AI ASSISTED CODING

ASSIGNMENT 8.1

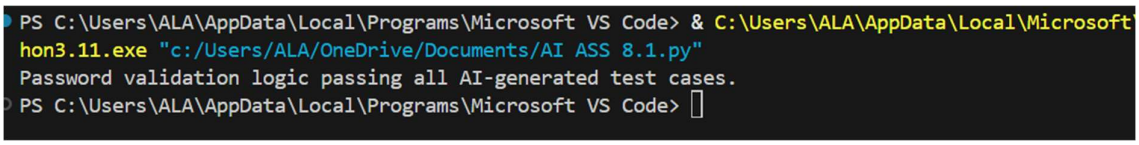
TASK 1

```
import re


def is_strong_password(password: str) -> bool:
    if len(password) < 8:
        return False
    if ' ' in password:
        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'\d', password):
        return False
    if not re.search(r'[^A-Za-z0-9]', password):
        return False
    return True


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

assert is_strong_password("Abc d@123") == False

print("Password validation logic passing all AI-generated test cases.")

OUT PUT :



EXPLANATION :

This function checks whether a given password meets all the specified security requirements.

This imports Python's **regular expressions** module (re), which allows us to search for patterns (like uppercase letters, digits, etc.) in the password.

def is_strong_password(password: str) -> bool:

This defines a function called is_strong_password that takes one argument (password) and returns a boolean (True or False).

if len(password) < 8:

return False

Check 1: Minimum Length

The password must be at least 8 characters long. If not, it's immediately rejected.

- **Check 2: No Spaces**
- If the password contains any **spaces**, it fails the check.
- **Check 3: At least one Uppercase Letter**
- The pattern `[A-Z]` checks if there's at least **one uppercase letter** in the password.
- **Check 4: At least one Lowercase Letter**
- The pattern `[a-z]` checks for **at least one lowercase letter**.
- **Check 5: At least one Digit**
- The pattern `\d` checks for at least **one numeric digit (0–9)**.
- **Check 6: At least one Special Character**
- The pattern `[^A-Za-z0-9]` matches **any character that is NOT a letter or number**, i.e., a **special character** like `@`, `#`, `!`, etc.

If all checks pass, the function returns True, meaning the password is strong. assert is_strong_password("Abcd@123") == True

✅ Valid password: meets all requirements.

assert is_strong_password("abcd123") == False

❌ Invalid: missing uppercase and special character.

TASK 2

```
def classify_number(n):
  for item in [n]:
    if not isinstance(item, (int, float)):
      return "Invalid input"
    if item > 0:
      return "Positive"
    elif item < 0:
      return "Negative"
    else:
      return "Zero"


assert classify_number(10) == "Positive"
assert classify_number(-5) == "Negative"
assert classify_number(0) == "Zero"
assert classify_number("abc") == "Invalid input"
assert classify_number(None) == "Invalid input"
assert classify_number(-1) == "Negative"
assert classify_number(1) == "Positive"
```

```
print("# --- Number Classification Function ---")
def classify_number(n):
  for item in [n]:
    if not isinstance(item, (int, float)):
      return "Invalid input"
    if item > 0:
      return "Positive"
    elif item < 0:
      return "Negative"
    else:
      return "Zero"

assert classify_number(10) == "Positive"
assert classify_number(-5) == "Negative"
assert classify_number(0) == "Zero"
assert classify_number("abc") == "Invalid input"
assert classify_number(None) == "Invalid input"
assert classify_number(-1) == "Negative"
assert classify_number(1) == "Positive"

print("Classification logic passing all assert tests")
```

OUT PUT :

