

AI ASSISTED CODING

Assignment-8.1

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

Code:

```
fibonacci.py > generate_strong_password
1 #generate python code for strong password generator Password must have at least 8 characters, Must include uppercase, lowercase, digit, and
2 import random
3 import string
4 def generate_strong_password(length=12):
5     """Generates a strong password.
6
7     Args:
8         length (int): The length of the password. Default is 12.
9
10    Returns:
11        str: A strong password that meets the specified criteria.
12    """
13    if length < 8:
14        raise ValueError("Password must be at least 8 characters long.")
15
16    # Define character sets
17    uppercase = string.ascii_uppercase
18    lowercase = string.ascii_lowercase
19    digits = string.digits
20    special_characters = string.punctuation
21
22    # Ensure the password includes at least one character from each set
23    password = [
24        random.choice(uppercase),
25        random.choice(lowercase),
26        random.choice(digits),
27        random.choice(special_characters)
28    ]
29
30    # Fill the remaining length with a mix of all character sets
31    all_characters = uppercase + lowercase + digits + special_characters
32    password += random.choices(all_characters, k=length - 4)
```

```

4  def generate_strong_password(length=12):
28     ]
29
30     # Fill the remaining length with a mix of all character sets
31     all_characters = uppercase + lowercase + digits + special_characters
32     password += random.choices(all_characters, k=length - 4)
33
34     # Shuffle the list to ensure randomness and convert to a string
35     random.shuffle(password)
36     return ''.join(password)
37
38 def main():
39     print(generate_strong_password(12))
40
41 if __name__ == "__main__":
42     main()
43
44 #generate assert test cases for the above code
45
46 def test_generate_strong_password():
47     # Test that the generated password meets the criteria
48     password = generate_strong_password(12)
49     assert len(password) >= 8, "Password must be at least 8 characters long."
50     assert any(c.isupper() for c in password), "Password must include an uppercase letter."
51     assert any(c.islower() for c in password), "Password must include a lowercase letter."
52     assert any(c.isdigit() for c in password), "Password must include a digit."
53     assert any(c in string.punctuation for c in password), "Password must include a special character."
54     assert ' ' not in password, "Password must not contain spaces."
55
56 test_generate_strong_password()
57
58 print("All tests passed!")
59
60

```

Output:

```

(Xom8G<G03BL
All tests passed!
PS C:\Users\nithe\OneDrive\Documents\python learning> & c:/Users/nithe/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/nithe/OneDrive/Documents/python
learning/fibo.py"
8L{t@4#\\of
All tests passed!
PS C:\Users\nithe\OneDrive\Documents\python 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:

```

21.py > test_classify_numbers
1 # generate code for number classification with loop include Classify numbers as Positive, Negative, or zero.Handle invalid inputs like strings and None
2 numbers = [-1, 0, 1, 2.5, -3.7, "invalid", None]
3 for num in numbers:
4     if num is None:
5         print("None is not a valid number.")
6     elif not isinstance(num, (int, float)):
7         print(f"{num} is not a valid number.")
8     elif num > 0:
9         print(f"{num} is Positive.")
10    elif num < 0:
11        print(f"{num} is Negative.")
12    else:
13        print(f"{num} is Zero.")
14 #generate assert test cases for the above code
15 def test_classify_numbers():
16     test_cases = [
17         (-1, "Negative"),
18         (0, "Zero"),
19         (1, "Positive"),
20         (2.5, "Positive"),
21         (-3.7, "Negative"),
22         ("invalid", "invalid is not a valid number."),
23         (None, "None is not a valid number.")
24     ]
25     for num, expected in test_cases:
26         if num is None:
27             assert expected == "None is not a valid number.", f"Expected 'None is not a valid number.' but got '{expected}'"
28         elif not isinstance(num, (int, float)):
29             assert expected == f"{num} is not a valid number.", f"Expected '{num} is not a valid number.' but got '{expected}'"
30         elif num > 0:
31             assert expected == "Positive", f"Expected 'Positive' but got '{expected}'"
32         elif num < 0:
33             assert expected == "Negative", f"Expected 'Negative' but got '{expected}'"
34         else:
35             assert expected == "Zero", f"Expected 'Zero' but got '{expected}'"
36 test_classify_numbers()
37 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\python 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\python 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:

```

1 #generate a python code for inventory class that manages add_item(name, quantity), remove_item(name, quantity), get_stock(name)
2 class Inventory:
3     def __init__(self):
4         """Initializes the inventory with an empty dictionary."""
5         self.stock = {}
6     def add_item(self, name, quantity):
7         """Adds a specified quantity of an item to the inventory.
8
9         Args:
10             name (str): The name of the item.
11             quantity (int): The quantity to add.
12         """
13         if name in self.stock:
14             self.stock[name] += quantity
15         else:
16             self.stock[name] = quantity
17     def remove_item(self, name, quantity):
18         """Removes a specified quantity of an item from the inventory.
19
20         Args:
21             name (str): The name of the item.
22             quantity (int): The quantity to remove.
23         """
24         if name in self.stock:
25             self.stock[name] -= quantity
26             if self.stock[name] < 0:
27                 self.stock[name] = 0
28     def get_stock(self, name):
29         return self.stock.get(name, 0)
30 def main():
31     inventory = Inventory()
32     inventory.add_item("Apple", 10)
33     inventory.add_item("Banana", 20)
34     print(inventory.get_stock("Apple")) # 10
35     print(inventory.get_stock("Banana")) # 20
36     inventory.remove_item("Apple", 5)
37     print(inventory.get_stock("Apple")) # 5
38     inventory.remove_item("Banana", 25)
39     print(inventory.get_stock("Banana")) # 0
40 if __name__ == "__main__":
41     main()
42 #generate assert test cases for the above code
43 def test_inventory():
44     inventory = Inventory()
45     inventory.add_item("Apple", 10)
46     inventory.add_item("Banana", 20)
47     assert inventory.get_stock("Apple") == 10, "Expected stock of Apple to be 10."
48     assert inventory.get_stock("Banana") == 20, "Expected stock of Banana to be 20."
49     inventory.remove_item("Apple", 5)
50     assert inventory.get_stock("Apple") == 5, "Expected stock of Apple to be 5 after removal."
51     inventory.remove_item("Banana", 25)
52     assert inventory.get_stock("Banana") == 0, "Expected stock of Banana to be 0 after removal."
53     assert inventory.get_stock("Orange") == 0, "Expected stock of Orange to be 0 as it is not in inventory."
54 test_inventory()
55 print("All tests passed!")

