

**School of Computer Science and Artificial Intelligence****Lab Assignment # 8.5**

---

<b>Program</b>	<b>:</b> B. Tech (CSE)
<b>Specialization</b>	<b>:</b> -
<b>Course Title</b>	<b>:</b> AI Assisted Coding
<b>Course Code</b>	<b>:</b> 23CS002PC304
<b>Semester</b>	<b>:</b> II
<b>Academic Session</b>	<b>:</b> 2025-2026
<b>Name of Student</b>	<b>:</b> G Mani prasad
<b>Enrollment No.</b>	<b>:</b> 2403A51L48
<b>Batch No.</b>	<b>:</b> 52
<b>Date</b>	<b>:</b> 30/01/26

---

**Submission Starts here****Screenshots:**

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

**Prompt:**

Generate at least 3 assert-based test cases for a Python function `is_valid_username(username)` with the following rules: - Username length must be between 5 and 15 characters. - Only alphabets and digits allowed. - Username must not start with a digit. - No spaces allowed. Then implement the function using Test-Driven Development so that all tests pass.

Code:

```
[23]  ✓ 0s  # Test Cases for is_valid_username
def is_valid_username(username):
    # Rule 1: Username length must be between 5 and 15 characters.
    if not (5 <= len(username) <= 15):
        return False

    # Rule 2: Only alphabets and digits allowed.
    if not username.isalnum():
        return False

    # Rule 3: Username must not start with a digit.
    if username[0].isdigit():
        return False

    # This is a stub, we will implement it using TDD
    return True # Temporarily return True to pass the current tests (length, alnum, and starts with digit are okay)

# Test 1: Valid username
assert is_valid_username("validUser123") == True, "Test 1 Failed: Valid username 'validUser123'"

# Test 2: Username too short
assert is_valid_username("usr") == False, "Test 2 Failed: Username 'usr' (too short)"

# Test 3: Username too long
assert is_valid_username("verylongusername12345") == False, "Test 3 Failed: Username 'verylongusername12345' (too long)"

# Test 4: Username with special characters
assert is_valid_username("user@name") == False, "Test 4 Failed: Username 'user@name' (special characters)"

# Test 5: Username starts with a digit
assert is_valid_username("1username") == False, "Test 5 Failed: Username '1username' (starts with digit)"

# Test 6: Username with spaces
assert is_valid_username("user name") == False, "Test 6 Failed: Username 'user name' (contains spaces)"

print("All initial tests defined. Now let's implement the function to pass them.")
```

Output:

... All initial tests defined. Now let's implement the function to pass them.

Task Description #2 (Even–Odd & Type Classification – Apply AI for Robust Input Handling)

Prompt:

Generate at least 3 assert test cases for a Python function `classify_value(x)` and implement the function using conditional logic and loops with the following rules:

- If input is integer, return "Even" or "Odd".
- If input is 0, return "Zero".
- If input is non-numeric, return "Invalid Input".

Ensure all tests pass.

Code:

```
◆ Gemini
Ds
# Test Cases for classify_value

def classify_value(x):
    if x == 0:
        return "Zero"
    if isinstance(x, int):
        if x % 2 == 0:
            return "Even"
        else:
            return "Odd"
    # Rule: If input is non-numeric, return "Invalid Input".
    # This also handles floats as they are not integers and not 0.
    return "Invalid Input"

# Test 1: Input is 0
assert classify_value(0) == "Zero", "Test 1 Failed: Input 0 should be 'Zero'"

# Test 2: Input is an even integer
assert classify_value(4) == "Even", "Test 2 Failed: Input 4 should be 'Even'"

# Test 3: Input is an odd integer
assert classify_value(7) == "Odd", "Test 3 Failed: Input 7 should be 'Odd'"

# Test 4: Input is a non-numeric string
assert classify_value("hello") == "Invalid Input", "Test 4 Failed: Input 'hello' should be 'Invalid Input'"

# Test 5: Input is a float (non-integer numeric)
assert classify_value(3.14) == "Invalid Input", "Test 5 Failed: Input 3.14 should be 'Invalid Input'"

print("All initial tests defined. Now let's implement the function to pass them.")
```

Output:

```
All initial tests defined. Now let's implement the function to pass them.
```

Task Description #3 (Palindrome Checker – Apply AI for String Normalization)

Prompt:

Generate at least 3 assert test cases for a function is\_palindrome(text) and implement the function.

Requirements:

- Ignore case, spaces, and punctuation.
- Handle empty string and single characters.
- Ensure all tests pass.

