

Assignment-13

2303A51272

A.Srivani

Batch-23

Task Description #1 (Refactoring – Removing Code Duplication)

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

- Instructions:

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

- o Ensure the refactored code maintains the same output.

- o Add docstrings to all functions.

- Sample Legacy Code:

Legacy script with repeated logic

```
print("Area of Rectangle:", 5 * 10)
```

```
print("Perimeter of Rectangle:", 2 * (5 + 10))
```

```
print("Area of Rectangle:", 7 * 12)
```

```
print("Perimeter of Rectangle:", 2 * (7 + 12))
```

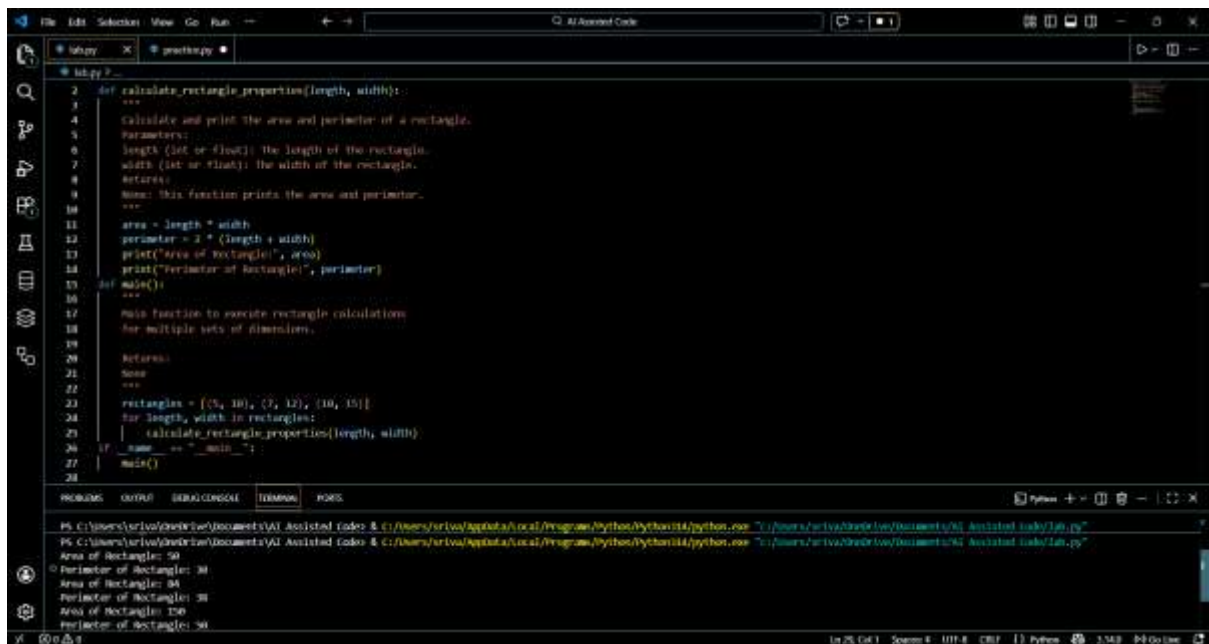
```
print("Area of Rectangle:", 10 * 15)
```

```
print("Perimeter of Rectangle:", 2 * (10 + 15))
```

- Expected Output:

- o Refactored code with a reusable function and no duplication.

o Well documented code



```
1 def calculate_rectangle_properties(length, width):
2     """
3     Calculate and print the area and perimeter of a rectangle.
4     Parameters:
5     length (int or float): the length of the rectangle.
6     width (int or float): the width of the rectangle.
7     Returns:
8     None: This function prints the area and perimeter.
9     """
10
11     area = length * width
12     perimeter = 2 * (length + width)
13     print("Area of Rectangle:", area)
14     print("Perimeter of Rectangle:", perimeter)
15
16 def main():
17     """
18     Main function to execute rectangle calculations
19     for multiple sets of dimensions.
20     Returns:
21     None
22     """
23
24     rectangles = [(5, 10), (2, 12), (10, 15)]
25     for length, width in rectangles:
26         calculate_rectangle_properties(length, width)
27
28 if __name__ == "__main__":
29     main()
```

The screenshot shows a Python IDE with a file named 'lab.py'. The code defines a function 'calculate_rectangle_properties' that takes 'length' and 'width' as arguments, calculates the area and perimeter, and prints them. It also includes a 'main' function that iterates over a list of rectangles and calls the calculation function. The terminal output shows the results for three rectangles: (5, 10), (2, 12), and (10, 15).

Task Description #2 (Refactoring – Extracting Reusable Functions)

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

- Instructions:

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

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

- Sample Legacy Code:

Week7

-

Monda

y

Legacy script with inline repeated logic

```
price = 250
```

```
tax = price * 0.18
```

```
total = price + tax
```

```
print("Total Price:", total)
```

```
price = 500
```

```
tax = price * 0.18
```

```
total = price + tax
```

