Lab Assignment-13

Name: Anand

Hallticket:2303A51090

Batch-02

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

screenshots:

assg_13.py > validate_dimensions

```python
def validate_dimensions(length, width):
    """
    Validate that dimensions are positive numbers.

    Args:
        length: The length of the rectangle
        width: The width of the rectangle

    Returns:
        bool: True if valid, raises ValueError otherwise
    """
    if not isinstance(length, (int, float)) or not isinstance(width, (int, float)):
        raise ValueError("Dimensions must be numbers")
    if length <= 0 or width <= 0:
        raise ValueError("Dimensions must be positive")
    return True


def calculate_area(length, width):
    """
    Calculate the area of a rectangle.

    Args:
        length: The length of the rectangle
        width: The width of the rectangle

    Returns:
        float: The area of the rectangle
    """
    validate_dimensions(length, width)
    return length * width


def calculate_perimeter(length, width):
    """
    Calculate the perimeter of a rectangle.

    Args:
        length: The length of the rectangle
        width: The width of the rectangle

    Returns:
        float: The perimeter of the rectangle
    """
    validate_dimensions(length, width)
    return 2 * (length + width)


def print_rectangle_info(length, width):
    """
    Print area and perimeter of a rectangle.

    Args:
        length: The length of the rectangle
        width: The width of the rectangle
    """
    try:
        area = calculate_area(length, width)
        perimeter = calculate_perimeter(length, width)
        print(f"Area of Rectangle: {area}")
        print(f"Perimeter of Rectangle: {perimeter}")
    except ValueError as e:
        print(f"Error: {e}")


# Usage
print_rectangle_info(5, 10)
print_rectangle_info(7, 12)
print_rectangle_info(10, 15)
```

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
Area of Rectangle: 50
Perimeter of Rectangle: 30
Area of Rectangle: 84
Perimeter of Rectangle: 38
Area of Rectangle: 150
Perimeter of Rectangle: 50
PS C:\Users\arell\Music\aiac>
```

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:

```
# 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

screenshots:

```python
#refactor the code identify the repleated or related logics from the code extract into the reusable function and use that function in the main code to avoid code repetition and make the code more modular and maintainable
def calculate_total_price(price):
    """
    Calculate the total price including tax.
    This function takes a price and calculates the total cost by adding an 18% tax
    to the original price.
    Args:
        price (int or float): The base price before tax. Must be a non-negative number.
    Returns:
        float: The total price including 18% tax.
    Raises:
        TypeError: If price is not a number (int or float).
        ValueError: If price is negative.
    Example:
        >>> calculate_total_price(100)
        118.0
        >>> calculate_total_price(50.5)
        59.59
    """
    if not isinstance(price, (int, float)):
        raise TypeError("Price must be a number.")
    if price < 0:
        raise ValueError("Price must be non-negative.")

    tax = price * 0.18
    total = price + tax
    return total
price = 1000
try:
    total_price = calculate_total_price(price)
    print("Total Price:", total_price)
except (TypeError, ValueError) as e:
    print(f"Error: {e}")
price = 500
try:
    total_price = calculate_total_price(price)
    print("Total Price:", total_price)
except (TypeError, ValueError) as e:
    print(f"Error: {e}")
price = 500
try:
    total_price = calculate_total_price(price)
    print("Total Price:", total_price)
except (TypeError, ValueError) as e:
    print(f"Error: {e}")

