

K. JASHWANTH

2303A52300

Question 1: Auto-Generating Function Documentation in a Shared Codebase

Code Prompt:

Generate python code for Calculator that can perform basic arithmetic operations such as addition, subtraction, multiplication, and division. The calculator should take user input for the operations.

```
# Function to add two numbers
def add(x, y):
    return x + y

# Function to subtract two numbers
def subtract(x, y):
    return x - y

# Function to multiply two numbers
def multiply(x, y):
    return x * y

# Function to divide two numbers
def divide(x, y):
    if y == 0:
        return "Error! Division by zero."
    return x / y

print("Select operation:")
print("1. Add")
print("2. Subtract")
print("3. Multiply")
print("4. Divide")

while True:
    # Take input from the user
    choice = input("Enter choice(1/2/3/4): ")

    # Check if choice is one of the four options
    if choice in ('1', '2', '3', '4'):
        try:
            num1 = float(input("Enter first number: "))
            num2 = float(input("Enter second number: "))
        except ValueError:
            print("Invalid input. Please enter numbers only.")
            continue

        if choice == '1':
            print(num1, "+", num2, "=", add(num1, num2))
        elif choice == '2':
            print(num1, "-", num2, "=", subtract(num1, num2))
        elif choice == '3':
            print(num1, "*", num2, "=", multiply(num1, num2))
        elif choice == '4':
            result = divide(num1, num2)
            if isinstance(result, str):
                print(result)
            else:
                print(num1, "/", num2, "=", result)

        # Ask if the user wants another calculation
        next_calculation = input("Let's do next calculation? (yes/no): ")
        if next_calculation.lower() == "no":
            print("Goodbye!")
            break
    else:
        print("Invalid Input")
```

DocString Prompt:

Generate python code and generate a google style docstring for every function for Calculator that can perform basic arithmetic operations such as addition, subtraction, multiplication, and division. The calculator should take user input for the operations. Provide a clean and efficient code.

```

def add(x, y):
    """Adds two numbers and returns the sum.

    Args:
        x (float): The first number.
        y (float): The second number.

    Returns:
        float: The sum of x and y.
    """
    return x + y

def subtract(x, y):
    """Subtracts the second number from the first and returns the difference.

    Args:
        x (float): The first number.
        y (float): The second number.

    Returns:
        float: The difference between x and y.
    """
    return x - y

def multiply(x, y):
    """Multiplies two numbers and returns the product.

    Args:
        x (float): The first number.

```

```

        y (float): The second number.

    Returns:
        float: The product of x and y.
    """
    return x * y

def divide(x, y):
    """Divides the first number by the second and returns the quotient.

    Args:
        x (float): The numerator.
        y (float): The denominator.

    Returns:
        float or str: The quotient of x and y, or an error message if y is zero.
    """
    if y == 0:
        return "Error: Division by zero."
    return x / y

print("Select operation")
print("1. Add")
print("2. Subtract")
print("3. Multiply")
print("4. Divide")

while True:
    choice = input("Enter choice(1/2/3/4): ")

```

```

choice = input("Enter choice(1/2/3/4): ")

if choice in ('1', '2', '3', '4'):
    try:
        num1 = float(input("Enter first number: "))
        num2 = float(input("Enter second number: "))
    except ValueError:
        print("Invalid input. Please enter numbers only.")
        continue

    if choice == '1':
        print(f"{num1} + {num2} = {add(num1, num2)}")
    elif choice == '2':
        print(f"{num1} - {num2} = {subtract(num1, num2)}")
    elif choice == '3':
        print(f"{num1} * {num2} = {multiply(num1, num2)}")
    elif choice == '4':
        result = divide(num1, num2)
        if isinstance(result, str):
            print(result)
        else:
            print(f"{num1} / {num2} = {result}")

    next_calculation = input("Let's do next calculation? (yes/no): ")
    if next_calculation.lower() == "no":
        print("Goodbye!")
        break
    elif next_calculation.lower() != "yes":
        print("Invalid input. Continuing with new calculation.")
    else:
        print("Invalid input")

```

Output:

```

... Select operation:
1. Add
2. Subtract
3. Multiply
4. Divide
Enter choice(1/2/3/4): 1
Enter first number: 2
Enter second number: 3
2.0 + 3.0 = 5.0
Let's do next calculation? (yes/no): no
Goodbye!

```

JUSTIFICATION:

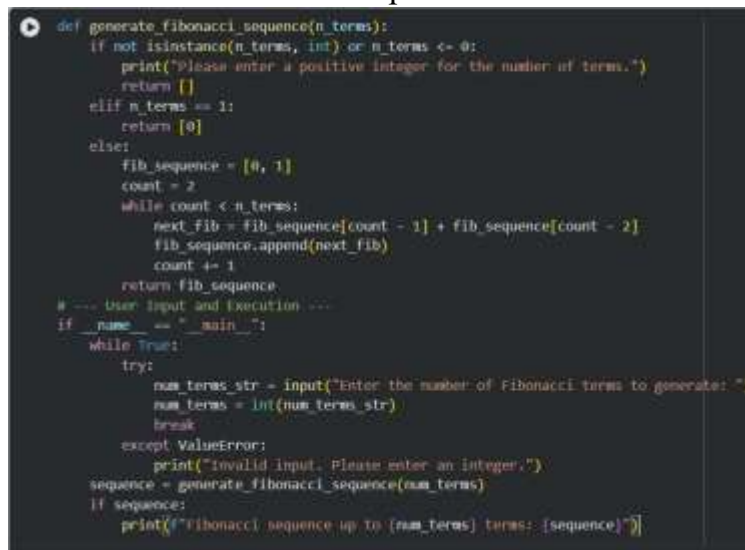
Using Google-style docstrings makes the calculator code easy to read and understand by clearly explaining each function's purpose, inputs, and outputs. This helps developers quickly learn the code without confusion. It also improves teamwork in a shared codebase by making maintenance and updates simpler. Well-documented functions are easier to test and debug. Overall, documentation ensures clean, organized, and professional code.

