AI Assisted Coding

Assignment – 13

Name : G.Samshray

Ht.no : 2303A51670

Batch : 23

Task Description #1 (Refactoring – Removing Code Duplication)

• **Task: Use AI to refactor a given Python script that contains multiple repeated code blocks.**

• Instructions:

o Prompt AI to identify duplicate logic and replace it with functions or classes.

o Ensure the refactored code maintains the same output.

o Add docstrings to all functions.

• Sample Legacy Code:

```
# Legacy script with repeated logic
print("Area of Rectangle:", 5 * 10)
print("Perimeter of Rectangle:", 2 * (5 + 10))
print("Area of Rectangle:", 7 * 12)
print("Perimeter of Rectangle:", 2 * (7 + 12))
print("Area of Rectangle:", 10 * 15)
print("Perimeter of Rectangle:", 2 * (10 + 15))
```

• **Expected Output:**

o Refactored code with a reusable function and no duplication.

o Well documented code.

Code :

```
# Refactored script with functions to eliminate code duplication
def calculate_rectangle_properties(length, width):
    """
    Calculate and print the area and perimeter of a rectangle.

    Args:
        length (float): The length of the rectangle.
        width (float): The width of the rectangle.
    """
    area = length * width
    perimeter = 2 * (length + width)
    print(f"Area of Rectangle: {area}")
    print(f"Perimeter of Rectangle: {perimeter}")

# Calling the function with different dimensions
calculate_rectangle_properties(5, 10)
calculate_rectangle_properties(7, 12)
calculate_rectangle_properties(10, 15)
```

Output :

```
Area of Rectangle: 50
Perimeter of Rectangle: 30
Area of Rectangle: 84
Perimeter of Rectangle: 38
Area of Rectangle: 150
Perimeter of Rectangle: 50
```

Task Description #2 (Refactoring – Extracting Reusable Functions)

• Task: Use AI to refactor a legacy script where multiple calculations are embedded directly inside the main code block.

• Instructions:

o Identify repeated or related logic and extract it into reusable functions.

o Ensure the refactored code is modular, easy to read, and documented with docstrings.

• Sample Legacy Code:

Week7

-

Monda

y

```
# Legacy script with inline repeated logic
price = 250
tax = price * 0.18
total = price + tax
print("Total Price:", total)
price = 500
tax = price * 0.18
total = price + tax
print("Total Price:", total)
```

• Expected Output:

o Code with a function calculate_total(price) that can be reused for multiple price inputs.

o Well documented code

Code :

```
# Refactored script with a reusable function to calculate total price
def calculate_total(price):
    """
    Calculate the total price including tax.


    Args:
        price (float): The original price of the item.


    Returns:
        float: The total price after adding tax.
    """
    tax = price * 0.18
    total = price + tax
    return total
# Calling the function with different price inputs
price1 = 250
total1 = calculate_total(price1)
print(f"Total Price for {price1}: {total1}")
price2 = 500
total2 = calculate_total(price2)
print(f"Total Price for {price2}: {total2}")
price3 = 750
total3 = calculate_total(price3)
print(f"Total Price for {price3}: {total3}")
```

Output :

```
Total Price for 250: 295.0
Total Price for 500: 590.0
Total Price for 750: 885.0
```

Task Description #3: Refactoring Using Classes and Methods (Eliminating Redundant Conditional Logic)
Refactor a Python script that contains repeated if–elif–else grading logic by implementing a structured, object-oriented solution using a class and a method.

Problem Statement

The given script contains duplicated conditional statements used to assign grades based on student marks. This redundancy violates clean code principles and reduces maintainability.

You are required to refactor the script using a class-based design to improve modularity, reusability, and readability while preserving the original grading logic.

Mandatory Implementation Requirements

1. Class Name: GradeCalculator
2. Method Name: calculate_grade(self, marks)
3. The method must:

o Accept marks as a parameter.

o Return the corresponding grade as a string.

o The grading logic must strictly follow the conditions below:

▪ Marks ≥ 90 and ≤ 100 → "Grade A"

▪ Marks ≥ 80 → "Grade B"

▪ Marks ≥ 70 → "Grade C"

▪ Marks ≥ 40 → "Grade D"

▪ Marks ≥ 0 → "Fail"

Note: Assume marks are within the valid range of 0 to 100.

