

# Lab 8.1

2303A51494

B27

## Task Description #1 (Password Strength Validator – Apply AI in Security Context)

### Prompt:

generate a python code to generate general test cases for is\_strong\_password and implement the validator function requirements are password must have atleast 8 characters and must contain uppercase, lowercase, digit and special character and must not contain spaces

### Code:

#### General test cases

```
#generate a python code to generate general test cases for is_strong_password and implement the validator function
#requirements are password must have atleast 8 characters and must contain uppercase, lowercase, digit and special character and must not contain spaces
import re
def is_strong_password(password):
    if len(password) < 8:
        return False
    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'[0-9]', password):
        return False
    if not re.search(r'[%!*?&]', password):
        return False
    if re.search(r'\s', password):
        return False
    return True

# Test cases for is_strong_password
test_cases = [
    ("Password123!", True),
    ("password", False),
    ("PASSWORD", False),
    ("Pass123", False),
    ("Pass word123!", False),
    ("P@ssw0rd", True),
    ("P@ss", False),
    ("P@ssw0rd ", False),
    ("P@ssw0rd!", True)
]
for password, expected in test_cases:
    result = is_strong_password(password)
    print(f"Password: '{password}' | Expected: {expected} | Result: {result} | Test Passed: {result == expected}")
```

### Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
Password: 'Pass word123!' | Expected: False | Result: False | Test Passed: True ...
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/ai-assisted-coding.py"
Password: 'Password123!' | Expected: True | Result: True | Test Passed: True
Password: 'password' | Expected: False | Result: False | Test Passed: True
Password: 'PASSWORD' | Expected: False | Result: False | Test Passed: True
Password: 'Pass123' | Expected: False | Result: False | Test Passed: True
Password: 'Pass word123!' | Expected: False | Result: False | Test Passed: True
Password: 'P@ssw0rd' | Expected: True | Result: True | Test Passed: True
Password: 'P@ss' | Expected: False | Result: False | Test Passed: True
Password: 'P@ssw0rd ' | Expected: False | Result: False | Test Passed: True
Password: 'P@ssw0rd!' | Expected: True | Result: True | Test Passed: True
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

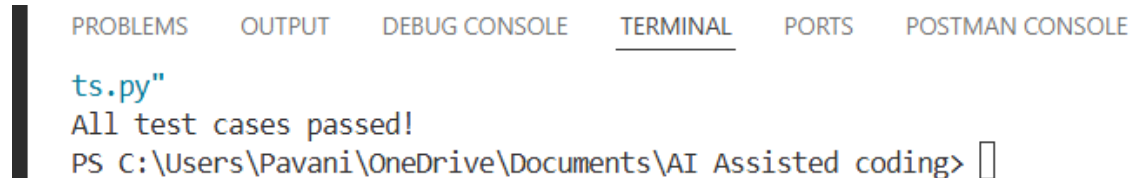
### Assert test cases:

**Prompt:** generate 5 test cases for assert test cases for is\_strong\_password function

### Code:

```
#generate 5 test cases for assert test cases for is_strong_password function
import re
def is_strong_password(password):
    if len(password) < 8:
        return False
    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'[0-9]', password):
        return False
    if not re.search(r'[@$!%*?&]', password):
        return False
    if re.search(r'\s', password):
        return False
    return True
# Assert test cases for is_strong_password
assert is_strong_password("Password123!") == True, "Test case 1 failed"
assert is_strong_password("password") == False, "Test case 2 failed"
assert is_strong_password("PASSWORD") == False, "Test case 3 failed"
assert is_strong_password("Pass123") == False, "Test case 4 failed"
assert is_strong_password("Pass word123!") == False, "Test case 5 failed"
print("All test cases passed!")
```

### Output:



The screenshot shows a code editor with a dark theme. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL', 'PORTS', and 'POSTMAN CONSOLE'. The 'TERMINAL' tab is active. Below the tabs, the text 'ts.py' is shown. The output of the program is 'All test cases passed!'. At the bottom, the command prompt shows the file path 'PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>' followed by a cursor.