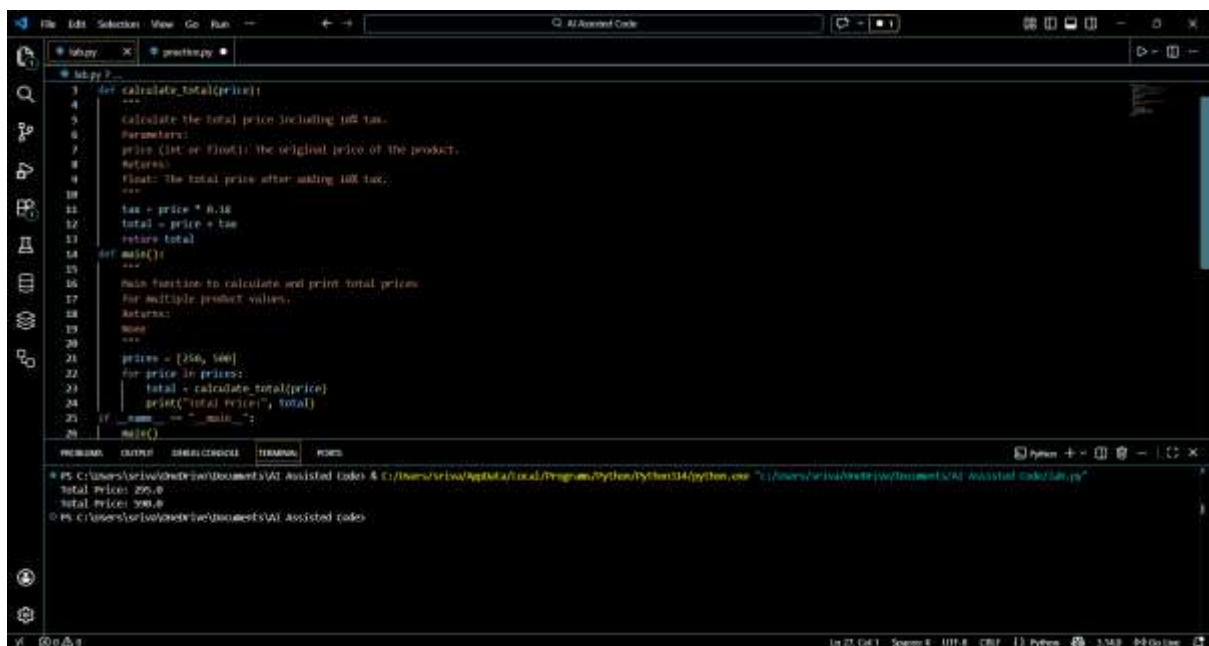
```
print("Total Price:", total)
```

• Expected Output:

o Code with a function `calculate_total(price)` that can be reused

for multiple price inputs.

o Well documented code



```
1 def calculate_total(price):
2     """
3     Calculate the total price including 10% tax.
4     Parameters:
5     price (int or float): the original price of the product.
6     Returns:
7     float: the total price after adding 10% tax.
8     """
9     tax = price * 0.18
10    total = price + tax
11    return total
12
13 def main():
14     """
15     Main function to calculate and print total prices
16     for multiple product values.
17     Returns:
18     None
19     """
20    prices = [250, 500]
21    for price in prices:
22        total = calculate_total(price)
23        print("Total Price:", total)
24
25 if __name__ == "__main__":
26    main()
```

Terminal Output:

```
total price: 295.0
total price: 590.0
```

Task Description #3: Refactoring Using Classes and Methods (Eliminating

Redundant Conditional Logic)

Refactor a Python script that contains repeated if-elif-else grading logic by implementing a structured, object-oriented solution using a class and a method.

Problem Statement

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

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

Mandatory Implementation Requirements

1. Class Name: GradeCalculator
2. Method Name: calculate_grade(self, marks)
3. The method must:
 - o Accept marks as a parameter.
 - o Return the corresponding grade as a string.
 - o The grading logic must strictly follow the conditions below:
 - Marks ≥ 90 and $\leq 100 \rightarrow$ "Grade A"
 - Marks $\geq 80 \rightarrow$ "Grade B"
 - Marks $\geq 70 \rightarrow$ "Grade C"
 - Marks $\geq 40 \rightarrow$ "Grade D"

- Marks $\geq 0 \rightarrow$ "Fail"

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

4. Include proper docstrings for:

- o The class

- o The method (with parameter and return descriptions)

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

- Given code:

```
marks = 85
```

```
if marks >= 90:
```

```
    print("Grade A")
```

```
elif marks >= 75:
```

```
    print("Grade B")
```

```
else:
```

```
    print("Grade C")
```

```
marks = 72
```

```
if marks >= 90:
```

```
    print("Grade A")
```

```
elif marks >= 75:
```

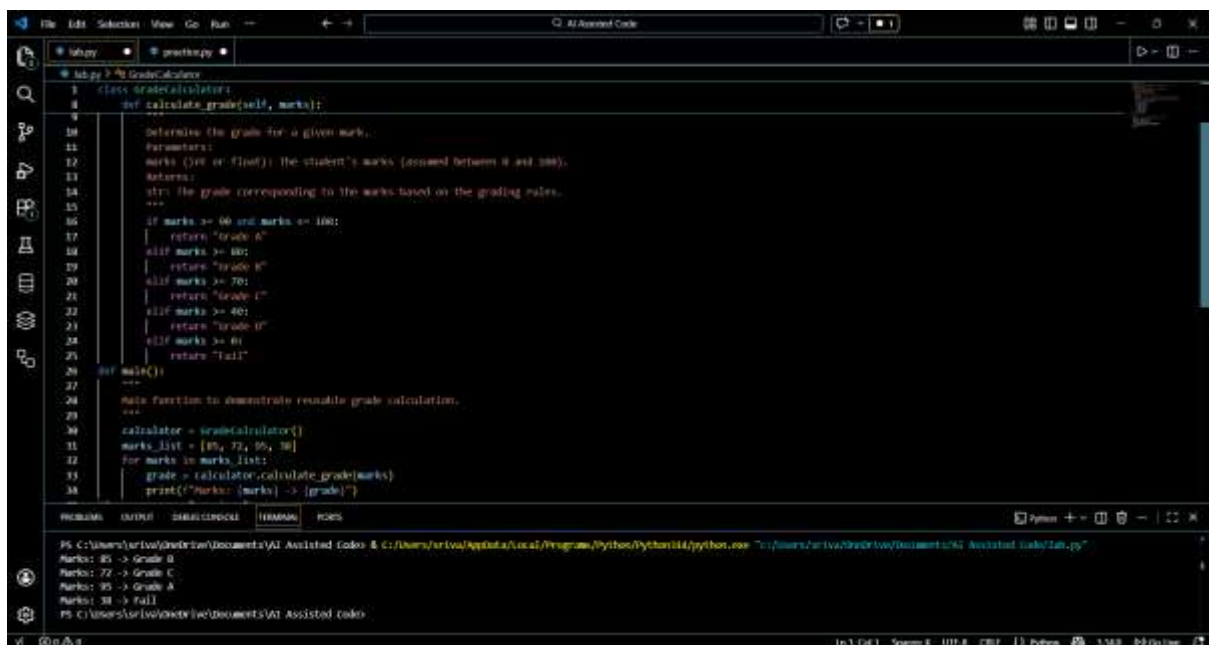
```
    print("Grade B")
```

```
else:
```

```
    print("Grade C")
```

Expected Output:

- Define a class named GradeCalculator.
- Implement a method `calculate_grade(self, marks)` inside the class.
- Create an object of the class.
- Call the method for different student marks.
- Print the returned grade values.



```
1 class GradeCalculator:
2     def calculate_grade(self, marks):
3         """
4         Determine the grade for a given mark.
5         Parameters:
6         marks (int or float): The student's marks (assumed between 0 and 100).
7         Returns:
8         str: The grade corresponding to the marks based on the grading rules.
9         """
10        if marks >= 90 and marks <= 100:
11            return "Grade A"
12        elif marks >= 80:
13            return "Grade B"
14        elif marks >= 70:
15            return "Grade C"
16        elif marks >= 60:
17            return "Grade D"
18        elif marks >= 50:
19            return "Grade E"
20        else:
21            return "Fail"
22
23    def main():
24        """
25        Main function to demonstrate reusable grade calculation.
26        """
27        calculator = GradeCalculator()
28        marks_list = [85, 72, 95, 38]
29        for marks in marks_list:
30            grade = calculator.calculate_grade(marks)
31            print(f"marks: {marks} -> {grade}")
32
33if __name__ == "__main__":
34    main()
```

OUTPUT

```
PS C:\Users\sriva\Documents> python .\AI Assisted Code\Task 4\Task 4.py
marks: 85 -> Grade B
marks: 72 -> Grade C
marks: 95 -> Grade A
marks: 38 -> 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:

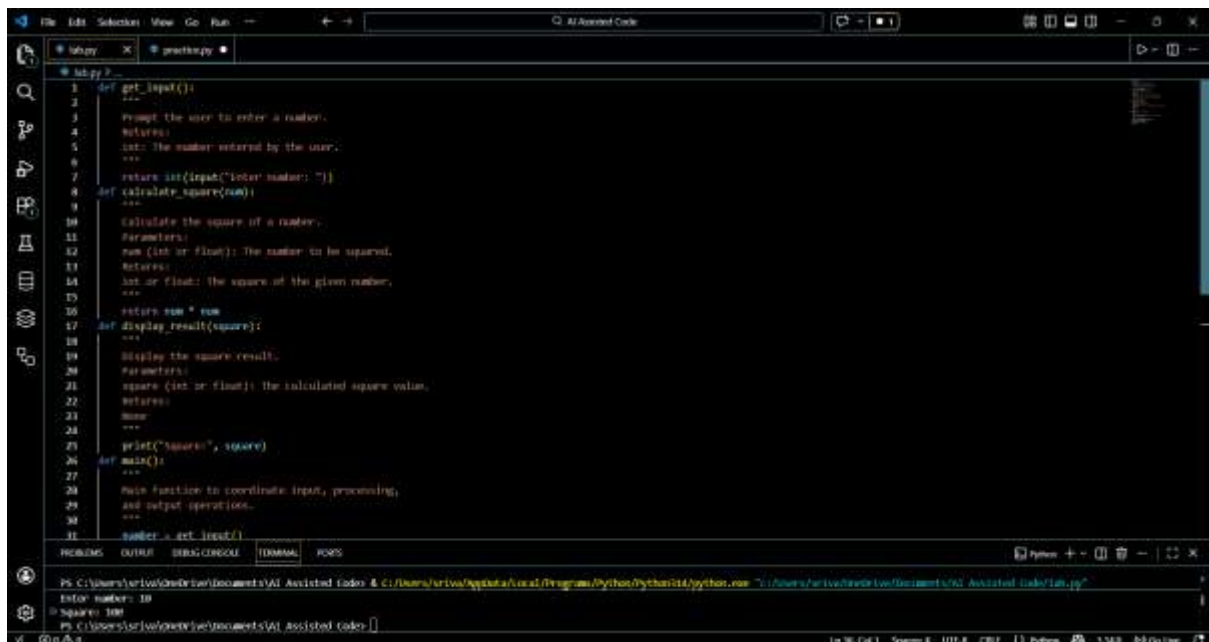
```
num = int(input("Enter number: "))
```

```
square = num * num
```

```
print("Square:", square)
```

- Expected Output:

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



```

1 def get_input():
2     """
3     Prompt the user to enter a number.
4     Returns:
5     int: The number entered by the user.
6     """
7     return int(input("Enter number: "))
8
9 def calculate_square(num):
10    """
11    Calculate the square of a number.
12    Parameters:
13    num (int or float): The number to be squared.
14    Returns:
15    int or float: The square of the given number.
16    """
17    return num * num
18
19 def display_result(square):
20    """
21    Display the square result.
22    Parameters:
23    square (int or float): The calculated square value.
24    Returns:
25    None
26    """
27    print("Square:", square)
28
29 def main():
30    """
31    Main function to coordinate input, processing,
32    and output operations.
33    """
34    number = get_input()
35    square = calculate_square(number)
36    display_result(square)
37
38 if __name__ == "__main__":
39     main()

```

The screenshot shows a Python IDE with the above code. The output console at the bottom shows the execution: "Enter number: 10" followed by "Square: 100".

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
```

