AI ASSISTED CODING

LAB ASSIGNMENT-8

A.Srichala

2303A52435

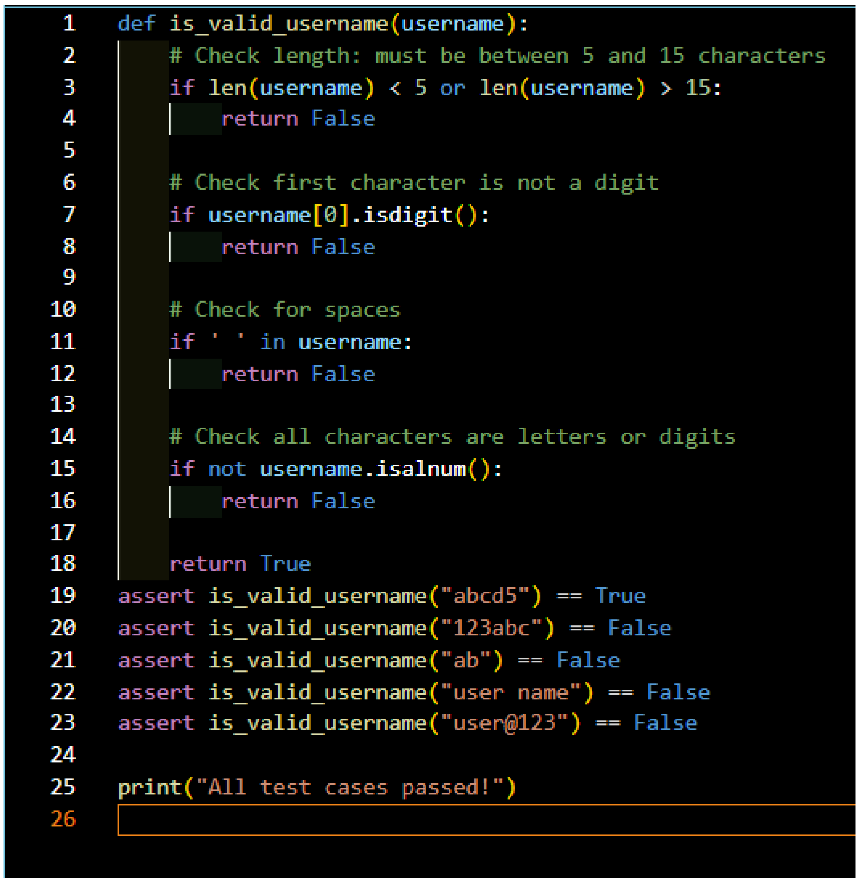
BT-31

CSE-AIML

18-Feb-2026

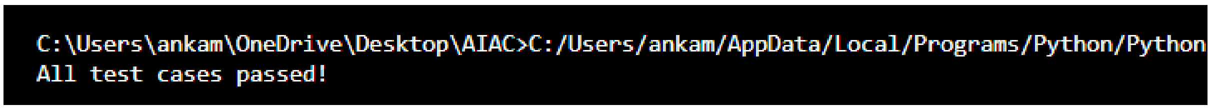
Task Description #1 (Username Validator – Apply AI in
Authentication Context)
• Task: Use AI to generate at least 3 assert test cases for a
function is_valid_username(username) and then implement
the function using Test-Driven Development principles.
• Requirements:
o Username length must be between 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.

The function is_valid_username() checks whether a username follows the required rules. It ensures the username is between 5 and 15 characters, starts with a letter, contains no spaces, and includes only letters and digits. If any condition fails, it returns False; otherwise, it returns True.



```python
def is_valid_username(username):
    # Check length: must be between 5 and 15 characters
    if len(username) < 5 or len(username) > 15:
        return False

    # Check first character is not a digit
    if username[0].isdigit():
        return False

    # Check for spaces
    if ' ' in username:
        return False

    # Check all characters are letters or digits
    if not username.isalnum():
        return False

    return True
assert is_valid_username("abcd5") == True
assert is_valid_username("123abc") == False
assert is_valid_username("ab") == False
assert is_valid_username("user name") == False
assert is_valid_username("user@123") == False

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

Output:-



```
C:\Users\ankam\OneDrive\Desktop\AIAC>C:/Users/ankam/AppData/Local/Programs/Python/Python
All test cases passed!
```

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"

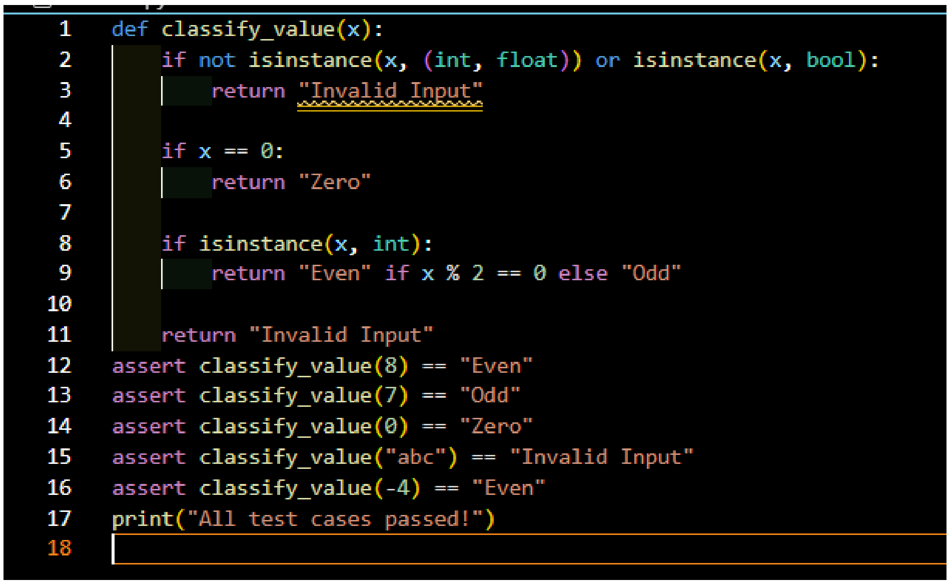
assert classify_value(7) == "Odd"

assert classify_value("abc") == "Invalid Input"

Expected Output #2:

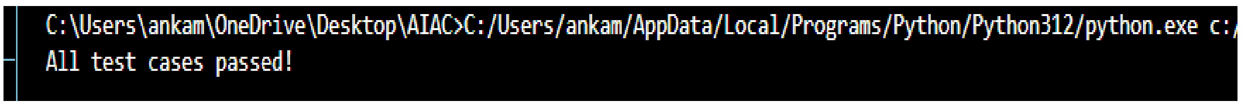
• Function correctly classifying values and passing all test cases.

The function classify_value(x) checks the type and value of the input. If the input is not a number, it returns "Invalid Input". If the input is 0, it returns "Zero". If the input is an integer, it checks whether the number is even or odd and returns "Even" or "Odd" accordingly.



```python
def classify_value(x):
    if not isinstance(x, (int, float)) or isinstance(x, bool):
        return "Invalid Input"

    if x == 0:
        return "Zero"

    if isinstance(x, int):
        return "Even" if x % 2 == 0 else "Odd"

    return "Invalid Input"
assert classify_value(8) == "Even"
assert classify_value(7) == "Odd"
assert classify_value(0) == "Zero"
assert classify_value("abc") == "Invalid Input"
assert classify_value(-4) == "Even"
print("All test cases passed!")

```

Output:-



