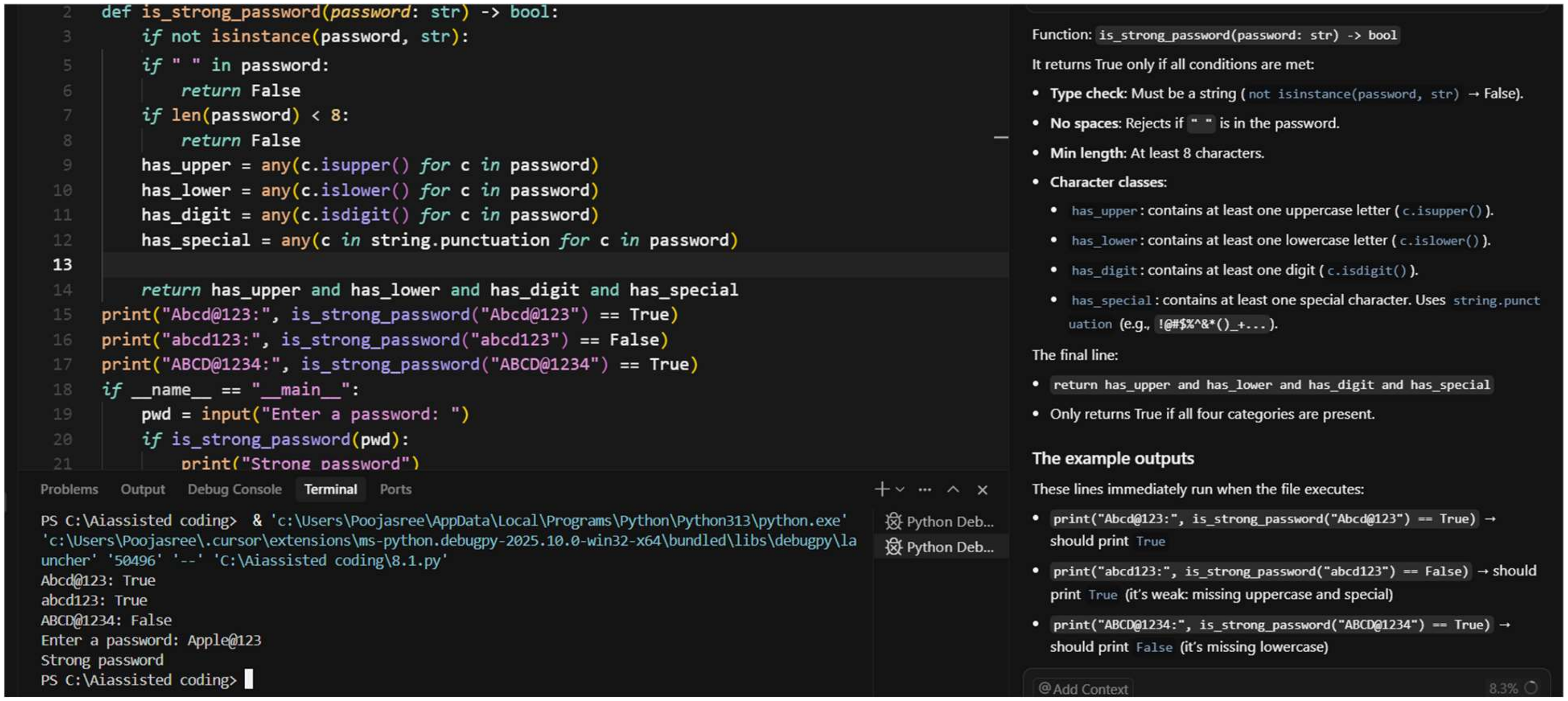
Ai assisted coding
Lab Assignment -8

Task-1:

Prompt: write a python code to generate at least 3 assert test cases for is_strong_password(password) and implement the validator function.



```python
def is_strong_password(password: str) -> bool:
    if not isinstance(password, str):
    if " " in password:
        return False
    if len(password) < 8:
        return False
    has_upper = any(c.isupper() for c in password)
    has_lower = any(c.islower() for c in password)
    has_digit = any(c.isdigit() for c in password)
    has_special = any(c in string.punctuation for c in password)

    return has_upper and has_lower and has_digit and has_special
print("Abcd@123:", is_strong_password("Abcd@123") == True)
print("abcd123:", is_strong_password("abcd123") == False)
print("ABCD@1234:", is_strong_password("ABCD@1234") == True)
if __name__ == "__main__":
    pwd = input("Enter a password: ")
    if is_strong_password(pwd):
        print("Strong password")
```

```
PS C:\Aiassisted coding>  & 'c:\Users\Poojasree\AppData\Local\Programs\Python\Python313\python.exe'
'c:\Users\Poojasree\.cursor\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\la
uncher' '50496' '--' 'C:\Aiassisted coding\8.1.py'
Abcd@123: True
abcd123: True
ABCD@1234: False
Enter a password: Apple@123
Strong password
PS C:\Aiassisted coding>
```