### Unit testing:

**Prompt:** generate 5 unit testing test cases for is\_strong\_password function

### Code:

```

#generate 5 unit testing test cases for is_strong_password function
import unittest
import re
def is_strong_password(password):
    if len(password) < 8:
        return False
    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'[0-9]', password):
        return False
    if not re.search(r'[@$!%*?&]', password):
        return False
    if re.search(r'\s', password):
        return False
    return True
class TestIsStrongPassword(unittest.TestCase):
    def test_valid_password(self):
        self.assertTrue(is_strong_password("Password123!"))

    def test_no_uppercase(self):
        self.assertFalse(is_strong_password("password123!"))

    def test_no_lowercase(self):
        self.assertFalse(is_strong_password("PASSWORD123!"))

    def test_no_digit(self):
        self.assertFalse(is_strong_password("Password!"))

    def test_no_special_character(self):
        self.assertFalse(is_strong_password("Password123"))

    def test_contains_space(self):
        self.assertFalse(is_strong_password("Pass word123!"))
if __name__ == '__main__':    unittest.main()

```

## Output:

```

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/tes
ts.py"
.....
Ran 6 tests in 0.001s

OK
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>

```

## Pytest testing

**Prompt:** generate 5 test cases for is\_strong\_password function using pytest framework

**Code:**

#generate 5 test cases for is\_strong\_password function using pytest framework

```
import re
def is_strong_password(password):
    if len(password) < 8:
        return False
    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'[0-9]', password):
        return False
    if not re.search(r'[@$!%*?&]', password):
        return False
    if re.search(r'\s', password):
        return False
    return True
def test_valid_password():
    assert is_strong_password("Password123!") == True
def test_no_uppercase():
    assert is_strong_password("password123!") == False
def test_no_lowercase():
    assert is_strong_password("PASSWORD123!") == False
def test_no_digit():
    assert is_strong_password("Password!") == False
def test_no_special_character():
    assert is_strong_password("Password123") == False
def test_contains_space():
    assert is_strong_password("Pass word123!") == False
```

## Output:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> python -m pytest tests.py

```
===== test session starts =====
platform win32 -- Python 3.12.10, pytest-9.0.1, pluggy-1.6.0
rootdir: C:\Users\Pavani\OneDrive\Documents\AI Assisted coding
plugins: dash-3.2.0
rootdir: C:\Users\Pavani\OneDrive\Documents\AI Assisted coding
plugins: dash-3.2.0
plugins: dash-3.2.0
collected 6 items
```

tests.py .....

[100%]

6 passed in 0.08s

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> []

## Explanation:

- Different testing methods are used to verify the correctness of the is\_strong\_password validator function.
- General test cases check functionality using sample inputs and printed outputs.
- Assert tests validate expected results automatically and show errors if tests fail.
- Unit testing and Pytest frameworks provide structured, automated, and detailed test reports.
- Together, these tests ensure password security rules are correctly implemented and reliable.

## Task Description #2 (Number Classification with Loops – Apply AI for Edge Case Handling)

### General Test cases:

#### Prompt:

generate 5 general test cases for a `classify_number(n)` function that classifies a number as "positive", "negative", or "zero" and handle invalid inputs like strings or None and include boundary cases (-1, 0, 1)

#### Code:

```
#generate 5 general test cases for a classify_number(n) function
#that classifies a number as "positive", "negative", or "zero" and handle invalid inputs like strings or None and include boundary cases (-1, 0, 1)
def classify_number(n):
    if n is None:
        return "Invalid input: None"
    if isinstance(n, str):
        return "Invalid input: String"
    if n > 0:
        return "positive"
    elif n < 0:
        return "negative"
    else:
        return "zero"

# Test cases for classify_number function
test_cases = [
    (10, "positive"),
    (-5, "negative"),
    (0, "zero"),
    ("string", "Invalid input: String"),
    (None, "Invalid input: None"),
    (-1, "negative"),
    (1, "positive")
]

for n, expected in test_cases:
    result = classify_number(n)
    print(f"Input: {n} | Expected: {expected} | Result: {result} | Test Passed: {result == expected}")
```

#### Output:

- PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/Windows/PowerShell/PowerShell.exe -c "python ts.py"
- Input: 10 | Expected: positive | Result: positive | Test Passed: True
- Input: -5 | Expected: negative | Result: negative | Test Passed: True
- Input: 0 | Expected: zero | Result: zero | Test Passed: True
- Input: string | Expected: Invalid input: String | Result: Invalid input: String | Test Passed: True
- Input: None | Expected: Invalid input: None | Result: Invalid input: None | Test Passed: True
- Input: -1 | Expected: negative | Result: negative | Test Passed: True
- Input: 1 | Expected: positive | Result: positive | Test Passed: True
- PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>

### Assert testcases

**Prompt:**generate 5 assert test cases for classify number function

#### Code:

```

#generate 5 assert test cases for classify_number function
def classify_number(n):
    if n is None:
        return "Invalid input: None"
    if isinstance(n, str):
        return "Invalid input: String"
    if n > 0:
        return "positive"
    elif n < 0:
        return "negative"
    else:
        return "zero"
# Assert test cases for classify_number function
assert classify_number(10) == "positive", "Test case 1 failed"
assert classify_number(-5) == "negative", "Test case 2 failed"
assert classify_number(0) == "zero", "Test case 3 failed"
assert classify_number("string") == "Invalid input: String", "Test case 4 failed"
assert classify_number(None) == "Invalid input: None", "Test case 5 failed"
print("All test cases passed!")

