

Lab 8: Test-Driven Development with AI – Generating and Working with Test Cases

NAME : R.Shivani

ROLL NO : 2403A52411

BATCH : 15

Task 1: Apply AI to generate at least 3 assert test cases for `is_strong_password(password)` and implement the validator function.

- Requirements:
 - Password must have at least 8 characters.
 - Must include uppercase, lowercase, digit, and special character.
 - Must not contain spaces.

Example Assert Test Cases:

```
assert is_strong_password("Abcd@123") == True  
assert is_strong_password("abcd123") == False  
assert is_strong_password("ABCD@1234") == True
```

Expected Output #1:

- Password validation logic passing all AI-generated test cases.

PROMPT : generate at least 3 assert test cases for is_strong_password(password) and implement the validator function.

- Requirements:
 - Password must have at least 8 characters.
 - Must include uppercase, lowercase, digit, and special character.
 - Must not contain spaces.

CODE : import re

```
def is_strong_password(password):
    # At least 8 characters if
    if len(password) < 8:
        return False

    # No spaces allowed
    if ' ' in password:
        return False

    # At least one uppercase, one lowercase, one digit, one special character
```

```

if not re.search(r'[A-Z]', password):
    return False if not re.search(r'[a-
z]', password): return False if not
    re.search(r'\d', password):
        return False if not re.search(r'^A-Za-z0-
9]', password):
    return False
return True

```

```

# AI-generated assert test cases assert
is_strong_password("Abcd@123") == True
print("All AI-generated test cases passed.")

```

OUTPUT :

The screenshot shows the Microsoft Visual Studio Code interface with the following details:

- File Explorer:** Shows a folder named "Python Dev" containing "ai lab assg8.1.py".
- Code Editor:** Displays the Python code for password validation and AI-generated test cases.
- Variables, Watch, Call Stack, Breakpoints:** Side panels showing no active variables or breakpoints.
- Terminal:** Shows command-line output indicating all AI-generated test cases passed.
- Output:** Shows AI-generated test cases and a note about the corrected function.
- CHAT:** A sidebar with a message from AI stating: "Expected Output #1: • Password validation logic passing all AI-generated test cases." and "Here is the corrected and completed is_strong_password function, along with at least 3 assert test cases:" followed by the corrected code.
- Status Bar:** Shows file path, line 8, column 4, and Python 3.11.9 (Microsoft Store) version.

OBSERVATION : Objective

Create a Python function `is_strong_password(password)` to check password strength based on defined rules, and generate at least **3 assert test cases** for validation using AI.

Password Requirements A

valid password must:

1. Be at least 8 characters long.
2. Include at least:
 - o One uppercase letter
 - o One lowercase letter
 - o One digit
 - o One special character (e.g., !@#\$%^&*)
3. Must NOT contain spaces

Task 2: Use AI to generate at least 3 assert test cases for a `classify_number(n)` function.

Implement using loops.

- Requirements:
 - o Classify numbers as Positive, Negative, or Zero.
 - o Handle invalid inputs like strings and None.
 - o Include boundary conditions (-1, 0, 1).

Example Assert Test Cases:

```
assert classify_number(10) == "Positive"
assert classify_number(-5) == "Negative"
assert classify_number(0) == "Zero"
```

Expected Output #2:

- Classification logic passing all assert tests.

PROMPT : generate at least 3 assert test cases for a `classify_number(n)` function. Implement using loops.

- Requirements:
 - o Classify numbers as Positive, Negative, or Zero.
 - o Handle invalid inputs like strings and None.
 - o Include boundary conditions (-1, 0, 1).

CODE : `def classify_number(n):`

"""

Classifies a number as:

- "Perfect" if the sum of its proper divisors equals the number.
- "Abundant" if the sum of its proper divisors is greater than the number.
- "Deficient" if the sum of its proper divisors is less than the number.

"""

```
if n <= 0: return "Invalid" # Only positive integers  
are valid  
  
divisor_sum = 0 for i in range(1, n // 2 + 1): # Loop  
through proper divisors if n % i == 0: divisor_sum += i  
  
if divisor_sum == n:  
    return "Perfect"  
elif divisor_sum > n:  
    return "Abundant"  
else:  
    return "Deficient"  
  
# Test cases assert classify_number(6) == "Perfect", "Test case 1 failed" # 6 = 1 + 2 + 3  
assert classify_number(12) == "Abundant", "Test case 2 failed" # 12 < 1 + 2 + 3 + 4 + 6  
assert classify_number(8) == "Deficient", "Test case 3 failed" # 8 > 1 + 2 + 4 assert  
classify_number(0) == "Invalid", "Test case 4 failed" # Invalid input assert  
classify_number(-5) == "Invalid", "Test case 5 failed" # Invalid input  
  
print("All test cases passed!")
```