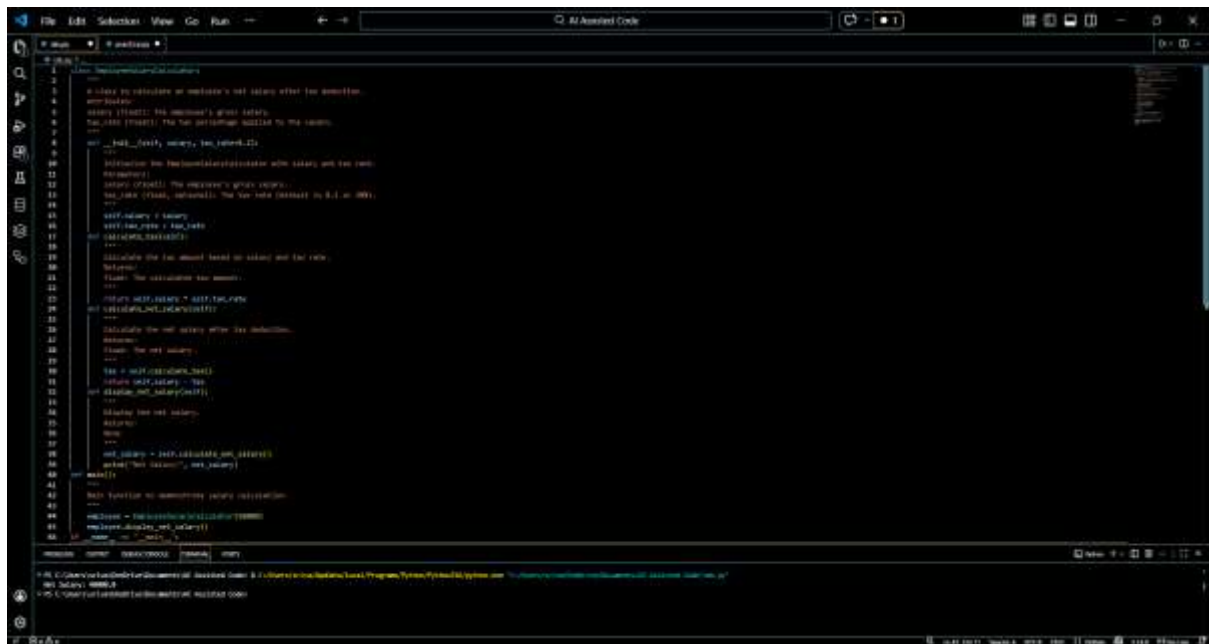
tax = salary * 0.2

net = salary - tax

print(net)

Expected Outcome:

o A class like EmployeeSalaryCalculator with methods and attributes.



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:
```

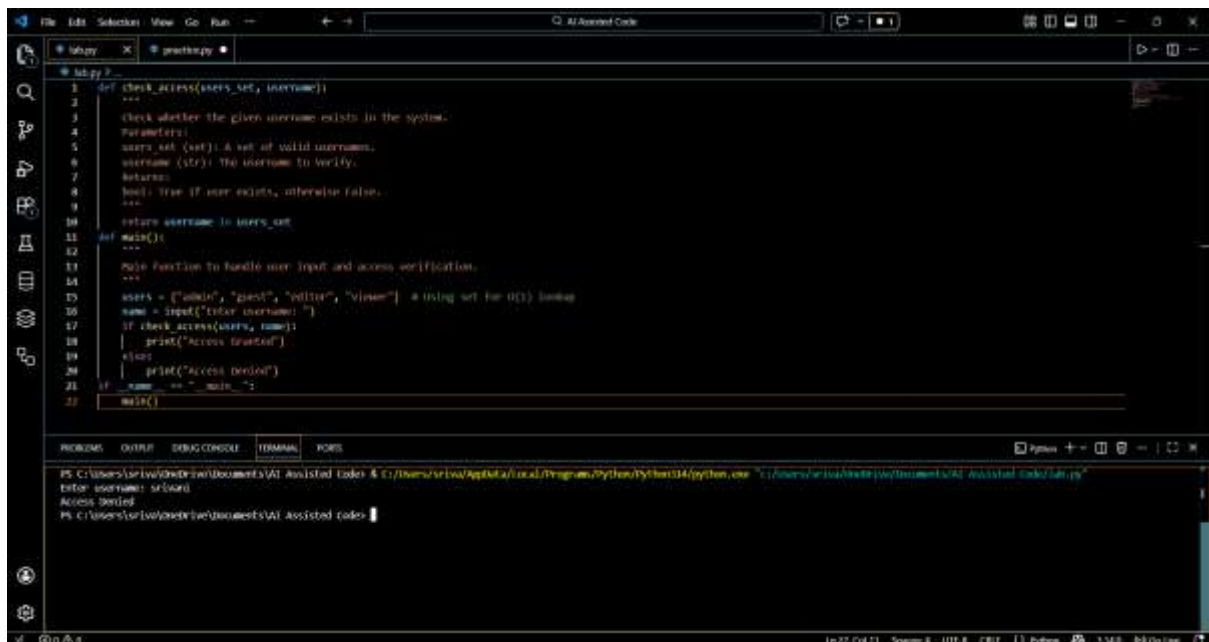
```
if u == name:
```

```
found = True
```

```
print("Access Granted" if found else "Access Denied")
```

Expected Outcome:

o Use of sets or dictionaries with complexity justification



```
1 def check_access(users_set, username):
2     """
3     Check whether the given username exists in the system.
4     Parameters:
5     users_set (set): A set of valid usernames.
6     username (str): The username to verify.
7     Returns:
8     bool: True if user exists, otherwise False.
9     """
10    return username in users_set
11
12 def main():
13    """
14    Main function to handle user input and access verification.
15    """
16    users = {"admin", "guest", "editor", "viewer"} # using set for O(1) lookup
17    name = input("Enter username: ")
18    if check_access(users, name):
19        print("Access Granted")
20    else:
21        print("Access Denied")
22
23 if __name__ == "__main__":
24    main()
```

REPLIEWS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\sriva\Documents\AI Assisted code & C:\Users\sriva\AppData\Local\Programs\Python\Python314\python.exe "C:\Users\sriva\Documents\AI Assisted code\lib.py"
Enter username: srivaa
Access Denied
PS C:\Users\sriva\Documents\AI Assisted code:
```

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.")

```

1
2 Library Management System Module
3 provides functions to manage a simple in-memory library database.
4
5 library_db = {}
6 def add_book(title: str, author: str, isbn: str) -> bool:
7     """
8     Adds a book to the library database.
9     Args:
10         title (str): The title of the book.
11         author (str): The author of the book.
12         isbn (str): The ISBN number of the book.
13     Returns:
14         bool: True if the book was added successfully, False if a book with the same ISBN already exists.
15     """
16
17     if isbn not in library_db:
18         library_db[isbn] = {"title": title, "author": author}
19         return True
20     return False
21 def remove_book(isbn: str) -> bool:
22     """
23     Removes a book from the library database by its ISBN.
24     Args:

```

[index](#)

library [c:\users\sriva\onedrive\documents\an assisted code\library.py](#)

Library Management System Module
Provides functions to manage a simple in-memory library database.

Functions

add_book(title: str, author: str, isbn: str) -> bool
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:
bool: True if the book was added successfully, False if a book with the same ISBN already exists.

remove_book(isbn: str) -> bool
Removes a book from the library database by its ISBN.
Args:
isbn (str): The ISBN number of the book to be removed.
Returns:
bool: True if the book was removed successfully, False if the book was not found in the database.

search_book(isbn: str) -> dict | None
Searches for a book in the library database by its ISBN.
Args: isbn (str): The ISBN number of the book to search for.
Returns: dict | None: A dictionary containing the book's title and author if found, or None if the book is not found in the database.