```
C:\Users\ankam\OneDrive\Desktop\AIAC>C:/Users/ankam/AppData/Local/Programs/Python/Python312/python.exe c:
All test cases passed!
```

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

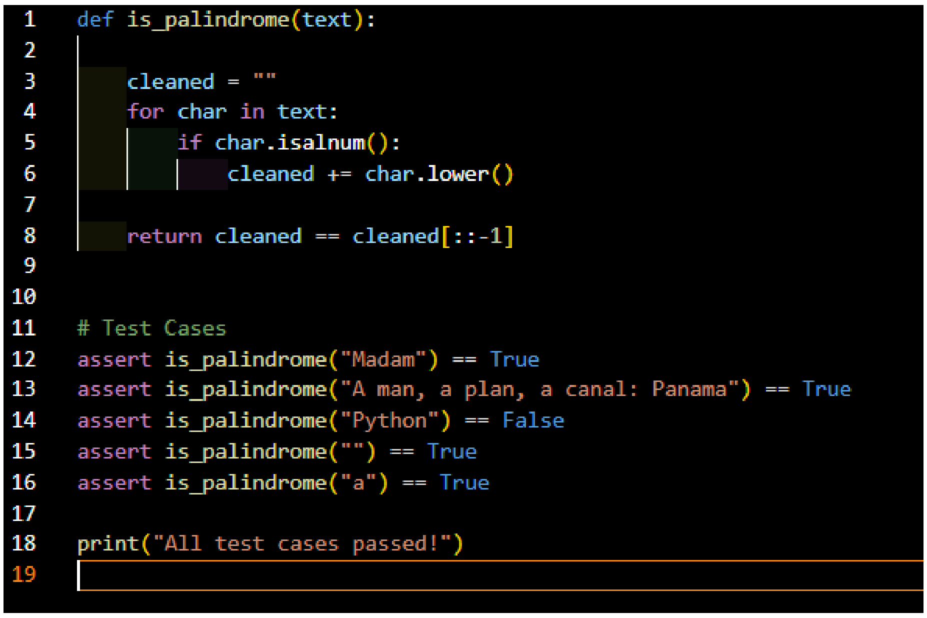
assert is_palindrome("A man a plan a canal Panama") == True

assert is_palindrome("Python") == False

Expected Output #3:

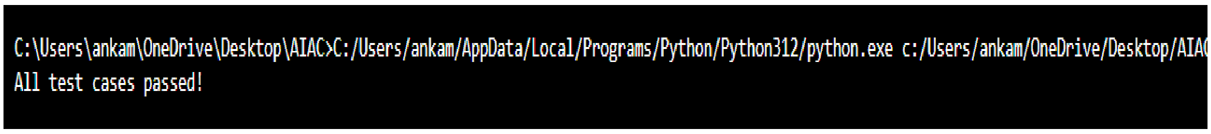
• Function correctly identifying palindromes and passing all AI-generated tests.

Write a Python program for a function is_palindrome(text) using the Test-Driven Development (TDD) approach. The function should check whether a given string is a palindrome while ignoring case, spaces, and punctuation. It must also handle edge cases such as empty strings and single characters. First define the function, then generate at least five assert-based test cases to validate different scenarios including palindromes and non-palindromes. Ensure the program runs without errors.



```python
def is_palindrome(text):

    cleaned = ""
    for char in text:
        if char.isalnum():
            cleaned += char.lower()

    return cleaned == cleaned[::-1]


# Test Cases
assert is_palindrome("Madam") == True
assert is_palindrome("A man, a plan, a canal: Panama") == True
assert is_palindrome("Python") == False
assert is_palindrome("") == True
assert is_palindrome("a") == True

print("All test cases passed!")

```

Output:-



```
C:\Users\ankam\OneDrive\Desktop\AIAC>C:/Users/ankam/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ankam/OneDrive/Desktop/AIAC
All test cases passed!
```

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

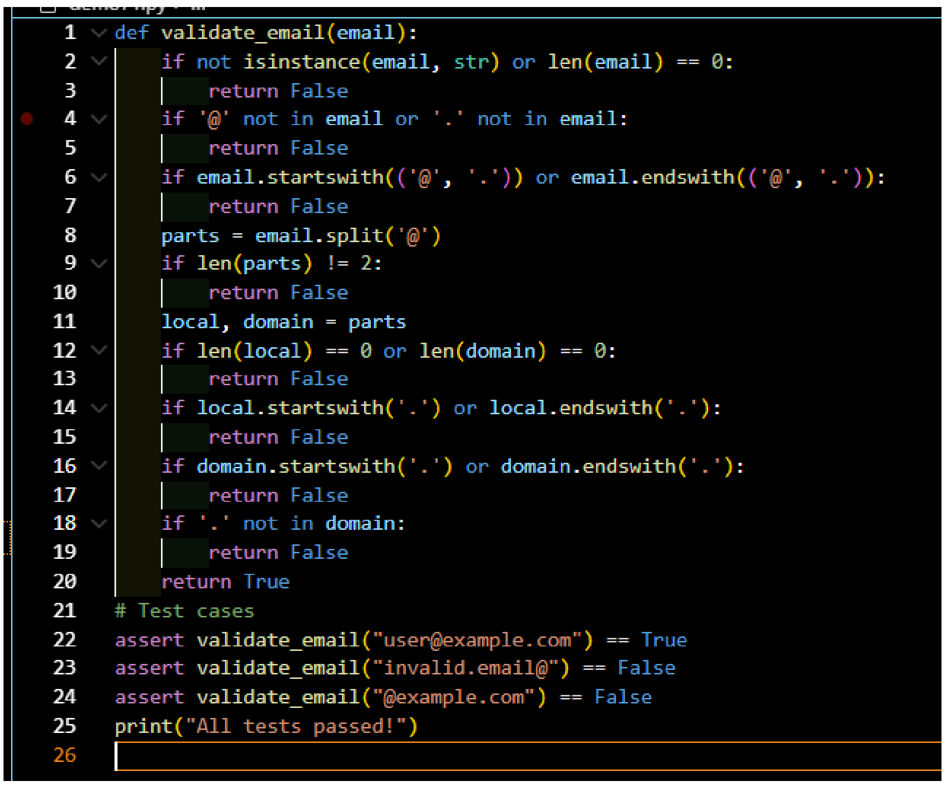
assert validate_email("userexample.com") == False

assert validate_email("@gmail.com") == False

Expected Output #5:

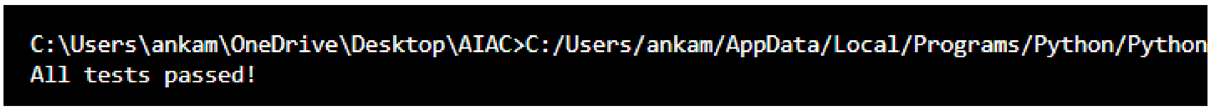
• Email validation function passing all AI-generated test cases and handling edge cases correctly.

Write a Python program for a function validate_email(email) using the Test-Driven Development (TDD) approach. The function validate_email(email) checks whether a given email address follows proper formatting rules. The email must contain both "@" and "." symbols. It must not start or end with special characters, and it should handle invalid formats gracefully by returning False. First define the function, then generate at least five assert-based test cases covering valid and invalid email formats. Ensure the program runs correctly without errors.



```python
def validate_email(email):
    if not isinstance(email, str) or len(email) == 0:
        return False
    if '@' not in email or '.' not in email:
        return False
    if email.startswith(('@', '.')) or email.endswith(('@', '.')):
        return False
    parts = email.split('@')
    if len(parts) != 2:
        return False
    local, domain = parts
    if len(local) == 0 or len(domain) == 0:
        return False
    if local.startswith('.') or local.endswith('.'):
        return False
    if domain.startswith('.') or domain.endswith('.'):
        return False
    if '.' not in domain:
        return False
    return True
# Test cases
assert validate_email("user@example.com") == True
assert validate_email("invalid.email@") == False
assert validate_email("@example.com") == False
print("All tests passed!")
```

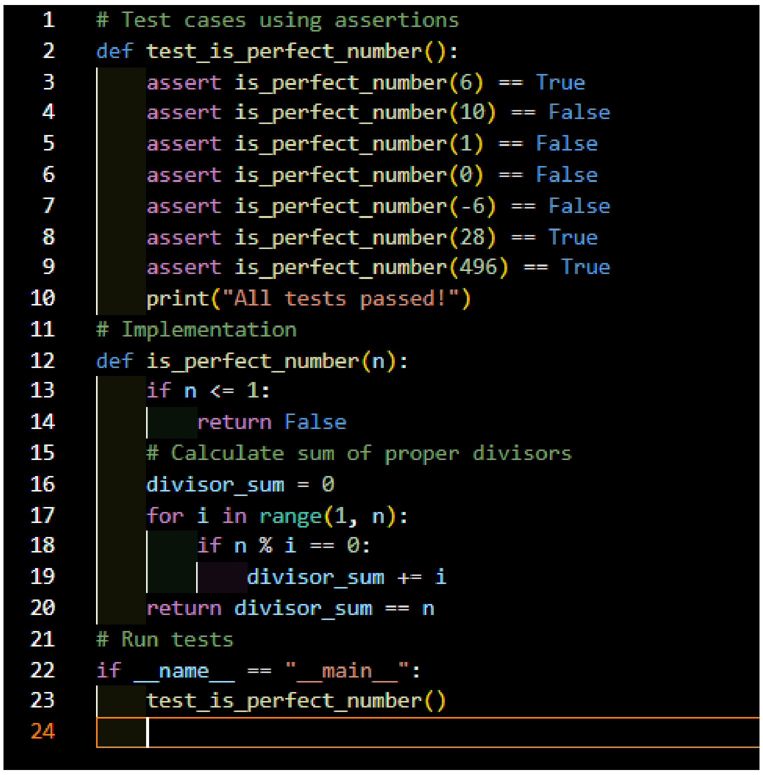
Output:



```
C:\Users\ankam\OneDrive\Desktop\AIAC>C:/Users/ankam/AppData/Local/Programs/Python/Python
All tests passed!
```

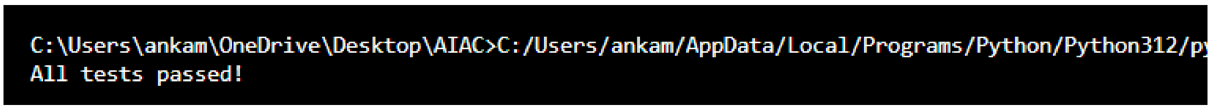
Task 5 (Perfect Number Checker – Test Case Design)
• Function: Check if a number is a perfect number (sum of divisors = number).
• Test Cases to Design:
o Normal case: 6 → True, 10 → False.
o Edge case: 1.
o Negative number case.
o Larger case: 28.
• Requirement: Validate correctness with assertions.

Write a Python program for a function is_perfect_number(n) using the Test-Driven Development (TDD) approach. The function should check whether a given number is a perfect number, meaning the sum of its proper divisors is equal to the number itself. First design assert-based test cases covering normal cases such as 6 (True) and 10 (False), an edge case like 1, a negative number case, and a larger number such as 28. Then implement the function so that it correctly passes all the assertions and validates the logic properly.



```python
# Test cases using assertions
def test_is_perfect_number():
    assert is_perfect_number(6) == True
    assert is_perfect_number(10) == False
    assert is_perfect_number(1) == False
    assert is_perfect_number(0) == False
    assert is_perfect_number(-6) == False
    assert is_perfect_number(28) == True
    assert is_perfect_number(496) == True
    print("All tests passed!")
# Implementation
def is_perfect_number(n):
    if n <= 1:
        return False
    # Calculate sum of proper divisors
    divisor_sum = 0
    for i in range(1, n):
        if n % i == 0:
            divisor_sum += i
    return divisor_sum == n
# Run tests
if __name__ == "__main__":
    test_is_perfect_number()

```

Output:



```
C:\Users\ankam\OneDrive\Desktop\AIAC>C:/Users/ankam/AppData/Local/Programs/Python/Python312/py
All tests passed!
```

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.

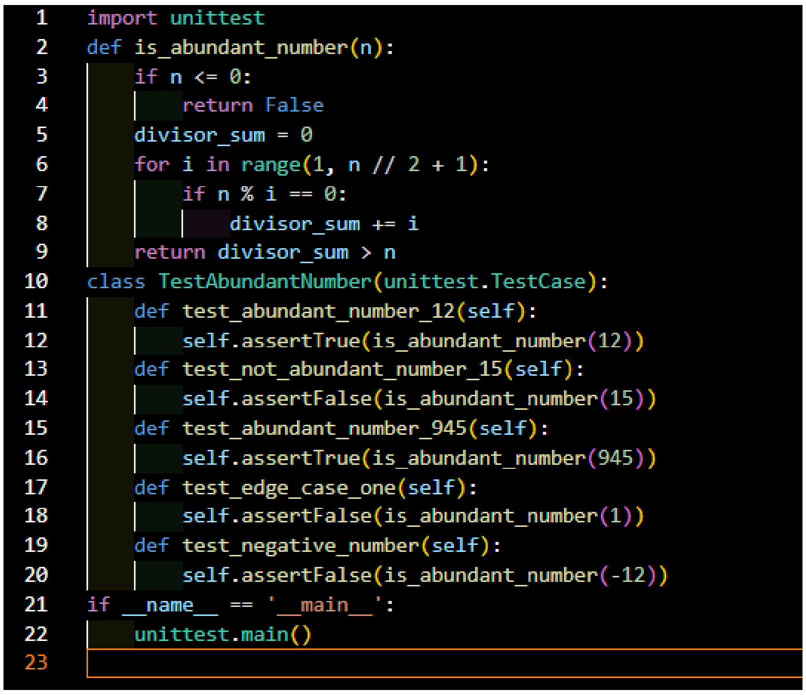
o Edge case: 1.

o Negative number case.

o Large case: 945.

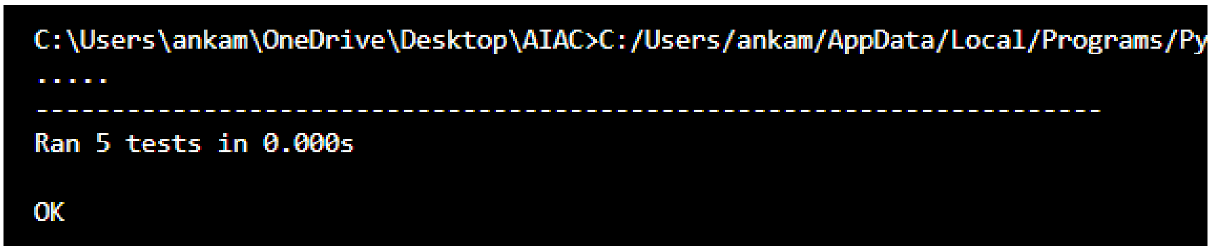
Requirement: Validate correctness with unittest

Write a Python program for a function is_abundant_number(n) using the Test-Driven Development (TDD) approach. The function should check whether a given number is an abundant number, meaning the sum of its proper divisors is greater than the number itself. Design test cases using the unittest framework, covering normal cases such as 12 (True) and 15 (False), an edge case like 1, a negative number case, and a larger number such as 945. Implement the function so that it passes all the unittest test cases correctly and validates the logic properly.



```
import unittest
def is_abundant_number(n):
    if n <= 0:
        return False
    divisor_sum = 0
    for i in range(1, n // 2 + 1):
        if n % i == 0:
            divisor_sum += i
    return divisor_sum > n
class TestAbundantNumber(unittest.TestCase):
    def test_abundant_number_12(self):
        self.assertTrue(is_abundant_number(12))
    def test_not_abundant_number_15(self):
        self.assertFalse(is_abundant_number(15))
    def test_abundant_number_945(self):
        self.assertTrue(is_abundant_number(945))
    def test_edge_case_one(self):
        self.assertFalse(is_abundant_number(1))
    def test_negative_number(self):
        self.assertFalse(is_abundant_number(-12))
if __name__ == '__main__':
    unittest.main()

```

Output:



```
C:\Users\ankam\OneDrive\Desktop\AIAC>C:/Users/ankam/AppData/Local/Programs/Py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

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.