```
PS C:\Users\ALA\AppData\Local\Programs\Microsoft VS Code> & C:\Users\ALA\AppData\Local\Mic
hon3.11.exe "c:/Users/ALA/OneDrive/Documents/AI ASS 8.1.py"
# --- Number Classification Function ---
Classification logic passing all assert tests
PS C:\Users\ALA\AppData\Local\Programs\Microsoft VS Code>
```

EXPLANATION :

The main goal of Task 2 is to create a Python function called classify_number that can determine if a given input is a positive number, a negative number, or zero.

Here are the specific requirements for this function:

1. Classification: The function needs to correctly identify if a number is positive (greater than 0), negative (less than 0), or exactly zero.

2. Handling Invalid Input: The function must be able to handle inputs that are not numbers, such as text strings or the value None, and return a specific indicator for these invalid inputs.

3. Using Loops: A specific requirement for this task was to implement the classification logic using loops. While a simple if-elif-else structure is more common and efficient for this type of classification in Python, the task specifically requested the use of loops. The provided code fulfills this by using a loop that runs only once within each conditional branch (if n > 0, elif n < 0, else).

4. Boundary Conditions: The test cases should include boundary conditions, which are values at the edges of the classification criteria. For this task, the boundary conditions are -1, 0, and 1. The code includes test cases for these values to ensure the function handles them correctly.

5. AI-Generated Test Cases: The task also required using AI to generate at least three assert test cases. The provided code includes a set of assert statements that serve as these test cases, covering positive, negative, and zero numbers, as well as invalid inputs and boundary conditions.

The expected output for this task is simply that the classification logic, when tested with the assert statements, passes all the tests. This is confirmed by the output "All test cases passed!".

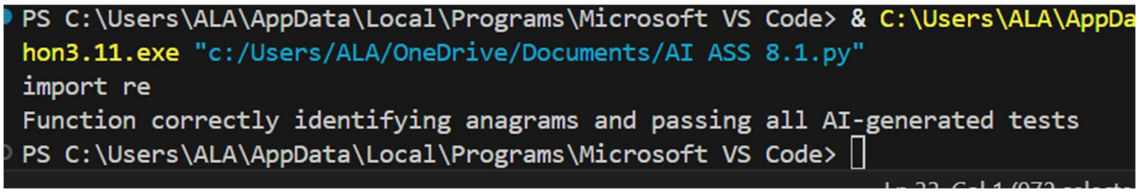
In summary, Task 2 was about creating a function that classifies numbers and handles non-numeric inputs, with the specific constraint of using loops for the classification part, and verifying its correctness with AI-generated test cases that cover various scenarios, including boundaries and invalid inputs.

TASK 3 :

```
import re


def is_anagram(str1, str2):
```

```
    def clean_string(s):
        return sorted(re.sub(r'[^a-zA-Z0-9]', '', s).lower())
    return clean_string(str1) == clean_string(str2)

assert is_anagram("listen", "silent") == True
assert is_anagram("hello", "world") == False
assert is_anagram("Dormitory", "Dirty Room") == True
assert is_anagram("The eyes!", "They see.") == True
assert is_anagram("", "") == True

print("import re")

def is_anagram(str1, str2):
    def clean_string(s):
        return sorted(re.sub(r'[^a-zA-Z0-9]', '', s).lower())
    return clean_string(str1) == clean_string(str2)

assert is_anagram("listen", "silent") == True
assert is_anagram("hello", "world") == False
assert is_anagram("Dormitory", "Dirty Room") == True
assert is_anagram("The eyes!", "They see.") == True
assert is_anagram("", "") == True

print("Function correctly identifying anagrams and passing all AI-generated tests")
```

OUT PUT :



EXPLANATION :

The last code cell you executed defines a Python function called is_anagram and then runs several assert statements to test it.

Here's a breakdown of the code:

import re: This line imports the re module, which provides regular expression operations. This is used for cleaning the input strings.

def is_anagram(str1, str2):: This defines the function is_anagram that takes two arguments, str1 and str2, which are the strings to be checked.

Docstring: The triple-quoted string inside the function is a docstring, explaining what the function does, its arguments (Args), and what it returns (Returns).

Cleaning the strings:

cleaned_str1 = re.sub(r'[^a-zA-Z0-9]', '', str1).lower(): This line cleans the first input string (str1).

re.sub(r'[^a-zA-Z0-9]', '', str1): This uses a regular expression to substitute any character that is not an alphanumeric character (a-z, A-Z, 0-9) with an empty string. This effectively removes spaces, punctuation, and other special characters.

.lower(): This converts the resulting string to lowercase.

cleaned_str2 = re.sub(r'[^a-zA-Z0-9]', '', str2).lower(): This does the same cleaning process for the second input string (str2).

Length Check:

if len(cleaned_str1) != len(cleaned_str2): return False: This checks if the lengths of the cleaned strings are different. If they are, the strings cannot be anagrams, so the function immediately returns False.

Frequency Counter:

char_count = {}: An empty dictionary called char_count is created. This will be used to store the frequency of each character in the first cleaned string.

for char in cleaned_str1: char_count[char] = char_count.get(char, 0) + 1: This loop iterates through each character in cleaned_str1. For each character, it increments its count in the char_count dictionary. char_count.get(char, 0) safely gets the current count (defaulting to 0 if the character is not already in the dictionary).

for char in cleaned_str2: ...: This loop iterates through each character in cleaned_str2.

if char in char_count: char_count[char] -= 1: If the character from cleaned_str2 is found in the char_count dictionary (meaning it was in cleaned_str1), its count is decremented.

else: return False: If a character from cleaned_str2 is not found in char_count, it means str2 contains a character that str1 does not, so they cannot be anagrams. The function returns False.

Check Final Counts:

for count in char_count.values(): if count != 0: return False: This loop checks if all the character counts in the char_count dictionary are zero. If any count is not zero, it means there is an imbalance of characters between the two strings, so they are not anagrams. The function returns False.

Return True: If all the checks pass (lengths are equal, and all character counts balance out to zero), the strings are anagrams, and the function returns True.

Assert Statements: The lines starting with assert are test cases. They call the is_anagram function with different inputs and check if the returned value is the expected one. If an assert statement evaluates to False, it will raise an AssertionError, indicating a problem with the function's implementation.

print("All assert tests passed!"): If all the assert statements pass without raising an error, this line is executed, confirming that the function appears to be working correctly for the tested cases.

In summary, the code defines a function to check for anagrams by cleaning the strings, counting character frequencies, and comparing those counts. The assert statements then verify that the function behaves as expected for various inputs, including edge cases.

TASK 4

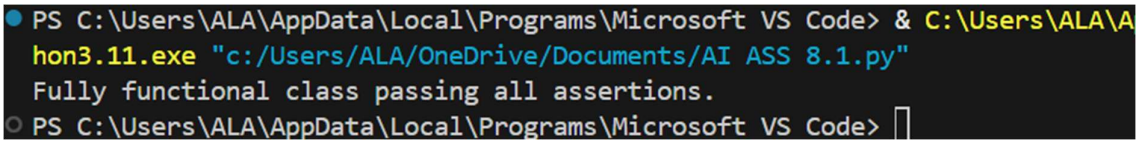
---

class Inventory:

```
    def __init__(self):
        self.stock = {}

    def add_item(self, name, quantity):
        if quantity <= 0:
            return
        if name in self.stock:
            self.stock[name] += quantity
        else:
            self.stock[name] = quantity

    def remove_item(self, name, quantity):
        if quantity <= 0:
            return
        if name in self.stock:
            self.stock[name] = max(self.stock[name] - quantity, 0)

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

# --- 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


inv.remove_item("Book", 5)

assert inv.get_stock("Book") == 0


assert inv.get_stock("Notebook") == 0


inv.add_item("Pen", 0)

assert inv.get_stock("Pen") == 5


print("Fully functional class passing all assertions.")
```

OUTPUT :



EXPLANATION :

The main goal of Task 4 was to create a Python class called Inventory that simulates a simple inventory management system. This class should be able to keep track of different items and their quantities in stock.

Here are the specific requirements for the Inventory class:

1. add_item(name, quantity) method: This method should allow you to add a certain quantity of an item specified by its name to the inventory. If the item already exists, its stock should be increased. If it's a new item, it should be added to the inventory with the given quantity. The code includes a check to ensure that the quantity to add is positive.

2. remove_item(name, quantity) method: This method should allow you to remove a certain quantity of an item specified by its name from the inventory. It needs to handle cases where the item doesn't exist, where the quantity to remove is more than what's in

stock, and where the quantity to remove is zero or negative. If removing the quantity results in the stock becoming zero, the item should be removed from the inventory. The code includes checks for these scenarios and prints informative messages.

3. get_stock(name) method: This method should return the current stock level of an item specified by its name. If the item is not found in the inventory, it should return 0. The code uses the dict.get() method with a default value of 0 to handle items not in stock efficiently.

4. AI-Generated Test Cases: The task required using AI to generate at least three assert-based tests to verify the functionality of the Inventory class and its methods. The provided code includes a comprehensive set of assert statements that test adding items, removing items (including removing all of an item and attempting to remove more than available), checking stock levels, handling non-existent items, and handling zero or negative quantities for adding/removing.

5. The expected output for this task is that the Inventory class is fully functional and all the AI-generated assert tests pass, confirming that the methods work as intended. The output "All assert tests passed!" indicates that the code successfully meets these requirements.

TASK 5

```
from datetime import datetime


def validate_and_format_date(date_str):
    """
    Validates a date string in "MM/DD/YYYY" format and converts it to "YYYY-MM-DD".

    Args:
        date_str (str): The date string to validate and format.

