

AI Assisted Coding Assignment-13.1



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

```
#2303A51543
#Task Description #1 (Refactoring – Removing Code Duplication)
# 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))
#Prompt: Identify duplicate logic and replace it with functions or classes.o Add docstring
s to all functions.
def calculate_rectangle_area(length, width):
    """
    Calculate the area of a rectangle.

    Parameters:
    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 (length * width).

    Example:
    >>> calculate_rectangle_area(5, 10)
```

```
50
"""
    return length * width
def calculate_rectangle_perimeter(length, width):
    """
    Calculate the perimeter of a rectangle.

    Parameters:
    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 ( $2 * (length + width)$ ).

    Example:
    >>> calculate_rectangle_perimeter(5, 10)
    30
    """
    return 2 * (length + width)

# Example usage
lengths = [5, 7, 10]
widths = [10, 12, 15]
for length, width in zip(lengths, widths):
    area = calculate_rectangle_area(length, width)
    perimeter = calculate_rectangle_perimeter(length, width)
    print(f"Area of Rectangle: {area}")
    print(f"Perimeter of Rectangle: {perimeter}")
```

The screenshot shows a code editor window titled "A.I.AC". The tab bar includes ".py", "ass_9_5.py", "Ass_10.2.py", "Ass_13.1.py 1" (which is the active tab), and "String_Ass1.py". The code in the editor is as follows:

```
5  #print("Area of Rectangle:", 7 * 12)
6  #print("Perimeter of Rectangle:", 2 * (7 + 12))
7  #print("Area of Rectangle:", 10 * 15)
8  #print("Perimeter of Re
9 #Prompt:Identify duplic
10 def calculate_rectangle_area(length, width):
    """
    Calculate the area of a rectangle.

    Parameters:
    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 (Length * width).

    Example:
    >>> calculate_rectangle_area(5, 10)
    50
    """
    return Length * width
11
```

The code is annotated with several comments and examples. Lines 5 through 8 are commented out with '#print'. Line 9 starts with '#Prompt:Identify duplic'. Line 10 defines a function 'calculate_rectangle_area' with parameters 'length' and 'width'. The function body contains a docstring starting with '"""', followed by a description of the function's purpose, parameter descriptions, and a return value. An example is provided below the docstring, showing the output of the function when called with arguments 5 and 10.

```

File Edit Selection View Go ...
Ass_13.1.py X
Ass_13.1.py > ...
26 def calculate_rectangle_perimeter(length, width):
27     """
28     Parameters:
29         length (int or float): The length of the rectangle.
30         width (int or float): The width of the rectangle.
31     Returns:
32         int or float: The perimeter of the rectangle (2 * (length + width)).
33     Example:
34     >>> calculate_rectangle_perimeter(5, 10)
35     30
36     """
37     return 2 * (length + width)
38
39 # Example usage
40 lengths = [5, 7, 10]
41 widths = [10, 12, 15]
42 for length, width in zip(lengths, widths):
43     area = calculate_rectangle_area(length, width)
44     perimeter = calculate_rectangle_perimeter(length, width)
45     print(f"Area of Rectangle: {area}")
46     print(f"Perimeter of Rectangle: {perimeter}")

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> & C:/Users/hruth/AppData/Local/Programs/Python/Python312/python/OneDrive/Desktop/A.I.AC/Ass_13.1.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\hruth\OneDrive\Desktop\A.I.AC>

```



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

```

#Task Description #2 (Refactoring – Extracting Reusable Functions)
# 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)
#Prompt:Execute a Well documented Code with a function calculate_total(price) that can be
reused for multiple price inputs.
def calculate_total(price):
    """
    Calculate the total price including tax.
    Parameters:
    price (int or float): The original price of the item.
    Returns:
    int or float: The total price after adding 18% tax.
    Example:
    >>> calculate_total(250)
    295.0
    """
    tax = price * 0.18 # Calculate tax as 18% of the price
    total = price + tax # Add tax to the original price to get the total
    return total

# Example usage
prices = [250, 500]
for price in prices:
    total_price = calculate_total(price)
    print(f"Total Price: {total_price}")

```

The screenshot shows a code editor window with several tabs at the top, including 'ass_9_3.py', 'Ass_10.2.py', 'Ass_13.py', 'String_Ass1.py', 'Ass_4.3.py', 'Ass_4.3(z).py', 'Ass_4.3(s).py', and 'Ass_3.3.py'. The main pane displays the refactored Python code. The 'calculate_total' function has been extracted from the original script and is now well-documented with docstrings and examples. The original script code is visible at the bottom of the editor.

```

53     #total = price + tax
54     #print("Total Price:", total)
55     #price = 500
56     #tax = price * 0.18
57     #total = price + tax
58     #print("Total Price:", total)
59     #Prompt:Execute a Well documented Code with a function calculate_total(price) that can be reused for multiple price inputs.
60
61 def calculate_total(price):
62     """
63         Calculate the total price including tax.
64
65         Parameters:
66             price (int or float): The base price.
67
68         Returns:
69             float: The total price including tax (18% tax rate).
70
71         Example:
72             >>> calculate_total(250)
73             295.0
74             """
75             tax = price * 0.18
76             return price + tax
77
78 # Example usage
79 prices = [250, 500]
80 for price in prices:
81     total = calculate_total(price)
82     print(f"Total Price: {total}")

```

The screenshot shows a code editor window with a dark theme. The main area displays a Python script named `Ass_13.1.py`. The code defines a function `calculate_total` that takes a price as input and returns the total price including 18% tax. The code includes comments explaining the parameters, returns, and example usage. Below the code editor is a terminal window showing the execution of the script and its output.

```
File Edit Selection View Go ... ↵ → AJAC 08 W
:py ass_9.5.py Ass_10.2.py Ass_13.1.py ✘ String.Ass1.py Ass_4.5.py Ass_4.5(2).py Ass_4.5(3).py Ass_5.5.py
Ass_13.1.py > calculate_total
58     #print('Total Price:', total)
59     #Prompt:Execute a Well documented Code with a function calculate_total(price) that can be reused for multiple price inputs.
60     def calculate_total(price):
61         """
62             Calculate the total price including tax.
63
64             Parameters:
65                 price (int or float): The original price of the item.
66
67             Returns:
68                 int or float: The total price after adding 18% tax.
69
70             Example:
71             >>> calculate_total(250)
72             295.0
73             """
74             tax = price * 0.18 # Calculate tax as 18% of the price
75             total = price + tax # Add tax to the original price to get the total
76             return total
77
78     # Example usage
79     prices = [250, 500]
80     for price in prices:
81         total_price = calculate_total(price)
82         print("Total Price: {total_price}")
83
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> & C:/Users/hruth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/hruth/OneDrive/Desktop/A.I.AC
Total Price: 295.0
Total Price: 590.0
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>
```



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 $\leq 100 \rightarrow$ "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.

