Name: D.varshitha

Hall Ticket:2303A51927

Batch:30

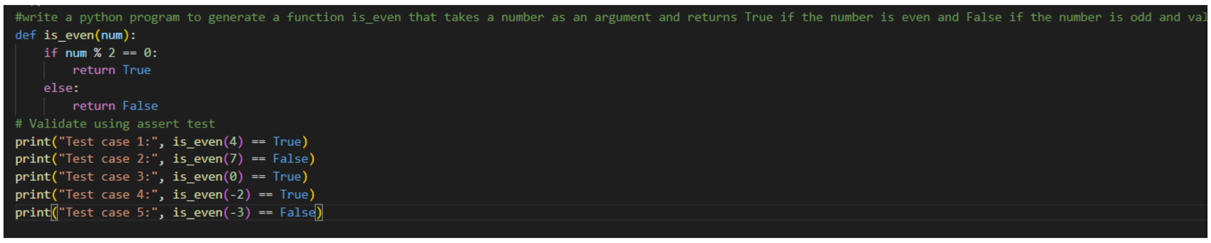
Assessment: 8.2

Task 1: Test-Driven Development for Even/Odd Number Validator

Prompt:

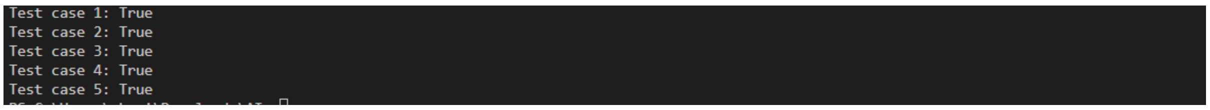
write a python program to generate a function is_even that takes a number as an argument and returns True if the number is even and False if the number is odd and validate using assest test

Code:



```python
#write a python program to generate a function is_even that takes a number as an argument and returns True if the number is even and False if the number is odd and val
def is_even(num):
    if num % 2 == 0:
        return True
    else:
        return False
# Validate using assert test
print("Test case 1:", is_even(4) == True)
print("Test case 2:", is_even(7) == False)
print("Test case 3:", is_even(0) == True)
print("Test case 4:", is_even(-2) == True)
print("Test case 5:", is_even(-3) == False)
```

Output:



```
Test case 1: True
Test case 2: True
Test case 3: True
Test case 4: True
Test case 5: True
```

Significance:

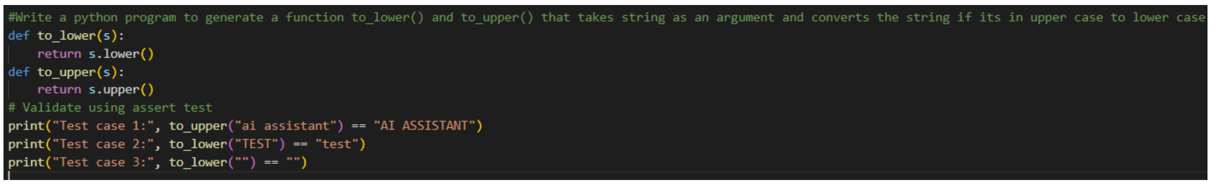
This task introduces **basic Test-Driven Development (TDD)** concepts. By generating test cases before writing the is_even() function, we learn how expected behavior guides implementation. It also emphasizes **input validation**, handling **zero, negative values, and large integers**, which improves code robustness. This task builds confidence in writing logic that strictly follows test expectations.

Task 2: Test-Driven Development for String Case Converter

Prompt:

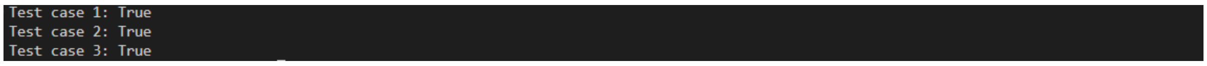
Write a python program to generate a function to_lower() and to_upper() that takes string as an argument and converts the string if its in upper case to lower case and if its in lower case to upper case and validate using assert test

Code:



```python
#Write a python program to generate a function to_lower() and to_upper() that takes string as an argument and converts the string if its in upper case to lower case
def to_lower(s):
    return s.lower()
def to_upper(s):
    return s.upper()
# Validate using assert test
print("Test case 1:", to_upper("ai assistant") == "AI ASSISTANT")
print("Test case 2:", to_lower("TEST") == "test")
print("Test case 3:", to_lower("") == "")
```

Output:



```
Test case 1: True
Test case 2: True
Test case 3: True
```

Significance:

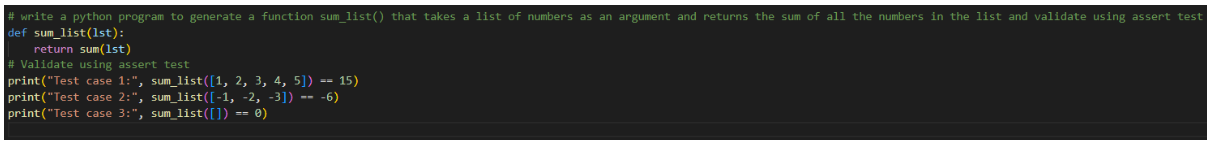
This task focuses on **string manipulation with safe error handling**. Writing tests first helps identify edge cases such as empty strings and invalid inputs (None, numbers). It teaches defensive programming and shows how TDD ensures functions behave correctly even with unexpected data.

