Assigment-9.2

K.SAI ROHAN
2303A52025
Batch-38

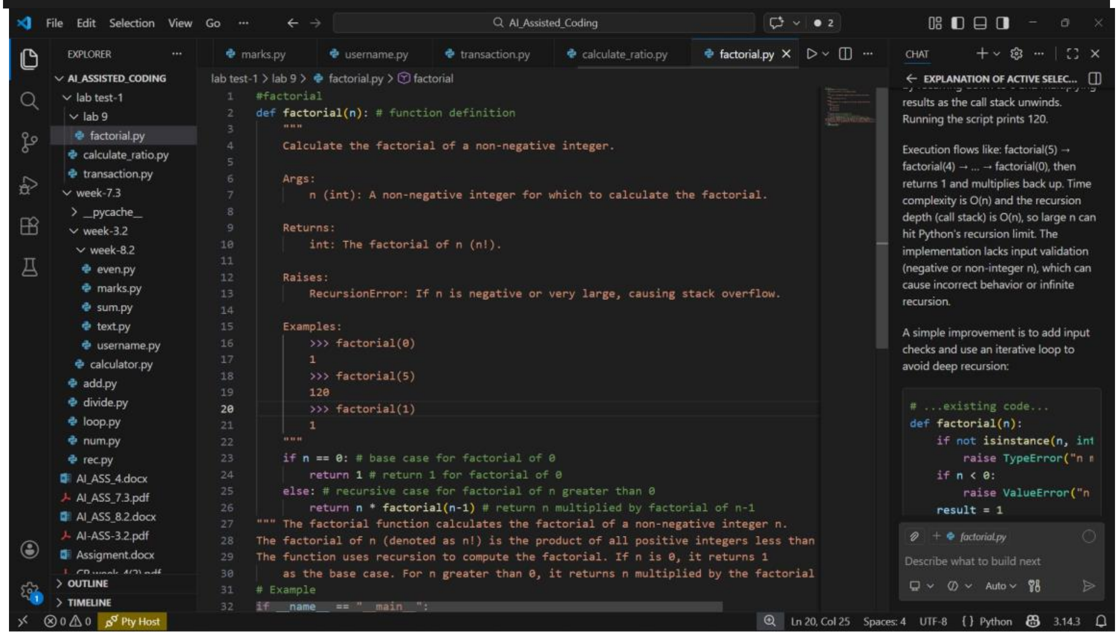
Task Description -1 (Documentation – Function Summary Generation) Task:
Use AI to generate concise functional summaries for each Python function in a given script.
Instructions:
- **Provide a Python script to the AI.**
- **Ask the AI to write a short summary describing the purpose of each function.**
- **Ensure summaries are brief and technically accurate.**
- **Do not include code implementation details.**

Expected Output -1:
A Python script where each function contains a clear and concise summary explaining its purpose.

Code:

```
#factorial def factorial(n): # function definition
    """
    Calculate the factorial of a non-negative integer.

    Args:
        n (int): A non-negative integer for which to
calculate the factorial.
    Returns:
        int: The factorial of n (n!).

Raises:
        RecursionError: If n is negative or very large,
causing stack overflow.

    Examples:
        >>> factorial(0)
        1
        >>> factorial(5)
        120
        >>> factorial(1)
        1
```

```
    """
    if n == 0: # base case for factorial of 0
        return 1 # return 1 for factorial of 0
    else: # recursive case for factorial of n greater
than 0
        return n * factorial(n-1) # return n multiplied
by factorial of n-1
""" The factorial function calculates the factorial of a
non-negative integer n.
The factorial of n (denoted as n!) is the product of all
positive integers less than or equal to n.
The function uses recursion to compute the factorial. If
n is 0, it returns 1
    as the base case. For n greater than 0, it returns n
multiplied by the factorial of n-1."""
# Example if __name__ ==
"__main__":
print(factorial(5))
```



Task Description -2 (Documentation – Logical Explanation for Conditions and Loops) Task:

Use AI to document the logic behind conditional statements and loops in a Python program.

Instructions:

• Provide a Python program without comments.