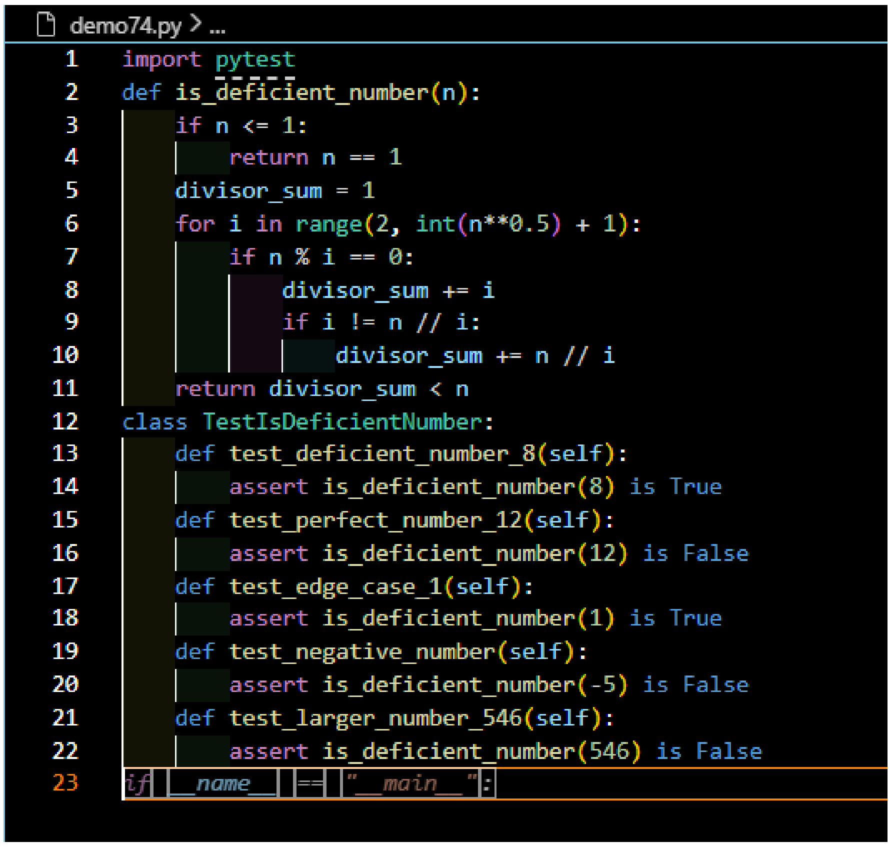
o Edge case: 1.

o Negative number case.

o Large case: 546.

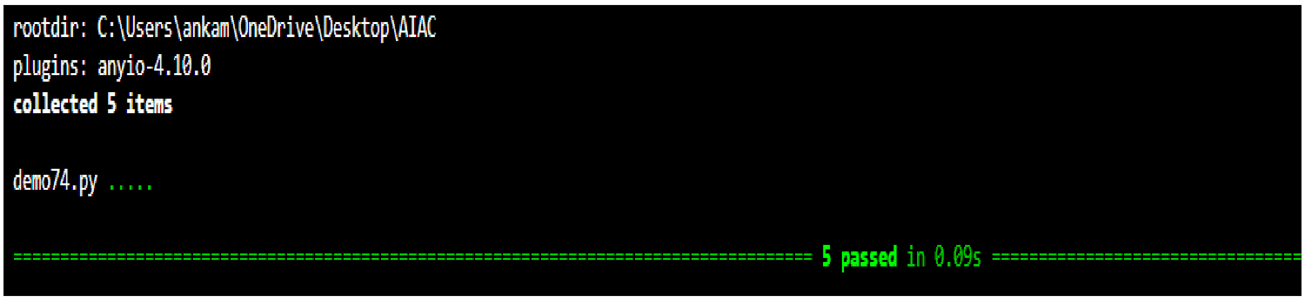
Requirement: Validate correctness with pytest.

Write a Python program for a function is_deficient_number(n) using the Test-Driven Development (TDD) approach. The function should check whether a given number is a deficient number, meaning the sum of its proper divisors is less than the number itself. Design test cases using the pytest framework, covering normal cases such as 8 (True) and 12 (False), an edge case like 1, a negative number case, and a larger number such as 546. Implement the function so that it correctly passes all pytest test cases and validates the logic properly.



```
demo74.py > ...
import pytest
def is_deficient_number(n):
    if n <= 1:
        return n == 1
    divisor_sum = 1
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            divisor_sum += i
            if i != n // i:
                divisor_sum += n // i
    return divisor_sum < n
class TestIsDeficientNumber:
    def test_deficient_number_8(self):
        assert is_deficient_number(8) is True
    def test_perfect_number_12(self):
        assert is_deficient_number(12) is False
    def test_edge_case_1(self):
        assert is_deficient_number(1) is True
    def test_negative_number(self):
        assert is_deficient_number(-5) is False
    def test_larger_number_546(self):
        assert is_deficient_number(546) is False
if __name__ == "__main__":
```

Output:-

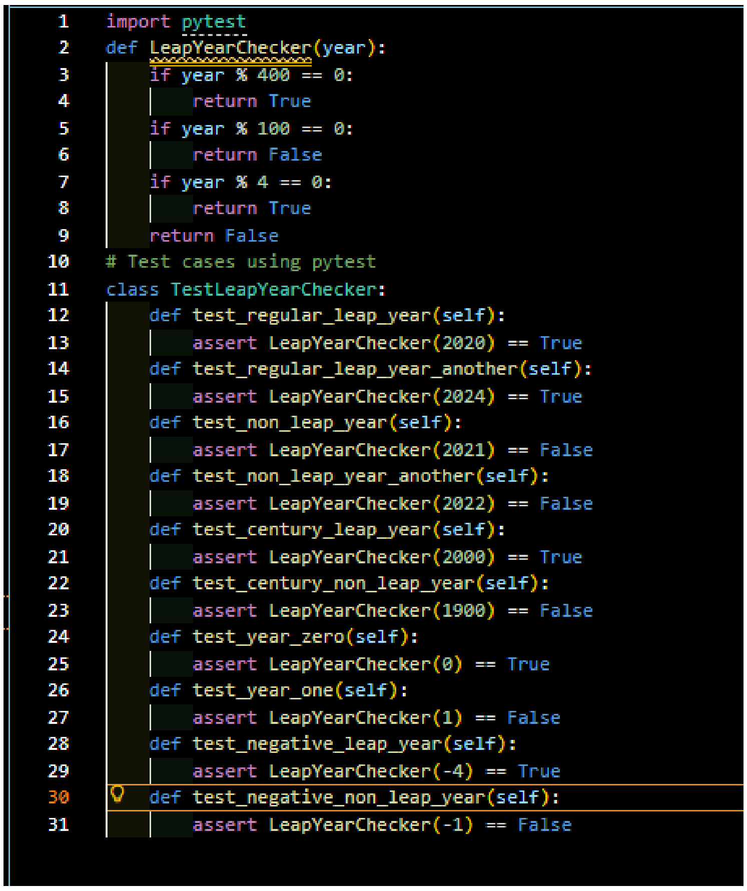


```
rootdir: C:\Users\ankam\OneDrive\Desktop\AIAC
plugins: anyio-4.10.0
collected 5 items

demo74.py .....

================================================================================= 5 passed in 0.09s ================================
```

Task 8 :

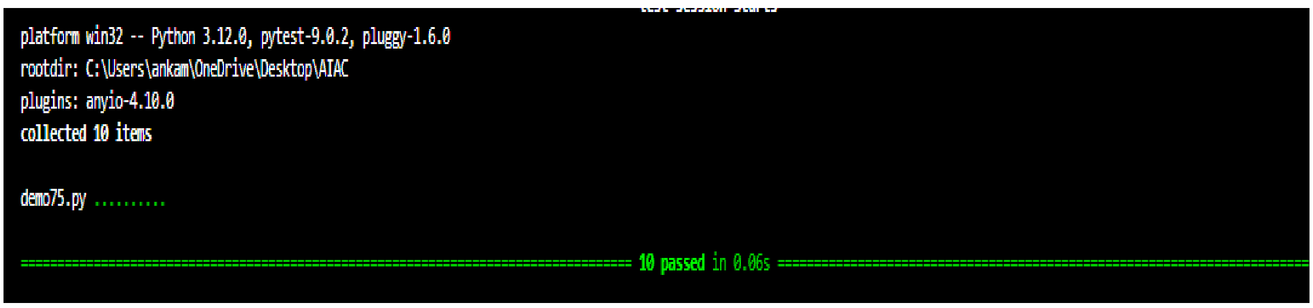
Write a function LeapYearChecker and validate its implementation using 10 pytest test cases

Write a Python program for a function LeapYearChecker(year) using the Test-Driven Development (TDD) approach. The function should determine whether a given year is a leap year. A year is a leap year if it is divisible by 4 but not divisible by 100, unless it is also divisible by 400. Use the pytest framework to create 10 test cases covering regular leap years, non-leap years, century years, edge cases such as year 0 and 1, and negative year values. Implement the function so that it passes all pytest test cases correctly and validates the logic properly.



```python
import pytest
def LeapYearChecker(year):
    if year % 400 == 0:
        return True
    if year % 100 == 0:
        return False
    if year % 4 == 0:
        return True
    return False
# Test cases using pytest
class TestLeapYearChecker:
    def test_regular_leap_year(self):
        assert LeapYearChecker(2020) == True
    def test_regular_leap_year_another(self):
        assert LeapYearChecker(2024) == True
    def test_non_leap_year(self):
        assert LeapYearChecker(2021) == False
    def test_non_leap_year_another(self):
        assert LeapYearChecker(2022) == False
    def test_century_leap_year(self):
        assert LeapYearChecker(2000) == True
    def test_century_non_leap_year(self):
        assert LeapYearChecker(1900) == False
    def test_year_zero(self):
        assert LeapYearChecker(0) == True
    def test_year_one(self):
        assert LeapYearChecker(1) == False
    def test_negative_leap_year(self):
        assert LeapYearChecker(-4) == True
    def test_negative_non_leap_year(self):
        assert LeapYearChecker(-1) == False
```

