Assignment – 13.1

2303A51176

Task 1 – Refactoring – Removing Code Duplication

PROMPT :

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))

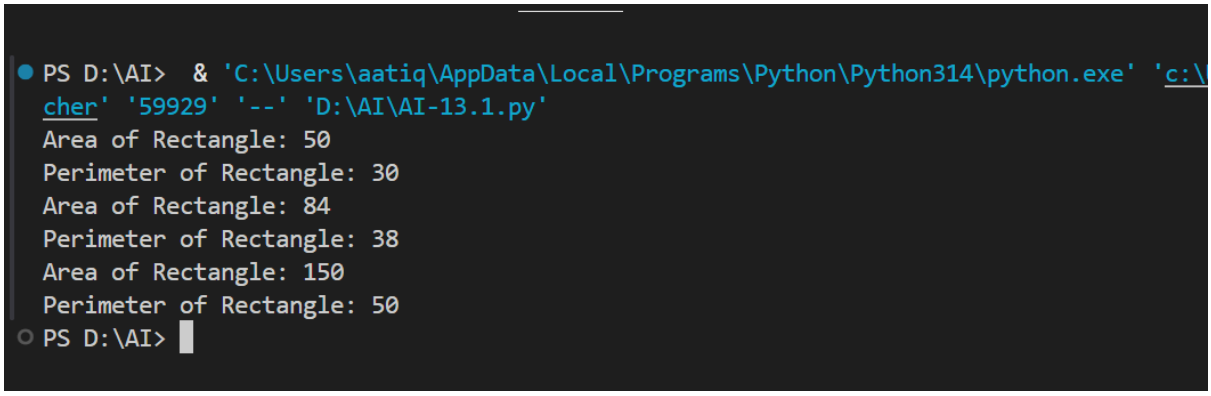
#give o Refactored code with a reusable function and no duplication. and add docstrings to all functions.

CODE :

AI-13.1.py > ...

```python
# 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))
#give o Refactored code with a reusable function and no duplication. and add docstrings to all functions.
def calculate_rectangle_properties(length, width):
    """Calculates and prints the area and perimeter of a rectangle given its length and width."""
    area = length * width  # Calculate area
    perimeter = 2 * (length + width)  # Calculate perimeter
    print(f"Area of Rectangle: {area}")  # Print area
    print(f"Perimeter of Rectangle: {perimeter}")  # Print perimeter
# Call the function with different rectangle dimensions
calculate_rectangle_properties(5, 10)
calculate_rectangle_properties(7, 12)
calculate_rectangle_properties(10, 15)
```

OUTPUT :



JUSTIFICATION :

The legacy code contained repeated calculations for rectangle area and perimeter, which violates the DRY (Don't Repeat Yourself) principle. Refactoring into a reusable function improves maintainability, readability, and scalability because any logic change needs to be updated only once.

Task 2 – Refactoring – Extracting Reusable Functions

PROMPT :

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)

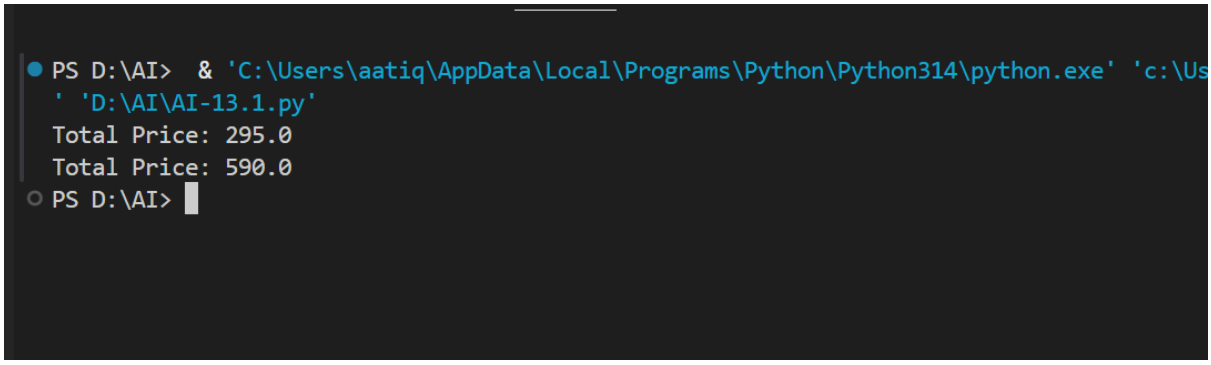
#give o Code with a function calculate_total(price) that can be reused for multiple price inputs.and add docstrings to all functions.