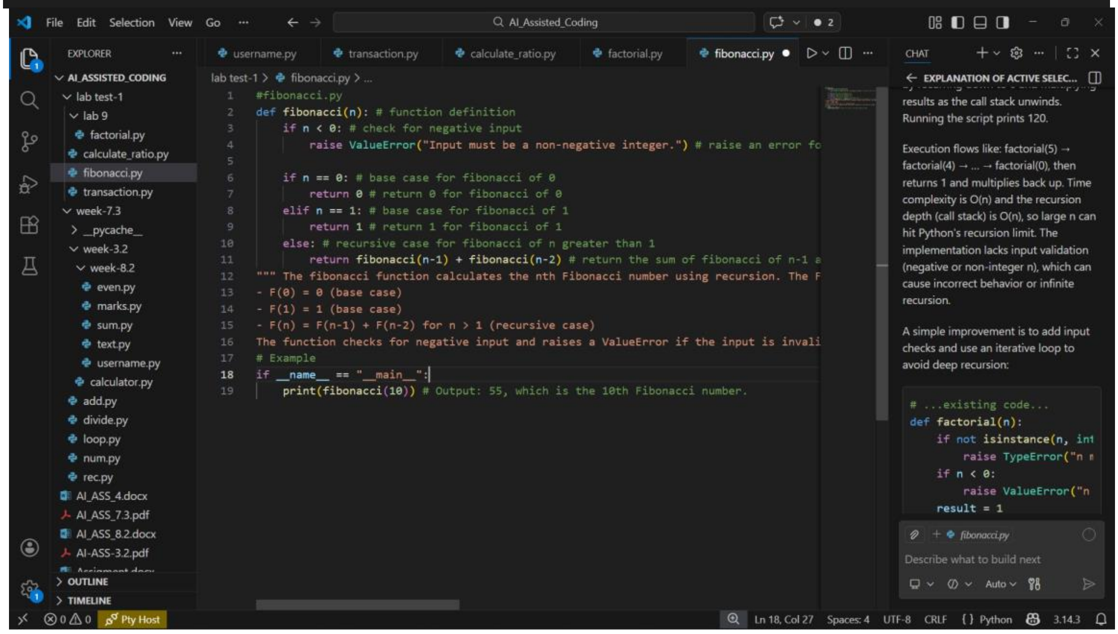
• Instruct AI to explain only decision-making logic and loop behavior.

• Skip basic syntax explanations. Expected Output -2:

Python code with clear explanations describing the logic of conditions and loops.

Code:\

```
#fibonacci.py
def fibonacci(n): # function definition
if n < 0: # check for negative input
        raise ValueError("Input must be a non-negative
integer.") # raise an error for negative input

    if n == 0: # base case for fibonacci of 0
        return 0 # return 0 for fibonacci of 0
    elif n == 1: # base case for fibonacci of 1
        return 1 # return 1 for fibonacci of 1
    else: # recursive case for fibonacci of n greater
than 1
        return fibonacci(n-1) + fibonacci(n-2) # return
the sum of fibonacci of n-1 and fibonacci of n-2
""" The fibonacci function calculates the nth Fibonacci
number using recursion. The Fibonacci sequence is defined
as follows:
- F(0) = 0 (base case)
- F(1) = 1 (base case)
- F(n) = F(n-1) + F(n-2) for n > 1 (recursive case)
The function checks for negative input and raises a
ValueError if the input is invalid. For valid inputs, it
computes the Fibonacci number recursively by summing the
two preceding Fibonacci numbers until it reaches the base
cases. """ # Example if
__name__ == "__main__":
    print(fibonacci(10)) # Output: 55, which is the 10th
Fibonacci number.
```



