

ASSIGNMENT

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:

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

Example Assert Test Cases:

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

Expected Output 1:

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

EXPECTED CODE AND OUTPUT :

```
▶ import re

def is_strong_password(password):
    """
    Validates if a password is strong based on the following requirements:
    - At least 8 characters long.
    - Includes uppercase, lowercase, digit, and special character.
    - Does not contain spaces.
    """
    if len(password) < 8:
        return False
    if ' ' in password:
        return False

    # Check for at least one uppercase letter
    if not any(char.isupper() for char in password):
        return False

    # Check for at least one lowercase letter
    if not any(char.islower() for char in password):
        return False

    # Check for at least one digit
    if not any(char.isdigit() for char in password):
        return False

    # Check for at least one special character (using a regex)
    special_chars = re.compile(r'[@#$%^&*,.?":{}|<>]')
    if not special_chars.search(password):
        return False

    return True
```

```

# Assert test cases
assert is_strong_password("Abcd@123") == True
assert is_strong_password("abcd123") == False # Missing uppercase and special char
assert is_strong_password("ABCD@1234") == False # Missing lowercase
assert is_strong_password("abcde FGH1") == False # Contains space
assert is_strong_password("Abcdefg") == False # Too short
assert is_strong_password("Abcdefg123") == False # Missing special char
assert is_strong_password("ABCDEFG!@#") == False # Missing digit
assert is_strong_password("abcdefg!@#") == False # Missing uppercase
assert is_strong_password("ABCDEFG123") == False # Missing lowercase

print("All assert tests passed!")

```

→ All assert tests passed!

EXPLANATION OF THE CODE :

This code defines a Python function called `is_strong_password` that checks if a given password meets certain criteria for strength.

Let's break down the code:

1. `import re`: This line imports the `re` module, which provides support for regular expressions. Regular expressions are used here to check for special characters in the password.
2. `def is_strong_password(password):`: This defines the function `is_strong_password` which takes one argument, `password` (the string to be validated).
3. `if len(password) < 8:`: This checks if the length of the password is less than 8 characters. If it is, the function immediately returns `False` because the password doesn't meet the minimum length requirement.
4. `if ' ' in password:`: This checks if there is a space character within the password string. If a space is found, the function returns `False` as spaces are not allowed.
5. `if not any(char.isupper() for char in password):`: This line checks if there is at least one uppercase letter in the password.

- `char.isupper()` checks if a character is uppercase.
- `(char.isupper() for char in password)` is a generator expression that creates a sequence of boolean values (True or False) for each character in the password.
- `any(...)` returns `True` if at least one of the values in the sequence is `True`.
- `not any(...)` means if *none* of the characters are uppercase, the condition is met and the function returns `False`.
- 6. `if not any(char.islower() for char in password):`: This is similar to the uppercase check, but it verifies that there is at least one lowercase letter in the password using `char.islower()`.
- 7. `if not any(char.isdigit() for char in password):`: This checks for at least one digit (0-9) in the password using `char.isdigit()`.
- 8. `special_chars = re.compile(r'[@#$%^&(),.?":{}|<>]')`: This line compiles a regular expression pattern that matches a set of common special characters. Compiling the pattern makes it slightly more efficient if you were to use it multiple times.

9. `if not special_chars.search(password) :` This uses the compiled regular expression to search for any of the defined special characters within the password string.
- o `special_chars.search(password)` returns a match object if a special character is found, otherwise it returns `None`.
 - o If `search` returns `None` (meaning no special character is found), `not special_chars.search(password)` evaluates to `True`, and the function returns `False`.
10. `return True` : If the password passes all the previous checks (length, no spaces, includes uppercase, lowercase, digit, and special character), the function reaches this line and returns `True`, indicating that the password is strong.

The code also includes several `assert` statements after the function definition. These are test cases that verify if the function behaves as expected with different inputs. If an `assert` statement evaluates to `False`, it will raise an `AssertionError`, indicating a problem with the function's logic. The `print("All assert tests passed!")` line is executed only if all the assert statements pass without raising an error.

In summary, the code provides a clear and robust way to validate password strength based on the specified criteria.

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.