Question 2: Enhancing Readability Through AI-Generated Inline

Comments

Code Prompt:

Generate a python code for Fibonacci sequence which contains complex logic that works correctly. The Python script should contain loops, Conditional logic and algorithms. It should take user inputs for the code.

A screenshot of a code editor showing a Python script. The script defines a function 'generate_fibonacci_sequence(n_terms)' which handles various input conditions: non-integer or non-positive values, zero, one, and two or more terms. It uses a while loop to calculate the sequence. Below the function, there's a main execution block that prompts the user for input, handles exceptions for non-integer inputs, and prints the resulting Fibonacci sequence.

```
def generate_fibonacci_sequence(n_terms):  
    if not isinstance(n_terms, int) or n_terms <= 0:  
        print(">Please enter a positive integer for the number of terms.")  
        return []  
    elif n_terms == 1:  
        return [0]  
    else:  
        fib_sequence = [0, 1]  
        count = 2  
        while count <= n_terms:  
            next_fib = fib_sequence[count - 1] + fib_sequence[count - 2]  
            fib_sequence.append(next_fib)  
            count += 1  
        return fib_sequence  
  
# --- User Input and Execution ---  
if __name__ == "__main__":  
    while True:  
        try:  
            num_terms_str = input("Enter the number of Fibonacci terms to generate: ")  
            num_terms = int(num_terms_str)  
            break  
        except ValueError:  
            print("Invalid input. Please enter an integer.")  
    sequence = generate_fibonacci_sequence(num_terms)  
    if sequence:  
        print(f"Fibonacci sequence up to {num_terms} terms: {sequence}")
```

DocString Prompt:

Generate a python code and docstring for every function for Fibonacci sequence which contains complex logic that works correctly. Insert inline comments only for complex or non-obvious logic. Avoid commenting on trivial or self explanatory syntax. The Python script should contain loops, Conditional logic and algorithms. It should take user inputs for the code. The goal is to improve clarity without cluttering the code.

```
def generate_fibonacci_sequence(n_terms):
    """Generates the Fibonacci sequence up to a specified number of terms.

    The Fibonacci sequence is defined by the rule that each number is the
    sum of the two preceding ones, typically starting with 0 and 1.

    Args:
        n_terms (int): The number of terms to generate in the Fibonacci sequence.
            Must be a positive integer.

    Returns:
        List: A list containing the Fibonacci sequence up to 'n_terms'.
            Returns an empty list if 'n_terms' is not a positive integer,
            or [0] if 'n_terms' is 1.

    """
    # Validate input to ensure 'n_terms' is a positive integer.
    if not isinstance(n_terms, int) or n_terms <= 0:
        print("Error: Please enter a positive integer for the number of terms.")
        return []
    # Handle the base case for 1 term.
    elif n_terms == 1:
        return [0]
    # Handle the general case for 2 or more terms.
    else:
        # Initialize the sequence with the first two Fibonacci numbers.
        fib_sequence = [0, 1]
        count = 2 # 'count' tracks the number of terms generated so far.
        while count <= n_terms:
            # Calculate the next Fibonacci number by summing the last two.
            next_fib = fib_sequence[count - 1] + fib_sequence[count - 2]
            fib_sequence.append(next_fib)
            count += 1
        return fib_sequence

# --- User Input and Execution Logic ---
if __name__ == "__main__":
    while True:
        try:
            num_terms_str = input("Enter the number of Fibonacci terms to generate: ")
            num_terms = int(num_terms_str)
            break # A full loop if input is a valid integer.
        except ValueError:
            print("Invalid input. Please enter an integer.")

    sequence = generate_fibonacci_sequence(num_terms)

    # Only print the sequence if it's not empty (i.e., input was valid).
    if sequence:
        print(f"Fibonacci sequence up to {num_terms} terms: {sequence}")
```

```
def generate_fibonacci_sequence(n_terms):
    """Generates the Fibonacci sequence up to a specified number of terms.

    The Fibonacci sequence is defined by the rule that each number is the
    sum of the two preceding ones, typically starting with 0 and 1.

    Args:
        n_terms (int): The number of terms to generate in the Fibonacci sequence.
            Must be a positive integer.

    Returns:
        List: A list containing the Fibonacci sequence up to 'n_terms'.
            Returns an empty list if 'n_terms' is not a positive integer,
            or [0] if 'n_terms' is 1.

    """
    # Validate input to ensure 'n_terms' is a positive integer.
    if not isinstance(n_terms, int) or n_terms <= 0:
        print("Error: Please enter a positive integer for the number of terms.")
        return []
    # Handle the base case for 1 term.
    elif n_terms == 1:
        return [0]
    # Handle the general case for 2 or more terms.
    else:
        # Initialize the sequence with the first two Fibonacci numbers.
        fib_sequence = [0, 1]
        count = 2 # 'count' tracks the number of terms generated so far.
        while count <= n_terms:
            # Calculate the next Fibonacci number by summing the last two.
            next_fib = fib_sequence[count - 1] + fib_sequence[count - 2]
            fib_sequence.append(next_fib)
            count += 1
        return fib_sequence

# --- User Input and Execution Logic ---
if __name__ == "__main__":
    while True:
        try:
            num_terms_str = input("Enter the number of Fibonacci terms to generate: ")
            num_terms = int(num_terms_str)
            break # A full loop if input is a valid integer.
        except ValueError:
            print("Invalid input. Please enter an integer.")

    sequence = generate_fibonacci_sequence(num_terms)

    # Only print the sequence if it's not empty (i.e., input was valid).
    if sequence:
        print(f"Fibonacci sequence up to {num_terms} terms: {sequence}")
```

Output:

```
... Enter the number of Fibonacci terms to generate: 4
Fibonacci sequence up to 4 terms: [0, 1, 1, 2]
```

JUSTIFICATION:

Adding clear docstrings and selective inline comments improves the readability of the Fibonacci program without making the code cluttered. The documentation explains each function's purpose, inputs, and outputs, helping users quickly understand the structure. Inline comments are added only for complex logic, making difficult parts easier to follow while avoiding unnecessary explanations. This keeps the code clean yet informative for both beginners and collaborators. Overall, it enhances clarity, maintainability, and efficient learning.

