

# Assignment–8.5

Name : J.Akshaya

Batch : 22

Ht.No : 2303A51629

## 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
assert is_valid_username("User_123") == False
valid usernames correctly.
```

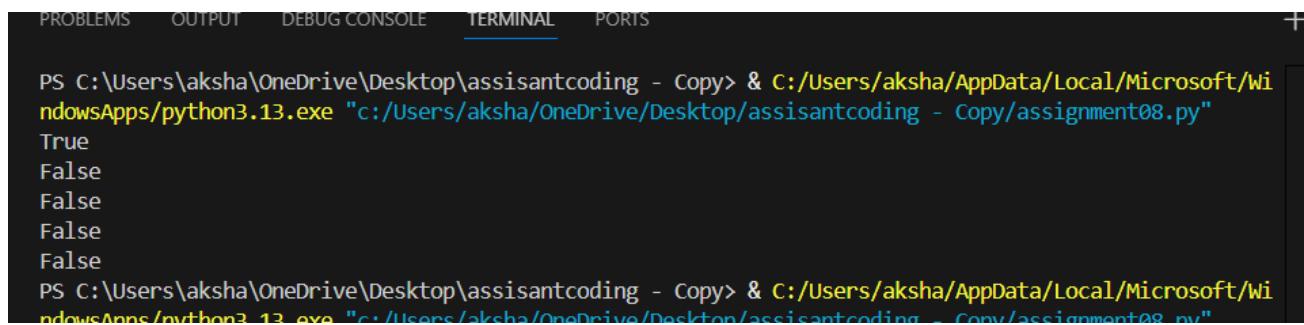
Code:

The screenshot shows a code editor interface with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, ...
- Search Bar:** assitantcoding - Copy
- Editor Area:** The code for "assignment08.py" is displayed, which includes a function to validate a username and several test cases.
- Explorer Panel:** Shows a file tree with various Python files like day02.py, leap\_year.py, assignmnt01.py, Untitled-1, assignment3.py, lab04.py, assignmnet5.5.py, assignment08.py, and Untitled-2.
- ASSANTCODING - COPY:** A folder containing files such as 13-02-2026.py, activity.log, assignmnt01.py, assignment 7.py, assignment3.py, assignment05.py, assignment08.py, assignmnet5.5.py, assss06.py, da01.py, day02.py, errorassignmnet.py, lab01question2.py, and lab03.ov.
- Outline Panel:** Located at the bottom left of the interface.

```
1 # 1.Task Description (Username Validator - Apply AI in Authentication Context) Promt:
2 # # Test Cases
3 # assert is_valid_username("User123") == True
4 # assert is_valid_username("12User") == False
5 # assert is_valid_username("Us er") == False
6 # assert is_valid_username("User") == False
7 # assert is_valid_username("User_123") == False
8 #code
9 def is_valid_username(username):
10     # Check if the username is at least 5 characters long
11     if len(username) < 5:
12         return False
13     # Check if the username starts with a letter
14     if not username[0].isalpha():
15         return False
16     # Check if the username contains only letters and digits
17     for char in username:
18         if not char.isalnum():
19             return False
20     return True
21 # Test Cases
22 print(is_valid_username("User123")) # True
23 print(is_valid_username("12User")) # False
24 print(is_valid_username("Us er")) # False
25 print(is_valid_username("User")) # False
26 print(is_valid_username("User_123")) # False
27 |
```

## Output:



A screenshot of a terminal window from a code editor. The tabs at the top are PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined), and PORTS. The terminal shows two lines of command-line output. The first line starts with 'PS C:\Users\aksha\OneDrive\Desktop\assistantcoding - Copy> & C:/Users/aksha/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/aksha/OneDrive/Desktop/assistantcoding - Copy/assignment08.py"' followed by several 'False' and 'True' values. The second line is identical. This indicates that the AI-generated assert statements are failing, suggesting the function does not fully implement the validation logic.