Output:-

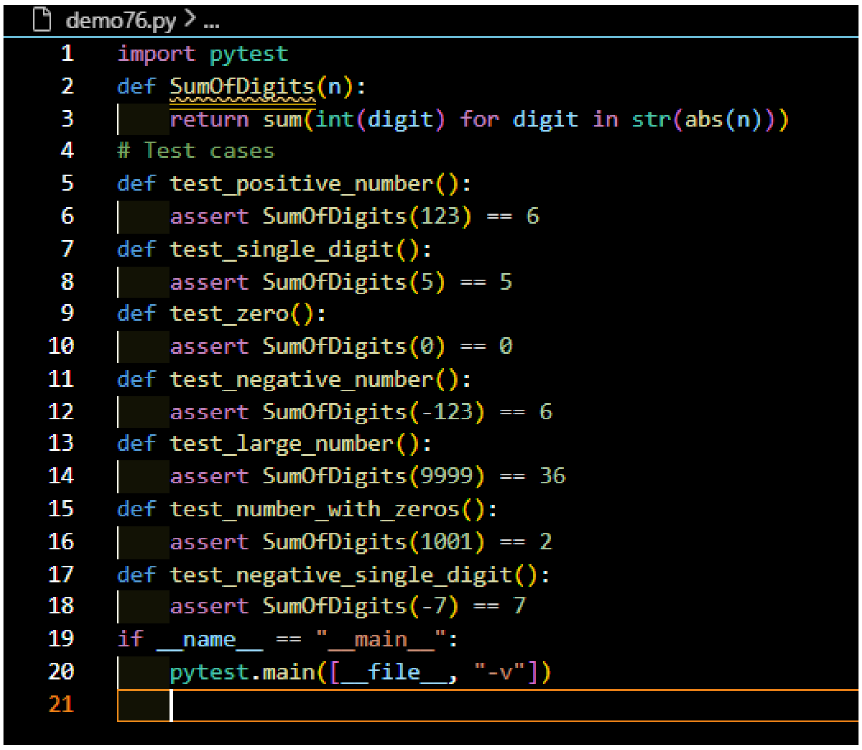


```
platform win32 -- Python 3.12.0, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\ankam\OneDrive\Desktop\AIAC
plugins: anyio-4.10.0
collected 10 items

demo75.py ..........

============================================================================= 10 passed in 0.06s ==================================================================
```

Task 9 :

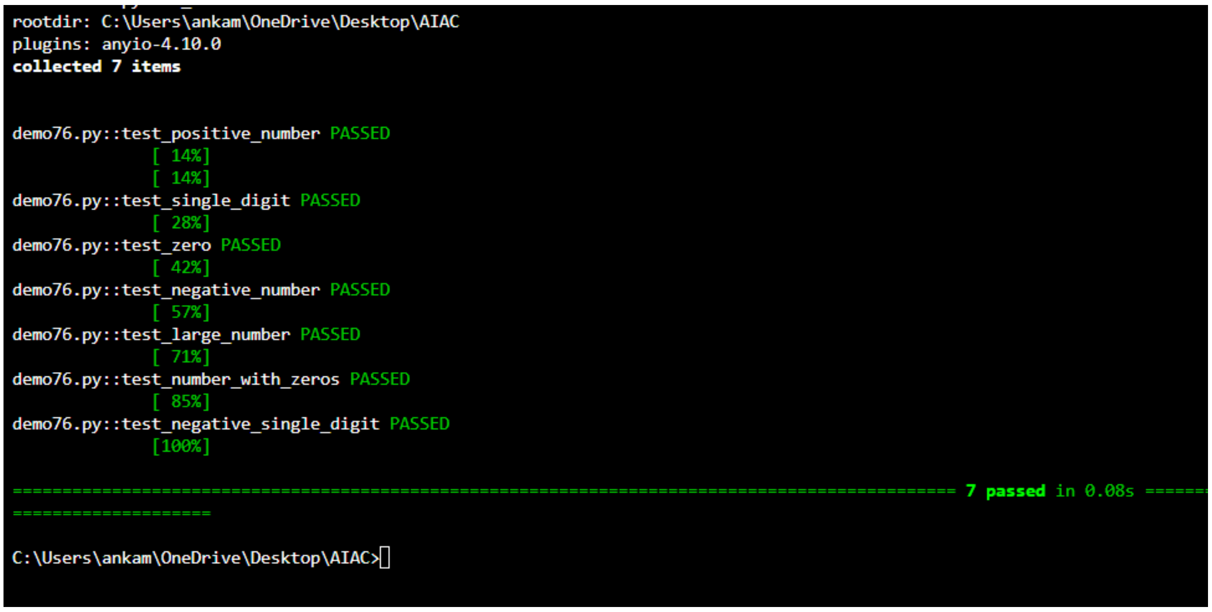
Write a function SumOfDigits and validate its implementation
using 7 pytest test cases.

Write a Python program for a function SumOfDigits(n) using the Test-Driven Development (TDD) approach. The function should calculate and return the sum of all digits in a given number. It must correctly handle positive numbers, negative numbers, and zero. Use the pytest framework to create 7 test cases covering normal inputs, edge cases, and negative values. Implement the function so that it passes all pytest test cases successfully.



```python
import pytest
def SumOfDigits(n):
    return sum(int(digit) for digit in str(abs(n)))
# Test cases
def test_positive_number():
    assert SumOfDigits(123) == 6
def test_single_digit():
    assert SumOfDigits(5) == 5
def test_zero():
    assert SumOfDigits(0) == 0
def test_negative_number():
    assert SumOfDigits(-123) == 6
def test_large_number():
    assert SumOfDigits(9999) == 36
def test_number_with_zeros():
    assert SumOfDigits(1001) == 2
def test_negative_single_digit():
    assert SumOfDigits(-7) == 7
if __name__ == "__main__":
    pytest.main([__file__, "-v"])

```

Output:

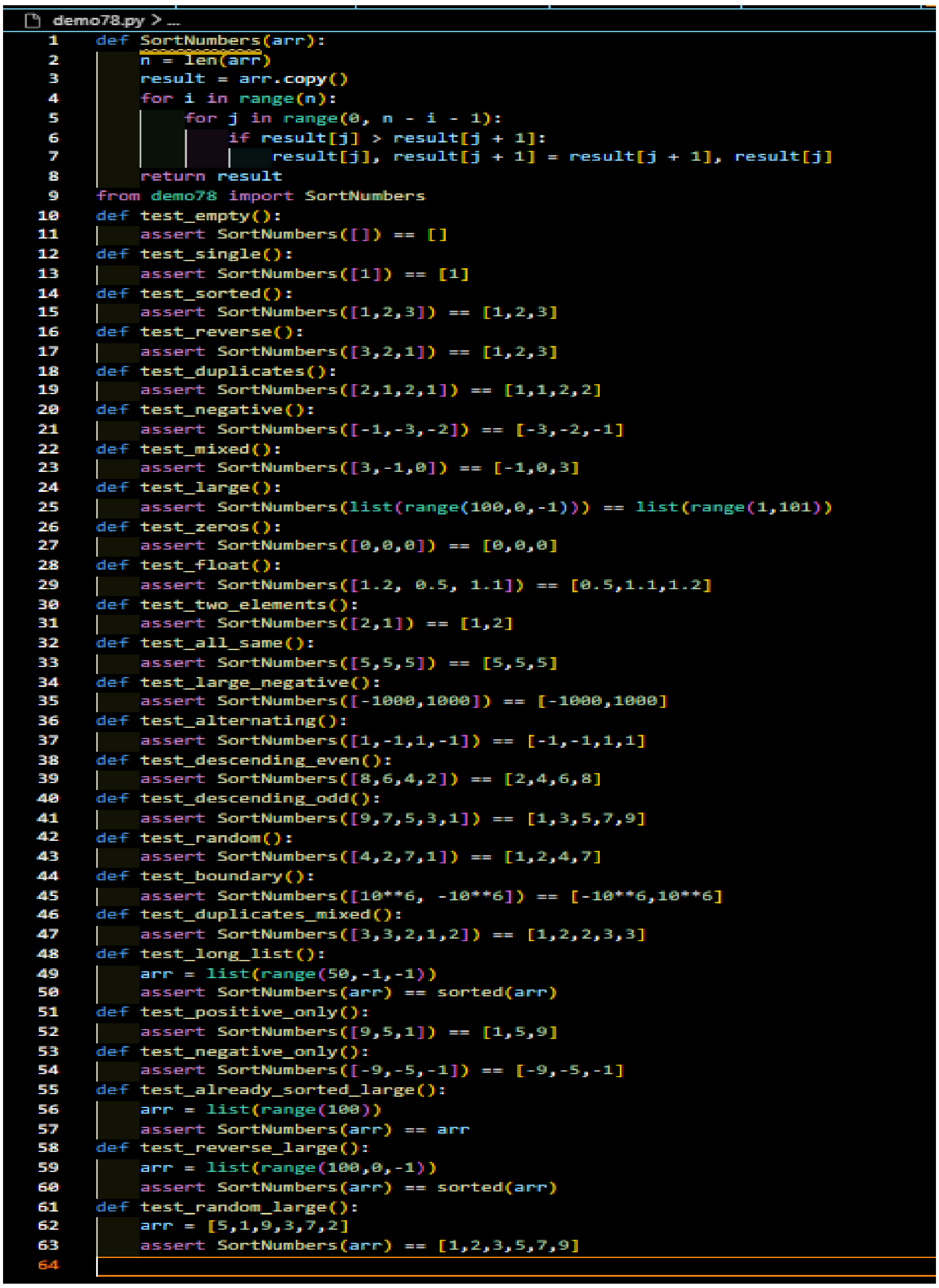


```
rootdir: C:\Users\ankam\OneDrive\Desktop\AIAC
plugins: anyio-4.10.0
collected 7 items


demo76.py::test_positive_number PASSED
              [ 14%]
              [ 14%]
demo76.py::test_single_digit PASSED
              [ 28%]
demo76.py::test_zero PASSED
              [ 42%]
demo76.py::test_negative_number PASSED
              [ 57%]
demo76.py::test_large_number PASSED
              [ 71%]
demo76.py::test_number_with_zeros PASSED
              [ 85%]
demo76.py::test_negative_single_digit PASSED
              [100%]

============================================================================================ 7 passed in 0.08s ======
====================

C:\Users\ankam\OneDrive\Desktop\AIAC>
```

Task 10 :

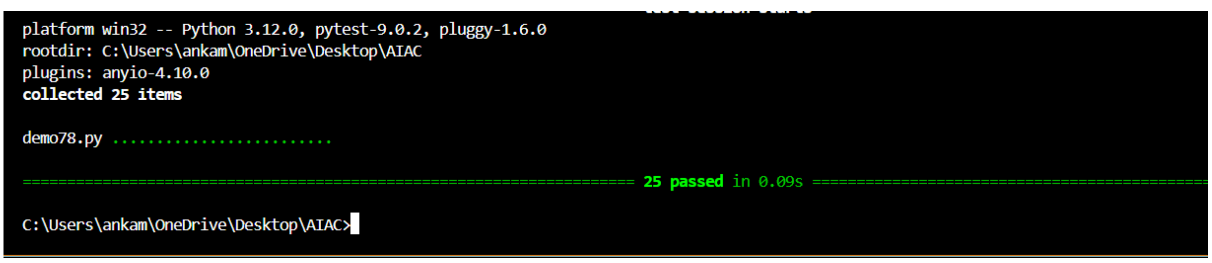
Write a function SortNumbers (implement bubble sort) and validate its implementation using 25 pytest test cases.

Write a Python program for a function SortNumbers(arr) that implements the Bubble Sort algorithm using the Test-Driven Development (TDD) approach. The function should sort a list of numbers in ascending order. Use the pytest framework to create 25 test cases covering different scenarios such as empty lists, single-element lists, already sorted lists, reverse sorted lists, duplicate values, negative numbers, and large inputs. Ensure the function passes all pytest test cases correctly.


demo78.py > ...

```python
def SortNumbers(arr):
    n = len(arr)
    result = arr.copy()
    for i in range(n):
        for j in range(0, n - i - 1):
            if result[j] > result[j + 1]:
                result[j], result[j + 1] = result[j + 1], result[j]
    return result
from demo78 import SortNumbers
def test_empty():
    assert SortNumbers([]) == []
def test_single():
    assert SortNumbers([1]) == [1]
def test_sorted():
    assert SortNumbers([1,2,3]) == [1,2,3]
def test_reverse():
    assert SortNumbers([3,2,1]) == [1,2,3]
def test_duplicates():
    assert SortNumbers([2,1,2,1]) == [1,1,2,2]
def test_negative():
    assert SortNumbers([-1,-3,-2]) == [-3,-2,-1]
def test_mixed():
    assert SortNumbers([3,-1,0]) == [-1,0,3]
def test_large():
    assert SortNumbers(list(range(100,0,-1))) == list(range(1,101))
def test_zeros():
    assert SortNumbers([0,0,0]) == [0,0,0]
def test_float():
    assert SortNumbers([1.2, 0.5, 1.1]) == [0.5,1.1,1.2]
def test_two_elements():
    assert SortNumbers([2,1]) == [1,2]
def test_all_same():
    assert SortNumbers([5,5,5]) == [5,5,5]
def test_large_negative():
    assert SortNumbers([-1000,1000]) == [-1000,1000]
def test_alternating():
    assert SortNumbers([1,-1,1,-1]) == [-1,-1,1,1]
def test_descending_even():
    assert SortNumbers([8,6,4,2]) == [2,4,6,8]
def test_descending_odd():
    assert SortNumbers([9,7,5,3,1]) == [1,3,5,7,9]
def test_random():
    assert SortNumbers([4,2,7,1]) == [1,2,4,7]
def test_boundary():
    assert SortNumbers([10**6, -10**6]) == [-10**6,10**6]
def test_duplicates_mixed():
    assert SortNumbers([3,3,2,1,2]) == [1,2,2,3,3]
def test_long_list():
    arr = list(range(50,-1,-1))
    assert SortNumbers(arr) == sorted(arr)
def test_positive_only():
    assert SortNumbers([9,5,1]) == [1,5,9]
def test_negative_only():
    assert SortNumbers([-9,-5,-1]) == [-9,-5,-1]
def test_already_sorted_large():
    arr = list(range(100))
    assert SortNumbers(arr) == arr
def test_reverse_large():
    arr = list(range(100,0,-1))
    assert SortNumbers(arr) == sorted(arr)
def test_random_large():
    arr = [5,1,9,3,7,2]
    assert SortNumbers(arr) == [1,2,3,5,7,9]