OUTPUT :

The screenshot shows a Microsoft Visual Studio Code interface. In the center, there's a code editor with a Python file named `t3.py` containing the following code:

```

1 def classify_number(n):
2     if n > 0:
3         return "Positive"
4     elif n < 0:
5         return "Negative"
6     else:
7         return "Zero"
8
9 # Test cases
10 assert classify_number(6) == "Positive", "Test case 1 failed" # 6 = 1 + 5
11 assert classify_number(12) == "Positive", "Test case 2 failed" # 12 > 1 + 11
12 assert classify_number(-8) == "Negative", "Test case 3 failed" # -8 < 0
13 assert classify_number(0) == "Zero", "Test case 4 failed" # Invalid input
14 assert classify_number(-5) == "Negative", "Test case 5 failed" # Invalid input
15
16 print("All test cases passed!")

```

Below the code editor is a terminal window showing the output of running the script:

```

All test cases passed!
PS C:\Users\DELL\OneDrive\Desktop>New folder> ^C
PS C:\Users\DELL\OneDrive\Desktop>New folder>
PS C:\Users\DELL\OneDrive\Desktop>New folder> c:; cd 'c:\Users\DELL\OneDrive\Desktop>New folder'; & 'c:\Users\DELL\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\DELL\vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '57072' '--' 'C:\Users\DELL\OneDrive\Desktop>New folder\t3.py'
PS C:\Users\DELL\OneDrive\Desktop>New folder> All test cases passed!
PS C:\Users\DELL\OneDrive\Desktop>New folder>

```

To the right of the terminal, a floating window titled "Classification with Loops – Apply AI for Edge Case Handling" provides instructions for generating test cases. It includes requirements like handling invalid inputs and boundary conditions (-1, 0, 1).

A WhatsApp message from "Varshini SR" is also visible in the bottom right corner.

OBSERVATION : Objective

Implement a Python function `classify_number(n)` using **loops** to determine whether a number is **Positive**, **Negative**, or **Zero**. Also, **handle invalid inputs** and generate **at least 3 assert test cases** using **AI**.

Requirements

1. Classification Rules:

- If $n > 0$: return "Positive" ○ If $n < 0$: return "Negative" ○ If $n == 0$: return "Zero"

2. Invalid Inputs:

- For None, strings, or non-numeric types: return "Invalid input"

3. Boundary Values to Handle:

- -1, 0, 1

Task 3: Use AI to generate at least 3 assert test cases for `is_anagram(str1, str2)` and implement the function.

- Requirements:

- Ignore case, spaces, and punctuation.
- Handle edge cases (empty strings, identical words).

Example Assert Test Cases:

```
assert is_anagram("listen", "silent") == True assert  
is_anagram("hello", "world") == False assert  
is_anagram("Dormitory", "Dirty Room") == True  
Expected Output #3:
```

- Function correctly identifying anagrams and passing all AI-generated tests.

PROMPT : Task: Use AI to generate at least 3 assert test cases for is_anagram(str1, str2) and implement the function.

- Requirements:
 - Ignore case, spaces, and punctuation.
 - Handle edge cases (empty strings, identical words).
- Function correctly identifying anagrams and passing all AI-generated tests.

CODE : def is_anagram(str1, str2):

"""

Checks if two strings are anagrams of each other.

Two strings are anagrams if they contain the same characters

in the same frequency, ignoring case and spaces.

"""

Remove spaces and convert to lowercase

str1 = str1.replace(" ", "").lower()

str2 = str2.replace(" ", "").lower()

Compare sorted versions of the strings

return sorted(str1) == sorted(str2)

```
# Test cases assert is_anagram("listen", "silent") == True, "Test case 1 failed" #  
Anagrams assert is_anagram("triangle", "integral") == True, "Test case 2 failed" #  
Anagrams assert is_anagram("hello", "world") == False, "Test case 3 failed" # Not  
anagrams  
assert is_anagram("Dormitory", "Dirty room") == True, "Test case 4 failed" # Anagrams with spaces  
and case differences assert is_anagram("Python", "Java") == False, "Test case 5 failed" # Not  
anagrams
```

```
print("All test cases passed!")
```

OUTPUT :