Function: `is_strong_password(password: str) -> bool`

It returns True only if all conditions are met:

- **Type check**: Must be a string (`not isinstance(password, str)` → False).
- **No spaces**: Rejects if `" "` is in the password.
- **Min length**: At least 8 characters.
- **Character classes**:
  - `has_upper` : contains at least one uppercase letter (`c.isupper()`).
  - `has_lower` : contains at least one lowercase letter (`c.islower()`).
  - `has_digit` : contains at least one digit (`c.isdigit()`).
  - `has_special` : contains at least one special character. Uses `string.punctuation` (e.g., `!@#$%^&*()_+...`).

The final line:

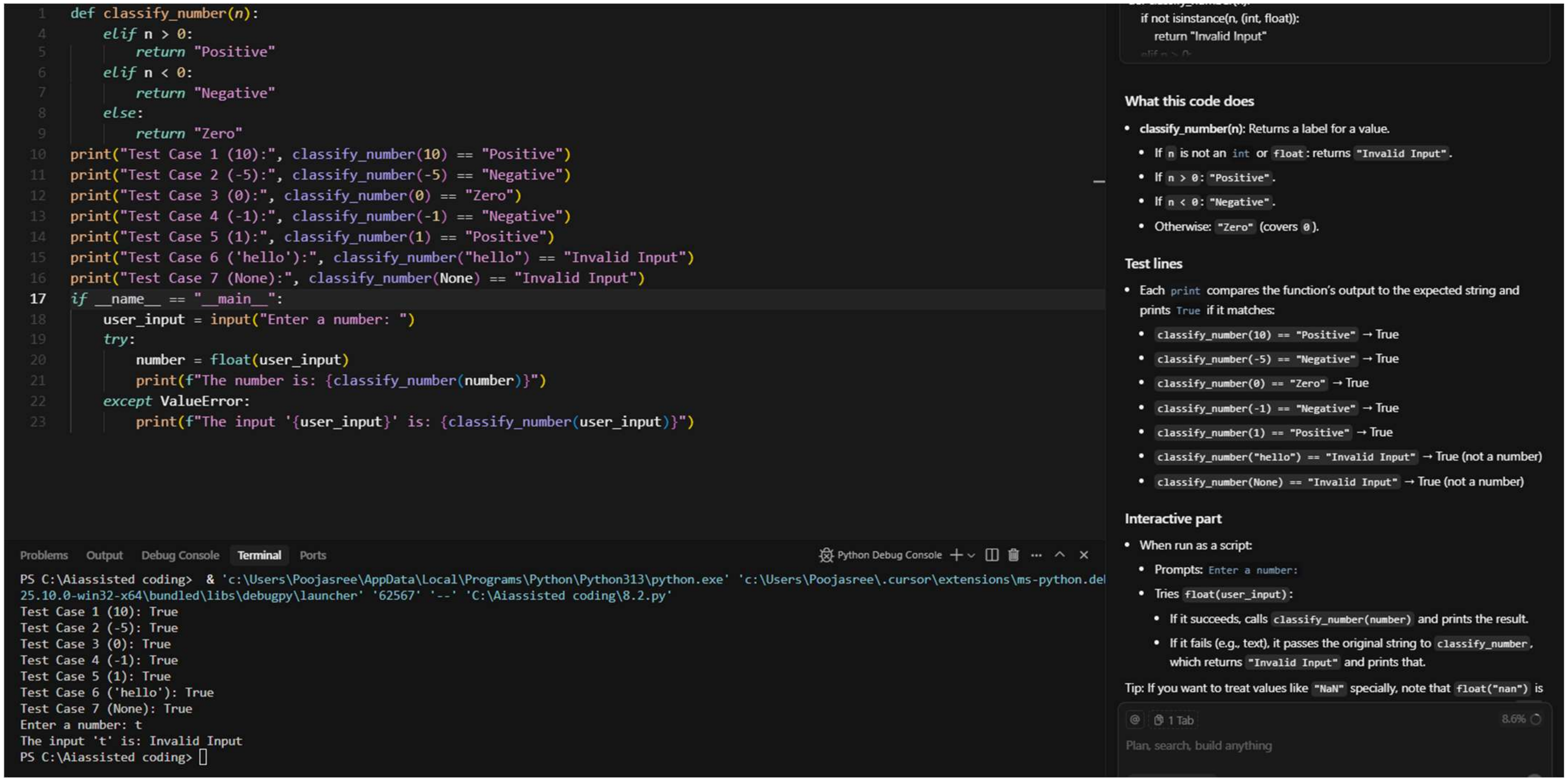
- `return has_upper and has_lower and has_digit and has_special`
- Only returns True if all four categories are present.

