AI ASSITED CODING

ASSIGNEMNT-9.4

B.shiva sai

2403A51257

BATCH-11

TASK 1-

Scenario: You have been given a Python function without comments. def calculate_discount(price, discount_rate): return price - (price * discount_rate / 100) • Use an AI tool (or manually simulate it) to generate line-by-line comments for the function. • Modify the function so that it includes a docstring in Google-style or NumPy-style format. • Compare the auto-generated comments with your manually written version.

PROMPT-

Add line-by-line comments to the function calculate_discount(price, discount_rate).
Also write a docstring in Google-style or NumPy-style.
Compare AI-generated comments with manual comments.

CODE-

ORIGINAL FUNCTION-

```
Tabnine | Edit | Test | Explain | Document
def calculate_discount(price, discount_rate):
    return price - (price * discount_rate / 100)

# Test
print("Original function:", calculate_discount(200, 10))
```

[11] ✓ 0.0s Python

```
Original function: 180.0
```

AI-Generated Comments (Simulated)-

```
Tabnine | Edit | Test | Explain | Document
def calculate_discount_ai(price, discount_rate):
    # Calculate the discount amount
    # Subtract discount from original price to get final price
    return price - (price * discount_rate / 100)

# Test
print("AI-commented function:", calculate_discount_ai(200, 10))
```

[12] ✓ 0.0s Python

```
AI-commented function: 180.0
```

Manual Comments-

```
Tabnine | Edit | Test | Explain | Document
def calculate_discount_manual(price, discount_rate):
    # price: original price of the product (int or float)
    # discount_rate: discount percentage to apply (0-100)
    # Compute discount amount: price * discount_rate / 100
    # Subtract discount from original price to get final price
    return price - (price * discount_rate / 100)

# Test
print("Manual-commented function:", calculate_discount_manual(200, 10))
```

[13] ✓ 0.0s Python

```
Manual-commented function: 180.0
```

Function with NumPy-Style Docstring

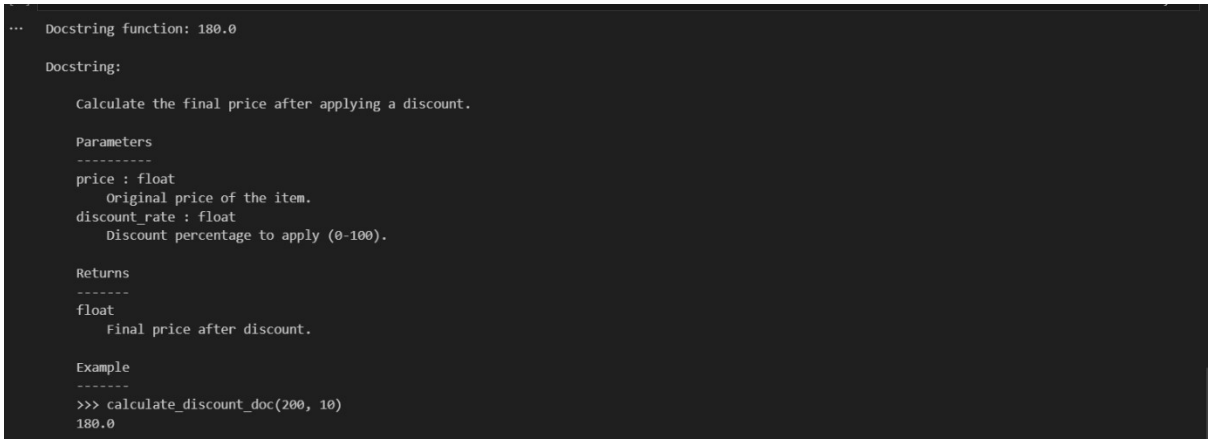
```
Tabnine | Edit | Test | Explain | Document
def calculate_discount_doc(price, discount_rate):
    """
    Calculate the final price after applying a discount.

    Parameters
    ----------
    price : float
        Original price of the item.
    discount_rate : float
        Discount percentage to apply (0-100).

    Returns
    -------
    float
        Final price after discount.

    Example
    -------
    >>> calculate_discount_doc(200, 10)
    180.0
    """
    return price - (price * discount_rate / 100)

# Test
print("Docstring function:", calculate_discount_doc(200, 10))
print("\nDocstring:\n", calculate_discount_doc.__doc__)
```



Comparison

AI comments: short and repeats code, little explanation.

Manual comments: detailed, explain parameters, calculation, and reasoning.

Docstring: structured, includes types, return value, and example usage.