The screenshot shows the Microsoft Visual Studio Code interface. The left sidebar includes sections for Variables, Watch, Call Stack, and Breakpoints. The main editor window contains the following Python code:

```
def is_anagram(str1, str2):
    # Compare sorted versions of the strings
    return sorted(str1) == sorted(str2)

# Test cases
assert is_anagram("listen", "silent") == True, "Test case 1 failed"
assert is_anagram("triangle", "integral") == True, "Test case 2 failed"
assert is_anagram("hello", "world") == False, "Test case 3 failed"
assert is_anagram("Dormitory", "Dirty room") == True, "Test case 4 failed"
assert is_anagram("Python", "Java") == False, "Test case 5 failed"
print("All test cases passed!")
```

The terminal tab shows the following output:

```
IndentationError: unexpected indent
PS C:\Users\DELL\OneDrive\Desktop>New folder> ^C
PS C:\Users\DELL\OneDrive\Desktop>New folder>
PS C:\Users\DELL\OneDrive\Desktop>New folder> c:; cd 'c:\Users\DELL\One
Drive\Desktop\New folder'; & 'c:\Users\DELL\.vscode\extensions\ms-python.debug
py-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '53910' '--' 'C:\U
ser\DELL\OneDrive\Desktop\New folder\t2.py'
PS C:\Users\DELL\OneDrive\Desktop>New folder> PS C:\Users\DELL\OneDrive\Desktop>New folder>
All test cases passed!
PS C:\Users\DELL\OneDrive\Desktop>New folder>
```

The Chat panel on the right displays AI-generated requirements for the task:

- Task: Use AI to generate at least 3 assert test cases for a classify_number(n) function. Implement using loops.
- Requirements:
 - Classify numbers as Positive, Negative, or Zero.
 - Handle invalid inputs like strings and None.
 - Include boundary conditions (-1, 0, 1).

Example Assert Test Cases:

```
assert classify_number(10) == "Positive"
assert classify_number(-5) == "Negative"
assert classify_number(0) == "Zero"
```

Expected Output #2:

- Classification logic passing all assert tests

OBSERVATION : Objective

Implement the function `is_anagram(str1, str2)` that determines if two strings are **anagrams**, and use **AI to generate at least 3 assert test cases** that the function must pass.

Requirements

1. Anagram Rules:

- Two strings are anagrams if they contain the same letters in a different order.
- Ignore case, spaces, and punctuation.**

2. Edge Cases to Handle:

- Empty strings ("") ○ Identical words ("note", "note") □ **Explanation** : clean()
 - removes punctuation/spaces, converts to lowercase, and sorts the characters.
 - isalnum() ensures only letters and digits are compared.

Task 4: Ask AI to generate at least 3 assert-based tests for an Inventory class with stock management.

- Methods:
 - add_item(name, quantity) ○
 - remove_item(name, quantity) ○
 - get_stock(name) Example Assert Test
 - Cases: inv = Inventory()
 - inv.add_item("Pen", 10) assert
 - inv.get_stock("Pen") == 10
 - inv.remove_item("Pen", 5) assert
 - inv.get_stock("Pen") == 5
 - inv.add_item("Book", 3) assert
 - inv.get_stock("Book") == 3 Expected
 - Output #4:
- Fully functional class passing all assertions.

PROMPT : generate at least 3 assert-based tests for an Inventory class with stock management.

- Methods:
 - add_item(name, quantity) ○
 - remove_item(name, quantity) ○
 - get_stock(name)

CODE : from datetime import datetime

```
def validate_and_format_date(date_str):
    try:
        # Parse date in MM/DD/YYYY format date_obj =
        datetime.strptime(date_str, "%m/%d/%Y")
```

```

# Return in YYYY-MM-DD format

return date_obj.strftime("%Y-%m-%d")

except ValueError:

    return "Invalid Date"

# AI-generated assert test cases assert validate_and_format_date("10/15/2023") == "2023-10-15" assert validate_and_format_date("02/30/2023") == "Invalid Date" # Invalid day in February assert validate_and_format_date("01/01/2024") == "2024-01-01" assert validate_and_format_date("13/01/2024") == "Invalid Date" # Invalid month assert validate_and_format_date("12/31/2022") == "2022-12-31" assert validate_and_format_date("2/29/2021") == "Invalid Date" # Not a leap year assert validate_and_format_date("02/29/2024") == "2024-02-29" # Leap year

print("All AI-generated test cases passed.")

