

## Assignment–8.5

Name : R.sai kundan

Batch : 21

Ht.No : 2303A51413

### 1. TaskDescription(UsernameValidator–ApplyAlinAuthenticationContext)

Prompt:

#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.

Code:

```
task.py > ...
1  ##TestCases
2  # assert is_valid_username("User123") == True
3  # assert is_valid_username("12User") == False
4  # assert is_valid_username("Us er") == False
5  # assert is_valid_username("User") == False
6  # assert is_valid_username("User_123")==False
7  def is_valid_username(username):
8      if len(username) < 5:
9          return False
10     if not username[0].isalpha():
11         return False
12     if not username.isalnum():
13         return False
14     return True
15 #Example Usage
16 print(is_valid_username("User123")) # True
17 print(is_valid_username("12User")) # False
18 print(is_valid_username("Us er")) # False
19 print(is_valid_username("User")) # False
20 print(is_valid_username("User_123")) # False
21 |
```

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/anaco
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

prompt:

```
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:

```
1 # assert classify_value(8)=="Even"  
2 # assert classify_value(7) == "Odd"  
3 # assert classify_value(0) == "Zero"  
4 # assert classify_value("abc")=="InvalidInput"  
5 # assert classify_value(2.5) == "Invalid input"  
6 def classify_value(value):  
7     if isinstance(value, int):  
8         if value == 0:  
9             return "Zero"  
10        elif value % 2 == 0:  
11            return "Even"  
12        else:  
13            return "Odd"  
14    else:  
15        return "Invalid input"  
16  
17 #Example Usage:  
18 print(classify_value(8)) # Even  
19 print(classify_value(7)) # Odd  
20 print(classify_value(0)) # Zero  
21 print(classify_value("abc")) # Invalid input  
22 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"
o 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

#### Prompt:

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

## Code:

```
task.py > ...
1  #Test cases:
2  # assert is_palindrome("Madam")==True
3  # assert is_palindrome("AmanaplanacanalPanama")==True
4  # assert is_palindrome("Python") == False
5  # assert is_palindrome("") == True
6  # assert is_palindrome("a")==True
7  def is_palindrome(s):
8      cleaned = ''.join(s.split()).lower()
9      return cleaned == cleaned[::-1]
10 #Example Usage:
11 print(is_palindrome("Madam")) # True
12 print(is_palindrome("AmanaplanacanalPanama")) # True
13 print(is_palindrome("Python")) # False
14 print(is_palindrome("")) # True
15 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 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.

## Task4 : Observation:Bank Account Class

### Prompt:

```
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
```

### Code:

```
task.py > BankAccount > __init__
1  #Test Cases:
2  # acc = BankAccount(1000)
3  # acc.deposit(500)
4  # assert acc.get_balance() == 1500
5  # acc.withdraw(300)
6  # assert acc.get_balance() == 1200
7  # acc.withdraw(1500) # Should print "Insufficient funds"
8  class BankAccount:
9      def __init__(self, initial_balance=0):
10         self.balance = initial_balance
11     def deposit(self, amount):
12         if amount > 0:
13             self.balance += amount
14     def withdraw(self, amount):
15         if amount > self.balance:
16             print("Insufficient funds")
17         elif amount > 0:
18             self.balance -= amount
19     def get_balance(self):
20         return self.balance
21 # Example Usage:
22 acc = BankAccount(1000)
23 acc.deposit(500)
24 print(acc.get_balance()) # 1500
25 acc.withdraw(300)
26 print(acc.get_balance()) # 1200
27 acc.withdraw(1500) # Should print "Insufficient funds"
28
```

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:

```
1  # Test Cases:
2  # assert validate_email("user@example.com") == True
3  # assert validate_email("user example.com") == False
4  # assert validate_email("@gmail.com") == False
5  # assert validate_email("user@.com") == False
6  def validate_email(email):
7      if email.count('@') != 1:
8          return False
9      local_part, domain_part = email.split('@')
10     if not local_part or not domain_part:
11         return False
12     if '.' not in domain_part:
13         return False
14     return True
15 # Example Usage:
16 print(validate_email("user@example.com"))
17 print(validate_email("user example.com"))
18 print(validate_email("@gmail.com"))
19 print(validate_email("user@.com"))
20
```

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. Edgescases such as missing symbols and improper placement were handled effectively, improving data validation reliability