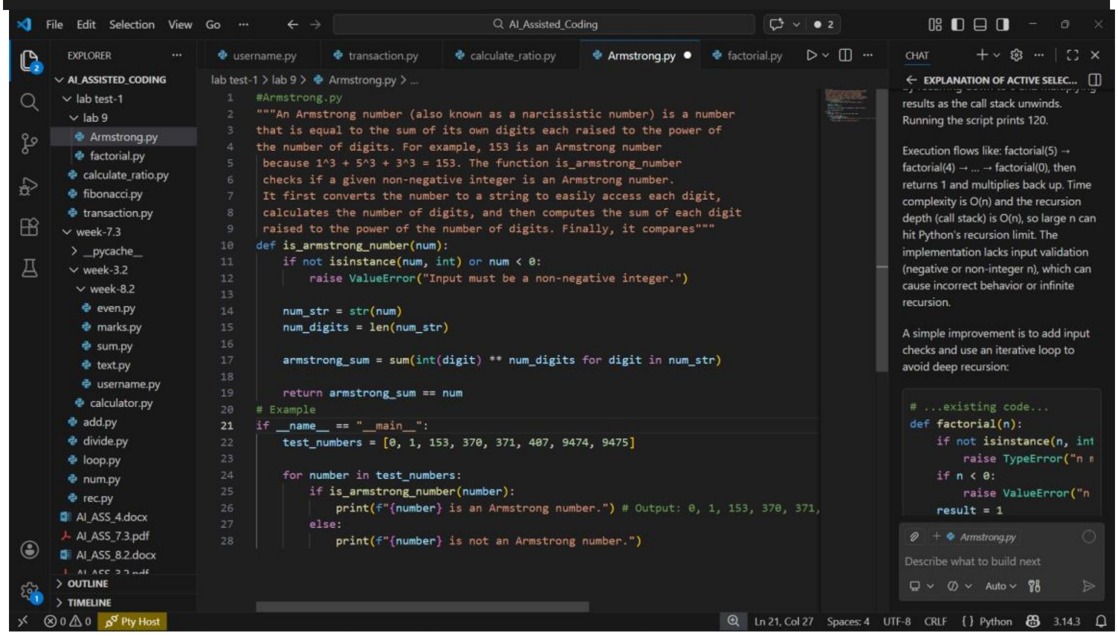
Task Description -3 (Documentation – File-Level Overview)
Task:
Use AI to generate a high-level overview describing the functionality of an entire Python file.
Instructions:
- **Provide the complete Python file to AI.**
- **Ask AI to write a brief overview summarizing the file's purpose and functionality.**
- **Place the overview at the top of the file.**

Expected Output -3:
A Python file with a clear and concise file-level overview at the Beginning

Code:

```
#Armstrong.py
"""An Armstrong number (also known as a narcissistic
number) is a number
that is equal to the sum of its own digits each raised to
the power of
the number of digits. For example, 153 is an Armstrong
number
 because 1^3 + 5^3 + 3^3 = 153. The function
is_armstrong_number
 checks if a given non-negative integer is an Armstrong
number.
 It first converts the number to a string to easily
access each digit,
 calculates the number of digits, and then computes the
sum of each digit
```

```
 raised to the power of the number of digits. Finally, it
compares""" def
is_armstrong_number(num):
    if not isinstance(num, int) or num < 0:
        raise ValueError("Input must be a non-negative
integer.")

    num_str = str(num)
num_digits = len(num_str)

    armstrong_sum = sum(int(digit) ** num_digits for
digit in num_str)

    return armstrong_sum == num
# Example if __name__ ==
"__main__":
    test_numbers = [0, 1, 153, 370, 371, 407, 9474, 9475]

    for number in test_numbers:
if is_armstrong_number(number):
            print(f"{number} is an Armstrong number.") #
Output: 0, 1, 153, 370, 371, 407, and 9474 are Armstrong
numbers, while 9475 is not.
        else:
            print(f"{number} is not an Armstrong
number.")
```



Task Description -4 (Documentation – Refine Existing Documentation) Task:

Use AI to improve clarity and consistency of existing documentation in Python code. Instructions:

- **Provide Python code containing basic or unclear comments. • Ask AI to rewrite the documentation to improve clarity and consistency.**
- **Ensure technical meaning remains unchanged.**

Expected Output -4:

Python code with refined and improved documentation that is clear and consistent.

Code:

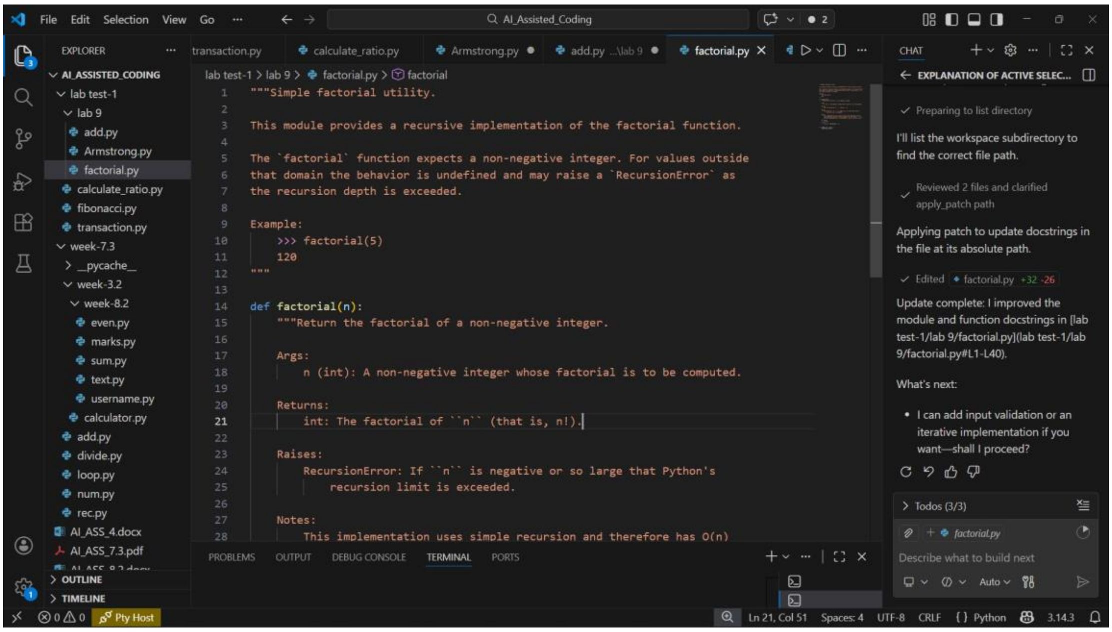
```
"""Simple factorial utility.

This module provides a recursive implementation of the
factorial function.
The `factorial` function expects a non-negative integer.
For values outside
that domain the behavior is undefined and may raise a
`RecursionError` as the
recursion depth is exceeded.
Example:
    >>> factorial(5)
    120 """
```

```
def factorial(n):
    """Return the factorial of a non-negative
integer.
```

```
    Args:
        n (int): A non-negative integer whose factorial is
to be computed.

    Returns:
        int: The factorial of ``n`` (that is, n!).

    Raises:
        RecursionError: If ``n`` is negative or so large
that Python's
            recursion limit is exceeded.

    Notes:
        This implementation uses simple recursion and
therefore has O(n)
        time complexity and O(n) recursion depth. For
very large ``n``, an
        iterative implementation is recommended to avoid
recursion limits.
    """     if n == 0:
        return 1
```

```python
    else:
        return n * factorial(n - 1)
```

```python
if __name__ == "__main__":
print(factorial(5))
```

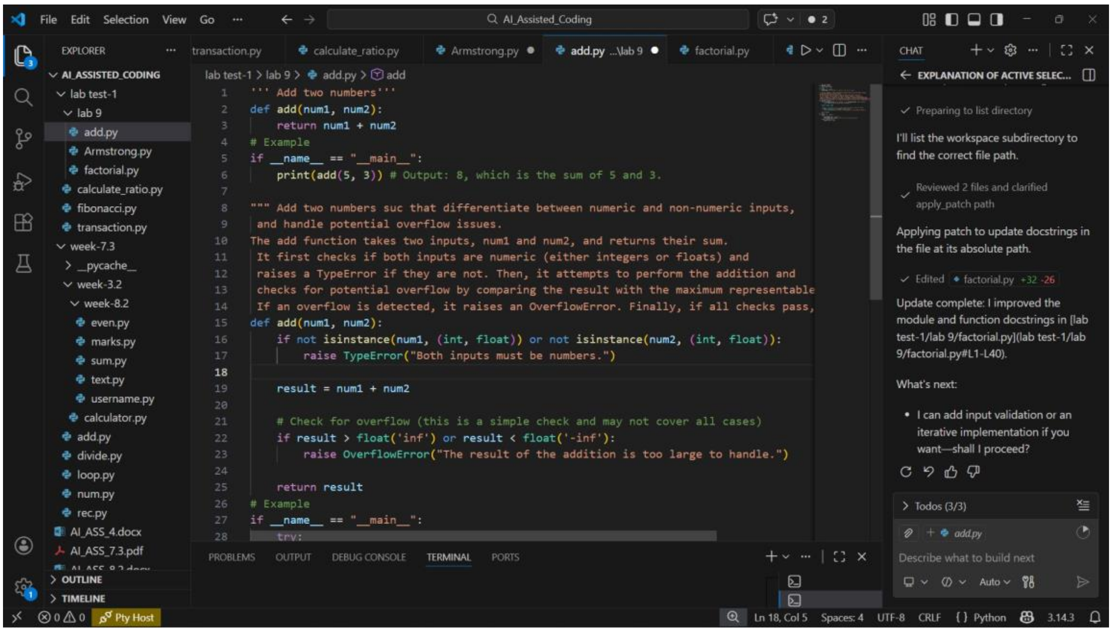


Task Description -5 (Documentation – Prompt Detail Impact Study)
Task:
Study the impact of prompt detail on AI-generated documentation quality.
Instructions:
Create two prompts: one brief and one detailed.
• Use both prompts to document the same Python function.
• Compare the generated outputs.
Expected Output -5:
A comparison table highlighting differences in completeness, clarity, and accuracy of documentation.

