

# AI ASS 9.4

Name:A.Harshita

H.T.NO:2303A52211

Batch:43

## Task 1

Generate google-style docstrings for the following Python functions.

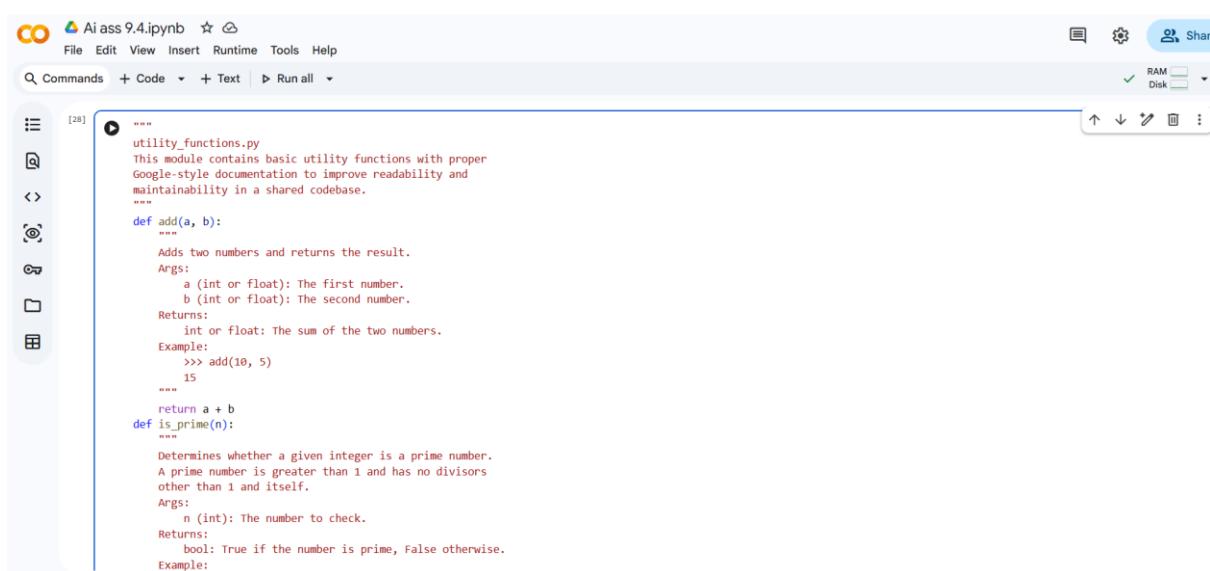
Each docstring should include:

- A brief description of the function
- Parameters with data types and short explanations
- Return values with data types
- At least one example usage in doctest format (if applicable)

Do not modify the original function logic.

Ensure the formatting follows the Google Python Style Guide.

Keep the tone professional and clear for use in a shared codebase.



The screenshot shows a Jupyter Notebook interface with a single code cell containing two Python functions: `add` and `is_prime`. The code is formatted with Google-style docstrings, including descriptions, parameter details, return types, and examples.

```
[28] utility_functions.py
This module contains basic utility functions with proper
Google-style documentation to improve readability and
maintainability in a shared codebase.

def add(a, b):
    """
    Adds two numbers and returns the result.
    Args:
        a (int or float): The first number.
        b (int or float): The second number.
    Returns:
        int or float: The sum of the two numbers.
    Example:
        >>> add(10, 5)
        15
    """

    return a + b

def is_prime(n):
    """
    Determines whether a given integer is a prime number.
    A prime number is greater than 1 and has no divisors
    other than 1 and itself.
    Args:
        n (int): The number to check.
    Returns:
        bool: True if the number is prime, False otherwise.
    Example:
        >>> is_prime(11)
        True
        >>> is_prime(4)
        False
    """

    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

```
[28] Example:
    >>> is_prime(7)
    True
    >>> is_prime(10)
    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 factorial(n):
    """
    Computes the factorial of a non-negative integer using recursion.
    The factorial of n (n!) is the product of all positive
    integers less than or equal to n.
    Args:
        n (int): A non-negative integer.
    Returns:
        int: The factorial of the given number.
    Raises:
        ValueError: If n is negative.
    Example:
        >>> factorial(5)
        120
    """
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    if n == 0 or n == 1:
```