1. Include proper docstrings for:

- The class
- The method (with parameter and return descriptions)

2. 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.

```
#Task Description #3: Refactoring Using Classes and Methods (Eliminating Redundant 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")
#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.
#Include proper docstrings for:
#The class
#The method (with parameter and return descriptions)
#The method must be reusable and called multiple times without rewriting conditional logic.
class GradeCalculator:
    """
        A class to calculate the grade based on student marks.
    Methods:

```

```

calculate_grade(marks): Accepts marks as input and returns the corresponding grade.
"""
def calculate_grade(self, marks):
    """
    Calculate the grade based on the given marks.
    Parameters:
    marks (int): The marks obtained by the student (0 to 100).
    Returns:
    str: The grade corresponding to the marks.
    Grading Logic:
    Marks ≥ 90 and ≤ 100 → "Grade A"
    Marks ≥ 80 → "Grade B"
    Marks ≥ 70 → "Grade C"
    Marks ≥ 40 → "Grade D"
    Marks ≥ 0 → "Fail"
    """
    if 90 <= marks <= 100:
        return "Grade A"
    elif marks >= 80:
        return "Grade B"
    elif marks >= 70:
        return "Grade C"
    elif marks >= 40:
        return "Grade D"
    elif marks >= 0:
        return "Fail"
    else:
        raise ValueError("Marks should be between 0 and 100.")
# Example usage
grade_calculator = GradeCalculator()
marks_list = [85, 72, 95, 65, 30]
for marks in marks_list:
    grade = grade_calculator.calculate_grade(marks)
    print(f"Marks: {marks}, Grade: {grade}")

```

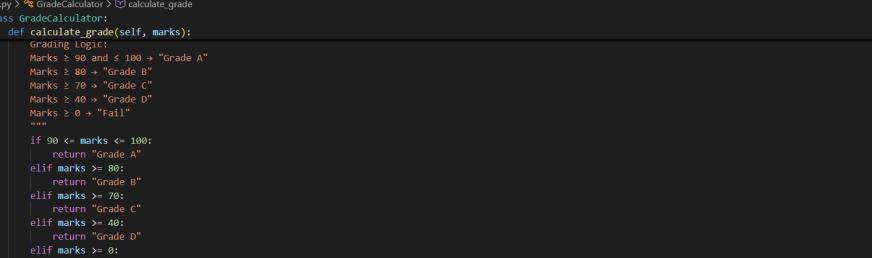
```
File Edit Selection View ... ← → Q A.IAC
.ass_9_5.py Ass_10.2.py Ass_13.1.py • String_Ass1.py Ass_4.5.py Ass_4.5(2).py Ass_4.5(3).py
1 Ass_13.1.py > ...
114     title class
115     #The method (with parameter and return descriptions)
116     #The method must be reusable and called multiple times without rewriting conditional logic.
117     class GradeCalculator:
        """
        A class to calculate the grade based on student marks.

        Methods:
        calculate_grade(marks): Accepts marks as input and returns the corresponding grade.
        """
        def calculate_grade(self, marks):
            """
            Calculate the grade based on the given marks.

            Parameters:
            marks (int): The marks obtained by the student (0 to 100).

            Returns:
            str: The grade corresponding to the marks.

            Grading Logic:
            Marks ≥ 90 and ≤ 100 → "Grade A"
            Marks ≥ 80 → "Grade B"
            Marks ≥ 70 → "Grade C"
            Marks ≥ 40 → "Grade D"
            Marks ≥ 0 → "Fail"
            """
            if 90 <= marks <= 100:
                return "Grade A"
            elif marks >= 80:
                return "Grade B"
            elif marks >= 70:
                return "Grade C"
            elif marks >= 40:
                return "Grade D"
            elif marks >= 0:
                return "Fail."
        
```



A screenshot of the Visual Studio Code (VS Code) interface. The title bar shows "File Edit Selection View ..." and "Q A.I.AC". The left sidebar has icons for file operations like Open, Save, Find, and others. The main editor area displays Python code for a "GradeCalculator" class. The code defines a "calculate_grade" method that takes marks as input and returns a grade based on the following logic:

- Marks ≥ 90 and < 100 → "Grade A"
- Marks ≥ 80 → "Grade B"
- Marks ≥ 70 → "Grade C"
- Marks ≥ 40 → "Grade D"
- Marks ≥ 0 → "Fail"

The code includes a check for marks between 0 and 100, and a raise statement for marks outside this range. Below the code, there's a section titled "# Example usage:" followed by a snippet of code demonstrating how to use the calculator. The bottom status bar shows the command palette open and the path "C:\Users\hruth\OneDrive\Desktop\A.I.AC> c:\Users\hruth\AppData\Local\Programs\Python\Python312\python.exe c:/Users/hruth/OneDrive/Desktop/A.I.AC/Ass_13.1.py".

```
File Edit Selection View ... ← → Q A.I.AC
.py ass_9_5.py Ass_10.2.py Ass_13.1.py ✘ String_Ass1.py Ass_4.5.py Ass_4.5(2).py Ass_4.5(3).py Ass_5.5.py Ass_6.5.py Ass_7.5.py

Ass_13.1.py ✘ GradeCalculator > calculate_grade
117 class GradeCalculator:
118     def calculate_grade(self, marks):
119         Grading logic:
120         Marks ≥ 90 and < 100 → "Grade A"
121         Marks ≥ 80 → "Grade B"
122         Marks ≥ 70 → "Grade C"
123         Marks ≥ 40 → "Grade D"
124         Marks ≥ 0 → "Fail"
125         """
126         if 90 < marks <= 100:
127             return "Grade A"
128         elif marks >= 80:
129             return "Grade B"
130         elif marks >= 70:
131             return "Grade C"
132         elif marks >= 40:
133             return "Grade D"
134         elif marks >= 0:
135             return "Fail"
136         else:
137             raise ValueError("Marks should be between 0 and 100.")
138
# Example usage:
139 grade_calculator = GradeCalculator()
140 marks_list = [85, 72, 95, 65, 30]
141 for marks in marks_list:
142     grade = grade_calculator.calculate_grade(marks)
143     print(f"Marks: {marks}, Grade: {grade}")

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> c:\Users\hruth\AppData\Local\Programs\Python\Python312\python.exe c:/Users/hruth/OneDrive/Desktop/A.I.AC/Ass_13.1.py
Marks: 85, Grade: Grade B
Marks: 72, Grade: Grade C
Marks: 95, Grade: Grade A
Marks: 65, Grade: Grade D
Marks: 30, Grade: Fall
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>
```



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().

```
#Task Description #4 (Refactoring – Converting Procedural Code to Functions)
#Sample Legacy Code:
#num = int(input("Enter number: "))
#square = num * num
#print("Square:", square)
#Prompt: Generate a modular code with docstrings using functions like get_input(), calculate_square(), and display_result(). To refactor procedural input-processing logic into functions.
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 given number.
    Parameters:
    num (int): The number to be squared.
    Returns:
    int: The square of the input number.
    """
    return num * num
def display_result(square):
    """
    Display the result of the square calculation.
    Parameters:
    square (int): The squared value to be displayed.
    """
    print("Square:", square)
# Main function to orchestrate the flow of the program
def main():
    number = get_input() # Get user input
```

```

    square = calculate_square(number) # Calculate the square of the input number
    display_result(square) # Display the result
# Execute the main function
if __name__ == "__main__":
    main()

