ASSIGNMENT:13

HALL TICKET:2303A51598

BATCH:29

Task Description #1 (Refactoring Removing Code Duplication)

PROMPT:

Use AI Use AI to refactor a given Python script that contains multiple repeated code blocks.

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))
```

```python
# Refactored Code:
def calculate_rectangle_area(length, width):
    """
    Calculate the area of a rectangle.
    Args:
        length (int or float): The length of the rectangle
        width (int or float): The width of the rectangle
    Returns:
        int or float: The area of the rectangle
    """
    return length * width
def calculate_rectangle_perimeter(length, width):
    """
    Calculate the perimeter of a rectangle.
    Args:
        length (int or float): The length of the rectangle
        width (int or float): The width of the rectangle
    Returns:
        int or float: The perimeter of the rectangle
    """
    return 2 * (length + width)
# List of rectangles with their dimensions
rectangles = [(5, 10), (7, 12), (10, 15)]
```

```python
    return 2 * (length + width)
# List of rectangles with their dimensions
rectangles = [(5, 10), (7, 12), (10, 15)]
for length, width in rectangles:
    area = calculate_rectangle_area(length, width)
    perimeter = calculate_rectangle_perimeter(length, width)
    print(f"Area of Rectangle: {area}")
    print(f"Perimeter of Rectangle: {perimeter}")
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.e
/lab1.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\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

The legacy code contained repeated calculations for rectangle area and perimeter, which violates the DRY (Don’t Repeat Yourself) principle. Refactoring into a

reusable function improves maintainability, readability, and scalability because any logic change needs to be updated only once.

and logical errors (wrong variables, incorrect output). It improves debugging skills and teaches how AI can assist in making code executable and correct. It also builds confidence in analyzing faulty programs.

Task Description #2 (Refactoring – Extracting Reusable Functions)

PROMPT:

- **Use AI 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

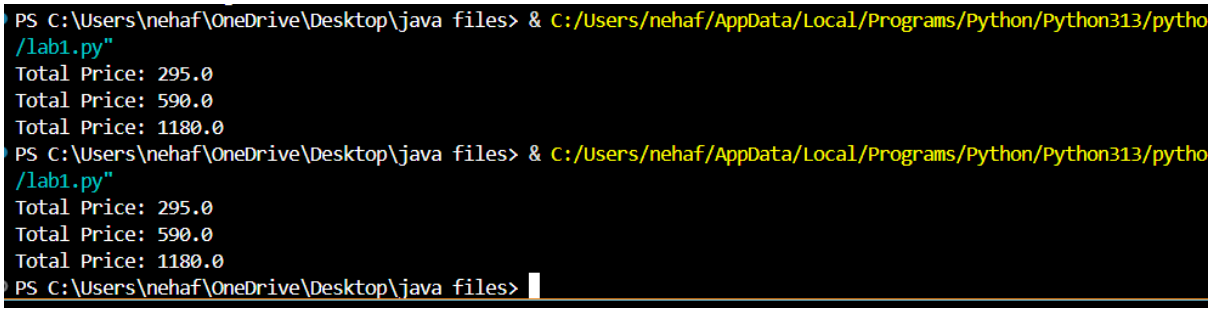
print("Total Price:", total)

```
#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)
# price = 1000
# tax = price * 0.18
# total = price + tax
# print("Total Price:", total)
# Refactored Code:
```

lab1.py > ...

```
1194  # Refactored Code:
1195  def calculate_total_price(price):
1196      """
1197      Calculate the total price including tax.
1198
1199      Args:
1200          price (int or float): The base price of the item
1201
1202      Returns:
1203          int or float: The total price including tax
1204      """
1205      tax = price * 0.18  # Calculate tax as 18% of the price
1206      total = price + tax  # Calculate total price by adding tax to the base price
1207      return total  # Return the total price
1208  # List of prices to calculate total price for
1209  prices = [250, 500, 1000]
1210  for price in prices:
1211      total_price = calculate_total_price(price)
1212      print(f"Total Price: {total_price}")
1213
1214
```

OUTPUT:



JUSTIFICATION:

This task The tax calculation logic was repeated for different price values. Creating a reusable function calculate_total(price) modularizes the logic, reduces redundancy, and enables reuse for multiple inputs without rewriting code.

Task Description #3: Refactoring Using Classes and Methods (Eliminating Redundant Conditional Logic)

PROMPT:

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")
```

```
class GradeCalculator:
    """
    A class to calculate grades based on student marks.
    """
    def calculate_grade(self, marks):
        """
        Calculate the grade for the given marks.