CODE :

```python
# 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)
#give o Code with a function calculate_total(price) that can be reused for multiple price inputs.and add docstrings to all functions.
def calculate_total(price):
    """Calculates and prints the total price including tax for a given price."""
    tax = price * 0.18  # Calculate tax
    total = price + tax  # Calculate total price
    print(f"Total Price: {total}")  # Print total price
# Call the function with different price inputs
calculate_total(250)
calculate_total(500)

```

OUTPUT :



JUSTIFICATION :

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

Task 3 – Refactoring Using Classes and Methods (Eliminating Redundant Conditional Logic)

PROMPT :

#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")

#give a code with class named GradeCalculator with method calculate_grade(self,marks) inside the class and create object of the class and call the method with different marks to get the grade. print the returned grade values.

CODE :

```python
#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")
#give a code with class named GradeCalculator with method calculate_grade(self,marks) inside the class and create object of the class and call the method with
class GradeCalculator:
    """A class to calculate and return the grade based on marks."""

    def calculate_grade(self, marks):
        """Calculates and returns the grade based on the given marks."""
        if marks >= 90:
            return "Grade A"
        elif marks >= 75:
            return "Grade B"
        else:
            return "Grade C"
# Create an object of the GradeCalculator class
grade_calculator = GradeCalculator()
# Call the calculate_grade method with different marks and print the results
print(grade_calculator.calculate_grade(85))  # Should print "Grade A"
print(grade_calculator.calculate_grade(72))  # Should print "Grade B"
print(grade_calculator.calculate_grade(60))  # Should print "Grade C"
```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe
Grade B
Grade C
Grade C
PS D:\AI>
```

JUSTIFICATION :

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

Task 4 – Refactoring – Converting Procedural Code to Functions

PROMPT :

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

square = num * num

print("Square:", square)

#give Modular code using functions like get_input(), calculate_square(), and display_result().

CODE :

```
# num = int(input("Enter number: "))
# square = num * num
# print("Square:", square)
#give Modular code using functions like get_input(), calculate_square(), and display_result().
def get_input():
    """Prompts the user to enter a number and returns it as an integer."""
    return int(input("Enter number: "))
def calculate_square(num):
    """Calculates and returns the square of the given number."""
    return num * num
def display_result(square):
    """Prints the square of the number."""
    print(f"Square: {square}")
# Main function to orchestrate the flow of the program
def main():
    num = get_input()  # Get user input
    square = calculate_square(num)  # Calculate the square of the number
    display_result(square)  # Display the result
# Call the main function to execute the program
if __name__ == "__main__":
    main()
```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe d:/AI/AI-13
Enter number: 6
Square: 36
PS D:\AI>
```

JUSTIFICATION :

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

Task 5 – Refactoring Procedural Code into OOP Design

PROMPT :

#salary = 50000

tax = salary * 0.2

net = salary - tax

print(net)

#give a code with A class like EmployeeSalaryCalculator with methods and attributes.