The example outputs

These lines immediately run when the file executes:

- `print("Abcd@123:", is_strong_password("Abcd@123") == True)` → should print `True`
- `print("abcd123:", is_strong_password("abcd123") == False)` → should print `True` (it's weak: missing uppercase and special)
- `print("ABCD@1234:", is_strong_password("ABCD@1234") == True)` → should print `False` (it's missing lowercase)

Task_2:
Prompt: Write a python code to generate at least 3 assert test cases for a classify_number(n) function. Implement using loops.



```python
def classify_number(n):
    elif n > 0:
        return "Positive"
    elif n < 0:
        return "Negative"
    else:
        return "Zero"
print("Test Case 1 (10):", classify_number(10) == "Positive")
print("Test Case 2 (-5):", classify_number(-5) == "Negative")
print("Test Case 3 (0):", classify_number(0) == "Zero")
print("Test Case 4 (-1):", classify_number(-1) == "Negative")
print("Test Case 5 (1):", classify_number(1) == "Positive")
print("Test Case 6 ('hello'):", classify_number("hello") == "Invalid Input")
print("Test Case 7 (None):", classify_number(None) == "Invalid Input")
if __name__ == "__main__":
    user_input = input("Enter a number: ")
    try:
        number = float(user_input)
        print(f"The number is: {classify_number(number)}")
    except ValueError:
        print(f"The input '{user_input}' is: {classify_number(user_input)}")
```

```
PS C:\Aiassisted coding>  & 'c:\Users\Poojasree\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\Poojasree\.cursor\extensions\ms-python.del
25.10.0-win32-x64\bundled\libs\debugpy\launcher' '62567' '--' 'C:\Aiassisted coding\8.2.py'
Test Case 1 (10): True
Test Case 2 (-5): True
Test Case 3 (0): True
Test Case 4 (-1): True
Test Case 5 (1): True
Test Case 6 ('hello'): True
Test Case 7 (None): True
Enter a number: t
The input 't' is: Invalid Input
PS C:\Aiassisted coding> 
```

```python
    if not isinstance(n, (int, float)):
        return "Invalid Input"
```

What this code does

- **classify_number(n):** Returns a label for a value.
  - If `n` is not an `int` or `float`: returns `"Invalid Input"`.
  - If `n > 0`: `"Positive"`.
  - If `n < 0`: `"Negative"`.
  - Otherwise: `"Zero"` (covers `0`).

Test lines

- Each `print` compares the function's output to the expected string and prints `True` if it matches:
  - `classify_number(10) == "Positive"` → True
  - `classify_number(-5) == "Negative"` → True
  - `classify_number(0) == "Zero"` → True
  - `classify_number(-1) == "Negative"` → True
  - `classify_number(1) == "Positive"` → True
  - `classify_number("hello") == "Invalid Input"` → True (not a number)
  - `classify_number(None) == "Invalid Input"` → True (not a number)

