AI Assisted Coding

Assignment-8.1

HTNO:2303A510F0

Batch:14

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

Security Context)

• Task: Apply AI to generate at least 3 assert test cases for

is_strong_password(password) and implement the validator

function.

• Requirements:

o Password must have at least 8 characters.

o Must include uppercase, lowercase, digit, and special

character.

o 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

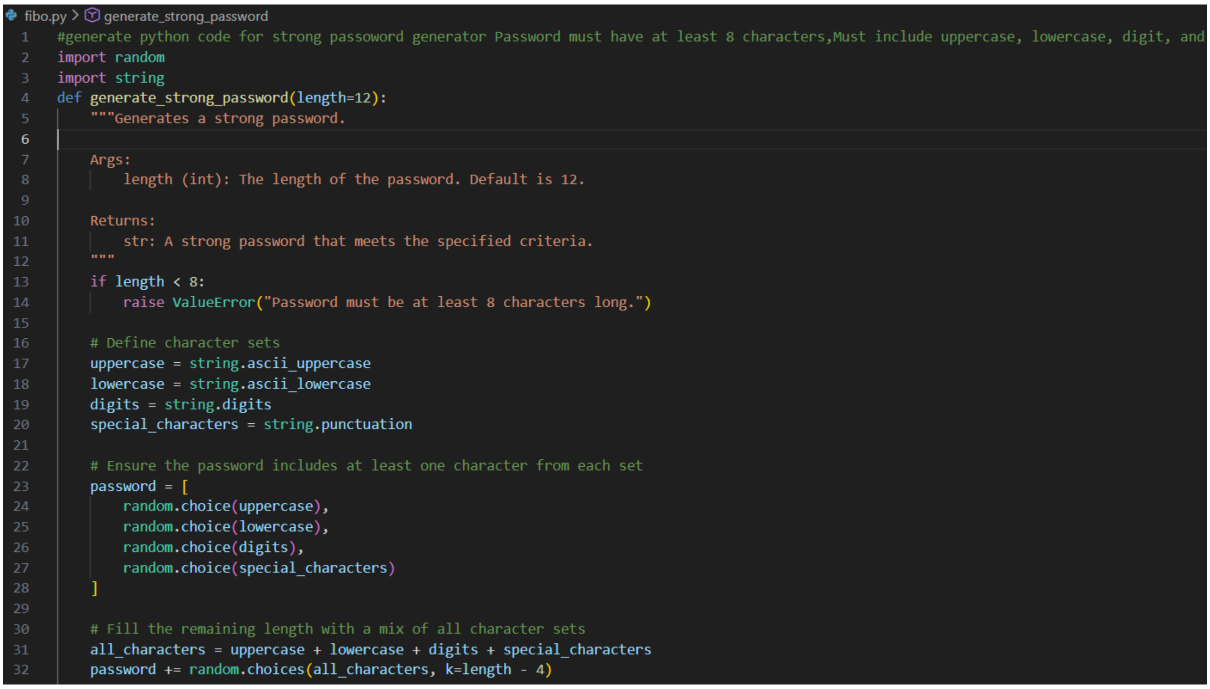
Expected Output #1:

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

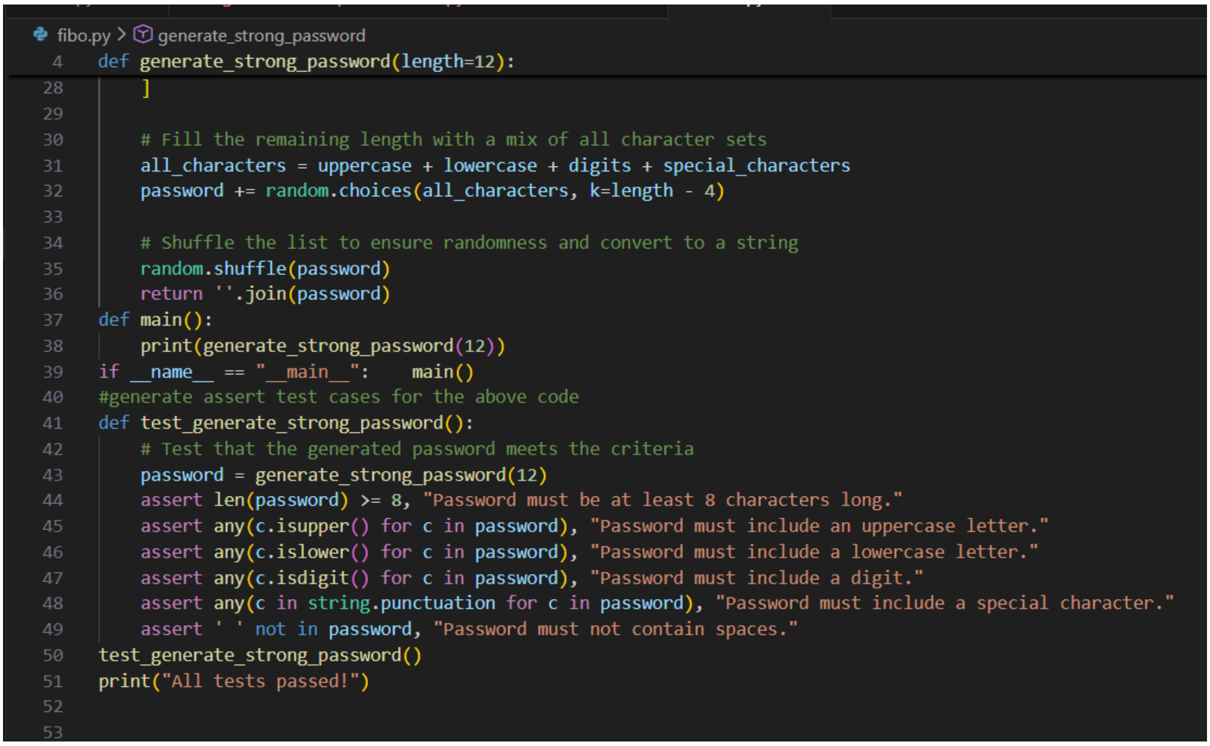
Prompt:

#generate python code for strong passoword generator Password must have at least 8 characters,Must include uppercase, lowercase, digit, and special characte, Must not contain spaces.

Code:

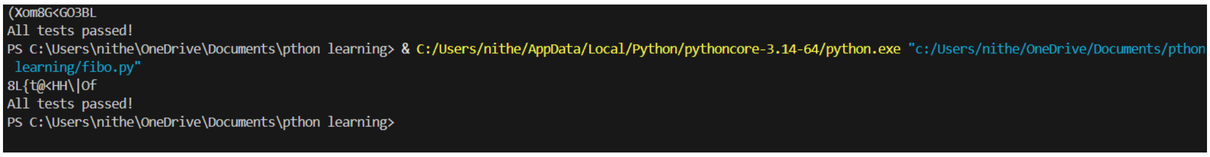


```python
#generate python code for strong passoword generator Password must have at least 8 characters,Must include uppercase, lowercase, digit, and
import random
import string
def generate_strong_password(length=12):
    """Generates a strong password.

    Args:
        length (int): The length of the password. Default is 12.

    Returns:
        str: A strong password that meets the specified criteria.
    """
    if length < 8:
        raise ValueError("Password must be at least 8 characters long.")

    # Define character sets
    uppercase = string.ascii_uppercase
    lowercase = string.ascii_lowercase
    digits = string.digits
    special_characters = string.punctuation

    # Ensure the password includes at least one character from each set
    password = [
        random.choice(uppercase),
        random.choice(lowercase),
        random.choice(digits),
        random.choice(special_characters)
    ]

    # Fill the remaining length with a mix of all character sets
    all_characters = uppercase + lowercase + digits + special_characters
    password += random.choices(all_characters, k=length - 4)
```



```python
def generate_strong_password(length=12):
        ]

        # Fill the remaining length with a mix of all character sets
        all_characters = uppercase + lowercase + digits + special_characters
        password += random.choices(all_characters, k=length - 4)

        # Shuffle the list to ensure randomness and convert to a string
        random.shuffle(password)
        return ''.join(password)
def main():
        print(generate_strong_password(12))
if __name__ == "__main__":     main()
#generate assert test cases for the above code
def test_generate_strong_password():
        # Test that the generated password meets the criteria
        password = generate_strong_password(12)
        assert len(password) >= 8, "Password must be at least 8 characters long."
        assert any(c.isupper() for c in password), "Password must include an uppercase letter."
        assert any(c.islower() for c in password), "Password must include a lowercase letter."
        assert any(c.isdigit() for c in password), "Password must include a digit."
        assert any(c in string.punctuation for c in password), "Password must include a special character."
        assert ' ' not in password, "Password must not contain spaces."
test_generate_strong_password()
print("All tests passed!")
```

Output:



```
(Xom8G<GO3BL
All tests passed!
PS C:\Users\nithe\OneDrive\Documents\pthon learning> & C:/Users/nithe/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/nithe/OneDrive/Documents/pthon
 learning/fibo.py"
8L{t@<HH\|Of
All tests passed!
PS C:\Users\nithe\OneDrive\Documents\pthon learning>
```

Justification:

By validating these conditions through assert test cases, we ensure the function reliably identifies strong passwords and rejects weak ones.