CODE :

```python
#salary = 50000
# tax = salary * 0.2
# net = salary - tax
# print(net)
#give a code with A class like EmployeeSalaryCalculator with methods and attributes.
class EmployeeSalaryCalculator:
    """A class to calculate the net salary of an employee after tax deduction."""

    def __init__(self, salary):
        """Initializes the EmployeeSalaryCalculator with the given salary."""
        self.salary = salary  # Store the salary as an attribute

    def calculate_tax(self):
        """Calculates and returns the tax based on the salary."""
        return self.salary * 0.2  # Calculate tax as 20% of the salary

    def calculate_net_salary(self):
        """Calculates and returns the net salary after deducting tax."""
        tax = self.calculate_tax()  # Get the calculated tax
        net_salary = self.salary - tax  # Calculate net salary
        return net_salary  # Return the net salary
# Create an object of the EmployeeSalaryCalculator class with a salary of 50000
employee_salary_calculator = EmployeeSalaryCalculator(50000)
# Calculate and print the net salary
net_salary = employee_salary_calculator.calculate_net_salary()  # Get the net salary
print(net_salary)  # Print the net salary

```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\pytho
40000.0
PS D:\AI>
```

JUSTIFICATION :

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

Task 6 – Optimizing Search Logic

PROMPT :

#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")

#give a code with Use of sets or dictionaries with complexity justification.

#Refactor inefficient linear searches using appropriate data structures.

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")
#give a code with Use of sets or dictionaries with complexity justification.
#Refactor inefficient linear searches using appropriate data structures.
users = {"admin", "guest", "editor", "viewer"}  # Use a set for O(1) average time complexity for lookups
name = input("Enter username: ")  # Get user input
if name in users:  # Check if the username is in the set
    print("Access Granted")  # Print access granted if found
else:
    print("Access Denied")  # Print access denied if not found
```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe d:
Enter username: viewer
Access Granted
PS D:\AI>
```

JUSTIFICATION :

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

Task 7 – Refactoring the Library Management System

PROMPT :

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:

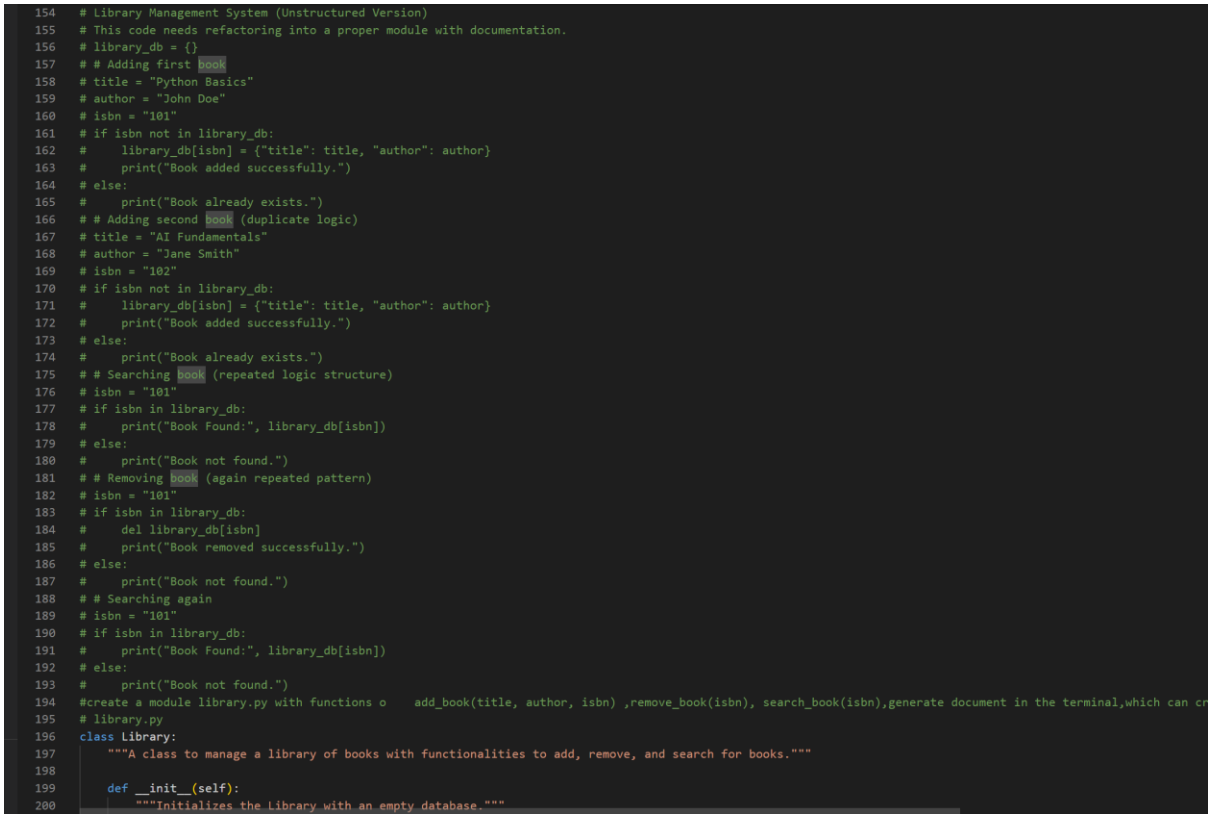
print("Book Found:", library_db[isbn])

else:

print("Book not found.")

#create a module library.py with functions o add_book(title, author, isbn) ,remove_book(isbn), search_book(isbn),generate document in the terminal,which can create html document, which can be able to open in a browser.

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.")