Data

library_db = {}

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

```

1 def generate_fibonacci(n):
2     """
3     generate fibonacci series up to n terms.
4     Parameters:
5     n (int): Number of terms to generate. Must be >= 0.
6     Returns:
7     List: A list containing the Fibonacci sequence up to n terms.
8     """
9     if n <= 0:
10         return []
11     if n == 1:
12         return [0]
13     fibonacci_series = [0, 1]
14     for _ in range(2, n):
15         fibonacci_series.append(
16             fibonacci_series[-1] + fibonacci_series[-2]
17         )
18     return fibonacci_series
19
20 def main():
21     """
22     Main function to demonstrate Fibonacci generation.
23     """
24     n = int(input("Enter limit: "))
25     result = generate_fibonacci(n)

```

Output: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17713 28657 46368 75025 121393 196418 317813 514229 832048 1346269 2178309 3524558 5702887 9223345 14930352 24157817 39088169 63245986 102334155 165580141 267914296 43286837 70148733 113490319 183631160 297121893 480752616 778042626 1258686005 2036558304 3295128009 5311609179 8620751772 13958383445 22585143121 36543529610 59128672629 95679208941 155398675920 251878811961 407278957881 662479031842 1069857991623 1732050908171 2814159665421 4546216676842 7360386544115 11906085549536 19260253454551 31167351904841 50424708347386 81591228303618 132070834606062 214177190481381 346145410288141 560343645488222 906525744676603 1466099136191466 2372046183584951 3838155497666644 6209256630682099 10047406927414151 16255663594149661 26308165546460842 42563869140610993 68872035136771635 111134110960461181 180002160944838326 291136276981609511 471170387939457837 762306554926416353 1233476934916016290 200577731705278991 3239054251968766301 5244731179120751101 8503706427077906102 13743081796849571403 22246783647938472504 36000485416516383605 58247267093455774609 94247026609394447814 152494293702864231823 246741320302258056837 399235516006152376460 645726836709070182497 1044962382711428664465 1690700018713581156954 273566239952470982651 442636241823829069146 716206481695197184841 115883773026341013 179917943268474185 28988719473816120 46980461107753839 75961138074636629 12285168151216738 19880741196622317 32360461369889965 52547887485123872 8492834892523777 1375088524470074096

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 `Al`.

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:

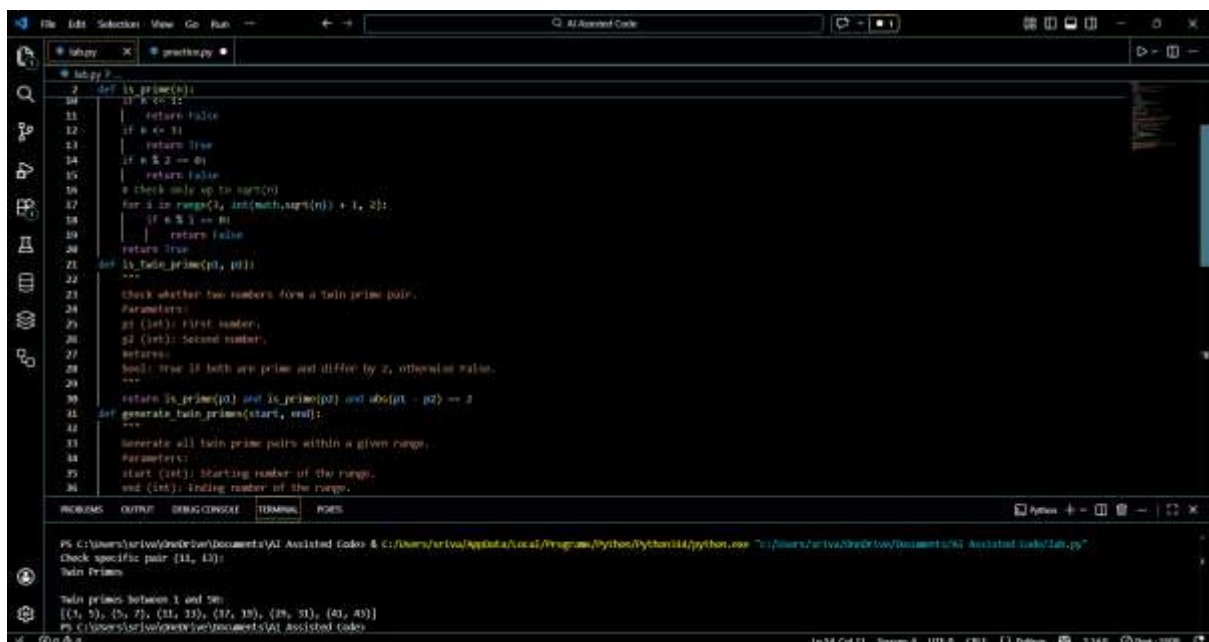
fb=1

if fa==0 and fb==0 and abs(a-b)==2:

print("Twin Primes")

else:

print("Not Twin Primes")



```
def is_prime(n):
    if n < 2:
        return False
    if n < 3:
        return True
    if n % 2 == 0:
        return False
    # Check only up to sqrt(n)
    for i in range(3, int(math.sqrt(n)) + 1, 2):
        if n % i == 0:
            return False
    return True

def is_twin_prime(p1, p2):
    """
    Check whether two numbers form a twin prime pair.
    Parameters:
    p1 (int): First number.
    p2 (int): Second number.
    Returns:
    bool: True if both are prime and differ by 2, otherwise False.
    """
    return is_prime(p1) and is_prime(p2) and abs(p1 - p2) == 2

def generate_twin_primes(start, end):
    """
    Generate all twin prime pairs within a given range.
    Parameters:
    start (int): Starting number of the range.
    end (int): Ending number of the range.
    """
    # Twin primes between 1 and 50:
    [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43)]
```

