# AIAC Lab Assignment -13

## Code Refactoring Using AI Suggestions

Name : M.Hasini
Hallticket: 2303A51109
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:
  - Prompt AI to identify duplicate logic and replace it with functions or classes.
  - Ensure the refactored code maintains the same output.
  - 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:
  - Refactored code with a reusable function and no duplication.
  - Well documented code

Area of Rectangle :

week 13 > rectangle.py > ...

```python
#write refactored code for the above code and identify the duplicate logic and correct it
def area_of_rectangle(length: float, breadth: float) -> float:
    """
    This function calculates the area of a rectangle given its length and breadth.
    Parameters:
    length (float): The length of the rectangle.
    breadth (float): The breadth of the rectangle.
    Returns:
    float: The area of the rectangle.
    """
    return length * breadth
```

Perimeter of rectangle :

```
    return length * breadth
def perimeter_of_rectangle(length: float, breadth: float) -> float:
    """This function calculates the perimeter of a rectangle given its length and breadth.
    Parameters:
    length (float): The length of the rectangle.
    breadth (float): The breadth of the rectangle.
    Returns:
    float: The perimeter of the rectangle.
    """
    return 2 * (length + breadth)
```

Output :

```
print("Area of Rectangle:", area_of_rectangle(5, 10))
print("Perimeter of Rectangle:", perimeter_of_rectangle(5, 10))
print("Area of Rectangle:", area_of_rectangle(7, 12))
print("Perimeter of Rectangle:", perimeter_of_rectangle(7, 12))
print("Area of Rectangle:", area_of_rectangle(10, 15))
```

```
/AIAC/week 13/rectangle.py"
Area of Rectangle: 50
Perimeter of Rectangle: 30
Area of Rectangle: 84
Perimeter of Rectangle: 38
Area of Rectangle: 150
```

**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:
  - Identify repeated or related logic and extract it into reusable functions.
  - 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:
  - Code with a function calculate_total(price) that can be reused for multiple price inputs.
  - Well documented code

Refactored Code:

```
week 13 > total_price.py > ...
  print("Total Price:", total)
  """
  #write a refcatored code for the above code and use google style documentation for the explanation
  def calculate_total(price):
      """
      This function calculates the total price of an item including tax.

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

      Returns:
          float: The total price of the item including tax.
      """
      tax = price * 0.18
      total = price + tax
      return total
  print("Total Price:", calculate_total(250))
  print("Total Price:", calculate_total(500))
```

Output:

```
al_price.py"
Total Price: 295.0
Total Price: 590.0
PS C:\Users\User\OneDrive\Desktop\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.

Refactored Code:

week 13 > grade.py > GradeCalculator > calculate_grade

```
30  class GradeCalculator:
31      """
32      A class to calculate grades based on marks.
33      """
34      def calculate_grade(self, marks):
35          """
36          Calculates the grade based on the given marks.
37
38          Args:
39              marks (int): The marks obtained by a student.
40
41          Returns:
42              str: The grade corresponding to the marks.
43          """
44          if marks >= 90 and marks <= 100:
45              return "Grade A"
46          elif marks >= 80:
47              return "Grade B"
48          elif marks >= 70:
49              return "Grade C"
50          elif marks >= 40:
51              return "Grade D"
52          else:
53              return "Fail"
```

Output:

```
# Example usage
grade_calculator = GradeCalculator()
print(grade_calculator.calculate_grade(85))  # Output: Grade B
print(grade_calculator.calculate_grade(72))  # Output: Grade C
print(grade_calculator.calculate_grade(95))  # Output: Grade A
print(grade_calculator.calculate_grade(65))  # Output: Grade D
print(grade_calculator.calculate_grade(30))  # Output: Fail
```

```
de.py"
Grade B
Grade C
Grade A
Grade D
Fail
```

**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().

get_input() :

```
week 13 > square.py > get_input
#   write a refactor and Modular code using functions like get_input(), calculate_s
def get_input():
    """
    This function prompts the user to enter a number and returns it as an integer.

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

calculate_square() :

```
week 13 > square.py > calculate_square
    return int(input("Enter number: "))
def calculate_square(num):
    """This function calculates the square of a given number.

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

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

Display_result :

```
week 13 > square.py > calculate_square
def display_result(square):
    """This function displays the calculated square.