Task Description #2 (Number Classification with Loops – Apply

AI for Edge Case Handling)

- Task: 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"

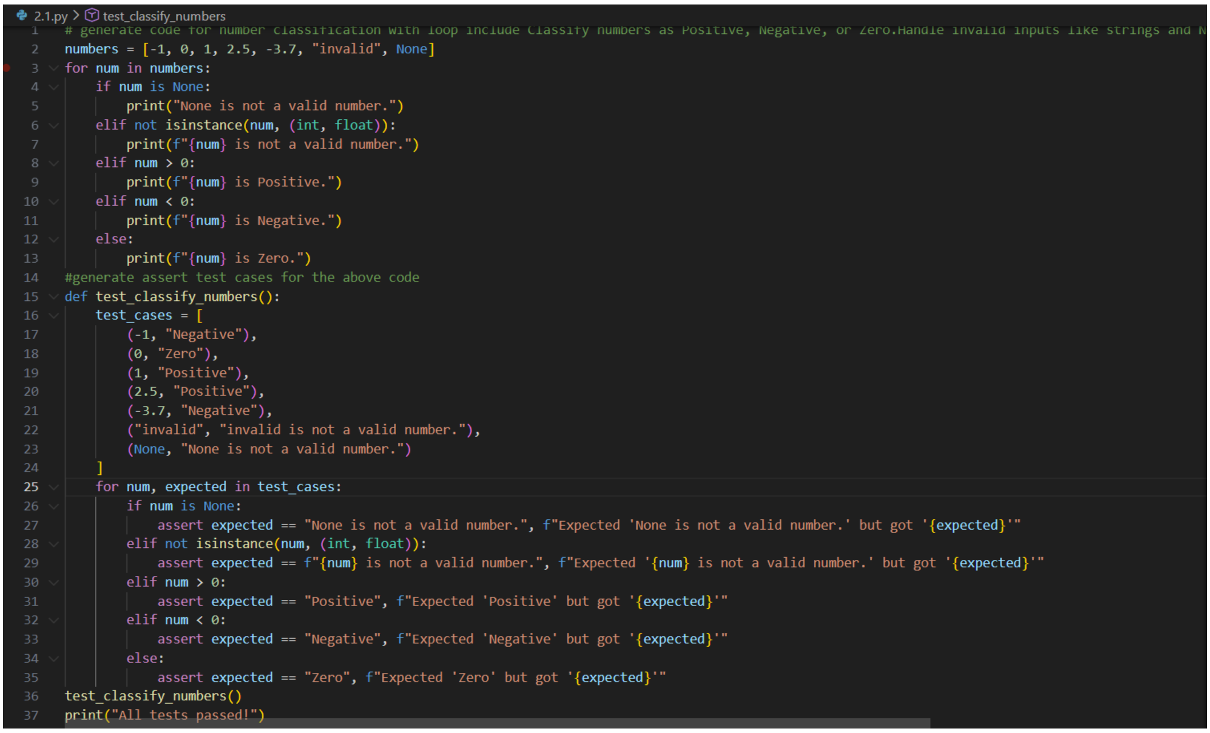
Expected Output #2:

- Classification logic passing all assert tests.

Prompt:

# generate code for number classification with loop include Classify numbers as Positive, Negative, or Zero.Handle invalid inputs like strings and None, Include boundary conditions (-1, 0, 1).

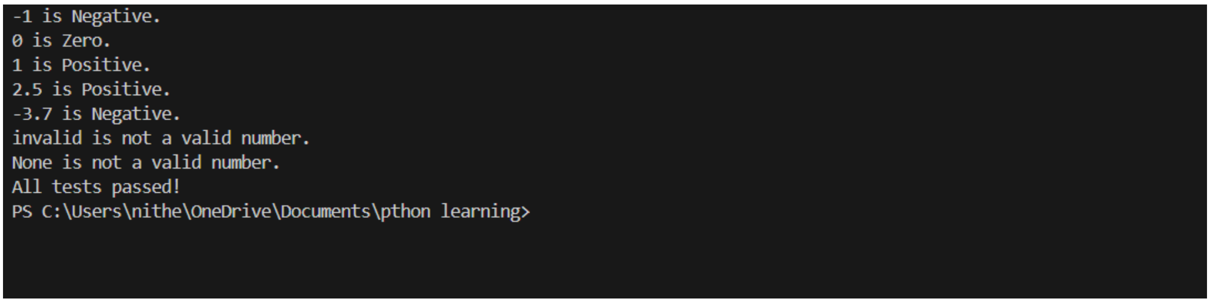
Code:



2.1.py > test_classify_numbers

```python
# generate code for number classification with loop include Classify numbers as Positive, Negative, or Zero.Handle invalid inputs like strings and N
numbers = [-1, 0, 1, 2.5, -3.7, "invalid", None]
for num in numbers:
    if num is None:
        print("None is not a valid number.")
    elif not isinstance(num, (int, float)):
        print(f"{num} is not a valid number.")
    elif num > 0:
        print(f"{num} is Positive.")
    elif num < 0:
        print(f"{num} is Negative.")
    else:
        print(f"{num} is Zero.")
#generate assert test cases for the above code
def test_classify_numbers():
    test_cases = [
        (-1, "Negative"),
        (0, "Zero"),
        (1, "Positive"),
        (2.5, "Positive"),
        (-3.7, "Negative"),
        ("invalid", "invalid is not a valid number."),
        (None, "None is not a valid number.")
    ]
    for num, expected in test_cases:
        if num is None:
            assert expected == "None is not a valid number.", f"Expected 'None is not a valid number.' but got '{expected}'"
        elif not isinstance(num, (int, float)):
            assert expected == f"{num} is not a valid number.", f"Expected '{num} is not a valid number.' but got '{expected}'"
        elif num > 0:
            assert expected == "Positive", f"Expected 'Positive' but got '{expected}'"
        elif num < 0:
            assert expected == "Negative", f"Expected 'Negative' but got '{expected}'"
        else:
            assert expected == "Zero", f"Expected 'Zero' but got '{expected}'"
test_classify_numbers()
print("All tests passed!")
```