Question 3: Generating Module-Level Documentation for a Python

Package

Code Prompt:

Generate a python code for Banking system which includes multiple functions like creating account, deposit, withdraw, check balance, transaction history. Take user inputs for the system. generate a minimal code.

```

import datetime

class BankAccount:
    def __init__(self, account_number, owner_name, initial_balance=0.0):
        self.account_number = account_number
        self.owner_name = owner_name
        self.balance = initial_balance
        self.transactions = []
        self.record_transaction("Initial deposit", initial_balance)

    def record_transaction(self, type, amount):
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        self.transactions.append({"timestamp": timestamp, "type": type, "amount": amount, "balance": self.balance})

    def deposit(self, amount):
        if amount <= 0:
            print("Deposit amount must be positive.")
            return False
        self.balance += amount
        self.record_transaction("Deposit", amount)
        print(f"Deposited ${amount:.2f}. New balance: ${self.balance:.2f}")
        return True

    def withdraw(self, amount):
        if amount <= 0:
            print("Withdrawal amount must be positive.")
            return False
        if self.balance < amount:
            print("Insufficient funds.")

```

```

        return False
        self.balance -= amount
        self.record_transaction("Withdrawal", -amount) # Record as negative for withdrawal
        print(f"Withdrew ${amount:.2f}. New balance: ${self.balance:.2f}")
        return True

    def get_balance(self):
        return self.balance

    def get_transaction_history(self):
        return self.transactions

# Dictionary to store all bank accounts
accounts = {}

def create_account():
    account_number = input("Enter new account number: ")
    if account_number in accounts:
        print("Account number already exists.")
        return
    owner_name = input("Enter account owner's name: ")
    try:
        initial_deposit = float(input("Enter initial deposit amount: "))
        if initial_deposit <= 0:
            print("Initial deposit cannot be negative. Setting to 0.")
            initial_deposit = 0.0
        except ValueError:
            print("Invalid amount. Setting initial deposit to 0.")
            initial_deposit = 0.0

```

```

    except ValueError:
        print("Invalid amount. Setting initial deposit to 0.")
        initial_deposit = 0.0

    new_account = BankAccount(account_number, owner_name, initial_deposit)
    accounts[account_number] = new_account
    print(f"Account {account_number} created for {owner_name} with initial balance ${initial_deposit:.2f}.")

def get_account_by_number(account_number):
    return accounts.get(account_number)

def perform_deposit():
    account_number = input("Enter account number: ")
    account = get_account_by_number(account_number)
    if not account:
        print("Account not found.")
        return
    try:
        amount = float(input("Enter amount to deposit: "))
        account.deposit(amount)
    except ValueError:
        print("Invalid amount. Please enter a number.")

def perform_withdraw():
    account_number = input("Enter account number: ")
    account = get_account_by_number(account_number)
    if not account:
        print("Account not found.")
        return

```

```

    try:
        amount = float(input("Enter amount to withdraw: "))
        account.withdraw(amount)
    except ValueError:
        print("Invalid amount. Please enter a number.")

def list_accounts():
    account_number = input("Enter account number: ")
    account = get_account_by_number(account_number)
    if not account:
        print("Account not found.")
        return
    print(f"Account {account_number} balance: ${account.get_balance():.2f}")

def view_transaction():
    account_number = input("Enter account number: ")
    account = get_account_by_number(account_number)
    if not account:
        print("Account not found.")
        return
    history = account.get_transaction_history()
    if not history:
        print("No transaction found for this account.")
        return
    print(f"> Transaction history for account {account_number} <")
    for transaction in history:
        print(f"Timestamp: {transaction['timestamp']} | Type: {transaction['type']} | Amount: {transaction['amount']:.2f} | Balance after: {transaction['balance']:.2f}")
    print("-----")

```

```

def main_menu():
    while True:
        print("\n--- Banking System Menu ---")
        print("1. Create Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Check Balance")
        print("5. View Transaction History")
        print("6. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            create_account()
        elif choice == '2':
            perform_deposit()
        elif choice == '3':
            perform_withdraw()
        elif choice == '4':
            check_balance()
        elif choice == '5':
            view_transactions()
        elif choice == '6':
            print("Exiting banking system. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main_menu()

```

DocString Prompt:

Generate the same python code for Banking system which includes multiple functions like creating account, deposit, withdraw, check balance, transaction history. Generate a module level docstring at the top of the file that includes the purpose of the module, required libraries or dependencies, a brief description of key functions and classes and a short example of how the module can be used. Focus on clarity and professional tone. Take user inputs for the system. generate a minimal code.

```
"""A simple banking system module for managing bank accounts and transactions.

This module provides functionality to create bank accounts, perform deposits and withdrawals,
check account balances, and view transaction histories. It uses a dictionary to store
BankAccount objects, identified by unique account numbers, and offers a menu-driven
interface for user interaction.

Dependencies:
- datetime: Standard library for timestamping transactions.

Classes:
- BankAccount: Manages individual bank account details, balance, and transactions.

Functions:
- create_account(): Guides the user through creating a new bank account.
- get_account_by_number(account_number): Retrieves a BankAccount object by its number.
- perform_deposit(): Handles depositing funds into an existing account.
- perform_withdrawal(): Handles withdrawing funds from an existing account.
- check_balance(): Displays the current balance of a specified account.
- view_transactions(): Shows the transaction history for a specified account.
- main_menu(): Presents the main interactive menu for the banking system.

Usage Example:
To run the banking system, execute this script directly. The main_menu()
function will be called, presenting a series of options to interact with
the bank accounts.

Example Interaction:
"""
```

```
Example Interaction:

--- Banking System Menu ---
1. Create Account
...

Enter your choice: 1
Enter new account number: 1001
Enter account owner's name: Alice
Enter initial deposit amount: 500
Account 1001 created for Alice with initial balance $500.00.

...

import datetime

class BankAccount:
    def __init__(self, account_number, owner_name, initial_balance=0.0):
        self.account_number = account_number
        self.owner_name = owner_name
        self.balance = initial_balance
        self.transactions = []
        self._record_transaction("Initial Deposit", initial_balance)

    def _record_transaction(self, type, amount):
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        self.transactions.append({"timestamp": timestamp, "type": type, "amount": amount, "balance": self.balance})

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit amount must be positive.")
```

```
        self.balance += amount
        self._record_transaction("Deposit", amount)
        print(f"Deposited ${amount:.2f}. New balance: ${self.balance:.2f}")
        return True

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive.")
        if self.balance < amount:
            raise ValueError("Insufficient funds.")
        self.balance -= amount
        self._record_transaction("Withdrawal", -amount) # Record as negative for withdrawal
        print(f"Withdrew ${amount:.2f}. New balance: ${self.balance:.2f}")
        return True

    def get_balance(self):
        return self.balance

    def get_transaction_history(self):
        return self.transactions

# Dictionary to store all bank accounts
accounts = {}

def create_account():
    account_number = input("Enter new account number: ")
    if account_number in accounts:
        print("Account number already exists.")
```

```
    try:
        initial_deposit = float(input("Enter initial deposit amount: "))
        if initial_deposit < 0:
            print("Initial deposit cannot be negative. Setting to 0.")
            initial_deposit = 0.0
        except ValueError:
            print("Invalid amount. Setting initial deposit to 0.")
            initial_deposit = 0.0

    new_account = BankAccount(account_number, owner_name, initial_deposit)
    accounts[account_number] = new_account
    print(f"Account {account_number} created for {owner_name} with initial balance ${initial_deposit:.2f}."

    def get_account_by_number(account_number):
        return accounts.get(account_number)

    def perform_deposit():
        account_number = input("Enter account number: ")
        account = get_account_by_number(account_number)
        if not account:
            print("Account not found.")
            return
        try:
            amount = float(input("Enter amount to deposit: "))
            account.deposit(amount)
        except ValueError:
            print("Invalid amount. Please enter a number.")

    def perform_withdrawal():
        account_number = input("Enter account number: ")
```

```
    try:
        amount = float(input("Enter amount to withdraw: "))
        account.withdraw(amount)
    except ValueError:
        print("Invalid amount. Please enter a number.")

    def check_balance():
        account_number = input("Enter account number: ")
        account = get_account_by_number(account_number)
        if not account:
            print("Account not found.")
            return
        print(f"Account {account_number} balance: ${account.get_balance():.2f}")

    def view_transactions():
        account_number = input("Enter account number: ")
        account = get_account_by_number(account_number)
        if not account:
            print("Account not found.")
            return
        history = account.get_transaction_history()
        if not history:
            print("No transactions found for this account.")
            return
        print(f"Transaction history for account {account_number} ---")
        for transaction in history:
            print(f"Timestamp: {transaction['timestamp']}, Type: {transaction['type']}, Amount: {transaction['amount']:.2f}, Balance After: {transaction['balance']:.2f}")
        print()
```

```

def main_menu():
    while True:
        print("\n--- Banking System Menu ---")
        print("1. Create Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Check Balance")
        print("5. View Transaction History")
        print("6. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            create_account()
        elif choice == '2':
            perform_deposit()
        elif choice == '3':
            perform_withdraw()
        elif choice == '4':
            check_balance()
        elif choice == '5':
            view_transactions()
        elif choice == '6':
            print("Exiting banking system. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main_menu()

```

Output:

```

--- Banking System Menu ---
1. Create Account
2. Deposit
3. Withdraw
4. Check Balance
5. View Transaction History
6. Exit
Enter your choice: 1
Enter new account number: 123456
Enter account owner's name: vicky
Enter initial deposit amount: 500
Account 123456 created for vicky with initial balance $500.00.

--- Banking System Menu ---
1. Create Account
2. Deposit
3. Withdraw
4. Check Balance
5. View Transaction History
6. Exit
Enter your choice: 2
Enter account number: 123456
Enter amount to deposit: 1000
Deposited $1000.00. New balance: $1500.00

--- Banking System Menu ---
1. Create Account
2. Deposit
3. Withdraw
4. Check Balance
5. View Transaction History
6. Exit
Enter your choice: 3

```

```

5. View Transaction History
6. Exit
Enter your choice: 3
Enter account number: 123456
Enter amount to withdraw: 100
Withdraw $100.00. New balance: $1400.00

--- Banking System Menu ---
1. Create Account
2. Deposit
3. Withdraw
4. Check Balance
5. View Transaction History
6. Exit
Enter your choice: 5
Enter account number: 123456

--- Transaction History for Account 123456 ---
Timestamp: 2026-02-05 15:22:42, Type: Initial Deposit, Amount: $500.00, Balance after: $500.00
Timestamp: 2026-02-05 15:22:53, Type: Deposit, Amount: $1000.00, Balance after: $1500.00
Timestamp: 2026-02-05 15:23:20, Type: Withdrawal, Amount: $-100.00, Balance after: $1400.00
-----

--- Banking System Menu ---
1. Create Account
2. Deposit
3. Withdraw
4. Check Balance
5. View Transaction History
6. Exit
Enter your choice: 6
Exiting banking system. Goodbye!

```

JUSTIFICATION:

Adding a module-level docstring provides a clear overview of the banking system's purpose, dependencies, and main features, helping users understand the program before reading the code. It explains key functions and how they interact, making the structure easier to follow. The usage example guides new users on how to run the system correctly. This improves readability and reduces onboarding time in a shared project. Overall, it ensures the code is organized, professional, and easy to maintain.

