

Assignment – 8.5

Name : Yashaswini

Nagireddi

Batch : 21

Ht.No : 2303A51418

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

Prompt:

Test Cases

```
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
```

code:

```

13-02-2026.py > is_valid_username
1  # Test Cases
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      # Check if the username is between 5 and 20 characters
9      if len(username) < 5 or len(username) > 20:
10         return False
11     # Check if the username starts with a letter
12     if not username[0].isalpha():
13         return False
14     # Check if the username contains only letters and digits
15     for char in username:
16         if not char.isalnum():
17             return False
18     return True
19 # Example usage
20 print(is_valid_username("User123")) # True
21 print(is_valid_username("12User")) # False
22 print(is_valid_username("Us er")) # False
23 print(is_valid_username("User")) # False
24 print(is_valid_username("User_123")) # False
25

```

output:

```

bugpy\launcher' '57356' '--' 'c:\Users\dell\OneDrive\Desktop\AI\1
3-02-2026.py'
True
False
False
False
False
PS C:\Users\dell\OneDrive\Desktop\AI>

```

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.

2. Task: Even–Odd & Type Classification

prompt:

```
assert classify_value(8) == "Even"
```

```
assert classify_value(7) == "Odd"
```

```
assert classify_value(0) == "Zero"
```

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

```
assert classify_value(2.5) == "Invalid Input"
```

Input:

```

13-02-2026.py > ...
26
27 # assert classify_value(8) == "Even"
28 #assert classify_value(7) == "Odd"
29 #assert classify_value(0) == "Zero"
30 #assert classify_value("abc") == "Invalid Input"
31 #assert classify_value(2.5) == "Invalid Input"
32 def classify_value(value):
33     if isinstance(value, int):
34         if value == 0:
35             return "Zero"
36         elif value % 2 == 0:
37             return "Even"
38         else:
39             return "Odd"
40     else:
41         return "Invalid Input"
42 # Example usage
43 print(classify_value(8))      # "Even"
44 print(classify_value(7))      # "Odd"
45 print(classify_value(0))      # "Zero"
46 print(classify_value("abc"))  # "Invalid Input"
47 print(classify_value(2.5))    # "Invalid Input"
48
49
50
51

```

Output:

```

25.18.0-win32-x64\bundled\libs\debugpy\launcher' '59098' '--' 'c:\Users\dell\One
rive\Desktop\AI\13-02-2026.py'
True
False
False
False
False
Even
Odd
Zero
Invalid Input
Invalid Input

```

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 3: Palindrome Checker**Prompt:**

```
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
```

input:

output:

```
13-02-2026.py > ...
48
49 # assert is_palindrome("Madam") == True
50 # assert is_palindrome("A man a plan a canal Panama") == True
51 # assert is_palindrome("Python") == False
52 # assert is_palindrome("") == True
53 # assert is_palindrome("a") == True
54 def is_palindrome(s):
55     # Remove spaces and convert to (function) lower: Any
56     cleaned_s = s.replace(" ", "").lower()
57     # Check if the cleaned string is equal to its reverse
58     return cleaned_s == cleaned_s[::-1]
59 # Example usage
60 print(is_palindrome("Madam")) # True
61 print(is_palindrome("A man a plan a canal Panama")) # True
62 print(is_palindrome("Python")) # False
63 print(is_palindrome("")) # True
64 print(is_palindrome("a")) # True
65
```

PROBLEMS OUTPUT TERMINAL ... Python Debug Console + - [] [] ...

```
True
True
False
True
True
PS C:\Users\dell\OneDrive\Desktop\AI>
```

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 4 Observation: BankAccount 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
```

input:

output:

```
13-02-2026.py > BankAccount > withdraw
70 # assert acc.get_balance() == 1200
71 # acc.withdraw(2000)
72 # assert acc.get_balance() == 1200
73 class BankAccount:
74     def __init__(self, initial_balance=0):
75         self.balance = initial_balance
76     def deposit(self, amount):
77         if amount > 0:
78             self.balance += amount
79     def withdraw(self, amount):
80         if 0 < amount <= self.balance:
81             self.balance -= amount
82     def get_balance(self):
83         return self.balance
84 # Example usage
85 acc = BankAccount(1000)
86 acc.deposit(500)
87 print(acc.get_balance()) # 1500
88 acc.withdraw(300)
89 print(acc.get_balance()) # 1200
90 acc.withdraw(2000)
91 print(acc.get_balance()) # 1200

PROBLEMS OUTPUT TERMINAL ... Python Debug Console + - [] [] ...

1500
1200
1200
PS C:\Users\dell\OneDrive\Desktop\AT>
```

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 5 : Email ID Validation

Prmot:

`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`

Input:

Output:

```
13-02-2026.py > ...
92
93 #assert validate_email("user@example.com") == True
94 #assert validate_email("userexample.com") == False
95 #assert validate_email("@gmail.com") == False
96 #assert validate_email("user@.com") == False
97 #assert validate_email("user@gmail") == False
98 def validate_email(email):
99     if not isinstance(email, str):
100         return False
101     if "@" not in email or "." not in email:
102         return False
103     if email[0] in "@._" or email[-1] in "@._":
104         return False
105     return True
106 # Example usage
107 print(validate_email("user@example.com")) # True
108 print(validate_email("userexample.com")) # False
109 print(validate_email("@gmail.com"))      # False
110 print(validate_email("user@.com"))        # False
111 print(validate_email("user@gmail"))       # False
112
113
```

PROBLEMS OUTPUT TERMINAL ... Python Debug Console + - □

```
True
False
False
True
False
PS C:\Users\dell\OneDrive\Desktop\AI> |
```

Observation:

All 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.