4. Include proper docstrings for:

o The class

o The method (with parameter and return descriptions)

5. The method must be reusable and called multiple times without rewriting conditional logic.

• Given code:

```
marks = 85
if marks >= 90:
print("Grade A")
```

```
elif marks >= 75:
print("Grade B")
else:
print("Grade C")
marks = 72
if marks >= 90:
print("Grade A")
elif marks >= 75:
print("Grade B")
else:
print("Grade C")
```

Expected Output:

• Define a class named GradeCalculator.

• Implement a method calculate_grade(self, marks) inside the class.

• Create an object of the class.

• Call the method for different student marks.

• Print the returned grade values.

Code :

```
class GradeCalculator:
    """
    A class to calculate grades based on student marks.
    """


    def calculate_grade(self, marks):
        """
        Calculate the grade based on the provided marks.


        Args:
            marks (float): The marks obtained by the student (0-100).

        Returns:
            str: The corresponding grade as a string.
        """
        if 90 <= marks <= 100:
            return "Grade A"
        elif marks >= 80:
            return "Grade B"
        elif marks >= 70:
            return "Grade C"
        elif marks >= 40:
            return "Grade D"
        elif marks >= 0:
            return "Fail"
        else:
            return "Invalid marks"
# Create an object of the GradeCalculator class
grade_calculator = GradeCalculator()
# Call the method for different student marks and print the results
marks_list = [85, 72, 95, 60, 30, -5]  # Example marks for testing
for marks in marks_list:
    grade = grade_calculator.calculate_grade(marks)
    print(f"Marks: {marks} - {grade}")
```

Output :

```
Marks: 85 - Grade B
Marks: 72 - Grade C
Marks: 95 - Grade A
```

```
Marks: 60 - Grade D
Marks: 30 - Fail
Marks: -5 - Invalid marks
```

Task Description #4 (Refactoring – Converting Procedural Code to Functions)

• Task: Use AI to refactor procedural input–processing logic into functions.

Instructions:

o Identify input, processing, and output sections.

o Convert each into a separate function.

o Improve code readability without changing behavior.

• Sample Legacy Code:

```
num = int(input("Enter number: "))
square = num * num
print("Square:", square)
```

• Expected Output:

o Modular code using functions like get_input(), calculate_square(), and display_result().

Code :

```
def get_input():
    """
    Get a number input from the user.

    Returns:
        int: The number entered by the user.
    """
    num = int(input("Enter number: "))
    return num

def calculate_square(num):
    """Calculate the square of a number.

    Args:
        num (int): The number to be squared.

    Returns:
        int: The square of the input number.
    """
    return num * num

def display_result(square):
    """Display the result of the square calculation.

    Args:
        square (int): The square of the input number.
    """
    print("Square:", square)

# Main function to orchestrate the flow of the program
def main():
    num = get_input()
    square = calculate_square(num)
    display_result(square)

# Run the main function
if __name__ == "__main__":
    main()
```

Output :

```
Enter number: 50

Square: 2500


Enter number: 40

Square: 1600
```

Task 5 (Refactoring Procedural Code into OOP Design)

• Task: Use AI to refactor procedural code into a class-based design.

Focus Areas:

o Object-Oriented principles

o Encapsulation

Legacy Code:

```
salary = 50000

tax = salary * 0.2

net = salary - tax

print(net)
```

Expected Outcome:

o A class like EmployeeSalaryCalculator with methods and attributes.

Code :

```
class EmployeeSalaryCalculator:
    """
    A class to calculate the net salary of an employee after tax.
    """


    def __init__(self, salary):
        """
        Initialize the EmployeeSalaryCalculator with the given salary.


        Args:
            salary (float): The gross salary of the employee.
        """
        self.salary = salary

    def calculate_tax(self):
        """
        Calculate the tax based on the salary.


        Returns:
            float: The calculated tax amount.
        """
        return self.salary * 0.2

    def calculate_net_salary(self):
        """
        Calculate the net salary after deducting tax.


        Returns:
            float: The net salary of the employee.
        """
        tax = self.calculate_tax()
        net_salary = self.salary - tax
        return net_salary
# Example usage
employee_salary_calculator = EmployeeSalaryCalculator(50000)
net_salary = employee_salary_calculator.calculate_net_salary()
```

```
print(f"Net Salary: {net_salary}")
```

Output :

```
Net Salary: 40000.0
```

Task 6 (Optimizing Search Logic)

• Task: Refactor inefficient linear searches using appropriate data structures.