Code:

```python
''' Add two numbers'''
def add(num1, num2):
return num1 + num2
# Example if __name__ ==
"__main__":
```

```
    print(add(5, 3)) # Output: 8, which is the sum of 5
and 3.

""" Add two numbers suc that differentiate between
numeric and non-numeric inputs,
 and handle potential overflow issues.
The add function takes two inputs, num1 and num2, and
returns their sum.
 It first checks if both inputs are numeric (either
integers or floats) and
 raises a TypeError if they are not. Then, it attempts to
perform the addition and
 checks for potential overflow by comparing the result
with the maximum representable float value.
 If an overflow is detected, it raises an OverflowError.
Finally, if all checks pass, it returns"""
def add(num1, num2):
    if not isinstance(num1, (int, float)) or not
isinstance(num2, (int, float)):
        raise TypeError("Both inputs must be numbers.")

    result = num1 + num2

    # Check for overflow (this is a simple check and may
not cover all cases)
    if result > float('inf') or result < float('-inf'):
        raise OverflowError("The result of the addition
is too large to handle.")

    return result
# Example if __name__ ==
"__main__":
    try:
        print(add(5, 3)) # Output: 8
print(add(1e308, 1e308)) # This will raise an
OverflowError     except
Exception as e:
        print(f"Error: {e}")
```



File Edit Selection View Go
AI_Assisted_Coding
EXPLORER
AI_ASSISTED_CODING
lab test-1
lab 9
add.py
Armstrong.py
factorial.py
calculate_ratio.py
fibonacci.py
transaction.py
week-7.3
__pycache__
week-3.2
week-8.2
even.py
marks.py
sum.py
text.py
username.py
calculator.py
add.py
divide.py
loop.py
num.py
rec.py
AI_ASS_4.docx
AI_ASS_7.3.pdf
OUTLINE
TIMELINE
transaction.py
calculate_ratio.py
Armstrong.py
add.py ...\lab 9
factorial.py
lab test-1 > lab 9 > add.py > add
''' Add two numbers'''
def add(num1, num2):
    return num1 + num2
# Example
if __name__ == "__main__":
    print(add(5, 3)) # Output: 8, which is the sum of 5 and 3.
""" Add two numbers suc that differentiate between numeric and non-numeric inputs,
 and handle potential overflow issues.
The add function takes two inputs, num1 and num2, and returns their sum.
 It first checks if both inputs are numeric (either integers or floats) and
 raises a TypeError if they are not. Then, it attempts to perform the addition and
 checks for potential overflow by comparing the result with the maximum representable
 If an overflow is detected, it raises an OverflowError. Finally, if all checks pass,
def add(num1, num2):
    if not isinstance(num1, (int, float)) or not isinstance(num2, (int, float)):
        raise TypeError("Both inputs must be numbers.")
    result = num1 + num2
    # Check for overflow (this is a simple check and may not cover all cases)
    if result > float('inf') or result < float('-inf'):
        raise OverflowError("The result of the addition is too large to handle.")
    return result
# Example
if __name__ == "__main__":
    try:
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
CHAT
EXPLANATION OF ACTIVE SELEC...
Preparing to list directory
I'll list the workspace subdirectory to find the correct file path.
Reviewed 2 files and clarified apply_patch path
Applying patch to update docstrings in the file at its absolute path.
Edited factorial.py +32 -26
Update complete: I improved the module and function docstrings in [lab test-1/lab 9/factorial.py](lab test-1/lab 9/factorial.py#L1-L40).
What's next:
I can add input validation or an iterative implementation if you want—shall I proceed?
Todos (3/3)
add.py
Describe what to build next
Auto
Ln 18, Col 5 Spaces: 4 UTF-8 CRLF { } Python 3.14.3
Pty Host