EXPECTED CODE WITH OUTPUT :

```
def classify_number(n):
    """
    Classifies a number as Positive, Negative, or Zero.
    Handles invalid inputs like strings and None.
    Uses a loop for checking conditions (though not strictly necessary for this simple logic,
    demonstrating loop usage as requested).
    """
    # Using a loop to check conditions, as requested.
    # In a real-world scenario, simple if/elif/else would be more direct.
    conditions = [
        (lambda x: isinstance(x, (int, float)) and x > 0, "Positive"),
        (lambda x: isinstance(x, (int, float)) and x < 0, "Negative"),
        (lambda x: isinstance(x, (int, float)) and x == 0, "Zero"),
    ]

    for condition, result in conditions:
        if condition(n):
            return result

    # Handle invalid input
    return "Invalid Input"

# Assert test cases
assert classify_number(10) == "Positive"
assert classify_number(-5) == "Negative"
assert classify_number(0) == "Zero"
assert classify_number(1) == "Positive"      # Boundary condition
assert classify_number(-1) == "Negative"     # Boundary condition
assert classify_number("hello") == "Invalid Input" # Invalid input
assert classify_number(None) == "Invalid Input" # Invalid input
assert classify_number(3.14) == "Positive" # Float input
assert classify_number(-2.5) == "Negative" # Float input

print("All assert tests for classify_number passed!")
```

All assert tests for classify_number passed!

EXPLANATION OF THE CODE :

◆ Okay, I can help you with that. I'll generate a Python function `classify_number(n)` that classifies a number as positive, negative, or zero, handles invalid inputs, and uses a loop. I'll also include at least three assert test cases.

Great! The `classify_number` function is working correctly and all the assert tests passed.

Since we have successfully implemented and tested the `classify_number` function, here are a few suggestions for what we could do next:

- We could add more diverse test cases to ensure the function handles a wider range of inputs, perhaps including edge cases with very large or very small floating-point numbers.
- We could explore alternative ways to implement the classification logic without using a loop, comparing the code complexity and readability.
- We could integrate this function into a larger program that requires classifying numbers based on user input or data processing.
- If you have another task in mind, feel free to tell me!

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:

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

Example Assert Test Cases:

```
assert is_anagram("listen", "silent") == True  
assert is_anagram("hello", "world") == False  
assert is_anagram("Dormitory", "Dirty Room") == True
```

Expected Output #3:

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

EXPECTED CODE WITH OUTPUT :

The screenshot shows a code editor window with Python code for an anagram checker. The code includes a docstring, a helper function `clean_string`, and logic to check if two strings are anagrams after cleaning. It also contains a series of assert statements and a final print statement.

```
def is_anagram(str1, str2):  
    """  
    Checks if two strings are anagrams, ignoring case, spaces, and punctuation.  
    Handles edge cases like empty strings and identical words.  
    """  
  
    def clean_string(s):  
        """Helper function to remove spaces, punctuation, and convert to lowercase."""  
        s = s.lower()  
        s = s.replace(" ", "")  
        s = s.translate(str.maketrans('', '', string.punctuation))  
        return s  
  
    cleaned_str1 = clean_string(str1)  
    cleaned_str2 = clean_string(str2)  
  
    # Anagrams must have the same length after cleaning  
    if len(cleaned_str1) != len(cleaned_str2):  
        return False  
  
    # Check if the sorted characters of both cleaned strings are the same  
    return sorted(cleaned_str1) == sorted(cleaned_str2)  
  
# Assert test cases  
assert is_anagram("listen", "silent") == True  
assert is_anagram("hello", "world") == False  
assert is_anagram(["Dormitory", "Dirty Room"]) == True  
assert is_anagram("", "") == True # Edge case: empty strings  
assert is_anagram("a", "a") == True # Edge case: identical single character words  
assert is_anagram("AaBb", "bBaA") == True # Case ignored  
assert is_anagram("Hello World!", "world hello") == True # Spaces and punctuation ignored  
assert is_anagram("restful", "fluster") == True # Another valid anagram  
assert is_anagram("The eyes", "They see") == True # Anagram with spaces  
  
print("All assert tests for is_anagram passed!")
```

All assert tests for is_anagram passed!

EXPLANATION OF THE CODE :

- ◆ Certainly! I can create the `is_anagram` function for you, along with assert test cases that cover your requirements, including ignoring case, spaces, and punctuation, and handling edge cases. After generating the code, I will provide a detailed explanation.

Great! The `is_anagram` function is working correctly, and all the assert tests passed.

Now, let's go over the code explanation as you requested:

This Python code defines a function `is_anagram` that determines if two strings are anagrams of each other, while ignoring case, spaces, and punctuation.

1. `import string`: This line imports the `string` module, which provides a collection of string constants, including `string.punctuation` which is used here.
2. `def is_anagram(str1, str2)`: This defines the main function `is_anagram` which takes two string arguments, `str1` and `str2`, to be compared.
3. `def clean_string(s)`: This defines a helper function `clean_string` inside `is_anagram`. This function is responsible for preparing a string for the anagram check by removing characters and normalizing case.
 - o `s = s.lower()`: Converts the input string `s` to lowercase. This ensures that the comparison is case-insensitive (e.g., "Listen" and "silent" are treated the same).