• Focus Areas:

o Time complexity

o Data structure choice

Legacy Code:

```
users = ["admin", "guest", "editor", "viewer"]
name = input("Enter username: ")
found = False
for u in users:
if u == name:
found = True
print("Access Granted" if found else "Access Denied")
```

Expected Outcome:

o Use of sets or dictionaries with complexity justification.

Code:

```
users = {"admin": "Access Granted", "guest": "Access Granted", "editor": "Access Granted",
"viewer": "Access Granted"}
name = input("Enter username: ")
if name in users:
   print(users[name])
else:
   print("Access Denied")
```

Output :

```
Enter username: Sathish kumar
Access Denied
```

Task 7 – Refactoring the Library Management System

Problem Statement

You are provided with a poorly structured Library Management script that:

• Contains repeated conditional logic

• Does not use reusable functions

• Lacks documentation

• Uses print-based procedural execution

• Does not follow modular programming principles

Your task is to refactor the code into a proper format

1. Create a module library.py with functions:

o add_book(title, author, isbn)

o remove_book(isbn)

o search_book(isbn)

2. Insert triple quotes under each function and let Copilot complete the docstrings.

3. Generate documentation in the terminal.

4. Export the documentation in HTML format.

5. Open the file in a browser.

Given Code

```
# Library Management System (Unstructured Version)
# This code needs refactoring into a proper module with documentation.
library_db = {}
# Adding first book
title = "Python Basics"
author = "John Doe"
```

```
isbn = "101"
if isbn not in library_db:
library_db[isbn] = {"title": title, "author": author}
print("Book added successfully.")
else:
print("Book already exists.")
# Adding second book (duplicate logic)
title = "AI Fundamentals"
author = "Jane Smith"
isbn = "102"
if isbn not in library_db:
library_db[isbn] = {"title": title, "author": author}
print("Book added successfully.")
else:
print("Book already exists.")
# Searching book (repeated logic structure)
isbn = "101"
if isbn in library_db:
print("Book Found:", library_db[isbn])
else:
print("Book not found.")
# Removing book (again repeated pattern)
isbn = "101"
if isbn in library_db:
del library_db[isbn]
print("Book removed successfully.")
else:
print("Book not found.")
# Searching again
```

```
isbn = "101"
if isbn in library_db:
print("Book Found:", library_db[isbn])
else:
print("Book not found.")
```

Code :

```
"""Library Management System Module
This module provides functions to manage a library database, including adding,
removing, and searching for books.
"""
library_db = {}
def add_book(title, author, isbn):
    """
    Add a book to the library database.

    Args:
        title (str): The title of the book.
        author (str): The author of the book.
        isbn (str): The ISBN number of the book.

    Returns:
        str: A message indicating whether the book was added successfully or if it already exists.
    """
    if isbn not in library_db:
        library_db[isbn] = {"title": title, "author": author}
        return "Book added successfully."
    else:
        return "Book already exists."
```

```
def remove_book(isbn):
    """
    Remove a book from the library database.


    Args:
        isbn (str): The ISBN number of the book to be removed.
    Returns:


        str: A message indicating whether the book was removed successfully or if it was not
found.
    """
    if isbn in library_db:
        del library_db[isbn]
        return "Book removed successfully."
    else:
        return "Book not found."
def search_book(isbn):
    """
    Search for a book in the library database.


    Args:
        isbn (str): The ISBN number of the book to search for.


    Returns:
        str: A message indicating whether the book was found along with its details, or if it was
not found.
    """
    if isbn in library_db:
        return f"Book Found: {library_db[isbn]}"
    else:
        return "Book not found."
# Example usage
if __name__ == "__main__":
    print(add_book("Python Basics", "John Doe", "101"))
    print(add_book("AI Fundamentals", "Jane Smith", "102"))
    print(search_book("101"))
    print(remove_book("101"))
    print(search_book("101"))
```

Output :

```
Book added successfully.
Book added successfully.
Book Found: {'title': 'Python Basics', 'author': 'John Doe'}
Book removed successfully.
Book not found.
```

Task 8– Fibonacci Generator

Write a program to generate Fibonacci series up to n.

The initial code has:

• Global variables.

• Inefficient loop.

• No functions or modularity.

Task for Students:

• Refactor into a clean reusable function (generate_fibonacci).

• Add docstrings and test cases.

• Compare AI-refactored vs original.