```

## Output:

```

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive\Documents\AI Assisted coding\ts.py"
All test cases passed!
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>

```

## Unit test:

**Prompt:** generate 5 unit testing test cases for classify\_number function

**Code:**

```

#generate 5 unit testing test cases for classify_number function
import unittest
def classify_number(n):
    if n is None:
        return "Invalid input: None"
    if isinstance(n, str):
        return "Invalid input: String"
    if n > 0:
        return "positive"
    elif n < 0:
        return "negative"
    else:
        return "zero"
class TestClassifyNumber(unittest.TestCase):
    def test_positive_number(self):
        self.assertEqual(classify_number(10), "positive")

    def test_negative_number(self):
        self.assertEqual(classify_number(-5), "negative")

    def test_zero(self):
        self.assertEqual(classify_number(0), "zero")

    def test_string_input(self):
        self.assertEqual(classify_number("string"), "Invalid input: String")

    def test_none_input(self):
        self.assertEqual(classify_number(None), "Invalid input: None")
if __name__ == '__main__':
    unittest.main()

```

## Output:

```

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Docu
ts.py"
.....
Ran 5 tests in 0.001s
Ran 5 tests in 0.001s

OK
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> 

```

## Pytest testing:

### Prompt:

generate 5 test cases for classify\_number function using pytest framework

## Code:

```
#generate 5 test cases for classify_number function using pytest framework
def classify_number(n):
    if n is None:
        return "Invalid input: None"
    if isinstance(n, str):
        return "Invalid input: String"
    if n > 0:
        return "positive"
    elif n < 0:
        return "negative"
    else:
        return "zero"

def test_positive_number():
    assert classify_number(10) == "positive"
def test_negative_number():
    assert classify_number(-5) == "negative"
def test_zero():
    assert classify_number(0) == "zero"
def test_string_input():
    assert classify_number("string") == "Invalid input: String"
def test_none_input():
    assert classify_number(None) == "Invalid input: None"
```

## Output:

```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/tes
ts.py"
.....
Ran 5 tests in 0.001s
rootdir: C:\Users\Pavani\OneDrive\Documents\AI Assisted coding
plugins: dash-3.2.0
collected 5 items

tests.py ..... [100%]

===== 5 passed in 0.06s =====
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> █
```

## Explanation:

- All testing types verify the correctness of the classify\_number function for positive, negative, zero, and invalid inputs.
- Manual/general tests are easy to write and useful for basic debugging but require manual result checking.
- Assert tests provide quick automated validation and immediately show failures during execution.
- unittest offers a structured, class-based approach suitable for large and maintainable test suites.
- pytest is the most concise and flexible framework, widely preferred for modern automated testing.



### Task Description #3 (Anagram Checker – Apply AI for String Analysis)

#### General test cases:

**Prompt:** write a python function `is_anagram` to run general test cases the requirements are ignore case space and punctuation and handle edge cases like empty string and identical words

#### Code:

```
#write a python function is_anagram to run general test cases the requirements are ignore case space and punctuation and handle edge cases like empty string and id
import re
def is_anagram(str1, str2):
    # Remove spaces and punctuation, and convert to lowercase
    str1_cleaned = re.sub(r'[\s\W]', '', str1).lower()
    str2_cleaned = re.sub(r'[\s\W]', '', str2).lower()

    # Check if the sorted characters of both strings are the same
    return sorted(str1_cleaned) == sorted(str2_cleaned)
# Test cases for is_anagram function
test_cases = [
    ("listen", "silent", True),
    ("Triangle", "Integral", True),
    ("Dormitory", "Dirty Room", True),
    ("The eyes", "They see", True),
    ("Astronomer", "Moon starrer", True),
    ("Hello", "World", False),
    ("Anagram", "Nag a ram!", True),
    ("", "", True), # Edge case: empty strings
    ("Same", "Same", True) # Edge case: identical words
]
for str1, str2, expected in test_cases:
    result = is_anagram(str1, str2)
    print(f"String 1: '{str1}' | String 2: '{str2}' | Expected: {expected} | Result: {result} | Test Passed: {result == expected}")
```