#create a module library.py with functions o    add_book(title, author, isbn) ,remove_book(isbn), search_book(isbn),generate document in the terminal,which can cr
# library.py
class Library:
    """A class to manage a library of books with functionalities to add, remove, and search for books."""

    def __init__(self):
        """Initializes the Library with an empty database."""
```

```python
#create a module library.py with functions o    add_book(title, author, isbn) ,remove_book(isbn), search_book(isbn),generate document in the terminal,which can create
# library.py
class Library:
    """A class to manage a library of books with functionalities to add, remove, and search for books."""

    def __init__(self):
        """Initializes the Library with an empty database."""
        self.library_db = {}  # Initialize an empty dictionary to store book information

    def add_book(self, title, author, isbn):
        """Adds a book to the library database if the ISBN is not already present."""
        if isbn not in self.library_db:
            self.library_db[isbn] = {"title": title, "author": author}  # Add book details to the database
            print("Book added successfully.")
        else:
            print("Book already exists.")

    def remove_book(self, isbn):
        """Removes a book from the library database based on the ISBN."""
        if isbn in self.library_db:
            del self.library_db[isbn]  # Remove the book from the database
            print("Book removed successfully.")
        else:
            print("Book not found.")

    def search_book(self, isbn):
        """Searches for a book in the library database based on the ISBN and returns its details."""
        if isbn in self.library_db:
            return self.library_db[isbn]  # Return book details if found
        else:
            return "Book not found."  # Return message if book is not found
# Example usage of the Library class
if __name__ == "__main__":
    library = Library()  # Create an instance of the Library class
    library.add_book("Python Basics", "John Doe", "101")  # Add a book
    library.add_book("AI Fundamentals", "Jane Smith", "102")  # Add another book
    print(library.search_book("101"))  # Search for a book
    library.remove_book("101")  # Remove a book
    print(library.search_book("101"))  # Search for the removed book
```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.e
Book added successfully.
Book added successfully.
{'title': 'Python Basics', 'author': 'John Doe'}
Book removed successfully.
Book not found.
PS D:\AI>
```

JUSTIFICATION :

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

Task 8– Fibonacci Generator

PROMPT :

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

#refactor into a clean reusable function,add docstrings and test cases,compare refactored and original code.

CODE :

