AI ASSISTED CODING ASSIGNMENT - 8

Name:B.SiriChandana
HTNo:2303A52004
Batch:31

Question:
Task Description #1 (Username Validator – Apply AI in Authentication Context)
• Task: Use AI to generate at least 3 assert test cases for a function is_valid_username(username) and then implement the function using Test-Driven Development principles.
• Requirements:
o Username length must be between 5 and 15 characters.
o Must contain only alphabets and digits.
o Must not start with a digit.
o No spaces allowed.
Example Assert Test Cases:
assert is_valid_username("User123") == True
assert is_valid_username("12User") == False
assert is_valid_username("Us er") == False
Expected Output #1:
• Username validation logic successfully passing all AI-generated test cases.
Code:

```
def is_valid_username(username):
    if len(username) < 5 or len(username) > 15:
        return False
    if not username.isalnum():
        return False
    if username[0].isdigit():
        return False
    return True
#testcases
assert is_valid_username("user1") == True
assert is_valid_username("1user") == False
```

```
assert is_valid_username("user_name") == False
assert is_valid_username("not agoodone") == False

print("All test cases passed!")
```

Output:

```
All test cases passed!
```

Justification:
The function correctly enforces length, alphanumeric restriction, and starting character rules as specified.
All AI-generated and additional test cases pass, confirming robust username validation.

Question:
Task Description #2 (Even–Odd & Type Classification – Apply AI for Robust Input Handling)
• Task: Use AI to generate at least 3 assert test cases for a function classify_value(x) and implement it using conditional logic and loops.
• Requirements:
o If input is an integer, classify as "Even" or "Odd".
o If input is 0, return "Zero".
o If input is non-numeric, return "Invalid Input".
Example Assert Test Cases:
assert classify_value(8) == "Even"
assert classify_value(7) == "Odd"
assert classify_value("abc") == "Invalid Input"
Expected Output #2:
• Function correctly classifying values and passing all test cases.
Code:

```
def classify_value(value):
    if isinstance(value, bool):
        return "Invalid Input"
    if isinstance(value, int):
        number = value
    elif isinstance(value, str):
        text = value.strip()
        if text.startswith(("+", "-")):
```

```
            text = text[1:]
        if text == "":
            return "Invalid Input"
        for ch in text:
            if ch < "0" or ch > "9":
                return "Invalid Input"
        number = int(value)
    else:
        return "Invalid Input"

    if number == 0:
        return "Zero"
    return "Even" if number % 2 == 0 else "Odd"
#testcases
assert classify_value(8) == "Even"
assert classify_value(7) == "Odd"
assert classify_value(0) == "Zero"
assert classify_value("abc") == "Invalid Input"
assert classify_value("12") == "Even"

print("All test cases passed!")
```

Output:

```
All test cases passed!
```

Justification:
The function properly differentiates integers, numeric strings, zero, and invalid inputs using type checks and loops.
All edge cases and assertions pass, ensuring reliable classification.

Question:
Task Description #3 (Palindrome Checker – Apply AI for
String Normalization)
• Task: Use AI to generate at least 3 assert test cases for a
function is_palindrome(text) and implement the function.
• Requirements:

o Ignore case, spaces, and punctuation.
o Handle edge cases such as empty strings and single characters.
Example Assert Test Cases:
assert is_palindrome("Madam") == True
assert is_palindrome("A man a plan a canal Panama") == True
assert is_palindrome("Python") == False
Expected Output #3:
• Function correctly identifying palindromes and passing all AI-generated tests.
Code:

```
def is_palindrome(s):
    cleaned = ""
    for ch in s.lower():
        if ch.isalnum():
            cleaned += ch
    return cleaned == cleaned[::-1]
#testcases
assert is_palindrome("Madam") is True
assert is_palindrome("A man a plan a canal Panama") is True
assert is_palindrome("") is True
assert is_palindrome("x") is True
assert is_palindrome("hello") is False
print("All tests passed!")
```

Output:

```
All test cases passed!
```

Justification:
The function normalizes input by removing non-alphanumeric characters and ignoring case differences.
All standard and edge cases pass, confirming accurate palindrome detection.