    Args:
        square (int): The calculated square.
    """
    print("Square:", square)
```

**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.

week 13 > salary.py > EmployeeSalaryCalculator > calculate_net_salary

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

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

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

    def calculate_tax(self):
        """
        Calculates the tax based on a fixed tax rate.

        Returns:
            float: The amount of tax to be deducted.
        """
        tax_rate = 0.2
        return self.salary * tax_rate
```

```python
    def calculate_net_salary(self):
        """
        Calculates the net salary after deducting ta

        Returns:
            float: The net salary of the employee.
        """
```

```
        tax = self.calculate_tax()
        net_salary = self.salary - tax
        return net_salary
# Example usage
if __name__ == "__main__":
    salary = 50000
    calculator = EmployeeSalaryCalculator(salary)
    net_salary = calculator.calculate_net_salary()
    print(net_salary)
```

Output:

```
PS C:\Users\User\OneDrive\Desktop\AIAC>
ary.py"
40000.0
```

**Task 6 (Optimizing Search Logic)**

- **Task: Refactor inefficient linear searches using appropriate data structures.**
- Focus Areas:
  - Time complexity
  - 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:**

- Use of sets or dictionaries with complexity justification

```
week 13 > username.py > UserAccessControl
10  # Refactored Code:Expected Outcome: Use of sets or dictionaries with complexity justification
11  class UserAccessControl:
12      """A class to manage user access control using a set for efficient lookups."""
13
14      def __init__(self, users):
15          """
16          Initializes the UserAccessControl with a list of users.
17
18          Args:
19              users (list): A list of usernames that have access.
20          """
21          self.users = set(users)  # Using a set for O(1) average time complexity for lookups
22
23      def check_access(self, username):
24          """
25          Checks if the given username has access.
26
27          Args:
28              username (str): The username to check for access.
29          Returns:
30              bool: True if access is granted, False otherwise.
31          """
32          return username in self.users  # O(1) average time complexity for lookups
```

```
33  # Example usage
34  if __name__ == "__main__":
35      users = ["admin", "guest", "editor", "viewer"]
36      access_control = UserAccessControl(users)
37      name = input("Enter username: ")
38      if access_control.check_access(name):
39          print("Access Granted")
40      else:
41          print("Access Denied")
```

Output:

```
PS C:\Users\User\OneDrive\Desktop\AIAC>
rname.py"
Enter username: admin
Access Granted
PS C:\Users\User\OneDrive\Desktop\AIAC>
rname.py"
Enter username: srijith
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:
    - add_book(title, author, isbn)
    - remove_book(isbn)
    - 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.")
```

week 13 > library.py > ...

```python
class Library:
    """A class to manage a library system with functionalities to add, search, and remove books."""

    def __init__(self):
        """Initializes the Library with an empty database."""
        self.library_db = {}

    def add_book(self, title, author, isbn):
        """
        Adds 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 or already exists.
        """
        if isbn not in self.library_db:
            self.library_db[isbn] = {"title": title, "author": author}
            return "Book added successfully."
        else:
            return "Book already exists."
```

week 13 > library.py > ...

```python
class Library:

    def search_book(self, isbn):
        """
        Searches for a book in the library database by its ISBN.

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

        Returns:
            str: A message indicating whether the book was found or not.
        """
        if isbn in self.library_db:
            return f"Book Found: {self.library_db[isbn]}"
        else:
            return "Book not found."
```

```
week 13 > library.py > ...
```

```python
class Library:
    def remove_book(self, isbn):
        """
        Removes a book from the library database by its ISBN.

        Args:
            isbn (str): The ISBN number of the book to remove.
        Returns:
            str: A message indicating whether the book was removed or not found.
        """
        if isbn in self.library_db:
            del self.library_db[isbn]
            return "Book removed successfully."
        else:
            return "Book not found."
```

```python
# Example usage
if __name__ == "__main__":
    library = Library()
    print(library.add_book("Python Basics", "John Doe", "101"))
    print(library.add_book("AI Fundamentals", "Jane Smith", "102"))
    print(library.search_book("101"))
    print(library.remove_book("101"))
    print(library.search_book("101"))
```

Output:

```
rary.py"
Book added successfully.
Book added successfully.
Book Found: {'title': 'Python Basics', 'author': 'John Doe
Book removed successfully.
```

```
Pary.py
PS C:\Users\User\OneDrive\Desktop\AIAC> cd "week 13"
PS C:\Users\User\OneDrive\Desktop\AIAC\week 13> python -m pydoc library
Help on module library:

NAME
    library

DESCRIPTION
    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}
```

```
Book not found.
PS C:\Users\User\OneDrive\Desktop\AIAC> cd "week 13"
PS C:\Users\User\OneDrive\Desktop\AIAC\week 13> python -m pydoc -w library
wrote library.html
PS C:\Users\User\OneDrive\Desktop\AIAC\week 13> python -m pydoc -p 1234
Server ready at http://localhost:1234/
Server commands: [b]rowser, [q]uit
server> b
```

# library

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.")
```

**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
```

---

#Refactored Code :

week 13 > fibonacci.py > ...

```
15  #write refactored code for above fibonacci code with proper documentation and
16  def fibonacci(n):
17      """
18      Generate Fibonacci sequence up to the nth term.
19
20      Args:
21          n (int): The number of terms in the Fibonacci sequence to generate.
22
23      Returns:
24          list: A list containing the Fibonacci sequence up to the nth term.
25      """
```

```
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]

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

    return sequence
# Test cases
if __name__ == "__main__":
    print(fibonacci(10))  # Expected output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
    print(fibonacci(0))   # Expected output: []
    print(fibonacci(1))   # Expected output: [0]
    print(fibonacci(2))   # Expected output: [0, 1]
    print(fibonacci(5))   # Expected output: [0, 1, 1, 2, 3]
    print(fibonacci(-1))  # Expected output: []
```

Output :

```
onacci.py"
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
[]
[0]
[0, 1]
[0, 1, 1, 2, 3]
[]
PS C:\Users\User\OneDrive\Desktop\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")
```

week 13 > prime.py > ...

```
    """
    # Refactored Code with Proper Documentation and Modular Structure
    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 <= 1:
            return False
        for i in range(2, int(num**0.5) + 1):
            if num % i == 0:
                return False
        return True
```

```python
def are_twin_primes(a, b):
    """Check if two numbers are twin primes.
    Args:
        a (int): The first number.
        b (int): The second number.
    Returns:
        bool: True if the numbers are twin primes, False otherwise.
    """
    return is_prime(a) and is_prime(b) and abs(a - b) == 2
# Test cases
if __name__ == "__main__":
    print(are_twin_primes(11, 13))  # Expected output: True
    print(are_twin_primes(17, 19))  # Expected output: True
    print(are_twin_primes(4, 6))    # Expected output: False
    print(are_twin_primes(29, 31))  # Expected output: True
    print(are_twin_primes(10, 12))  # Expected output: False
```

#Output :

```
PS C:\Users\User\OneDrive\Desktop\AIAC> &
me.py"
True
True
False
True
False
PS C:\Users\User\OneDrive\Desktop\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.

#Refactored Code :

```
week 13 > chinese.py > ...
38  # Refactored Code with Proper Documentation and Modular Structure
39  def get_zodiac(year):
40      """
41      Get the Chinese Zodiac sign for a given year.
42
43      Args:
44          year (int): The year for which to determine the Chinese Zodiac sign.
45      Returns:
46          str: The Chinese Zodiac sign corresponding to the given year.
47      """
48      zodiac_signs = [
49          "Monkey", "Rooster", "Dog", "Pig", "Rat", "Ox",
50          "Tiger", "Rabbit", "Dragon", "Snake", "Horse", "Goat"
51      ]
52      return zodiac_signs[year % 12]
53  if __name__ == "__main__":
54      try:
55          year_input = int(input("Enter a year: "))
56          zodiac_sign = get_zodiac(year_input)
57          print(f"The Chinese Zodiac sign for the year {year_input} is: {zodiac_sign}")
58      except ValueError:
59          print("Invalid input. Please enter a valid year.")
```

#Output :

```
PS C:\Users\User\OneDrive\Desktop\AIAC> & C:/Users/User/AppD
nese.py"
Enter a year: 2012
The Chinese Zodiac sign for the year 2012 is: 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
---
  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:
   - Accept an integer parameter.
   - Return True if the number is divisible by the sum of its digits.
   - 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).

#Refactored Code:

```
week 13 > harshad.py > ...
19  def is_harshad(num):
20      """
21      Check if a number is a Harshad number.
22
23      A Harshad number (or Niven number) is an integer that is divisible by the sum of its digits.
24
25      Args:
26          num (int): The number to check.
27      Returns:
28          bool: True if the number is a Harshad number, False otherwise.
29      """
30      if num <= 0:
31          return False
32
33      sum_digits = sum(int(digit) for digit in str(num))
34
35      if sum_digits == 0:
36          return False
37
38      return num % sum_digits == 0
```

Output:

```
39  # Test cases
40  if __name__ == "__main__":
41      test_numbers = [18, 19, 21, 0, -5]
42      for number in test_numbers:
43          result = is_harshad(number)
44          print(f"{number} is a Harshad number: {result}")
45
```

```
PS C:\Users\User\OneDrive\Desktop\AIAC> &
shad.py"
18 is a Harshad number: True
19 is a Harshad number: False
21 is a Harshad number: True
0 is a Harshad number: False
-5 is a Harshad number: False
PS C:\Users\User\OneDrive\Desktop\AIAC>
```

**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:
    - Accept a non-negative integer n.
    - 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).

week 13 > trailling.py > ...

```python
# Refactored Code with Proper Documentation and Modular Structure
def count_trailing_zeros(n):
    """
    Count the number of trailing zeros in the factorial of a given number.

    Args:
        n (int): The number for which to calculate the factorial and count trailing zeros.
    Returns:
        int: The number of trailing zeros in 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
```

#Output :

```python
# Test cases
if __name__ == "__main__":
    test_numbers = [0, 5, 10, 25, 100]
    for number in test_numbers:
        zeros = count_trailing_zeros(number)
        print(f"The number of trailing zeros in {number}! is: {zeros}")
```

```
PS C:\Users\User\OneDrive\Desktop\AIAC> & C:/U
illing.py"
The number of trailing zeros in 0! is: 0
The number of trailing zeros in 5! is: 1
The number of trailing zeros in 10! is: 2
The number of trailing zeros in 25! is: 6
The number of trailing zeros in 100! is: 24
```

## Test Cases Design

**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]

week 13 > Collatz.py > ...

```python
#write a python program Collatz Sequence
def collatz_sequence(n):
    """
    Generate the Collatz sequence for a given number n.

    The Collatz sequence is defined as:
    - If n is even, divide it by 2.
    - If n is odd, multiply by 3 and add 1.
    - Continue until reaching 1.

    Args:
        n (int): The starting number for the Collatz sequence.

    Returns:
        list: The Collatz sequence as a list of integers.
    """
    if n <= 0:
        raise ValueError("Input must be a positive integer.")

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

```

#Output :

```python

# Example usage
if __name__ == "__main__":
    try:
        number = int(input("Enter a positive integer: "))
        result = collatz_sequence(number)
        print(f"Collatz sequence for {number}: {result}")
    except ValueError as e:
        print(f"Error: {e}")
```

```
latz.py"
Enter a positive integer: 6
Collatz sequence for 6: [6, 3, 10, 5, 16, 8, 4, 2, 1]
PS C:\Users\User\OneDrive\Desktop\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.

week 13 > Lucas.py > ...

```python
#write a python program for Lucas Sequence use pytest
def lucas(n):
    """
    Generate the Lucas sequence up to the nth term.

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

    Returns:
        list: A list containing the Lucas sequence up to the nth term.
    """
    if n <= 0:
        return []
    elif n == 1:
        return [2]
    elif n == 2:
        return [2, 1]

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

    return sequence
# Test cases using pytest
def test_lucas():
    assert lucas(0) == []
    assert lucas(1) == [2]
    assert lucas(2) == [2, 1]
    assert lucas(5) == [2, 1, 3, 4, 7]
    assert lucas(10) == [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

#Output :

```
PS C:\Users\User\OneDrive\Desktop\AIAC> cd "week 13"
PS C:\Users\User\OneDrive\Desktop\AIAC\week 13> python -m pytest Lucas.py
============================= test session starts =
platform win32 -- Python 3.14.2, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\User\OneDrive\Desktop\AIAC\week 13
collected 1 item

Lucas.py .
```

**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

week 13 > vowel.py > test_count_vowels_and_consonants

```python
#write a python program to Vowel & Consonant Counter and write test cases using pytest
def count_vowels_and_consonants(text):
    """
    Count the number of vowels and consonants in a given text.

    Args:
        text (str): The input string to analyze.
    Returns:
        tuple: A tuple containing the count of vowels and consonants (vowels_count, consonants_count).
    """
    vowels = "aeiouAEIOU"
    vowels_count = 0
    consonants_count = 0

    for char in text:
        if char.isalpha():  # Check if the character is an alphabet
            if char in vowels:
                vowels_count += 1
            else:
                consonants_count += 1

    return vowels_count, consonants_count
```

#Output :

```
PS C:\Users\User\OneDrive\Desktop\AIAC\week 13> & C:/Users/User/AppData/Local/Python/python
k 13/vowel.py"
PS C:\Users\User\OneDrive\Desktop\AIAC\week 13> python -m pytest vowel.py
============================= test session starts =====
platform win32 -- Python 3.14.2, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\User\OneDrive\Desktop\AIAC\week 13
collected 1 item

vowel.py .

============================== 1 passed in 0.12s ======
```