AI Assisted Coding Assignment – 9.2

2403A52110

Batch: 04

* **Task 1:** Use AI to add Google-style docstrings to all functions in a given Python script.

**Prompt:** Add Google-style docstrings to all functions in the following Python script. For each function, the docstring should include: a brief description, a list of parameters with type hints, a description of return values with type hints, and an example of how to use the function. Do not provide any input-output examples; generate the docstrings based solely on the function names, parameters, and code logic

Code:

Before Adding Google Doc string styling:  
class ShoppingCart:

    """Simple shopping cart to add, remove items and compute total price."""

    def \_\_init\_\_(*self*):

*# items: dict[name] = {"price": float, "quantity": int}*

        self.\_items = {}

    def add\_item(*self*, *name*, *price*, *quantity*=1):

*if* price <= 0:

*raise* ValueError("price must be positive")

*if* quantity <= 0:

*raise* ValueError("quantity must be positive")

*if* name in self.\_items:

*# Accumulate quantity, keep latest price for simplicity*

            self.\_items[name]["quantity"] += quantity

            self.\_items[name]["price"] = price

*else*:

            self.\_items[name] = {"price": float(price), "quantity": int(quantity)}

    def remove\_item(*self*, *name*, *quantity*=1):

*if* quantity <= 0:

*raise* ValueError("quantity must be positive")

*if* name not in self.\_items:

*return*  *# noop*

        current\_qty = self.\_items[name]["quantity"]

*if* quantity >= current\_qty:

*del* self.\_items[name]

*else*:

            self.\_items[name]["quantity"] = current\_qty - quantity

    def get\_total\_price(*self*):

        total = 0.0

*for* item *in* self.\_items.values():

            total += item["price"] \* item["quantity"]

*return* round(total, 2)

After Adding Google docstring Styling:

def \_\_init\_\_(self):

    # items: dict[name] = {"price": float, "quantity": int}

    self.\_items = {}

def add\_item(self, name, price, quantity=1):

    if price <= 0:

        raise ValueError("price must be positive")

    if quantity <= 0:

        raise ValueError("quantity must be positive")

    if name in self.\_items:

        # Accumulate quantity, keep latest price for simplicity

        self.\_items[name]["quantity"] += quantity

        self.\_items[name]["price"] = price

    else:

        self.\_items[name] = {"price": float(price), "quantity": int(quantity)}

def remove\_item(self, name, quantity=1):

    if quantity <= 0:

        raise ValueError("quantity must be positive")

    if name not in self.\_items:

        return  # noop

    current\_qty = self.\_items[name]["quantity"]

    if quantity >= current\_qty:

        del self.\_items[name]

    else:

        self.\_items[name]["quantity"] = current\_qty - quantity

def get\_total\_price(self):

    total = 0.0

    for item in self.\_items.values():

        total += item["price"] \* item["quantity"]

    return round(total, 2)

* **Task 2:** Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.
* **Prompt:** "Add concise, meaningful inline comments to the following Python script. Focus only on complex or non-intuitive sections of the code. Do not add comments for simple or self-explanatory syntax like variable assignments, function calls, or basic loops. The comments should explain the purpose of the code block and how it achieves its goal, improving the overall readability and maintainability of the script."

**Code:**  
Before adding meaningful concise and meaningful inline comment:  
import re

def validate\_email(email):

    """

    Validate an email address format.

    This function checks if an email address has a valid format by ensuring:

    - Contains exactly one @ symbol

    - Has a valid local part (before @)

    - Has a valid domain part (after @)

    - Domain has a valid extension (like .com, .org, etc.)

    Args:

        email (str): The email address to validate

    Returns:

        bool: True if the email format is valid, False otherwise

    Examples:

        >>> validate\_email("user@example.com")

        True

        >>> validate\_email("invalid-email")

        False

        >>> validate\_email("user@domain")

        False

    """

    if not email or not isinstance(email, str):

        return False

    email = email.strip()

    if not email:

        return False

    if email.count('@') != 1:

        return False

    local\_part, domain\_part = email.split('@')

    if not local\_part:

        return False

    if not domain\_part:

        return False

    if local\_part.startswith('.') or local\_part.endswith('.'):

        return False

    if '..' in local\_part:

        return False

    if ' ' in local\_part:

        return False

    if ' ' in domain\_part:

        return False

    if '.' not in domain\_part:

        return False

    if domain\_part.startswith('.') or domain\_part.endswith('.'):

        return False

    if '..' in domain\_part:

        return False

    extension = domain\_part.split('.')[-1]

    if len(extension) < 2:

        return False

    if not extension.isalpha():

        return False

    email\_pattern = r'^[a-zA-Z0-9.\_+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    if not re.match(email\_pattern, email):

        return False

    return True

def validate\_email\_regex\_only(email):

    """

    Alternative email validation using only regex.

