Assignment–8.5

Name : K.Tejaswini
Batch : 21
Ht.No : 2303A51425

1. TaskDescription(UsernameValidator–ApplyAIinAuthenticationContext)
Promt:
#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.

Code:

task.py > ...

```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
# assert is_valid_username("User_123")==False
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:

```
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> &
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> &
OWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/WINDOWS/anacc
True
False
False
False
False
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

Observation:

AI-generated assert test cases helped define the user name 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.

2. **Task:Even–Odd&TypeClassification**

promt:

assertclassify_value(8)=="Even"

assert classify_value(7) == "Odd"

assertclassify_value(0) == "Zero"

assertclassify_value("abc")=="InvalidInput"

assert classify_value(2.5) == "Invalid Input"

Input:

```python
# assert classify_value(8)=="Even"
# assert classify_value(7) == "Odd"
# assert classify_value(0) == "Zero"
# assert classify_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
```

Output:

```
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/
coding/task.py"
coding/task.py"
Even
Odd
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/
coding/task.py"
Even
Odd
Even
Odd
Zero
Invalid input
Invalid input
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit 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.

Task3:PalindromeChecker

Promt:

assertis_palindrome("Madam")==True
assertis_palindrome("AmanaplanacanalPanama")==True assert is_palindrome("Python") == False
assert is_palindrome("") == True
assertis_palindrome("a")==True

Code:

```python
#Test cases:
# assert is_palindrome("Madam")==True
# assert is_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()
    return cleaned == cleaned[::-1]
#Example Usage:
print(is_palindrome("Madam"))  # True
print(is_palindrome("AmanaplanacanalPanama"))  # True
print(is_palindrome("Python"))  # False
print(is_palindrome(""))  # True
print(is_palindrome("a"))  # True
```

Output:

```
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/U
coding/task.py"
True
True
False
True
True
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

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

Task4 : Observation:Bank Account Class
Promt:
acc=BankAccount(1000)
acc.deposit(500)
assertacc.get_balance()== 1500

acc.withdraw(300)
assertacc.get_balance()== 1200

acc.withdraw(2000)
assertacc.get_balance()==1200

Code:

task.py > BankAccount > __init__

```python
#Test Cases:
# acc = BankAccount(1000)
# acc.deposit(500)
# assert acc.get_balance() == 1500
# acc.withdraw(300)
# assert acc.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())  # 1500
acc.withdraw(300)
print(acc.get_balance())  # 1200
acc.withdraw(1500)  # Should print "Insufficient funds"

```

Output:

```
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
coding/task.py"
1500
1200
Insufficient funds
```

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.

Task5:EmailIDValidation

Prmot:

assertvalidate_email("user@example.com")==True
assert validate_email("userexample.com") == False
assert validate_email("@gmail.com") == False
assert validate_email("user@.com") == False
assertvalidate_email("user@gmail")==False

Code:

```python
# Test Cases:
# assert validate_email("user@example.com") == True
# assert validate_email("user example.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("user example.com"))
print(validate_email("@gmail.com"))
print(validate_email("user@.com"))

```

Output:

```
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> &
coding/task.py"
True
False
False
True
```

Observation:

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