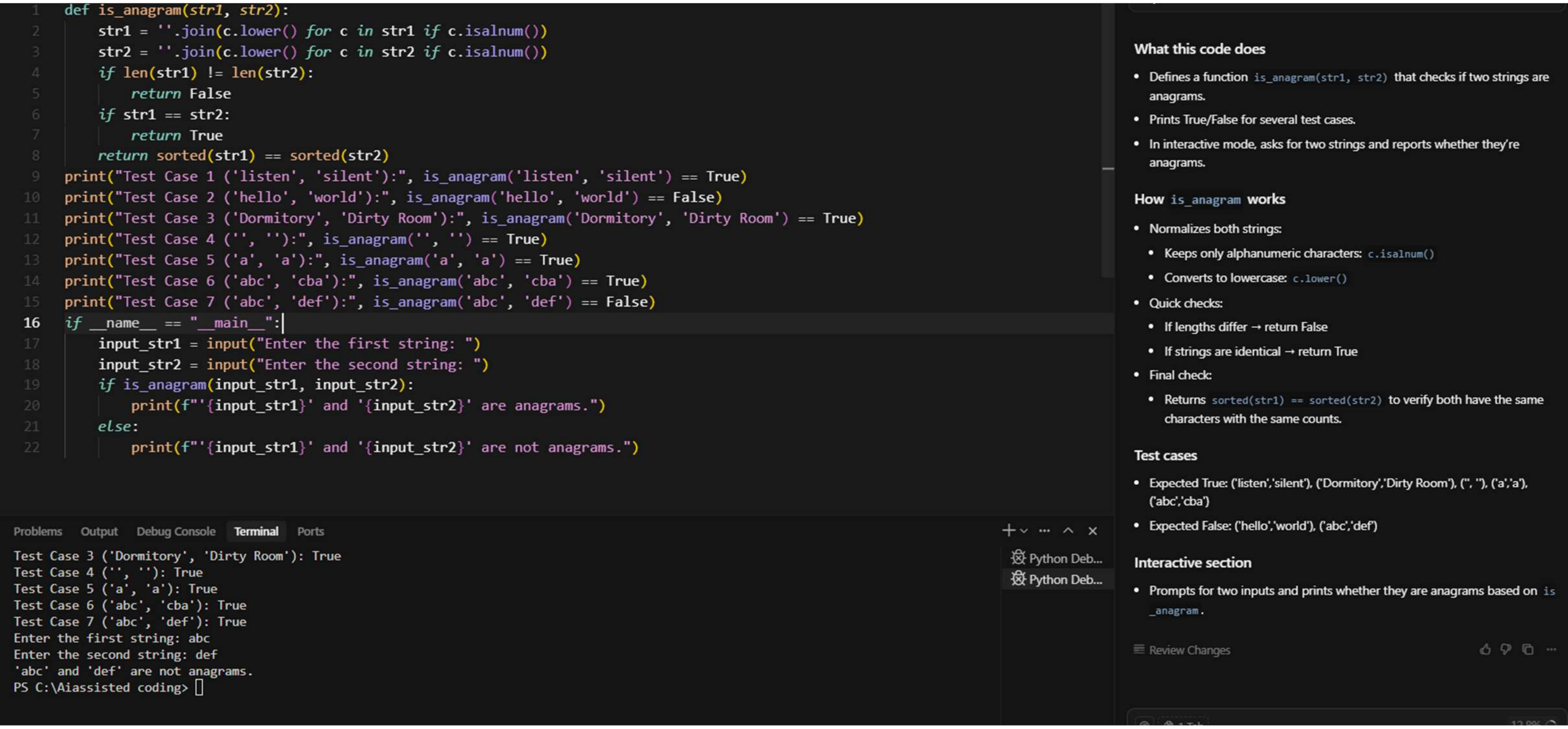
Interactive part

- When run as a script:
  - Prompts: `Enter a number:`
  - Tries `float(user_input)`:
    - If it succeeds, calls `classify_number(number)` and prints the result.
    - If it fails (e.g., text), it passes the original string to `classify_number`, which returns `"Invalid Input"` and prints that.

Tip: If you want to treat values like `"NaN"` specially, note that `float("nan")` is

Task-3:

Prompt: write a python code generate at least 3 assert test cases for is_anagram(str1, str2) and implement the function.



```python
def is_anagram(str1, str2):
    str1 = ''.join(c.lower() for c in str1 if c.isalnum())
    str2 = ''.join(c.lower() for c in str2 if c.isalnum())
    if len(str1) != len(str2):
        return False
    if str1 == str2:
        return True
    return sorted(str1) == sorted(str2)
print("Test Case 1 ('listen', 'silent'):", is_anagram('listen', 'silent') == True)
print("Test Case 2 ('hello', 'world'):", is_anagram('hello', 'world') == False)
print("Test Case 3 ('Dormitory', 'Dirty Room'):", is_anagram('Dormitory', 'Dirty Room') == True)
print("Test Case 4 ('', ''):", is_anagram('', '') == True)
print("Test Case 5 ('a', 'a'):", is_anagram('a', 'a') == True)
print("Test Case 6 ('abc', 'cba'):", is_anagram('abc', 'cba') == True)
print("Test Case 7 ('abc', 'def'):", is_anagram('abc', 'def') == False)
if __name__ == "__main__":
    input_str1 = input("Enter the first string: ")
    input_str2 = input("Enter the second string: ")
    if is_anagram(input_str1, input_str2):
        print(f"'{input_str1}' and '{input_str2}' are anagrams.")
    else:
        print(f"'{input_str1}' and '{input_str2}' are not anagrams.")
```

```
Test Case 3 ('Dormitory', 'Dirty Room'): True
Test Case 4 ('', ''): True
Test Case 5 ('a', 'a'): True
Test Case 6 ('abc', 'cba'): True
Test Case 7 ('abc', 'def'): True
Enter the first string: abc
Enter the second string: def
'abc' and 'def' are not anagrams.
PS C:\Aiassisted coding> 
```

What this code does

- Defines a function `is_anagram(str1, str2)` that checks if two strings are anagrams.
- Prints True/False for several test cases.
- In interactive mode, asks for two strings and reports whether they're anagrams.

How `is_anagram` works

- Normalizes both strings:
  - Keeps only alphanumeric characters: `c.isalnum()`
  - Converts to lowercase: `c.lower()`
- Quick checks:
  - If lengths differ → return False
  - If strings are identical → return True
- Final check:
  - Returns `sorted(str1) == sorted(str2)` to verify both have the same characters with the same counts.