Code:

```
▶ # Test Cases for is_palindrome

def is_palindrome(text):
    # Normalize the string: convert to lowercase and remove non-alphanumeric characters
    normalized_text = ''.join(char.lower() for char in text if char.isalnum())
    # Check if the normalized string is equal to its reverse
    return normalized_text == normalized_text[::-1]

# Test 1: Basic palindrome
assert is_palindrome("madam") == True, "Test 1 Failed: 'madam'"

# Test 2: Palindrome with mixed case
assert is_palindrome("Racecar") == True, "Test 2 Failed: 'Racecar'"

# Test 3: Palindrome with spaces and punctuation
assert is_palindrome("A man, a plan, a canal: Panama") == True, "Test 3 Failed: 'A man, a plan, a canal: Panama'"

# Test 4: Not a palindrome
assert is_palindrome("hello") == False, "Test 4 Failed: 'hello'"

# Test 5: Empty string
assert is_palindrome("") == True, "Test 5 Failed: Empty string"

# Test 6: Single character
assert is_palindrome("a") == True, "Test 6 Failed: Single character 'a'"

# Test 7: Single character with space
assert is_palindrome(" A ") == True, "Test 7 Failed: Single character with space ' A '"

# Test 8: Not a palindrome with spaces and punctuation
assert is_palindrome("not a palindrome") == False, "Test 8 Failed: 'not a palindrome'"

print("All initial tests defined. Now let's implement the function to pass them.")
```

Output:

```
All initial tests defined. Now let's implement the function to pass them.
```

Task Description #4 (BankAccount Class – Apply AI for Object-Oriented Test-Driven Development)

Prompt:

Generate at least 3 assert-based test cases for a Python BankAccount class and then implement the class.

Methods required:

- deposit(amount)
- withdraw(amount)
- get\_balance()

Ensure implementation passes all tests.

Code:

```
▶ # Test Cases for BankAccount class

class BankAccount:
    def __init__(self):
        # This is a stub, we will implement it using TDD
        self._balance = 0 # Initialize with a default, but tests will dictate behavior

    def deposit(self, amount):
        # Implement deposit functionality
        if amount > 0:
            self._balance += amount
            return True
        return False # For negative amounts, as per Test 5

    def withdraw(self, amount):
        # Implement withdraw functionality
        if amount > 0 and self._balance >= amount:
            self._balance -= amount
            return True
        return False

    def get_balance(self):
        # This is a stub, we will implement it using TDD
        return self._balance # Return the actual balance

# Test 1: Initial balance should be zero
account = BankAccount()
assert account.get_balance() == 0, "Test 1 Failed: Initial balance not zero"

# Test 2: Deposit a positive amount
account = BankAccount()
assert account.deposit(100) == True, "Test 2 Failed: Deposit of 100 should succeed"
assert account.get_balance() == 100, "Test 2 Failed: Balance incorrect after deposit"

# Test 3: Withdraw with sufficient funds
account = BankAccount()
account.deposit(200)
assert account.withdraw(50) == True, "Test 3 Failed: Withdrawal of 50 should succeed"
assert account.get_balance() == 150, "Test 3 Failed: Balance incorrect after withdrawal"

# Test 4: Withdraw with insufficient funds
account = BankAccount()
account.deposit(50)
assert account.withdraw(100) == False, "Test 4 Failed: Withdrawal of 100 (insufficient) should fail"
assert account.get_balance() == 50, "Test 4 Failed: Balance changed after failed withdrawal"

# Test 5: Try to deposit a negative amount
account = BankAccount()
assert account.deposit(-50) == False, "Test 5 Failed: Deposit of negative amount should fail"
assert account.get_balance() == 0, "Test 5 Failed: Balance changed after failed negative deposit"

# Test 6: Try to withdraw a negative amount
account = BankAccount()
account.deposit(100)
assert account.withdraw(-20) == False, "Test 6 Failed: Withdrawal of negative amount should fail"
assert account.get_balance() == 100, "Test 6 Failed: Balance changed after failed negative withdrawal"

print("All initial tests defined. Now let's implement the BankAccount class to pass them.")
```

```
All initial tests defined. Now let's implement the BankAccount class to pass them.
```

Task Description #5 (Email ID Validation – Apply AI for Data Validation)

Prompt:

Generate at least 3 assert test cases for a function validate\_email(email) and implement the function.

Rules:

- Must contain @ and .
- Must not start or end with special characters.
- Handle invalid formats gracefully.

Ensure all tests pass.

Code:

```
# Test Cases for validate_email

def validate_email(email):
    # Rule: Must contain @ and .
    if "@" not in email or "." not in email:
        return False

    # Rule: Must not start or end with special characters.
    # Using isalnum() to check for alphanumeric characters
    if not email[0].isalnum() or not email[-1].isalnum():
        return False

    # This is a stub, we will implement it using TDD
    return True

# Test 1: Valid email
assert validate_email("test@example.com") == True, "Test 1 Failed: Valid email"

# Test 2: Missing '@'
assert validate_email("testexample.com") == False, "Test 2 Failed: Missing '@'"

# Test 3: Missing '..'
assert validate_email("test@examplecom") == False, "Test 3 Failed: Missing '..'"

# Test 4: Starts with special character
assert validate_email("@test@example.com") == False, "Test 4 Failed: Starts with special character"

# Test 5: Ends with special character
assert validate_email("test@example.com.") == False, "Test 5 Failed: Ends with special character"

# Test 6: Invalid format (e.g., '@.com')
assert validate_email("@.com") == False, "Test 6 Failed: Invalid format '@.com'"

# Test 7: Valid with numbers/hyphens in domain
assert validate_email("user@my-domain123.net") == True, "Test 7 Failed: Valid email with complex domain"

# Test 8: Valid with numbers/hyphens in local part
assert validate_email("user.name-123@domain.org") == True, "Test 8 Failed: Valid email with complex local part"

print("All initial tests defined. Now let's implement the function to pass them.")
```

Output:

```
... All initial tests defined. Now let's implement the function to pass them.
```