```

OUTPUT :

The screenshot shows the VS Code interface with the Python Development extension installed. The terminal at the bottom displays the message "All test cases passed!". The code editor shows the AI-generated test cases for the validate_and_format_date function. The sidebar includes sections for VARIABLES, WATCH, CALL STACK, and BREAKPOINTS. A floating panel on the right provides AI-generated test cases and requirements for classifying numbers.

```

All test cases passed!
PS C:\Users\DELL\OneDrive\Desktop\New folder> ^C
PS C:\Users\DELL\OneDrive\Desktop\New folder>
PS C:\Users\DELL\OneDrive\Desktop\New folder> <:; cd 'c:\Users\DELL\OneDrive\Desktop\New folder'; & 'c:\Users\DELL\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\DELL\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\DELL\vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '64515' '--' 'c:\Users\DELL\OneDrive\Desktop\New folder\t4.jl'
All AI-generated test cases passed.
PS C:\Users\DELL\OneDrive\Desktop\New folder>

```

OBSERVATION : Objective

Implement an Inventory class to manage stock, and use **AI to generate at least 3 assert-based test cases** to verify its methods:

Inventory Class Methods

1. **add_item(name, quantity)** ○ Adds a new item or increases stock.
2. **remove_item(name, quantity)** ○ Decreases stock if available; ignore or prevent negatives.
3. **get_stock(name)** ○ Returns current stock (default to 0 if item not present).

Task 5: Use AI to generate at least 3 assert test cases for validate_and_format_date(date_str) to check and convert dates.

- Requirements:
 - "YYYY" format. ○ Handle invalid dates.
 - Convert valid dates to "YYYY-MM-DD".

Example Assert Test Cases: assert

```
validate_and_format_date("10/15/2023") == "2023-10-15" assert  
validate_and_format_date("02/30/2023") == "Invalid Date" assert  
validate_and_format_date("01/01/2024") == "2024-01-01"
```

Expected Output #5:

- Function passes all AI-generated assertions and handles edge cases.

PROMPT : generate at least 3 assert test cases for validate_and_format_date(date_str) to check and convert dates.

- Requirements:
 - Validate "MM/DD/YYYY" format. ○ Handle invalid dates.
 - Convert valid dates to "YYYY-MM-DD".

CODE : From datetime import datetime

```
def validate_and_format_date(date_str):  
    try:
```

```

# Try to parse the date in MM/DD/YYYY format
date_obj = datetime.strptime(date_str, "%m/%d/%Y") #

Return the date in YYYY-MM-DD format return

date_obj.strftime("%Y-%m-%d") except ValueError:

return "Invalid Date"

# AI-generated assert test cases assert validate_and_format_date("10/15/2023") == "2023-
10-15" assert validate_and_format_date("02/30/2023") == "Invalid Date" # Invalid day in
February assert validate_and_format_date("01/01/2024") == "2024-01-01" assert
validate_and_format_date("13/01/2024") == "Invalid Date" # Invalid month assert
validate_and_format_date("12/31/2022") == "2022-12-31" assert
validate_and_format_date("2/29/2021") == "Invalid Date" # Not a leap year assert
validate_and_format_date("02/29/2024") == "2024-02-29" # Leap year

print("All AI-generated test cases passed.")

```

OUTPUT :

The screenshot shows the VS Code interface with the Python extension installed. The code editor has the file 't5.jl' open, containing the provided Python script. The 'Variables' sidebar shows the variable 'date_obj' is a datetime object. The 'Watch' sidebar shows the variable 'date_str' is '2023-10-15'. The 'Call Stack' sidebar is empty. The 'Breakpoints' sidebar shows a breakpoint at the final print statement. The bottom status bar shows 'Python Debugger: Current File (New folder)'. A floating AI assistant window titled 'Classification with Loops - Apply AI for Edge Case Handling' provides instructions for generating test cases for a classify_number(n) function using loops.

OBSERVATION : **Objective**

Create a function validate_and_format_date(date_str) that:

- **Validates** if the input string is a valid date in "YYYY-MM-DD" or "YYYY/MM/DD" format.
- **Converts** valid dates to "YYYY-MM-DD" format.
- **Handles invalid dates** (returns "Invalid date").

Also, use **AI to generate at least 3 assert test cases** to check functionality.

Requirements

1. **Input Format:** Accepts input like "YYYY-MM-DD" or "YYYY/MM/DD".
2. **Output Format:** Always returns "YYYY-MM-DD" (standardized).
3. **Invalid Dates:**
 - Wrong format (e.g. "2023-13-40")
 - Non-date strings (e.g. "abcd")
 - Incomplete date strings (e.g. "2022-07")