Lab Assignment-13

Name: P.Chandra Vardhan Reddy

Hall ticket no: 2303A51034

Batch-01

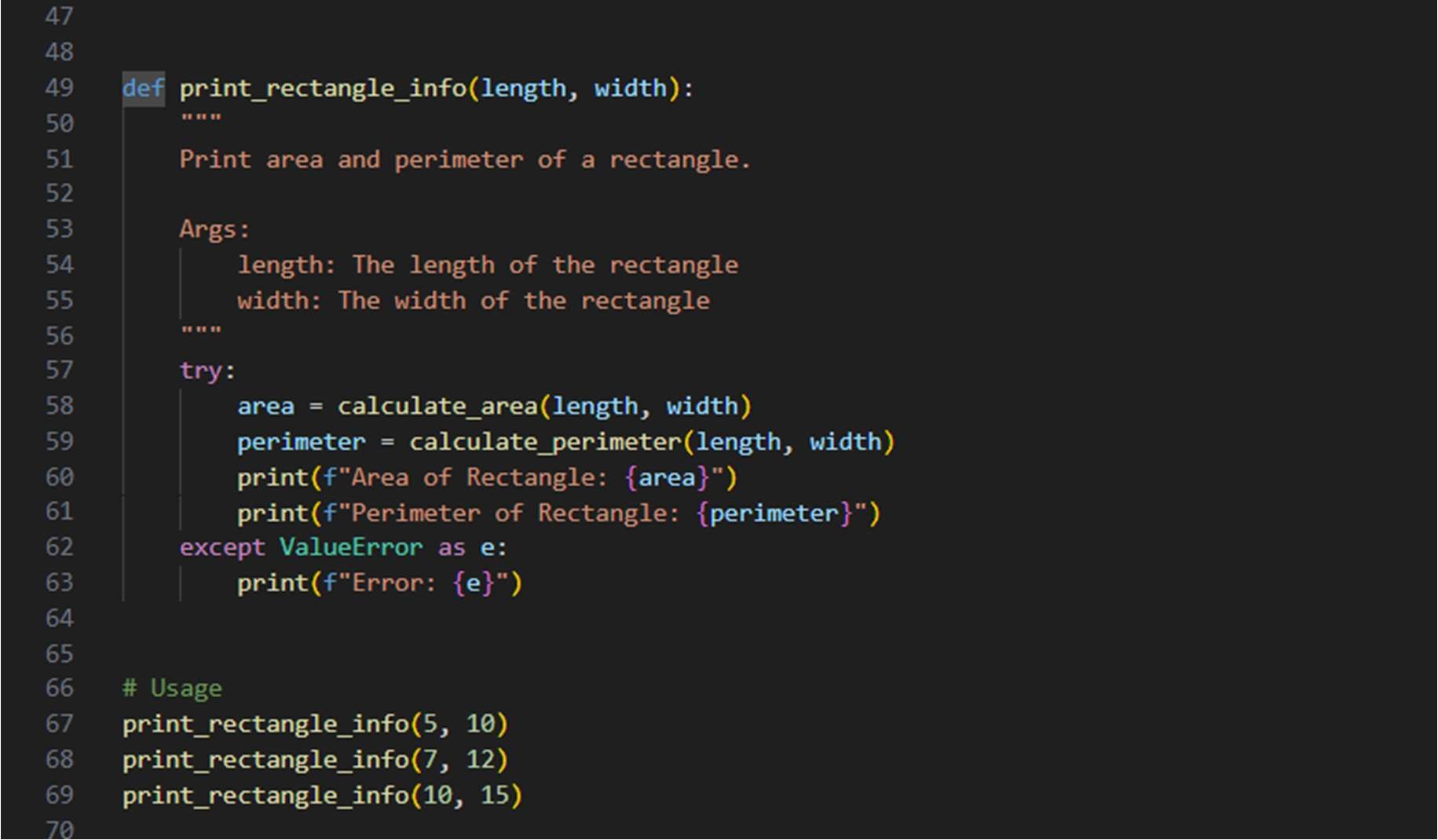
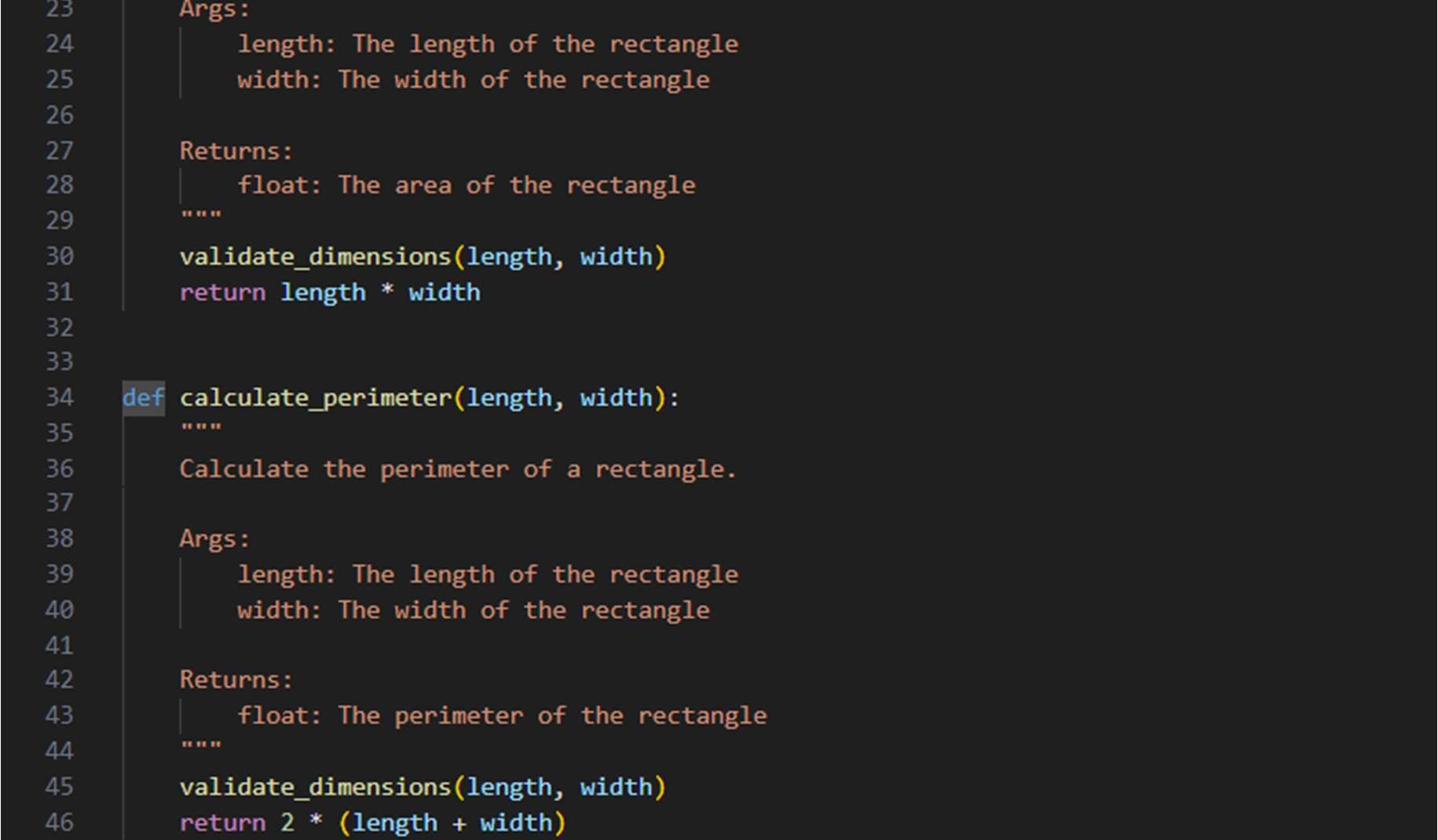
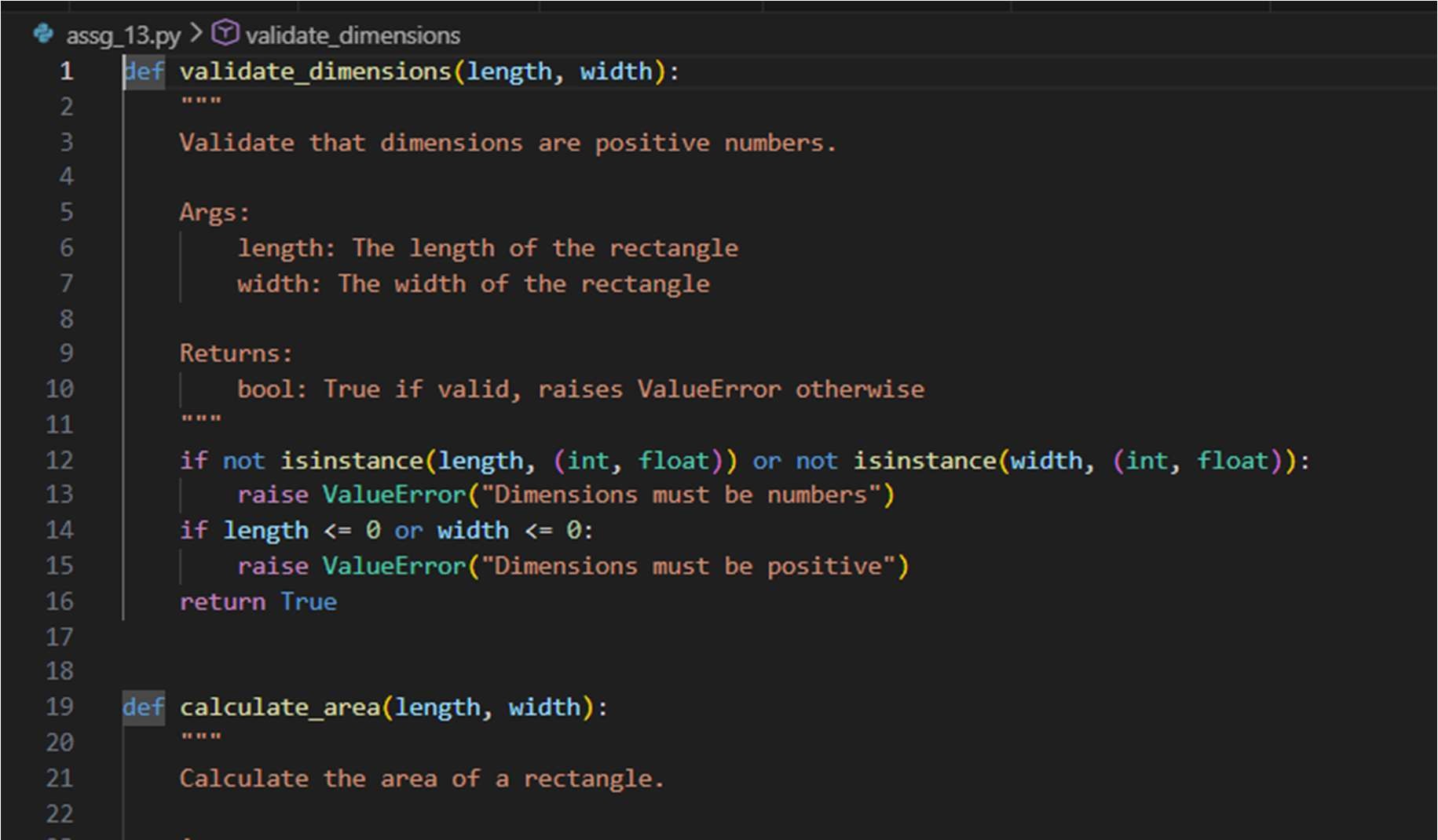
Task Descrip on #1 (Refactoring – Removing Code Duplica on)

* Task: Use AI to refactor a given Python script that contains mul ple repeated code blocks.
* Instruc ons: o Prompt AI to iden fy duplicate logic and replace it with func ons or classes. o Ensure the refactored code maintains the same output. o Add docstrings to all func ons.
* Sample Legacy Code:

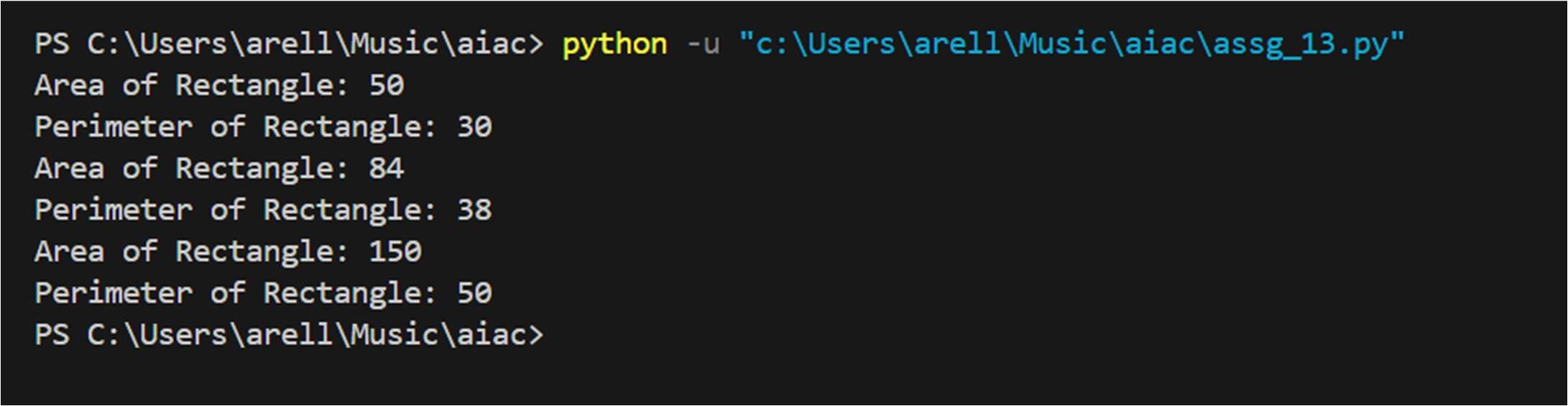
# Legacy script with repeated logic print("Area of Rectangle:", 5 \* 10) print("Perimeter of Rectangle:", 2 \* (5 + 10)) print("Area of Rectangle:", 7 \* 12) print("Perimeter of Rectangle:", 2 \* (7 + 12)) print("Area of Rectangle:", 10 \* 15) print("Perimeter of Rectangle:", 2 \* (10 + 15))

* Expected Output:
  + Refactored code with a reusable func on and no duplica on.
  + Well documented code

screenshots:



Output:



Task Descrip on #2 (Refactoring – Extrac ng Reusable Func ons)