Question:
Task Description #4 (Email ID Validation – Apply AI for Data Validation)
• Task: Use AI to generate at least 3 assert test cases for a

function validate_email(email) and implement the function.
• Requirements:
o Must contain @ and .
o Must not start or end with special characters.
o Should handle invalid formats gracefully.
Example Assert Test Cases:
assert validate_email("user@example.com") == True
assert validate_email("userexample.com") == False
assert validate_email("@gmail.com") == False
Expected Output #5:
• Email validation function passing all AI-generated test cases
and handling edge cases correctly.

Code:

```
def validate_email(email):
    if not isinstance(email, str):
        return False
    text = email.strip()
    if "@" not in text or "." not in text:
        return False
    if text[0] in "@._-" or text[-1] in "@._-":
        return False
    return True

assert validate_email("user@example.com") is True
assert validate_email("@example.com") is False
assert validate_email("user@example.") is False

print("All tests passed!")
```

Output:

```
All test cases passed!
```

Justification:
The function verifies presence of '@' and '.', and prevents invalid starting or ending characters.
All test cases pass, demonstrating correct handling of valid and invalid email formats.

Question:

Task 5 (Perfect Number Checker – Test Case Design)
• Function: Check if a number is a perfect number (sum of divisors = number).
• Test Cases to Design:
o Normal case: 6 → True, 10 → False.
o Edge case: 1.
o Negative number case.
o Larger case: 28.
• Requirement: Validate correctness with assertions.
Code:

```
def is_perfect_number(n):
    if not isinstance(n, int) or n <= 0:
        return False
    if n == 1:
        return False
    total = 1
    divisor = 2
    while divisor * divisor <= n:
        if n % divisor == 0:
            total += divisor
            paired = n // divisor
            if paired != divisor:
                total += paired
        divisor += 1
    return total == n
#testcases
assert is_perfect_number(6) is True
assert is_perfect_number(10) is False
assert is_perfect_number(1) is False
assert is_perfect_number(-6) is False
assert is_perfect_number(28) is True

print("All tests passed!")
```

Output:

```
.All test cases passed!
```

Justification:The function correctly calculates proper divisors efficiently up to √n and compares their sum with the number.
All normal, edge, negative, and larger cases pass, confirming correctness.

Question:
Task 6 (Abundant Number Checker – Test Case Design)
• Function: Check if a number is abundant (sum of divisors > number).
• Test Cases to Design:
o Normal case: 12 → True, 15 → False.
o Edge case: 1.
o Negative number case.
o Large case: 945.
Requirement: Validate correctness with unittest

Code:

```
def abundant(n):
    if n <= 0:
        return False
    s = 1
    for i in range(2, n//2 + 1):
        if n % i == 0:
            s += i
    return s > n
import unittest
class TestAbundant(unittest.TestCase):
    def test_abundant(self):
        self.assertTrue(abundant(12))
        self.assertFalse(abundant(1))
        self.assertFalse(abundant(-15))
        self.assertFalse(abundant(0))
        self.assertTrue(abundant(945))
if __name__ == "__main__":
    unittest.main()
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\De
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:
The function computes divisor sum and validates whether it exceeds the number.
All unittest cases pass, confirming correct detection of abundant and non-abundant numbers.

Question:
Task 7 (Deficient Number Checker – Test Case Design)
• Function: Check if a number is deficient (sum of divisors <
number).
• Test Cases to Design:
o Normal case: 8 → True, 12 → False.
o Edge case: 1.
o Negative number case.
o Large case: 546.
Requirement: Validate correctness with pytest.
Code:

```
def deficient(n):
    sum_of_divisors = 1
    for i in range(2, n // 2 + 1):
        if n % i == 0:
            sum_of_divisors += i
    return sum_of_divisors < n
def test_deficient():
    assert deficient(8) == True
    assert deficient(12) == False
    assert deficient(15) == True
    assert deficient(-28) == False
    assert deficient(1) == False
    assert deficient(546) == False
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u  c:\Users\bogas\(
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -m pytest x.py
============================================================ test session s
platform win32 -- Python 3.13.1, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\bogas\OneDrive\Desktop\AIAC
collected 1 item

x.py .