#### Output:

```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/ts.py"
String 1: 'listen' | String 2: 'silent' | Expected: True | Result: True | Test Passed: True
String 1: 'Triangle' | String 2: 'Integral' | Expected: True | Result: True | Test Passed: True
String 1: 'Dormitory' | String 2: 'Dirty Room' | Expected: True | Result: True | Test Passed: True
String 1: 'The eyes' | String 2: 'They see' | Expected: True | Result: True | Test Passed: True
String 1: 'Astronomer' | String 2: 'Moon starrer' | Expected: True | Result: True | Test Passed: True
String 1: 'Hello' | String 2: 'World' | Expected: False | Result: False | Test Passed: True
String 1: 'Anagram' | String 2: 'Nag a ram!' | Expected: True | Result: True | Test Passed: True
String 1: '' | String 2: '' | Expected: True | Result: True | Test Passed: True
String 1: 'Same' | String 2: 'Same' | Expected: True | Result: True | Test Passed: True
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

#### Assert test cases

**Prompt:** write assert test cases for `is_anagram` function with the same requirements as above

#### Code:

```
#write assert test cases for is_anagram function with the same requirements as above
import re
def is_anagram(str1, str2):
    # Remove spaces and punctuation, and convert to lowercase
    str1_cleaned = re.sub(r'[\s\W]', '', str1).lower()
    str2_cleaned = re.sub(r'[\s\W]', '', str2).lower()

    # Check if the sorted characters of both strings are the same
    return sorted(str1_cleaned) == sorted(str2_cleaned)

# Assert test cases for is_anagram function
assert is_anagram("listen", "silent") == True, "Test case 1 failed"
assert is_anagram("Triangle", "Integral") == True, "Test case 2 failed"
assert is_anagram("Dormitory", "Dirty Room") == True, "Test case 3 failed"
assert is_anagram("The eyes", "They see") == True, "Test case 4 failed"
assert is_anagram("Astronomer", "Moon staler") == True, "Test case 5 failed"
assert is_anagram("Hello", "World") == False, "Test case 6 failed"
assert is_anagram("Anagram", "Nag a ram!") == True, "Test case 7 failed"
assert is_anagram("", "") == True, "Test case 8 failed" # Edge case: empty strings
assert is_anagram("Same", "Same") == True, "Test case 9 failed" # Edge case: identical words
print("All test cases passed!")
```

## Output:

```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/Windows/PowerShell/PowerShell.exe -Command python ts.py
All test cases passed!
```

## Unit testing

**Prompt:**write unit testing test cases for is\_anagram function with the same requirements as above

## Code:

```
#write unit testing test cases for is_anagram function with the same requirements as above
import unittest
import re
def is_anagram(str1, str2):
    # Remove spaces and punctuation, and convert to lowercase
    str1_cleaned = re.sub(r'[\s\W]', '', str1).lower()
    str2_cleaned = re.sub(r'[\s\W]', '', str2).lower()

    # Check if the sorted characters of both strings are the same
    return sorted(str1_cleaned) == sorted(str2_cleaned)

class TestIsAnagram(unittest.TestCase):
    def test_anagram(self):
        self.assertTrue(is_anagram("listen", "silent"))
        self.assertTrue(is_anagram("Triangle", "Integral"))
        self.assertTrue(is_anagram("Dormitory", "Dirty Room"))
        self.assertTrue(is_anagram("The eyes", "They see"))
        self.assertTrue(is_anagram("Astronomer", "Moon staler"))
        self.assertTrue(is_anagram("Anagram", "Nag a ram!"))

    def test_not_anagram(self):
        self.assertFalse(is_anagram("Hello", "World"))

    def test_empty_strings(self):
        self.assertTrue(is_anagram("", ""))

    def test_identical_words(self):
        self.assertTrue(is_anagram("Same", "Same"))

if __name__ == '__main__':
    unittest.main()
```

## Output:

```

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Docum
ts.py"
....
-----
Ran 4 tests in 0.001s

OK
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>

```

## Pytest testing

### Prompt:

write test cases for is\_anagram function using pytest framework with the same requirements as above

### Code:

```

#write test cases for is_anagram function using pytest framework with the same requirements as above
import re
def is_anagram(str1, str2):
    # Remove space and punctuation, and convert to lowercase
    str1_cleaned = re.sub(r'[\s\W]', '', str1).lower()
    str2_cleaned = re.sub(r'[\s\W]', '', str2).lower()