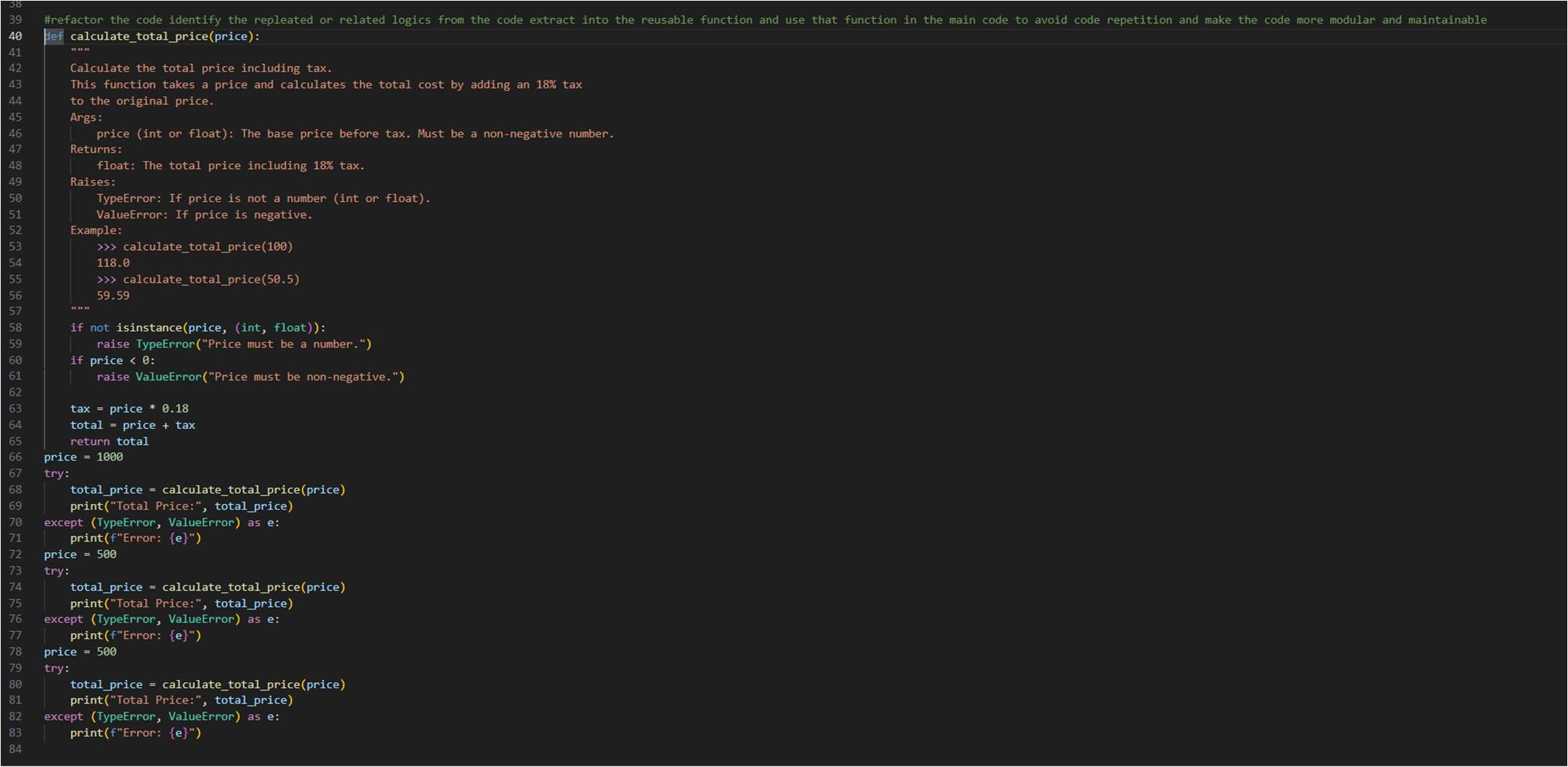
* Task: Use AI to refactor a legacy script where mul ple calcula ons are embedded directly inside the main code block.
* Instruc ons: o Iden fy repeated or related logic and extract it into reusable func ons.

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

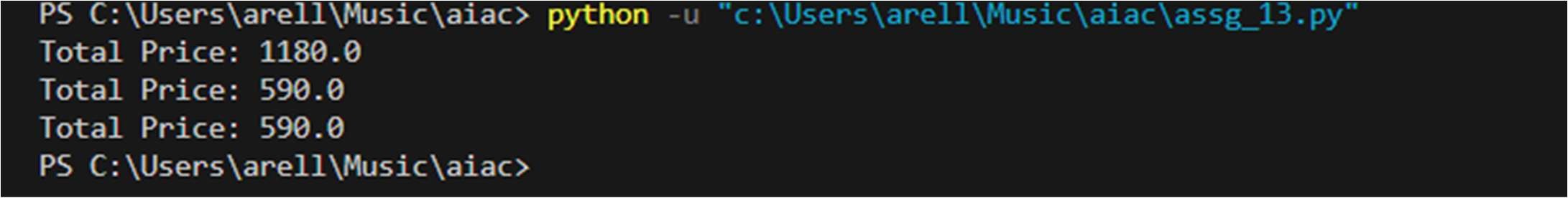
* Sample Legacy Code:

# Legacy script with inline repeated logic price = 250 tax = price \* 0.18 total = price + tax print("Total Price:", total) price = 500 tax = price \* 0.18 total = price + tax print("Total Price:", total)

* Expected Output: o Code with a func on calculate\_total(price) that can be reused for mul ple price inputs. o Well documented code screenshots:



Output:



Task Descrip on #3: Refactoring Using Classes and Methods (Elimina ng

Redundant Condi onal Logic)

Refactor a Python script that contains repeated if–elif–else grading logic by implemen ng a structured, object-oriented solu on using a class and a method.

Problem Statement

The given script contains duplicated condi onal 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 Implementa on Requirements

1. Class Name: GradeCalculator
2. Method Name: calculate\_grade(self, marks)
3. The method must:
   * Accept marks as a parameter. o Return the corresponding grade as a string.
   * The grading logic must strictly follow the condi ons 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.

1. Include proper docstrings for:
   * The class o The method (with parameter and return descrip ons) 5. The method must be reusable and called mul ple mes without rewri ng condi onal logic.

* Given code: marks = 85 if marks >= 90: print("Grade A") elif marks >= 75: print("Grade B") else:

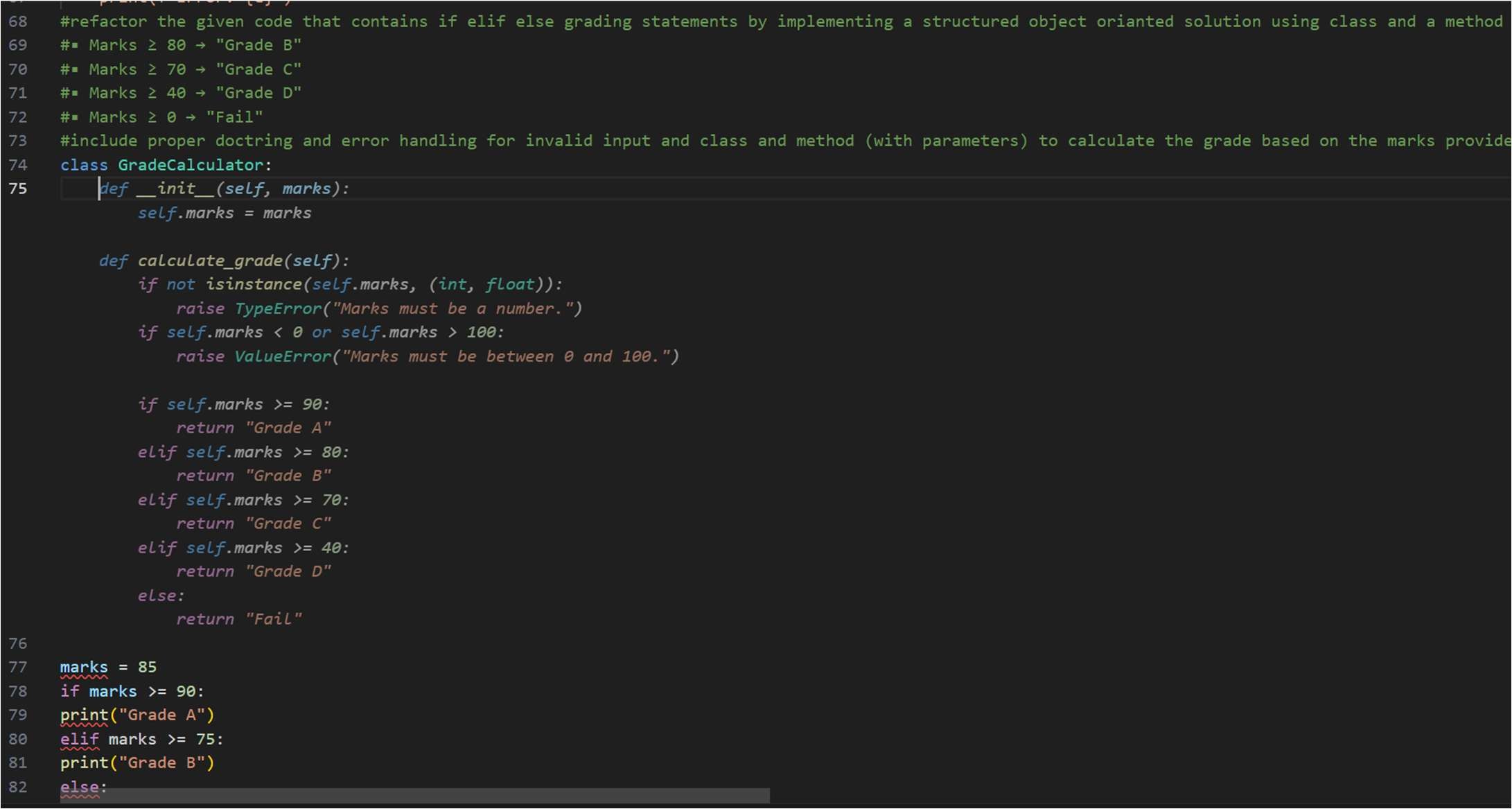
print("Grade C") marks = 72 if marks >= 90: print("Grade A") elif marks >= 75: print("Grade B") else:

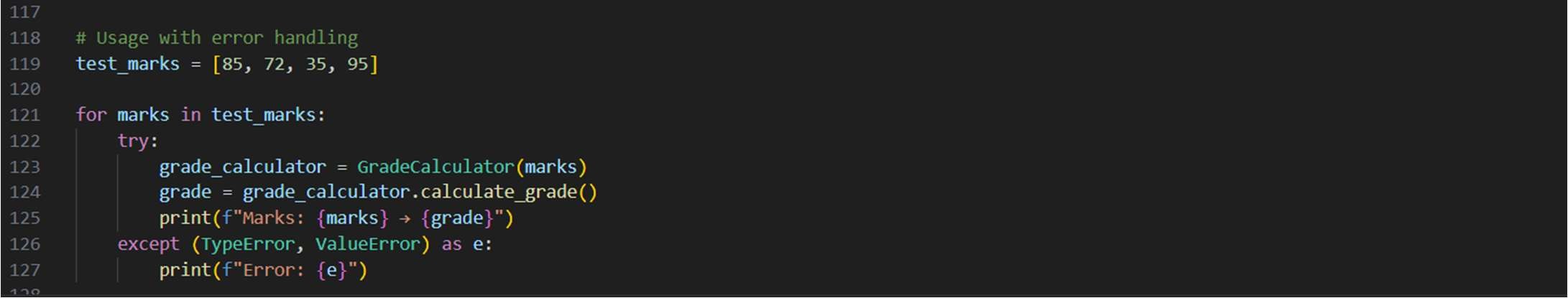
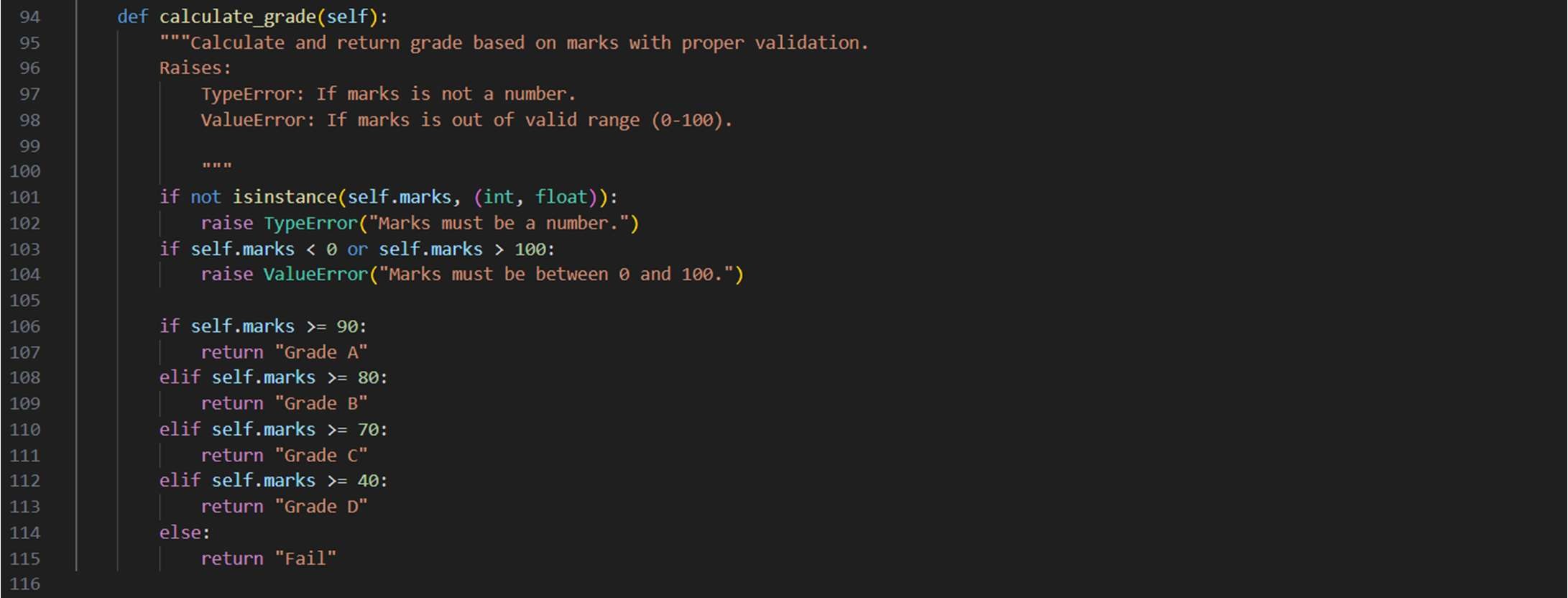
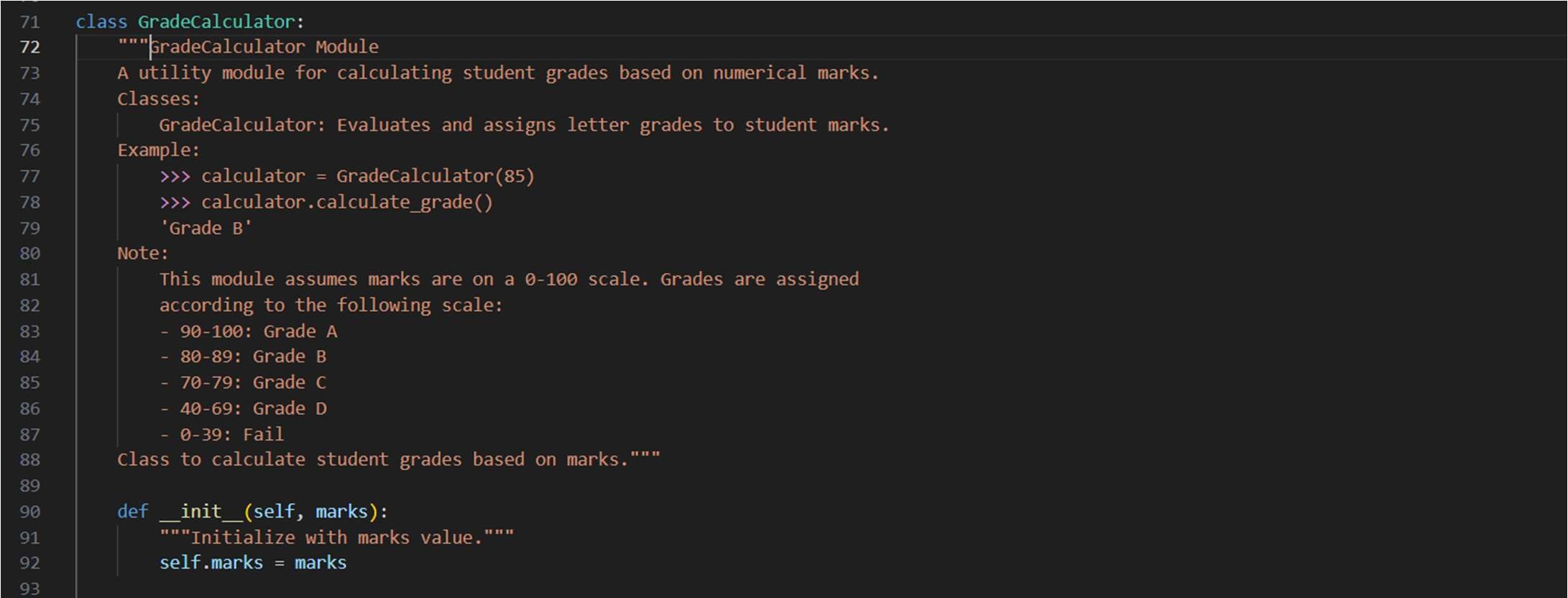
print("Grade C")