    This is more concise but less readable and debuggable.

    """

    if not email or not isinstance(email, str):

        return False

    email = email.strip()

    if not email:

        return False

    pattern = r'^[a-zA-Z0-9.\_+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    return bool(re.match(pattern, email))

**Code after adding meaningful inline comments:**import re

def validate\_email(email):

    """

    Validate an email address format.

    ...existing docstring...

    """

    # Check if email is empty or not a string

    if not email or not isinstance(email, str):

        return False

    email = email.strip()

    if not email:

        return False

    # Ensure exactly one @ symbol is present

    if email.count('@') != 1:

        return False

    # Split into local and domain parts

    local\_part, domain\_part = email.split('@')

    if not local\_part:

        return False

    if not domain\_part:

        return False

    # Local part should not start/end with dot or have consecutive dots

    if local\_part.startswith('.') or local\_part.endswith('.'):

        return False

    if '..' in local\_part:

        return False

    if ' ' in local\_part:

        return False

    # Domain part should not have spaces, start/end with dot, or have consecutive dots

    if ' ' in domain\_part:

        return False

    if '.' not in domain\_part:

        return False

    if domain\_part.startswith('.') or domain\_part.endswith('.'):

        return False

    if '..' in domain\_part:

        return False

    # Extract and validate extension (e.g., 'com' in 'example.com')

    extension = domain\_part.split('.')[-1]

    if len(extension) < 2:

        return False

    if not extension.isalpha():

        return False

    # Final comprehensive regex check for strict format validation

    email\_pattern = r'^[a-zA-Z0-9.\_+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    if not re.match(email\_pattern, email):

        return False

    return True

# Alternative implementation using only regex (concise, but less debuggable)

def validate\_email\_regex\_only(email):

    """

    Alternative email validation using only regex.

    This is more concise but less readable and debuggable.

    """

    if not email or not isinstance(email, str):

        return False

    email = email.strip()

    if not email:

        return False

    # Comprehensive regex pattern for email validation

    pattern = r'^[a-zA-Z0-9.\_+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    return bool(re.match(pattern, email))

* **Task 3:** Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.

**Prompt:**Draft a single, multi-line module-level docstring for the top of the following Python file. The docstring should provide a high-level summary of the module's overall purpose, list any external dependencies, and briefly describe the main functions or classes it contains. Do not include excessive detail or rewrite the code's functionality within the docstring.

**Code:**  
import re

def validate\_email(email):

    """

    Validate an email address format.

    ...existing docstring...

    """

    # Check if email is empty or not a string

    if not email or not isinstance(email, str):

        return False

    email = email.strip()

    if not email:

        return False

    # Ensure exactly one @ symbol is present

    if email.count('@') != 1:

        return False

    # Split into local and domain parts

    local\_part, domain\_part = email.split('@')

    if not local\_part:

        return False

    if not domain\_part:

        return False

    # Local part should not start/end with dot or have consecutive dots

    if local\_part.startswith('.') or local\_part.endswith('.'):

        return False

    if '..' in local\_part:

        return False

    if ' ' in local\_part:

        return False

    # Domain part should not have spaces, start/end with dot, or have consecutive dots

    if ' ' in domain\_part:

        return False

    if '.' not in domain\_part:

        return False

    if domain\_part.startswith('.') or domain\_part.endswith('.'):

        return False

    if '..' in domain\_part:

        return False

    # Extract and validate extension (e.g., 'com' in 'example.com')

    extension = domain\_part.split('.')[-1]

    if len(extension) < 2:

        return False

    if not extension.isalpha():

        return False

    # Final comprehensive regex check for strict format validation

    email\_pattern = r'^[a-zA-Z0-9.\_+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    if not re.match(email\_pattern, email):

        return False

    return True

# Alternative implementation using only regex (concise, but less debuggable)

def validate\_email\_regex\_only(email):

    """

    Alternative email validation using only regex.

    This is more concise but less readable and debuggable.

    """

    if not email or not isinstance(email, str):

        return False

    email = email.strip()

    if not email:

        return False

    # Comprehensive regex pattern for email validation

    pattern = r'^[a-zA-Z0-9.\_+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    return bool(re.match(pattern, email))

Output:  
"""

Email Validation Module

This module provides utility functions for validating email address formats in Python.

It offers both a comprehensive, step-by-step validation approach and a concise regular expression-based alternative.

Dependencies:

    - re (Python standard library module for regular expressions)

Main Functions:

    - validate\_email(email): Performs detailed validation of an email address, checking structure and format.

    - validate\_email\_regex\_only(email): Validates an email address using a single regular expression.

These functions are intended for basic email format validation and do not verify the existence or deliverability of the