Bad Code Version:

```
# fibonacci bad version
n=int(input("Enter limit: "))
```

```
a=0

b=1

print(a)

print(b)

for i in range(2,n):

c=a+b

print(c)

a=b

b=c
```

Code :

# Refactored the Fibonacci script into a clean, reusable function generate_fibonacci(n) with peoper docString,no global variable,add test cases,and improve modularity while keeping the same output:

```
def generate_fibonacci(n):
    """
    Generate Fibonacci series up to n.


    Args:
        n (int): The limit up to which the Fibonacci series should be generated.


    Returns:
        list: A list containing the Fibonacci series up to n.
    """
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]

    fib_series = [0, 1]
    for i in range(2, n):
        next_fib = fib_series[i-1] + fib_series[i-2]
        fib_series.append(next_fib)

    return fib_series
# Test cases
if __name__ == "__main__":
    print(generate_fibonacci(10))  # Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
    print(generate_fibonacci(5))   # Output: [0, 1, 1, 2, 3]
    print(generate_fibonacci(0))   # Output: []
    print(generate_fibonacci(1))   # Output: [0]
    print(generate_fibonacci(2))   # Output: [0, 1]
```

Output :

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
[0, 1, 1, 2, 3]
[]
[0]
[0, 1]
```

Task 9 – Twin Primes Checker

Twin primes are pairs of primes that differ by 2 (e.g., 11 and 13, 17 and 19).

The initial code has:

• Inefficient prime checking.

• No functions.

• Hardcoded inputs.

Task for Students:

• Refactor into is_prime(n) and is_twin_prime(p1, p2).

• Add docstrings and optimize.

• Generate a list of twin primes in a given range using AI.

Bad Code Version:

```
# twin primes bad version
a=11
b=13
fa=0
for i in range(2,a):
if a%i==0:
fa=1
fb=0
for i in range(2,b):
if b%i==0:
fb=1
if fa==0 and fb==0 and abs(a-b)==2:
print("Twin Primes")
else:
print("Not Twin Primes")
```

Code :

```
# twin prime bad version.
a=11
b=13
fa = 0
for i in range(2,a):
    if a%i==0:
        fa=1
```

```
fb = 0
for i in range(2,b):
    if b%i==0:
        fb=1
if fa==0 and fb==0:
    print("Twin Prime")
else:
    print("Not Twin Prime")
```

# Refactored the twin prime checker by creating a reusable function is_twin_prime(a, b) ad is_twin_prime(p1,p2),add docstrings,optimize the prime checking logic, and remove hardcoded logic and generate twin primes within a given range.

```
import math
def is_prime(num):
    """
    Check if a number is prime.


    Args:
        num (int): The number to check for primality.
    Returns:        bool: True if the number is prime, False otherwise.
    """
    if num < 2 :
        return False
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            return False
    return True
def is_twin_prime(p1, p2):
    """
    Check if two numbers are twin primes.
    Args:
        p1 (int): The first number.
        p2 (int): The second number.

    Returns:
        bool: True if both numbers are prime and their difference is 2, False otherwise.
    """
    if is_prime(p1) and is_prime(p2) and abs(p1 - p2) == 2:
        return True
    return False
def generate_twin_primes(limit):
    """
    Generate twin primes up to a given limit.

    Args:
        limit (int): The upper limit for generating twin primes.

    Returns:
        list: A list of tuples, each containing a pair of twin primes.
    """
    twin_primes = []
    for num in range(2, limit - 1):
        if is_twin_prime(num, num + 2):
            twin_primes.append((num, num + 2))
    return twin_primes
# Test cases
if __name__ == "__main__":
    print(is_twin_prime(11, 13))  # Output: True
    print(is_twin_prime(17, 19))  # Output: True
    print(is_twin_prime(23, 25))  # Output: False
    print(generate_twin_primes(30))  # Output: [(3, 5), (11, 13), (17, 19), (29, 31)]
```

Output :

```
Twin Prime
True
True
False
[(3, 5), (5, 7), (11, 13), (17, 19)]
```

Task 10 – Refactoring the Chinese Zodiac Program

Objective

Refactor the given poorly structured Python script into a clean, modular, and reusable implementation.

The current program reads a year from the user and prints the corresponding Chinese Zodiac sign. However, the implementation contains repetitive conditional logic, lacks modular design, and does not follow clean coding principles.

Your task is to refactor the code to improve readability, maintainability, and structure.

Chinese Zodiac Cycle (Repeats Every 12 Years)

1. Rat
2. Ox
3. Tiger
4. Rabbit
5. Dragon
6. Snake
7. Horse
8. Goat (Sheep)
9. Monkey

10. Rooster

11. Dog

12. Pig