```
# 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
#refactor into a clean reusable function,add docstrings and test cases,compare refactored and original code.
def fibonacci(n):
    """Generates and prints the Fibonacci sequence up to the nth term."""
    a, b = 0, 1  # Initialize the first two Fibonacci numbers
    for i in range(n):
        print(a)  # Print the current Fibonacci number
        a, b = b, a + b  # Update a and b to the next two Fibonacci numbers
# Test cases for the fibonacci function
if __name__ == "__main__":
    print("Fibonacci sequence up to 10 terms:")
    fibonacci(10)  # Test with n=10
    print("\nFibonacci sequence up to 15 terms:")
    fibonacci(15)  # Test with n=15
# Comparison of refactored and original code:
# The original code had a lot of repeated logic for printing the first two Fibonacci numbers and updating
# # the values of a and b. The refactored code uses a single loop to generate the Fibonacci sequence, eliminating redundancy and improving readability
```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe d:/AI/AI-1
Fibonacci sequence up to 15 terms:
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

JUSTIFICATION :

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

Task 9 – Twin Primes Checker

PROMPT :

twin primes 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")

#refactor into is_prime(n) and is_twin_prime(p1,p2),add docstrings and generate a list of twin primes in a given range.

CODE :

```python
# 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")
#refactor into is_prime(n) and is_twin_prime(p1,p2),add docstrings and generate a list of twin primes in a given range.
def is_prime(n):
    """Checks if a number n is prime."""
    if n <= 1:
        return False  # 0 and 1 are not prime numbers
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False  # n is divisible by a number other than 1 and itself
    return True  # n is prime
def is_twin_prime(p1, p2):
    """Checks if two numbers p1 and p2 are twin primes."""
    return is_prime(p1) and is_prime(p2) and abs(p1 - p2) == 2  # Check if both numbers are prime and differ by 2
def generate_twin_primes(start, end):
    """Generates a list of twin primes within a given range."""
    twin_primes = []
    for num in range(start, end):
        if is_twin_prime(num, num + 2):
            twin_primes.append((num, num + 2))  # Add the twin prime pair to the list
    return twin_primes  # Return the list of twin primes
# Example usage of the twin prime functions
if __name__ == "__main__":
    start_range = 10
    end_range = 100
    twin_primes_list = generate_twin_primes(start_range, end_range)  # Generate twin primes in the specified range
    print(f"Twin primes between {start_range} and {end_range}: {twin_primes_list}")  # Print the list of twin primes

```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe d:/AI/AI-13.1.py
Twin primes between 10 and 100: [(11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
PS D:\AI>
```

JUSTIFICATION :

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

Task 10 – Refactoring the Chinese Zodiac Program

PROMPT :

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")

#give a reusable function get_zodiac(year) replace if-elif with a list or dictionary,add docstrings.

CODE :

```python
# 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")
#give a reusable function get_zodiac(year) replace if-elif with a list or dictionary,add docstrings.
def get_zodiac(year):
    """Returns the Chinese zodiac sign for a given year."""
    zodiac_signs = [
        "Monkey", "Rooster", "Dog", "Pig", "Rat", "Ox",
        "Tiger", "Rabbit", "Dragon", "Snake", "Horse", "Goat"
    ]  # List of zodiac signs in order
    return zodiac_signs[year % 12]  # Return the zodiac sign based on the year modulo 12
# Example usage of the get_zodiac function
if __name__ == "__main__":
    year_input = int(input("Enter a year: "))  # Get user input for the year
    zodiac_sign = get_zodiac(year_input)  # Get the zodiac sign for the input year
    print(f"The Chinese zodiac sign for the year {year_input} is: {zodiac_sign}")  # Print the zodiac sign

```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe d:/AI/AI-13.1.py
Enter a year: 6
The Chinese zodiac sign for the year 6 is: Tiger
PS D:\AI>
```

JUSTIFICATION :

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

Task 11 – Refactoring the Harshad (Niven) Number Checker

PROMPT :

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")

#create a resuable function is_harshad(number),accepts an integer,return true if the number is divisible by sum of its digits,return false,add docstrings.

CODE :