- o `s = s.replace(" ", "")`: Removes all space characters from the string. Spaces are ignored in the anagram check.
- o `s = s.translate(str.maketrans('', '', string.punctuation))`: This is a more efficient way to remove multiple characters from a string. `string.punctuation` is a string containing all common punctuation characters. `str.maketrans('', '', string.punctuation)` creates a translation table that maps each punctuation character to `None`, effectively removing them when `translate` is called.

4. `cleaned_str1 = clean_string(str1)`: Calls the `clean_string` helper function to process the first input string.
5. `cleaned_str2 = clean_string(str2)`: Calls the `clean_string` helper function to process the second input string.
6. `if len(cleaned_str1) != len(cleaned_str2)`: After cleaning, if the lengths of the two strings are different, they cannot be anagrams. The function immediately returns `False` in this case.

7. `return sorted(cleaned_str1) == sorted(cleaned_str2)`: This is the core logic for checking if the cleaned strings are anagrams.

- `sorted(cleaned_str1)` converts the cleaned string into a list of its characters and then sorts that list alphabetically.
- `sorted(cleaned_str2)` does the same for the second cleaned string.
- If two strings are anagrams (ignoring the characters that were removed), their sorted character lists will be identical. The `==` operator compares these two sorted lists. If they are equal, the function returns `True`; otherwise, it returns `False`.

The code also includes several `assert` statements to test the `is_anagram` function with various inputs, including edge cases as requested. These asserts confirm that the function behaves as expected for valid anagrams, non-anagrams, empty strings, identical words, and strings with different casing, spaces, and punctuation.

Since we have successfully implemented and tested the `is_anagram` function and I've provided the explanation, the original request is fully satisfied.

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:

- `add_item(name, quantity)`
- `remove_item(name, quantity)`
- `get_stock(name)`

Example Assert Test Cases:

```
inv = Inventory()
inv.add_item("Pen", 10)
assert inv.get_stock("Pen") == 10
inv.remove_item("Pen", 5)
assert inv.get_stock("Pen") == 5
inv.add_item("Book", 3)
assert inv.get_stock("Book") == 3
```

Expected Output #4:

- Fully functional class passing all assertions

EXPECTED CODE WITH OUTPUT :

```

class Inventory:
    """
    A simple Inventory class to manage stock of items.
    """

    def __init__(self):
        """Initializes the inventory with an empty dictionary to store items."""
        self.items = {}

    def add_item(self, name, quantity):
        """Adds a specified quantity of an item to the inventory."""
        if not isinstance(name, str) or not name:
            print("Error: Item name must be a non-empty string.")
            return
        if not isinstance(quantity, int) or quantity <= 0:
            print("Error: Quantity must be a positive integer.")
            return

        if name in self.items:
            self.items[name] += quantity
        else:
            self.items[name] = quantity

    def remove_item(self, name, quantity):
        """Removes a specified quantity of an item from the inventory."""
        if not isinstance(name, str) or not name:
            print("Error: Item name must be a non-empty string.")
            return
        if not isinstance(quantity, int) or quantity <= 0:
            print("Error: Quantity must be a positive integer.")
            return

```

```

if name in self.items:
    if self.items[name] >= quantity:
        self.items[name] -= quantity
        if self.items[name] == 0:
            del self.items[name] # Remove item from dictionary if stock is zero
    else:
        print(f"Warning: Not enough stock of {name} to remove {quantity}. Current stock: {self.items[name]}")
else:
    print(f"Warning: Item '{name}' not found in inventory.")

def get_stock(self, name):
    """Returns the current stock quantity of an item."""
    if not isinstance(name, str) or not name:
        print("Error: Item name must be a non-empty string.")
        return None
    return self.items.get(name, 0) # Return 0 if item not found

# Assert test cases
inv = Inventory()

# Test adding items
inv.add_item("Pen", 10)
assert inv.get_stock("Pen") == 10

inv.add_item("Book", 3)
assert inv.get_stock("Book") == 3

inv.add_item("Pen", 5) # Add more pens
assert inv.get_stock("Pen") == 15

# Test removing items
inv.remove_item("Pen", 5)
assert inv.get_stock("Pen") == 10