Test cases

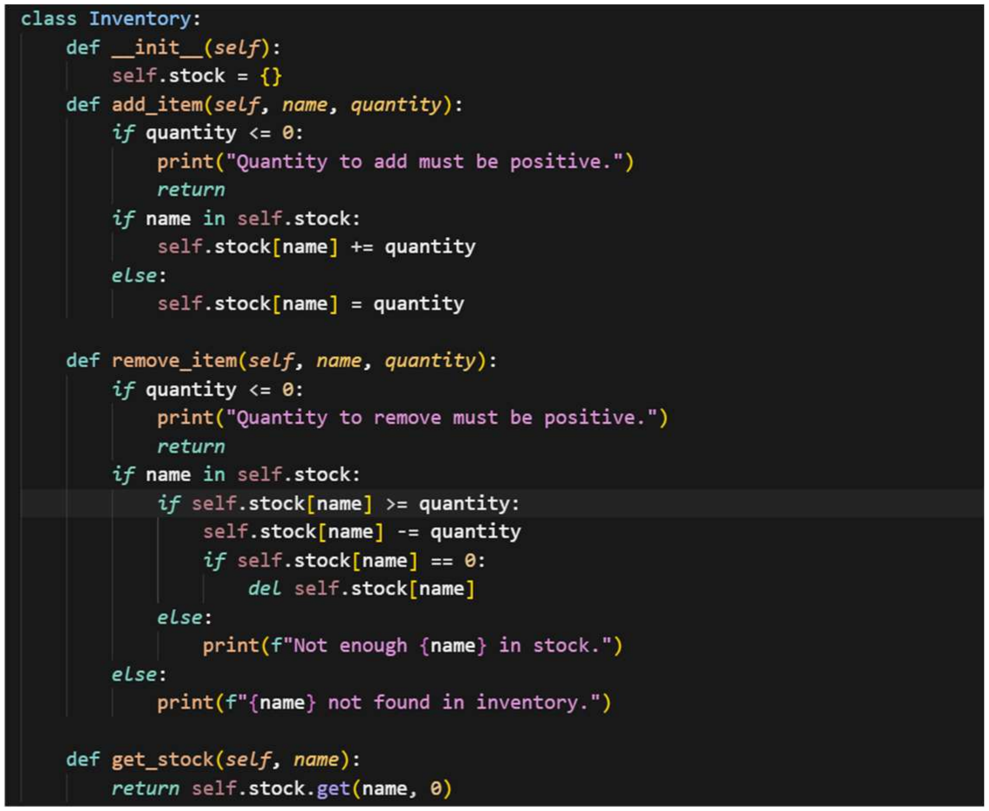
- Expected True: ('listen','silent'), ('Dormitory','Dirty Room'), ('', ''), ('a','a'), ('abc','cba')
- Expected False: ('hello','world'), ('abc','def')

Interactive section

- Prompts for two inputs and prints whether they are anagrams based on `is_anagram`.

Task-4:

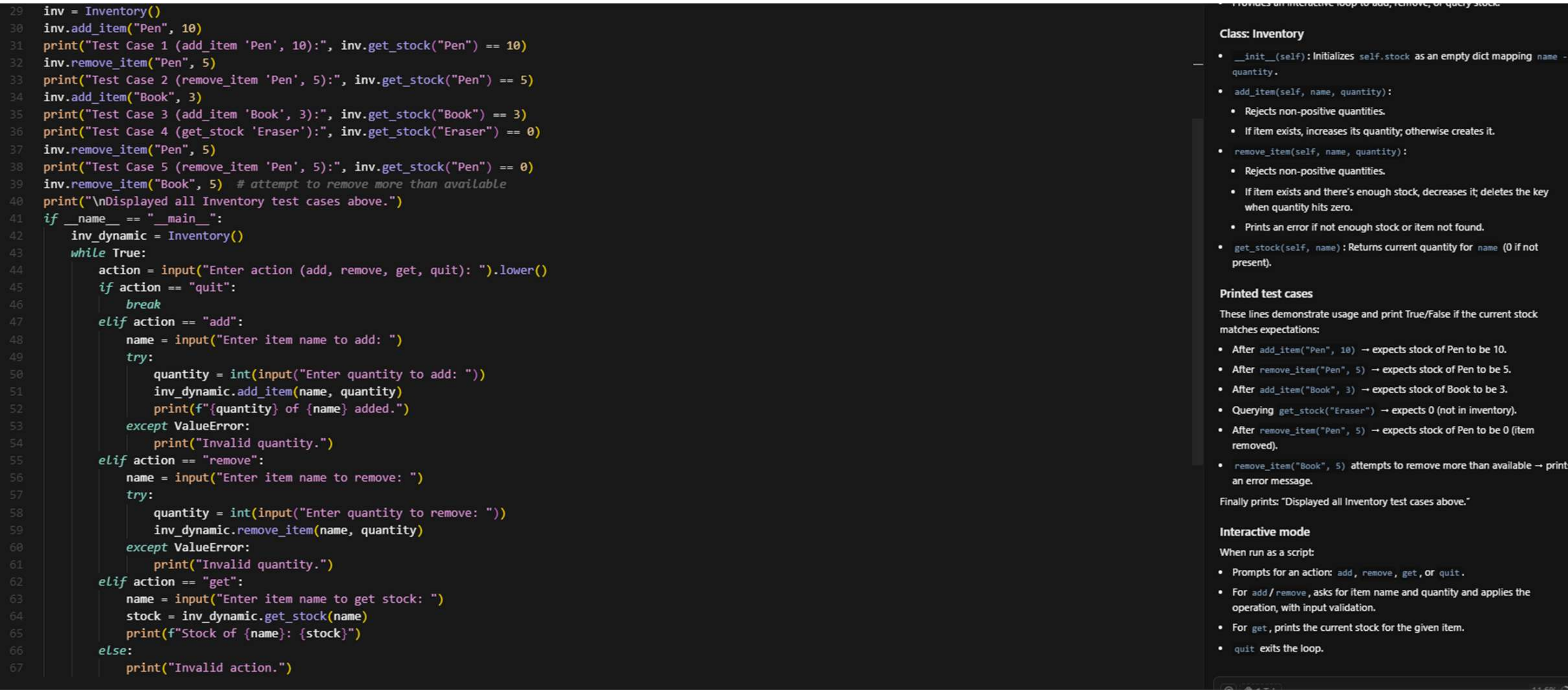
Prompt: write a python code to generate at least 3 assert-based tests for an Inventory class with stock management.



```python
class Inventory:
    def __init__(self):
        self.stock = {}
    def add_item(self, name, quantity):
        if quantity <= 0:
            print("Quantity to add must be positive.")
            return
        if name in self.stock:
            self.stock[name] += quantity
        else:
            self.stock[name] = quantity

    def remove_item(self, name, quantity):
        if quantity <= 0:
            print("Quantity to remove must be positive.")
            return
        if name in self.stock:
            if self.stock[name] >= quantity:
                self.stock[name] -= quantity
                if self.stock[name] == 0:
                    del self.stock[name]
            else:
                print(f"Not enough {name} in stock.")
        else:
            print(f"{name} not found in inventory.")

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