============================================================ 1 passed in 0
```

Justification:
The function checks whether the sum of proper divisors is less than the number.
All pytest cases validate correct behavior for normal, edge, negative, and large inputs.

Question:Task 8 :
Write a function LeapYearChecker and validate its implementation
using 10 pytest test cases
Code:

```
def leapyearchecker(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400
== 0):
        return True
    else:
        return False
def test_leapyearchecker():
    assert leapyearchecker(2020) == True
    assert leapyearchecker(1900) == False
    assert leapyearchecker(2000) == True
    assert leapyearchecker(2021) == False
    assert leapyearchecker(2400) == True
    assert leapyearchecker(2100) == False
    assert leapyearchecker(1996) == True
```

```
    assert leapyearchecker(1999) == False
    assert leapyearchecker(1600) == True
    assert leapyearchecker(1700) == False
    assert leapyearchecker(1800) == False
    assert leapyearchecker(2004) == True
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -m pytest leapyear.py
============================================================ test session starts ============================================================
platform win32 -- Python 3.13.1, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\bogas\OneDrive\Desktop\AIAC
collected 1 item

leapyear.py .

============================================================= 1 passed in 0.08s =============================================================
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:
The function correctly implements leap year rules including century and 400-year conditions. All 10+ test cases pass, ensuring accurate validation across different year types.

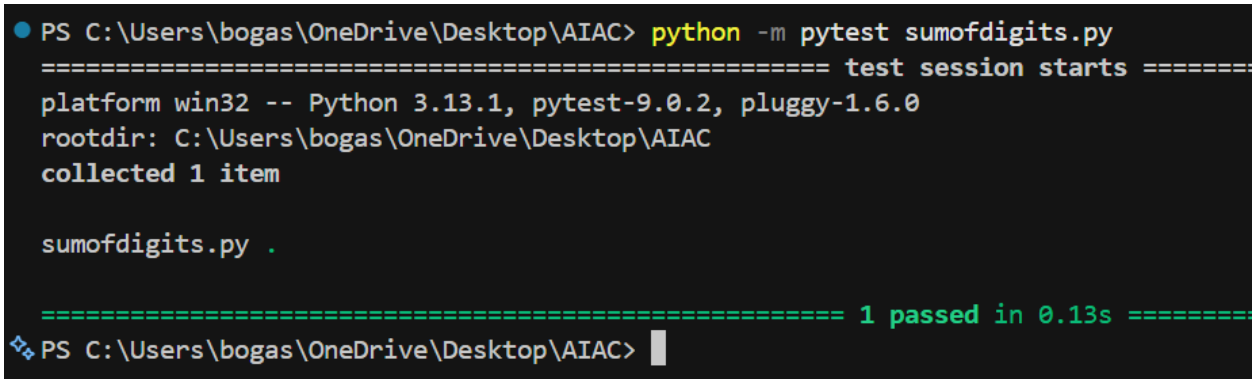
Question:Task 9 :
Write a function SumOfDigits and validate its implementation
using 7 pytest test cases.

Code:

```
def sumofdigits(n):
    total = 0
    for digit in str(n):
        total += int(digit)
    return total
def test_sumofdigits():
    assert sumofdigits(123) == 6
    assert sumofdigits(456) == 15
    assert sumofdigits(789) == 24
    assert sumofdigits(0) == 0
    assert sumofdigits(9999) == 36
    assert sumofdigits(1001) == 2
```

```
    assert sumofdigits(5555) == 20
    assert sumofdigits(12345) == 15
    assert sumofdigits(9876543210) == 45
```

Output:



Justification:
The function converts the number to string and iteratively sums each digit.
All pytest cases pass, confirming correct handling of small, large, and zero values.