```

Output:

```

10
20
5
0
All tests passed!
PS C:\Users\nithe\OneDrive\Documents\python 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:

```

2.1.py > ...
1 #generate a python code for Date validation and formatting class that manages Validate "MM/DD/YYYY" format,Handle invalid dates,convert valid dates
2 class DateValidator:
3     def __init__(self):
4         self.valid_date = None
5
6     def validate_date(self, date_string):
7         """Validates a date string in MM/DD/YYYY format."""
8         parts = date_string.split('/')
9         if len(parts) != 3:
10             return False
11         try:
12             month = int(parts[0])
13             day = int(parts[1])
14             year = int(parts[2])
15         except ValueError:
16             return False
17         if not (1 <= month <= 12):
18             return False
19         if not (1 <= day <= 31):
20             return False
21         if not (1000 <= year <= 9999):
22             return False
23         self.valid_date = (month, day, year)
24         return True
25
26     def format_date(self, date_string):
27         """Formats a valid date string to YYYY-MM-DD format."""
28         if self.validate_date(date_string):
29             month, day, year = self.valid_date
30             return f"{year:04d}-{month:02d}-{day:02d}"
31         else:
32             raise ValueError("Invalid date format or invalid date.")
33
34 # Example usage:
35 validator = DateValidator()
36 print(validator.validate_date("12/25/2023")) # True
37 print(validator.format_date("12/25/2023")) # 2023-12-25
38 print(validator.validate_date("13/45/2023")) # False

```

```

36 print(validator.format_date("12/25/2023")) # 2023-12-25
37 print(validator.validate_date("13/45/2023")) # False
38 try:
39     validator.format_date("13/45/2023")
40 except ValueError as e:
41     print(e) # Invalid date format or invalid date.
42 #generate assert test cases for the above code and handle edge cases
43 def test_date_validator():
44     validator = DateValidator()
45     # Valid date test
46     assert validator.validate_date("12/25/2023") == True, "Expected valid date to return True."
47     assert validator.format_date("12/25/2023") == "2023-12-25", "Expected formatted date to be '2023-12-25'."
48     # Invalid date tests
49     assert validator.validate_date("13/45/2023") == False, "Expected invalid month and day to return False."
50     assert validator.validate_date("02/30/2023") == False, "Expected invalid day for February to return False."
51     assert validator.validate_date("00/10/2023") == False, "Expected month 0 to return False."
52     assert validator.validate_date("10/00/2023") == False, "Expected day 0 to return False."
53     assert validator.validate_date("10/10/999") == False, "Expected year less than 1000 to return False."
54     assert validator.validate_date("10/10/10000") == False, "Expected year greater than 9999 to return False."
55     # Edge case: Non-numeric input
56     assert validator.validate_date("MM/DD/YYYY") == False, "Expected non-numeric input to return False."
57     assert validator.validate_date("12/AB/2023") == False, "Expected non-numeric day to return False."
58     assert validator.validate_date("AB/10/2023") == False, "Expected non-numeric month to return False."
59     assert validator.validate_date("12/10/ABCD") == False, "Expected non-numeric year to return False."
60     test_date_validator()
61     print("All tests passed!")
62
63

```

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\python 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.