```
PS C:\Users\aksha\OneDrive\Desktop\assistantcoding - Copy> & C:/Users/aksha/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/aksha/OneDrive/Desktop/assistantcoding - Copy/assignment08.py"
False
False
False
False
False
PS C:\Users\aksha\OneDrive\Desktop\assistantcoding - Copy> & C:/Users/aksha/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/aksha/OneDrive/Desktop/assistantcoding - Copy/assignment08.py"
```

## 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:

```
27  
28  
29 # Task 2: Even-Odd & Type Classification prompt:  
30 # assert classify_value(8) == "Even"  
31 # assert classify_value(7) == "Odd"  
32 # assert classify_value(0) == "zero"  
33 # assert classify_value("abc") == "Invalid Input"  
34 # assert classify_value(2.5) == "Invalid Input"  
35 def classify_value(value):  
36     # Check if the value is an integer  
37     if not isinstance(value, int):  
38         return "Invalid Input"  
39     # Check if the value is zero  
40     if value == 0:  
41         return "Zero"  
42     # Check if the value is even or odd  
43     elif value % 2 == 0:  
44         return "Even"  
45     else:  
46         return "Odd"  
47 # Test Cases  
48 print(classify_value(8))    # Even  
49 print(classify_value(7))    # Odd  
50 print(classify_value(0))    # Zero  
51 print(classify_value("abc")) # Invalid Input  
52 print(classify_value(2.5))  # Invalid Input  
53
```

## Output:

```
Zero
Invalid Input
Invalid Input
PS C:\Users\aksha\OneDrive\Desktop\assisantcoding - Copy> & C:/Users/aksha/AppData/Local/Microsoft/Wi
ndowsApps/python3.13.exe "c:/Users/aksha/OneDrive/Desktop/assisantcoding - Copy/assignment08.py"
Even
Odd
Zero
Invalid Input
Invalid Input
PS C:\Users\aksha\OneDrive\Desktop\assisantcoding - Copy>
```

## 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:

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

## Code:

```
assignment08.py > ...
54  # for task2
55  #
56 # Task 3: Palindrome Checker Promt:
57 # assert is_palindrome("Madam") == True
58 # assert is_palindrome("A man a plan a canal Panama") == True
59 # assert is_palindrome("Python") == False
60 # assert is_palindrome("") == True
61 # assert is_palindrome("a") == True
62 def is_palindrome(s):
63     # Remove spaces and convert to lowercase
64     cleaned_s = s.replace(" ", "").lower()
65     # Check if the cleaned string is equal to its reverse
66     return cleaned_s == cleaned_s[::-1]
67 # Test Cases
68 print(is_palindrome("Madam")) # True
69 print(is_palindrome("A man a plan a canal Panama")) # True
70 print(is_palindrome("Python")) # False
71 print(is_palindrome("")) # True
72 print(is_palindrome("a")) # True
73 print(is_palindrome("No 'x' in Nixon")) # True
```

## Output:

```
False
PS C:\Users\aksha\OneDrive\Desktop\assisantcoding - Copy> & C:/Users/aksha/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/aksha/OneDrive/Desktop/assisantcoding - Copy/assignment08.py"
True
True
False
True
True
False
PS C:\Users\aksha\OneDrive\Desktop\assisantcoding - Copy>
```

**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****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 4 Observation: BankAccount Class Promt:  
# 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  
        else:  
            print("Deposit amount must be positive.")  
  
    def withdraw(self, amount):  
        if amount > self.balance:  
            print("Insufficient funds. Withdrawal denied.")  
        elif amount <= 0:  
            print("Withdrawal amount must be positive.")  
        else:  
            self.balance -= amount
```

```
def withdraw(self, amount):
    if amount > self.balance:
        print("Insufficient funds. Withdrawal denied.")
    elif amount <= 0:
        print("Withdrawal amount must be positive.")
    else:
        self.balance -= amount

def get_balance(self):
    return self.balance

# Test Cases
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 (unchanged due to insufficient funds)
```

**Output:**

```
True
False
PS C:\Users\aksha\OneDrive\Desktop\assisantcoding - Copy> & C:/Users/aksha/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/aksha/OneDrive/Desktop/assisantcoding - Copy/assignment08.py"
1500
1200
Insufficient funds. Withdrawal denied.
1200
PS C:\Users\aksha\OneDrive\Desktop\assisantcoding - Copy>
```