Task 3: Test-Driven Development for List Sum Calculator

Prompt:

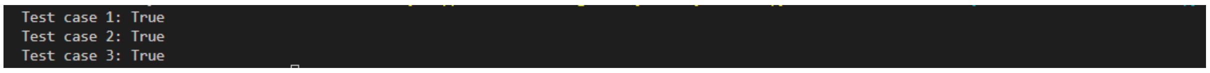
write a python program to generate a function sum_list() that takes a list of numbers as an argument and returns the sum of all the numbers in the list and validate using assert test

Code:



```python
# write a python program to generate a function sum_list() that takes a list of numbers as an argument and returns the sum of all the numbers in the list and validate using assert test
def sum_list(lst):
    return sum(lst)
# Validate using assert test
print("Test case 1:", sum_list([1, 2, 3, 4, 5]) == 15)
print("Test case 2:", sum_list([-1, -2, -3]) == -6)
print("Test case 3:", sum_list([]) == 0)
```

Output:



```
Test case 1: True
Test case 2: True
Test case 3: True
```

significance:

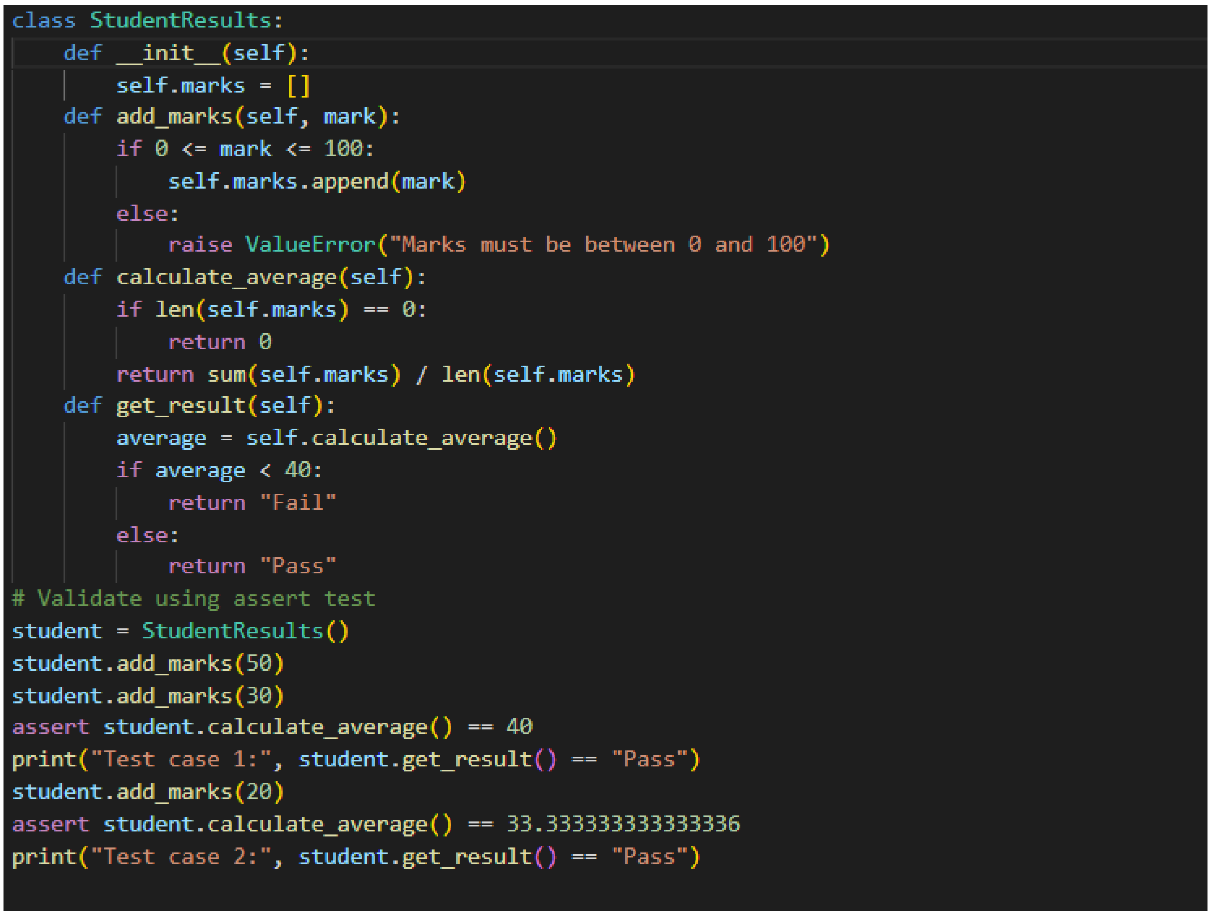
This task demonstrates how TDD helps design **fault-tolerant data processing functions**. Test cases guide the implementation to ignore non-numeric values while correctly summing valid numbers. It reinforces handling **empty lists and negative values**, which are common real-world scenarios.