```
[28] if n < 0:
    raise ValueError("Factorial is not defined for negative numbers.")
if n == 0 or n == 1:
    return 1
return n * factorial(n - 1)
def reverse_string(s):
    """
    Reverses the given string.
    Args:
        s (str): The string to reverse.
    Returns:
        str: The reversed version of the input string.
    Example:
        >>> reverse_string("hello")
        'olleh'
    """
    return s[::-1]
# Example usage when running this file directly
if __name__ == "__main__":
    print("Add:", add(3, 4))
    print("Is Prime:", is_prime(11))
    print("Factorial:", factorial(5))
    print("Reverse String:", reverse_string("Python"))

...
Add: 7
Is Prime: True
Factorial: 120
Reverse String: nohtyP
```

## Task2

You are a senior Python developer reviewing a codebase for readability improvements.

The following Python script contains complex logic, including loops, conditional statements, and algorithms. The code works correctly but lacks clarity.

## Your task:

- Insert concise inline comments ONLY where the logic is complex or non-obvious.
- Explain WHY the logic exists, not what basic Python syntax does.
- Avoid commenting on trivial or self-explanatory lines.
- Do not modify the original logic or structure.
- Keep comments professional and minimal to avoid clutter.

Return the improved version of the script with meaningful inline comments added.

```
readability_enhanced.py
This script demonstrates improved readability by adding concise,
meaningful inline comments to complex algorithmic logic.
Only non-obvious reasoning is explained to avoid clutter.

def fibonacci(n):
    if n <= 0:
        return []
    if n == 1:
        return [0]
    sequence = [0, 1]
    # Each number is derived from the sum of the previous two numbers
    for i in range(2, n):
        sequence.append(sequence[i - 1] + sequence[i - 2])
    return sequence

def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    # Narrow the search space until the element is found or exhausted
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        # Eliminate the left half if target is greater than mid element
        elif arr[mid] < target:
            left = mid + 1
        # Otherwise eliminate the right half
        else:
            right = mid - 1
```

```
right = mid - 1
# Target does not exist in array
return -1

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False # Track if any swap occurs in this pass
        # Largest element in each pass settles at the end,
        # so we reduce the comparison range by i
        for j in range(0, n - i - 1):
            # Swap only when elements are out of order
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no swaps occurred, array is already sorted (optimization)
        if not swapped:
            break
    return arr

if __name__ == "__main__":
    print("Fibonacci:", fibonacci(8))
    print("Binary Search:", binary_search([1, 3, 5, 7, 9], 7))
    print("Bubble Sort:", bubble_sort([5, 1, 4, 2, 8]))
```

## Task 3

You are a Python documentation expert preparing a module for internal or public use.

Analyze the following complete Python module and generate a professional multi-line module-level docstring to be placed at the top of the file.

The docstring must include:

- A clear explanation of the module's purpose
- Required libraries or dependencies
- A brief description of key functions and classes
- A short example demonstrating how to use the module
- Professional tone suitable for production code

Do not modify the existing code.

Only generate the module-level docstring.

```
student_management.py
Student Management Module
=====
Purpose:
-----
This module provides utility functions and a class to manage student records, including adding students, calculating average marks, searching for students, and sorting records.
It is designed to demonstrate clean structure, maintainability, and production-ready documentation suitable for internal tools or repository-based projects.

Dependencies:
-----
- Python 3.x (standard library only)
- No external libraries required

Key Components:
-----
1. Student (class)
   - Represents a student with name, roll number, and marks.
   - Provides method to calculate average marks.
2. add_student(records, student)
   - Adds a Student object to the records list.
3. find_student(records, roll_number)
   - Searches for a student using roll number.
4. sort_students_by_average(records)
   - Sorts students based on their average marks.

Example Usage:
-----
```

The screenshot shows a Jupyter Notebook interface with the following code:

```
class Student:
    def __init__(self, name, roll_number, marks):
        self.name = name
        self.roll_number = roll_number
        self.marks = marks
    def calculate_average(self):
        return sum(self.marks) / len(self.marks)
def add_student(records, student):
    records.append(student)
def find_student(records, roll_number):
    for student in records:
        if student.roll_number == roll_number:
            return student
    return None
def sort_students_by_average(records):
    return sorted(records, key=lambda s: s.calculate_average(), reverse=True)
if __name__ == "__main__":
    records = []
    s1 = Student("Alice", 101, [85, 90, 88])
    s2 = Student("Bob", 102, [78, 82, 80])
    add_student(records, s1)
    add_student(records, s2)
    print("Sorted by Average:")
    for student in sort_students_by_average(records):
        print(student.name, student.calculate_average())
```

Output:

```
... Sorted by Average:
Alice 87.66666666666666
Bob 80.0
```

## Task 4

You are a Python refactoring specialist helping standardize documentation in a legacy project.

The following script contains detailed inline comments inside functions explaining their behavior. These should be converted into structured docstrings.

Your task:

- Convert relevant explanatory comments into proper Google-style (or NumPy-style) docstrings.
- Preserve the original meaning and intent.
- Include:
  - Description
  - Parameters with types
  - Return values with types
- Remove redundant inline comments after conversion.
- Do not modify function logic.

Return the cleaned and standardized version of the script.

File Edit View Insert Runtime Tools Help

Commands + Code + Text ▶ Run all

data\_processing.py

```
This module provides basic numerical data processing utilities.  
Legacy inline comments have been converted into standardized  
Google-style docstrings for improved readability and consistency.  
  
def calculate_average(numbers):  
    """  
        Calculates the arithmetic mean of a list of numbers.  
        The function sums all values in the list and divides  
        the result by the total number of elements.  
        Args:  
            numbers (list of int or float): A non-empty list  
                containing numeric values.  
        Returns:  
            float: The average of the given numbers.  
        Raises:  
            ValueError: If the input list is empty.  
  
        Example:  
            >>> calculate_average([10, 20, 30])  
            20.0  
    """  
    if not numbers:  
        raise ValueError("List cannot be empty.")  
    return sum(numbers) / len(numbers)  
  
def find_maximum(numbers):  
    """  
        Finds the largest value in a list of numbers.  
        The function iterates through the list and keeps track  
        of the maximum value found.  
        Args:  
            numbers (list of int or float): A non-empty list  
                containing numeric values.  
        Returns:  
            int or float: The maximum value in the list.  
        Raises:  
            ValueError: If the input list is empty.  
  
        Example:  
            >>> find_maximum([3, 7, 2, 9])  
            9  
    """  
    if not numbers:  
        raise ValueError("List cannot be empty.")  
    max_value = numbers[0]  
    for num in numbers:  
        if num > max_value:  
            max_value = num  
  
    return max_value  
  
def remove_duplicates(items):  
    """  
        Removes duplicate elements from a list while preserving order.  
        The function iterates through the list and stores  
        only elements that have not been seen before.  
        Args:  
            items (list): A list containing hashable elements.  
        Returns:  
            list: A list with duplicates removed.
```

File Edit View Insert Runtime Tools Help

Commands + Code + Text ▶ Run all

data\_processing.py

```
Args:  
    numbers (list of int or float): A non-empty list  
        of numeric values.  
Returns:  
    int or float: The maximum value in the list.  
Raises:  
    ValueError: If the input list is empty.  
  
Example:  
    >>> find_maximum([3, 7, 2, 9])  
    9  
    """  
    if not numbers:  
        raise ValueError("List cannot be empty.")  
    max_value = numbers[0]  
    for num in numbers:  
        if num > max_value:  
            max_value = num  
  
    return max_value  
  
def remove_duplicates(items):  
    """  
        Removes duplicate elements from a list while preserving order.  
        The function iterates through the list and stores  
        only elements that have not been seen before.  
        Args:  
            items (list): A list containing hashable elements.  
        Returns:  
            list: A list with duplicates removed.
```

The screenshot shows a Jupyter Notebook interface with a code cell containing Python code. The code defines a function `remove_duplicates` that takes a list of hashable elements and returns a new list with duplicates removed. It includes docstrings for `Args`, `Returns`, and `Example`. The output of the cell shows the function's behavior on a sample list [1, 2, 2, 3, 1]. The notebook also displays memory usage information at the top right.

```
only elements that have not been seen before.
Args:
    items (list): A list containing hashable elements.