```

```

7
8     #Task Description #4 (Refactoring - Converting Procedural Code to Functions)
9     #Sample Legacy Code:
10    #num = int(input("Enter number: "))
11    #square = num * num
12    #print("Square:", square)
13    #Prompt:Generate a modular code with docstrings using functions like get_input(), calculate_square(), and display_
14    #result()
15
16    """
17        Prompt the user to enter a number and return it as an integer.
18
19        Returns:
20            int: The number entered by the user.
21        """
22
23    return int(input("Enter number: "))
24
25
26
27
28
29

```

The screenshot shows the Visual Studio Code interface with the file `Ass_13.1.py` open. The code has been refactored into modular functions:

```

def calculate_square(num):
    """
    return num * num
"""

def display_result(square):
    """
    Display the result of the square calculation.
    Parameters:
        square (int): The squared value to be displayed.
    """
    print("Square:", square)

def main():
    number = get_input() # Get user input
    square = calculate_square(number) # Calculate the square of the input number
    display_result(square) # Display the result

if __name__ == "__main__":
    main()

```

The terminal at the bottom shows the output of running the program:

```

PS C:\Users\hruth\OneDrive\Desktop\A.I.AC & C:/Users/hruth/AppData/Local/Programs/Python/Python312/python Ass_13.1.py
Marks: 30, Grade: Fail
Enter number: 1543
Square: 2380849

```



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 5 (Refactoring Procedural Code into OOP Design)
#Legacy Code:
#salary = 50000
#tax = salary * 0.2
#net = salary - tax
#print(net)
#Prompt: Refactor procedural code into a class-based design like EmployeeSalaryCalculator
with methods and attributes.
class EmployeeSalaryCalculator:
    """
    A class to calculate the net salary of an employee after tax deduction.
    Attributes:
        salary (float): The gross salary of the employee.
    Methods:
        calculate_net_salary(): Calculates and returns the net salary after tax deduction.
    """
    def __init__(self, salary):
        """
        Initialize the EmployeeSalaryCalculator with the given salary.
        Parameters:
            salary (float): The gross salary of the employee.
        """
        self.salary = salary
    def calculate_net_salary(self):
        """
        Calculate the net salary by deducting 20% tax from the gross salary.
        Returns:
            float: The net salary after tax deduction.
        """
        tax = self.salary * 0.2 # Calculate tax as 20% of the gross salary
        net_salary = self.salary - tax # Subtract tax from gross salary to get net salary
        return net_salary
    # Example usage
    employee_salary_calculator = EmployeeSalaryCalculator(50000) # Create an instance with a
    salary of 50000
    net_salary = employee_salary_calculator.calculate_net_salary() # Calculate the net salary
```

```
y
print(net_salary) # Expected output: 40000.0
```

```

205 #Prompt:Refactor procedural code into a class-based design like EmployeeSalaryCalculator with methods and attributes.
206 class EmployeeSalaryCalculator:
    """
    A class to calculate the net salary of an employee after tax deduction.

    Attributes:
    salary (float): The gross salary of the employee.

    Methods:
    calculate_net_salary(): Calculates and returns the net salary after tax deduction.
    """
    def __init__(self, salary):
        """
        Initialize the EmployeeSalaryCalculator with the given salary.

        Parameters:
        salary (float): The gross salary of the employee.
        """
        self.salary = salary
    def calculate_net_salary(self):
        """
        Calculate the net salary by deducting 20% tax from the gross salary.

        Returns:
        float: The net salary after tax deduction.
        """
        tax = self.salary * 0.2 # Calculate tax as 20% of the gross salary
        net_salary = self.salary - tax # Subtract tax from gross salary to get net salary
        return net_salary

```

```

206 class EmployeeSalaryCalculator:
    """
    calculate_net_salary(): Calculates and returns the net salary after tax deduction.
    """
    def __init__(self, salary):
        """
        Initialize the EmployeeSalaryCalculator with the given salary.

        Parameters:
        salary (float): The gross salary of the employee.
        """
        self.salary = salary
    def calculate_net_salary(self):
        """
        Calculate the net salary by deducting 20% tax from the gross salary.

        Returns:
        float: The net salary after tax deduction.
        """
        tax = self.salary * 0.2 # Calculate tax as 20% of the gross salary
        net_salary = self.salary - tax # Subtract tax from gross salary to get net salary
        return net_salary
    # Example usage
    employee_salary_calculator = EmployeeSalaryCalculator(50000) # Create an instance with a salary of 50000
    net_salary = employee_salary_calculator.calculate_net_salary() # Calculate the net salary
    print(net_salary) # Expected output: 40000.0

```



Task 6 (Optimizing Search Logic)

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

Legacy Code:

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

Expected Outcome:

- o Use of sets or dictionaries with complexity justification

```
#Task 6 (Optimizing Search Logic)
#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")
#Prompt:Refactor inefficient linear searches using appropriate data structures.
#Output:Use of sets or dictionaries with complexity justification
#Focus Areas:Time complexity and Data structure choice
def check_user_access(username, user_set):
    """
    Check if the given username exists in the user set and return access status.
    Parameters:
    username (str): The username to be checked.
    user_set (set): A set of valid usernames for access control.
    Returns:
    str: "Access Granted" if the username is found, otherwise "Access Denied".
    Time Complexity:
    The use of a set allows for O(1) average time complexity for membership checks,
    making it more efficient than a list which has O(n) time complexity for the same operation.
    """
    return "Access Granted" if username in user_set else "Access Denied"
# Example usage
users = {"admin", "guest", "editor", "viewer"} # Using a set
name = input("Enter username: ") # Get username input from the user
access_status = check_user_access(name, users) # Check access status
print(access_status) # Print the access status
```

The screenshot shows a dark-themed code editor with a sidebar containing various icons. The main area displays a Python script named `Ass_13.1.py`. The code defines a function `check_user_access` that checks if a given username is in a user set. It includes comments explaining the use of sets for O(1) time complexity and the function's parameters and returns.

```
242     name = input("Enter username: ")
243     found = False
244     for u in users:
245         if u == name:
246             found = True
247     #print("Access Granted" if found else "Access Denied")
248     #Prompt:Refactor inefficient linear searches using appropriate data structures.
249     #Output:Use of sets or dictionaries with complexity justification
250     #Focus Areas:Time complexity and Data structure choice
251     def check_user_access(username, user_set):
252         """
253             Check if the given username exists in the user set and return access status.
254
255             Parameters:
256             username (str): The username to be checked.
257             user_set (set): A set of valid usernames for access control.
258
259             Returns:
260             str: "Access Granted" if the username is found, otherwise "Access Denied".
261
262             Time Complexity:
263             The use of a set allows for O(1) average time complexity for membership checks,
264             making it more efficient than a list which has O(n) time complexity for the same operation.
265         """
266         return "Access Granted" if username in user_set else "Access Denied"
```

The screenshot shows the same code editor with the script `Ass_13.1.py` running. The terminal tab at the bottom shows the command `python Ass_13.1.py` being run, followed by the output: "Enter username: admin" and "Access Granted".

