc"))  # Output: "Invalid Input"
print(classify_value(2.5))  # Output: "Invalid Input"
```

Output:



```
PS C:\Users\keert> python -u "C:\Users\keert\AppDa
File.python"
Even
Odd
Zero
Invalid Input
Invalid Input
PS C:\Users\keert>
```

Explanation:

The AI-generated assert test cases played an important role in clearly establishing the username validation requirements prior to implementation. By creating and reviewing the test cases first, the function was developed to comply with all necessary constraints, including minimum and maximum length, permitted characters, and valid starting character rules. This structured approach enhanced the accuracy, robustness, and reliability of the function, ensuring that invalid usernames were effectively detected and properly handled.

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

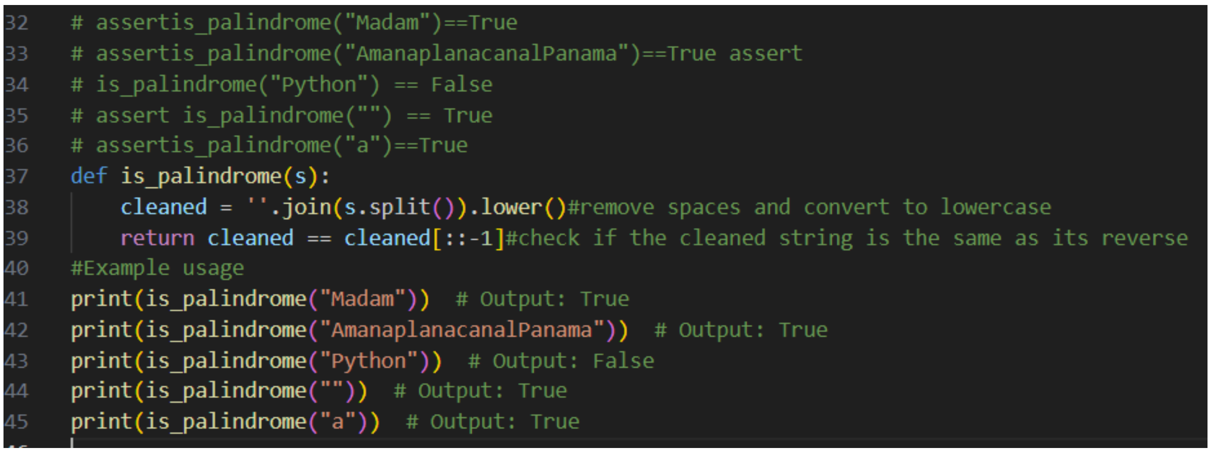
assert is_palindrome("A man a plan a canal Panama") ==True

assert is_palindrome("Python") == False

Expected Output #3:

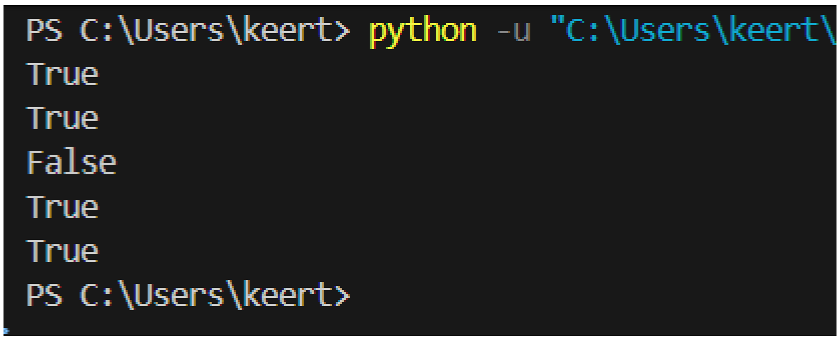
• Function correctly identifying palindromes and passing all AI-generated tests.

Code:



```python
# assertis_palindrome("Madam")==True
# assertis_palindrome("AmanaplanacanalPanama")==True assert
# is_palindrome("Python") == False
# assert is_palindrome("") == True
# assertis_palindrome("a")==True
def is_palindrome(s):
    cleaned = ''.join(s.split()).lower()#remove spaces and convert to lowercase
    return cleaned == cleaned[::-1]#check if the cleaned string is the same as its reverse
#Example usage
print(is_palindrome("Madam"))  # Output: True
print(is_palindrome("AmanaplanacanalPanama"))  # Output: True
print(is_palindrome("Python"))  # Output: False
print(is_palindrome(""))  # Output: True
print(is_palindrome("a"))  # Output: True
```

Output:



```
PS C:\Users\keert> python -u "C:\Users\keert\
True
True
False
True
True
PS C:\Users\keert>
```

Explanation:

The AI-generated test cases were effective in identifying important edge cases such as spaces, punctuation marks, and variations in letter casing. To ensure accurate palindrome detection, appropriate string normalization techniques were applied, including removing non-alphanumeric characters and converting all letters to a uniform case. As a result, the function was able to correctly evaluate palindromes while also handling special cases like empty strings and single-character inputs without errors.

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

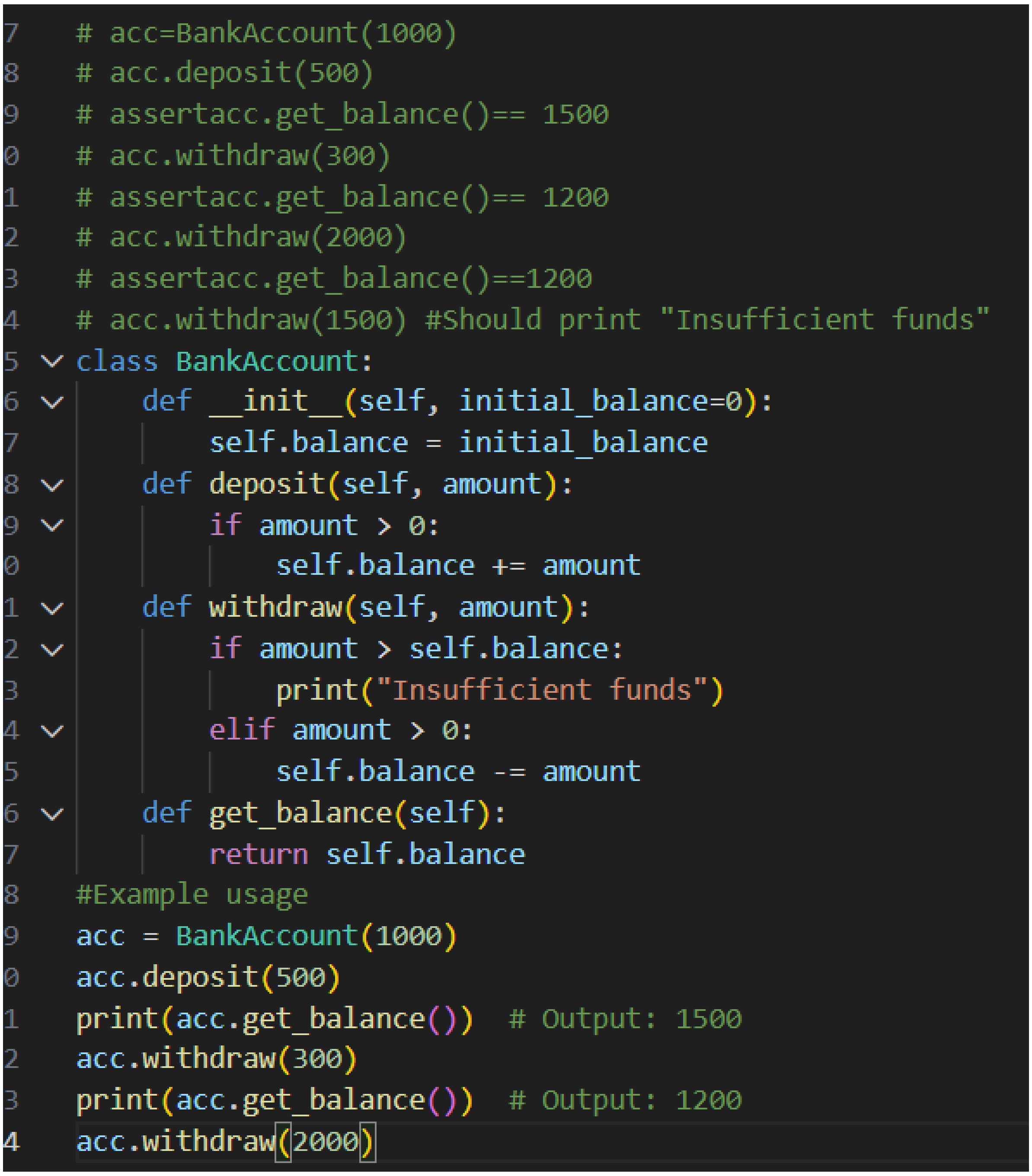
acc.withdraw(300)

assert acc.get_balance() == 1200

Expected Output #4:

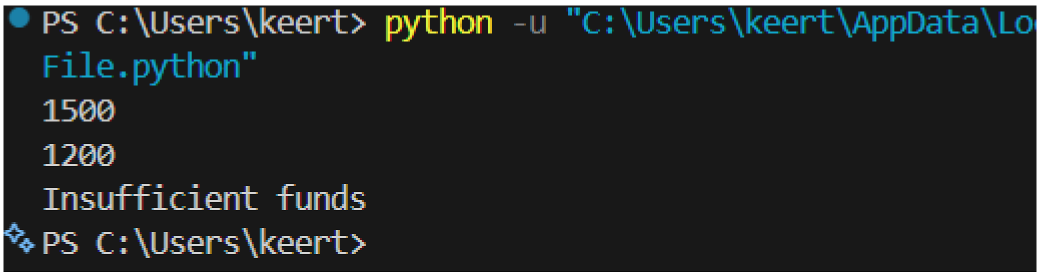
• Fully functional class that passes all AI-generated assertions.

Code:



```python
# acc=BankAccount(1000)
# acc.deposit(500)
# assertacc.get_balance()== 1500
# acc.withdraw(300)
# assertacc.get_balance()== 1200
# acc.withdraw(2000)
# assertacc.get_balance()==1200
# acc.withdraw(1500) #Should print "Insufficient funds"
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 amount > self.balance:
            print("Insufficient funds")
        elif amount > 0:
            self.balance -= amount
    def get_balance(self):
        return self.balance
#Example usage
acc = BankAccount(1000)
acc.deposit(500)
print(acc.get_balance())  # Output: 1500
acc.withdraw(300)
print(acc.get_balance())  # Output: 1200
acc.withdraw(2000)
```

Output:



```
PS C:\Users\keert> python -u "C:\Users\keert\AppData\Lo
File.python"
1500
1200
Insufficient funds
PS C:\Users\keert>
```

Explanation:

The AI-generated test cases assisted in structuring the object-oriented methods prior to actual implementation. By defining the expected behavior in advance, the class was developed to accurately manage core operations such as deposits, withdrawals, and balance inquiries. This test-driven approach ensured correctness in financial transactions, improved code reliability, and minimized potential logical errors in the overall system.

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

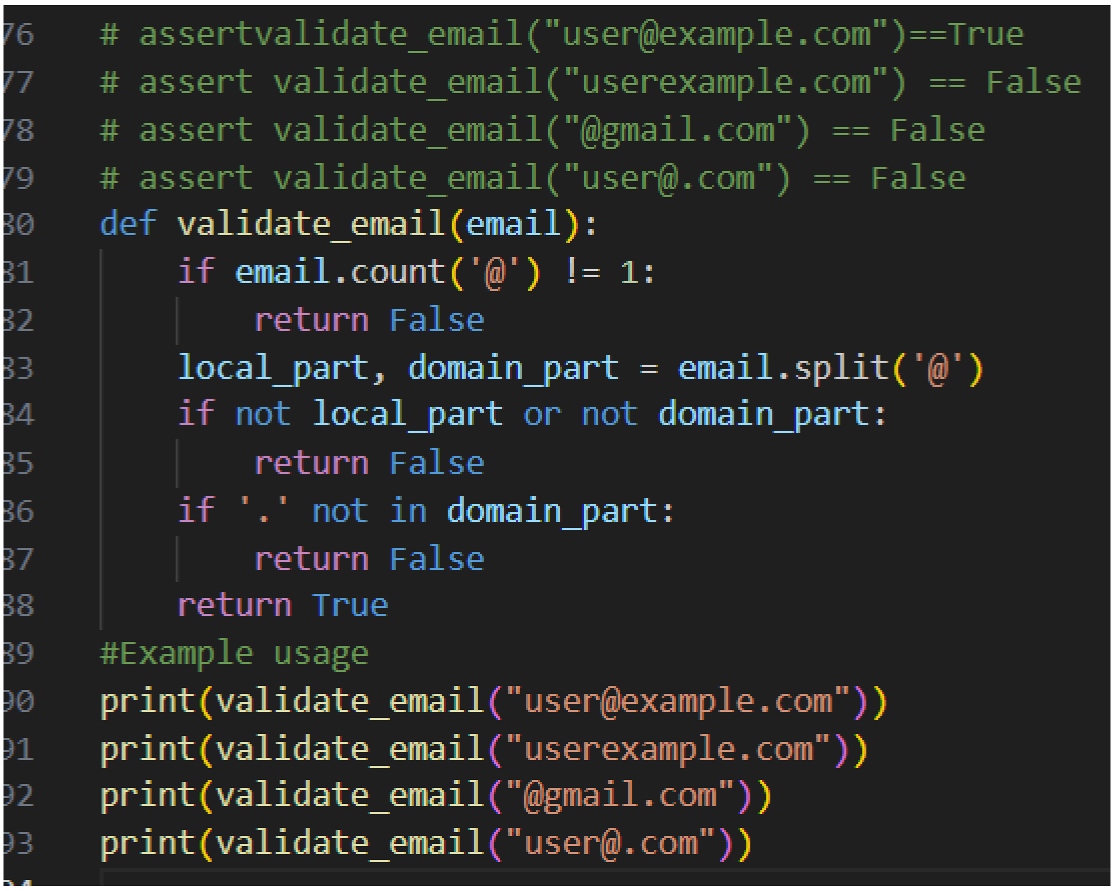
assert validate_email("userexample.com") == False

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