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.

Screenshots:





Output:



Task Descrip on #4 (Refactoring – Conver ng Procedural Code to

Func ons)

* Task: Use AI to refactor procedural input–processing logic into func ons.

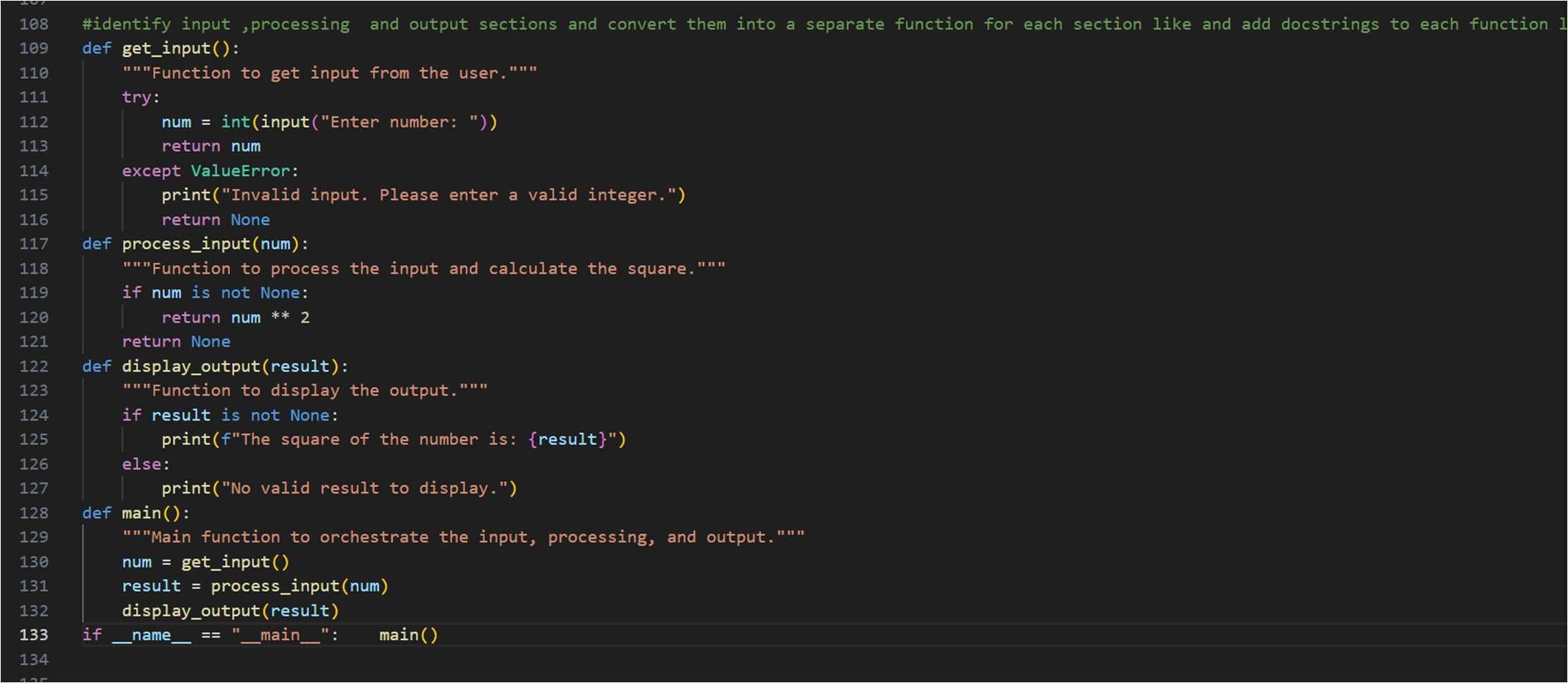
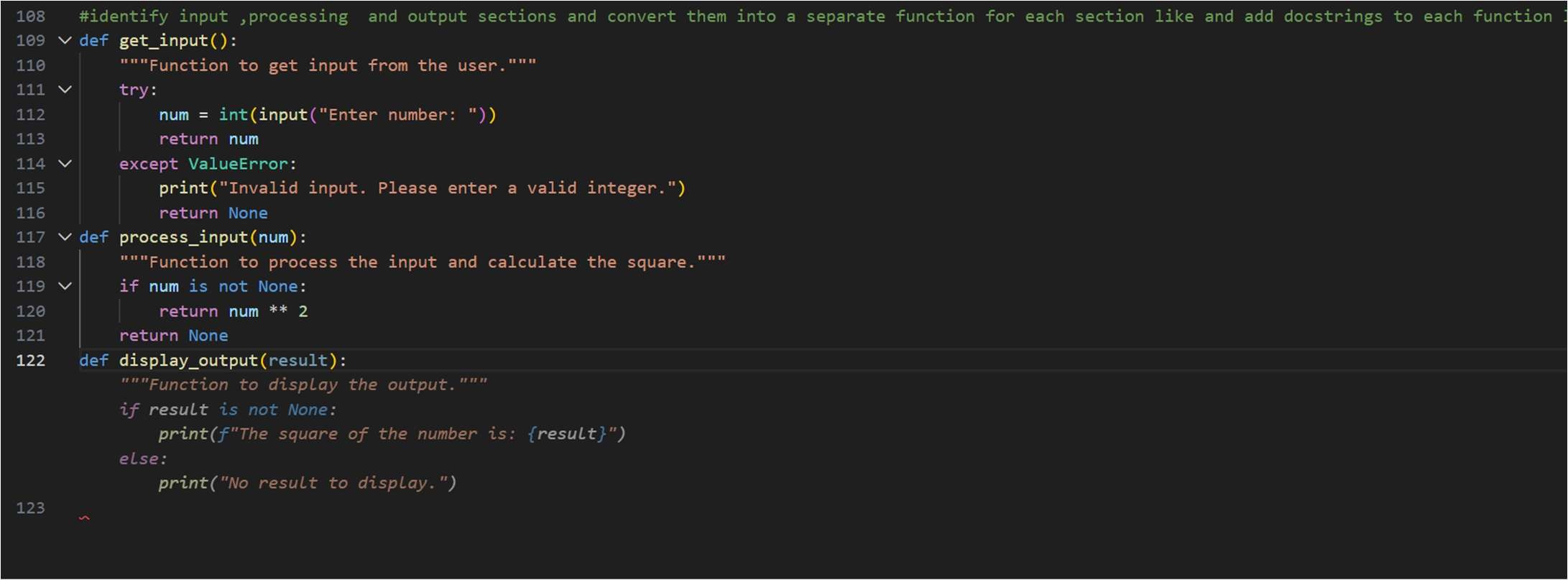
Instruc ons:

* Iden fy input, processing, and output sec ons.
* Convert each into a separate func on.
* Improve code readability without changing behavior.
* Sample Legacy Code:

num = int(input("Enter number: ")) square = num \* num print("Square:", square)

* Expected Output:
  + Modular code using func ons like get\_input(), calculate\_square(), and display\_result().

Screenshots:



Output:



Task 5 (Refactoring Procedural Code into OOP Design)

* Task: Use AI to refactor procedural code into a class-based design.

Focus Areas:

* + Object-Oriented principles o Encapsula on

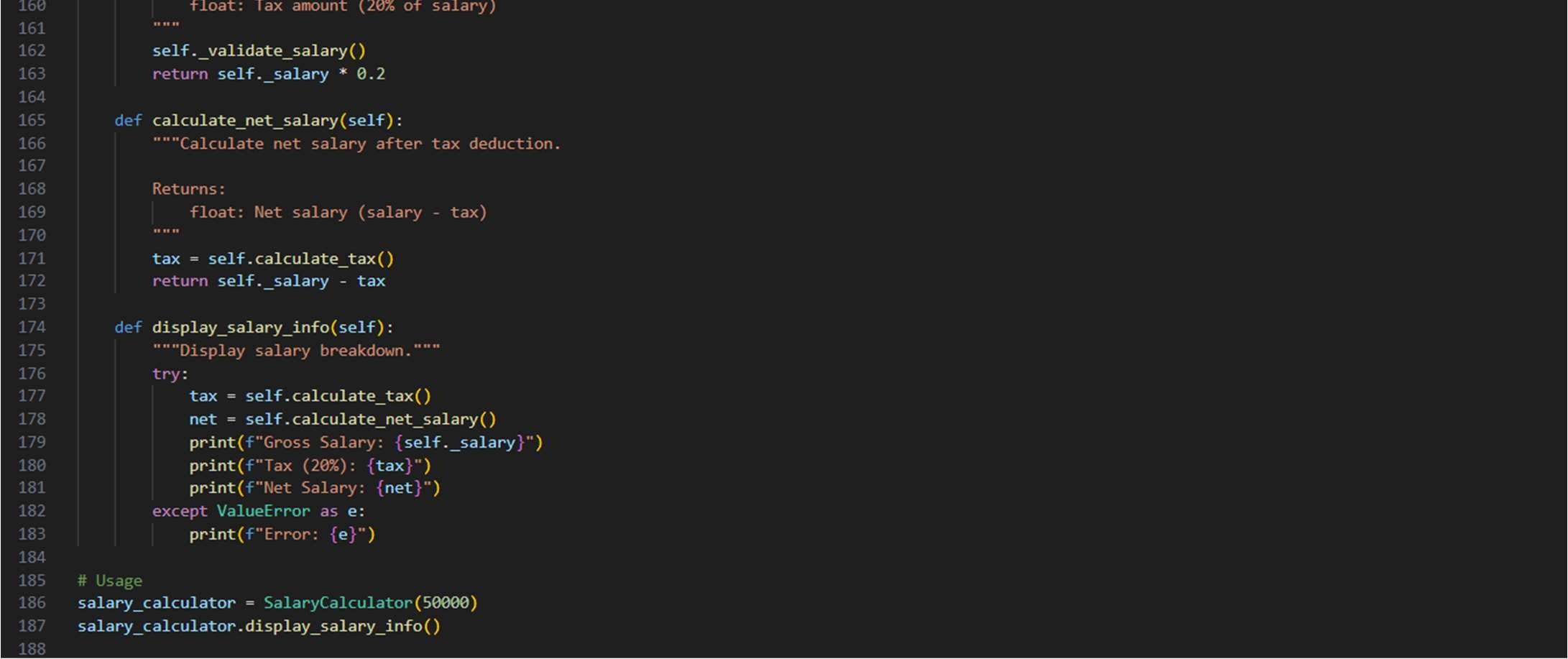
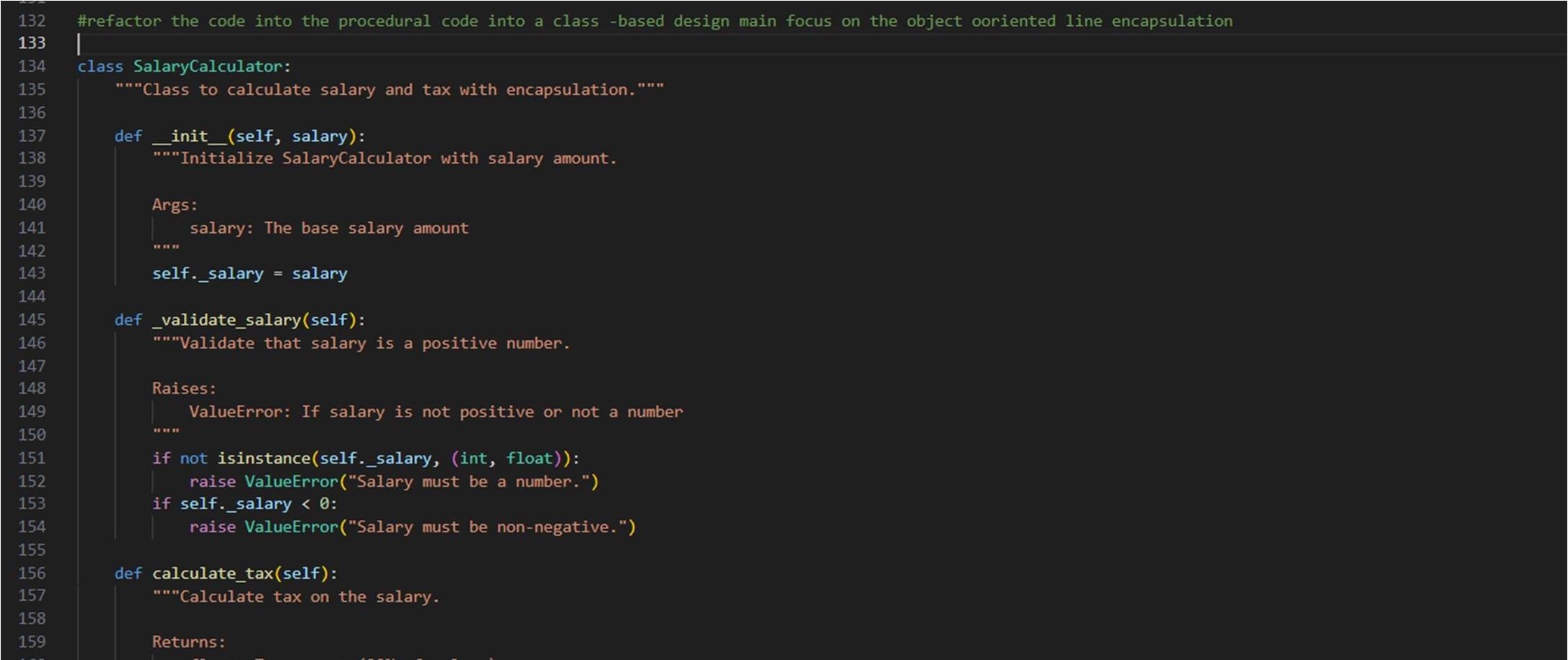
Legacy Code:

salary = 50000 tax = salary \* 0.2 net = salary - tax print(net)