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:
```

```
elif year % 12 == 10:
```

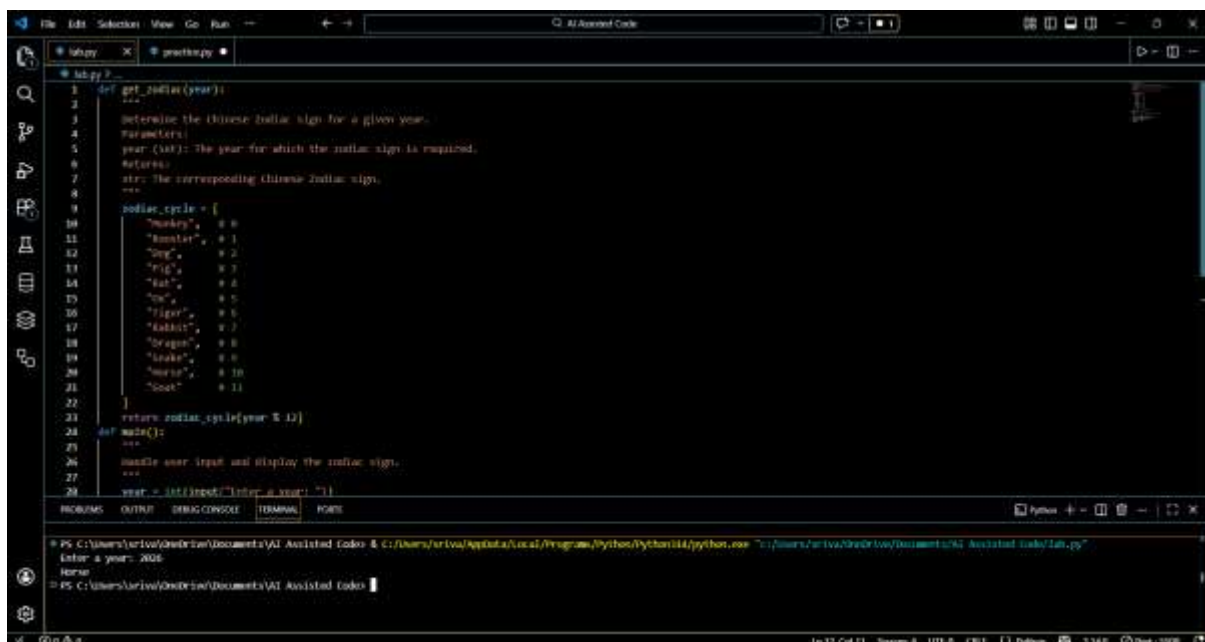
```
print("Horse")
```

```
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.



```
1 def get_zodiac(year):
2     """
3     Determine the Chinese zodiac sign for a given year.
4     Parameters:
5     year (int): The year for which the zodiac sign is required.
6     Returns:
7     str: The corresponding Chinese zodiac sign.
8     """
9     zodiac_cycle = {
10         "Monkey": 0,
11         "Rooster": 1,
12         "Dog": 2,
13         "Pig": 3,
14         "Rat": 4,
15         "Ox": 5,
16         "Tiger": 6,
17         "Rabbit": 7,
18         "Dragon": 8,
19         "Snake": 9,
20         "Horse": 10,
21         "Goat": 11
22     }
23     return zodiac_cycle[year % 12]
24
25 def main():
26     """
27     Handle user input and display the zodiac sign.
28     """
29     year = int(input("Enter a year: "))
30
31     print(get_zodiac(year))
32
33 if __name__ == "__main__":
34     main()
```

Python 3.14.0 (tags/v3.14.0:2024/08/24) [AMD64] (64-bit)

Enter a year: 2026

Goat

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:

- $18 \rightarrow 1 + 8 = 9 \rightarrow 18 \div 9 = 2$ \square (Harshad Number)
- $19 \rightarrow 1 + 9 = 10 \rightarrow 19 \div 10 \neq \text{integer}$ \square (Not Harshad)

Problem Statement

The current implementation:

- Mixes logic and input handling
- Uses redundant variables
- Does not use reusable functions properly
- Returns print statements instead of boolean values
- Lacks documentation

You must refactor the code to follow clean coding principles.

Harshad Number Checker (Unstructured Version)

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

```
temp = num
```

```
sum_digits = 0
```

```
while temp > 0:
```

```
    digit = temp % 10
```

```
    sum_digits = sum_digits + digit
```

```
    temp = temp // 10
```

```
if sum_digits != 0:
```

```
    if num % sum_digits == 0:
```

```
        print("True")
```

```
    else:
```

```
        print("False")
```

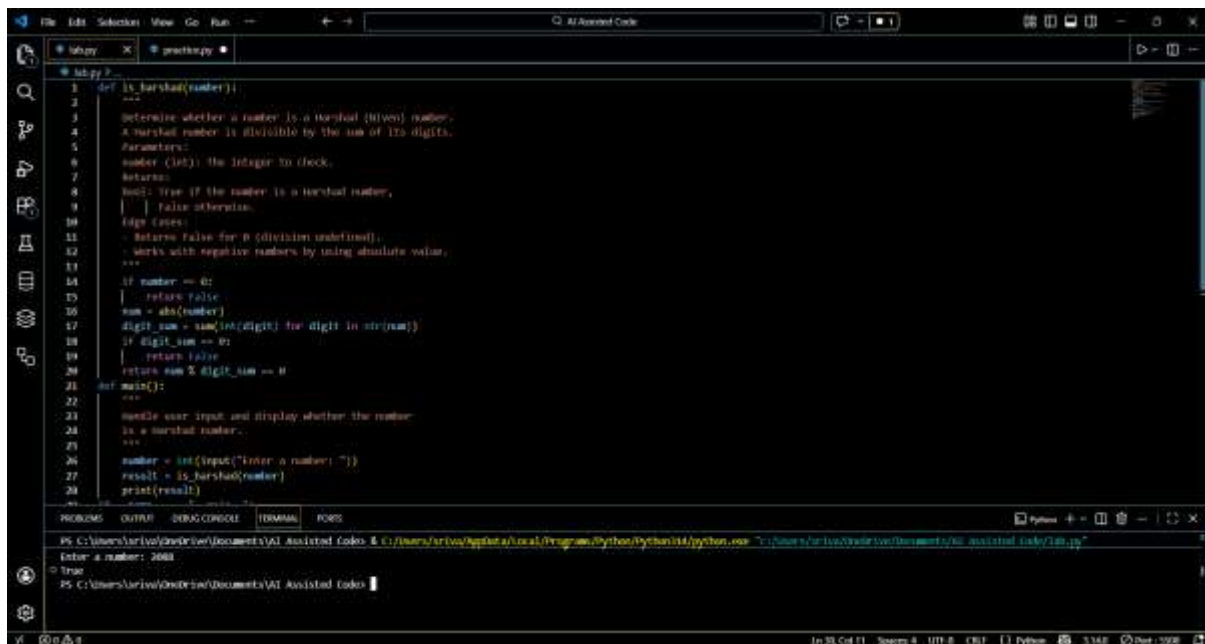
```
else:
```

```
    print("False")
```

You must:

1. Create a reusable function: `is_harshad(number)`
2. The function must:
 - o Accept an integer parameter.
 - o Return True if the number is divisible by the sum of its digits.
 - o Return False otherwise.
3. Separate user input from core logic.
4. Add proper docstrings.
5. Improve readability and maintainability.