```

```

inv.remove_item("Book", 3) # Remove all books
assert inv.get_stock("Book") == 0
assert "Book" not in inv.items # Check if item is removed when stock is zero

# Test removing more than available stock (should not change stock and print a warning)
inv.add_item("Eraser", 2)
inv.remove_item("Eraser", 5)
assert inv.get_stock("Eraser") == 2 # Stock should remain unchanged

# Test removing non-existent item (should print a warning)
inv.remove_item("Stapler", 1)
assert inv.get_stock("Stapler") == 0

# Test adding with invalid input
inv.add_item("", 10)
inv.add_item("Paper", -5)
assert inv.get_stock("Paper") == 0 # Paper should not be added

# Test removing with invalid input
inv.remove_item("", 5)
inv.remove_item("Pen", -2)
assert inv.get_stock("Pen") == 10 # Pen stock should not change

print("All assert tests for Inventory class passed!")

```

```

Warning: Not enough stock of Eraser to remove 5. Current stock: 2
Warning: Item 'Stapler' not found in inventory.
Error: Item name must be a non-empty string.
Error: Quantity must be a positive integer.
Error: Item name must be a non-empty string.
Error: Quantity must be a positive integer.
All assert tests for Inventory class passed!

```

EXPLANATION OF THE CODE :

This Python code defines a class named `Inventory` to simulate a simple inventory management system. It allows you to add, remove, and check the stock of different items.

1. `class Inventory:`: This line defines the beginning of the `Inventory` class.
2. `""" A simple Inventory class to manage stock of items. """`: This is a docstring that provides a brief description of the class's purpose.
3. `def __init__(self):`: This is the constructor method of the class. It's called automatically when you create a new `Inventory` object.
 - o `"""Initializes the inventory with an empty dictionary to store items."""`: This is the docstring for the `__init__` method.
 - o `self.items = {}`: This line initializes an empty dictionary called `items` as an instance variable of the `Inventory` object. This dictionary will store the items in the inventory, where the keys will be the item names (strings) and the values will be their quantities (integers).
4. `def add_item(self, name, quantity):`: This method is used to add a specified `quantity` of an item with a given `name` to the inventory.
 - o `"""Adds a specified quantity of an item to the inventory."""`: Docstring for the `add_item` method.

- o `if not isinstance(name, str) or not name:`: This checks if the `name` is not a string or if it's an empty string. If either condition is true, it prints an error message and returns, preventing invalid item names from being added.
- o `if not isinstance(quantity, int) or quantity <= 0:`: This checks if the `quantity` is not an integer or if it's not a positive integer. If either condition is true, it prints an error message and returns, preventing invalid quantities from being added.
- o `if name in self.items:`: This checks if the item `name` already exists as a key in the `self.items` dictionary.
 - `self.items[name] += quantity`: If the item exists, the `quantity` is added to the current stock of that item.
- o `else:`: If the item does not exist in the inventory.
 - `self.items[name] = quantity`: The item is added to the `self.items` dictionary with the specified `quantity` as its initial stock.

5. `def remove_item(self, name, quantity):`: This method is used to remove a specified `quantity` of an item with a given `name` from the inventory.

- o `"""Removes a specified quantity of an item from the inventory."""`: Docstring for the `remove_item` method.
- o `if not isinstance(name, str) or not name:`: Similar input validation as in `add_item` for the item name.
- o `if not isinstance(quantity, int) or quantity <= 0:`: Similar input validation as in `add_item` for the quantity.
- o `if name in self.items:`: Checks if the item exists in the inventory.
 - `if self.items[name] >= quantity:`: If the item exists, this checks if there is enough stock to remove the specified `quantity`.
 - `self.items[name] -= quantity`: If there is enough stock, the `quantity` is subtracted from the current stock.
 - `if self.items[name] == 0:`: After removing the quantity, this checks if the stock of the item has become zero.

- `(if self.items[name] == 0:)`: After removing the quantity, this checks if the stock of the item has become zero.
- `del self.items[name]`: If the stock is zero, the item is removed entirely from the `self.items` dictionary.
- `else:`: If there is not enough stock to remove.
 - `print(f"Warning: Not enough stock of {name} to remove {quantity}. Current stock: {self.items[name]}")`: A warning message is printed indicating insufficient stock.
 - `else:`: If the item does not exist in the inventory.
 - `print(f"Warning: Item '{name}' not found in inventory.")`: A warning message is printed indicating that the item was not found.

printed indicating that the item was not found.