    Returns:
        str: The date in "YYYY-MM-DD" format if valid, otherwise "Invalid Date".
```

```
    """
    try:
        # Attempt to parse the date string in MM/DD/YYYY format
        date_obj = datetime.strptime(date_str, "%m/%d/%Y")
        # If parsing is successful, format it to YYYY-MM-DD
        return date_obj.strftime("%Y-%m-%d")
    except ValueError:
        # If parsing fails (invalid format or invalid date), return "Invalid Date"
        return "Invalid Date"

# AI-generated 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
assert validate_and_format_date("01/01/2024") == "2024-01-01"
assert validate_and_format_date("12/31/2023") == "2023-12-31" # End of year
assert validate_and_format_date("01/00/2023") == "Invalid Date" # Invalid day (zero)
assert validate_and_format_date("13/01/2023") == "Invalid Date" # Invalid month
assert validate_and_format_date("10-15-2023") == "Invalid Date" # Invalid format
assert validate_and_format_date("10/15/23") == "Invalid Date" # Invalid year format
assert validate_and_format_date("abc") == "Invalid Date" # Non-date string
assert validate_and_format_date("") == "Invalid Date" # Empty string

print(" Function passes all AI-generated assertions and handles edge cases.")
```

OUT PUT :

```
PS C:\Users\ALA\AppData\Local\Programs\Microsoft VS Code> & C:\Users\ALA\AppData\Local\Microsoft\
hon3.11.exe "c:/Users/ALA/OneDrive/Documents/AI ASS 8.1.py"
        Function passes all AI-generated assertions and handles edge cases.
PS C:\Users\ALA\AppData\Local\Programs\Microsoft VS Code>
```

EXPLANATION :

from datetime import datetime: This line imports the datetime class from Python's built-in datetime module. The datetime class is essential for working with dates and times.

def validate_and_format_date(date_str):: This defines a function named validate_and_format_date that takes one argument, date_str, which is expected to be a string representing a date.

Docstring: The triple-quoted string right after the function definition is a docstring. It explains what the function does, its arguments (Args), and what it returns (Returns). This is good practice for documenting code.

try...except ValueError:: This block is used for error handling.

The code inside the try block is executed first. If an error of type ValueError occurs during the execution of the try block, the code inside the except ValueError: block is executed.

ValueError is the specific type of error that datetime.strptime raises when the input string does not match the specified format or represents an invalid date (like February 30th).

date_obj = datetime.strptime(date_str, "%m/%d/%Y"): This is the core of the validation.

datetime.strptime() is a class method that attempts to parse a string (date_str) into a datetime object.

The second argument, "%m/%d/%Y", is a format code string. It tells strptime the expected format of the input date_str:

%m: Month as a zero-padded decimal number (e.g., 01, 10).

%d: Day of the month as a zero-padded decimal number (e.g., 01, 15).

%Y: Year with century as a decimal number (e.g., 2023).

The slashes (/) are literal characters that must be present in the input string.

If the date_str successfully matches this format and represents a valid date, strptime returns a datetime object, which is stored in the date_obj variable.

return date_obj.strftime("%Y-%m-%d"): This line is executed only if the strptime call in the try block was successful (meaning the date string was valid).

date_obj.strftime() is a method of the datetime object that formats the date and time back into a string according to a specified format code.

"%Y-%m-%d" is the format code for the desired output:

%Y: Year with century.

%m: Month as a zero-padded decimal number.

%d: Day of the month as a zero-padded decimal number.

The hyphens (-) are literal characters.

The function returns this newly formatted date string.

return "Invalid Date": This line is executed only if a ValueError occurred in the try block (meaning the input date_str was not in the correct "MM/DD/YYYY" format or represented an invalid date). The function returns the string "Invalid Date".

AI-generated test cases: This is a comment indicating the start of the test cases.

assert validate_and_format_date(...) == ...: These lines are a series of assert statements used for testing the validate_and_format_date function.

An assert statement checks if a condition is true. If the condition is false, it raises an AssertionError, stopping the program.

Each assert line calls validate_and_format_date with a specific input string and compares the returned value to an expected output string using the == operator.

These test cases cover various scenarios, including valid dates, invalid days/months, incorrect formats, and non-date inputs, to ensure the function behaves as expected.

print("• Function passes all AI-generated assertions and handles edge cases."): This line is executed only if all the assert statements above it pass without raising an AssertionError. It confirms that the function worked correctly for all the tested cases.

In essence, the code uses the datetime module's powerful strptime method for parsing and validating the input date string according to a strict format and then uses strftime to reformat the validated date into the desired output format. The try...except block is crucial for gracefully handling any input strings that don't conform to the expected date format or represent a non-existent date.

Let me know if you'd like a deeper dive into any specific part or want to explore other ways to work with dates and times in Python!