Question 4: Converting Developer Comments into Structured Docstrings

Code Prompt:

Generate a python code for student grade system which includes multiple functions like marks, calculated_total, calculate_average, find_grade, find_highest, find_lowest, display_report with long explanatory comments. Take user inputs for the code. Generate a minimal code.

```
# A dictionary to store student data. The key will be the student's name (string),
# and the value will be another dictionary containing their marks for different subjects.
# Example: {'Alice': {'math': 88, 'science': 92, 'art': {'music': 85, 'dance': 80}}
student_data = {}

def add_student_marks():
    """Allows the user to input marks for a new or existing student.
    Each subject's mark is taken as input. If a student already exists,
    their marks will be updated or new subjects added.
    """
    student_name = input("Enter student's name: ").strip().capitalize()
    if student_name not in student_data:
        student_data[student_name] = {}
        print(f"New student '{student_name}' added.")
    else:
        print(f"Updating marks for existing student '{student_name}'.")

    while True:
        subject = input("Enter subject name for {student_name} (or 'done' to finish): ").strip().capitalize()
        if subject.lower() == "done":
            break
        try:
            marks = float(input(f"Enter marks for {subject} (out of 100): "))
            if 0 <= marks <= 100:
                student_data[student_name][subject] = marks
                print(f"Marks for {subject} added/updated for {student_name}.")
            else:
                print("Marks must be between 0 and 100. Please try again.")
        except ValueError:
            print("Invalid input. Please enter a number between 0 and 100.")

def calculate_total_marks(student_name):
    """Calculates the total marks for a given student across all subjects.

    Args:
        student_name (str): The name of the student.

    Returns:
        float: The sum of all marks for the student, or 0.0 if no marks are found.
    """
    if student_name in student_data and student_data[student_name]:
        # Sums all the values (marks) in the student's marks dictionary.
        return sum(student_data[student_name].values())
    return 0.0

def calculate_average_marks(student_name):
    """Calculates the average marks for a given student.

    Args:
        student_name (str): The name of the student.

    Returns:
        float: The average mark for the student, or 0.0 if no marks are found.
    """
    if student_name in student_data and student_data[student_name]:
        total = calculate_total_marks(student_name)
        # Divides the total marks by the number of subjects to get the average.
        return total / len(student_data[student_name])
    return 0.0

def find_grade(average_marks):
    """Determines the letter grade based on the student's average marks.

    Args:
        average_marks (float): The average marks of the student.

    Returns:
        str: The letter grade (A, B, C, D, F).
    """
    if average_marks >= 90:
        return 'A'
    elif average_marks >= 80:
        return 'B'
    elif average_marks >= 70:
        return 'C'
    elif average_marks >= 60:
        return 'D'
    else:
        return 'F'

def find_highest_mark(student_name):
    """Finds the highest mark achieved by a student in any subject.

    Args:
        student_name (str): The name of the student.

    Returns:
        float: The highest mark, or None if no marks are available.
    """
    if student_name in student_data and student_data[student_name]:
        # Uses the max() function on the values (marks) of the student's dictionary.
        return max(student_data[student_name].values())
    return None

def find_lowest_mark(student_name):
    """Finds the lowest mark achieved by a student in any subject.

    Args:
        student_name (str): The name of the student.

    Returns:
        float: The lowest mark, or None if no marks are available.
    """
    if student_name in student_data and student_data[student_name]:
        # Uses the min() function on the values (marks) of the student's dictionary.
        return min(student_data[student_name].values())
    return None

def display_report(student_name):
    """Generates and displays a detailed report for a single student.
    Includes total marks, average, grade, highest, and lowest marks.

    Args:
        student_name (str): The name of the student.
    """
    if student_name not in student_data:
        print(f"Report: Student '{student_name}' not found in the system.")
        return

    total = calculate_total_marks(student_name)
    average = calculate_average_marks(student_name)
    grade = find_grade(average)
    highest = find_highest_mark(student_name)
    lowest = find_lowest_mark(student_name)

    print(f"Student Report for {student_name} ---")
    if not student_data[student_name]:
        print("No marks entered for this student yet.")
        print("-----")
        return

    # Display individual subject marks
    print("Subject Marks:")
    for subject, marks in student_data[student_name].items():
        print(f"  {subject}: {marks:.2f}")

    print(f"Total Marks: {total:.2f}")
    print(f"Average Marks: {average:.2f}")
    print(f"Grade: {grade}")
    print(f"Highest Mark: {highest:.2f}")
    print(f"Lowest Mark: {lowest:.2f}")
    print("-----")

def main_menu():
    """Presents the main interactive menu for the student grade system.
    Allows the user to choose between adding marks, viewing a report, or exiting.
    """
    while True:
        print("\n--- Student Grade System Menu ---")
        print("1. Add/Update Student Marks")
        print("2. View Student Report")
        print("3. Exit")
        choice = input("Enter your choice: ").strip()

        if choice == "1":
            add_student_marks()
        elif choice == "2":
            student_name_report = input("Enter the name of the student for report: ").strip().capitalize()
            display_report(student_name_report)
        elif choice == "3":
            print("Exiting Student Grade System. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")

# Ensure that the main_menu() function is called when the script is executed directly.
if __name__ == "__main__":
    main_menu()
```


DocString Prompt:

Generate the same python code for student grade system which includes multiple functions like marks, calculated_total, calculate_average, find_grade, find_highest, find_lowest, display_report with proper docstrings. Convert the comments into structured google style or numpy style docstrings. Remove redundant inline comments after conversion. Take user inputs for the code. Generate a minimal and standardize documentation.

```
"""A Dictionary to store student data.
Args:
    student_name (str):
    A Value Dictionary of subject marks (e.g., {'Math': 85, 'Science': 88})
student_data = {}

def add_student_marks():
    """Adds or updates marks for a student in various subjects.

    Prompts the user for a student's name. If the student is new, an entry is created.
    Then, continuously prompts for subject names and marks until 'done' is received.
    Marks are validated to be between 0 and 100.
    """
    student_name = input("Enter student's name: ").strip().capitalize()
    if student_name not in student_data:
        student_data[student_name] = {}
        print(f"New student: '{student_name}' added.")
    else:
        print(f"Updating marks for existing student: '{student_name}'")

    while True:
        subject = input("Enter subject name for {student_name} (or 'done' to finish): ").strip().capitalize()
        if subject.lower() == "done":
            break
        try:
            marks = float(input(f"Enter marks for {subject} (out of 100): "))
            if 0 <= marks <= 100:
                student_data[student_name][subject] = marks
                print(f"Marks for {subject} added/updated for {student_name}.")
            else:
                print(f"Invalid marks for {subject}. Please enter a value between 0 and 100.")
        except ValueError:
            print(f"Invalid input for {subject}. Please enter a numeric value between 0 and 100.")
```

```
def calculate_total_marks(student_name):
    """Calculates the total marks for a given student across all subjects.

    Args:
        student_name (str): The name of the student.

    Returns:
        float: The sum of all marks for the student, or 0.0 if no marks are found.
    """
    if student_name in student_data and student_data[student_name]:
        return sum(student_data[student_name].values())
    return 0.0

def calculate_average_marks(student_name):
    """Calculates the average marks for a given student.

    Args:
        student_name (str): The name of the student.

    Returns:
        float: The average mark for the student, or 0.0 if no marks are found.
    """
    if student_name in student_data and student_data[student_name]:
        total = calculate_total_marks(student_name)
        return total / len(student_data[student_name])
    return 0.0

def find_grade(average_marks):
    """Determines the letter grade based on the student's average marks.
```

```
print(f"Subject Marks:")
for subject, marks in student_data[student_name].items():
    print(f"    {subject}: {marks:.2f}")

total = calculate_total_marks(student_name)
average = calculate_average_marks(student_name)
grade = find_grade(average)
highest = find_highest_mark(student_name)
lowest = find_lowest_mark(student_name)

print(f"Total Marks: {total:.2f}")
print(f"Average Marks: {average:.2f}")
print(f"Grade: {grade}")
print(f"Highest Mark: {highest:.2f}")
print(f"Lowest Mark: {lowest:.2f}")
print(f"-----")

def main_menu():
    """Presents the main interactive menu for the student grade system.

    Allows the user to choose between adding marks, viewing a report, or exiting.
    """
    while True:
        print("\n--- Student Grade System Menu ---")
        print("1. Add/Update Student Marks")
        print("2. View Student Report")
        print("3. Exit")
        choice = input("Enter your choice: ").strip()
```

```
"""Finds the lowest mark achieved by a student in any subject.

Args:
    student_name (str): The name of the student.

Returns:
    float: The lowest mark, or None if no marks are available.
"""
if student_name in student_data and student_data[student_name]:
    return min(student_data[student_name].values())
return None

def display_report(student_name):
    """Generates and displays a detailed report for a single student.

    Includes total marks, average, grade, highest, and lowest marks.

    Args:
        student_name (str): The name of the student.
    """
    if student_name not in student_data:
        print(f"Report: Student '{student_name}' not found in the system.")
        return

    print(f"\n--- Student Report for {student_name} ---")
    if not student_data[student_name]:
        print(f"No marks entered for this student yet.")
    else:
        print(f"Total Marks: {calculate_total_marks(student_name):.2f}")
        print(f"Average Marks: {calculate_average_marks(student_name):.2f}")
        print(f"Grade: {find_grade(calculate_average_marks(student_name))}")
        print(f"Highest Mark: {find_highest_mark(student_name):.2f}")
        print(f"Lowest Mark: {find_lowest_mark(student_name):.2f}")
```

```
def find_highest_mark(student_name):
    """Finds the highest mark achieved by a student in any subject.

    Args:
        student_name (str): The name of the student.

    Returns:
        float: The highest mark, or None if no marks are available.
    """
    if student_name in student_data and student_data[student_name]:
        return max(student_data[student_name].values())
    return None

def find_lowest_mark(student_name):
    """Finds the lowest mark achieved by a student in any subject.

    Args:
        student_name (str): The name of the student.

    Returns:
        float: The lowest mark, or None if no marks are available.
    """
    if student_name in student_data and student_data[student_name]:
        return min(student_data[student_name].values())
    return None

def calculate_total_marks(student_name):
    """Calculates the total marks for a given student across all subjects.

    Args:
        student_name (str): The name of the student.

    Returns:
        float: The sum of all marks for the student, or 0.0 if no marks are found.
    """
    if student_name in student_data and student_data[student_name]:
        return sum(student_data[student_name].values())
    return 0.0

def calculate_average_marks(student_name):
    """Calculates the average marks for a given student.

    Args:
        student_name (str): The name of the student.

    Returns:
        float: The average mark for the student, or 0.0 if no marks are found.
    """
    if student_name in student_data and student_data[student_name]:
        total = calculate_total_marks(student_name)
        return total / len(student_data[student_name])
    return 0.0

def find_grade(average_marks):
    """Determines the letter grade based on the student's average marks.

    Returns:
        str: The letter grade (A, B, C, D, F).
    """
    if average_marks >= 90:
        return 'A'
    elif average_marks >= 80:
        return 'B'
    elif average_marks >= 70:
        return 'C'
    elif average_marks >= 60:
        return 'D'
    else:
        return 'F'
```

```
print(f"Highest Mark: {highest:.2f}")
print(f"Lowest Mark: {lowest:.2f}")
print(f"-----")

def main_menu():
    """Presents the main interactive menu for the student grade system.

    Allows the user to choose between adding marks, viewing a report, or exiting.
    """
    while True:
        print("\n--- Student Grade System Menu ---")
        print("1. Add/Update Student Marks")
        print("2. View Student Report")
        print("3. Exit")
        choice = input("Enter your choice: ").strip()

        if choice == "1":
            add_student_marks()
        elif choice == "2":
            student_name_report = input("Enter the name of the student for report: ").strip().capitalize()
            display_report(student_name_report)
        elif choice == "3":
            print("Exiting Student Grade System. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main_menu()
```