Returns:
    list: A new list with duplicates removed.
Example:
    >>> remove_duplicates([1, 2, 2, 3, 1])
    [1, 2, 3]
```
unique_items = []
seen = set()

for item in items:
    if item not in seen:
        unique_items.append(item)
    seen.add(item)

return unique_items

if __name__ == "__main__":
    print("Average:", calculate_average([10, 20, 30]))
    print("Maximum:", find_maximum([5, 12, 3, 9]))
    print("No Duplicates:", remove_duplicates([1, 2, 2, 3, 4]))
```
...
Average: 20.0
Maximum: 12
No Duplicates: [1, 2, 3, 4]
```

## Task 5

You are a Python developer building an internal documentation scaffolding tool. Design a small Python utility that:

- Reads a given .py file
- Automatically detects all functions and classes
- Inserts placeholder Google-style docstrings if none exist
- Preserves indentation and formatting
- Does not modify existing docstrings

The generated placeholder docstrings should include:

- Short description placeholder - Args section
- Returns section (if applicable)

The goal is documentation scaffolding, not perfect documentation.

Provide:

1. A complete working Python script
2. Clear explanation of how it works
3. Example of how to run it

Ensure the solution is clean, readable, and suitable for internal tooling.

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

RAM Disk

```
[27] In [1]:
```

```
import ast
import os

def generate_placeholder(name, node_type):
    """Return a Google-style placeholder docstring."""
    if node_type == "function":
        return [
            "    """",
            f"        {name} function.",
            "    Args:",
            "        TODO: Add parameter descriptions.",
            "    Returns:",
            "        TODO: Add return description.",
            "    """"
        ]
    else:
        return [
            "    """",
            f"        {name} class.",
            "    Attributes:",
            "        TODO: Describe attributes.",
            "    Methods:",
            "        TODO: Describe important methods.",
            "    """"
        ]

def add_docstrings(input_file, output_file):
```

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

RAM Disk

```
[27] In [1]:
```

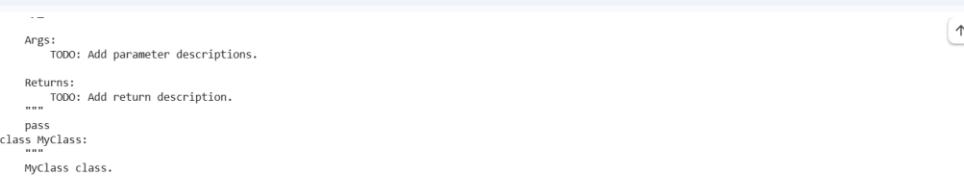
```
import ast
import os

def generate_placeholder(name, node_type):
    """Return a Google-style placeholder docstring."""
    if node_type == "function":
        return [
            "    """",
            f"        {name} function.",
            "    Args:",
            "        TODO: Add parameter descriptions.",
            "    Returns:",
            "        TODO: Add return description.",
            "    """"
        ]
    else:
        return [
            "    """",
            f"        {name} class.",
            "    Attributes:",
            "        TODO: Describe attributes.",
            "    Methods:",
            "        TODO: Describe important methods.",
            "    """"
        ]

def add_docstrings(input_file, output_file):
```

The screenshot shows a Jupyter Notebook interface with the following details:

- Header:** Ai ass 9.4.ipynb
- Toolbar:** File Edit View Insert Runtime Tools Help
- Search Bar:** Commands + Code + Text | Run all
- Code Cell:** [27] `dummy_code = ...  
def my_function(arg1, arg2):  
 pass  
class MyClass:  
 def __init__(self):  
 pass  
def another_func():  
 """Existing docstring"""  
 pass  
...  
 with open("input.py", "w") as f:  
 f.write(dummy_code)  
 add_docstrings("input.py", "output.py")  
 print("Documentation scaffolding completed successfully!")  
# Optionally, print the content of the output file  
print("Content of output.py")  
with open("output.py", "r") as f:  
 print(f.read())  
# Clean up the dummy files  
os.remove("input.py")  
os.remove("output.py")`
- Output Cell:** Documentation scaffolding completed successfully!
- Content of output.py:**  
`def my_function(arg1, arg2):  
 """  
 my_function function.`
- Right Panel:** RAM Disk



The screenshot shows a Jupyter Notebook interface with the following details:

- Header:** Ai ass 9.4.ipynb
- Menu Bar:** File, Edit, View, Insert, Runtime, Tools, Help
- Toolbar:** Share, RAM (green checkmark), Disk (green checkmark)
- Code Completion Panel:** Shows code snippets for class and function definitions, including docstring placeholders like "Args:" and "Returns:", each followed by a "TODO: Add parameter descriptions." or "TODO: Add return description." placeholder.
- Code Area:** Displays the actual Python code being typed, which includes the generated docstring snippets.