Output:



```
-1 is Negative.
0 is Zero.
1 is Positive.
2.5 is Positive.
-3.7 is Negative.
invalid is not a valid number.
None is not a valid number.
All tests passed!
PS C:\Users\nithe\OneDrive\Documents\pthon learning>
```

Justification:

The classification function uses loop-based logic and input validation to correctly identify positive, negative, and zero values while safely handling invalid inputs like strings or None.

By testing boundary conditions such as -1, 0, and 1 through assert statements, we ensure accurate behavior across edge cases and prevent unexpected runtime errors.

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

Analysis)

• Task: Use AI to generate at least 3 assert test cases for

is_anagram(str1, str2) and implement the function.

• Requirements:

o Ignore case, spaces, and punctuation.

o 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:

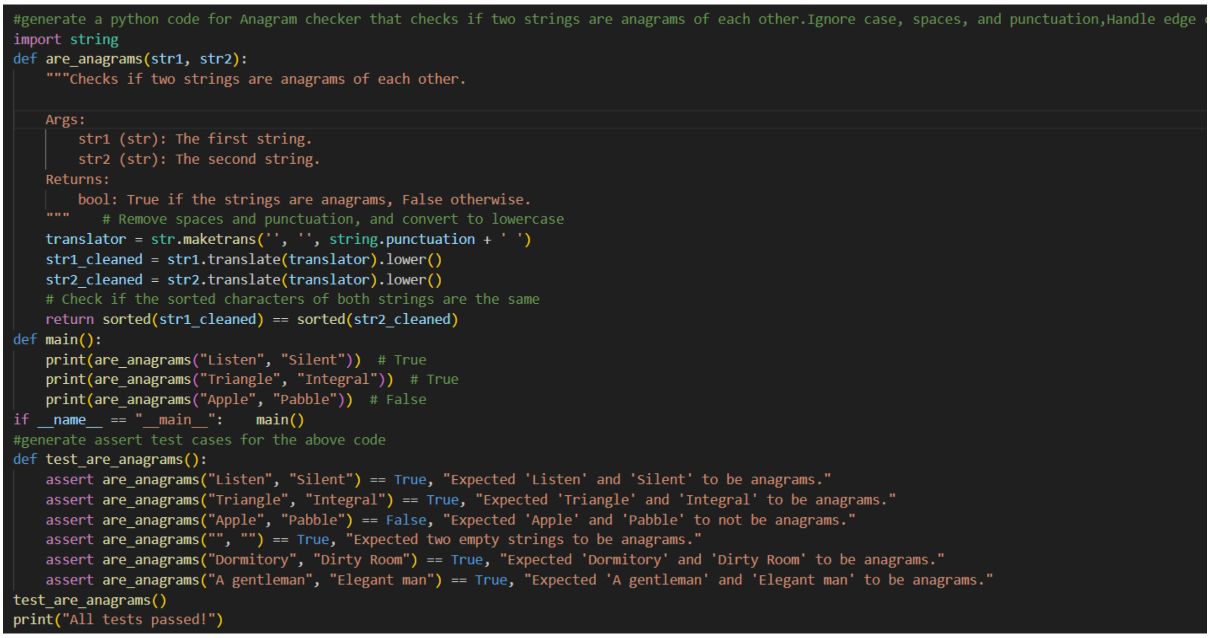
- Function correctly identifying anagrams and passing all AI-

generated tests

Prompt:

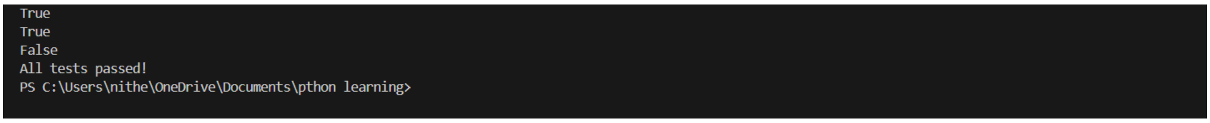
#generate a python code for Anagram checker that checks if two strings are anagrams of each other.Ignore case, spaces, and punctuation,Handle edge cases (empty strings, identical words)

