

Assignment-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))

```

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

- File Explorer:** Shows a project structure with files like marks.py, username.py, transaction.py, calculate_ratio.py, factorial.py, and several .pycache_ files.
- Code Editor:** Displays the factorial.py script. The code defines a factorial function that calculates the factorial of a non-negative integer n. It includes a docstring, type hints for args and returns, examples of function calls, and a note about recursion depth.
- Right-hand pane:** Contains an "EXPLANATION OF ACTIVE SELECTION" section with the following text:

results as the call stack unwinds.
Running the script prints 120.

Execution flows like: factorial() → factorial(4) → ... → factorial(0), then returns 1 and multiplies back up. Time complexity is O(n) and the recursion depth (call stack) is O(n), so large n can hit Python's recursion limit. The implementation lacks input validation (negative or non-integer n), which can cause incorrect behavior or infinite recursion.

A simple improvement is to add input checks and use an iterative loop to avoid deep recursion:

```
# ...existing code...
def factorial(n):
    if not isinstance(n, int)
        raise TypeError("n must be an integer")
    if n < 0:
        raise ValueError("n must be non-negative")
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

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:\b

```

#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.

```

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

- File Explorer:** Shows a tree view of files in the project. The "fibbonacci.py" file is selected.
- Code Editor:** Displays the content of the "fibbonacci.py" file, identical to the one above.
- Output/Run View:** Shows the execution results of the script. It includes an explanation of the active selection, a call stack unwind, and a detailed analysis of the execution flow and complexity.
- Status Bar:** Shows the current file is "fibbonacci.py", with 18 lines of code, 0 errors, 0 warnings, and 0 info messages. It also shows the build status as "0△ 0" and the host as "Py Host".

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.")

```

The screenshot shows the PyCharm IDE interface with the following details:

- File Structure:** The left sidebar shows a project structure with folders like 'EXPLORER', 'AI ASSISTED CODING', and files such as 'username.py', 'transaction.py', 'calculate_ratio.py', 'Armstrong.py', 'factorial.py', 'fibonacci.py', 'transaction.py', 'week-7.3', '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', and 'AI_ASS.4.docx', 'AI_ASS.7.3.pdf', 'AI_ASS.8.2.docx'.
- Editor:** The main editor window displays the 'Armstrong.py' code. The code defines a function `is_armstrong_number` that checks if a given number is an Armstrong number by comparing the sum of its digits raised to the power of their count with the original number. It also includes an example section demonstrating the function with a list of test numbers.
- Right Sidebar:** This sidebar contains several sections:
 - CHAT:** Shows a message from AI: "EXPLANATION OF ACTIVE SELECTION results as the call stack unwinds. Running the script prints 120."
 - EXPLANATION OF ACTIVE SELECTION:** Details the execution flow of the factorial function, noting O(n) time complexity and O(n) recursion depth, and mentioning Python's recursion limit.
 - Code Snippet:** Displays a snippet of the factorial implementation with comments explaining the logic.
 - Build Status:** Shows '0' errors and '0' warnings.
 - Build Log:** Shows 'In 21, Col 27' and 'Spaces: 4'.
 - File Encoding:** Shows 'UTF-8'.
 - Python Version:** Shows '3.14.3'.

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
```

```

else:
    return n * factorial(n - 1)

```

```

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

```

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a folder structure for "AI_ASSISTED_CODING" containing files like transaction.py, calculate_ratio.py, Armstrong.py, add.py, factorial.py, fibonacci.py, and lab test-1, lab 9, week-7.3, week-3.2, week-8.2.
- Editor:** Displays the content of factorial.py. The code includes a multi-line docstring at the top and several sections of generated documentation (Args, Returns, Raises, Notes) below it.
- Chat Panel:** Shows AI-generated explanations and tasks, such as "Preparing to list directory" and "Reviewing patch to update docstrings in the file at its absolute path".
- Todos:** Shows a list of 3 items related to the factorial function.
- Status Bar:** Shows the current file is factorial.py, with 32 changes made and 26 removed. It also shows the line count (Ln 21), column count (Col 51), and other settings.

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:

```

''' 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}")
```

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows the workspace structure under "AL ASSISTED CODING". The "add.py" file is selected.
- Editor:** Displays the Python code for "add.py". The code adds two numbers, handles numeric and non-numeric inputs, and includes overflow checking.
- Output:** Shows AI-generated notes and tasks:
 - 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).
- Todos:** Shows 3 tasks related to the current file.
- Bottom Status Bar:** Shows the current file is "add.py", line 18, column 5, with 4 spaces, encoding is UTF-8, and the Python extension version is 3.14.3.