```
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
File Edit Selection View Go ...
Q AI.AC
242     name = input("Enter username: ")
243     found = False
244     for u in users:
245         if u == name:
246             found = True
247     #print("Access Granted" if found else "Access Denied")
248     #Prompt:Refactor inefficient linear searches using appropriate data structures.
249     #Output:Use of sets or dictionaries with complexity justification
250     #Focus Areas:Time complexity and Data structure choice
251     def check_user_access(username, user_set):
252         """
253             Check if the given username exists in the user set and return access status.
254
255             Parameters:
256             username (str): The username to be checked.
257             user_set (set): A set of valid usernames for access control.
258
259             Returns:
260             str: "Access Granted" if the username is found, otherwise "Access Denied".
261
262             Time Complexity:
263             The use of a set allows for O(1) average time complexity for membership checks,
264             making it more efficient than a list which has O(n) time complexity for the same operation.
265         """
266         return "Access Granted" if username in user_set else "Access Denied"
267
268 users = {"admin", "guest", "editor", "viewer"} # Using a set
269 name = input("Enter username: ") # Get username input from the user
270 access_status = check_user_access(name, users) # Check access status
271 print(access_status) # Print the access status
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> & C:/Users/hruth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/Ass_13.1.py
Enter username: admin
Access Granted
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>
```



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.")

```

```

#Task 7 – Refactoring the Library Management System
# 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.")
#Prompt: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.
# library.py
class Library:
    """
    A class to represent a library management system.

    Attributes:
    library_db (dict): A dictionary to store book information with ISBN as the key.

    Methods:
    add_book(title, author, isbn): Adds a book to the library database.
    remove_book(isbn): Removes a book from the library database using its ISBN.
    search_book(isbn): Searches for a book in the library database using its ISBN.
    """
    def __init__(self):
        """Initialize the Library with an empty database."""
        self.library_db = {}
    def add_book(self, title, author, isbn):
        """
        Add a book to the library database.

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

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

```

```

Remove a book from the library database using its ISBN.

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

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

Parameters:
isbn (str): The ISBN number of the book to be searched.

Returns:
str: A message indicating whether the book was found along with its details or if
it was not found.
"""
if isbn in self.library_db:
    return f"Book Found: {self.library_db[isbn]}"
else:
    return "Book not found."
# Example usage
library = Library() # Create an instance of the Library class
print(library.add_book("Python Basics", "John Doe", "101")) # Add first book
print(library.add_book("AI Fundamentals", "Jane Smith", "102")) # Add second book
print(library.search_book("101")) # Search for the first book
print(library.remove_book("101")) # Remove the first book
print(library.search_book("101")) # Search for the first book again after removal

```

```

Ass_13.1.py > ...
23     # library.py
24     class Library:
25         """
26             A class to represent a Library management system.
27
28             Attributes:
29                 library_db (dict): A dictionary to store book information with ISBN as the key.
30
31             Methods:
32                 add_book(title, author, isbn): Adds a book to the Library database.
33                 remove_book(isbn): Removes a book from the Library database using its ISBN.
34                 search_book(isbn): Searches for a book in the Library database using its ISBN.
35             """
36
37     def __init__(self):
38         """Initialize the Library with an empty database."""
39         self.library_db = {}
40
41     def add_book(self, title, author, isbn):
42         """
43             Add a book to the Library database.
44
45             Parameters:
46                 title (str): The title of the book.
47                 author (str): The author of the book.
48                 isbn (str): The ISBN number of the book.
49
50             Returns:
51                 str: A message indicating whether the book was added successfully or if it already exists.
52             """
53
54         if isbn not in self.library_db:
55             self.library_db[isbn] = {"title": title, "author": author}
56             return f"Book Added: {isbn}"
57         else:
58             return f"Book already exists: {isbn}"
59
60     def remove_book(self, isbn):
61         """
62             Remove a book from the Library database using its ISBN.
63
64             Parameters:
65                 isbn (str): The ISBN number of the book to be removed.
66
67             Returns:
68                 str: A message indicating whether the book was removed successfully or not.
69             """
70
71         if isbn in self.library_db:
72             del self.library_db[isbn]
73             return f"Book Removed: {isbn}"
74         else:
75             return f"Book not found: {isbn}"
76
77     def search_book(self, isbn):
78         """
79             Search for a book in the library database using its ISBN.
80
81             Parameters:
82                 isbn (str): The ISBN number of the book to be searched.
83
84             Returns:
85                 str: A message indicating whether the book was found along with its details or if it was not found.
86             """
87
88         if isbn in self.library_db:
89             return f"Book Found: {self.library_db[isbn]}"
90         else:
91             return f"Book not found: {isbn}"
92
93     # Example usage
94
95     library = Library() # Create an instance of the Library class
96     print(library.add_book("Python Basics", "John Doe", "101")) # Add first book
97     print(library.add_book("AI Fundamentals", "Jane Smith", "102")) # Add second book
98     print(library.search_book("101")) # Search for the first book
99     print(library.remove_book("101")) # Remove the first book
100    print(library.search_book("101")) # Search for the first book again after removal

```

```

File Edit Selection View Go ... ← → Q AI.AC
Ass_13.1.py > Library > search_book
324     class Library:
325         def search_book(self, isbn):
326             """
327                 Search for a book in the library database using its ISBN.
328
329                 Parameters:
330                     isbn (str): The ISBN number of the book to be searched.
331
332                 Returns:
333                     str: A message indicating whether the book was found along with its details or if it was not found.
334             """
335
336             if isbn in self.library_db:
337                 return f"Book Found: {self.library_db[isbn]}"
338             else:
339                 return "Book not found."
340
341     # Example usage
342
343     library = Library() # Create an instance of the Library class
344     print(library.add_book("Python Basics", "John Doe", "101")) # Add first book
345     print(library.add_book("AI Fundamentals", "Jane Smith", "102")) # Add second book
346     print(library.search_book("101")) # Search for the first book
347     print(library.remove_book("101")) # Remove the first book
348     print(library.search_book("101")) # Search for the first book again after removal

```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> & C:/Users/hruth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/hruth/OneDrive/Desktop/A.I.AC/Ass_13.1.py

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

PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>

```
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> python -m pydoc library
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> python -m pydoc library
Book added successfully.
Book added successfully.
Book Found: {'title': 'Python Basics', 'author': 'John Doe'}
Book removed successfully.
Book not found.
Help on module library:

NAME
    library - # library.py

CLASSES
    builtins.object
        Library

    class Library(builtins.object)
        | A class to represent a library management system.

        | Attributes:
        |     library_db (dict): A dictionary to store book information with ISBN as the key.

        | Methods:
        |     add_book(title, author, isbn): Adds a book to the library database.
        |     remove_book(isbn): Removes a book from the library database using its ISBN.
        |     search_book(isbn): Searches for a book in the library database using its ISBN.

        | Methods defined here:
        |
        |     __init__(self)
        |
        |     ... More ...
```

```
    | Initialize the Library with an empty database.

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

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

        Returns:
        str: A message indicating whether the book was added successfully or if it already exists.

    remove_book(self, isbn)
        Remove a book from the library database using its ISBN.

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

        Returns:
        str: A message indicating whether the book was removed successfully or if it was not found.

    search_book(self, isbn)
        Search for a book in the library database using its ISBN.

        Parameters:
        isbn (str): The ISBN number of the book to be searched.

        Returns:
        str: A message indicating whether the book was found along with its details or if it was not found.

    -----
    Data descriptors defined here:
    _dict_
```

Ln 393, Col 1 Spaces: 4 UTF-8 CRLF { } Python 3.12.3

```

ASS-9.5(1543).pdf
ASS-10.2(1543).pdf
data.txt
labexam.py
library.html
library.py
String_Ass1.py
String.py
user_activity.log
Week5_Documentatio...