    # Check if the sorted characters of both strings are the same
    return sorted(str1_cleaned) == sorted(str2_cleaned)
def test_anagram():
    assert is_anagram("listen", "silent") == True
    assert is_anagram("Triangle", "Integral") == True
    assert is_anagram("Dormitory", "Dirty Room") == True
    assert is_anagram("The eyes", "They see") == True
    assert is_anagram("Astronomer", "Moon starrer") == True
    assert is_anagram("Anagram", "Nag a ram!") == True
def test_not_anagram():
    assert is_anagram("Hello", "World") == False
def test_empty_strings():
    assert is_anagram("", "") == True
def test_identical_words():
    assert is_anagram("Same", "Same") == True

```

### Output:

```

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> python -m pytest tests.py
===== test session starts =====
platform win32 -- Python 3.12.10, pytest-9.0.1, pluggy-1.6.0
rootdir: C:\Users\Pavani\OneDrive\Documents\AI Assisted coding
plugins: dash-3.2.0
collected 4 items

tests.py ....

===== 4 passed in 0.08s =====
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>

```

### Explanation:

- All test types validate the `is_anagram` function by checking true anagrams, non-anagrams, and edge cases like empty strings and identical words.
- General/manual tests help visualize results for multiple string combinations but require manual checking of outputs.
- Assert tests provide quick automated verification and immediately raise errors if any anagram condition fails.
- `unittest` offers a structured and reusable testing approach with grouped methods for anagram, non-anagram, and edge case validation.
- `pytest` gives a cleaner, concise testing style and is more flexible for scalable automated testing of string normalization rules (case, spaces, punctuation).

#### **Task Description #4 (Inventory Class – Apply AI to Simulate Real- World Inventory System)**

##### **General test cases:**

generate 5 general test cases for an inventory class with stock management it should include methods like `add_item(name,quantity)`,`remove_item(name,quantity)`,`get_stock(name)`

##### **Code:**

```

#generate 5 general test cases for an inventory class with stock management it should include methods
#like add_item(name,quantity),remove_item(name,quantity),get_stock(name)
class Inventory:
    def __init__(self):
        self.stock = {}

    def add_item(self, name, quantity):
        if name in self.stock:
            self.stock[name] += quantity
        else:
            self.stock[name] = quantity

    def remove_item(self, name, quantity):
        if name in self.stock and self.stock[name] >= quantity:
            self.stock[name] -= quantity
            if self.stock[name] == 0:
                del self.stock[name]
        else:
            raise ValueError("Not enough stock to remove")

    def get_stock(self, name):
        return self.stock.get(name, 0)

# Test cases for Inventory class
inventory = Inventory()
inventory.add_item("apple", 10)
inventory.add_item("banana", 5)
print(inventory.get_stock("apple")) # Expected: 10
print(inventory.get_stock("banana")) # Expected: 5
inventory.remove_item("apple", 3)
print(inventory.get_stock("apple")) # Expected: 7
inventory.remove_item("banana", 5)
print(inventory.get_stock("banana")) # Expected: 0
try:
    inventory.remove_item("apple", 10) # This should raise an error
except ValueError as e:
    print(e) # Expected: Not enough stock to remove

```

## Output:

| PROBLEMS   | OUTPUT                     | DEBUG CONSOLE | TERMINAL | PORTS | POSTMAN CONSOLE |
|--|----------------------------|---------------|----------|-------|-----------------|
| ● PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Micr | 10                         |               |          |       |                 |
|  | 5                          |               |          |       |                 |
|  | 7                          |               |          |       |                 |
|  | 0                          |               |          |       |                 |
|  | Not enough stock to remove |               |          |       |                 |
| ○ PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>                                      |                            |               |          |       |                 |

## Assert test cases:

### Prompt:

Generate assert based test cases for the inventory class

### Code:

```

tests.py > ...
363
364 #generate assertbased test cases for the Inventory class
365 class Inventory:
366     def __init__(self):
367         self.stock = {}
368
369     def add_item(self, name, quantity):
370         if name in self.stock:
371             self.stock[name] += quantity
372         else:
373             self.stock[name] = quantity
374
375     def remove_item(self, name, quantity):
376         if name in self.stock and self.stock[name] >= quantity:
377             self.stock[name] -= quantity
378             if self.stock[name] == 0:
379                 del self.stock[name]
380         else:
381             raise ValueError("Not enough stock to remove")
382
383     def get_stock(self, name):
384         return self.stock.get(name, 0)
385
386 # Assert test cases for Inventory class
387 inventory = Inventory()
388 inventory.add_item("apple", 10)
389 inventory.add_item("banana", 5)
390 assert inventory.get_stock("apple") == 10, "Test case 1 failed"
391 assert inventory.get_stock("banana") == 5, "Test case 2 failed"
392 inventory.remove_item("apple", 3)
393 assert inventory.get_stock("apple") == 7, "Test case 3 failed"
394 inventory.remove_item("banana", 5)
395 assert inventory.get_stock("banana") == 0, "Test case 4 failed"
396 try:
397     inventory.remove_item("apple", 10) # This should raise an error
398 except ValueError as e:
399     assert str(e) == "Not enough stock to remove", "Test case 5 failed"
400 print("All test cases passed!")

```

## Output:

```

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Micros
All test cases passed!
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>

```

## Unit testing

### Prompt:

generate unit testing test cases for the Inventory class

### Code

```

0
1 #generate unit testing test cases for the Inventory class
2 import unittest
3 class Inventory:
4     def __init__(self):
5         self.stock = {}
6
7     def add_item(self, name, quantity):
8         if name in self.stock:
9             self.stock[name] += quantity
10        else:
11            self.stock[name] = quantity
12
13    def remove_item(self, name, quantity):
14        if name in self.stock and self.stock[name] >= quantity:
15            self.stock[name] -= quantity
16            if self.stock[name] == 0:
17                del self.stock[name]
18        else:
19            raise ValueError("Not enough stock to remove")
20
21    def get_stock(self, name):
22        return self.stock.get(name, 0)
23
24 class TestInventory(unittest.TestCase):
25     def setUp(self):
26         self.inventory = Inventory()
27
28     def test_add_item(self):
29         self.inventory.add_item("apple", 10)
30         self.assertEqual(self.inventory.get_stock("apple"), 10)
31
32     def test_remove_item(self):
33         self.inventory.add_item("banana", 5)
34         self.inventory.remove_item("banana", 3)
35         self.assertEqual(self.inventory.get_stock("banana"), 2)
36
37     def test_remove_item_not_enough_stock(self):
38         self.inventory.add_item("orange", 4)
39         with self.assertRaises(ValueError):
40             self.inventory.remove_item("orange", 5)
41
42     def test_get_stock_nonexistent_item(self):
43         self.assertEqual(self.inventory.get_stock("grape"), 0)
44
45     def test_remove_item_completely(self):
46         self.inventory.add_item("melon", 2)
47         self.inventory.remove_item("melon", 2)
48         self.assertEqual(self.inventory.get_stock("melon"), 0)
49
50 if __name__ == '__main__':
51     unittest.main()
52

```

## Output:

```

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> ^C
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/tests.py"
.....
Ran 5 tests in 0.002s
OK
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>

```

## Pytest testing

**Prompt:** generate test cases for the Inventory class using pytest framework

**Code:**

```

19
20 #generate test cases for the Inventory class using pytest framework
21 class Inventory:
22     def __init__(self):
23         self.stock = {}
24
25     def add_item(self, name, quantity):
26         if name in self.stock:
27             self.stock[name] += quantity
28         else:
29             self.stock[name] = quantity
30
31     def remove_item(self, name, quantity):
32         if name in self.stock and self.stock[name] >= quantity:
33             self.stock[name] -= quantity
34             if self.stock[name] == 0:
35                 del self.stock[name]
36         else:
37             raise ValueError("Not enough stock to remove")
38
39     def get_stock(self, name):
40         return self.stock.get(name, 0)
41
42 def test_add_item():
43     inventory = Inventory()
44     inventory.add_item("apple", 10)
45     assert inventory.get_stock("apple") == 10
46
47 def test_remove_item():
48     inventory = Inventory()
49     inventory.add_item("banana", 5)
50     inventory.remove_item("banana", 3)
51     assert inventory.get_stock("banana") == 2
52
53 def test_remove_item_not_enough_stock():
54     inventory = Inventory()
55     inventory.add_item("orange", 4)
56     try:
57         inventory.remove_item("orange", 5)
58     except ValueError as e:
59         assert str(e) == "Not enough stock to remove"
60
61 def test_get_stock_nonexistent_item():
62     inventory = Inventory()
63     assert inventory.get_stock("grape") == 0
64
65 def test_remove_item_completely():
66     inventory = Inventory()
67     inventory.add_item("melon", 2)
68     inventory.remove_item("melon", 2)
69     assert inventory.get_stock("melon") == 0
70
71

```

## Output:

```

C:\Users\pavan\OneDrive\Documents\AI Assisted coding> python -m pytest tests.py
platform win32 -- Python 3.12.10, pytest-9.0.1, pluggy-1.6.0
===== test session starts =====
platform win32 -- Python 3.12.10, pytest-9.0.1, pluggy-1.6.0
platform win32 -- Python 3.12.10, pytest-9.0.1, pluggy-1.6.0
rootdir: C:\Users\pavan\OneDrive\Documents\AI Assisted coding
plugins: dash-3.2.0
collected 5 items

tests.py .....
===== 5 passed in 0.05s =====
PS C:\Users\pavan\OneDrive\Documents\AI Assisted coding>

```

## Explanation:

- All testing types validate the core inventory operations — adding items, removing items, checking stock levels, and handling stock updates correctly.
- General/manual tests demonstrate real workflow execution but rely on printed outputs and manual verification.
- Assert tests provide quick automated checks for stock values and exception handling during removal.



- unittest ensures structured validation with setup reuse, method-wise testing, and error condition verification.
- pytest offers a concise and flexible approach to test stock management logic, edge cases, and exception scenarios efficiently.

## Task Description #5 (Date Validation & Formatting – Apply AI for Data Validation)

### General test cases

**Prompt:** generate 5 general test cases for `validate_and_format_date(date_str)` to check and convert dates the requirements are the function should accept date in "DD/MM/YYYY" format and convert it to "YYYY-MM-DD" format and handle invalid date formats

### Code:

```
#generate 5 general test cases for validate_and_format_date(date_str) to check and convert dates
#the requirements are the function should accept date in "DD/MM/YYYY" format and convert it to "YYYY-MM-DD" format and handle invalid date formats
class DateFormatter:
    @staticmethod
    def validate_and_format_date(date_str):
        try:
            day, month, year = map(int, date_str.split('/'))
            if 1 <= day <= 31 and 1 <= month <= 12 and year > 0:
                return f"{year:04d}-{month:02d}-{day:02d}"
            else:
                raise ValueError("Invalid date")
        except (ValueError, IndexError):
            raise ValueError("Invalid date format")
# Test cases for validate_and_format_date function
test_cases = [
    ("15/08/2021", "2021-08-15"),
    ("01/01/2000", "2000-01-01"),
    ("31/12/1999", "1999-12-31"),
    ("30/02/2020", "Invalid date"), # Invalid date
    ("2021/08/15", "Invalid date format"), # Invalid format
]
for date_str, expected in test_cases:
    try:
        result = DateFormatter.validate_and_format_date(date_str)
        print(f"Input: '{date_str}' | Expected: '{expected}' | Result: '{result}' | Test Passed: {result == expected}")
    except ValueError as e:
        print(f"Input: '{date_str}' | Expected: '{expected}' | Result: '{str(e)}' | Test Passed: {str(e) == expected}")
```

### Output:

```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:\Users\Pavani\AppData\Local\Microsoft\WindowsApps\python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/tests.py"
Input: '15/08/2021' | Expected: '2021-08-15' | Result: '2021-08-15' | Test Passed: True
Input: '01/01/2000' | Expected: '2000-01-01' | Result: '2000-01-01' | Test Passed: True
Input: '31/12/1999' | Expected: '1999-12-31' | Result: '1999-12-31' | Test Passed: True
Input: '30/02/2020' | Expected: 'Invalid date' | Result: '2020-02-30' | Test Passed: False
Input: '2021/08/15' | Expected: 'Invalid date format' | Result: 'Invalid date format' | Test Passed: True
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> []
```

### Assert based

### Prompt:

generate assert test cases for `validate_and_format_date` function

### Code:

```

#generate assert test cases for validate_and_format_date function
class DateFormatter:
    @staticmethod
    def validate_and_format_date(date_str):
        try:
            day, month, year = map(int, date_str.split('/'))
            if 1 <= day <= 31 and 1 <= month <= 12 and year > 0:
                return f"{year:04d}-{month:02d}-{day:02d}"
            else:
                raise ValueError("Invalid date")
        except (ValueError, IndexError):
            raise ValueError("Invalid date format")
# Assert test cases for validate_and_format_date function
try:
    assert DateFormatter.validate_and_format_date("15/08/2021") == "2021-08-15", "Test case 1 failed"
    assert DateFormatter.validate_and_format_date("01/01/2000") == "2000-01-01", "Test case 2 failed"
    assert DateFormatter.validate_and_format_date("31/12/1999") == "1999-12-31", "Test case 3 failed"
    try:
        DateFormatter.validate_and_format_date("30/02/2020")
    except ValueError as e:
        assert str(e) == "Invalid date", "Test case 4 failed"
    try:
        DateFormatter.validate_and_format_date("2021/08/15")
    except ValueError as e:
        assert str(e) == "Invalid date format", "Test case 5 failed"
    print("All test cases passed!")
except AssertionError as e:
    print(str(e))

```

## Output:

```

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> ^C
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI
All test cases passed! ...
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> 

```

## Unit testing:

### Prompt:

generate unit testing test cases for validate\_and\_format\_date function

### Code:

```

#generate unit testing test cases for validate_and_format_date function
import unittest
class DateFormatter:
    @staticmethod
    def validate_and_format_date(date_str):
        try:
            day, month, year = map(int, date_str.split('/'))
            if 1 <= day <= 31 and 1 <= month <= 12 and year > 0:
                return f"{year:04d}-{month:02d}-{day:02d}"
            else:
                raise ValueError("Invalid date")
        except (ValueError, IndexError):
            raise ValueError("Invalid date format")
class TestDateFormatter(unittest.TestCase):
    def test_valid_date(self):
        self.assertEqual(DateFormatter.validate_and_format_date("15/08/2021"), "2021-08-15")
        self.assertEqual(DateFormatter.validate_and_format_date("01/01/2000"), "2000-01-01")
        self.assertEqual(DateFormatter.validate_and_format_date("31/12/1999"), "1999-12-31")

    def test_invalid_date(self):
        with self.assertRaises(ValueError) as context:
            DateFormatter.validate_and_format_date("30/02/2020")
        self.assertEqual(str(context.exception), "Invalid date")

    def test_invalid_format(self):
        with self.assertRaises(ValueError) as context:
            DateFormatter.validate_and_format_date("2021/08/15")
        self.assertEqual(str(context.exception), "Invalid date format")
if __name__ == '__main__':
    unittest.main()

```

## Output:

```

PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> ^
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/tests.py"
F...
FAIL: test_invalid_date (__main__.TestDateFormatter.test_invalid_date)
Traceback (most recent call last):
  File "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/tests.py", line 574, in test_invalid_date
    with self.assertRaises(ValueError) as context:
AssertionError: ValueError not raised

Ran 3 tests in 0.001s

FAILED (failures=1)
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>

```

## Pytest :

## Prompt:

generate test cases for validate\_and\_format\_date function using pytest framework

## Code:

```
#generate test cases for validate_and_format_date function using pytest framework
class DateFormatter:
    @staticmethod
    def validate_and_format_date(date_str):
        try:
            day, month, year = map(int, date_str.split('/'))
            if 1 <= day <= 31 and 1 <= month <= 12 and year > 0:
                return f"{year:04d}-{month:02d}-{day:02d}"
            else:
                raise ValueError("Invalid date")
        except (ValueError, IndexError):
            raise ValueError("Invalid date format")

def test_valid_date():
    assert DateFormatter.validate_and_format_date("15/08/2021") == "2021-08-15"
    assert DateFormatter.validate_and_format_date("01/01/2000") == "2000-01-01"
    assert DateFormatter.validate_and_format_date("31/12/1999") == "1999-12-31"

def test_invalid_date():
    try:
        DateFormatter.validate_and_format_date("30/02/2020")
    except ValueError as e:
        assert str(e) == "Invalid date"

def test_invalid_format():
    try:
        DateFormatter.validate_and_format_date("2021/08/15")
    except ValueError as e:
        assert str(e) == "Invalid date format"
```

## Output:

```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> python -m pytest tests.py
===== test session starts =====
platform win32 -- Python 3.12.10, pytest-9.0.1, pluggy-1.6.0
rootdir: C:\Users\Pavani\OneDrive\Documents\AI Assisted coding
plugins: dash-3.2.0
collected 3 items

tests.py ...

===== 3 passed in 0.05s =====
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

## Explanation:

- All testing types verify date validation and correct conversion from “DD/MM/YYYY” to “YYYY-MM-DD” format.
- General/manual tests help observe formatted outputs and error messages but depend on manual result checking.
- Assert tests quickly confirm valid conversions and ensure proper exceptions for invalid dates and formats.
- unittest provides structured validation with dedicated methods for valid dates, invalid dates, and format errors.
- pytest offers a concise and flexible way to test date parsing, formatting accuracy, and exception handling.