Expected Output #5:

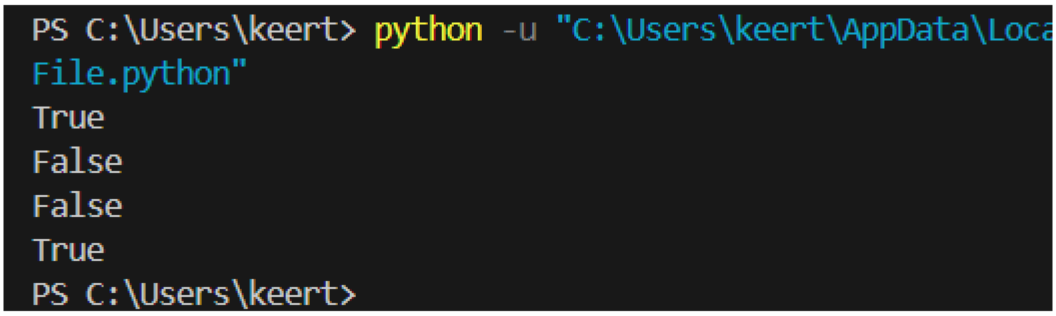
• Email validation function passing all AI-generated test cases and handling edge cases correctly.

Code:



```python
# assertvalidate_email("user@example.com")==True
# assert validate_email("userexample.com") == False
# assert validate_email("@gmail.com") == False
# assert validate_email("user@.com") == False
def validate_email(email):
    if email.count('@') != 1:
        return False
    local_part, domain_part = email.split('@')
    if not local_part or not domain_part:
        return False
    if '.' not in domain_part:
        return False
    return True
#Example usage
print(validate_email("user@example.com"))
print(validate_email("userexample.com"))
print(validate_email("@gmail.com"))
print(validate_email("user@.com"))
```

Output:



```
PS C:\Users\keert> python -u "C:\Users\keert\AppData\Loca
File.python"
True
False
False
True
PS C:\Users\keert>
```

Explanation:

The AI-generated test cases played a key role in defining the validation rules for the email format before implementation. The function was designed to verify the presence of essential symbols and detect invalid formatting patterns. Edge cases, such as missing special characters or incorrect symbol placement, were carefully handled. This systematic testing approach enhanced the accuracy and reliability of the overall email validation process.