```

Data descriptors defined here:

- `__dict__` dictionary for instance variables
- `__weakref__` list of weak references to the object

DATA

```
library = <library.Library object>
```

FILE

```
c:\users\hruth\onedrive\Desktop\A.I.AC\library.py
```

PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>

← ⏪ ⓘ 127.0.0.1:5500/library.html

Home - NetMirror Classroom - GDB o... Google Cisco Networking... DataCamp H

`remove_book(isbn)`: Removes a book from the library database using its ISBN.
`search_book(isbn)`: Searches for a book in the library database using its ISBN.

Methods defined here:

`__init__(self)`
 Initialize the `Library` with an empty database.

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

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

Returns:
`str`: A message indicating whether the book was added successfully or if it already exists.

`remove_book(self, isbn)`
 Remove a book from the library database using its ISBN.

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

Returns:
`str`: A message indicating whether the book was removed successfully or if it was not found.

`search_book(self, isbn)`
 Search for a book in the library database using its ISBN.

Parameters:
`isbn` (str): The ISBN number of the book to be searched.

Returns:
`str`: A message indicating whether the book was found along with its details or if it was not found.

Data descriptors defined here:

- `__dict__` dictionary for instance variables
- `__weakref__` list of weak references to the object

Data

```
library = <library.Library object>
```



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
```

```
#Task 8– Fibonacci Generator
#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
#Prompt:Refactor into a clean reusable function (generate_fibonacci).Add docstrings and test cases.Compare AI-refactored vs original.
def generate_fibonacci(n):
    """
    Generate a list of Fibonacci numbers up to the nth number.
    Parameters:
    n (int): The number of Fibonacci numbers to generate.
    Returns:
    list: A list containing the Fibonacci sequence up to the nth number.
    Example:
```

```

>>> generate_fibonacci(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
"""

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

# Example usage
print(generate_fibonacci(10)) # Expected output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

```

```

◆ Ass_13.1.py > ...
400  #print(b)
401  #for i in range(2,n):
402  # c=a+b
403  # print(c)
404  # a=b
405  # b=c
406 #Prompt:Refactor into a clean reusable function (generate_fibonacci).Add docstrings and test cases.Compare AI-refactored vs original.
407 def generate_fibonacci():
    """
    Generate a List of Fibonacci numbers up to the nth number.

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

    Returns:
    List: A list containing the Fibonacci sequence up to the nth number.

    Example:
    >>> generate_fibonacci(10)
    [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
    """
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]

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

    return fib_sequence
408
409

```

```

❸ Ass_13.1.py > ...
405 # D=C
406 #prompt:Refactor into a clean reusable function (generate_fibonacci).Add docstrings and test cases.Compare AI-refactored vs original.
407 def generate_fibonacci(n):
408     """
409         Generate a list of Fibonacci numbers up to the nth number.
410
411     Parameters:
412         n (int): The number of Fibonacci numbers to generate.
413
414     Returns:
415         list: A list containing the Fibonacci sequence up to the nth number.
416
417     Example:
418     >>> generate_fibonacci(10)
419     [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
420     """
421     if n <= 0:
422         return []
423     elif n == 1:
424         return [0]
425     elif n == 2:
426         return [0, 1]
427
428     fib_sequence = [0, 1]
429     for i in range(2, n):
430         next_fib = fib_sequence[i-1] + fib_sequence[i-2]
431         fib_sequence.append(next_fib)
432
433     return fib_sequence
434 # Example usage
435 print(generate_fibonacci(10)) # Expected output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
436

```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\hruthi\OneDrive\Desktop\A.I.AC]
```



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")
```

```

#Task 9 – Twin Primes Checker
#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")
#Prompt:Refactor into is_prime(n) and is_twin_prime(p1, p2).Add docstrings and optimize.G
enerate a list of twin primes in a given range.
def is_prime(n):
"""
Check if a number is prime.
Parameters:
n (int): The number to check for primality
Returns:
bool: True if the number is prime, False otherwise.
Example:
>>> is_prime(11)
True
>>> is_prime(12)
False
"""
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.
Parameters:
p1 (int): The first prime number.
p2 (int): The second prime number.
Returns:
bool: True if the numbers are twin primes, False otherwise.
Example:
>>> is_twin_prime(11, 13)
True
>>> is_twin_prime(12, 14)

```

```

False
"""

    return is_prime(p1) and is_prime(p2) and abs(p1 - p2) == 2
def generate_twin_primes(start, end):
    """
    Generate a list of twin primes within a given range.

    Parameters:
    start (int): The starting number of the range.
    end (int): The ending number of the range.

    Returns:
    list: A list of tuples, each containing a pair of twin primes.

    Example:
    >>> generate_twin_primes(10, 30)
    [(11, 13), (17, 19), (29, 31)]
    """

    twin_primes = []
    for num in range(start, end - 1):
        if is_twin_prime(num, num + 2):
            twin_primes.append((num, num + 2))
    return twin_primes

# Example usage
print(generate_twin_primes(10, 30)) # Expected output: [(11, 13), (17, 19)]

```

The screenshot shows a code editor window with the following details:

- File Explorer:** Shows files like `ass_9.5.py`, `Ass_10.2.py`, `Ass_13.1.py`, `library.py`, `String_Ass1.py`, `Ass_4.5.py`, `Ass_4.5(2).py`, `Ass_4.5(3).py`, and `Ass_4.5(4).py`.
- Code Area:**

```

455     #Prompt: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.
456     def is_prime(n):
        """
        Check if a number is prime.

        Parameters:
        n (int): The number to check for primality.

        Returns:
        bool: True if the number is prime, False otherwise.

        Example:
        >>> is_prime(11)
        True
        >>> is_prime(12)
        False
        """
        if n <= 1:
            return False
        for i in range(2, int(n**0.5) + 1):
            if n % i == 0:
                return False
        return True
    
```
- Right Panel:** Shows a vertical stack of tabs or windows related to the current project.
- Bottom:** Shows page numbers 456 and 457.

The screenshot shows a code editor interface with a dark theme. The main area displays Python code for generating twin primes. The code includes docstrings and examples. The terminal tab at the bottom shows the output of running the script.