```
# Chinese Zodiac Program (Unstructured Version)
# This code needs refactoring.
year = int(input("Enter a year: "))
if year % 12 == 0:
print("Monkey")
elif year % 12 == 1:
print("Rooster")
elif year % 12 == 2:
print("Dog")
elif year % 12 == 3:
print("Pig")
elif year % 12 == 4:
print("Rat")
elif year % 12 == 5:
print("Ox")
elif year % 12 == 6:
print("Tiger")
elif year % 12 == 7:
print("Rabbit")
elif year % 12 == 8:
print("Dragon")
elif year % 12 == 9:
print("Snake")
elif year % 12 == 10:
print("Horse")
elif year % 12 == 11:
```

```
print("Goat")
```

You must:

1. Create a reusable function: get_zodiac(year)
2. Replace the if-elif chain with a cleaner structure (e.g., list or dictionary).
3. Add proper docstrings.
4. Separate input handling from logic.
5. Improve readability and maintainability.
6. Ensure output remains correct.

Code :

```
# chinese zodiac program(unstructured version) and needs refactoring.

year = int(input("Enter year: "))
if(year % 12 == 0):
    print("Monkey")
elif(year % 12 == 1):
    print("Rooster")
elif(year % 12 == 2):
    print("Dog")
elif(year % 12 == 3):
    print("Pig")
elif(year % 12 == 4):
    print("Rat")
elif(year % 12 == 5):
    print("Ox")
elif(year % 12 == 6):
    print("Tiger")
elif(year % 12 == 7):
    print("Rabbit")
elif(year % 12 == 8):
    print("Dragon")
elif(year % 12 == 9):
    print("Snake")
elif(year % 12 == 10):
    print("Horse")
elif(year % 12 == 11):
    print("Goat")
```

# Refactor the Chinese Zodiac program into a clean, reusable function get_chinese_zodiac(year) with proper docstrings, remove hardcoded logic by using a list to store the zodiac signs, and add test cases.

```
"""
  chinese zodiac module
  This module provides a reusable function to determine
  the chinese zodiac sign for a given year.
  """
def get_zodiac(year):
    """
    Determine the chinese zodiac sign for a given year.
    Parameters:
    year(int):The year for which the zodiac sign is required.
    Returns:
    str: The chinese zodiac sign corresponding to the given year.
    """
    zodiac_signs = ["Monkey", "Rooster", "Dog", "Pig", "Rat", "Ox", "Tiger", "Rabbit",
"Dragon", "Snake", "Horse", "Goat"]
    return zodiac_signs[year % 12]
if __name__ == "__main__":
    # Test cases
```

```
print(get_zodiac(2020))  # Output: Rat
print(get_zodiac(2021))  # Output: Ox
print(get_zodiac(2022))  # Output: Tiger
print(get_zodiac(2023))  # Output: Rabbit
print(get_zodiac(2024))  # Output: Dragon
```

Output :

```
Enter year: 2024
Dragon
Rat
Rat
Ox
Tiger
Rabbit
Dragon
```

Task 11 – Refactoring the Harshad (Niven) Number Checker

Refactor the given poorly structured Python script into a clean, modular, and reusable implementation.

A Harshad (Niven) number is a number that is divisible by the sum of its digits.

For example:

• 18 → 1 + 8 = 9 → 18 ÷ 9 = 2 □ (Harshad Number)

• 19 → 1 + 9 = 10 → 19 ÷ 10 ≠ integer □ (Not Harshad)

Problem Statement

The current implementation:

• Mixes logic and input handling

• Uses redundant variables

• Does not use reusable functions properly

• Returns print statements instead of boolean values

• Lacks documentation

You must refactor the code to follow clean coding principles.

```
# Harshad Number Checker (Unstructured Version)
num = int(input("Enter a number: "))
temp = num
sum_digits = 0
while temp > 0:
digit = temp % 10
sum_digits = sum_digits + digit
temp = temp // 10
if sum_digits != 0:
if num % sum_digits == 0:
print("True")
else:
print("False")
else:
print("False")
```

You must:

1. Create a reusable function: is_harshad(number)

2. The function must:

o Accept an integer parameter.

o Return True if the number is divisible by the sum of its digits.

o Return False otherwise.

3. Separate user input from core logic.

4. Add proper docstrings.

5. Improve readability and maintainability.

6. Ensure the program handles edge cases (e.g., 0, negative numbers).

Code :

```
# Harshad number checker(unstructured version) and needs refactoring.

num = int(input("Enter number: "))

temp = num

sum_of_digits = 0

while temp > 0:

    digit = temp % 10

    sum_of_digits += digit

    temp //= 10

if sum_of_digits != 0:

    if num % sum_of_digits == 0:

        print("True")

    else:

        print("False")

else:

    print("False")

# Refactor the Harshad number checker into a clean, reusable function
is_harshad_number(num) with proper docstrings, optimize the logic for calculating the sum
of digits, and add test cases.

def is_harshad_number(num):

    """

    Check if a number is a Harshad number.


    Args:

        num (int): The number to check.

    Returns:

        bool: True if the number is a Harshad number, False otherwise.

    """

    if num <= 0:

        return False
    sum_of_digits = sum(int(digit) for digit in str(num))
    return num % sum_of_digits == 0
# Test cases
if __name__ == "__main__":
    print(is_harshad_number(18))  # Output: True
    print(is_harshad_number(19))  # Output: False
    print(is_harshad_number(21))  # Output: True
    print(is_harshad_number(22))  # Output: False
    print(is_harshad_number(0))   # Output: False
    print(is_harshad_number(-10)) # Output: False
```

Output :

```
Enter number: 18
True
True
False
True
False
False
False
```

Task 12 – Refactoring the Factorial Trailing Zeros Program

Refactor the given poorly structured Python script into a clean, modular, and efficient implementation.

The program calculates the number of trailing zeros in n! (factorial of n).

Problem Statement

The current implementation:

- Calculates the full factorial (inefficient for large n)
- Mixes input handling with business logic

• Uses print statements instead of return values

• Lacks modular structure and documentation

You must refactor the code to improve efficiency, readability, and maintainability.

```
# Factorial Trailing Zeros (Unstructured Version)
n = int(input("Enter a number: "))
fact = 1
i = 1
while i <= n:
fact = fact * i
i = i + 1
count = 0
while fact % 10 == 0:
count = count + 1
fact = fact // 10
print("Trailing zeros:", count)
```

You must:

1. Create a reusable function: count_trailing_zeros(n)
2. The function must:
   o Accept a non-negative integer n.
   o Return the number of trailing zeros in n!.
3. Do NOT compute the full factorial.
4. Use an optimized mathematical approach (count multiples of 5).
5. Add proper docstrings.
6. Separate user interaction from core logic.
7. Handle edge cases (e.g., negative numbers, zero).

Code :

```
# factorial Trailing zeroes(unstructured version) and needs refactoring.
```

```
n = int(input("Enter number: "))

fact = 1

i = 1

while i <= n:

    fact *= i

    i += 1

count = 0

while fact % 10 == 0:

    count += 1

    fact //= 10

print("Trailing zeroes in factorial:", count)

# Refactor the factorial trailing zeroes calculator into a clean, reusable function
count_trailing_zeroes(n) with proper docstrings, optimize the logic to count trailing zeroes
without calculating the factorial, and add test cases.

"""

Factorial Trailing Zeroes Module

This module provides a function to count the number of trailing zeroes in the factorial of a
given number.

"""

def count_trailing_zeroes(n):

    """

    Count the number of trailing zeroes in the factorial of a given number.


    Args:

        n (int): The number for which to count trailing zeroes in its factorial.


    Returns:

        int: The number of trailing zeroes in n!.

    """

    if n < 0:
      raise ValueError("Input must be a non-negative integer.")
    count = 0
    divisor = 5
    while n >= divisor:
        count += n // divisor
        divisor *= 5
    return count
if __name__ == "__main__":
    try:
      number = int(input("Enter number: "))
      result = count_trailing_zeroes(number)
      print(f"Trailing zeroes in factorial of {number}: {result}")
    except ValueError as e:
      print("Error:", e)
```

Output :

```
Enter number: 500
Trailing zeroes in factorial: 124
Enter number: 20
Trailing zeroes in factorial of 20: 4
```

Task 13 (Collatz Sequence Generator – Test Case Design)

• **Function: Generate Collatz sequence until reaching 1.**

• Test Cases to Design:

• Normal: 6 → [6,3,10,5,16,8,4,2,1]

• Edge: 1 → [1]

• Negative: -5

• Large: 27 (well-known long sequence)

• Requirement: Validate correctness with pytest.

Explanation:

We need to write a function that:

• Takes an integer n as input.

• Generates the Collatz sequence (also called the 3n+1 sequence).

• The rules are:

o If n is even → next = n / 2.

o If n is odd → next = 3n + 1.

• Repeat until we reach 1.

• Return the full sequence as a list.

Example

Input: 6

Steps:

• 6 (even → 6/2 = 3)

• 3 (odd → 3*3+1 = 10)

• 10 (even → 10/2 = 5)

• 5 (odd → 3*5+1 = 16)

• 16 (even → 16/2 = 8)

• 8 (even → 8/2 = 4)

• 4 (even → 4/2 = 2)

• 2 (even → 2/2 = 1)

Output:

```
[6, 3, 10, 5, 16, 8, 4, 2, 1]
```

Code :

```
"""
collatz sequence module
This module provides a function to generate the Collatz sequence for a given number.
"""
```

```
def collatz_sequence(n):
    """
    Generate the Collatz sequence for a given number.


    Args:
        n (int): The starting number for the Collatz sequence.
    Returns:
        list: A list containing the Collatz sequence starting from n and ending at 1.
    """
    if n < 1:
        raise ValueError("Input must be a positive integer.")
    sequence = [n]
    while n != 1:
        if n % 2 == 0:
            n //= 2
        else:
            n = 3 * n + 1
        sequence.append(n)
    return sequence
    import pytest
from collatz import collatz_sequence
def test_normal_cases():
    assert collatz_sequence(6) == [6, 3, 10, 5, 16, 8, 4, 2, 1]
    assert collatz_sequence(19) == [19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5,
16, 8, 4, 2, 1]
def test_edge_cases():
    assert collatz_sequence(1) == [1]
    with pytest.raises(ValueError):
        collatz_sequence(0)
    with pytest.raises(ValueError):
        collatz_sequence(-5)
def test_large_case():
    assert collatz_sequence(100) == [100, 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52,
26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
def test_negative_case():
        with pytest.raises(ValueError):
            collatz_sequence(-1)
```

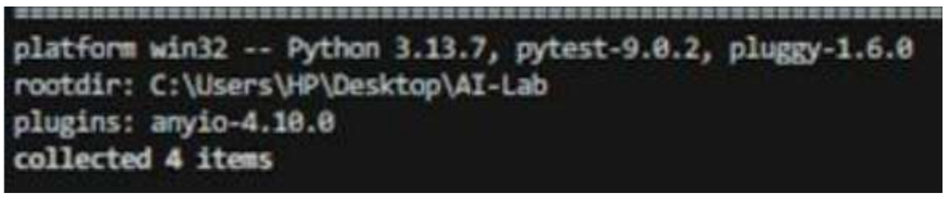
Code:

```
"""
collatz sequence module
This module provides a function to generate the Collatz sequence for a given number.
"""
def collatz_sequence(n):
    """
    Generate the Collatz sequence for a given number.


    Args:
        n (int): The starting number for the Collatz sequence.
    Returns:
        list: A list containing the Collatz sequence starting from n and ending at 1.
    """
    if n < 1:
        raise ValueError("Input must be a positive integer.")
    sequence = [n]
    while n != 1:
        if n % 2 == 0:
            n //= 2
        else:
            n = 3 * n + 1
        sequence.append(n)
    return sequence
import pytest
from collatz import collatz_sequence


def test_normal_cases():
    assert collatz_sequence(6) == [6, 3, 10, 5, 16, 8, 4, 2, 1]
def test_edge_cases():
    assert collatz_sequence(1) == [1]
def test_large_case():
    result = collatz_sequence(19)
    assert result[-1] == 1
    assert result[0] == 19
def test_negative_case():
        with pytest.raises(ValueError):
            collatz_sequence(-5)
```

Output :



```
platform win32 -- Python 3.13.7, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\HP\Desktop\AI-Lab
plugins: anyio-4.10.0
collected 4 items
```

Task 14 (Lucas Number Sequence – Test Case Design)

• Function: Generate Lucas sequence up to n terms.

(Starts with 2,1, then Fn = Fn-1 + Fn-2)

- Test Cases to Design:
- Normal: 5 → [2, 1, 3, 4, 7]
- Edge: 1 → [2]
- Negative: -5 → Error
- Large: 10 (last element = 76).
- Requirement: Validate correctness with pytest.

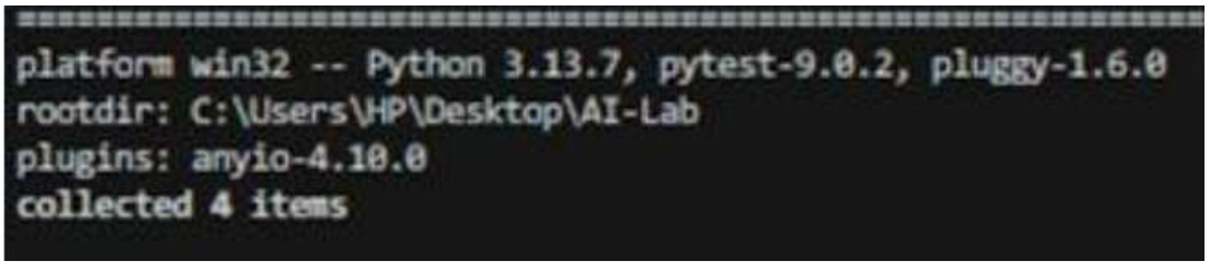
Code :