Task 4: Test Cases for Student Result Class

Prompt:

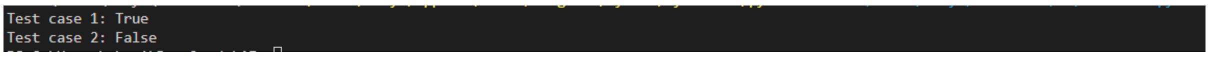
generate a class StudentResults() with the following methods add_marks(mark),calculate_average(),get_result() and marks must be between 0 and 100 ,if calculate_average()<40 then get_result() should return "Fail" otherwise it should return "Pass" and validate using assert test

Code:



```python
class StudentResults:
    def __init__(self):
        self.marks = []
    def add_marks(self, mark):
        if 0 <= mark <= 100:
            self.marks.append(mark)
        else:
            raise ValueError("Marks must be between 0 and 100")
    def calculate_average(self):
        if len(self.marks) == 0:
            return 0
        return sum(self.marks) / len(self.marks)
    def get_result(self):
        average = self.calculate_average()
        if average < 40:
            return "Fail"
        else:
            return "Pass"
# Validate using assert test
student = StudentResults()
student.add_marks(50)
student.add_marks(30)
assert student.calculate_average() == 40
print("Test case 1:", student.get_result() == "Pass")
student.add_marks(20)
assert student.calculate_average() == 33.333333333333336
print("Test case 2:", student.get_result() == "Pass")
```

Output:



```
Test case 1: True
Test case 2: False
```

Significance:

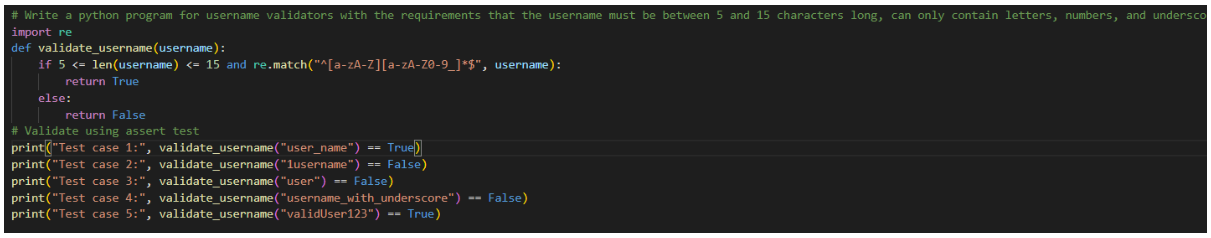
This task introduces **class-based TDD**. Writing tests for methods like add_marks(), calculate_average(), and get_result() ensures proper object behavior. It highlights validation rules, exception handling, and business logic (Pass/Fail), showing how TDD supports structured and reliable class design.

Task 5: Test-Driven Development for Username Validator

Prompt:

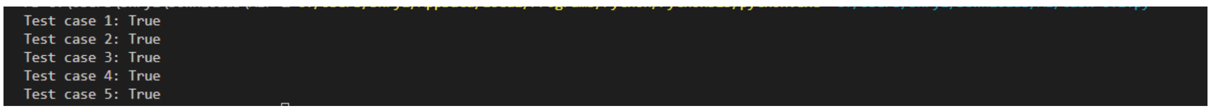
Write a python program for username validators with the requirements that the username must be between 5 and 15 characters long, can only contain letters, numbers, and underscores, and must start with a letter. Validate using assert test

Code:



```python
# Write a python program for username validators with the requirements that the username must be between 5 and 15 characters long, can only contain letters, numbers, and undersco
import re
def validate_username(username):
    if 5 <= len(username) <= 15 and re.match("^[a-zA-Z][a-zA-Z0-9_]*$", username):
        return True
    else:
        return False
# Validate using assert test
print("Test case 1:", validate_username("user_name") == True)
print("Test case 2:", validate_username("1username") == False)
print("Test case 3:", validate_username("user") == False)
print("Test case 4:", validate_username("username_with_underscore") == False)
print("Test case 5:", validate_username("validUser123") == True)
```

Output:



```
Test case 1: True
Test case 2: True
Test case 3: True
Test case 4: True
Test case 5: True
```

Significance:

This task applies TDD to **input validation and security basics**. Tests enforce rules like minimum length, no spaces, and alphanumeric characters. It demonstrates how TDD prevents invalid user input early and helps build safer applications.