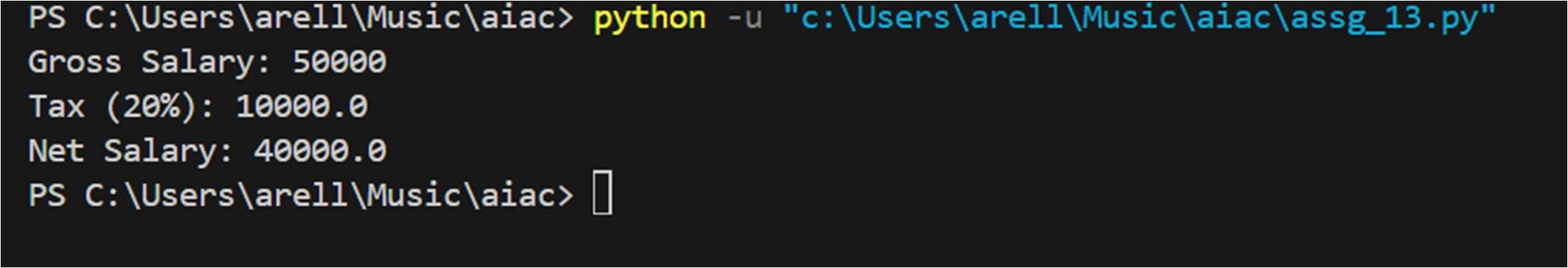
Expected Outcome:

A class like EmployeeSalaryCalculator with methods and a ributes.

Screenshots:



Output:



Task 6 (Op mizing Search Logic)

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

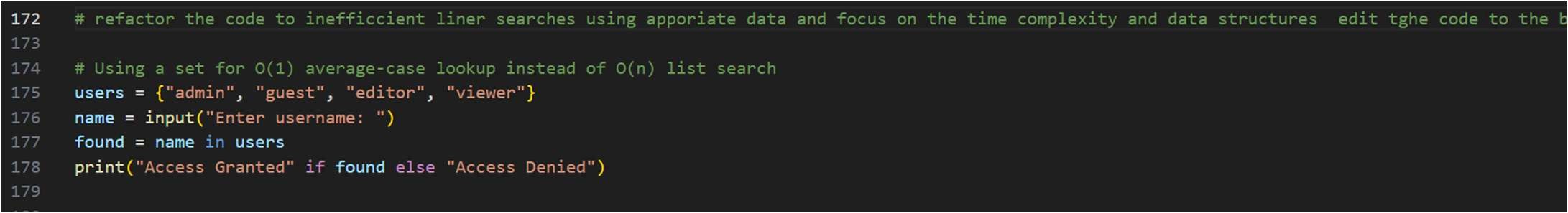
Legacy Code:

users = ["admin", "guest", "editor", "viewer"]

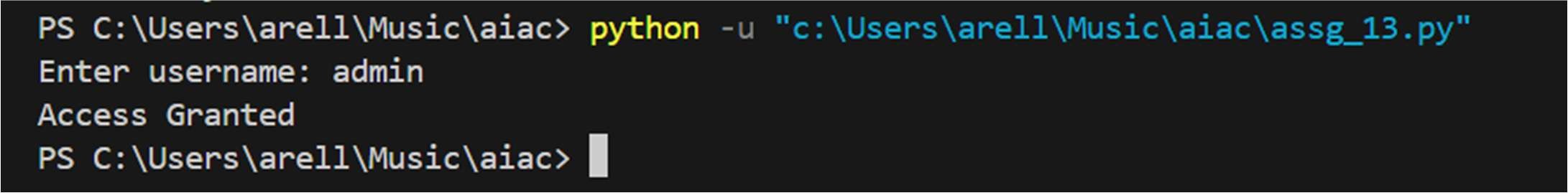
name = input("Enter username: ") found = False for u in users: if u == name: found = True print("Access Granted" if found else "Access Denied")

Expected Outcome:

* + Use of sets or dic onaries with complexity jus fica on screenshots:



Output:



Task 7 – Refactoring the Library Management System

Problem Statement

You are provided with a poorly structured Library Management script that:

* Contains repeated condi onal logic
* Does not use reusable func ons
* Lacks documenta on
* Uses print-based procedural execu on
* Does not follow modular programming principles Your task is to refactor the code into a proper format

1. Create a module library.py with func ons:

o add\_book( tle, author, isbn) o remove\_book(isbn) o search\_book(isbn)

1. Insert triple quotes under each func on and let Copilot complete the

docstrings.

1. Generate documenta on in the terminal.
2. Export the documenta on in HTML format.
3. Open the file in a browser.

Given Code

# Library Management System (Unstructured Version)

# This code needs refactoring into a proper module with documenta on.

library\_db = {}

# Adding first book tle = "Python Basics" author = "John Doe" isbn = "101" if isbn not in library\_db:

library\_db[isbn] = {" tle": tle, "author": author} print("Book added successfully.") else:

print("Book already exists.")

# Adding second book (duplicate logic) tle = "AI Fundamentals" author = "Jane Smith" isbn = "102" if isbn not in library\_db:

library\_db[isbn] = {" tle": tle, "author": author} print("Book added successfully.") else:

print("Book already exists.")

# Searching book (repeated logic structure) isbn = "101" if isbn in library\_db:

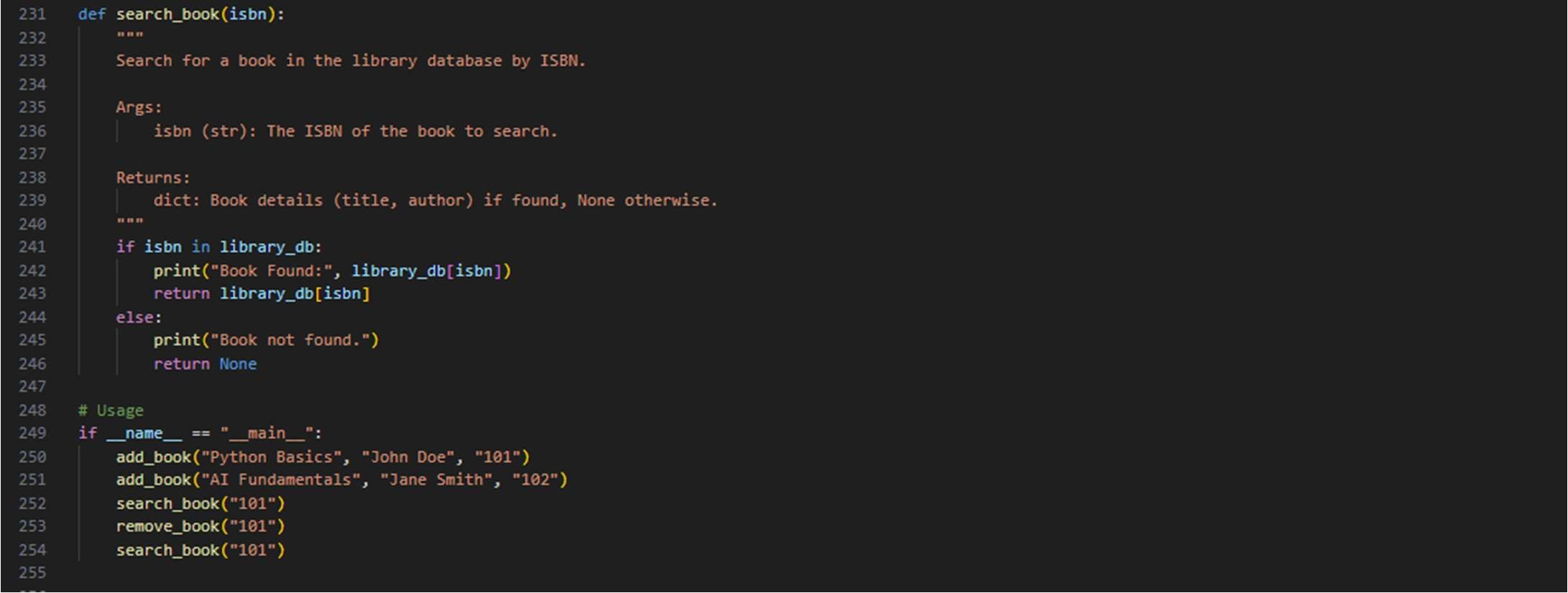
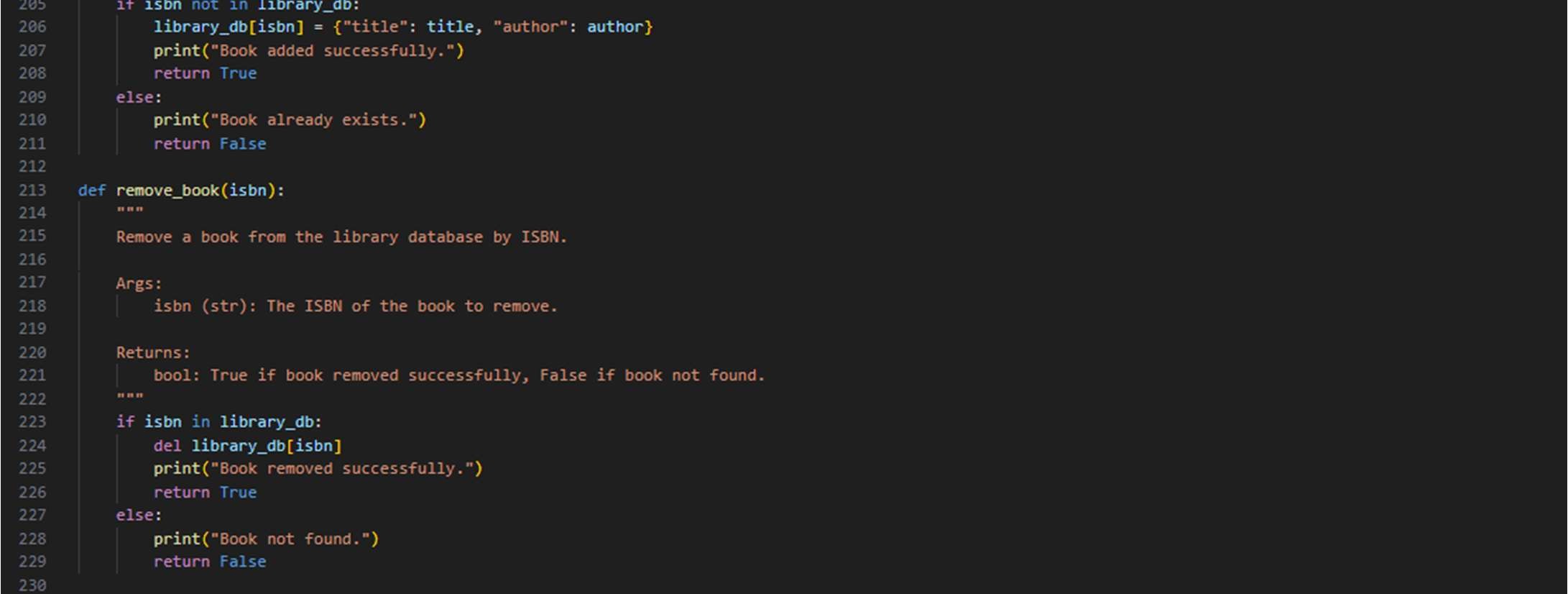
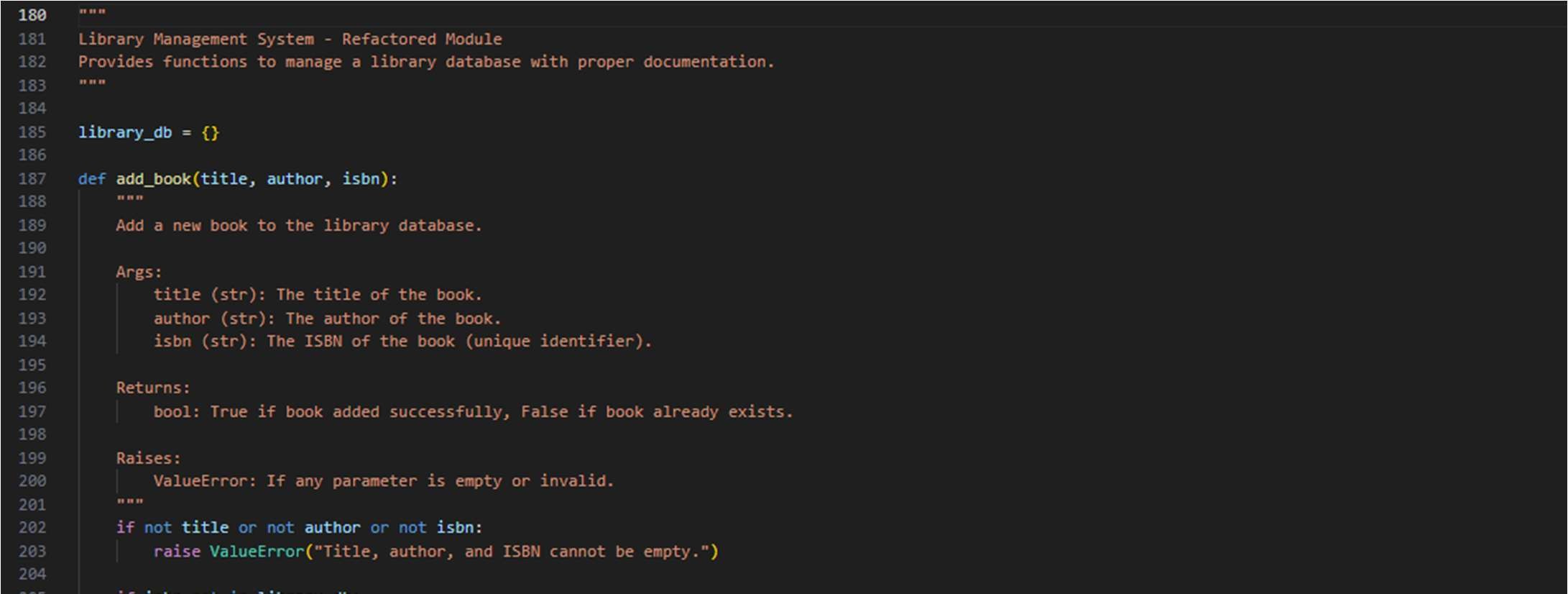
print("Book Found:", library\_db[isbn]) else:

print("Book not found.") # Removing book (again repeated pa ern) 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:

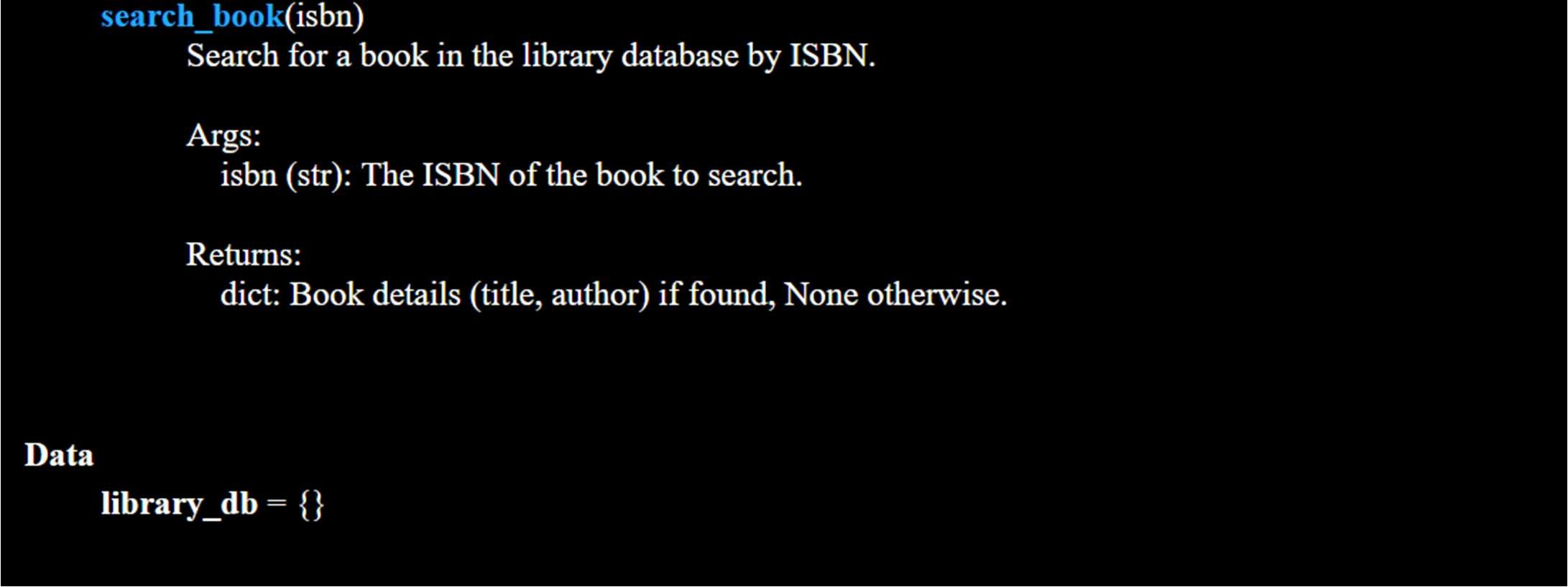
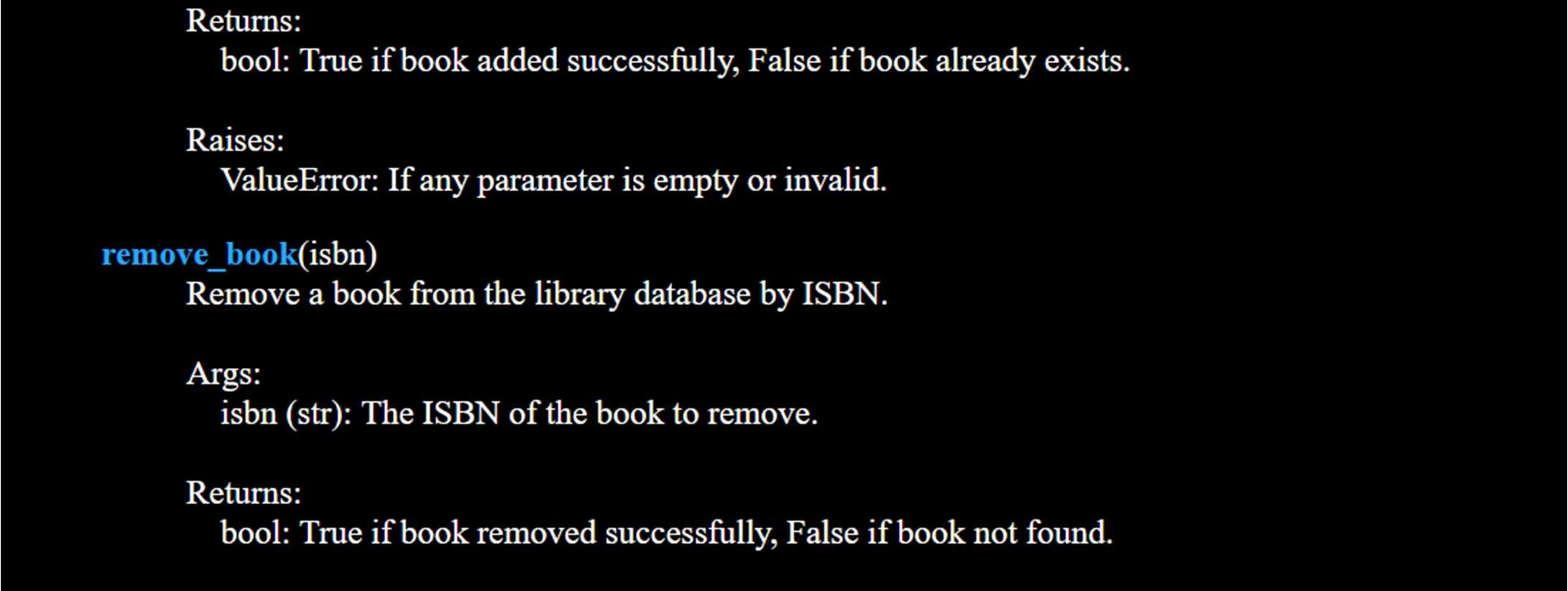
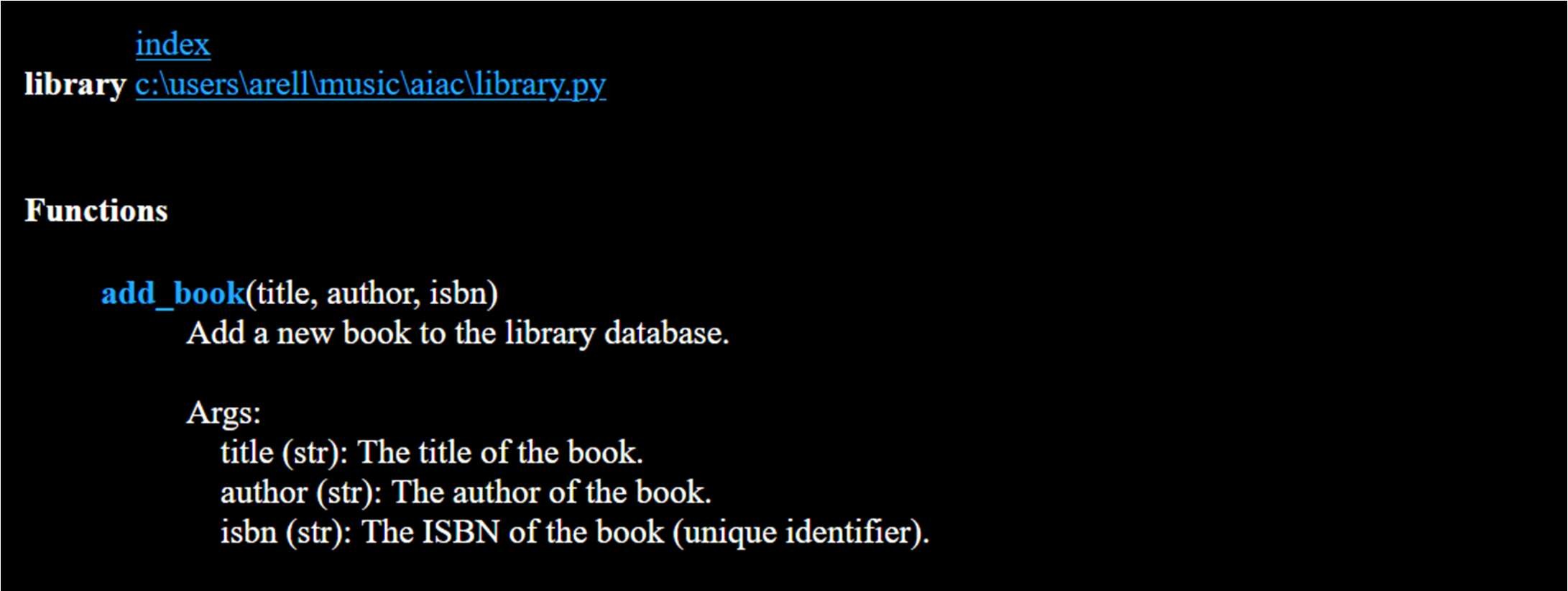
print("Book Found:", library\_db[isbn]) else:

print("Book not found.") screenshots:



Output:





Task 8– Fibonacci Generator

Write a program to generate Fibonacci series up to n.

The ini al code has:

* Global variables.
* Inefficient loop.
* No func ons or modularity.

Task for Students:

* Refactor into a clean reusable func on (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 screenshots:



Output:



Task 9 – Twin Primes Checker

Twin primes are pairs of primes that differ by 2 (e.g., 11 and 13, 17 and 19).

The ini al code has:

* Inefficient prime checking.
* No func ons.
* Hardcoded inputs.

Task for Students:

* Refactor into is\_prime(n) and is\_twin\_prime(p1, p2).
* Add docstrings and op mize.
* 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

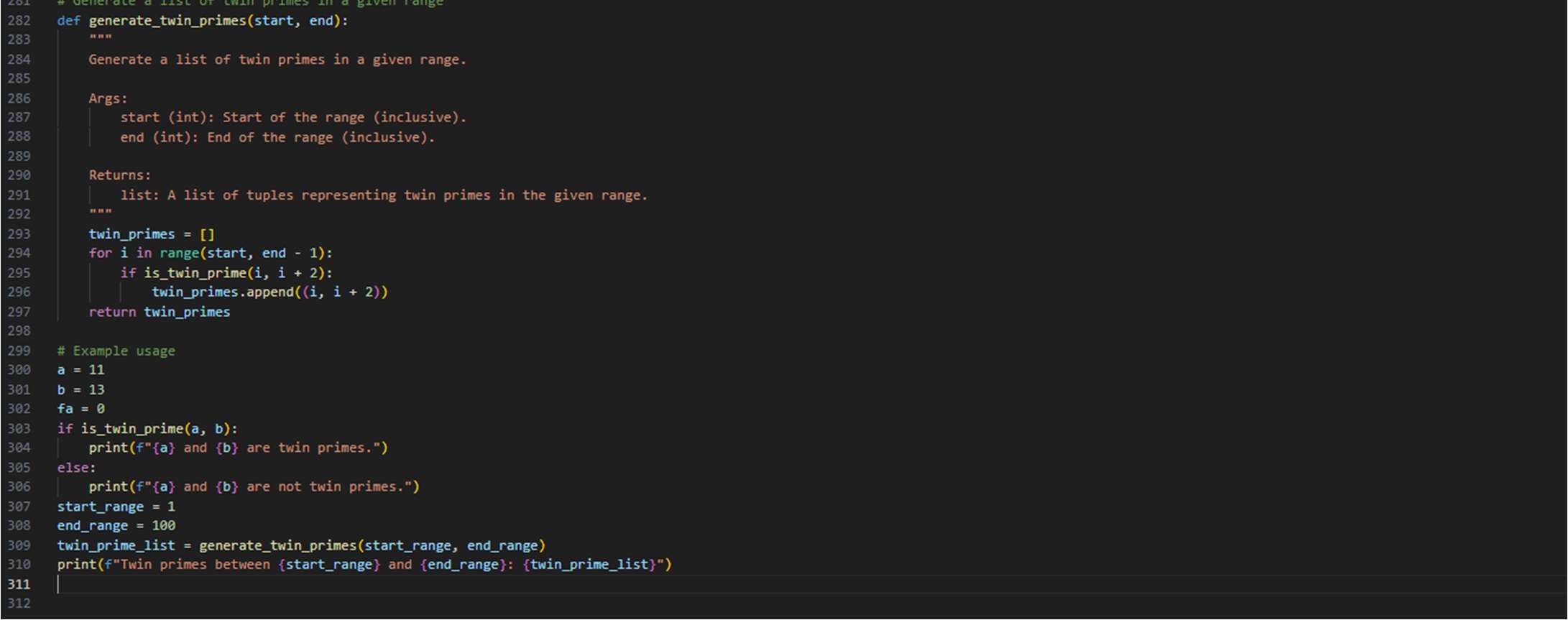
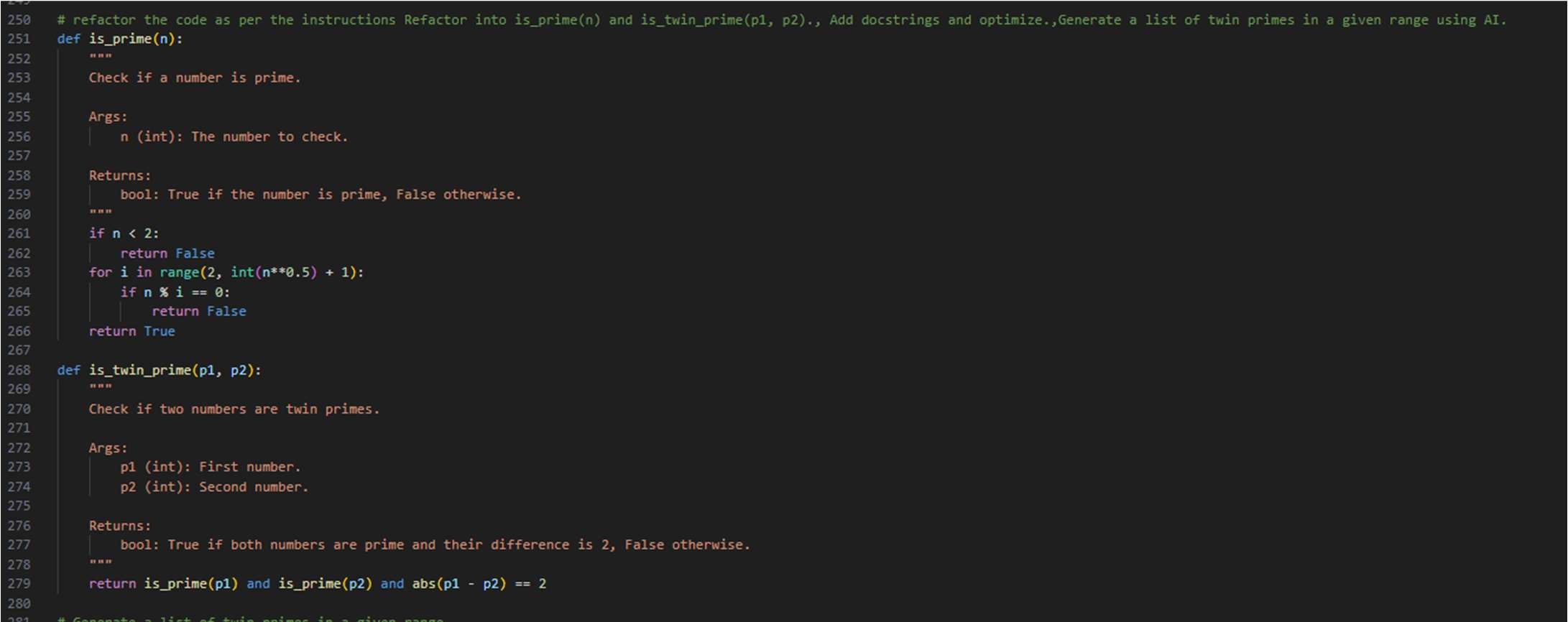
=0 for i in range(2,b):