        Args:
            marks (int): The marks obtained by the student (0 to 100)
        Returns:
            str: The grade corresponding to the marks
        """
        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 >= 70:
            return "Grade C"
        elif marks >= 40:
            return "Grade D"
        elif marks >= 0:
            return "Fail"
        else:
            raise ValueError("Marks should be between 0 and 100.")
# Example usage:
grade_calculator = GradeCalculator()
print(grade_calculator.calculate_grade(85))  # Output: Grade B
print(grade_calculator.calculate_grade(72))  # Output: Grade C
```

JUSTIFICATION:

This task Repeated conditional grading logic reduces maintainability and increases error risk. Implementing a class-based design encapsulates grading behavior, promotes reusability, and follows object-oriented principles such as abstraction and modularity.

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

PROMPT:

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
```

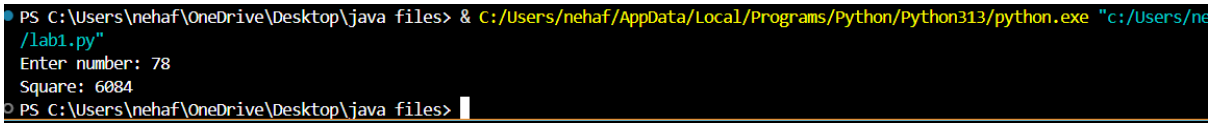
print("Square:", square)

```
# 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().
def get_input():
    """
    Prompt the user to enter a number and return it as an integer.

    Returns:
        int: The number entered by the user
    """
    return int(input("Enter number: "))
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_output(square):
    """
    Display the squared value.

    Args:
        square (int): The squared value to be displayed

    Returns:
        None
    """
    print("Square:", square)
def main():
    """Main function to orchestrate the input, processing, and output.

    Returns:
```

```python
def main():
    """Main function to orchestrate the input, processing, and output.

    Returns:
        None
    """
    num = get_input()  # Get input from the user
    square = calculate_square(num)  # Process the input to calculate the square
    display_output(square)  # Display the output
if __name__ == "__main__":
    main()  # Run the main function
```

OUTPUT:



JUSTIFICATION:

This Separating input, processing, and output into individual functions enhances code clarity, supports testing of individual components, and follows functional decomposition for better maintainability.

Task Description - 5 (Refactoring Procedural Code into OOP Design)

PROMPT:

Use AI to refactor procedural code into a class-based design.
Focus Areas:
o Object-Oriented principles
o Encapsulation

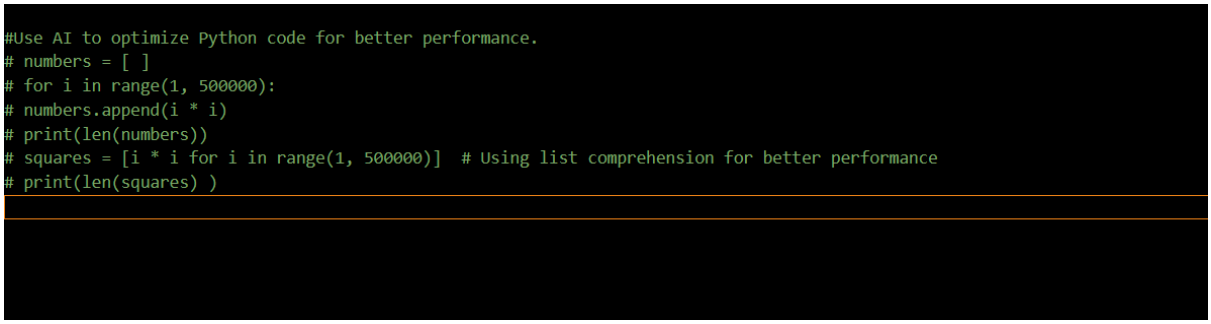
Legacy Code:

salary = 50000

tax = salary * 0.2

net = salary - tax

print(net)



JUSTIFICATION:

This task Encapsulating salary and tax computation within a class improves data hiding and organization. The OOP approach allows reuse, extension, and better representation of real-world entities like employees.