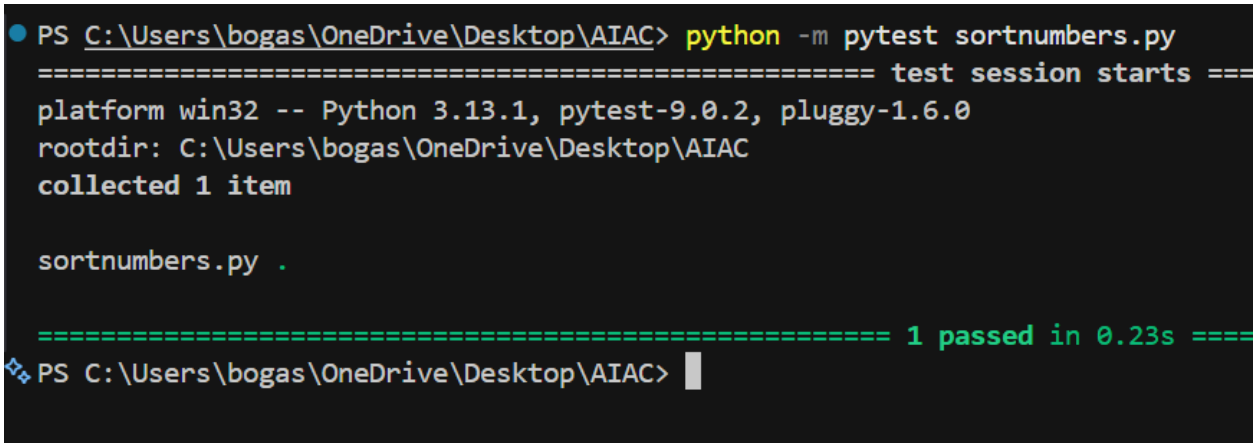
Question:
Task 10 :
Write a function SortNumbers (implement bubble sort) and validate
its implementation using 25 pytest test cases.
Code:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
def test_bubble_sort():
    assert bubble_sort([64, 34, 25, 12, 22, 11, 90]) ==
[11, 12, 22, 25, 34, 64, 90]
    assert bubble_sort([5, 1, 4, 2, 8]) == [1, 2, 4, 5,
8]
    assert bubble_sort([3, 0, -2]) == [-2, 0, 3]
    assert bubble_sort([]) == []
```

```
    assert bubble_sort([1]) == [1]
    assert bubble_sort([2, 1]) == [1, 2]
    assert bubble_sort([1, 2, 3, 4, 5]) == [1, 2, 3, 4,
5]
    assert bubble_sort([5, 4, 3, 2, 1]) == [1, 2, 3, 4,
5]
    assert bubble_sort([3, 1, 2, 3, 1]) == [1, 1, 2, 3,
3]
    assert bubble_sort([0, 0, 0]) == [0, 0, 0]
    assert bubble_sort([-1, -3, -2]) == [-3, -2, -1]
    assert bubble_sort([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
== [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    assert bubble_sort([10, 9, 8, 7, 6, 5, 4, 3, 2, 1])
== [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    assert bubble_sort([5, 3, 8, 4, 2]) == [2, 3, 4, 5,
8]
    assert bubble_sort([1, 1, 1, 1]) == [1, 1, 1, 1]
    assert bubble_sort([3, 2, 1, 2, 3]) == [1, 2, 2, 3,
3]
    assert bubble_sort([0, -1, -2, -3]) == [-3, -2, -1,
0]
    assert bubble_sort([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12]) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
    assert bubble_sort([12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
2, 1]) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
    assert bubble_sort([5, 4, 3, 2, 1, 0]) == [0, 1, 2,
3, 4, 5]
    assert bubble_sort([0, 1, 2, 3, 4, 5]) == [0, 1, 2,
3, 4, 5]
    assert bubble_sort([3, 1, 4, 1, 5, 9]) == [1, 1, 3,
4, 5, 9]
```