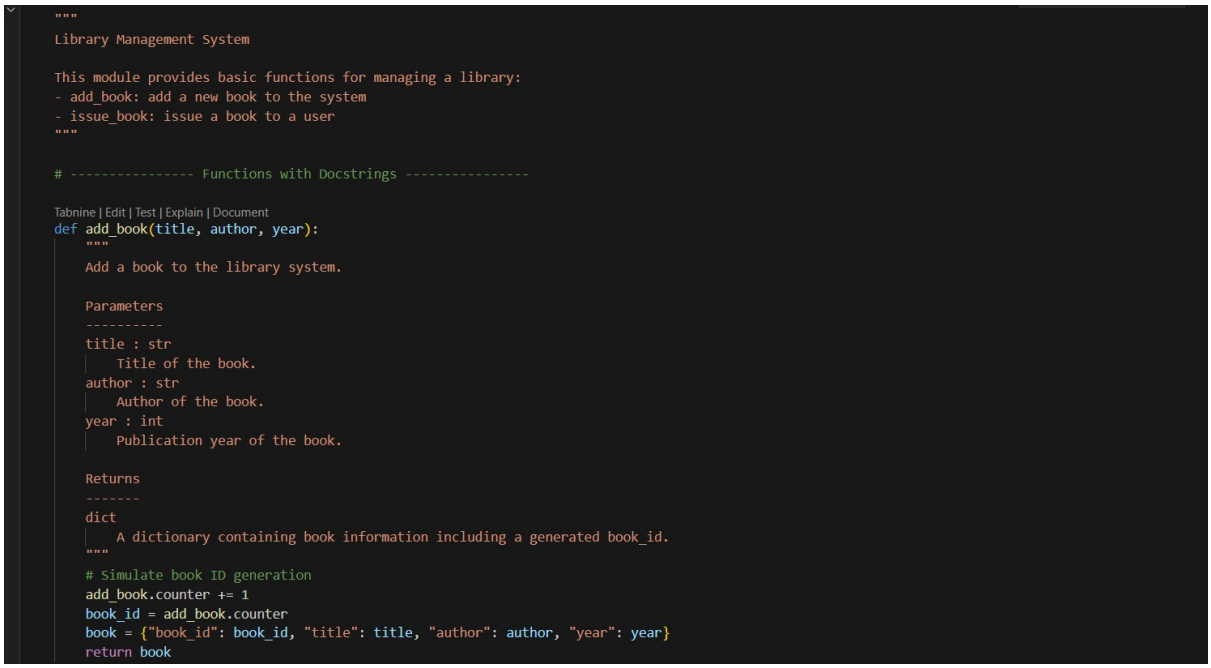
TASK 2-

A team is building a Library Management System with
multiple functions.
def add_book(title, author, year):
# code to add book
pass
def issue_book(book_id, user_id):
# code to issue book
Pass
• Write a Python script that uses docstrings for each function (with input, output, and description)

PROMPT-

Write a Python script for a Library Management System with functions add_book(title, author, year) and issue_book(book_id, user_id).
Add **docstrings** for each function describing input parameters, output, and function purpose.
Include print() statements to show example usage.

CODE-



```
    """
    Library Management System

    This module provides basic functions for managing a library:
    - add_book: add a new book to the system
    - issue_book: issue a book to a user
    """

    # ---------------- Functions with Docstrings ----------------

    Tabnine | Edit | Test | Explain | Document
    def add_book(title, author, year):
        """
        Add a book to the library system.

        Parameters
        ----------
        title : str
            Title of the book.
        author : str
            Author of the book.
        year : int
            Publication year of the book.

        Returns
        -------
        dict
            A dictionary containing book information including a generated book_id.
        """
        # Simulate book ID generation
        add_book.counter += 1
        book_id = add_book.counter
        book = {"book_id": book_id, "title": title, "author": author, "year": year}
        return book
```

```
# Initialize counter
add_book.counter = 0


Tabnine | Edit | Test | Explain | Document
def issue_book(book_id, user_id):
    """
    Issue a book to a user.

    Parameters
    ----------
    book_id : int
        Unique ID of the book.
    user_id : int
        Unique ID of the user.

    Returns
    -------
    str
        Confirmation message indicating the book has been issued.
    """
    return f"Book {book_id} has been issued to user {user_id}."


# ---------------- Main Execution ----------------
if __name__ == "__main__":
    # Add multiple books
    books = []
    books.append(add_book("Python Basics", "John Doe", 2025))
    books.append(add_book("Data Science 101", "Jane Smith", 2024))
    books.append(add_book("Algorithms", "Alice Brown", 2023))

    # Print added books
    print("=== Added Books ===")
    for book in books:
        print(book)
```

```python
# Issue books to users
print("\n=== Issued Books ===")
print(issue_book(books[0]['book_id'], 101))
print(issue_book(books[1]['book_id'], 102))
print(issue_book(books[2]['book_id'], 103))

# Print module docstring
print("\n=== Module Docstring ===")
print(__doc__)

# Print function docstrings
print("\n=== Function Docstrings ===")
print("add_book docstring:\n", add_book.__doc__)
print("issue_book docstring:\n", issue_book.__doc__)
```