Task 6 (Optimizing Search Logic)

PROMPT:

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")

```
# 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")
# Refactored Code:

users = {"admin", "guest", "editor", "viewer"}  # Using a set for O(1) average time complexity
name = input("Enter username: ")
if name in users:  # Checking membership in a set is more efficient than a list
    print("Access Granted")
else:
    print("Access Denied")
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Pyth
/lab1.py"
Enter username: NEHA
Access Denied
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

Linear search has O(n) complexity. Using a set or dictionary enables average O(1) lookup time, improving performance and demonstrating appropriate data structure selection.

Task 7 – Refactoring the Library Management System

PROMPT:

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.")
```

```python
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 (unique identifier)

    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 using its ISBN.

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

```python
    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."
ef search_book(isbn):
    """
    Search for a book in the library database using its ISBN.

    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."
```

JUSTIFICATION:

The original script lacked modularity and repeated logic patterns. Creating reusable functions within a module improves code organization, enables documentation generation, and supports scalable system design.

Task 8– Fibonacci Generator

PROMPT:

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
```

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

    Args:
        n (int): The number of Fibonacci numbers to generate

    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 for generate_fibonacci function
test_cases = [
    (0, [], "Fibonacci series with n=0 should return an empty list"),
```

```python
# Test cases for generate_fibonacci function
test_cases = [
    (0, [], "Fibonacci series with n=0 should return an empty list"),
    (1, [0], "Fibonacci series with n=1 should return [0]"),
    (2, [0, 1], "Fibonacci series with n=2 should return [0, 1]"),
    (5, [0, 1, 1, 2, 3], "Fibonacci series with n=5 should return the first 5 Fibonacci numbers"),
    (10, [0, 1, 1, 2, 3, 5, 8, 13, 21, 34], "Fibonacci series with n=10 should return the first 10 Fibonacci numbers"),
]
# Run test cases for generate_fibonacci function
print("Running test cases for generate_fibonacci function:")
print("-" * 50)
all_passed = True
for n, expected, description in test_cases:
    result = generate_fibonacci(n)
    status = "PASS" if result == expected else "FAIL"
    if result != expected:
        all_passed = False
    print(f"Test: {description}")
    print(f"  Input: {n}, Expected: {expected}, Got: {result} - {status}")
print("-" * 50)
if all_passed:
    print("All test cases for generate_fibonacci PASSED!")
else:
    print("Some test cases for generate_fibonacci FAILED!")
```

Output:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313
/lab1.py"
Running test cases for generate_fibonacci function:
--------------------------------------------------
Test: Fibonacci series with n=0 should return an empty list
  Input: 0, Expected: [], Got: [] - PASS
Test: Fibonacci series with n=1 should return [0]
  Input: 1, Expected: [0], Got: [0] - PASS
Test: Fibonacci series with n=2 should return [0, 1]
  Input: 2, Expected: [0, 1], Got: [0, 1] - PASS
Test: Fibonacci series with n=5 should return the first 5 Fibonacci nu
```

File Explorer

JUSTIFICATION:

The initial implementation used global variables and procedural flow. Refactoring into a function enhances modularity, reusability, and testability while improving readability.

Task 9 – Twin Primes Checker

PROMPT:

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.**

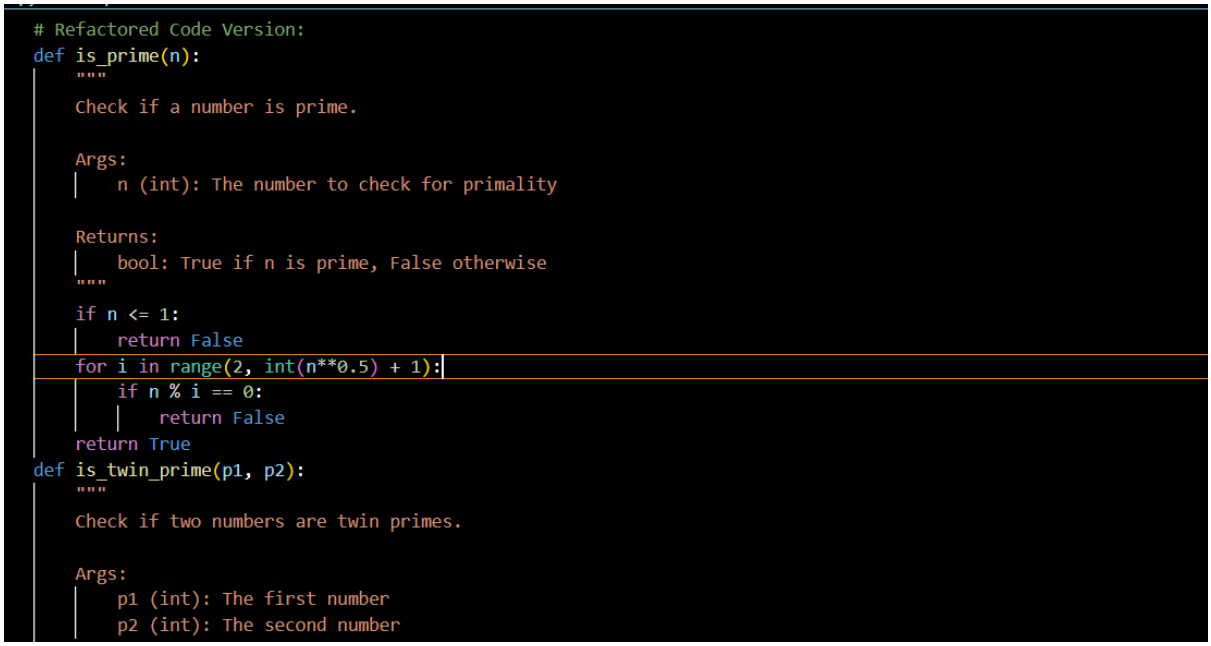
Task for Students:

- **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")
```



```
# Refactored Code Version:
def is_prime(n):
    """
    Check if a number is prime.

    Args:
        n (int): The number to check for primality

    Returns:
        bool: True if n is prime, False otherwise
    """
    if n <= 1:
        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): The first number
        p2 (int): The second number
```

```
    Returns:
        bool: True if p1 and p2 are twin primes, False otherwise
    """
    if is_prime(p1) and is_prime(p2) and abs(p1 - p2) == 2:
        return True
    return False
def generate_twin_primes(start, end):
    """
    Generate a list of twin primes within a given range.

    Args:
        start (int): The starting number of the range (inclusive)
        end (int): The ending number of the range (inclusive)

    Returns:
        list: A list of tuples representing twin primes within 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:
twin_prime_list = generate_twin_primes(1, 100)
print("Twin primes between 1 and 100:", twin_prime_list)
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/neha
/lab1.py"
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\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

Inefficient prime checking and hardcoded values reduced flexibility. Creating reusable functions for prime and twin-prime checks improves efficiency, generalization, and maintainability.

Task 10 – Refactoring the Chinese Zodiac Program

PROMPT:

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.

```python
# Refactored Code Version:
def get_zodiac(year):
    """
    Get the Chinese Zodiac sign for a given year.

    Args:
        year (int): The year for which to determine the Zodiac sign
    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]
def main():
    """Main function to handle user input and display the Zodiac sign."""
    try:
        year = int(input("Enter a year: "))
        zodiac_sign = get_zodiac(year)
        print(f"The Chinese Zodiac sign for the year {year} is: {zodiac_sign}")
    except ValueError:
        print("Invalid input. Please enter a valid year as an integer.")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/
/lab1.py"
Enter a year: 2007
The Chinese Zodiac sign for the year 2007 is: Pig
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

A long if–elif chain reduces readability and increases maintenance complexity. Using a list or dictionary simplifies mapping logic, improves clarity, and supports easier updates.

Task 11 – Refactoring the Harshad (Niven) Number Checker

PROMPT:

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:
    - 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).

```python
def is_harshad(number):
    """
    Check if a number is a Harshad (Niven) number.

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

    Args:
        number (int): The number to check for being a Harshad number
    Returns:
        bool: True if the number is a Harshad number, False otherwise
    """
    if number < 0:
        return False  # Harshad numbers are typically defined for non-negative integers
    sum_digits = sum(int(digit) for digit in str(number))  # Calculate the sum of digits
    if sum_digits == 0:
        return False  # Avoid division by zero
    return number % sum_digits == 0  # Check if the number is divisible by the sum of its digits
def main():
    """Main function to handle user input and display whether the number is a Harshad number."""
    try:
        num = int(input("Enter a number: "))
        if is_harshad(num):
            print(f"{num} is a Harshad (Niven) number.")
        else:
            print(f"{num} is not a Harshad (Niven) number.")
    except ValueError:
        print("Invalid input. Please enter a valid integer.")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.e
/lab1.py"
Enter a number: 67
67 is not a Harshad (Niven) number.
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

Mixing input handling with logic and using print-based output limits reuse. Refactoring into a boolean-returning

function separates concerns, improves modularity, and supports testing.