```python
# 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")
#create a resuable function is_harshad(number),accepts an integer,return true if the number is divisible by sum of its digits,return false,add docstrings.
def is_harshad(number):
    """Checks if a number is a Harshad number (divisible by the sum of its digits)."""
    temp = number  # Store the original number in a temporary variable
    sum_digits = 0  # Initialize the sum of digits to 0
    while temp > 0:
        digit = temp % 10  # Get the last digit
        sum_digits += digit  # Add the digit to the sum
        temp //= 10  # Remove the last digit
    if sum_digits != 0:  # Check if the sum of digits is not zero to avoid division by zero
        return number % sum_digits == 0  # Return True if number is divisible by sum of digits, else False
    return False  # Return False if sum of digits is zero
# Example usage of the is_harshad function
if __name__ == "__main__":
    number_input = int(input("Enter a number: "))  # Get user input for the number
    result = is_harshad(number_input)  # Check if the number is a Harshad number
    print(result)  # Print the result (True or False)
```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe d:/AI/AI-13.1.p
Enter a number: 10
True
PS D:\AI>
```

JUSTIFICATION :

Mixing input handling with logic and using print-based output limits reuse. Refactoring into a boolean-returning function separates concerns, improves modularity, and supports testing.

Task 12 – Refactoring the Factorial Trailing Zeros Program

PROMPT :

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)

#create a reusable function count_trailing_zeros(n),accepts a non-negative integer,return the number of trailing zeros in n!,add docstrings.

CODE :

```
# 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)
#create a reusable function count_trailing_zeros(n),accepts a non-negative integer,return the number of trailing zeros in n!,add docstrings.
def count_trailing_zeros(n):
    """Returns the number of trailing zeros in n! (factorial of n)."""
    count = 0  # Initialize count of trailing zeros to 0
    power_of_5 = 5  # Start with the first power of 5
    while power_of_5 <= n:  # Loop until the power of 5 is less than or equal to n
        count += n // power_of_5  # Add the number of multiples of the current power of 5
        power_of_5 *= 5  # Move to the next power of 5
    return count  # Return the total count of trailing zeros
# Example usage of the count_trailing_zeros function
if __name__ == "__main__":
    n_input = int(input("Enter a non-negative integer: "))  # Get user input for n
    trailing_zeros = count_trailing_zeros(n_input)  # Calculate the number of trailing zeros in n!
    print(f"Trailing zeros in {n_input}!: {trailing_zeros}")  # Print the result
```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe d:/AI/AI-13.1.py
Enter a non-negative integer: 5
Trailing zeros in 5!: 1
PS D:\AI>
```

JUSTIFICATION :

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

Task 13 – Collatz Sequence Generator – Test Case Design

PROMPT :

#Collatz Sequence Generator – Test Case Design ,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.

CODE :

```python
#Collatz Sequence Generator – Test Case Design ,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 se
def collatz_sequence(n):
    """Generates the Collatz sequence for a given positive integer n."""
    if n <= 0:
        raise ValueError("Input must be a positive integer.")  # Validate input
    sequence = []  # Initialize an empty list to store the sequence
    while n != 1:  # Loop until n becomes 1
        sequence.append(n)  # Add the current value of n to the sequence
        if n % 2 == 0:  # If n is even
            n //= 2  # Divide n by 2
        else:  # If n is odd
            n = 3 * n + 1  # Multiply n by 3 and add 1
    sequence.append(1)  # Append the final value of 1 to the sequence
    return sequence  # Return the generated Collatz sequence
# Example usage of the collatz_sequence function
if __name__ == "__main__":
    print("Collatz sequence for 6:", collatz_sequence(6))  # Normal case
    print("Collatz sequence for 1:", collatz_sequence(1))  # Edge case
    try:
        print("Collatz sequence for -5:", collatz_sequence(-5))  # Negative case (should raise an error)
    except ValueError as e:
        print(e)  # Print the error message
    print("Collatz sequence for 27:", collatz_sequence(27))  # Large case

```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe d:/AI/AI-13.1.py
Collatz sequence for 6: [6, 3, 10, 5, 16, 8, 4, 2, 1]
Collatz sequence for 1: [1]
Input must be a positive integer.
Collatz sequence for 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, 350, 175, 526, 263, 790, 395,
 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734,
 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
PS D:\AI>
```

JUSTIFICATION :

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

Task 14 – Lucas Number Sequence – Test Case Design

PROMPT : #Lucas Number Sequence – Test Case Design,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.