```python
inv = Inventory()
inv.add_item("Pen", 10)
print("Test Case 1 (add_item 'Pen', 10):", inv.get_stock("Pen") == 10)
inv.remove_item("Pen", 5)
print("Test Case 2 (remove_item 'Pen', 5):", inv.get_stock("Pen") == 5)
inv.add_item("Book", 3)
print("Test Case 3 (add_item 'Book', 3):", inv.get_stock("Book") == 3)
print("Test Case 4 (get_stock 'Eraser'):", inv.get_stock("Eraser") == 0)
inv.remove_item("Pen", 5)
print("Test Case 5 (remove_item 'Pen', 5):", inv.get_stock("Pen") == 0)
inv.remove_item("Book", 5)  # attempt to remove more than available
print("\nDisplayed all Inventory test cases above.")
if __name__ == "__main__":
    inv_dynamic = Inventory()
    while True:
        action = input("Enter action (add, remove, get, quit): ").lower()
        if action == "quit":
            break
        elif action == "add":
            name = input("Enter item name to add: ")
            try:
                quantity = int(input("Enter quantity to add: "))
                inv_dynamic.add_item(name, quantity)
                print(f"{quantity} of {name} added.")
            except ValueError:
                print("Invalid quantity.")
        elif action == "remove":
            name = input("Enter item name to remove: ")
            try:
                quantity = int(input("Enter quantity to remove: "))
                inv_dynamic.remove_item(name, quantity)
            except ValueError:
                print("Invalid quantity.")
        elif action == "get":
            name = input("Enter item name to get stock: ")
            stock = inv_dynamic.get_stock(name)
            print(f"Stock of {name}: {stock}")