```
def lucas_sequence(n):
    """
    Generate the Lucas sequence up to n terms.

    Args:
        n (int): The number of terms in the Lucas sequence to generate.

    Returns:
        list: A list containing the Lucas sequence up to n terms.
    """
    if n < 1:
        raise ValueError("Input must be a positive integer.")
    sequence = []
    for i in range(n):
        if i == 0:
            sequence.append(2)
        elif i == 1:
            sequence.append(1)
        else:
            next_lucas = sequence[i-1] + sequence[i-2]
            sequence.append(next_lucas)
    return sequence
    import pytest
    from lucas import lucas_sequence
def test_normal_case():
    assert lucas_sequence(5) == [2, 1, 3, 4, 7]
def test_edge_case():
    assert lucas_sequence(1) == [2]
def test_negative_case():
    with pytest.raises(ValueError):
        lucas_sequence(-5)
def test_large_case():
    result = lucas_sequence(10)
    assert result[-1] == 76
    assert result[0] == 2
```

Output :



```
==================================================================
platform win32 -- Python 3.13.7, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\HP\Desktop\AI-Lab
plugins: anyio-4.10.0
collected 4 items
```

Task 15 (Vowel & Consonant Counter – Test Case Design)

• Function: Count vowels and consonants in string.

• Test Cases to Design:

• Normal: "hello" → (2,3)

• Edge: "" → (0,0)

• Only vowels: "aeiou" → (5,0)

Large: Long text

• Requirement: Validate correctness with pytest.

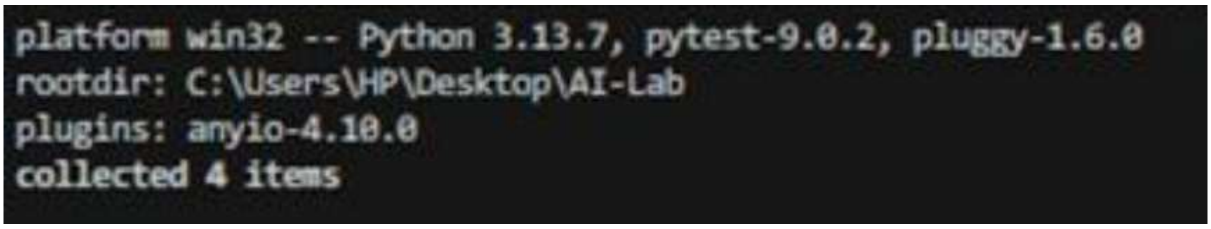
Code :

```
"""
vowel and consonant counter module
this module provides a function to count vowels
and consonats in a gievn string
"""
def count_vowels_consonants(s):
    """
    Count the number of vowels and consonants in a given string.

    Args:
        s (str): The input string to analyze.
    Returns:
        tuple: A tuple containing the count of vowels and consonants in the format (vowel_count, consonant_count).
    """
    vowels = 'aeiouAEIOU'
    vowel_count = sum(1 for char in s if char in vowels)
    consonant_count = sum(1 for char in s if char.isalpha() and char not in vowels)
    return vowel_count, consonant_count
import pytest
from vowel_consonant import count_vowels_consonants
def test_normal_case():
    assert count_vowels_consonants("hello") == (2, 3)
def test_edge_case():
    assert count_vowels_consonants("") == (0, 0)
```

```
def test_only_vowels():
    assert count_vowels_consonants("aeiou") == (5, 0)
def test_large_case():
    long_text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit."
    assert count_vowels_consonants(long_text) == (15, 24)
```

output :



```
platform win32 -- Python 3.13.7, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\HP\Desktop\AI-Lab
plugins: anyio-4.10.0
collected 4 items
```