Code:



```
#generate a python code for Anagram checker that checks if two strings are anagrams of each other.Ignore case, spaces, and punctuation,Handle edge 
import string
def are_anagrams(str1, str2):
    """Checks if two strings are anagrams of each other.

    Args:
        str1 (str): The first string.
        str2 (str): The second string.
    Returns:
        bool: True if the strings are anagrams, False otherwise.
    """    # Remove spaces and punctuation, and convert to lowercase
    translator = str.maketrans('', '', string.punctuation + ' ')
    str1_cleaned = str1.translate(translator).lower()
    str2_cleaned = str2.translate(translator).lower()
    # Check if the sorted characters of both strings are the same
    return sorted(str1_cleaned) == sorted(str2_cleaned)
def main():
    print(are_anagrams("Listen", "Silent"))  # True
    print(are_anagrams("Triangle", "Integral"))  # True
    print(are_anagrams("Apple", "Pabble"))  # False
if __name__ == "__main__":    main()
#generate assert test cases for the above code
def test_are_anagrams():
    assert are_anagrams("Listen", "Silent") == True, "Expected 'Listen' and 'Silent' to be anagrams."
    assert are_anagrams("Triangle", "Integral") == True, "Expected 'Triangle' and 'Integral' to be anagrams."
    assert are_anagrams("Apple", "Pabble") == False, "Expected 'Apple' and 'Pabble' to not be anagrams."
    assert are_anagrams("", "") == True, "Expected two empty strings to be anagrams."
    assert are_anagrams("Dormitory", "Dirty Room") == True, "Expected 'Dormitory' and 'Dirty Room' to be anagrams."
    assert are_anagrams("A gentleman", "Elegant man") == True, "Expected 'A gentleman' and 'Elegant man' to be anagrams."
test_are_anagrams()
print("All tests passed!")
```

Output:



```
True
True
False
All tests passed!
PS C:\Users\nithe\OneDrive\Documents\pthon learning>
```

Justification:

The anagram checker normalizes input by ignoring case, spaces, and punctuation to ensure accurate comparison of character frequency between two strings.

Task Description #4 (Inventory Class – Apply AI to Simulate Real-

World Inventory System)

• Task: Ask AI to generate at least 3 assert-based tests for an

Inventory class with stock management.

• Methods:

o add_item(name, quantity)

o remove_item(name, quantity)

o 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
```

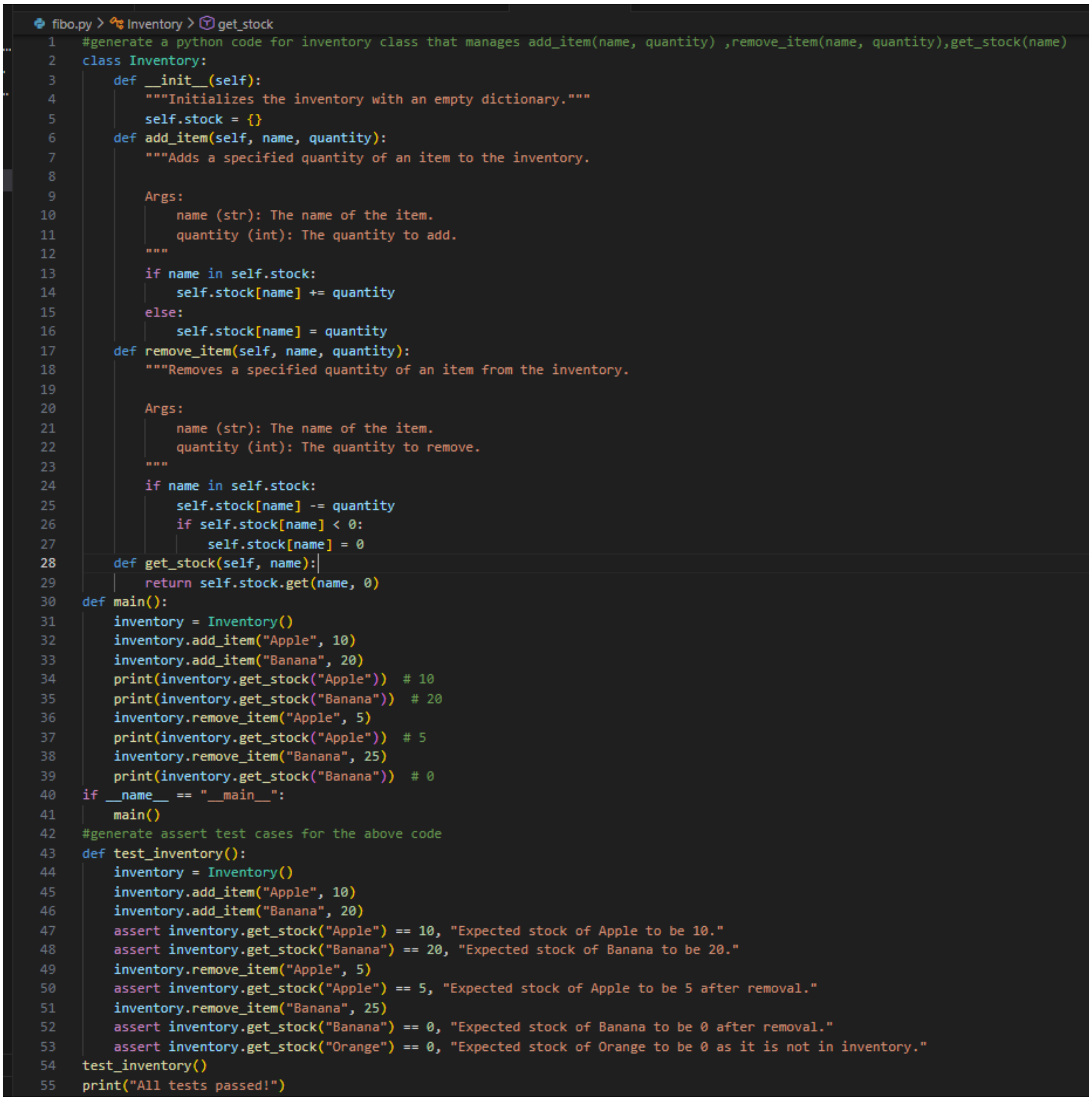
Expected Output #4:

- Fully functional class passing all assertions.

Prompt:

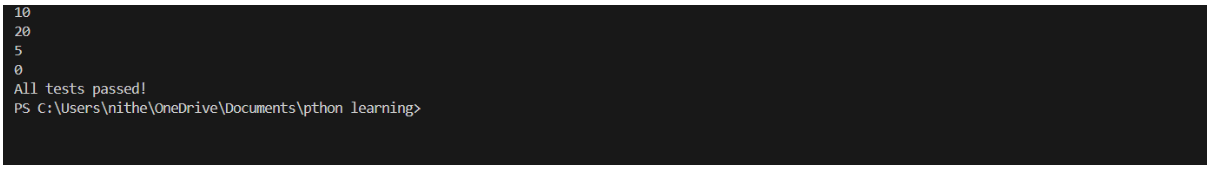
#generate a python code for inventory class that manages add_item(name, quantity) ,remove_item(name, quantity),get_stock(name)

Code:



```python
#generate a python code for inventory class that manages add_item(name, quantity) ,remove_item(name, quantity),get_stock(name)
class Inventory:
    def __init__(self):
        """Initializes the inventory with an empty dictionary."""
        self.stock = {}
    def add_item(self, name, quantity):
        """Adds a specified quantity of an item to the inventory.

        Args:
            name (str): The name of the item.
            quantity (int): The quantity to add.
        """
        if name in self.stock:
            self.stock[name] += quantity
        else:
            self.stock[name] = quantity
    def remove_item(self, name, quantity):
        """Removes a specified quantity of an item from the inventory.

        Args:
            name (str): The name of the item.
            quantity (int): The quantity to remove.
        """
        if name in self.stock:
            self.stock[name] -= quantity
            if self.stock[name] < 0:
                self.stock[name] = 0
    def get_stock(self, name):
        return self.stock.get(name, 0)
def main():
    inventory = Inventory()
    inventory.add_item("Apple", 10)
    inventory.add_item("Banana", 20)
    print(inventory.get_stock("Apple"))  # 10
    print(inventory.get_stock("Banana"))  # 20
    inventory.remove_item("Apple", 5)
    print(inventory.get_stock("Apple"))  # 5
    inventory.remove_item("Banana", 25)
    print(inventory.get_stock("Banana"))  # 0
if __name__ == "__main__":
    main()
#generate assert test cases for the above code
def test_inventory():
    inventory = Inventory()
    inventory.add_item("Apple", 10)
    inventory.add_item("Banana", 20)
    assert inventory.get_stock("Apple") == 10, "Expected stock of Apple to be 10."
    assert inventory.get_stock("Banana") == 20, "Expected stock of Banana to be 20."
    inventory.remove_item("Apple", 5)
    assert inventory.get_stock("Apple") == 5, "Expected stock of Apple to be 5 after removal."
    inventory.remove_item("Banana", 25)
    assert inventory.get_stock("Banana") == 0, "Expected stock of Banana to be 0 after removal."
    assert inventory.get_stock("Orange") == 0, "Expected stock of Orange to be 0 as it is not in inventory."
test_inventory()
print("All tests passed!")
```

Output:



```
10
20
5
0
All tests passed!
PS C:\Users\nithe\OneDrive\Documents\pthon learning>
```

Justification:

The Inventory class ensures accurate stock updates through add and remove operations.
Assert tests confirm correct quantity tracking and reliable inventory management.

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

Data Validation)

• Task: Use AI to generate at least 3 assert test cases for

validate_and_format_date(date_str) to check and convert

dates.

• Requirements:

o Validate "MM/DD/YYYY" format.

o Handle invalid dates.

o 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:

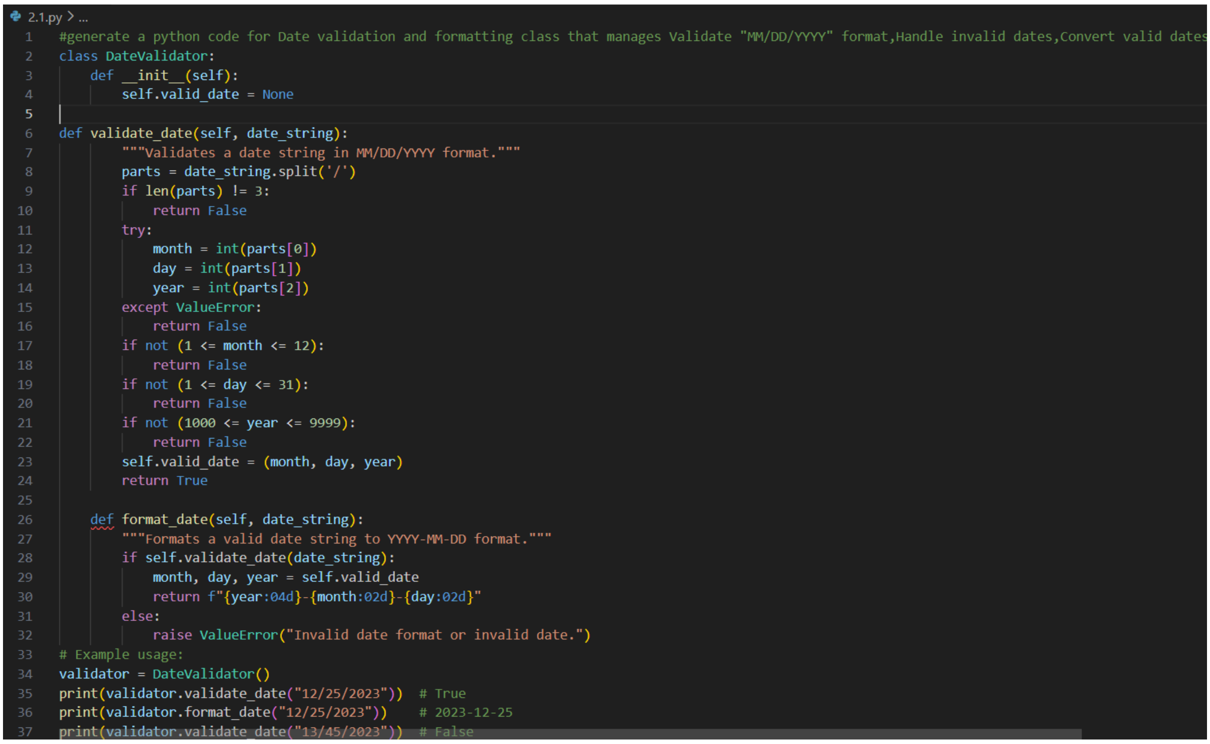
- Function passes all AI-generated assertions and handles edge

cases.

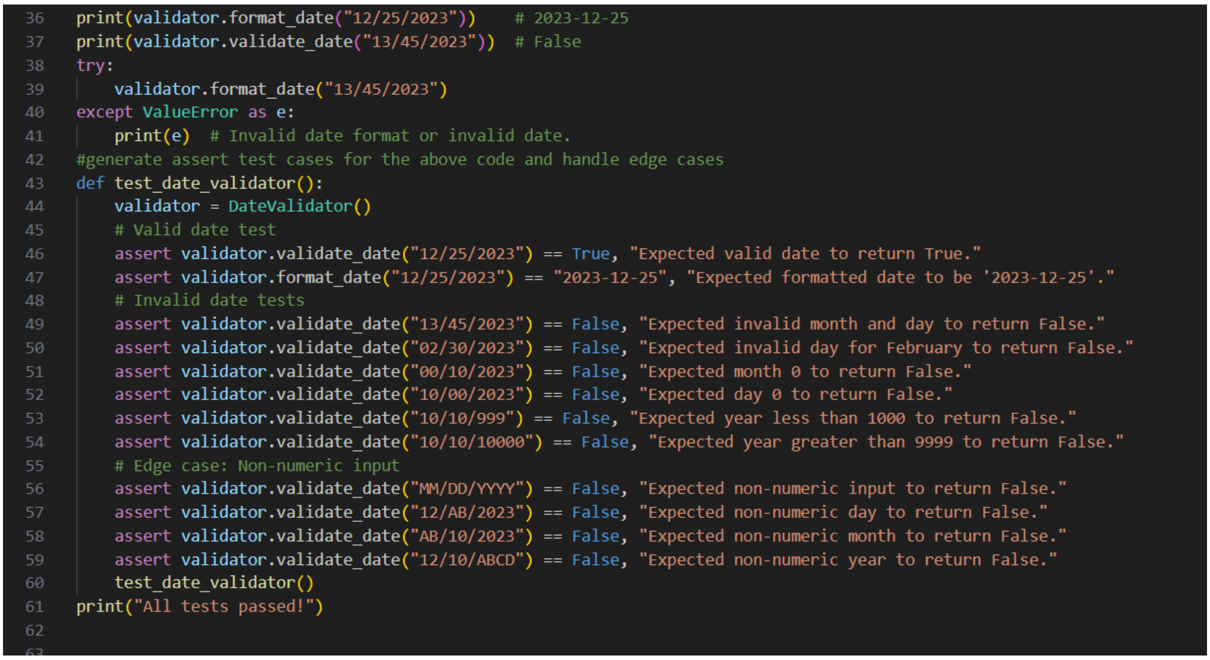
Prompt:

#generate a python code for Date validation and formatting class that manages Validate "MM/DD/YYYY" format,Handle invalid dates,Convert valid dates to "YYYY-MM-DD".

Code:

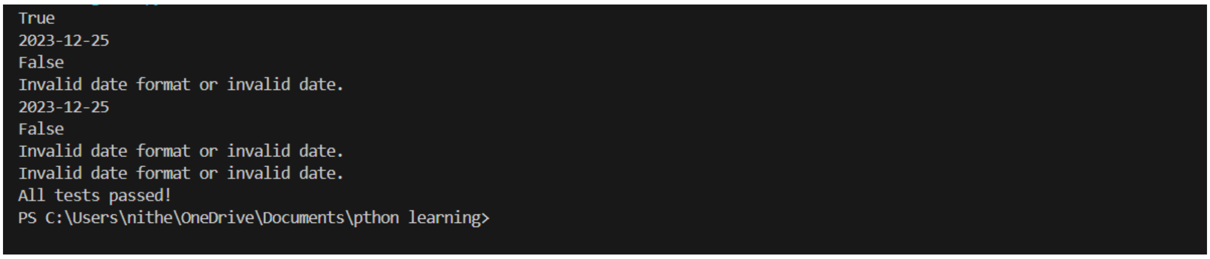