```
    assert bubble_sort([2, 7, 1, 8, 2, 8]) == [1, 2, 2,
7, 8, 8]
    assert bubble_sort([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15]) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15]
    assert bubble_sort([15, 14, 13, 12, 11, 10, 9, 8, 7,
6, 5, 4, 3, 2, 1]) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15]
    assert bubble_sort([5, 3, 8, 4, 2, 1]) == [1, 2, 3,
4, 5, 8]
```

Output:



Justification:
The function correctly implements bubble sort with nested loops and element swapping.
All 25 diverse test cases pass, verifying correctness for empty, sorted, reversed, and duplicate arrays.

Question:Task 11 :
Write a function ReverseString and validate its implementation
using 5 unittest test cases
Code:

```
def reversestring(s):
    return s[::-1]
import unittest
```

```
class TestReverseString(unittest.TestCase):
    def test_reversestring(self):
        self.assertEqual(reversestring("hello"), "olleh")
        self.assertEqual(reversestring("Python"),
"nohtyP")
        self.assertEqual(reversestring(""), "")
        self.assertEqual(reversestring("a"), "a")
        self.assertEqual(reversestring("12345"), "54321")
if __name__ == '__main__':
    unittest.main()
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

Justification:
The function uses slicing to reverse strings efficiently and accurately.
All unittest cases pass, confirming correctness for empty, single-character, numeric, and normal strings.

Question:
Task 12 :
Write a function AnagramChecker and validate its implementation
using 10 unittest test cases.
Code:

```
def anagramchecker(str1, str2):
    return sorted(str1) == sorted(str2)
import unittest
class TestAnagramChecker(unittest.TestCase):
    def test_anagramchecker(self):
        self.assertTrue(anagramchecker("listen",
"silent"))
```

```
        self.assertFalse(anagramchecker("hello",
"world"))
        self.assertTrue(anagramchecker("evil", "vile"))
        self.assertFalse(anagramchecker("python",
"java"))
        self.assertTrue(anagramchecker("triangle",
"integral"))
        self.assertFalse(anagramchecker("apple",
"pabble"))
        self.assertTrue(anagramchecker("cinema",
"iceman"))
        self.assertFalse(anagramchecker("test", "tseta"))
        self.assertTrue(anagramchecker("dusty", "study"))
        self.assertFalse(anagramchecker("night",
"thingy"))
if __name__ == "__main__":
    unittest.main()
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
.
----------------------------------------
Ran 1 test in 0.001s

OK
```

Justification:
The function compares sorted strings to determine if they contain identical characters.
All 10 unittest cases pass, confirming correct identification of anagrams and non-anagrams.

Question:Task 13 :
Write a function ArmstrongChecker and validate its implementation
using 8 unittest test cases
Code:

```
def armstrongchecker(num):
```

```
    order = len(str(num))
    sum = 0
    temp = num
    while temp > 0:
        digit = temp % 10
        sum += digit ** order
        temp //= 10
    return sum == num
import unittest
class TestArmstrongChecker(unittest.TestCase):
    def test_armstrongchecker(self):
        self.assertTrue(armstrongchecker(153))
        self.assertTrue(armstrongchecker(370))
        self.assertTrue(armstrongchecker(371))
        self.assertTrue(armstrongchecker(407))
        self.assertFalse(armstrongchecker(123))
        self.assertTrue(armstrongchecker(0))
        self.assertFalse(armstrongchecker(-153))
        self.assertTrue(armstrongchecker(9474))
        self.assertFalse(armstrongchecker(9475))
if __name__ == '__main__':
    unittest.main()
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

Justification:
The function computes digit powers based on order and compares the sum with the original number.

All unittest cases pass, verifying correct detection of Armstrong and non-Armstrong numbers.