        else:
            print("Invalid action.")
```

Class: Inventory

- `__init__(self)`: Initializes `self.stock` as an empty dict mapping `name -` `quantity`.
- `add_item(self, name, quantity)`:
  - Rejects non-positive quantities.
  - If item exists, increases its quantity; otherwise creates it.
- `remove_item(self, name, quantity)`:
  - Rejects non-positive quantities.
  - If item exists and there's enough stock, decreases it; deletes the key when quantity hits zero.
  - Prints an error if not enough stock or item not found.
- `get_stock(self, name)`: Returns current quantity for `name` (0 if not present).

Printed test cases

These lines demonstrate usage and print True/False if the current stock matches expectations:

- After `add_item("Pen", 10)` → expects stock of Pen to be 10.
- After `remove_item("Pen", 5)` → expects stock of Pen to be 5.
- After `add_item("Book", 3)` → expects stock of Book to be 3.
- Querying `get_stock("Eraser")` → expects 0 (not in inventory).
- After `remove_item("Pen", 5)` → expects stock of Pen to be 0 (item removed).
- `remove_item("Book", 5)` attempts to remove more than available → print an error message.

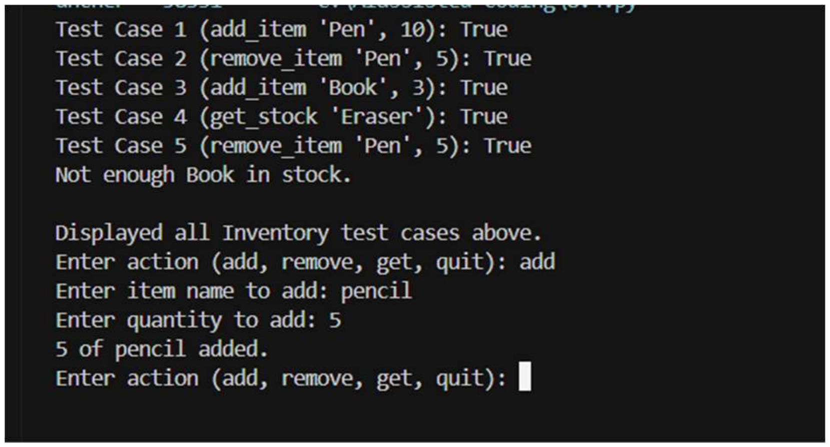
Finally prints: "Displayed all Inventory test cases above."

Interactive mode

When run as a script:

- Prompts for an action: `add`, `remove`, `get`, or `quit`.
- For `add` / `remove`, asks for item name and quantity and applies the operation, with input validation.
- For `get`, prints the current stock for the given item.
- `quit` exits the loop.

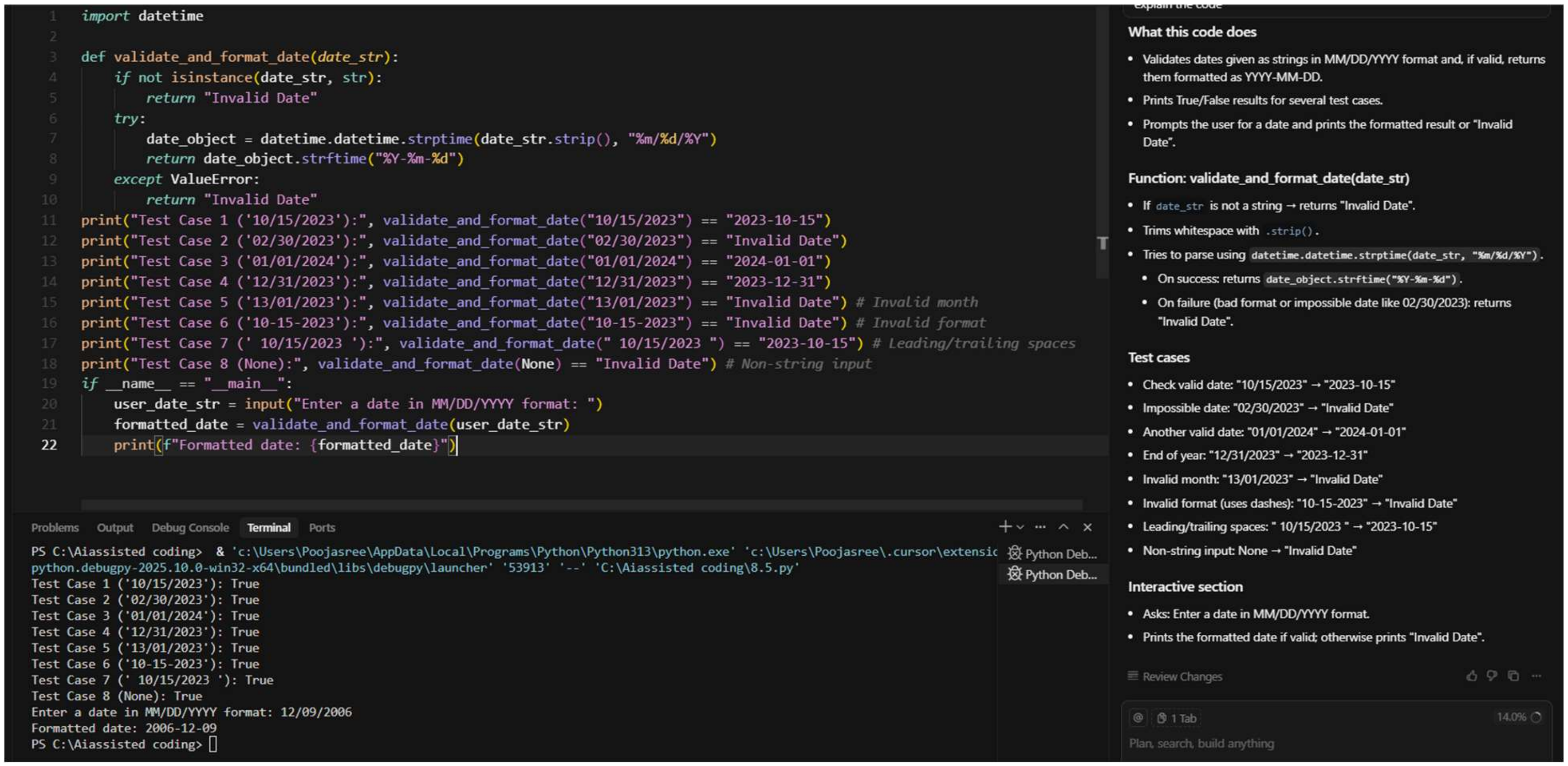
Output:



```
Test Case 1 (add_item 'Pen', 10): True
Test Case 2 (remove_item 'Pen', 5): True
Test Case 3 (add_item 'Book', 3): True
Test Case 4 (get_stock 'Eraser'): True
Test Case 5 (remove_item 'Pen', 5): True
Not enough Book in stock.

Displayed all Inventory test cases above.
Enter action (add, remove, get, quit): add
Enter item name to add: pencil
Enter quantity to add: 5
5 of pencil added.
Enter action (add, remove, get, quit):
```

Task-5:
Prompt: write a python code to generate at least 3 assert test cases for validate_and_format_date(date_str) to check and convert dates.



```python
import datetime

def validate_and_format_date(date_str):
    if not isinstance(date_str, str):
        return "Invalid Date"
    try:
        date_object = datetime.datetime.strptime(date_str.strip(), "%m/%d/%Y")
        return date_object.strftime("%Y-%m-%d")
    except ValueError:
        return "Invalid Date"
print("Test Case 1 ('10/15/2023'):", validate_and_format_date("10/15/2023") == "2023-10-15")
print("Test Case 2 ('02/30/2023'):", validate_and_format_date("02/30/2023") == "Invalid Date")
print("Test Case 3 ('01/01/2024'):", validate_and_format_date("01/01/2024") == "2024-01-01")
print("Test Case 4 ('12/31/2023'):", validate_and_format_date("12/31/2023") == "2023-12-31")
print("Test Case 5 ('13/01/2023'):", validate_and_format_date("13/01/2023") == "Invalid Date") # Invalid month
print("Test Case 6 ('10-15-2023'):", validate_and_format_date("10-15-2023") == "Invalid Date") # Invalid format
print("Test Case 7 (' 10/15/2023 '):", validate_and_format_date(" 10/15/2023 ") == "2023-10-15") # Leading/trailing spaces
print("Test Case 8 (None):", validate_and_format_date(None) == "Invalid Date") # Non-string input
if __name__ == "__main__":
    user_date_str = input("Enter a date in MM/DD/YYYY format: ")
    formatted_date = validate_and_format_date(user_date_str)
    print(f"Formatted date: {formatted_date}")
```

```
PS C:\Aiassisted coding>  & 'c:\Users\Poojasree\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\Poojasree\.cursor\extensic
python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '53913' '--' 'C:\Aiassisted coding\8.5.py'
Test Case 1 ('10/15/2023'): True
Test Case 2 ('02/30/2023'): True
Test Case 3 ('01/01/2024'): True
Test Case 4 ('12/31/2023'): True
Test Case 5 ('13/01/2023'): True
Test Case 6 ('10-15-2023'): True
Test Case 7 (' 10/15/2023 '): True
Test Case 8 (None): True
Enter a date in MM/DD/YYYY format: 12/09/2006
Formatted date: 2006-12-09
PS C:\Aiassisted coding> 
```

What this code does

- Validates dates given as strings in MM/DD/YYYY format and, if valid, returns them formatted as YYYY-MM-DD.
- Prints True/False results for several test cases.
- Prompts the user for a date and prints the formatted result or "Invalid Date".

Function: validate_and_format_date(date_str)

- If `date_str` is not a string → returns "Invalid Date".
- Trims whitespace with `.strip()`.
- Tries to parse using `datetime.datetime.strptime(date_str, "%m/%d/%Y")`.
  - On success: returns `date_object.strftime("%Y-%m-%d")`.
  - On failure (bad format or impossible date like 02/30/2023): returns "Invalid Date".

Test cases

- Check valid date: "10/15/2023" → "2023-10-15"
- Impossible date: "02/30/2023" → "Invalid Date"
- Another valid date: "01/01/2024" → "2024-01-01"
- End of year: "12/31/2023" → "2023-12-31"
- Invalid month: "13/01/2023" → "Invalid Date"
- Invalid format (uses dashes): "10-15-2023" → "Invalid Date"
- Leading/trailing spaces: " 10/15/2023 " → "2023-10-15"
- Non-string input: None → "Invalid Date"

Interactive section

- Asks: Enter a date in MM/DD/YYYY format.
- Prints the formatted date if valid; otherwise prints "Invalid Date".