

## **ASSIGNMENT-8.5**

### **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:

- Username length must be between 5 and 15 characters.
- Must contain only alphabets and digits.
- Must not start with a digit.
- 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.

Output:

The screenshot shows the Gemini AI development environment. In the center is a code editor window titled "username validator" containing Python code. The code defines a function `is\_valid\_username` that checks if a username is valid based on length and character rules. Below the code, there are three test cases using `assert` statements. A tooltip from the AI assistant says "#1 (Username Validator – Apply AI in Authentication Context)". The AI assistant interface includes buttons for "Accept & Run", "Accept", and "Cancel". A message box asks "What can I help you build?" with a plus sign icon. The bottom right of the interface shows "Gemini 2.5 Flash".

```

# Function
def is_valid_username(username):
    if not isinstance(username, str):
        return False
    if len(username) < 5:
        return False
    if not username.isalnum():
        return False
    if username[0].isdigit():
        return False
    return True

# Test cases
assert is_valid_username('user123')
assert is_valid_username('user_123')
assert is_valid_username('user123_')

```

## 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.

## Output:

The screenshot shows the Gemini AI interface. On the left, there's a code editor window titled 'Gemini' containing Python code for a 'classify\_input' function and some test cases. A tooltip box is overlaid on the code, providing instructions: 'Let's create a Python function classify\_input that determines if an input is an even or odd integer, or classifies its type if it's not an integer, and add some test cases.' Below the code editor, a text input field says 'What can I help you build?'. At the bottom right, it says 'Gemini 2.5 Flash'.

```
+# Function
+def classify_input(value):
+    if isinstance(value, int):
+        if value % 2 == 0:
+            return f"{value} is an even integer."
+        else:
+            return f"{value} is an odd integer."
+    else:
+        return f"{value} is not an integer."
+
+# Test cases
+assert classify_input(4) == "4 is an even integer."
+assert classify_input(7) == "7 is an odd integer."
+assert classify_input("hello") == "hello is not an integer."
+assert classify_input(3.14) == "3.14 is not an integer."
+assert classify_input([1, 2]) == "[1, 2] is not an integer."
```

## 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:
  - Ignore case, spaces, and punctuation.
  - 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.

Output:

The screenshot shows a code editor interface with a sidebar on the left containing icons for file operations like New, Open, Save, and Run. The main area is titled "Task-03" and contains a code block labeled "Gemini". The code is as follows:

```
+# Function
+# Function
+def is_palindrome(text):
+    # Normalize the string: convert to lowercase and remove non-alphanumeric characters
+    normalized_text = text.lower().replace(" ", "")
+    # Check if the normalized text is equal to its reverse
+    return normalized_text == normalized_text[::-1]
+
+# Test cases
+assert is_palindrome("Racecar") == True
+assert is_palindrome("A man a plan a canal Panama") == True
+assert is_palindrome("Hello") == False
+assert is_palindrome("No") == True
+assert is_palindrome("Python") == False
+assert is_palindrome("Was it a car or a cat I saw") == True
```

A modal window titled "#3 (Palindrome Checker – Apply AI for String Normalization)" is open, containing the following text:

Let's create a Python function `is_palindrome` that checks if a string is a palindrome, ignoring non-alphanumeric characters and case, and add some test cases.

Accept & Run    Accept    Cancel

What can I help you build?

Gemini 2.5 Flash

## 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.

Output:

The screenshot shows a code editor interface with a sidebar on the left containing icons for search, code, text, run, and settings. The main area has tabs for 'Task-04' and 'Gemini'. The 'Gemini' tab is active, displaying a Python code snippet for a `BankAccount` class. The code includes methods for `deposit` and `withdraw`, both of which raise `ValueError` if the input is not a float or if the withdrawal amount is greater than the current balance. A tooltip from Gemini suggests creating a `BankAccount` class with methods for `deposit`, `withdraw`, and `get_balance`. The 'Task-04' tab shows a test script with several assert statements. The first assert checks for a `ValueError` when initializing with a negative balance. Subsequent tests check for correct behavior with positive balances and various deposit and withdraw amounts. The script concludes with a print statement indicating all tests passed.

```
+# Function: BankAccount Class
+class BankAccount:
+    def __init__(self, initial_balance=0):
+        if not isinstance(initial_balance, (int, float)) or initial_balance < 0:
+            raise ValueError
+        self.balance = initial_balance
+
+    def deposit(self, amount):
+        if not isinstance(amount, (int, float)):
+            raise ValueError
+        self.balance += amount
+        return self.balance
+
+    def withdraw(self, amount):
+        if not isinstance(amount, (int, float)):
+            raise ValueError
+        if amount > self.balance:
+            raise ValueError
+        self.balance -= amount
+        return self.balance
+
try:
    account10 = BankAccount(-50)
    assert False, "Test 10 Failed: Expected ValueError for negative initial balance"
except ValueError as e:
    assert str(e) == "Initial balance must be a non-negative number.", f"Test 10 Failed: Wrong error message: {e}"

# Test 11: Initial balance as float
account11 = BankAccount(100.50)
assert account11.get_balance() == 100.50, f"Test 11 Failed: Expected 100.50, got {account11.get_balance()}"

# Test 12: Deposit float amount
account12 = BankAccount(50)
account12.deposit(25.75)
assert account12.get_balance() == 75.75, f"Test 12 Failed: Expected 75.75, got {account12.get_balance()}"

# Test 13: Withdraw float amount
account13 = BankAccount(100.25)
account13.withdraw(18.15)
assert account13.get_balance() == 82.1, f"Test 13 Failed: Expected 82.1, got {account13.get_balance()}"

print("All BankAccount tests passed!")

... All BankAccount tests passed!
```

## 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.

Output:

```

Task-05
Gemini
+ Function
import re

def is_valid_email(email):
    if not isinstance(email, str):
        return False
    # Regular expression to check if the email has a valid format
    # This pattern checks if it contains '@' and '.'
    # The top-level domain
    pattern = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"
    if re.fullmatch(pattern, email):
        return True
    else:
        return False

# Test cases
assert is_valid_email("test@example.com")
assert is_valid_email("john.doe@example.org")
assert is_valid_email("invalid_email@.com")

```

**M** #5 (Email ID Validation – Apply AI for Data Validation)

<> Empty cell

Let's create a Python function `is_valid_email` that validates email addresses based on common patterns, and include test cases for various

Accept & Run ✓ Accept ✖ Cancel

What can I help you build?

Gemini 2.5 Flash ▾ ▶

Gemini can make mistakes so double-check it and use code with caution. Learn more