```

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
Total Price: 1180.0
Total Price: 590.0
Total Price: 590.0
PS C:\Users\arell\Music\aiac>
```

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:

- Accept marks as a parameter.
- Return the corresponding grade as a string.
- 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:
   - The class
   - The method (with parameter and return descriptions)

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

• Given code:

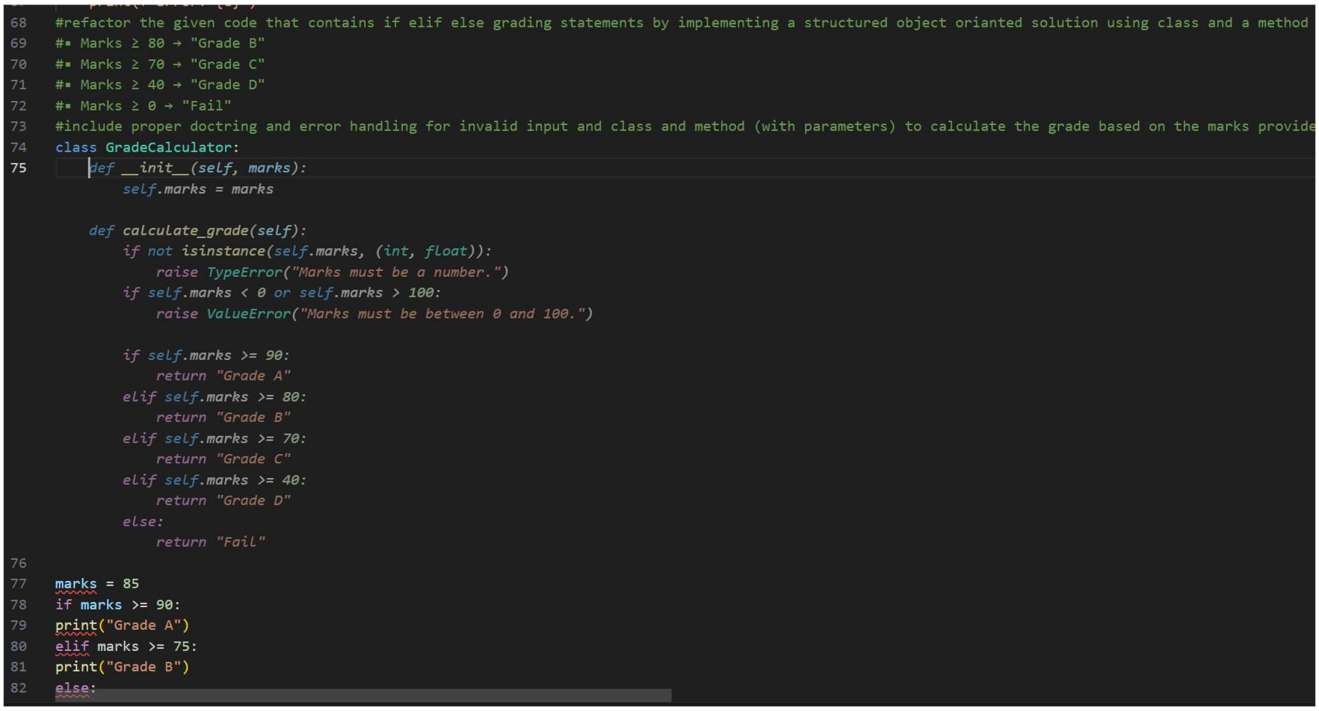
```
marks = 85
if marks >= 90:
print("Grade A")
elif marks >= 75:
print("Grade B")
else:
print("Grade C")
marks = 72
if marks >= 90:
print("Grade A")
elif marks >= 75:
print("Grade B")
else:
print("Grade C")
```

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.

Screenshots:



```
#refactor the given code that contains if elif else grading statements by implementing a structured object orianted solution using class and a method
#* Marks ≥ 80 → "Grade B"
#* Marks ≥ 70 → "Grade C"
#* Marks ≥ 40 → "Grade D"
#* Marks ≥ 0 → "Fail"
#include proper doctring and error handling for invalid input and class and method (with parameters) to calculate the grade based on the marks provide
class GradeCalculator:
    def __init__(self, marks):
        self.marks = marks

    def calculate_grade(self):
        if not isinstance(self.marks, (int, float)):
            raise TypeError("Marks must be a number.")
        if self.marks < 0 or self.marks > 100:
            raise ValueError("Marks must be between 0 and 100.")

        if self.marks >= 90:
            return "Grade A"
        elif self.marks >= 80:
            return "Grade B"
        elif self.marks >= 70:
            return "Grade C"
        elif self.marks >= 40:
            return "Grade D"
        else:
            return "Fail"

marks = 85
if marks >= 90:
print("Grade A")
elif marks >= 75:
print("Grade B")
else:
```

```python
class GradeCalculator:
    """GradeCalculator Module
    A utility module for calculating student grades based on numerical marks.
    Classes:
        GradeCalculator: Evaluates and assigns letter grades to student marks.
    Example:
        >>> calculator = GradeCalculator(85)
        >>> calculator.calculate_grade()
        'Grade B'
    Note:
        This module assumes marks are on a 0-100 scale. Grades are assigned
        according to the following scale:
        - 90-100: Grade A
        - 80-89: Grade B
        - 70-79: Grade C
        - 40-69: Grade D
        - 0-39: Fail
    Class to calculate student grades based on marks."""

    def __init__(self, marks):
        """Initialize with marks value."""
        self.marks = marks

    def calculate_grade(self):
        """Calculate and return grade based on marks with proper validation.
        Raises:
            TypeError: If marks is not a number.
            ValueError: If marks is out of valid range (0-100).