Task 12 – Refactoring the Factorial Trailing Zeros Program

PROMPT:

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).

```python
def count_trailing_zeros(n):
    """
    Count the number of trailing zeros in n! (factorial of n).

    Trailing zeros in a factorial are created by the factors of 10, which are made from pairs of 2 and 5.
    Since there are usually more factors of 2 than 5 in a factorial, we can count the number of times 5 is a factor in the numbers from 1 to n.

    Args:
        n (int): A non-negative integer for which to count trailing zeros in its
        factorial.
    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
def main():
    """Main function to handle user input and display the number of trailing zeros in n!."""
    try:
        n = int(input("Enter a non-negative integer: "))
```

```python
def main():
    """Main function to handle user input and display the number of trailing zeros in n!."""
    try:
        n = int(input("Enter a non-negative integer: "))
        if n < 0:
            print("Please enter a non-negative integer.")
            return
        zeros = count_trailing_zeros(n)
        print(f"The number of trailing zeros in {n}! is: {zeros}")
    except ValueError:
        print("Invalid input. Please enter a valid non-negative integer.")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
& C:/Users/nehaf/AppData/Local/Programs/Python/Python313/
/lab1.py"
Enter a non-negative integer: 8
The number of trailing zeros in 8! is: 1
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

Computing full factorial is inefficient for large inputs. Using the mathematical approach of counting multiples of 5 reduces time complexity significantly and improves performance.

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]

```python
def collatz_sequence(n):
    """
    Generate the Collatz sequence for a given integer n until reaching 1.

    The Collatz sequence is defined as follows:
    - If n is even, the next number is n / 2.
    - If n is odd, the next number is 3n + 1.
    The sequence continues until it reaches 1.

    Args:
        n (int): The starting integer for the Collatz sequence. Must be a positive integer.
    Returns:
        list: A list containing the Collatz sequence starting from n and ending at 1.
    Raises:
        ValueError: If n is not a positive integer.
    """
    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)

    return sequence
# Example usage:
```

```python
# Example usage:
if __name__ == "__main__":
    print(collatz_sequence(6))  # Output: [6, 3, 10, 5, 16, 8, 4, 2, 1]
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/De
/lab1.py"
[6, 3, 10, 5, 16, 8, 4, 2, 1]
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

Designing normal, edge, negative, and large test cases ensures functional correctness, robustness, and reliability of the Collatz sequence implementation across input scenarios.

Task 14 (Lucas Number Sequence – Test Case Design)

PROMPT:

- **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**

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

    The Lucas sequence is defined as follows:
    - L(0) = 2
    - L(1) = 1
    - L(n) = L(n-1) + L(n-2) for n > 1

    Args:
    |   n (int): The number of terms in the Lucas sequence to generate. Must be a non-negative integer.
    Returns:
    |   list: A list containing the first n terms of the Lucas sequence.
    Raises:
    |   ValueError: If n is a negative integer.
    """
    if n < 0:
    |   raise ValueError("Input must be a non-negative integer.")

    if n == 0:
    |   return []
```

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

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

    return sequence
Example usage:
__name__ == "__main__":
    print(lucas_sequence(5))  # Output: [2, 1, 3, 4, 7]
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Pytho
/lab1.py"
[2, 1, 3, 4, 7]
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

Test case coverage validates sequence correctness, boundary handling, and error management, ensuring the function behaves reliably for different inputs.

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

PROMPT:

- **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.**

```python
def count_vowels_consonants(s):
    """
    Count the number of vowels and consonants in a given string.

    Vowels are defined as 'a', 'e', 'i', 'o', 'u' (both uppercase and lowercase).
    Consonants are defined as any alphabetic character that is not a vowel.

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

    for char in s:
        if char.isalpha():  # Check if the character is an alphabet
            if char in vowels:
                vowel_count += 1
            else:
```

```
                consonant_count += 1

        return vowel_count, consonant_count
# Example usage:
if __name__ == "__main__":
    print(count_vowels_consonants("hello"))  # Output: (2, 3)
    print(count_vowels_consonants(""))  # Output: (0, 0)
    print(count_vowels_consonants("aeiou"))  # Output: (5, 0)

```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Loc
/lab1.py"
(2, 3)
(0, 0)
(5, 0)
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

Multiple test cases including empty strings, vowel-only strings, and large inputs verify accuracy, edge-case handling, and performance of the counting logic.