if b%i==0:

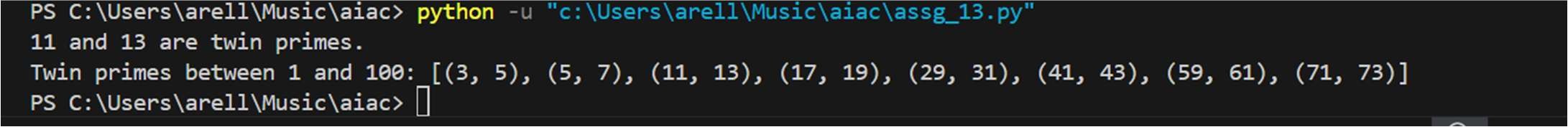
=1 if fa==0 and ==0 and abs(a-b)==2:

print("Twin Primes")

else: print("Not Twin Primes") screenshots:



Output:



Task 10 – Refactoring the Chinese Zodiac Program

Objec ve

Refactor the given poorly structured Python script into a clean, modular, and reusable implementa on.

The current program reads a year from the user and prints the corresponding Chinese Zodiac sign. However, the implementa on contains repe ve condi onal 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")

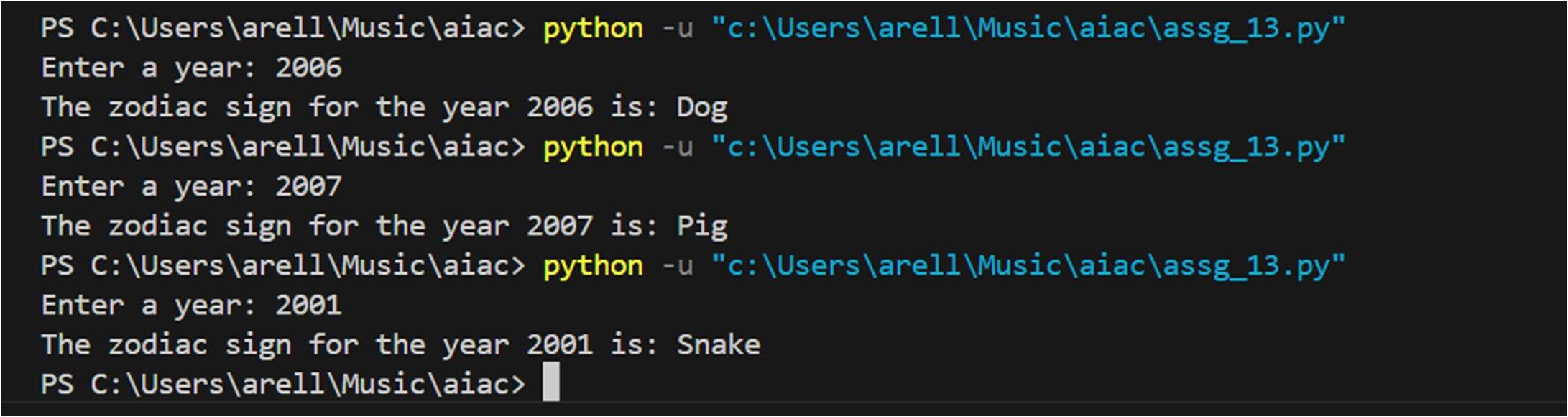
You must:

1. Create a reusable func on: get\_zodiac(year)
2. Replace the if-elif chain with a cleaner structure (e.g., list or dic onary).
3. Add proper docstrings.
4. Separate input handling from logic.
5. Improve readability and maintainability.
6. Ensure output remains correct.

Screenshots:



Output:



Task 11 – Refactoring the Harshad (Niven) Number Checker

Refactor the given poorly structured Python script into a clean, modular, and reusable implementa on.

A Harshad (Niven) number is a number that is divisible by the sum of its digits.

For example:

* 18 → 1 + 8 = 9 → 18 ÷ 9 = 2 ô (Harshad Number)
* 19 → 1 + 9 = 10 → 19 ÷ 10 ≠ integer ô (Not Harshad)

Problem Statement

The current implementa on:

* Mixes logic and input handling
* Uses redundant variables
* Does not use reusable func ons properly
* Returns print statements instead of boolean values
* Lacks documenta on

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")

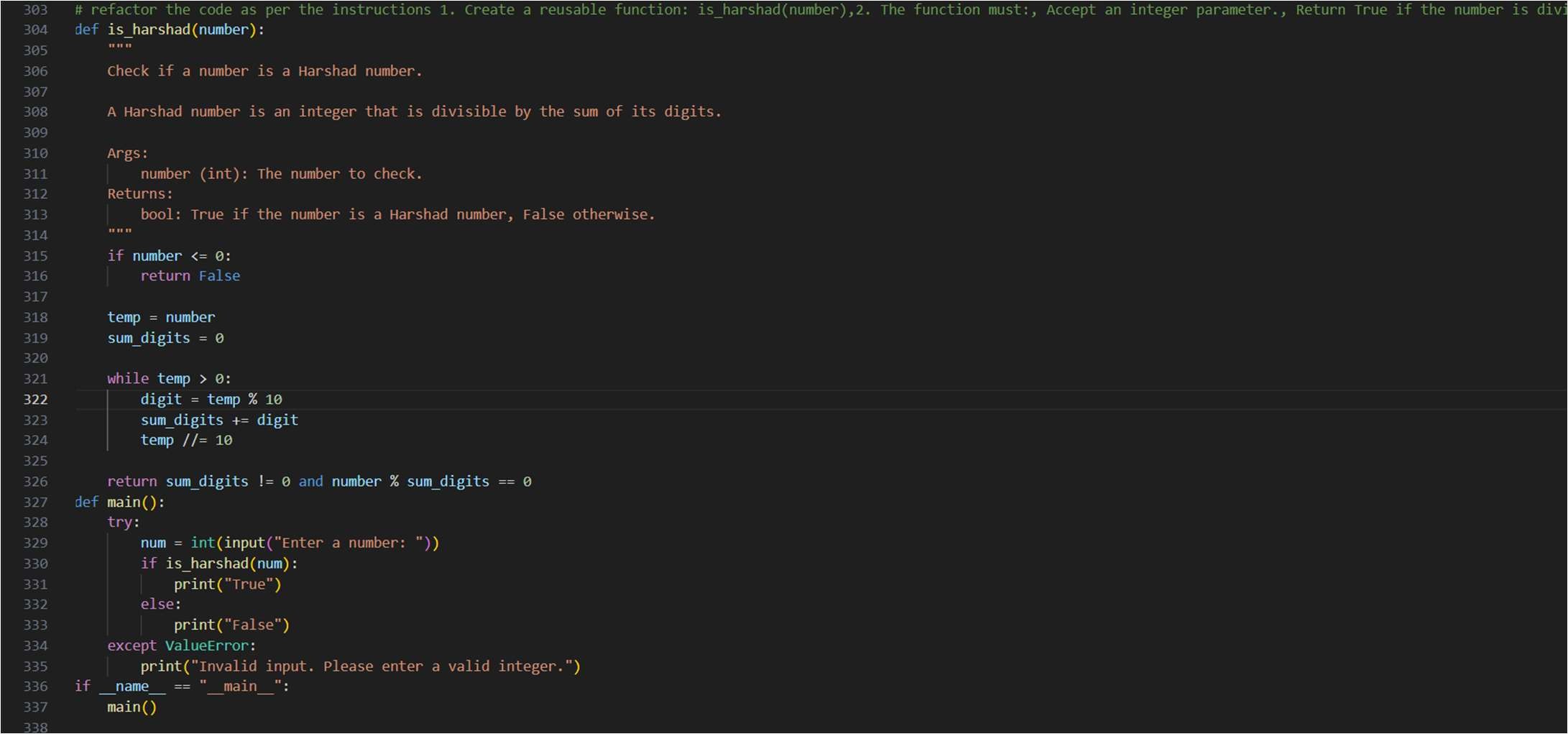
You must:

1. Create a reusable func on: is\_harshad(number)
2. The func on must:

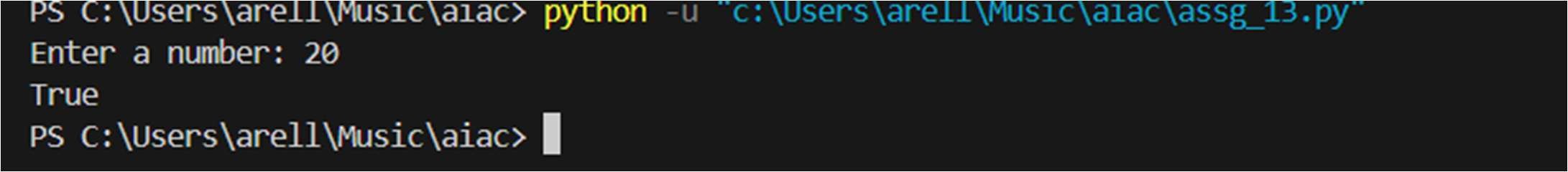
o Accept an integer parameter. o Return True if the number is divisible by the sum of its digits. o Return False otherwise.

1. Separate user input from core logic.
2. Add proper docstrings.
3. Improve readability and maintainability.

Screenshots:



Output:



Task 12 – Refactoring the Factorial Trailing Zeros Program

Refactor the given poorly structured Python script into a clean, modular, and efficient implementa on.

The program calculates the number of trailing zeros in n! (factorial of n).

Problem Statement

The current implementa on:

* 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 documenta on

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)

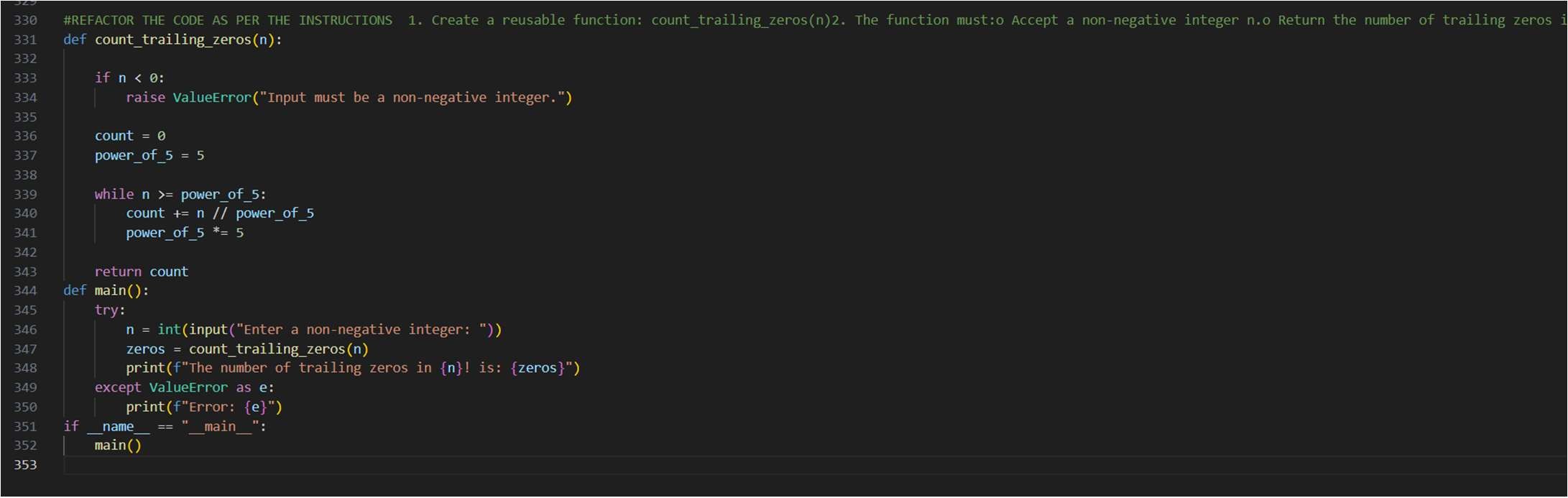
You must:

1. Create a reusable func on: count\_trailing\_zeros(n)
2. The func on must:

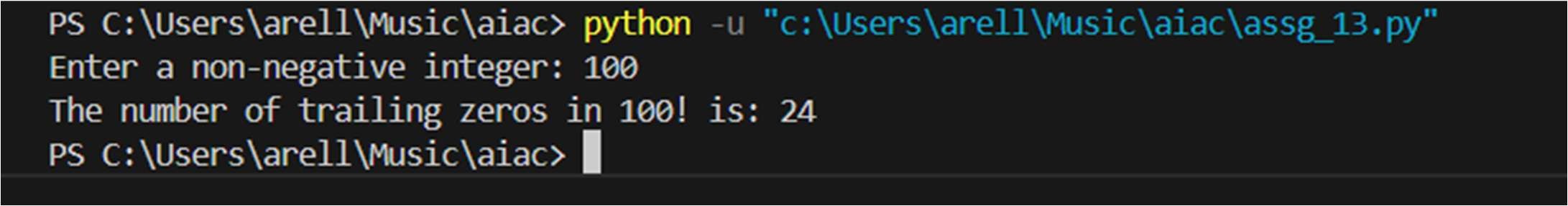
* Accept a non-nega ve integer n.
* Return the number of trailing zeros in n!.