CODE :

```
#Lucas Number Sequence – Test Case Design,Test Cases to Design:,Normal: 5 → [2, 1, 3, 4, 7],Edge: 1 → [2],Negative: -5 → Error,Large: 10 (last element = 76).,Requir
def lucas_sequence(n):
    """Generates the Lucas number sequence up to the nth term."""
    if n <= 0:
        raise ValueError("Input must be a positive integer.")  # Validate input
    sequence = []  # Initialize an empty list to store the sequence
    for i in range(n):
        if i == 0:
            sequence.append(2)  # First term of Lucas sequence
        elif i == 1:
            sequence.append(1)  # Second term of Lucas sequence
        else:
            next_value = sequence[i - 1] + sequence[i - 2]  # Calculate the next term
            sequence.append(next_value)  # Append the next term to the sequence
    return sequence  # Return the generated Lucas sequence
# Example usage of the lucas_sequence function
if __name__ == "__main__":
    print("Lucas sequence for 5:", lucas_sequence(5))  # Normal case
    print("Lucas sequence for 1:", lucas_sequence(1))  # Edge case
    try:
        print("Lucas sequence for -5:", lucas_sequence(-5))  # Negative case (should raise an error)
    except ValueError as e:
        print(e)  # Print the error message
    print("Lucas sequence for 10:", lucas_sequence(10))  # Large case
```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe d:/AI/AI-13.1.py
Lucas sequence for 5: [2, 1, 3, 4, 7]
Lucas sequence for 1: [2]
Input must be a positive integer.
Lucas sequence for 10: [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
PS D:\AI>
```

JUSTIFICATION :

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

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

PROMPT : #Vowel & Consonant Counter – Test Case Design,Test Cases to Design:Normal: "hello" → (2,3),Edge: "" → (0,0),Only vowels: "aeiou" → (5,0).

CODE :

```python
#Vowel & Consonant Counter – Test Case Design,Test Cases to Design:Normal: "hello" → (2,3),Edge: "" → (0,0),Only vowels: "aeiou" → (5,0).
#Only consonants: "bcdfg" → (0,5),Requirement: Validate correctness with pytest.
def count_vowels_consonants(s):
    """Counts the number of vowels and consonants in a given string s."""
    vowels = "aeiouAEIOU"  # Define a string of vowels (both lowercase and uppercase)
    vowel_count = 0  # Initialize vowel count
    consonant_count = 0  # Initialize consonant count
    for char in s:  # Iterate through each character in the string
        if char.isalpha():  # Check if the character is an alphabet
            if char in vowels:  # Check if the character is a vowel
                vowel_count += 1  # Increment vowel count
            else:
                consonant_count += 1  # Increment consonant count
    return vowel_count, consonant_count  # Return the counts as a tuple
# Example usage of the count_vowels_consonants function
if __name__ == "__main__":
    print("Vowels and consonants in 'hello':", count_vowels_consonants("hello"))  # Normal case
    print("Vowels and consonants in '':", count_vowels_consonants(""))  # Edge case
    print("Vowels and consonants in 'aeiou':", count_vowels_consonants("aeiou"))  # Only vowels case
    print("Vowels and consonants in 'bcdfg':", count_vowels_consonants("bcdfg"))  # Only consonants case


```

OUTPUT :

```
PS D:\AI> & C:\Users\aatiq\AppData\Local\Programs\Python\Python314\python.exe d:/AI/AI-13.1.py
Vowels and consonants in 'hello': (2, 3)
Vowels and consonants in '': (0, 0)
Vowels and consonants in 'aeiou': (5, 0)
Vowels and consonants in 'bcdfg': (0, 5)
PS D:\AI>
```

JUSTIFICATION :

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