            """
        if not isinstance(self.marks, (int, float)):
            raise TypeError("Marks must be a number.")
        if self.marks < 0 or self.marks > 100:
            raise ValueError("Marks must be between 0 and 100.")

        if self.marks >= 90:
            return "Grade A"
        elif self.marks >= 80:
            return "Grade B"
        elif self.marks >= 70:
            return "Grade C"
        elif self.marks >= 40:
            return "Grade D"
        else:
            return "Fail"


# Usage with error handling
test_marks = [85, 72, 35, 95]

for marks in test_marks:
    try:
        grade_calculator = GradeCalculator(marks)
        grade = grade_calculator.calculate_grade()
        print(f"Marks: {marks} → {grade}")
    except (TypeError, ValueError) as e:
        print(f"Error: {e}")
```

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
Marks: 85 → Grade B
Marks: 72 → Grade C
Marks: 35 → Fail
Marks: 95 → Grade A
PS C:\Users\arell\Music\aiac>
```

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().

Screenshots:

```
#identify input ,processing  and output sections and convert them into a separate function for each section like and add docstrings to each function
def get_input():
    """Function to get input from the user."""
    try:
        num = int(input("Enter number: "))
        return num
    except ValueError:
        print("Invalid input. Please enter a valid integer.")
        return None
def process_input(num):
    """Function to process the input and calculate the square."""
    if num is not None:
        return num ** 2
    return None
def display_output(result):
    """Function to display the output."""
    if result is not None:
        print(f"The square of the number is: {result}")
    else:
        print("No result to display.")

^
```

```
#identify input ,processing  and output sections and convert them into a separate function for each section like and add docstrings to each function
def get_input():
    """Function to get input from the user."""
    try:
        num = int(input("Enter number: "))
        return num
    except ValueError:
        print("Invalid input. Please enter a valid integer.")
        return None
def process_input(num):
    """Function to process the input and calculate the square."""
    if num is not None:
        return num ** 2
    return None
def display_output(result):
    """Function to display the output."""
    if result is not None:
        print(f"The square of the number is: {result}")
    else:
        print("No valid result to display.")
def main():
    """Main function to orchestrate the input, processing, and output."""
    num = get_input()
    result = process_input(num)
    display_output(result)
if __name__ == "__main__":    main()
```

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
Enter number: 20
The square of the number is: 400
PS C:\Users\arell\Music\aiac>
```

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:

A class like EmployeeSalaryCalculator with methods and attributes.

Screenshots:

```python
#refactor the code into the procedural code into a class -based design main focus on the object ooriented line encapsulation

class SalaryCalculator:
    """Class to calculate salary and tax with encapsulation."""

    def __init__(self, salary):
        """Initialize SalaryCalculator with salary amount.

        Args:
            salary: The base salary amount
        """
        self._salary = salary

    def _validate_salary(self):
        """Validate that salary is a positive number.

        Raises:
            ValueError: If salary is not positive or not a number
        """
        if not isinstance(self._salary, (int, float)):
            raise ValueError("Salary must be a number.")
        if self._salary < 0:
            raise ValueError("Salary must be non-negative.")

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

        Returns:
            float: Tax amount (20% of salary)
        """
        self._validate_salary()
        return self._salary * 0.2

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

        Returns:
            float: Net salary (salary - tax)
        """
        tax = self.calculate_tax()
        return self._salary - tax

    def display_salary_info(self):
        """Display salary breakdown."""
        try:
            tax = self.calculate_tax()
            net = self.calculate_net_salary()
            print(f"Gross Salary: {self._salary}")
            print(f"Tax (20%): {tax}")
            print(f"Net Salary: {net}")
        except ValueError as e:
            print(f"Error: {e}")

# Usage
salary_calculator = SalaryCalculator(50000)
salary_calculator.display_salary_info()
```

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
Gross Salary: 50000
Tax (20%): 10000.0
Net Salary: 40000.0
PS C:\Users\arell\Music\aiac>
```

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

screenshots:

```
# refactor the code to inefficcient liner searches using apporiate data and focus on the time complexity and data structures  edit tghe code to the b

# Using a set for O(1) average-case lookup instead of O(n) list search
users = {"admin", "guest", "editor", "viewer"}
name = input("Enter username: ")
found = name in users
print("Access Granted" if found else "Access Denied")

```

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
Enter username: admin
Access Granted
PS C:\Users\arell\Music\aiac>
```

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.")
```

screenshots:

```python
"""
Library Management System - Refactored Module
Provides functions to manage a library database with proper documentation.
"""

library_db = {}

def add_book(title, author, isbn):
    """
    Add a new book to the library database.

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

    Returns:
        bool: True if book added successfully, False if book already exists.

    Raises:
        ValueError: If any parameter is empty or invalid.
    """
    if not title or not author or not isbn:
        raise ValueError("Title, author, and ISBN cannot be empty.")

    if isbn not in library_db:
        library_db[isbn] = {"title": title, "author": author}
        print("Book added successfully.")
        return True
    else:
        print("Book already exists.")
        return False

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

    Args:
        isbn (str): The ISBN of the book to remove.

    Returns:
        bool: True if book removed successfully, False if book not found.
    """
    if isbn in library_db:
        del library_db[isbn]
        print("Book removed successfully.")
        return True
    else:
        print("Book not found.")
        return False

def search_book(isbn):
    """
    Search for a book in the library database by ISBN.

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

    Returns:
        dict: Book details (title, author) if found, None otherwise.
    """
    if isbn in library_db:
        print("Book Found:", library_db[isbn])
        return library_db[isbn]
    else:
        print("Book not found.")
        return None

# Usage
if __name__ == "__main__":
    add_book("Python Basics", "John Doe", "101")
    add_book("AI Fundamentals", "Jane Smith", "102")
    search_book("101")
    remove_book("101")
    search_book("101")
```

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
Book added successfully.
Book added successfully.
Book Found: {'title': 'Python Basics', 'author': 'John Doe'}
Book removed successfully.
Book not found.
PS C:\Users\arell\Music\aiac>
```

index
library c:\users\arell\music\aiac\library.py

Functions

add_book(title, author, isbn)
Add a new book to the library database.

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

Returns:
bool: True if book added successfully, False if book already exists.

Raises:
ValueError: If any parameter is empty or invalid.

remove_book(isbn)
Remove a book from the library database by ISBN.

Args:
isbn (str): The ISBN of the book to remove.

Returns:
bool: True if book removed successfully, False if book not found.

search_book(isbn)
Search for a book in the library database by ISBN.

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

Returns:
dict: Book details (title, author) if found, None otherwise.

Data
library_db = {}

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
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.

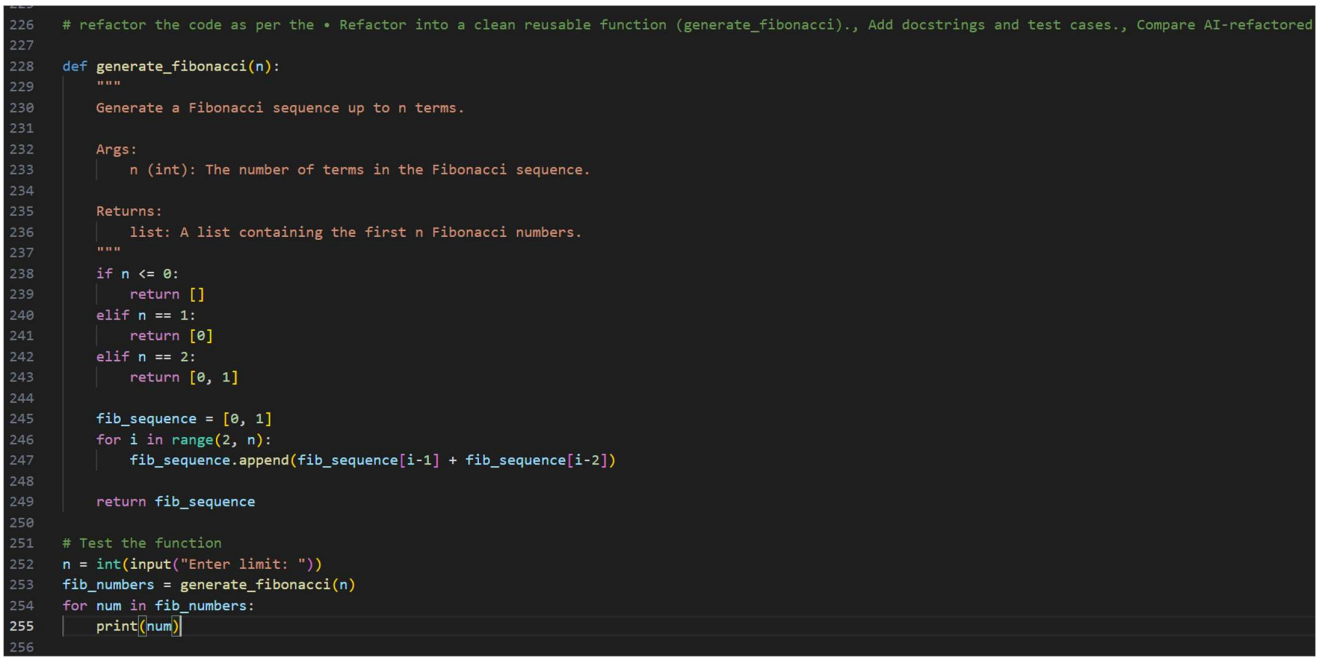
Task for Students:

- 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
```

screenshots:



```python
# refactor the code as per the • Refactor into a clean reusable function (generate_fibonacci)., Add docstrings and test cases., Compare AI-refactored

def generate_fibonacci(n):
    """
    Generate a Fibonacci sequence up to n terms.

    Args:
        n (int): The number of terms in the Fibonacci sequence.

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

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

    return fib_sequence

# Test the function
n = int(input("Enter limit: "))
fib_numbers = generate_fibonacci(n)
for num in fib_numbers:
    print(num)
```

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
Enter limit: 5
0
1
1
2
3
PS C:\Users\arell\Music\aiac>
```

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")

screenshots:

```python
# refactor the code as per the instructions 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.
def is_prime(n):
    """
    Check if a number is prime.

    Args:
        n (int): The number to check.

    Returns:
        bool: True if the number is prime, False otherwise.
    """
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

def is_twin_prime(p1, p2):
    """
    Check if two numbers are twin primes.

    Args:
        p1 (int): First number.
        p2 (int): Second number.

    Returns:
        bool: True if both numbers are prime and their difference is 2, False otherwise.
    """
    return is_prime(p1) and is_prime(p2) and abs(p1 - p2) == 2

# Generate a list of twin primes in a given range
def generate_twin_primes(start, end):
    """
    Generate a list of twin primes in a given range.

    Args:
        start (int): Start of the range (inclusive).
        end (int): End of the range (inclusive).

    Returns:
        list: A list of tuples representing twin primes in the given range.
    """
    twin_primes = []
    for i in range(start, end - 1):
        if is_twin_prime(i, i + 2):
            twin_primes.append((i, i + 2))
    return twin_primes

# Example usage
a = 11
b = 13
fa = 0
if is_twin_prime(a, b):
    print(f"{a} and {b} are twin primes.")
else:
    print(f"{a} and {b} are not twin primes.")
start_range = 1
end_range = 100
twin_prime_list = generate_twin_primes(start_range, end_range)
print(f"Twin primes between {start_range} and {end_range}: {twin_prime_list}")

```

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
11 and 13 are twin primes.
Twin primes between 1 and 100: [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
PS C:\Users\arell\Music\aiac>
```

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")

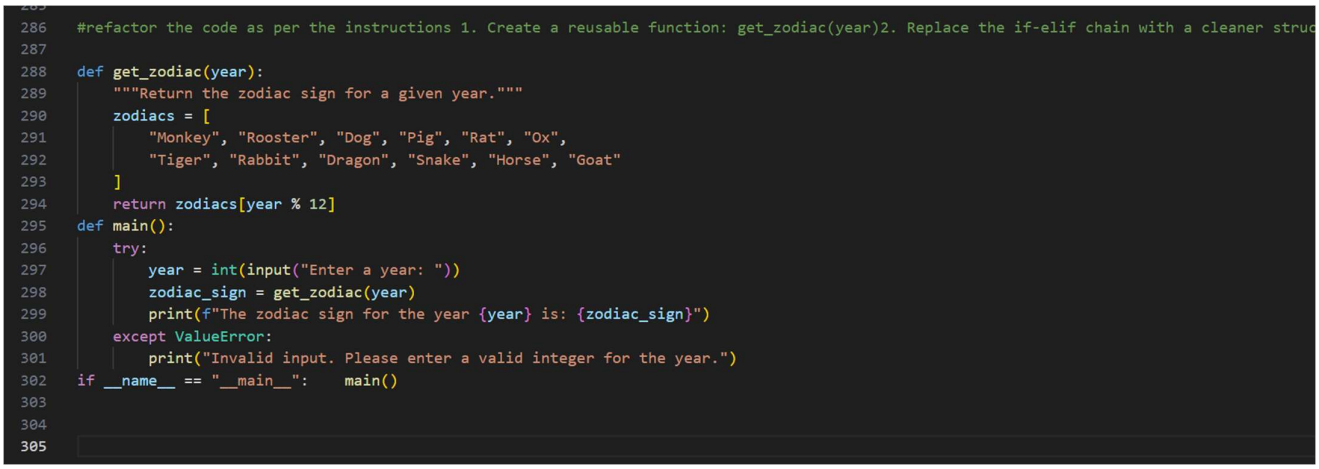
elif year % 12 == 11:

print("Goat")

You must:

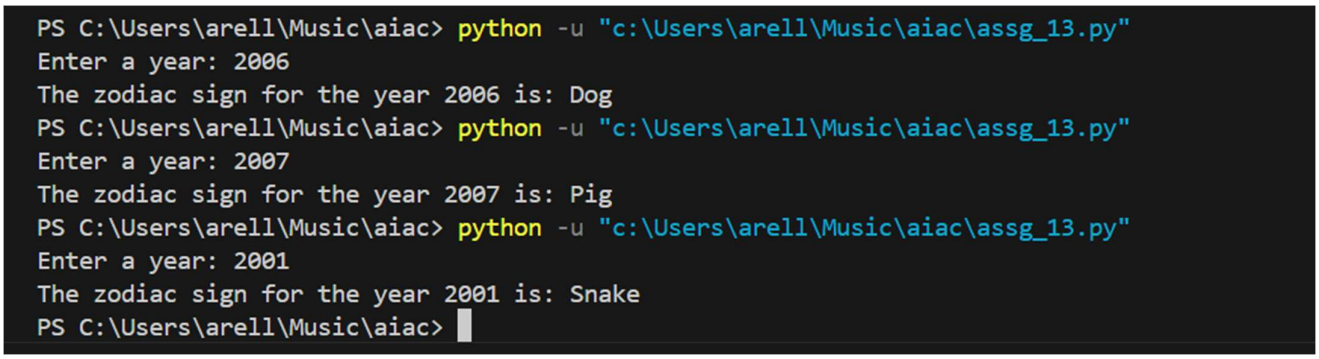
1. Create a reusable function: get_zodiac(year)
2. Replace the if-elif chain with a cleaner structure (e.g., list or dictionary).
3. Add proper docstrings.
4. Separate input handling from logic.
5. Improve readability and maintainability.
6. Ensure output remains correct.

Screenshots:



```
#refactor the code as per the instructions 1. Create a reusable function: get_zodiac(year)2. Replace the if-elif chain with a cleaner struc

def get_zodiac(year):
    """Return the zodiac sign for a given year."""
    zodiacs = [
        "Monkey", "Rooster", "Dog", "Pig", "Rat", "Ox",
        "Tiger", "Rabbit", "Dragon", "Snake", "Horse", "Goat"
    ]
    return zodiacs[year % 12]
def main():
    try:
        year = int(input("Enter a year: "))
        zodiac_sign = get_zodiac(year)
        print(f"The zodiac sign for the year {year} is: {zodiac_sign}")
    except ValueError:
        print("Invalid input. Please enter a valid integer for the year.")
if __name__ == "__main__":    main()
```

Output:



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.

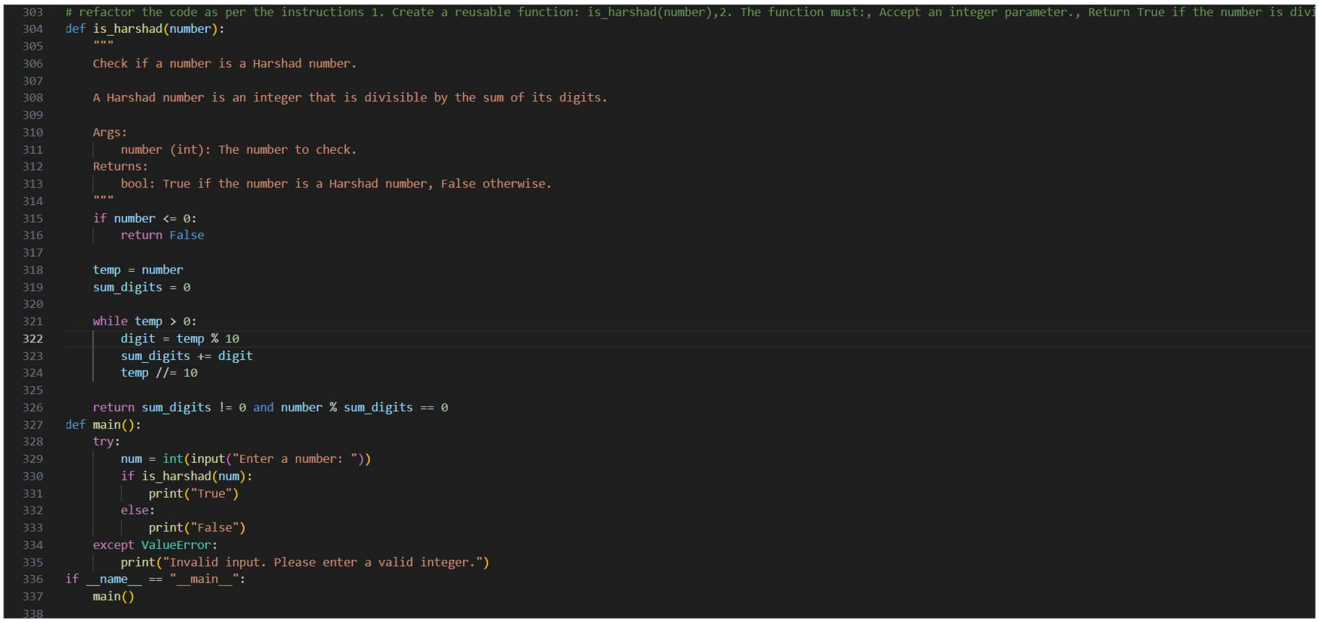
o Return False otherwise.

3. Separate user input from core logic.

4. Add proper docstrings.

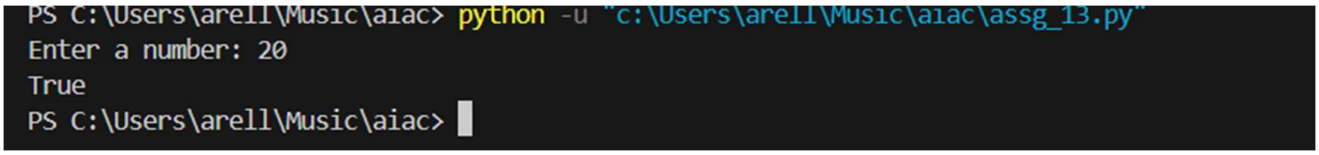
5. Improve readability and maintainability.

Screenshots:



```python
# refactor the code as per the instructions 1. Create a reusable function: is_harshad(number),2. The function must:, Accept an integer parameter., Return True if the number is divi
def is_harshad(number):
    """
    Check if a number is a Harshad number.

    A Harshad number is an integer that is divisible by the sum of its digits.

    Args:
        number (int): The number to check.
    Returns:
        bool: True if the number is a Harshad number, False otherwise.
    """
    if number <= 0:
        return False

    temp = number
    sum_digits = 0

    while temp > 0:
        digit = temp % 10
        sum_digits += digit
        temp //= 10

    return sum_digits != 0 and number % sum_digits == 0
def main():
    try:
        num = int(input("Enter a number: "))
        if is_harshad(num):
            print("True")
        else:
            print("False")
    except ValueError:
        print("Invalid input. Please enter a valid integer.")
if __name__ == "__main__":
    main()
```

Output:



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).

Screenshots:

```python
#REFACTOR THE CODE AS PER THE INSTRUCTIONS  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 i
def count_trailing_zeros(n):

    if n < 0:
        raise ValueError("Input must be a non-negative integer.")

    count = 0
    power_of_5 = 5

    while n >= power_of_5:
        count += n // power_of_5
        power_of_5 *= 5

    return count
def main():
    try:
        n = int(input("Enter a non-negative integer: "))
        zeros = count_trailing_zeros(n)
        print(f"The number of trailing zeros in {n}! is: {zeros}")
    except ValueError as e:
        print(f"Error: {e}")
if __name__ == "__main__":
    main()

```

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
Enter a non-negative integer: 100
The number of trailing zeros in 100! is: 24
PS C:\Users\arell\Music\aiac>
```

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:
    - If n is even → next = n / 2.
    - 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]

Screenshots:

```python
"""
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.

"""
def generate_collatz_sequence(n):
    """
    Generate the Collatz sequence for a given positive integer.

    Args:
        n: A positive integer

    Returns:
        A list representing the Collatz sequence

    Raises:
        ValueError: If n is not a positive integer
    """
    if not isinstance(n, int) or n <= 0:
        raise ValueError("Input must be a positive integer.")

    sequence = []

    while n != 1:
        sequence.append(n)
        if n % 2 == 0:
            n //= 2
        else:
            n = 3 * n + 1

    sequence.append(1)
    return sequence

def main():
    try:
        n = int(input("Enter a positive integer: "))
        collatz_sequence = generate_collatz_sequence(n)
        print(f"Collatz sequence for {n}: {collatz_sequence}")
    except ValueError as e:
        print(f"Error: {e}")

if __name__ == "__main__":
    main()

# Test cases
assert generate_collatz_sequence(6) == [6, 3, 10, 5, 16, 8, 4, 2, 1]
assert generate_collatz_sequence(19) == [19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
assert generate_collatz_sequence(1) == [1]

try:
    generate_collatz_sequence(0)
    assert False, "Should raise ValueError"
except ValueError:
    pass

try:
    generate_collatz_sequence(-5)
    assert False, "Should raise ValueError"
except ValueError:
    pass

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

Output:

```
PS C:\Users\arell\Music\aiac> python -u "c:\Users\arell\Music\aiac\assg_13.py"
Enter a positive integer: 200
Collatz sequence for 200: [200, 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]
All tests passed!
PS C:\Users\arell\Music\aiac>
```

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

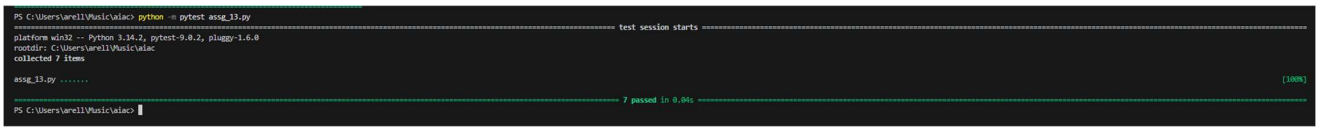
• Large: 10 (last element = 76).