```
File Edit Selection View Go ... A.I.AC
ass_9.py Ass_10.2.py Ass_13.1.py library.py library.html String_Ass1.py

1 Ass_13.1.py > ...
478 def is_twin_prime(p1, p2):
489     >>> is_twin_prime(12, 14)
490     False
491     """
492     |     return is_prime(p1) and is_prime(p2) and abs(p1 - p2) == 2
493     def generate_twin_primes(start, end):
494         """
495         Generate a list of twin primes within a given range.
496         Parameters:
497         start (int): The starting number of the range.
498         end (int): The ending number of the range.
499         Returns:
500         list: A list of tuples, each containing a pair of twin primes.
501         Example:
502         >>> generate_twin_primes(10, 30)
503         [(11, 13), (17, 19), (29, 31)]
504         """
505         twin_primes = []
506         for num in range(start, end - 1):
507             if is_twin_prime(num, num + 2):
508                 twin_primes.append((num, num + 2))
509         return twin_primes
510 # Example usage
511 print(generate_twin_primes(10, 30)) # Expected output: [(11, 13), (17, 19)]
```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> & C:/Users/hruth/AppData/Local/Programs/Python/Python312/python.
[(11, 13), (17, 19)]
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> 
```



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.

```
#Task 10 – Refactoring the Chinese Zodiac Program
# 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")

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

def get_zodiac(year):
    """
    Get the Chinese Zodiac sign for a given year.

    Parameters:
```

```

year (int): The year for which to determine the Chinese Zodiac sign.
Returns:
str: The Chinese Zodiac sign corresponding to the given year.
Example:
>>> get_zodiac(2020)
'Rat'
>>> get_zodiac(2021)
'0x'
"""
zodiac_signs = [
    "Monkey", "Rooster", "Dog", "Pig", "Rat", "0x",
    "Tiger", "Rabbit", "Dragon", "Snake", "Horse", "Goat"
]
return zodiac_signs[year % 12]

# Example usage
year = int(input("Enter a year: ")) # Get user input for the year
zodiac_sign = get_zodiac(year) # Get the corresponding Chinese Zodiac sign
print(zodiac_sign) # Print the Zodiac sign

```

```

1 Ass_13.1.py > ...
543     #Separate input handling from logic.
544     #Improve readability and maintainability.
545     #Ensure output remains correct.
546 def get_zodiac(year):
    """
    Get the Chinese Zodiac sign for a given year.

    Parameters:
    year (int): The year for which to determine the Chinese Zodiac sign.

    Returns:
    str: The Chinese Zodiac sign corresponding to the given year.

    Example:
    >>> get_zodiac(2020)
    'Rat'
    >>> get_zodiac(2021)
    '0x'
    """
    zodiac_signs = [
        "Monkey", "Rooster", "Dog", "Pig", "Rat", "0x",
        "Tiger", "Rabbit", "Dragon", "Snake", "Horse", "Goat"
    ]
    return zodiac_signs[year % 12]

```

The screenshot shows a code editor interface with a dark theme. The top menu bar includes File, Edit, Selection, View, Go, and a search bar labeled 'A.I.AC'. Below the menu is a tab bar with several files: ass_9_5.py, Ass_10.2.py, Ass_13.1.py (which is the active file), library.py, library.html, and String_Ass1.py. The main code area contains the following Python script:

```
546 def get_zodiac(year):
547     """
548     Parameters:
549     year (int): The year for which to determine the Chinese Zodiac sign.
550
551     Returns:
552     str: The Chinese Zodiac sign corresponding to the given year.
553
554     Example:
555     >>> get_zodiac(2020)
556     'Rat'
557     >>> get_zodiac(2021)
558     'Ox'
559     """
560
561     zodiac_signs = [
562         "Monkey", "Rooster", "Dog", "Pig", "Rat", "Ox",
563         "Tiger", "Rabbit", "Dragon", "Snake", "Horse", "Goat"
564     ]
565     return zodiac_signs[year % 12]
566
567 # Example usage
568 year = int(input("Enter a year: ")) # Get user input for the year
569 zodiac_sign = get_zodiac(year) # Get the corresponding Chinese Zodiac sign
570 print(zodiac_sign) # Print the Zodiac sign
```

The terminal pane at the bottom shows the output of the script:

```
...  
Enter a year: 2026  
Horse  
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>
```



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:**

- $18 \rightarrow 1 + 8 = 9 \rightarrow 18 \div 9 = 2 \checkmark$ (Harshad Number)
- $19 \rightarrow 1 + 9 = 10 \rightarrow 19 \div 10 \neq \text{integer} \times$ (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).

```
#Task 11 – Refactoring the Harshad (Niven) Number Checker
#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")
#Prompt: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. Ensure the program handles edge cases (e.g., 0, negative numbers).
def is_harshad(number):
"""
Check if a number is a Harshad (Niven) number.
Parameters:
number (int): The number to be checked.
Returns:
bool: True if the number is a Harshad number, False otherwise.
A Harshad number is an integer that is divisible by the sum of its digits.
Example:
>>> is_harshad(18)
True
>>> is_harshad(19)
False
>>> is_harshad(0)
False
>>> is_harshad(-12)
False
"""
if number < 0:
    return False # Harshad numbers are typically defined for non-negative integers
temp = abs(number) # Use absolute value to handle negative numbers
sum_digits = 0
while temp > 0:
    digit = temp % 10
    sum_digits += digit
    temp //= 10
if sum_digits == 0:
    return False # Avoid division by zero
return number % sum_digits == 0
# Example usage
num = int(input("Enter a number: ")) # Get user input for the number

```

```
result = is_harshad(num) # Check if the number is a Harsh
print(result) # Print the result (True or False)
```

The screenshot shows a code editor window with a dark theme. The file is named 'Ass_13.1.py'. The code defines a function 'is_harshad' that checks if a number is a Harshad number (Niven number). The function includes a docstring with parameters, returns, examples, and a main logic block for numbers other than zero. The code uses standard Python syntax with imports like 'int' and 'str'. The status bar at the bottom indicates the file is at line 594, column 3, with 4 spaces and UTF-8 encoding.

```
 Ass_13.1.py > ...
593     #4. Add proper docstrings.
594     #5. Ensure the program handles edge cases (e.g., 0, negative numbers).
595
596 ✓ def is_harshad(number):
597     """
598         Check if a number is a Harshad (Niven) number.
599
600     Parameters:
601         number (int): The number to check.
602
603     Returns:
604         bool: True if the number is a Harshad number, False otherwise.
605
606     Example:
607     >>> is_harshad(18)
608     True
609     >>> is_harshad(19)
610     False
611     """
612
613     if number == 0:
614         return False
615     sum_digits = sum(int(digit) for digit in str(abs(number)))
616     if sum_digits == 0:
617         return False
618     return number % sum_digits == 0
619
620     # Example usage
621     number = int(input("Enter a number: "))
622     print(is_harshad(number))
```

The screenshot shows a code editor interface with a dark theme. The main area displays Python code for determining if a number is a Harshad number. The code uses a while loop to calculate the sum of digits and checks if the number is divisible by the sum. The terminal below shows the code running and outputting 'True' for the input '18'. The status bar at the bottom indicates the file is saved in Python.

```
596 def is_harshad(number):
597     """
598     >>> is_harshad(19)
599     False
600     >>> is_harshad(0)
601     False
602     >>> is_harshad(-12)
603     False
604     """
605
606     if number < 0:
607         return False # Harshad numbers are typically defined for non-negative integers
608     temp = abs(number) # Use absolute value to handle negative numbers
609     sum_digits = 0
610     while temp > 0:
611         digit = temp % 10
612         sum_digits += digit
613         temp //= 10
614     if sum_digits == 0:
615         return False # Avoid division by zero
616     return number % sum_digits == 0
617
618 # Example usage
619 num = int(input("Enter a number: ")) # Get user input for the number
620 result = is_harshad(num) # Check if the number is a Harshad number
621 print(result) # Print the result (True or False)
622
623
624
625
626
627
628
629
630
631
632
633
```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

