AI Assisted Coding

Assignment 8

Name: M.Goutham

HT No: 2303A52010

Batch: 31

Question 1: Username Validator

The function is_valid_username() checks whether a given username meets a set of rules. It returns True only when all conditions are satisfied, and False the moment any rule is broken. The function first checks the length if the username is shorter than 5 or longer than 15 characters, it immediately returns False. Next, it verifies that the first character is a letter (not a digit), because usernames must not start with a number. It then checks for spaces, since usernames should be continuous without gaps. Finally, it loops through every character and confirms each one is either a letter or a digit any symbol like underscore or hyphen would be rejected.

Code:

```
def is_valid_username(username):
    if len(username) < 5 or len(username) > 15:
        return False
    if not username[0].isalpha():
        return False
    if ' ' in username:
        return False
    for char in username:
        if not char.isalnum():
            return False
    return True

assert is_valid_username('user1') == True
assert is_valid_username('1user') == False
assert is_valid_username('user_name') == False
print('All test cases passed!')
```

Output:

```
All test cases passed!
```

Question 2: Even–Odd & Type Classification

The function classify_value() determines what category a given input belongs to. It starts by filtering out any non-numeric input if the value is not an integer or float (or if it's a boolean, which Python treats as a subtype of int), it returns 'Invalid Input'. After that, it checks if the value is exactly zero and returns 'Zero'. For all other numeric values, it uses the modulo operator to check divisibility by 2 if the remainder is 0, the number is 'Even'; otherwise it is 'Odd'. This order of checks matters because zero would also pass the even check if we didn't catch it first.

Code:

```
def classify_value(x):
    if not isinstance(x, (int, float)) or isinstance(x, bool):
        return 'Invalid Input'
    if x == 0:
        return 'Zero'
    elif int(x) % 2 == 0:
        return 'Even'
    else:
        return 'Odd'

assert classify_value(0) == 'Zero'
assert classify_value(2) == 'Even'
assert classify_value(3) == 'Odd'
assert classify_value('i') == 'Invalid Input'
print('All test cases passed!')
```

Output:

```
All test cases passed!
```

Question 3: Palindrome Checker

The function is_palindrome() checks whether a piece of text reads the same forwards and backwards, ignoring case, spaces, and punctuation. It first handles the empty string case by returning True, since an empty input is trivially a palindrome. It then builds a cleaned version of the input by keeping only alphanumeric characters and converting them all to lowercase this removes punctuation and treats upper and lower case as the same. If the cleaned string has one character or none, it's automatically a palindrome. Finally, it compares the cleaned string to its reverse using Python's slice notation; if they match, the word or phrase is a palindrome.

Code:

```
import string

def is_palindrome(text):
    if not text:
        return True
    cleaned = ''.join(char.lower() for char in text if char.isalnum())
    if len(cleaned) <= 1:
        return True
    return cleaned == cleaned[::-1]

assert is_palindrome('') == True
assert is_palindrome('A man a plan a canal Panama') == True
assert is_palindrome("No 'x' in Nixon") == True
print('All test cases passed!')
```

Output:

```
All test cases passed!
```

Question 4: Email ID Validation

The function validate_email() checks if an email address follows the correct format. It first confirms the input is a non-empty string, then rejects any address that contains a space. It enforces exactly one '@' symbol too few or too many would indicate a broken format. The address is then split at '@' into a local part and a domain. Both must be non-empty. The function also prevents the local part and domain from starting or ending with a dot. The domain must contain at least one dot, and the part after the last dot (the top-level domain) must have at least two characters this rules out things like '.c' which are not valid domain endings.

Code:

```
def validate_email(email):
    if not isinstance(email, str) or not email:
        return False
    if ' ' in email:
        return False
    if email.count('@') != 1:
        return False
    local, domain = email.split('@')
    if not local or not domain:
        return False
    if local.startswith('.') or local.endswith('.'):
        return False
    if domain.startswith('.') or domain.endswith('.'):
        return False
    if '.' not in domain:
        return False
    if len(domain.split('.')[-1]) < 2:
        return False
    return True

assert validate_email('user@example.com') == True
assert validate_email('user@.example.com') == False
assert validate_email('user@example.') == False
assert validate_email('user@exam ple.com') == False
assert validate_email('user@example.c') == False
print('All test cases passed!')
```

Output:

```
All test cases passed!
```

Question 5: Perfect Number Checker

A perfect number is one whose proper divisors (all divisors excluding the number itself) add up to exactly the number. For example, 6 has divisors 1, 2, and 3 and 1+2+3 = 6. The function is_perfectnumber() starts with 1 as the base divisor, since 1 divides every number. It then iterates only up to the square root of the number for efficiency, adding both i and n//i when i is a divisor but only if they are different (to avoid double-counting perfect squares). At the end, it checks whether the total matches the original number. Numbers less than or equal to 1 are excluded since they cannot be perfect by definition.