```

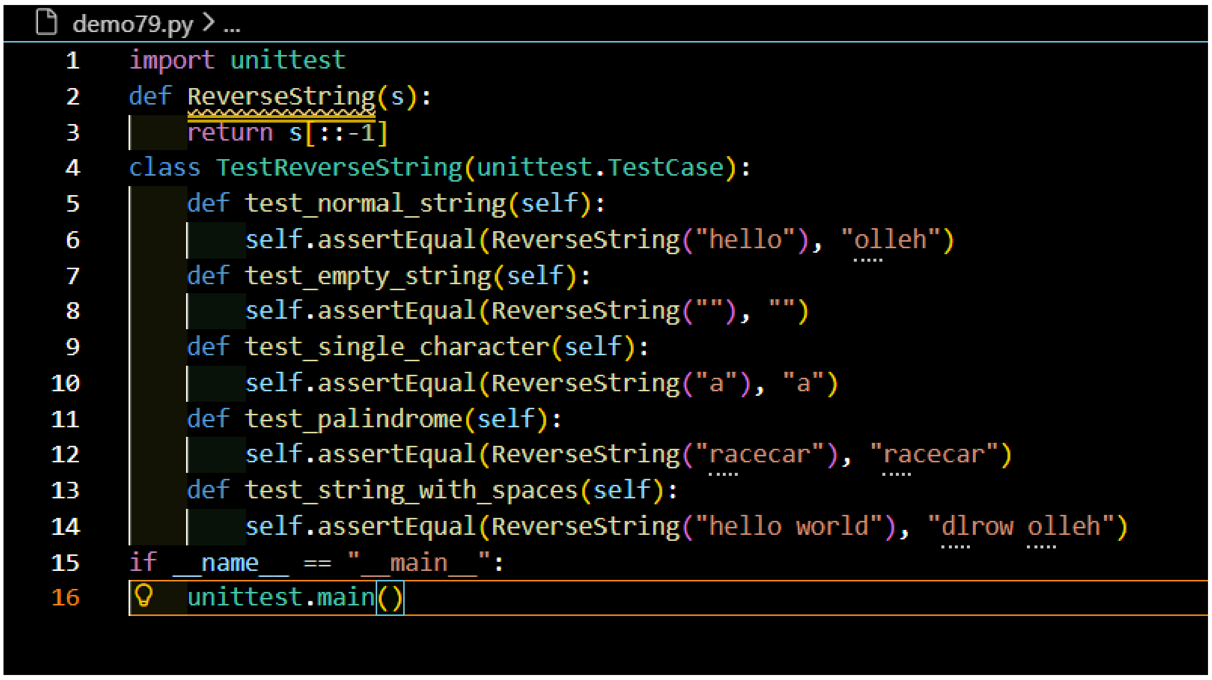
Output:



```
platform win32 -- Python 3.12.0, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\ankam\OneDrive\Desktop\AIAC
plugins: anyio-4.10.0
collected 25 items

demo78.py .........................

================================================================ 25 passed in 0.09s ================================================================

C:\Users\ankam\OneDrive\Desktop\AIAC>
```

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

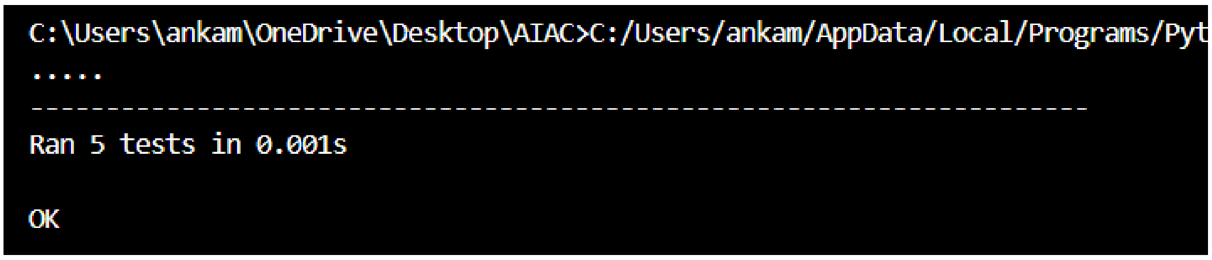
Write a Python program for a function ReverseString(s) using the Test-Driven Development (TDD) approach. The function should return the reverse of a given string. Use the unittest framework to create 5 test cases covering normal strings, empty strings, single characters, palindromes, and strings with spaces. Implement the function so that it passes all unittest test cases successfully.



demo79.py > ...

```python
import unittest
def ReverseString(s):
    return s[::-1]
class TestReverseString(unittest.TestCase):
    def test_normal_string(self):
        self.assertEqual(ReverseString("hello"), "olleh")
    def test_empty_string(self):
        self.assertEqual(ReverseString(""), "")
    def test_single_character(self):
        self.assertEqual(ReverseString("a"), "a")
    def test_palindrome(self):
        self.assertEqual(ReverseString("racecar"), "racecar")
    def test_string_with_spaces(self):
        self.assertEqual(ReverseString("hello world"), "dlrow olleh")
if __name__ == "__main__":
    unittest.main()
```

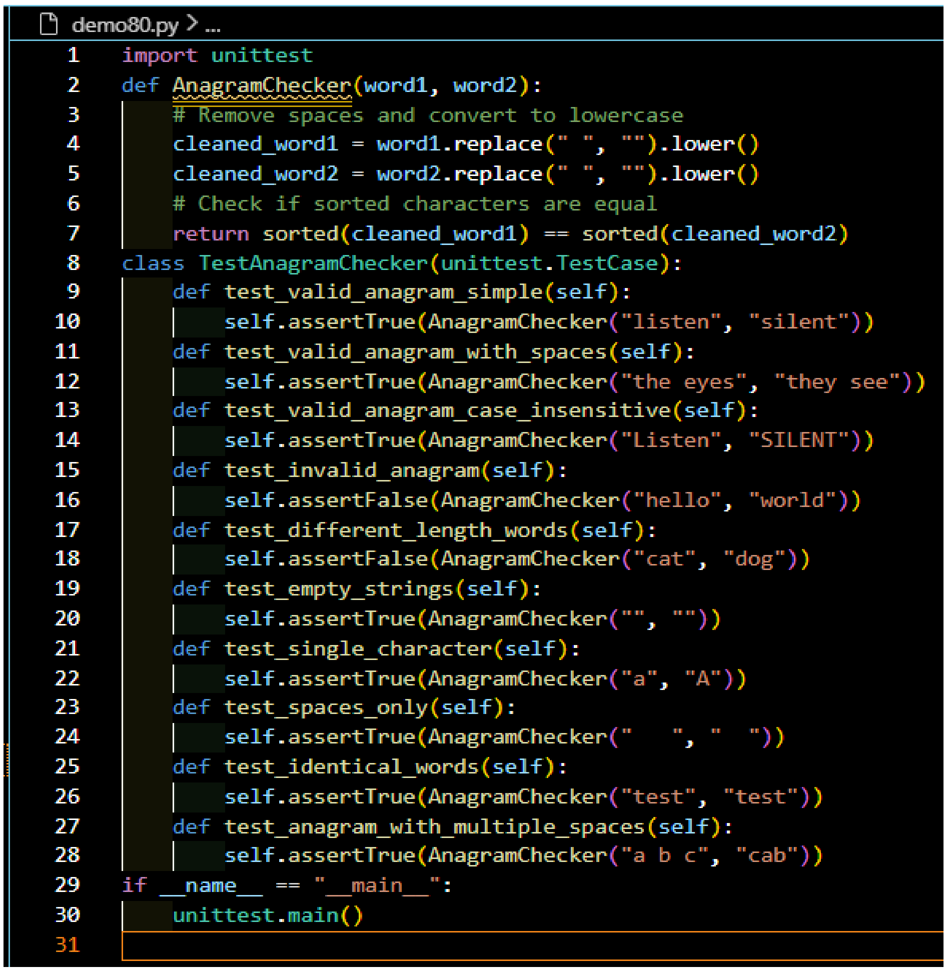
Output:



```
C:\Users\ankam\OneDrive\Desktop\AIAC>C:/Users/ankam/AppData/Local/Programs/Pyt
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
```

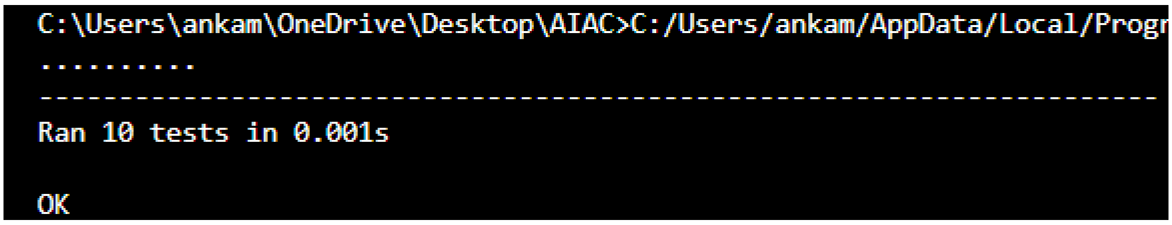
Task 12 :
Write a function AnagramChecker and validate its implementation
using 10 unittest test cases.