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



```
1 def is_harshad(number):
2     """
3     Determine whether a number is a Harshad (Niven) number.
4     A Harshad number is divisible by the sum of its digits.
5     Parameters:
6     number (int): the integer to check.
7     Returns:
8     bool: True if the number is a Harshad number,
9           False otherwise.
10    Edge cases:
11    - Returns False for 0 (division undefined).
12    - Works with negative numbers by using absolute value.
13    """
14    if number == 0:
15        return False
16    num = abs(number)
17    digit_sum = sum(int(digit) for digit in str(num))
18    if digit_sum == 0:
19        return False
20    return num % digit_sum == 0
21
22 def main():
23     """Handle user input and display whether the number
24     is a Harshad number."""
25     number = int(input("Enter a number: "))
26     result = is_harshad(number)
27     print(result)
28
29 if __name__ == "__main__":
30     main()
```

The screenshot shows a Python IDE with a file named 'task12.py'. The code defines a function 'is_harshad' that checks if a number is a Harshad number. It handles edge cases like 0 and negative numbers. The 'main' function prompts the user for input and prints the result. The terminal shows the program running and outputting 'True' for the input '2880'.

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.

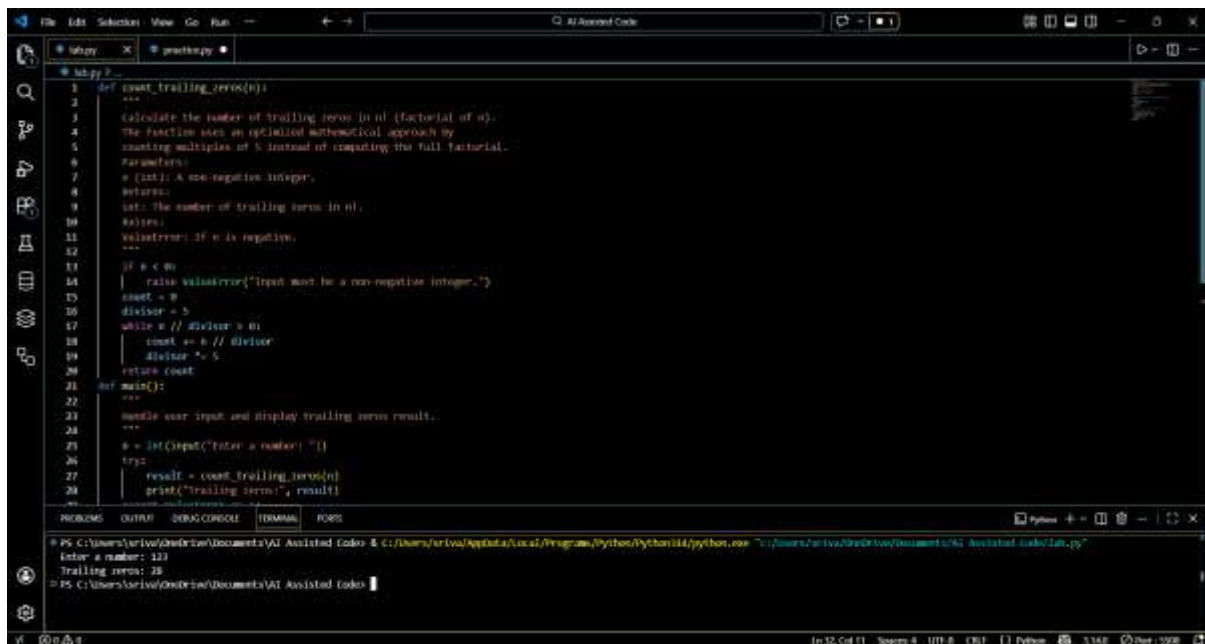
4. Use an optimized mathematical approach (count multiples of 5).

5. Add proper docstrings.

6. Separate user interaction from core logic.

7. Handle edge cases (e.g., negative numbers, zero).

Test Cases Design



```
1 def count_trailing_zeros(n):
2     """
3     Calculate the number of trailing zeros in n! (factorial of n).
4     The function uses an optimized mathematical approach by
5     counting multiples of 5 instead of computing the full factorial.
6     Parameters:
7     n (int): A non-negative integer.
8     Returns:
9     int: The number of trailing zeros in n!.
10    Raises:
11    ValueError: If n is negative.
12    """
13    if n < 0:
14        raise ValueError("Input must be a non-negative integer.")
15    count = 0
16    divisor = 5
17    while n // divisor > 0:
18        count += n // divisor
19        divisor *= 5
20    return count
21
22 def main():
23     """
24     Handle user input and display trailing zeros result.
25     """
26     n = int(input("Enter a number: "))
27     try:
28         result = count_trailing_zeros(n)
29         print("Trailing zeros:", result)
30     except ValueError as e:
31         print(e)
32
33 if __name__ == "__main__":
34     main()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\sriva\OneDrive\Documents\AI Assisted Code & c:\Users\sriva\AppData\Local\Programs\Python\Python314\python.exe "c:\Users\sriva\OneDrive\Documents\AI Assisted Code\Task13.py"

Enter a number: 22

Trailing zeros: 28

PS C:\Users\sriva\OneDrive\Documents\AI Assisted Code

Task 13 (Collatz Sequence Generator – Test Case Design)

- Function: Generate Collatz sequence until reaching 1.
- Test Cases to Design:
- Normal: $6 \rightarrow [6, 3, 10, 5, 16, 8, 4, 2, 1]$
- Edge: $1 \rightarrow [1]$
- Negative: -5
- Large: 27 (well-known long sequence)
- Requirement: Validate correctness with pytest.

Explanation:

We need to write a function that:

- Takes an integer n as input.
- Generates the Collatz sequence (also called the $3n+1$ sequence).
- The rules are:

o If n is even \rightarrow next = $n / 2$.

o If n is odd \rightarrow next = $3n + 1$.

• Repeat until we reach 1.

• Return the full sequence as a list.

Example

Input: 6

Steps:

• 6 (even $\rightarrow 6/2 = 3$)

• 3 (odd $\rightarrow 3*3+1 = 10$)

• 10 (even $\rightarrow 10/2 = 5$)

• 5 (odd $\rightarrow 3*5+1 = 16$)

• 16 (even $\rightarrow 16/2 = 8$)

• 8 (even $\rightarrow 8/2 = 4$)

• 4 (even $\rightarrow 4/2 = 2$)

• 2 (even $\rightarrow 2/2 = 1$)

Output:

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

The screenshot shows a code editor with two files: `lab.py` and `test_lab.py`. The `lab.py` file contains a function `collatz_sequence(n)` that generates the Lucas sequence. The function takes an integer `n` and returns a list of the first `n` terms of the sequence. The sequence starts with 2 and 1, and each subsequent term is the sum of the two preceding terms. The function includes a check for positive integers and a loop to generate the sequence.