Code:

```
def is_perfectnumber(n):
    if n <= 1:
        return False
    sum_of_divisors = 1
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            sum_of_divisors += i
            if i != n // i:
                sum_of_divisors += n // i
    return sum_of_divisors == n

assert is_perfectnumber(6) == True
assert is_perfectnumber(28) == True
assert is_perfectnumber(12) == False
assert is_perfectnumber(-1) == False
print('All test cases passed!')
```

Output:

```
All test cases passed!
```

Question 6: Abundant Number Checker

An abundant number is one where the sum of its proper divisors is greater than the number itself. For example, 12 has divisors 1, 2, 3, 4, and 6, which sum to 16 more than 12 itself. The function abundant() adds up all divisors from 1 up to n-1, then compares the total to n. Numbers less than or equal to 1 are immediately returned as False since they cannot be abundant. The test cases use Python's unittest framework, which organizes tests into a class and runs them with assertEqual, assertTrue, and assertFalse methods for structured and readable validation.

Code:

```
def abundant(n):
    if n <= 1:
        return False
    sum_of_divisors = 0
    for i in range(1, n):
        if n % i == 0:
            sum_of_divisors += i
    return sum_of_divisors > n

import unittest

class TestAbundant(unittest.TestCase):
    def test_abundant_normal_true(self):
        self.assertTrue(abundant(12))
    def test_abundant_normal_false(self):
        self.assertFalse(abundant(15))
    def test_abundant_edge_one(self):
        self.assertFalse(abundant(1))
    def test_abundant_negative(self):
        self.assertFalse(abundant(-12))
    def test_abundant_large(self):
        self.assertTrue(abundant(945))

if __name__ == '__main__':
    unittest.main()
```

Output:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s
```

```
OK
```

Question 7: Deficient Number Checker

A deficient number is one where the sum of its proper divisors is less than the number. The function Deficient() uses an optimized approach instead of iterating all the way to n, it only goes up to the square root of n and adds both paired divisors at once. This makes it significantly faster for large numbers. The number 1 is treated as a special case and is considered deficient by definition (its only divisor sum is 0, which is less than 1). Negative numbers return False since divisibility is only meaningful for positive integers. The tests use pytest, which automatically discovers functions starting with 'test_' and runs them.

Code:

```
def Deficient(n):
    if n <= 1:
        return n == 1
    sum_of_divisors = 1
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            sum_of_divisors += i
            if i != n // i:
                sum_of_divisors += n // i
    return sum_of_divisors < n

def test_deficient():
    assert Deficient(8) == True
    assert Deficient(15) == True

def test_edge_cases():
    assert Deficient(1) == True

def test_negative():
    assert Deficient(-5) == False

def test_large():
    assert Deficient(28) == False
```

Output:

```
python -m pytest 7.py
collected 4 items
7.py ....                [100%]
4 passed in 0.02s
```

Question 8: Leap Year Checker

A leap year follows two rules: it must be divisible by 4 but not by 100, OR it must be divisible by 400. The second rule is an exception that overrides the first so while 1900 is divisible by 4, it fails because it's also divisible by 100 without being divisible by 400. The year 2000 passes because it is divisible by 400. The function LeapYearChecker() directly evaluates this compound condition. The 10 test cases cover regular leap years, century years that are not leap years, century years that are leap years, edge cases like year 0 and year 1, and negative years all of which follow the same mathematical rules.

Code:

```
def LeapYearChecker(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    return False

def test_leap_year_checker():
    assert LeapYearChecker(2020) == True
    assert LeapYearChecker(1900) == False
    assert LeapYearChecker(2000) == True
    assert LeapYearChecker(0) == True
    assert LeapYearChecker(1) == False
    assert LeapYearChecker(4) == True
    assert LeapYearChecker(100) == False
    assert LeapYearChecker(-4) == True
    assert LeapYearChecker(-100) == False
    assert LeapYearChecker(-400) == True
```

Output:

```
python -m pytest 8.py

collected 1 item

8.py .                   [100%]

1 passed in 0.02s
```

Question 9: Sum of Digits

The function SumOfDigits() extracts and adds each individual digit of a number. It begins by converting any negative number to its absolute value, so the sign is ignored during computation. It then repeatedly extracts the last digit using the modulo operator (n % 10), adds it to a running total, and removes it from the number by integer division (n //= 10). This loop continues until the number is fully consumed. For the input 0, the while loop never executes, so the function correctly returns 0.

Code:

```
def SumOfDigits(n):
    n = abs(n)
    total = 0
    while n > 0:
        digit = n % 10
        total += digit
        n //= 10
    return total

def test_sum_of_digits():
    assert SumOfDigits(123) == 6
    assert SumOfDigits(456) == 15
    assert SumOfDigits(0) == 0
    assert SumOfDigits(9) == 9
    assert SumOfDigits(123456789) == 45
    assert SumOfDigits(-123) == 6
    assert SumOfDigits(-987) == 24
```

Output:

```
python -m pytest 9.py
collected 1 item
9.py .                [100%]
1 passed in 0.02s
```

Question 10: Sort Numbers (Bubble Sort)

Bubble Sort is a simple comparison-based sorting algorithm. In each pass through the array, adjacent elements are compared and swapped if they are in the wrong order. After the first full pass, the largest element 'bubbles up' to the last position. After the second pass, the second largest settles into place, and so on. The outer loop controls how many passes are made, while the inner loop shrinks on each pass because the already-sorted end of the array doesn't need to be visited again. The 25 pytest test cases cover a wide range of scenarios including empty lists, single elements, duplicates, negative numbers, already-sorted arrays, reverse-sorted arrays, and very large inputs.

Code:

```
def SortNumbers_bubble(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_normal_unsorted_array():
    assert SortNumbers_bubble([5, 2, 9, 1, 5, 6]) == [1, 2, 5, 5, 6, 9]
# ... (25 total test cases)
```

Output:

```
python -m pytest 10.py
collected 25 items
10.py .........................     [100%]
25 passed in 0.10s
```

Question 11: Reverse String

The function ReverseString() uses Python's slice notation with a step of -1 to reverse any string in a single line. Slicing with [::-1] starts from the end of the string and works backwards to the beginning, effectively producing the mirror image of the input. This approach handles all edge cases naturally an empty string reversed is still empty, and a single character reversed is itself. For palindromes like 'madam', the reversed version is identical to the original. The 5 unittest test cases cover these scenarios systematically.

Code:

```
def ReverseString(s):
    return s[::-1]

import unittest

class TestReverseString(unittest.TestCase):
    def test_simple_cases(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('madam'), 'madam')
    def test_with_spaces(self):
        self.assertEqual(ReverseString('hello world'), 'dlrow olleh')

if __name__ == '__main__':
    unittest.main()
```

Output:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s


OK
```

Question 12: Anagram Checker

Two words are anagrams if they contain exactly the same letters, just arranged differently. The function AnagramChecker() handles this by first converting both inputs to lowercase and removing spaces, then sorting all characters alphabetically. If the sorted versions are equal, the two words are anagrams. Sorting works here because anagrams have the same characters in any order sorting gives them a common canonical form. The function naturally handles case differences, spaces between words, and even special characters or digits, since it just sorts whatever characters remain after removing spaces.

Code:

```
import unittest

def AnagramChecker(word1, word2):
    return sorted(word1.lower().replace(' ', '')) == sorted(word2.lower().replace(' ',
''))

class TestAnagramChecker(unittest.TestCase):
    def test_simple_anagrams(self):
        self.assertTrue(AnagramChecker('listen', 'silent'))
    def test_non_anagrams(self):
        self.assertFalse(AnagramChecker('hello', 'world'))
    # ... (10 total test cases)

if __name__ == '__main__':
    unittest.main()
```

Output:

```
..........
----------------------------------------------------------------------
Ran 10 tests in 0.000s


OK
```

Question 13: Armstrong Number Checker

An Armstrong number (also called a narcissistic number) is one where the sum of each digit raised to the power of the total number of digits equals the number itself. For example, 153 is Armstrong because it has 3 digits and $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$. The function ArmstrongChecker() converts the number to a string to count digits and iterate over each one, then raises each digit to that power and sums them up. Negative numbers are excluded since the concept applies only to positive integers. Zero is also excluded since it doesn't satisfy the definition in conventional usage. Single-digit numbers like 1 through 9 are always Armstrong since $n^1 = n$.

Code:

```
def ArmstrongChecker(num):
    if num < 0:
        return False
    num_str = str(num)
    num_digits = len(num_str)
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
    return armstrong_sum == num and num != 0

import unittest

class TestArmstrongChecker(unittest.TestCase):
    def test_armstrong_numbers(self):
        self.assertTrue(ArmstrongChecker(153))
        self.assertTrue(ArmstrongChecker(370))
        self.assertTrue(ArmstrongChecker(371))
        self.assertTrue(ArmstrongChecker(407))
    def test_non_armstrong_numbers(self):
        self.assertFalse(ArmstrongChecker(123))
        self.assertFalse(ArmstrongChecker(0))
        self.assertFalse(ArmstrongChecker(-153))
    def test_single_digit_armstrong_numbers(self):
        self.assertTrue(ArmstrongChecker(1))

if __name__ == '__main__':
    unittest.main()
```

Output:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s


OK
```