Output:

```
--- Student Grade System Menu ---
1. Add/update Student Marks
2. View Student Report
3. Exit
Enter your choice: 1
Enter student's name: vicky
New student 'Vicky' added.
Enter subject name for Vicky (or 'done' to finish): maths
Enter marks for Maths (out of 100): 90
Marks for Maths added/updated for Vicky.
Enter subject name for Vicky (or 'done' to finish): done

--- Student Grade System Menu ---
1. Add/update Student Marks
2. View Student Report
3. Exit
Enter your choice: 2
Enter the name of the student for report: vicky

--- Student Report for Vicky ---
Subject Marks:
- Maths: 90.00

Total Marks: 90.00
Average Marks: 90.00
Grade: A
Highest Mark: 90.00
Lowest Mark: 90.00
-----

--- Student Grade System Menu ---
1. Add/update Student Marks
```

JUSTIFICATION:

Converting long developer comments into structured docstrings makes the student grade system more organized and professional. Docstrings clearly describe each function's purpose, inputs, and outputs in a standard format, improving readability. Removing redundant inline comments keeps the code clean and less cluttered. This helps other developers quickly understand and reuse the functions. Overall, standardized documentation improves maintainability and consistency across the project.

Question 5: Building a Mini Automatic Documentation Generator

Code Prompt:

Generate a python code in a .py file for a To-Do List Manager which includes multiple utility functions like add_task, remove_task, update_task. Take user inputs for the code. Generate a minimal code.

```
"""Initializes an empty list to store tasks"""
tasks = []

def display_tasks():
    """Displays all current tasks with their indices."""
    if not tasks:
        print("Your to-do list is empty!")
        return
    print("\n--- Your to-do list ---")
    for i, task in enumerate(tasks):
        print(f"{i + 1}. {task}")
    print("-----")

def add_task():
    """Prompts the user to enter a new task and adds it to the list."""
    task = input("Enter the new task: ").strip()
    if task:
        tasks.append(task)
        print(f"Task '{task}' added.")
    else:
        print("Task cannot be empty.")

def remove_task():
    """Prompts the user for a task index to remove and removes the task."""
    display_tasks()
    if not tasks:
        return
    try:
```

```
        print("Invalid task number.")
    except ValueError:
        print("Invalid input. Please enter a number.")

def update_task():
    """Prompts the user for a task index to update and modifies the task."""
    display_tasks()
    if not tasks:
        return
    try:
        task_index = int(input("Enter the number of the task to update: ")) - 1
        if 0 <= task_index < len(tasks):
            new_task = input("Enter the new description for task '{tasks[task_index]}': ").strip()
            if new_task:
                tasks[task_index] = new_task
                print("Task updated successfully.")
            else:
                print("Task description cannot be empty. Task not updated.")
        else:
            print("Invalid task number.")
    except ValueError:
        print("Invalid input. Please enter a number.")

def main_menu():
    """Displays the main menu and handles user choices for task management."""
    while True:
        print("\n--- To-Do List Manager ---")
        print("1. View Tasks")
        print("2. Add Task")
        print("3. Remove Task")
        print("4. Update Task")
        print("5. Exit")
        choice = input("Enter your choice: ")
        if choice == "1":
            display_tasks()
        elif choice == "2":
            add_task()
        elif choice == "3":
            remove_task()
        elif choice == "4":
            update_task()
        elif choice == "5":
            break
```

```

print("Invalid input. Please enter a number.")

def main_menu():
    """Displays the main menu and handles user choices for task management."""
    while True:
        print("\n--- To-Do List Manager ---")
        print("1. View Tasks")
        print("2. Add Task")
        print("3. Update Task")
        print("4. Remove Task")
        print("5. Exit")
        choice = input("Enter your choice: ").strip()

        if choice == '1':
            display_tasks()
        elif choice == '2':
            add_task()
        elif choice == '3':
            update_task()
        elif choice == '4':
            remove_task()
        elif choice == '5':
            print("Exiting To-Do List Manager. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main_menu()

```

DocString Prompt:

Generate the same python code with proper docstrings for a To-Do List Manager which includes multiple utility functions like `add_task`, `remove_task`, `update_task`. Take user inputs for the code. The task is to detect functions and classes and insert placeholder google style docstrings for each detected function or class. The goal is documentation scaffolding. Generate a minimal and efficient code.

```

import ast
import os
import datetime

# --- To-Do List Manager Components ---

tasks = []

def display_tasks():
    """Displays all current tasks with their indices.

    This function iterates through the global 'tasks' list and prints each task prefixed with its 1-based index. If the list is empty, it informs the user.
    """
    if not tasks:
        print("Your to-do list is empty!")
        return
    print("\n--- Your To-Do List ---")
    for i, task in enumerate(tasks):
        print(f"{i + 1}. {task}")
    print(".....")

def add_task():
    """Prompts the user to enter a new task and adds it to the list.

    The function takes input from the user, strips whitespace, and if the input is not empty, appends it to the global 'tasks' list.
    """

```

```

def remove_task():
    """Prompts the user for a task index to remove and removes the task.

    First, it displays the current tasks. Then, it prompts the user for a 1-based index. If the index is valid, the corresponding task is removed from the global 'tasks' list. Handles 'ValueError' for non-integer input and 'IndexError' for out-of-bounds indices.
    """
    display_tasks()
    if not tasks:
        return
    try:
        task_index = int(input("Enter the number of the task to remove: ")) - 1
        if 0 <= task_index < len(tasks):
            removed_task = tasks.pop(task_index)
            print(f"Task '{removed_task}' removed.")
        else:
            print("Invalid task number.")
    except ValueError:
        print("Invalid input. Please enter a number.")

def update_task():
    """Prompts the user for a task index to update and modifies the task.

    First, it displays the current tasks. Then, it prompts the user for a 1-based index. If the index is valid, it asks for a new description for the task and updates it in the global 'tasks' list. Handles 'ValueError' for non-integer input and ensures the new task description is not empty.
    """