```
def collatz_sequence(n):  
    if n <= 0:  
        raise ValueError("n must be a positive integer")  
    sequence = [2]  
    while len(sequence) < n:  
        if len(sequence) == 1:  
            sequence.append(1)  
        else:  
            sequence.append(sequence[-1] + sequence[-2])  
    return sequence
```

The `test_lab.py` file contains several test cases for the `collatz_sequence` function. The tests include: `test_normal_case()` which checks the sequence for `n=6`; `test_edge_case_one()` which checks the sequence for `n=1`; `test_negative_input()` which checks that a negative input raises a `ValueError`; and `test_large_case_27()` which checks the sequence for `n=27` and verifies that the last element is 76 and the length is 27.

```
def test_normal_case():  
    assert collatz_sequence(6) == [2, 1, 3, 4, 7, 11]  
  
def test_edge_case_one():  
    assert collatz_sequence(1) == [2]  
  
def test_negative_input():  
    with pytest.raises(ValueError):  
        collatz_sequence(-5)  
  
def test_large_case_27():  
    result = collatz_sequence(27)  
    # Basic correctness checks:  
    assert result[0] == 2  
    assert result[-1] == 76  
    assert len(result) == 27
```

The bottom panel of the editor shows the output of the tests, indicating that all tests passed.

Task 14 (Lucas Number Sequence – Test Case Design)

- Function: Generate Lucas sequence up to n terms.

(Starts with 2,1, then $F_n = F_{n-1} + F_{n-2}$)

- Test Cases to Design:

- Normal: $5 \rightarrow [2, 1, 3, 4, 7]$

- Edge: $1 \rightarrow [2]$

- Negative: $-5 \rightarrow \text{Error}$

- Large: 10 (last element = 76).

- Requirement: Validate correctness with pytest.

```
lab.py 2: lucas_sequence
1 def lucas_sequence(n: int) -> list[int]:
2     if n < 0:
3         raise ValueError("n must be non-negative")
4
5     if n == 0:
6         return []
7
8     if n == 1:
9         return [2]
10
11     sequence = [2, 1]
12
13     for _ in range(2, n):
14         sequence.append(sequence[-1] + sequence[-2])
15
16     return sequence

test_lab.py 2: test_lab.py
1 from lab import lucas_sequence
2 import pytest
3
4 def test_normal_case():
5     assert lucas_sequence(5) == [2, 1, 3, 4, 7]
6
7 def test_edge_case_zero():
8     assert lucas_sequence(0) == []
9
10 def test_negative_input():
11     with pytest.raises(ValueError):
12         lucas_sequence(-5)
13
14 def test_large_case():
15     result = lucas_sequence(10)
16     assert len(result) == 10
17     assert result[-1] == 76

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\briva\OneDrive\Documents\AI Assisted code> python -m pytest
PS C:\Users\briva\OneDrive\Documents\AI Assisted code> python -m pytest
===== test session starts =====
platform win32 -- python 3.11.8, pytest-8.0.2, pluggy-1.6.0
rootdir: C:\Users\briva\OneDrive\Documents\AI Assisted code
collected 4 items

test_lab.py .....

===== 4 passed in 0.00s =====
% ps c:\users\briva\one\drive\documents\ai assisted code
```

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.

```
File Edit Selection View Go Run -- + - AI Assisted Code
+ lab.py - test_lab.py

1 def count_vowels_consonants(text: str) -> tuple[int, int]:
2     vowels = set("aeiouAEIOU")
3     vowel_count = 0
4     consonant_count = 0
5
6     for char in text:
7         if char.isalpha():
8             if char in vowels:
9                 vowel_count += 1
10            else:
11                consonant_count += 1
12
13     return vowel_count, consonant_count
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

+ test_lab.py -
1 from lab import count_vowels_consonants
2
3 def test_normal_case():
4     assert count_vowels_consonants("hello") == (2, 3)
5
6 def test_empty_string():
7     assert count_vowels_consonants("") == (0, 0)
8
9 def test_only_vowels():
10    assert count_vowels_consonants("aaaaa") == (5, 0)
11
12 def test_large_text():
13    text = "This is a very long text " * 1000
14    vowels, consonants = count_vowels_consonants(text)
15
16    assert vowels > 0
17    assert consonants > 0
18    assert vowels + consonants == sum(c.isalpha() for c in text)
19
20
21
22
23
24
25
26
27
28
29
30

WINDOWS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + - - - - -
* PS C:\Users\seriv\OneDrive\Documents\AI Assisted code & C:\Users\seriv\AppData\Local\Programs\Python\Python314\python.exe "C:\Users\seriv\OneDrive\Documents\AI Assisted code\test_lab.py"
* PS C:\Users\seriv\OneDrive\Documents\AI Assisted code python -m pytest
===== test session starts =====
platform win32 -- Python 3.14.0, pytest-9.0.3, pluggy-1.6.0
rootdir: C:\Users\seriv\OneDrive\Documents\AI Assisted code
collected 4 items

test_lab.py ..... [100%]

===== 4 passed in 0.00s =====
* PS C:\Users\seriv\OneDrive\Documents\AI Assisted code >
```