Write a Python program for a function AnagramChecker(word1, word2) using the Test-Driven Development (TDD) approach. The function should determine whether two given words are anagrams of each other. It should ignore case and spaces while comparing the words. Use the unittest framework to create 10 test cases covering valid anagrams, invalid cases, case sensitivity checks, and edge cases. Ensure the function passes all unittest test cases correctly.



```
demo80.py > ...
import unittest
def AnagramChecker(word1, word2):
    # Remove spaces and convert to lowercase
    cleaned_word1 = word1.replace(" ", "").lower()
    cleaned_word2 = word2.replace(" ", "").lower()
    # Check if sorted characters are equal
    return sorted(cleaned_word1) == sorted(cleaned_word2)
class TestAnagramChecker(unittest.TestCase):
    def test_valid_anagram_simple(self):
        self.assertTrue(AnagramChecker("listen", "silent"))
    def test_valid_anagram_with_spaces(self):
        self.assertTrue(AnagramChecker("the eyes", "they see"))
    def test_valid_anagram_case_insensitive(self):
        self.assertTrue(AnagramChecker("Listen", "SILENT"))
    def test_invalid_anagram(self):
        self.assertFalse(AnagramChecker("hello", "world"))
    def test_different_length_words(self):
        self.assertFalse(AnagramChecker("cat", "dog"))
    def test_empty_strings(self):
        self.assertTrue(AnagramChecker("", ""))
    def test_single_character(self):
        self.assertTrue(AnagramChecker("a", "A"))
    def test_spaces_only(self):
        self.assertTrue(AnagramChecker("   ", "  "))
    def test_identical_words(self):
        self.assertTrue(AnagramChecker("test", "test"))
    def test_anagram_with_multiple_spaces(self):
        self.assertTrue(AnagramChecker("a b c", "cab"))
if __name__ == "__main__":
    unittest.main()
```

Output:

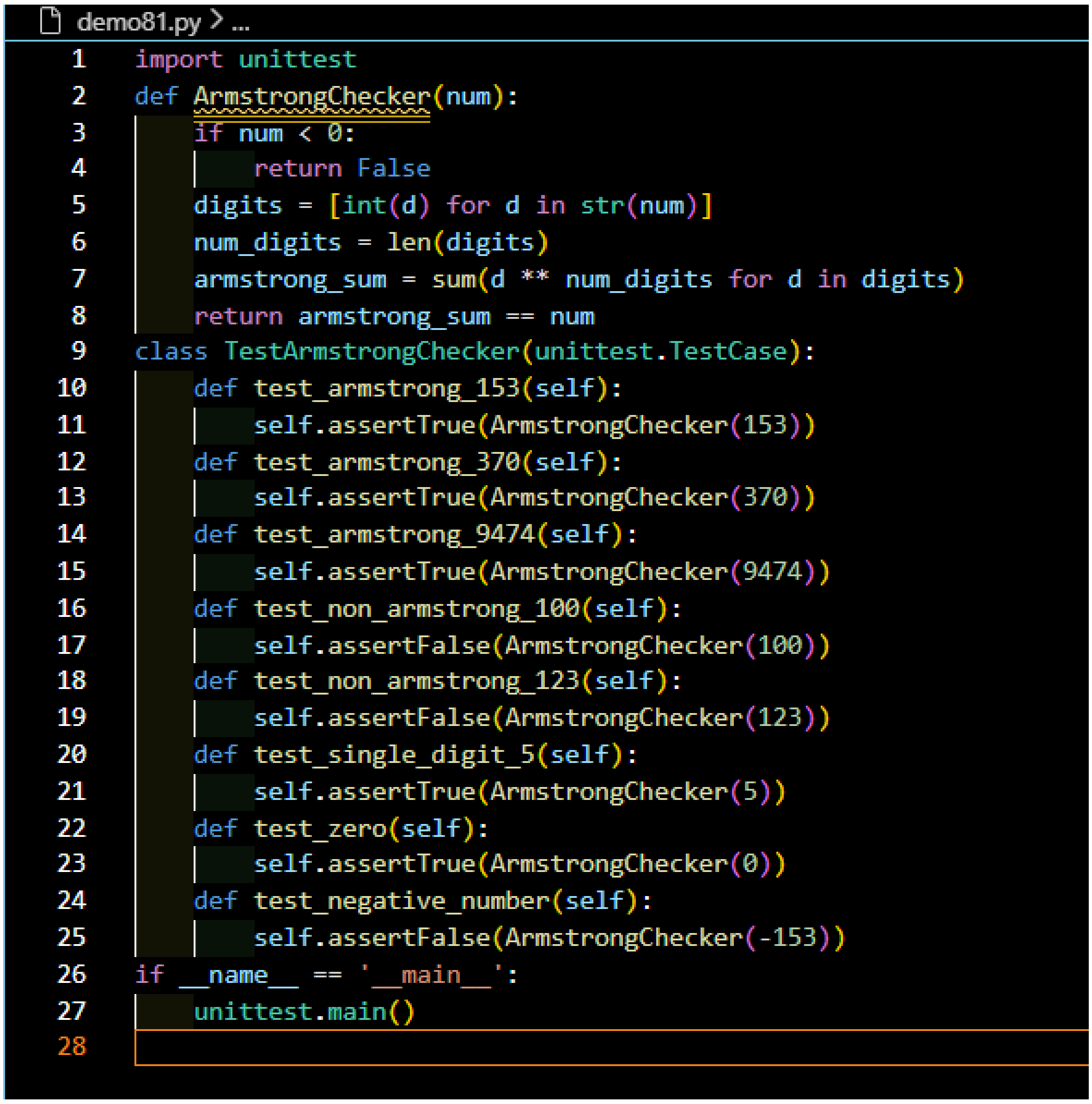


```
C:\Users\ankam\OneDrive\Desktop\AIAC>C:/Users/ankam/AppData/Local/Progr
..........
----------------------------------------------------------------------
Ran 10 tests in 0.001s

OK
```

Task 13 :

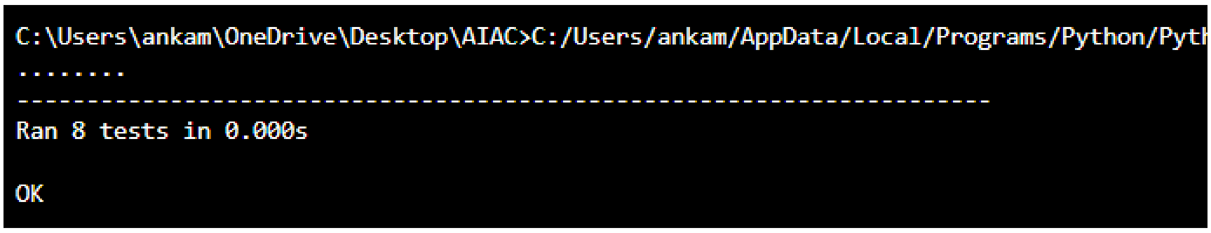
Write a function ArmstrongChecker and validate its implementation using 8 unittest test cases.

Write a Python program for a function ArmstrongChecker(num) using the Test-Driven Development (TDD) approach. The function should determine whether a given number is an Armstrong number, where the sum of each digit raised to the power of the total number of digits equals the number itself. Use the unittest framework to create 8 test cases covering valid Armstrong numbers, non-Armstrong numbers, single-digit numbers, zero, and negative values. Implement the function so that it passes all unittest test cases successfully.



```
demo81.py > ...
import unittest
def ArmstrongChecker(num):
    if num < 0:
        return False
    digits = [int(d) for d in str(num)]
    num_digits = len(digits)
    armstrong_sum = sum(d ** num_digits for d in digits)
    return armstrong_sum == num
class TestArmstrongChecker(unittest.TestCase):
    def test_armstrong_153(self):
        self.assertTrue(ArmstrongChecker(153))
    def test_armstrong_370(self):
        self.assertTrue(ArmstrongChecker(370))
    def test_armstrong_9474(self):
        self.assertTrue(ArmstrongChecker(9474))
    def test_non_armstrong_100(self):
        self.assertFalse(ArmstrongChecker(100))
    def test_non_armstrong_123(self):
        self.assertFalse(ArmstrongChecker(123))
    def test_single_digit_5(self):
        self.assertTrue(ArmstrongChecker(5))
    def test_zero(self):
        self.assertTrue(ArmstrongChecker(0))
    def test_negative_number(self):
        self.assertFalse(ArmstrongChecker(-153))
if __name__ == '__main__':
    unittest.main()
```

Output:



```
C:\Users\ankam\OneDrive\Desktop\AIAC>C:/Users/ankam/AppData/Local/Programs/Python/Pyt
........
----------------------------------------------------------------------
Ran 8 tests in 0.000s

OK
```