✓ 0.0s Python

```
=== Added Books ===
{'book_id': 1, 'title': 'Python Basics', 'author': 'John Doe', 'year': 2025}
{'book_id': 2, 'title': 'Data Science 101', 'author': 'Jane Smith', 'year': 2024}
{'book_id': 3, 'title': 'Algorithms', 'author': 'Alice Brown', 'year': 2023}

=== Issued Books ===
Book 1 has been issued to user 101.
Book 2 has been issued to user 102.
Book 3 has been issued to user 103.

=== Module Docstring ===

Library Management System

This module provides basic functions for managing a library:
- add_book: add a new book to the system
- issue_book: issue a book to a user
```

```
    Returns
    -------
    dict
        A dictionary containing book information including a generated book_id.

issue_book docstring:

    Issue a book to a user.

    Parameters
    ----------
    book_id : int
        Unique ID of the book.
    user_id : int
        Unique ID of the user.

    Returns
    -------
    str
        Confirmation message indicating the book has been issued.
```

TASK 3 -

Scenario: You are reviewing a colleague's codebase containing long functions.

```
def process_sensor_data(data):
cleaned = [x for x in data if x is not None]
avg = sum(cleaned)/len(cleaned)
anomalies = [x for x in cleaned if abs(x - avg) > 10]
```

return {"average": avg, "anomalies": anomalies}

• Generate a summary comment explaining the purpose of the function in 2–3 lines.

• Create a flow-style comment (step-by-step explanation).

• Write a short paragraph of documentation describing possible use cases of this function in real-world scenarios.

PROMPT-

Explain this function in 2–3 lines, add step-by-step comments, write a short real-life use case paragraph, add a Google-style docstring, and include print statements to show cleaned data, average, anomalies, and final result:

```
def process_sensor_data(data):
    cleaned = [x for x in data if x is not None]
    avg = sum(cleaned)/len(cleaned)
    anomalies = [x for x in cleaned if abs(x - avg) > 10]
    return {"average": avg, "anomalies": anomalies}
```

CODE-

```
Tabnine | Edit | Test | Explain | Document
def process_sensor_data(data):
    """
    Process a list of sensor readings to compute the average and detect anomalies.
    Summary:
        Removes None values from the input data, calculates the average of valid readings,
        and identifies readings that deviate from the average by more than 10 units.
    Step-by-Step Flow:
        1. Remove all None values from the input data to retain only valid readings.
        2. Calculate the average of the cleaned readings.
        3. Identify anomalies as readings that differ from the average by more than 10 units.
        4. Print and return the results.
    Args:
        data (list[float | None]): A list of sensor readings, which may include None values.
    Returns:
        dict: A dictionary with keys:
            - "average" (float): The average of valid readings.
            - "anomalies" (list[float]): Readings deviating significantly from the average.
    """
    # Step 1: Remove None values
    cleaned = [x for x in data if x is not None]
    print("Cleaned Data:", cleaned)
    # Step 2: Calculate average
    avg = sum(cleaned) / len(cleaned)
    print("Average Value:", avg)
    # Step 3: Identify anomalies
    anomalies = [x for x in cleaned if abs(x - avg) > 10]
    print("Anomalies Detected:", anomalies)
    # Step 4: Return results
    result = {"average": avg, "anomalies": anomalies}
    print("Result:", result)
    return result
# Example usage
sensor_readings = [12, 15, None, 40, 13, 14, 100]
process_sensor_data(sensor_readings)
```

OUTPUT-

```
Cleaned Data: [12, 15, 40, 13, 14, 100]
Average Value: 32.333333333333336
Anomalies Detected: [12, 15, 13, 14, 100]
Result: {'average': 32.333333333333336, 'anomalies': [12, 15, 13, 14, 100]}

{'average': 32.333333333333336, 'anomalies': [12, 15, 13, 14, 100]}
```