• Requirement: Validate correctness with pytest.

Screenshots:

```python
def generate_lucas_sequence(n):
    """
    Generate Lucas sequence up to n terms.

    Args:
        n (int): Number of terms to generate

    Returns:
        list: Lucas sequence up to n terms

    Raises:
        ValueError: If n is not a positive integer
    """
    if not isinstance(n, int):
        raise TypeError("Input must be an integer.")
    if n <= 0:
        raise ValueError("Input must be a positive integer.")

    sequence = []
    a, b = 2, 1

    for _ in range(n):
        sequence.append(a)
        a, b = b, a + b

    return sequence


# Test cases using pytest
def test_generate_lucas_normal():
    """Test normal case with 5 terms"""
    assert generate_lucas_sequence(5) == [2, 1, 3, 4, 7]

def test_generate_lucas_single_term():
    """Test edge case with 1 term"""
    assert generate_lucas_sequence(1) == [2]

def test_generate_lucas_large():
    """Test with 10 terms"""
    assert generate_lucas_sequence(10) == [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]

def test_generate_lucas_negative():
    """Test error handling for negative input"""
    with pytest.raises(ValueError):
        generate_lucas_sequence(-5)

def test_generate_lucas_zero():
    """Test error handling for zero input"""
    with pytest.raises(ValueError):
        generate_lucas_sequence(0)

def test_generate_lucas_non_integer():
    """Test error handling for non-integer input"""
    with pytest.raises(TypeError):
        generate_lucas_sequence(5.5)

def test_generate_lucas_two_terms():
    """Test with 2 terms"""
    assert generate_lucas_sequence(2) == [2, 1]

if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

Output:



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.

Screenshots:

```
"""
• 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.

"""
def count_vowels_consonants(s):
    """
    Count vowels and consonants in a string.

    Args:
        s (str): Input string to analyze.

    Returns:
        tuple: (vowel_count, consonant_count)

    Raises:
        TypeError: If input is not a string.
    """
    if not isinstance(s, str):
        raise TypeError("Input must be a string.")

    vowels = set("aeiouAEIOU")
    vowel_count = 0
    consonant_count = 0

    for char in s:
        if char.isalpha():
            if char in vowels:
                vowel_count += 1
            else:
                consonant_count += 1

    return vowel_count, consonant_count


# pytest test cases

def test_normal_case():
    """Test with normal string."""
    assert count_vowels_consonants("hello") == (2, 3)

def test_empty_string():
    """Test with empty string."""
    assert count_vowels_consonants("") == (0, 0)

def test_only_vowels():
    """Test with only vowels."""
    assert count_vowels_consonants("aeiou") == (5, 0)

def test_long_text():
    """Test with long text."""
    assert count_vowels_consonants("The quick brown fox jumps over the lazy dog") == (11, 24)

def test_only_consonants():
    """Test with only consonants."""
    assert count_vowels_consonants("bcdfg") == (0, 5)

def test_with_numbers_and_special_chars():
    """Test with numbers and special characters."""
    assert count_vowels_consonants("Hello123!@#") == (2, 3)

def test_uppercase():
    """Test with uppercase letters."""
    assert count_vowels_consonants("AEIOU") == (5, 0)

def test_mixed_case():
    """Test with mixed case."""
    assert count_vowels_consonants("HeLLo") == (2, 3)

def test_invalid_input():
    """Test with invalid input type."""
    with pytest.raises(TypeError):
        count_vowels_consonants(123)

def test_invalid_input_none():
    """Test with None input."""
    with pytest.raises(TypeError):
        count_vowels_consonants(None)


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

```

Output:

```
PS C:\Users\arell\Music\aiac> python -m pytest assg_13.py
============================================================================================================================================================================================================
========= test session starts ==========================================================================================================================================================================
====================================
platform win32 -- Python 3.14.2, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\arell\Music\aiac
collected 10 items

assg_13.py ..........

                                    [100%]

============================================================================================================================================================================================================
========== 10 passed in 0.05s ==========================================================================================================================================================================
====================================
PS C:\Users\arell\Music\aiac>
```