Enter a number: 18
True

PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>

Ln 614, Col 10 Spaces: 4 UTF-8 CRLF {} Python



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).

```
#Task 12 – Refactoring the Factorial Trailing Zeros Program
#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)
#Prompt: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).
def count_trailing_zeros(n):
    """
    Count the number of trailing zeros in the factorial of a given non-negative integer.

    Parameters:
    n (int): A non-negative integer for which to count trailing zeros in n!.

    Returns:
    int: The number of trailing zeros in n!.

    The number of trailing zeros in n! can be found by counting the number of times 5 is
    a factor in the numbers from 1 to n,
    since 10 is made of 2 and 5, and there are usually more factors of 2 than 5 in a fact
    orial.

    Example:
    >>> count_trailing_zeros(100)
    24
    >>> count_trailing_zeros(0)
    0
    >>> count_trailing_zeros(-5)
    ValueError: Input must be a non-negative integer.

    """
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    count = 0
    power_of_5 = 5
    while power_of_5 <= n:
        count += n // power_of_5
        power_of_5 *= 5
    return count

# Example usage
n = int(input("Enter a number: ")) # Get user input for the number
try:
    trailing_zeros = count_trailing_zeros(n) # Count the trailing zeros in n!
    print("Trailing zeros:", trailing_zeros) # Print the result
except ValueError as e:
    print(e) # Print the error message if input is invalid

```

```
ass_9_5.py Ass_10_2.py Ass_13.1.py library.py library.html String_Ass1.py Ass_4_3.py Ass_4_3(2).py Ass_13.1.py > ...
654     # Handle edge cases (e.g., negative numbers, zero).
655     def count_trailing_zeros(n):
656         """
657             Count the number of trailing zeros in the factorial of a given non-negative integer.
658
659             Parameters:
660                 n (int): A non-negative integer for which to count trailing zeros in n!.
661
662             Returns:
663                 int: The number of trailing zeros in n!.
664
665             The number of trailing zeros in n! can be found by counting the number of times 5 is a factor in the numbers from 1 to n, since 10 is made of 2 and 5, and there are usually more factors of 2 than 5 in a factorial.
666
667             Example:
668                 >>> count_trailing_zeros(100)
669                 24
670                 >>> count_trailing_zeros(0)
671                 0
672                 >>> count_trailing_zeros(-5)
673                 ValueError: Input must be a non-negative integer.
674
675             if n < 0:
676                 raise ValueError("Input must be a non-negative integer.")
677
678             count = 0
679             power_of_5 = 5
680             while power_of_5 <= n:
681                 count += n // power_of_5
682                 power_of_5 *= 5
683
684
685             return count
686
687             # Example usage
688             n = int(input("Enter a number: ")) # Get user input for the number
689             try:
690                 trailing_zeros = count_trailing_zeros(n) # Count the trailing zeros in n!
691                 print("Trailing zeros:", trailing_zeros) # Print the result
692             except ValueError as e:
693                 print(e) # Print the error message if input is invalid
```

In 655, Col 1 Spaces:4 UTF-8 CRLF [] Python 3.12.3 Python 3.12 (64-bit) Port

The screenshot shows a code editor interface with a sidebar containing icons for file operations like new file, search, and refresh. The main area displays the content of the file `Ass_13.1.py`. The code defines a function `count_trailing_zeros` that calculates the number of trailing zeros in the factorial of a given non-negative integer. It handles edge cases (negative numbers, zero) and raises a `ValueError` if the input is not non-negative. The code uses a loop to iterate through factors of 5 and counts how many times they divide the number. The terminal below shows the execution of the script and its output.

```
ass_9_5.py Ass_10_2.py Ass_13.1.py X library.py library.html String_Ass1.py A
Ass_13.1.py > ...
655     def count_trailing_zeros(n):
656         """
657             Count the number of trailing zeros in the factorial of a given non-negative integer.
658
659             Parameters:
660                 n (int): A non-negative integer for which to count trailing zeros in n!.
661
662             Returns:
663                 int: The number of trailing zeros in n!.
664
665             The number of trailing zeros in n! can be found by counting the number of times 5 is a factor in the numbers from 1 to n, since 10 is made of 2 and 5, and there are usually more factors of 2 than 5 in a factorial.
666
667             Example:
668                 >>> count_trailing_zeros(100)
669                 24
670                 >>> count_trailing_zeros(0)
671                 0
672                 >>> count_trailing_zeros(-5)
673                 ValueError: Input must be a non-negative integer.
674
675             if n < 0:
676                 raise ValueError("Input must be a non-negative integer.")
677
678             count = 0
679             power_of_5 = 5
680             while power_of_5 <= n:
681                 count += n // power_of_5
682                 power_of_5 *= 5
683
684
685             return count
686
687             # Example usage
688             n = int(input("Enter a number: ")) # Get user input for the number
689             try:
690                 trailing_zeros = count_trailing_zeros(n) # Count the trailing zeros in n!
691                 print("Trailing zeros:", trailing_zeros) # Print the result
692             except ValueError as e:
693                 print(e) # Print the error message if input is invalid
```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> ...
Enter a number: 15430900
Trailing zeros: 3857720
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>



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]

```
#Task 13 (Collatz Sequence Generator – Test Case Design)
#Prompt:Generate Collatz sequence until reaching 1. 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.
#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.
#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]
def generate_collatz_sequence(n):
    """
    Generate the Collatz sequence (3n+1 sequence) for a given integer n until it reaches 1.

    Parameters:
    n (int): The starting integer for the Collatz sequence.

    Returns:
    list: A list containing the Collatz sequence starting from n and ending at 1.

    Example:
    >>> generate_collatz_sequence(6)
    [6, 3, 10, 5, 16, 8, 4, 2, 1]
    >>> generate_collatz_sequence(1)
    [1]
    >>> generate_collatz_sequence(-5)
    ValueError: Input must be a positive integer.
    >>> generate_collatz_sequence(27)
    [27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484,
     242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155,
     466, 233, 700, ... ,1] # (long sequence truncated for brevity)
    """
    if n <= 0:
        raise ValueError("Input must be a positive integer.")
    sequence = []
    while n != 1:
        sequence.append(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    sequence.append(1) # Append the final element '1' to the sequence
    return sequence
# Example usage
n = int(input("Enter a number: ")) # Get user input for the starting number
try:
    collatz_sequence = generate_collatz_sequence(n) # Generate the Collatz sequence

```

```
    print(collatz_sequence) # Print the generated sequence
except ValueError as e:
    print(e) # Print the error message if input is invalid
```

```
ass_9_5.py Ass_10.2.py Ass_13.1.py library.py library.html String_Ass1.py Ass_4.5.py Ass_4_4

Ass_13.1.py > ...
def generate_collatz_sequence(n):
    """
    Generate the Collatz sequence (3n+1 sequence) for a given integer n until it reaches 1.

    Parameters:
    n (int): The starting integer for the Collatz sequence.

    Returns:
    List: A List containing the Collatz sequence starting from n and ending at 1.

    Example:
    >>> generate_collatz_sequence(6)
    [6, 3, 10, 5, 16, 8, 4, 2, 1]
    >>> generate_collatz_sequence(1)
    [1]
    >>> generate_collatz_sequence(-5)
    ValueError: Input must be a positive integer.
    >>> generate_collatz_sequence(27)
    [27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484,
     242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155,
     466, 233, 700, ... ,1] # (Long sequence truncated for brevity)
    """
    if n <= 0:
        raise ValueError("Input must be a positive integer")

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



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

```
#Task 14 (Lucas Number Sequence – Test Case Design)
#Prompt: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 → [2, 1, 3, 4, 7]
#Edge: 1 → [2]
#Negative: -5 → Error
#Large: 10 (last element = 76).
#Requirement: Validate correctness with pytest.

def generate_lucas_sequence(n):
    """
    Generate the Lucas number sequence up to n terms.