```

```

if not task:
    return
try:
    task_index = int(input("Enter the number of the task to update: ") - 1)
    if 0 <= task_index < len(tasks):
        new_task = input("Enter the new description for task '{tasks[task_index]}': ").strip()
        if new_task:
            tasks[task_index] = new_task
            print("Task updated successfully.")
        else:
            print("Task description cannot be empty, task not updated.")
    else:
        print("Invalid task number.")
except ValueError:
    print("Invalid input, please enter a number.")

def run_todo_list_manager():
    """Presents the main interactive menu for the to-do list manager.

    This function enters a loop, displaying options to the user for managing
    tasks (view, add, update, remove, exit). It calls the appropriate task
    management function based on user input until the user chooses to exit.
    """
    while True:
        print("\n--- To-Do List Manager ---")
        print("\n1. view tasks")
        print("\n2. add task")
        print("\n3. update task")
        print("\n4. remove task")
        print("\n5. exit")

```

```

if choice == '1':
    display_tasks()
elif choice == '2':
    add_task()
elif choice == '3':
    update_task()
elif choice == '4':
    remove_task()
elif choice == '5':
    print("Exiting To-Do List Manager. Goodbye!")
    break
else:
    print("Invalid choice, please try again.")

# --- Docstring scaffolding utility components ---

def generate_function_docstring(function_node):
    """Constructs a proposed Google-style docstring for a function node.

    Args:
        function_node (ast.FunctionDef): The AST node representing the function.

    Returns:
        str: A string containing the proposed Google-style docstring.
    """
    docstring = f"""{function_node.name} short description here. A more detailed explanation of what {function_node.name} does, here."""
    if function_node.args.args or function_node.args.kwonlyargs or function_node.args.vararg or function_node.args.kwarg:

```

```

# function_node.args.args or function_node.args.kwonlyargs or function_node.args.vararg or function_node.args.kwarg:
        docstring += "\nArgs:\n"
        for arg in function_node.args.args:
            docstring += f"    {arg.arg} (type): description of {arg.arg},\n"
        if function_node.args.vararg:
            docstring += f"    {function_node.args.vararg.arg} (type): description of variable arguments,\n"
        if function_node.args.kwonlyargs:
            for arg in function_node.args.kwonlyargs:
                docstring += f"    {arg.arg} (type): description of keyword-only argument {arg.arg},\n"
        if function_node.args.kwarg:
            docstring += f"    {function_node.args.kwarg.arg} (type): description of keyword arguments,\n"

    docstring += "\nReturns:\n"
    docstring += f"    str: A string containing the proposed Google-style docstring.\n"

    docstring += """
    """
    return docstring

def generate_class_docstring(class_node):
    """Constructs a proposed Google-style docstring for a class node.

    Args:
        class_node (ast.ClassDef): The AST node representing the class.

    Returns:
        str: A string containing the proposed Google-style docstring.
    """
    docstring = f"""{class_node.name} short description here. A more detailed explanation of what {class_node.name} represents or does, here. Attributes (in
    """
    docstring += """
    """
    return docstring

```



```

class DocstringScaffolder(ast.NodeVisitor):
    """An AST visitor that identifies functions and classes and prints proposed docstrings.

    This visitor traverses the Abstract Syntax Tree (AST) of a Python file.
    It overrides 'visit_FunctionDef' and 'visit_ClassDef' to check if a function
    or class already has a docstring. If not, it calls helper functions to
    generate and print a proposed Google-style docstring for the node.
    """

    def visit_FunctionDef(self, node):
        """Visits a function definition node.

        If the function does not have a docstring, it generates and prints one.

        Args:
            node (ast.FunctionDef): The function definition AST node.
        """
        if not ast.get_docstring(node):
            print(f"\n--- Proposed docstring for function: {node.name} ---")
            print(generate_function_docstring(node))
            self.generic_visit(node)

    def visit_ClassDef(self, node):
        """Visits a class definition node.

        If the class does not have a docstring, it generates and prints one.

        Args:
            node (ast.ClassDef): The class definition AST node.
        """

```

```

# --- Main Program Entry Point ---

def main():
    """Main entry point for the combined utility program.

    Presents a menu to the user allowing them to choose between the To-Do List
    Manager and the Docstring Scaffolding utility, or to exit the program.
    It continuously loops until the user chooses to exit.
    """
    while True:
        print("\n-----")
        print("  Combined Python Utilities Menu")
        print("-----")
        print("1. Run To-Do List Manager")
        print("2. Run Docstring Scaffolding Utility")
        print("3. Exit Program")
        main_choice = input("Enter your choice (1, 2, or 3): ").strip()

        if main_choice == '1':
            run_todo_list_manager()
        elif main_choice == '2':
            run_docstring_scaffolding_utility()
        elif main_choice == '3':
            print("Exiting program. Goodbye!")
            break
        else:
            print("Invalid choice. Please enter 1, 2, or 3.")

if __name__ == '__main__':
    main()

```

Output:

```

--- To-Do List Manager ---
1. View Tasks
2. Add Task
3. Update Task
4. Remove Task
5. Exit
Enter your choice: 1
Your to-do list is empty!

--- To-Do List Manager ---
1. View Tasks
2. Add Task
3. Update Task
4. Remove Task
5. Exit
Enter your choice: 2
Enter the new task: dance
Task 'dance' added.

--- To-Do List Manager ---
1. View Tasks
2. Add Task
3. Update Task
4. Remove Task
5. Exit
Enter your choice: 1

--- Your To-Do List ---
1. dance
-----

--- To-Do List Manager ---
1. View Tasks

```

JUSTIFICATION:

Adding placeholder Google-style docstrings to the To-Do List Manager automatically creates a clear structure for future documentation. It helps developers quickly understand the purpose of each function even before detailed explanations are written. This scaffolding improves readability and makes collaboration easier in shared projects. Detecting functions and inserting docstrings ensures consistent and standardized documentation across the codebase. Overall, it promotes clean, maintainable, and well-prepared code for future development.