6. `def get_stock(self, name):`: This method returns the current stock quantity of an item with the given `name`.
 - `"""Returns the current stock quantity of an item."""`: Docstring for the `get_stock` method.
 - `if not isinstance(name, str) or not name:`: Similar input validation for the item name.
 - `return self.items.get(name, 0)`: This line uses the dictionary's `get()` method.
 - `self.items.get(name, 0)` attempts to retrieve the value associated with the key `name` from the `self.items` dictionary.
 - If the `name` is found, its corresponding value (the stock quantity) is returned.
 - If the `name` is *not* found, the `get()` method returns the default value specified, which is `0` in this case. This is a convenient way to handle requests for items that are not in the inventory without raising a `KeyError`.

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:

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

Example Assert Test Cases:

```
assert validate_and_format_date("10/15/2023") == "2023-10-15"
```

```
assert validate_and_format_date("02/30/2023") == "Invalid Date"
```

```
assert validate_and_format_date("01/01/2024") == "2024-01-01"
```

Expected Output #5:

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

EXPECTED CODE AND OUTPUT :

```
▶ 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".
    Handles invalid dates by returning "Invalid Date".
    """
    try:
        # Attempt to parse the date string with the expected format
        date_object = datetime.strptime(date_str, "%m/%d/%Y")
        # If parsing is successful, format the date to "YYYY-MM-DD"
        return date_object.strftime("%Y-%m-%d")
    except ValueError:
        # If parsing fails (due to incorrect format or invalid date), return "Invalid Date"
        return "Invalid Date"

# Assert test cases
assert validate_and_format_date("10/15/2023") == "2023-10-15"
assert validate_and_format_date("02/30/2023") == "Invalid Date" # Invalid day for February
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("04/31/2023") == "Invalid Date" # Invalid day for April
assert validate_and_format_date("13/01/2023") == "Invalid Date" # Invalid month
assert validate_and_format_date("10-15-2023") == "Invalid Date" # Incorrect format
assert validate_and_format_date("October 15, 2023") == "Invalid Date" # Incorrect format
assert validate_and_format_date("") == "Invalid Date" # Empty string

print("All assert tests for validate_and_format_date passed!")
```

→ All assert tests for validate_and_format_date passed!

EXPLANATION OF THE CODE :

◆ Alright, I can generate the Python code for the `validate_and_format_date` function with the specified requirements, including handling invalid dates and providing assert test cases. After generating the code, I will provide a detailed explanation.

Excellent! The `validate_and_format_date` function is working correctly, and all the assert tests passed.

Now, let's go over the explanation of the code:

This Python code defines a function `validate_and_format_date` that takes a date string in "MM/DD/YYYY" format, validates it, and if valid, converts it to "YYYY-MM-DD" format. It's designed to handle invalid date formats or values gracefully.

1. `from datetime import datetime`: This line imports the `datetime` class from the built-in `datetime` module. The `datetime` class is used to work with dates and times.
2. `def validate_and_format_date(date_str):`: This defines the function `validate_and_format_date` that takes one argument, `date_str`, which is the string containing the date to be validated and formatted.
3. `""" ... """`: This is the docstring for the function, explaining what it does, its parameters, and what it returns.
4. `try:`: This block starts a `try...except` block, which is used to handle potential errors that might occur during the execution of the code within the `try` block. This is crucial for handling invalid date strings.

5. `date_object = datetime.strptime(date_str, "%m/%d/%Y")`: This is the key line for validation and initial parsing.

- `datetime.strptime()` is a class method of `datetime` that parses a string (`date_str`) into a `datetime` object based on a specified format code (`"%m/%d/%Y"`).
- `"%m/%d/%Y"` is the format code that tells `strptime` to expect the date string in the "Month/Day/Year" format (e.g., "10/15/2023").
- If the `date_str` *exactly* matches this format and represents a *valid* date (e.g., you can't have "02/30/2023" as February doesn't have 30 days), `strptime` successfully creates a `datetime` object and assigns it to the `date_object` variable.
- If the `date_str` does not match the format or represents an invalid date, `strptime` raises a `ValueError`.

6. `return date_object.strftime("%Y-%m-%d")`: This line is executed only if the `strftime` call in the `try` block was successful (meaning the date string was valid and in the correct format).

- `date_object.strftime("%Y-%m-%d")` is a method of the `datetime` object that formats the date into a string according to a specified format code.
- `"%Y-%m-%d"` is the format code that tells `strftime` to format the date as "Year-Month-Day" (e.g., "2023-10-15").
- The formatted date string is then returned by the function.

7. `except ValueError:` : This block is executed if a `ValueError` is raised within the `try` block (which happens when `strftime` fails due to an invalid date string).

- `return "Invalid Date"`: If a `ValueError` occurs, the function catches the exception and returns the string "Invalid Date", indicating that the input date string was not valid or not in the expected format.