```python
#generate a python code for Date validation and formatting class that manages Validate "MM/DD/YYYY" format,Handle invalid dates,Convert valid dates
class DateValidator:
    def __init__(self):
        self.valid_date = None

def validate_date(self, date_string):
        """Validates a date string in MM/DD/YYYY format."""
        parts = date_string.split('/')
        if len(parts) != 3:
            return False
        try:
            month = int(parts[0])
            day = int(parts[1])
            year = int(parts[2])
        except ValueError:
            return False
        if not (1 <= month <= 12):
            return False
        if not (1 <= day <= 31):
            return False
        if not (1000 <= year <= 9999):
            return False
        self.valid_date = (month, day, year)
        return True

    def format_date(self, date_string):
        """Formats a valid date string to YYYY-MM-DD format."""
        if self.validate_date(date_string):
            month, day, year = self.valid_date
            return f"{year:04d}-{month:02d}-{day:02d}"
        else:
            raise ValueError("Invalid date format or invalid date.")
# Example usage:
validator = DateValidator()
print(validator.validate_date("12/25/2023"))  # True
print(validator.format_date("12/25/2023"))    # 2023-12-25
print(validator.validate_date("13/45/2023"))  # False
```



```python
print(validator.format_date("12/25/2023"))    # 2023-12-25
print(validator.validate_date("13/45/2023"))  # False
try:
    validator.format_date("13/45/2023")
except ValueError as e:
    print(e)  # Invalid date format or invalid date.
#generate assert test cases for the above code and handle edge cases
def test_date_validator():
    validator = DateValidator()
    # Valid date test
    assert validator.validate_date("12/25/2023") == True, "Expected valid date to return True."
    assert validator.format_date("12/25/2023") == "2023-12-25", "Expected formatted date to be '2023-12-25'."
    # Invalid date tests
    assert validator.validate_date("13/45/2023") == False, "Expected invalid month and day to return False."
    assert validator.validate_date("02/30/2023") == False, "Expected invalid day for February to return False."
    assert validator.validate_date("00/10/2023") == False, "Expected month 0 to return False."
    assert validator.validate_date("10/00/2023") == False, "Expected day 0 to return False."
    assert validator.validate_date("10/10/999") == False, "Expected year less than 1000 to return False."
    assert validator.validate_date("10/10/10000") == False, "Expected year greater than 9999 to return False."
    # Edge case: Non-numeric input
    assert validator.validate_date("MM/DD/YYYY") == False, "Expected non-numeric input to return False."
    assert validator.validate_date("12/AB/2023") == False, "Expected non-numeric day to return False."
    assert validator.validate_date("AB/10/2023") == False, "Expected non-numeric month to return False."
    assert validator.validate_date("12/10/ABCD") == False, "Expected non-numeric year to return False."
    test_date_validator()
print("All tests passed!")
```

Output:



```
True
2023-12-25
False
Invalid date format or invalid date.
2023-12-25
False
Invalid date format or invalid date.
Invalid date format or invalid date.
All tests passed!
PS C:\Users\nithe\OneDrive\Documents\pthon learning>
```

Justification:

The function ensures proper date validation by checking the correct "MM/DD/YYYY" format and detecting invalid calendar dates.
Assert-based tests confirm accurate conversion to "YYYY-MM-DD" and reliable handling of edge cases.