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

    Returns:
    list: A list containing the Lucas sequence up to n terms.

    Example:
    >>> generate_lucas_sequence(5)
    [2, 1, 3, 4, 7]
    >>> generate_lucas_sequence(1)
    [2]
    >>> generate_lucas_sequence(-5)
    ValueError: Input must be a non-negative integer.
    >>> generate_lucas_sequence(10)
    [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
    """

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

    lucas_sequence = []
    for i in range(n):
        if i == 0:
            lucas_sequence.append(2)
        elif i == 1:
            lucas_sequence.append(1)
        else:
            next_lucas = lucas_sequence[i-1] + lucas_sequence[i-2]
            lucas_sequence.append(next_lucas)

    return lucas_sequence

# Example usage
n = int(input("Enter the number of terms: ")) # Get user input for the
```

```

try:
    lucas_sequence = generate_lucas_sequence(n) # Generate the Lucas sequence
    print(lucas_sequence) # Print the generated sequence
except ValueError as e:
    print(e) # Print the error message if input is invalid

```

```

ass_9.5.py Ass_10.2.py Ass_13.1.py library.py library.html String_Ass1.py A
Ass_13.1.py > ...

769     #Requirement: Validate correctness with pytest.
770     def generate_lucas_sequence(n):
        """
        Generate the Lucas number sequence up to n terms.

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

        Returns:
        List: A List containing the Lucas sequence up to n terms.

        Example:
        >>> generate_lucas_sequence(5)
        [2, 1, 3, 4, 7]
        >>> generate_lucas_sequence(1)
        [2]
        >>> generate_lucas_sequence(-5)
        ValueError: Input must be a non-negative integer.
        >>> generate_lucas_sequence(10)
        [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
        """
        if n < 0:
            raise ValueError("Input must be a non-negative integer.")

        Lucas_sequence = []
        for i in range(n):
            if i == 0:
                Lucas_sequence.append(2)
            elif i == 1:

```

```

File Edit Selection View Go ...
ass_9_5.py Ass_10.2.py Ass_13.1.py X library.py library.html String_Ass1.py

Ass_13.1.py > generate_lucas_sequence
770 def generate_lucas_sequence(n):
788     [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
789     """
790     if n < 0:
791         raise ValueError("Input must be a non-negative integer.")
792
793     lucas_sequence = []
794     for i in range(n):
795         if i == 0:
796             lucas_sequence.append(2)
797         elif i == 1:
798             lucas_sequence.append(1)
799         else:
800             next_lucas = lucas_sequence[i-1] + lucas_sequence[i-2]
801             lucas_sequence.append(next_lucas)
802
803     return lucas_sequence
804 # Example usage
805 n = int(input("Enter the number of terms: ")) # Get user input for the
806 try:
807     lucas_sequence = generate_lucas_sequence(n) # Generate the Lucas sequence
808     print(lucas_sequence) # Print the generated sequence
809 except ValueError as e:
810     print(e) # Print the error message if input is invalid
811

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
-----
Enter the number of terms: 5
[2, 1, 3, 4, 7]
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>

```



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

```

#Task 15 (Vowel & Consonant Counter – Test Case Design)
#Prompt:Count vowels and consonants in string.
#Test Cases to Design:
#Normal: "hello" → (2,3)
#Edge: "" → (0,0)
#Only vowels: "aeiou" → (5,0)
#Large: Long text
#Requirement: Validate correctness with pytest.
def count_vowels_consonants(s):
    """
    Count the number of vowels and consonants in a given string.
    Parameters:
    s (str): The input string to be analyzed.
    Returns:

```

```

tuple: A tuple containing the count of vowels and consonants in the format (vowel_count, consonant_count).

Example:
>>> count_vowels_consonants("hello")
(2, 3)
>>> count_vowels_consonants("")
(0, 0)
>>> count_vowels_consonants("aeiou")
(5, 0)
"""

vowels = 'aeiouAEIOU'
vowel_count = sum(1 for char in s if char in vowels)
consonant_count = sum(1 for char in s if char.isalpha() and char not in vowels)

return vowel_count, consonant_count

# Example usage
input_string = input("Enter a string: ") # Get user input for the string
vowel_count, consonant_count = count_vowels_consonants(input_string) # Count vowels and consonants
print(f"Vowels: {vowel_count}, Consonants: {consonant_count}") # Print the counts

```

```

Ass_13.1.py > ...
820 def count_vowels_consonants(s):
821     """
822         Parameters:
823             s (str): The input string to be analyzed.
824
825         Returns:
826             tuple: A tuple containing the count of vowels and consonants in the format (vowel_count, consonant_count).
827
828         Example:
829             >>> count_vowels_consonants("hello")
830             (2, 3)
831             >>> count_vowels_consonants("")
832             (0, 0)
833             >>> count_vowels_consonants("aeiou")
834             (5, 0)
835             """
836
837             vowels = 'aeiouAEIOU'
838             vowel_count = sum(1 for char in s if char in vowels)
839             consonant_count = sum(1 for char in s if char.isalpha() and char not in vowels)
840
841
842             return vowel_count, consonant_count
843
844 # Example usage
845 input_string = input("Enter a string: ") # Get user input for the string

```

The screenshot shows a code editor interface with a dark theme. The main area displays a Python script named `Ass_13.1.py`. The code defines a function `count_vowels_consonants(s)` that counts the number of vowels and consonants in a given string. It includes docstrings for parameters and returns, and examples of how it works with different inputs. The code editor has tabs for PROBLEMS, OUTPUT, TERMINAL, PORTS, and DEBUG CONSOLE. The TERMINAL tab is active, showing the output of running the script and entering a string to count.

```
1 Ass_13.1.py ... 11 ASSE_13.1.py 12 ASSE_13.1.py 13 library.py 14 library.mym 15 String_ASSE_13.1.py 16 ASSE_13.1.py 17 ASSE_13.1.py
  1 #Large: Long text
  2 #Requirement: Validate correctness with pytest.
  3 def count_vowels_consonants(s):
  4     """
  5         Count the number of vowels and consonants in a given string.
  6
  7     Parameters:
  8         s (str): The input string to be analyzed.
  9
 10    Returns:
 11        tuple: A tuple containing the count of vowels and consonants in the format (vowel_count, consonant_count).
 12
 13    Example:
 14        >>> count_vowels_consonants("hello")
 15        (2, 3)
 16        >>> count_vowels_consonants("")
 17        (0, 0)
 18        >>> count_vowels_consonants("aeiou")
 19        (5, 0)
 20        """
 21
 22        vowels = 'aeiouAEIOU'
 23        vowel_count = sum(1 for char in s if char in vowels)
 24
 25        ...
 26
 27        Enter a string: Hruthika
 28        Vowels: 3, Consonants: 5
 29
 30
 31
 32
 33
 34
 35
 36
 37
 38
 39
```