### **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:**

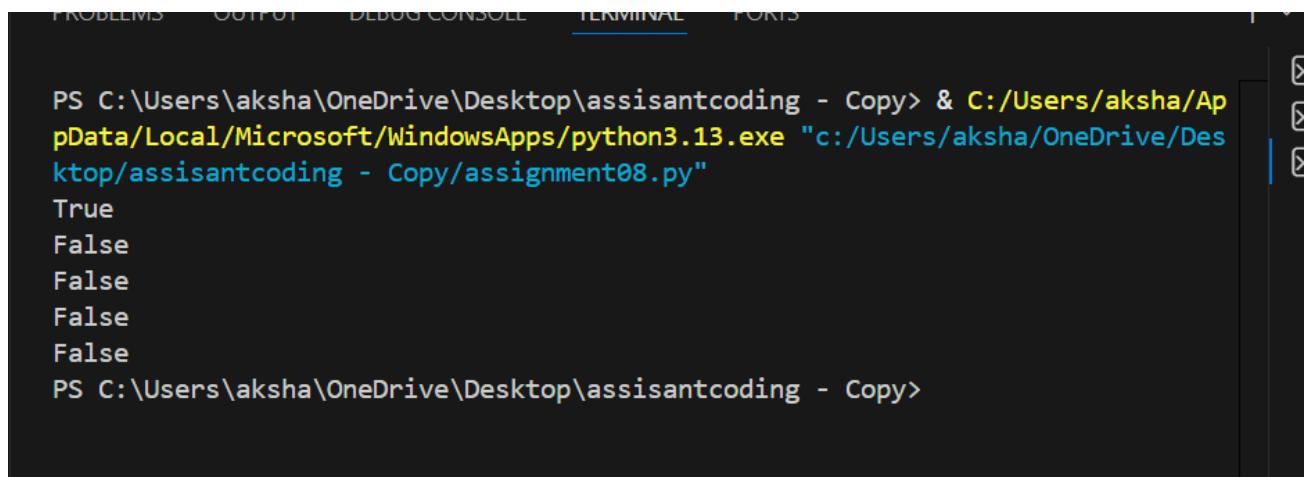
```
assertvalidate_email("user@example.com") == True
assert validate_email("userexample.com") == False
assert validate_email("@gmail.com") == False
assert validate_email("user@.com") == False
assertvalidate_email("user@gmail") == False
```

Code:

```
assignment08.py > ...
115  # Task5:EmailIDValidation Prmot:
116  # assertvalidate_email("user@example.com")==True
117  # assert validate_email("userexample.com") == False
118  # assert validate_email("@gmail.com") == False
119  # assert validate_email("user@.com") == False
120  # assertvalidate_email("user@gmail")==False
121  def validate_email(email):
122      # Check if the email contains exactly one '@' symbol
123      if email.count('@') != 1:
124          |   return False
125      local_part, domain_part = email.split('@')
126      # Check if the local part is not empty
127      if not local_part:
128          |   return False
129      # Check if the domain part contains at least one '.' symbol
130      if '.' not in domain_part:
131          |   return False
132      # Check if the domain part has a valid format (e.g., "example.com")
133      domain_name, extension = domain_part.rsplit('.', 1)
134      if not domain_name or not extension:
135          |   return False
136      return True
137  # Test Cases
138  print(validate_email("user@example.com")) # True
139  print(validate_email("userexample.com")) # False
140  print(validate_email("@gmail.com")) # False
141  print(validate_email("user@.com")) # False
142  print(validate_email("user@gmail")) # False
```

⊕ Ln 142, Col 49 Spaces: 4 UTF-8 CRLF

Output:



A screenshot of a terminal window from a code editor. The tabs at the top are PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined), and PORTS. The terminal output shows the following:

```
PS C:\Users\aksha\OneDrive\Desktop\assisantcoding - Copy> & C:/Users/aksha/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/aksha/OneDrive/Desktop/assisantcoding - Copy/assignment08.py"
True
False
False
False
False
PS C:\Users\aksha\OneDrive\Desktop\assisantcoding - Copy>
```

### **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