1. Do NOT compute the full factorial.
2. Use an op mized mathema cal approach (count mul ples of 5).
3. Add proper docstrings.
4. Separate user interac on from core logic.
5. Handle edge cases (e.g., nega ve numbers, zero).

Screenshots:



Output:



Task 13 (Collatz Sequence Generator – Test Case Design) • Func on: Generate Collatz sequence un l reaching 1.

* Test Cases to Design:
* Normal: 6 → [6,3,10,5,16,8,4,2,1]
* Edge: 1 → [1]
* Nega ve: -5
* Large: 27 (well-known long sequence)
* Requirement: Validate correctness with pytest.

Explana on:

We need to write a func on 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 un l we reach 1.
* Return the full sequence as a list.

Example

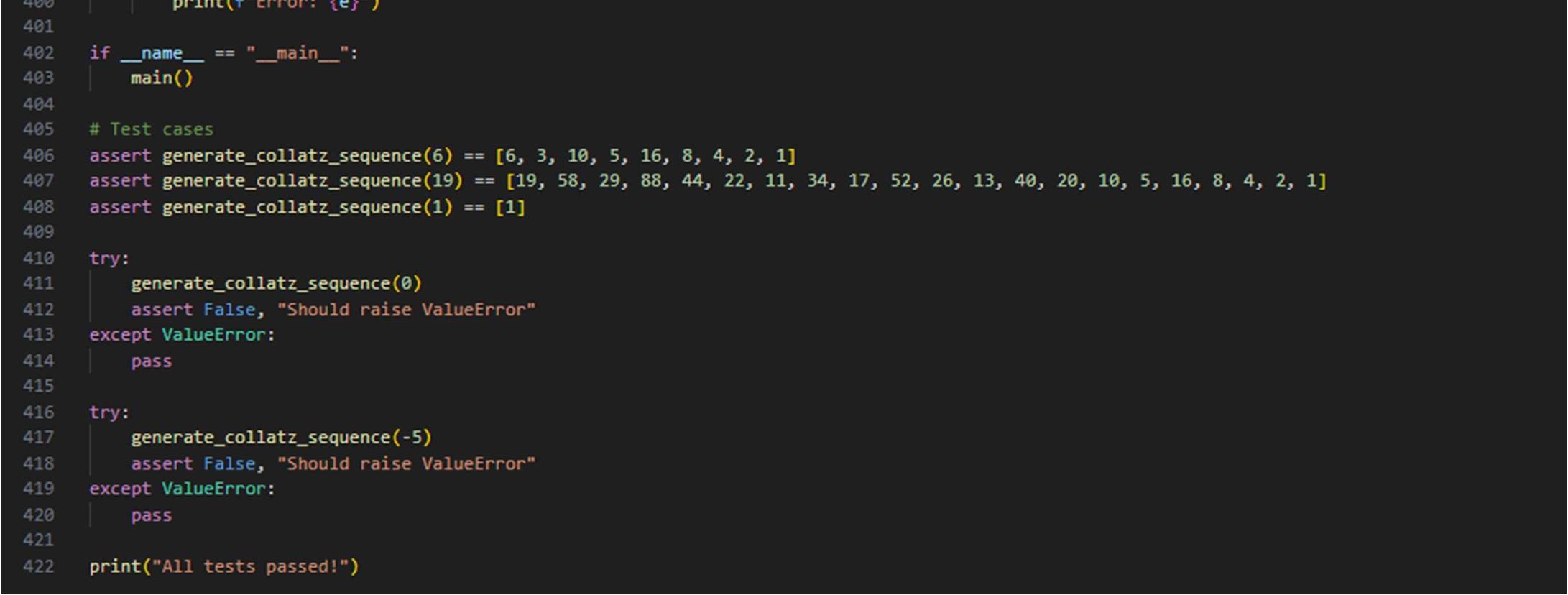
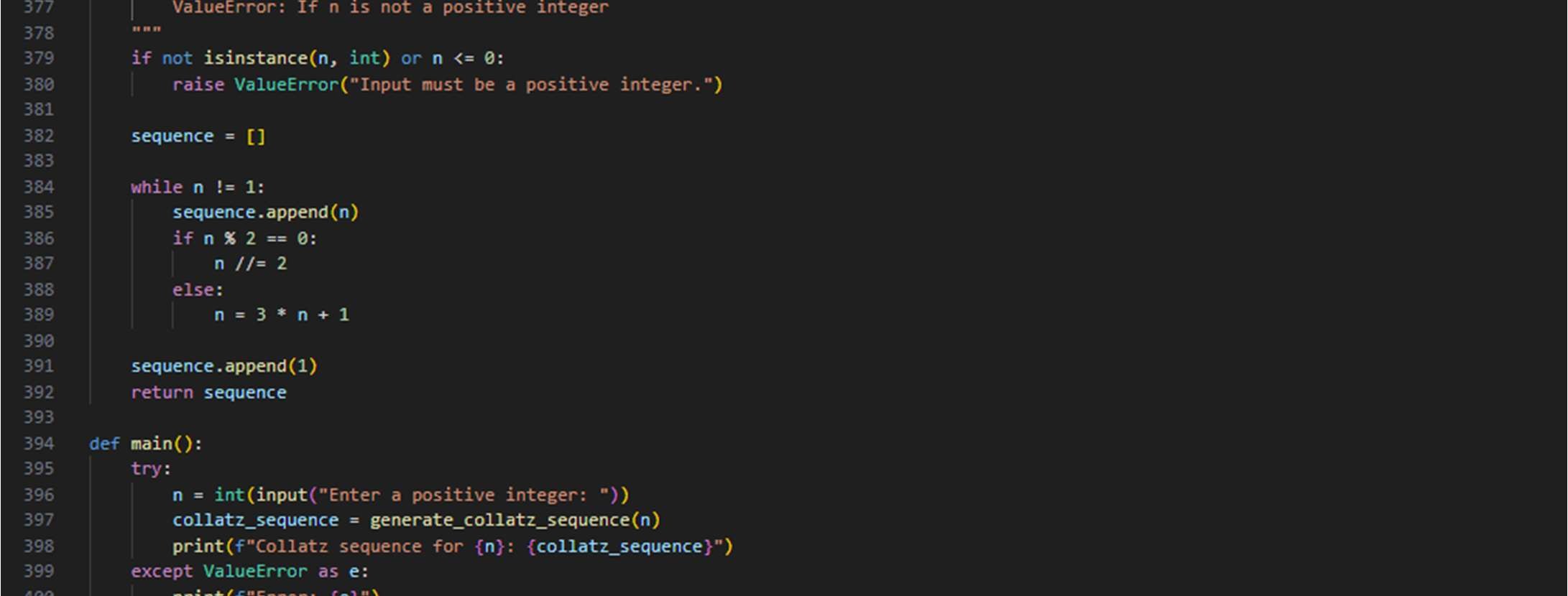
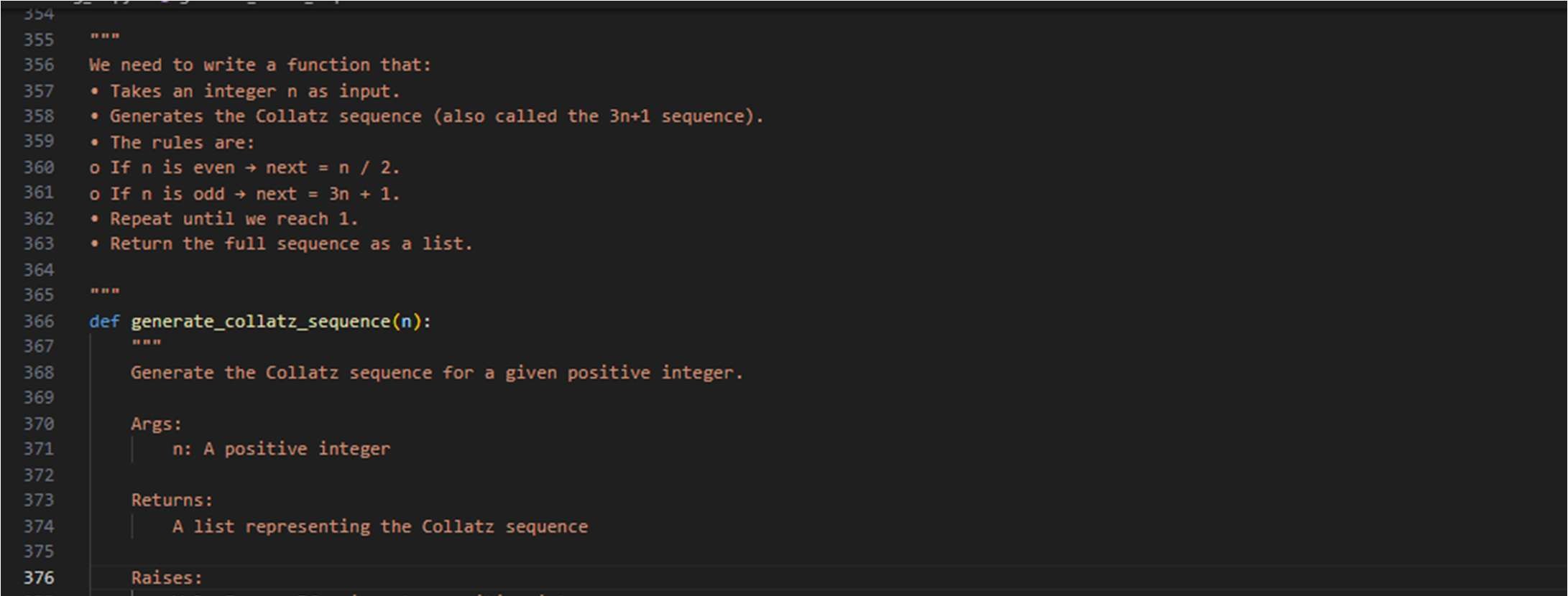
Input: 6

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]

Screenshots:



Output:



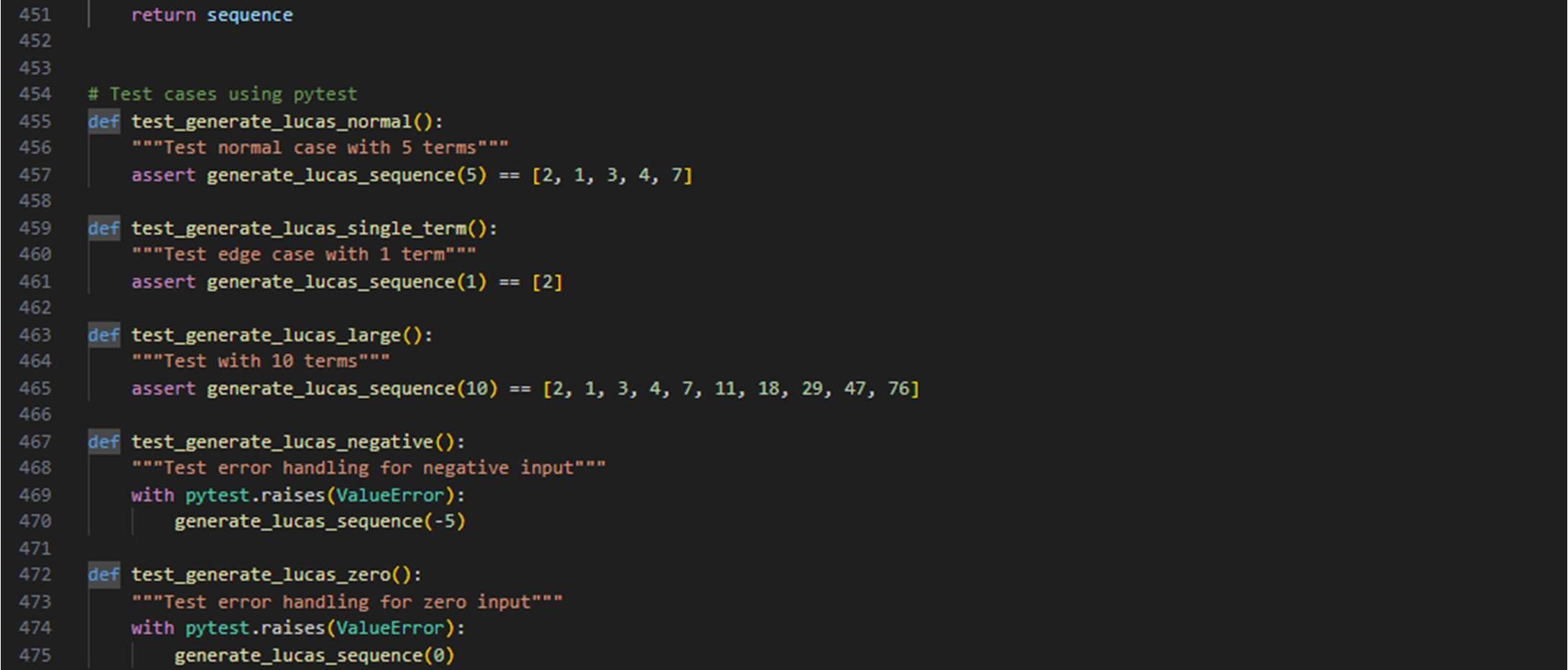
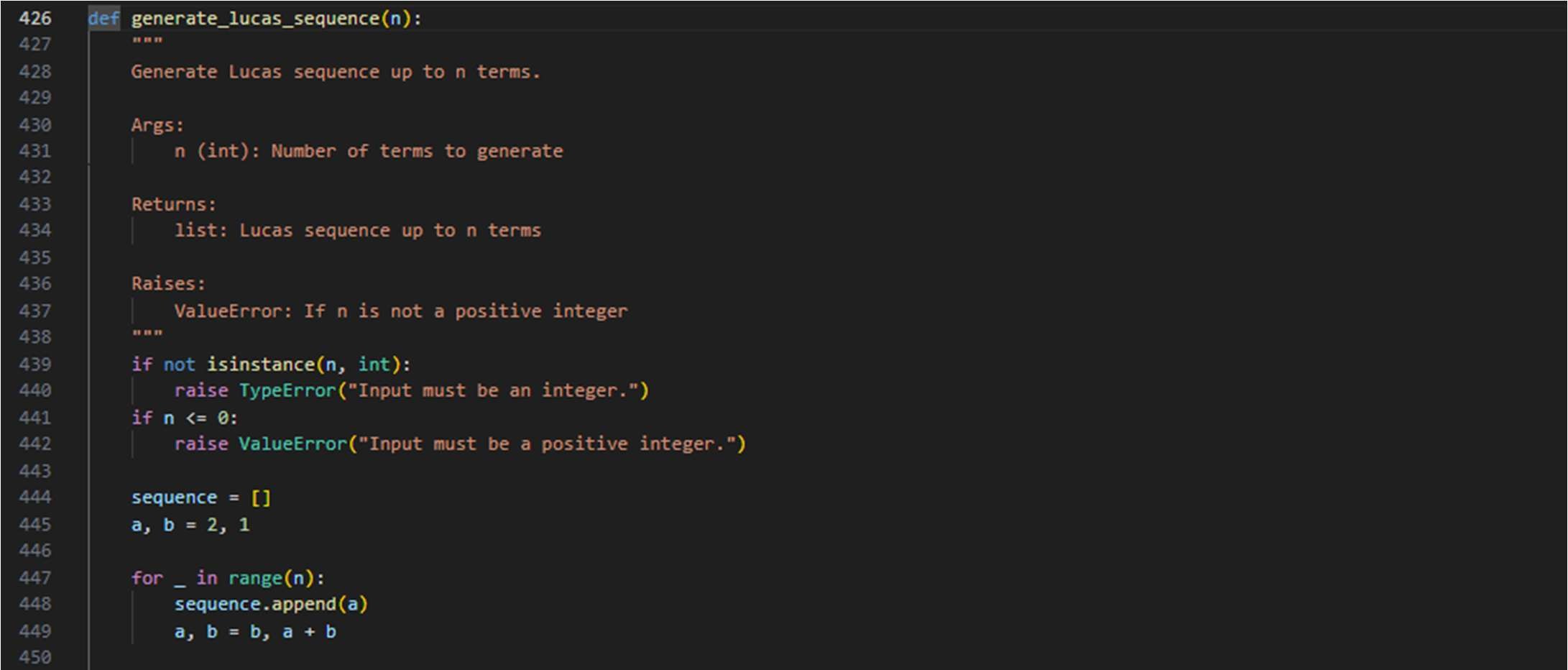
Task 14 (Lucas Number Sequence – Test Case Design)

* Func on: Generate Lucas sequence up to n terms.

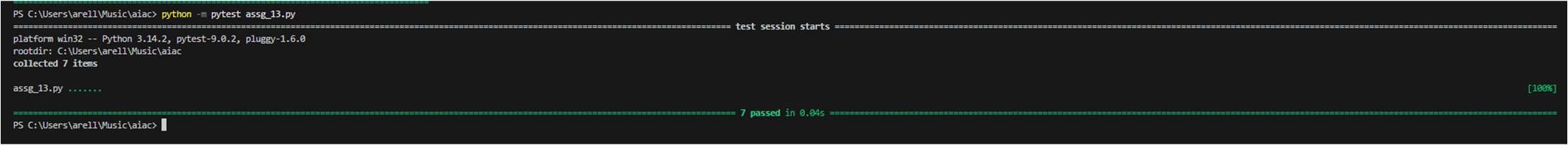
(Starts with 2,1, then Fn = Fn-1 + Fn-2)

* Test Cases to Design:
* Normal: 5 → [2, 1, 3, 4, 7]
* Edge: 1 → [2]
* Nega ve: -5 → Error
* Large: 10 (last element = 76).
* Requirement: Validate correctness with pytest.

Screenshots:



Output:



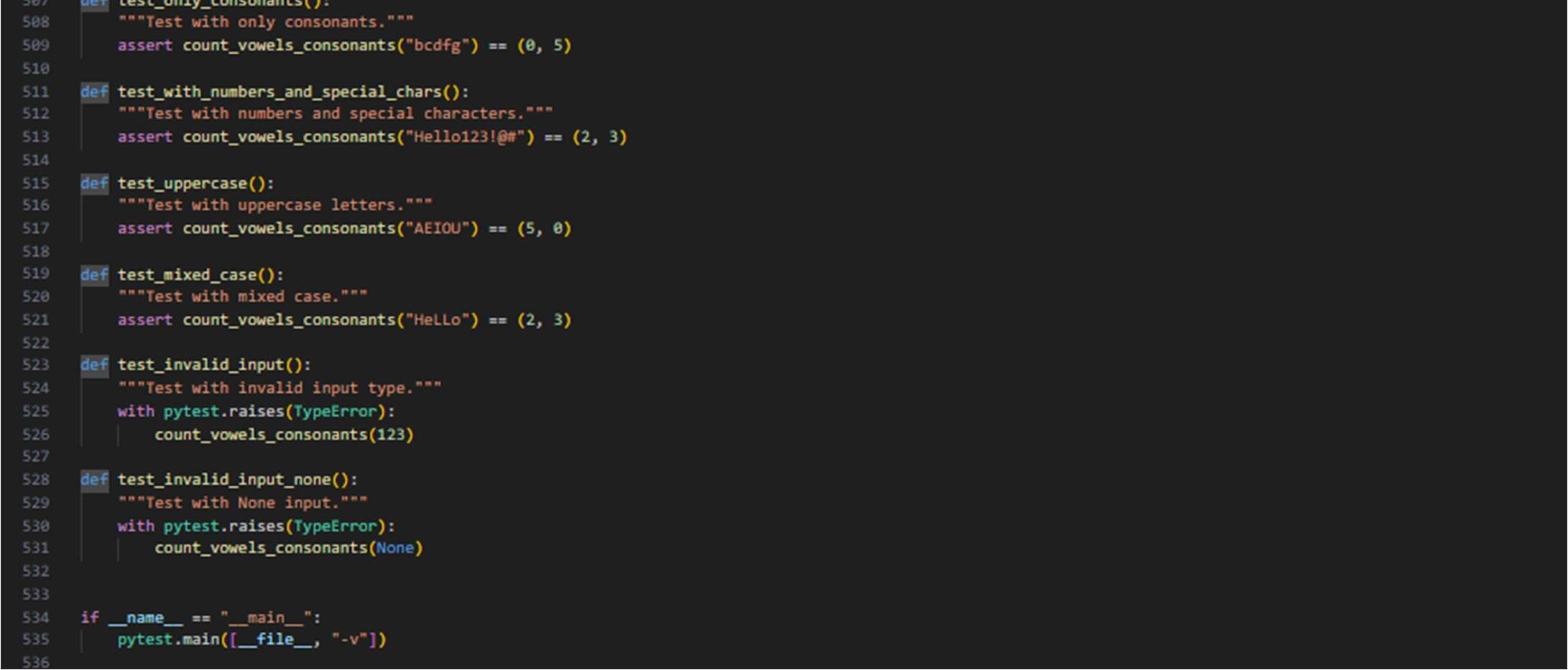
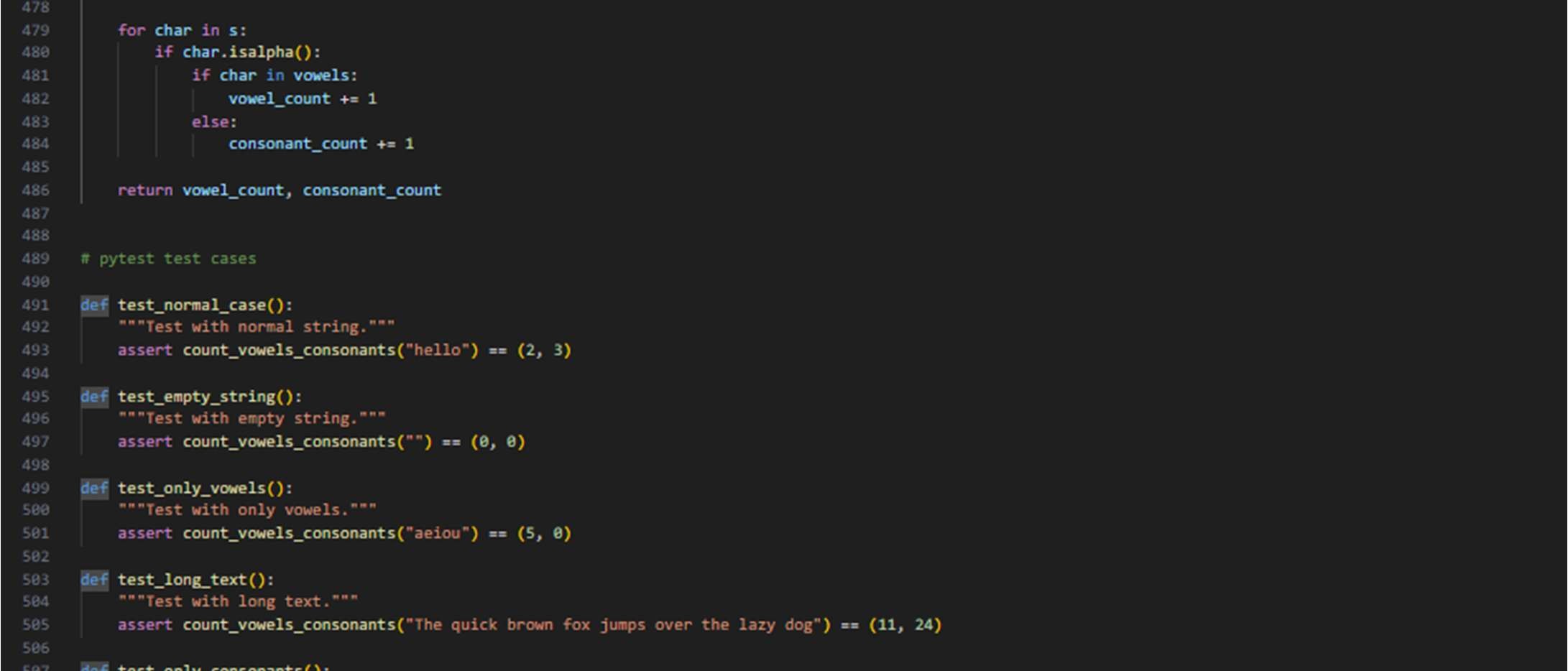
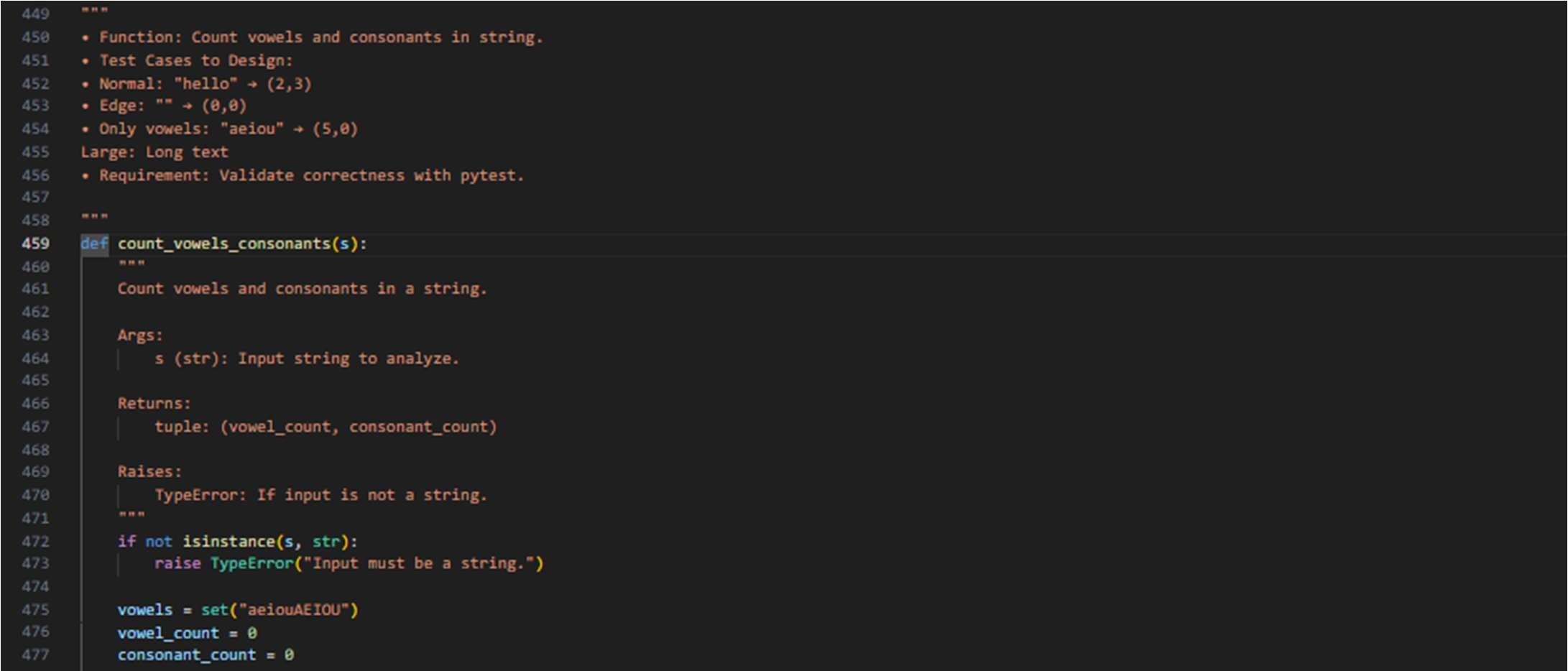
Task 15 (Vowel & Consonant Counter – Test Case Design)

* Func on: 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.

Screenshots:



Output:

