Assignment-9.4

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

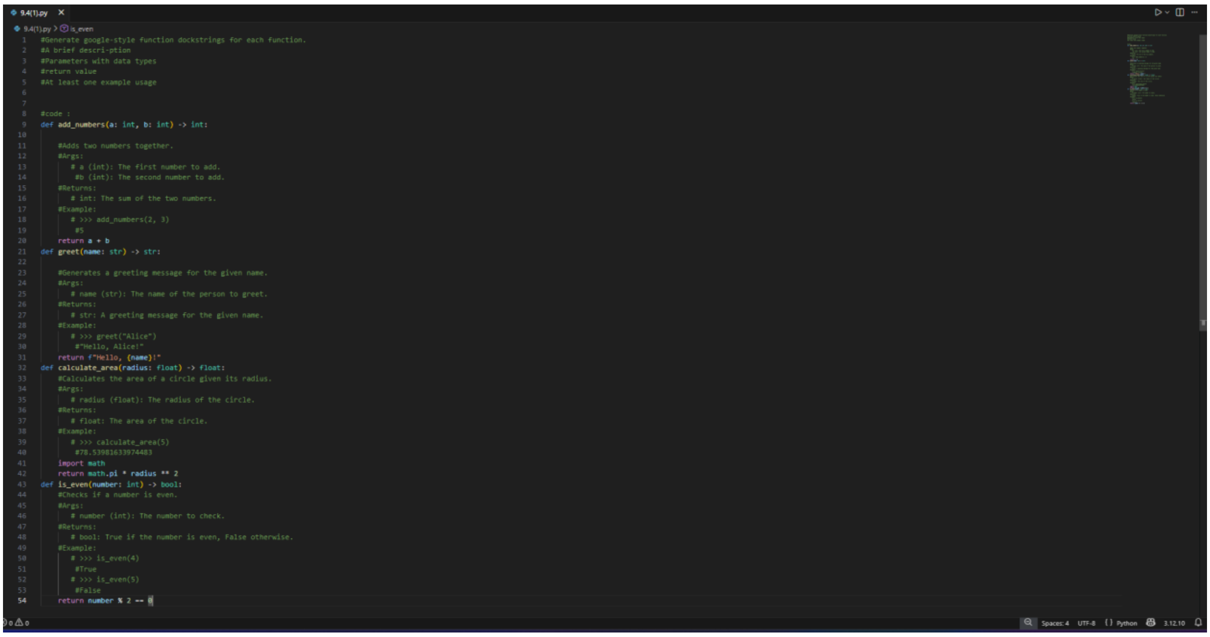
Scenario

You have joined a development team where several utility functions are already implemented, but the code lacks proper documentation. New team members are struggling to understand how these functions should be used.

Task Description

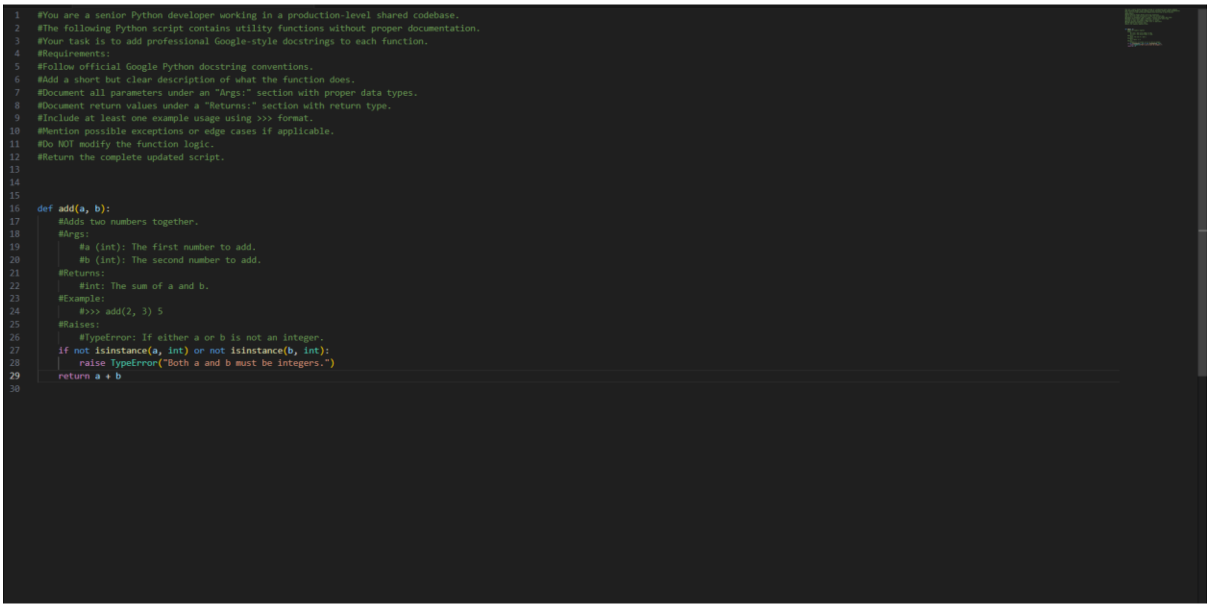
You are given a Python script containing multiple functions without any docstrings.

Zero shot:



```python
#Generate google-style function dockstrings for each function.
#A brief descri-ption
#Parameters with data types
#return value
#At least one example usage


#code :
def add_numbers(a: int, b: int) -> int:

    #Adds two numbers together.
    #Args:
       # a (int): The first number to add.
        #b (int): The second number to add.
    #Returns:
       # int: The sum of the two numbers.
    #Example:
       # >>> add_numbers(2, 3)
        #5
    return a + b
def greet(name: str) -> str:

    #Generates a greeting message for the given name.
    #Args:
       # name (str): The name of the person to greet.
    #Returns:
       # str: A greeting message for the given name.
    #Example:
       # >>> greet("Alice")
        #"Hello, Alice!"
    return f"Hello, {name}!"
def calculate_area(radius: float) -> float:
    #Calculates the area of a circle given its radius.
    #Args:
       # radius (float): The radius of the circle.
    #Returns:
       # float: The area of the circle.
    #Example:
       # >>> calculate_area(5)
        #78.53981633974483
    import math
    return math.pi * radius ** 2
def is_even(number: int) -> bool:
    #Checks if a number is even.
    #Args:
       # number (int): The number to check.
    #Returns:
       # bool: True if the number is even, False otherwise.
    #Example:
       # >>> is_even(4)
        #True
       # >>> is_even(5)
        #False
    return number % 2 == 0
```

Context based:

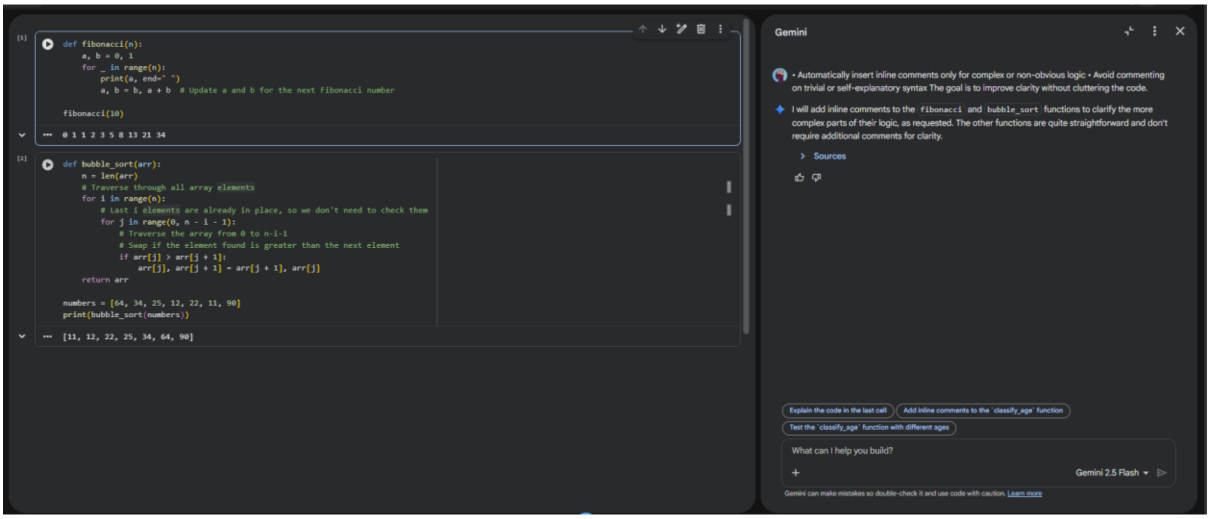


```python
#You are a senior Python developer working in a production-level shared codebase.
#The following Python script contains utility functions without proper documentation.
#Your task is to add professional Google-style docstrings to each function.
#Requirements:
#Follow official Google Python docstring conventions.
#Add a short but clear description of what the function does.
#Document all parameters under an "Args:" section with proper data types.
#Document return values under a "Returns:" section with return type.
#Include at least one example usage using >>> format.
#Mention possible exceptions or edge cases if applicable.
#Do NOT modify the function logic.
#Return the complete updated script.



def add(a, b):
    #Adds two numbers together.
    #Args:
        #a (int): The first number to add.
        #b (int): The second number to add.
    #Returns:
        #int: The sum of a and b.
    #Example:
        #>>> add(2, 3) 5
    #Raises:
        #TypeError: If either a or b is not an integer.
    if not isinstance(a, int) or not isinstance(b, int):
        raise TypeError("Both a and b must be integers.")
    return a + b

```

Task 2: Enhancing Readability Through AI-Generated Inline Comments

Scenario

A Python program contains complex logic that works correctly but is difficult to understand at first glance. Future maintainers may find it hard to debug or extend this code.



```python
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        print(a, end=" ")
        a, b = b, a + b  # Update a and b for the next Fibonacci number

fibonacci(10)
```

```
0 1 1 2 3 5 8 13 21 34
```

```python
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Last i elements are already in place, so we don't need to check them
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

numbers = [64, 34, 25, 12, 22, 11, 90]
print(bubble_sort(numbers))
```

```
[11, 12, 22, 25, 34, 64, 90]
```

• Automatically insert inline comments only for complex or non-obvious logic • Avoid commenting on trivial or self-explanatory syntax The goal is to improve clarity without cluttering the code.

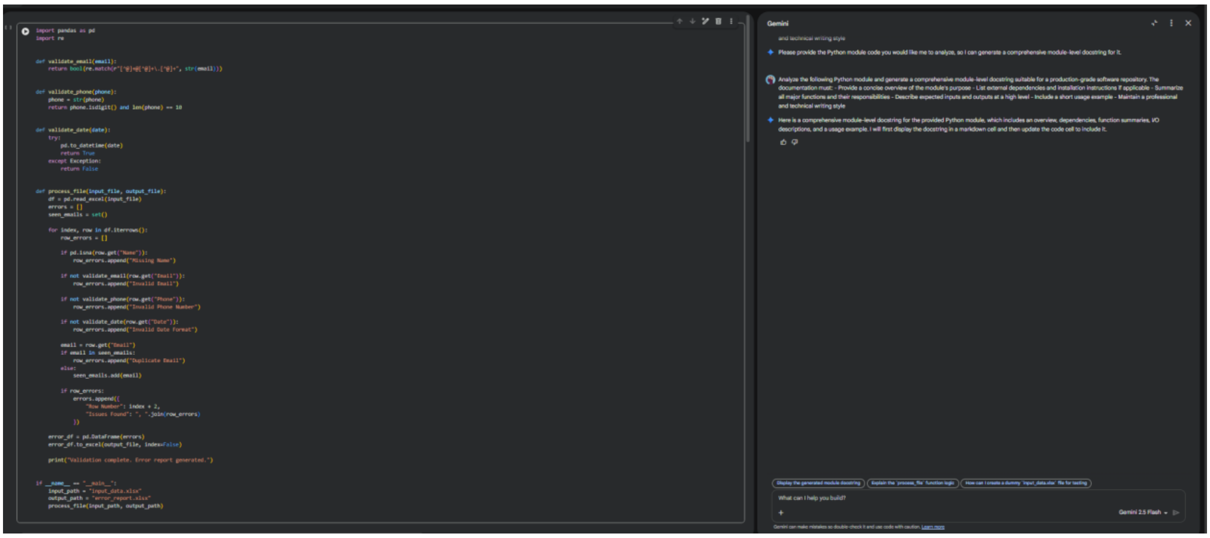
I will add inline comments to the `fibonacci` and `bubble_sort` functions to clarify the more complex parts of their logic, as requested. The other functions are quite straightforward and don't require additional comments for clarity.

Sources
Explain the code in the last cell
Add inline comments to the `classify_age` function
Test the `classify_age` function with different ages
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more

Task 3: Generating Module-Level Documentation for a Python Package

Scenario

Your team is preparing a Python module to be shared internally (or uploaded to a repository). Anyone opening the file should immediately understand its purpose and structure.



```python
import pandas as pd
import re


def validate_email(email):
    return bool(re.match(r"[^@]+@[^@]+\.[^@]+", str(email)))


def validate_phone(phone):
    phone = str(phone)
    return phone.isdigit() and len(phone) == 10


def validate_date(date):
    try:
        pd.to_datetime(date)
        return True
    except Exception:
        return False


def process_file(input_file, output_file):
    df = pd.read_excel(input_file)
    errors = []
    seen_emails = set()

    for index, row in df.iterrows():
        row_errors = []

        if pd.isna(row.get("Name")):
            row_errors.append("Missing Name")

        if not validate_email(row.get("Email")):
            row_errors.append("Invalid Email")

        if not validate_phone(row.get("Phone")):
            row_errors.append("Invalid Phone Number")

        if not validate_date(row.get("Date")):
            row_errors.append("Invalid Date Format")

        email = row.get("Email")
        if email in seen_emails:
            row_errors.append("Duplicate Email")
        else:
            seen_emails.add(email)

        if row_errors:
            errors.append({
                "Row Number": index + 2,
                "Issues Found": ", ".join(row_errors)
            })

    error_df = pd.DataFrame(errors)
    error_df.to_excel(output_file, index=False)

    print("Validation complete. Error report generated.")


if __name__ == "__main__":
    input_path = "input_data.xlsx"
    output_path = "error_report.xlsx"
    process_file(input_path, output_path)
```

Gemini

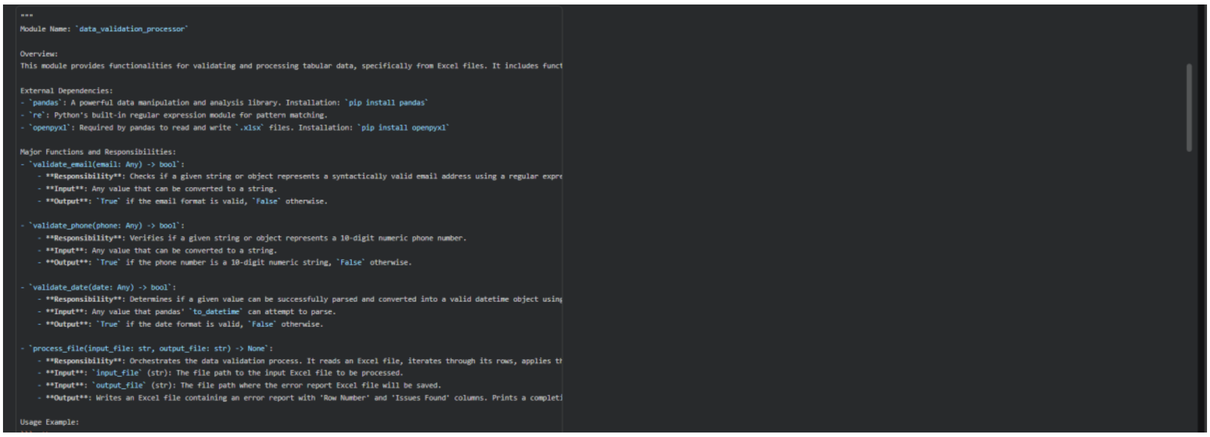
and technical writing style

Please provide the Python module code you would like me to analyze, so I can generate a comprehensive module-level docstring for it.

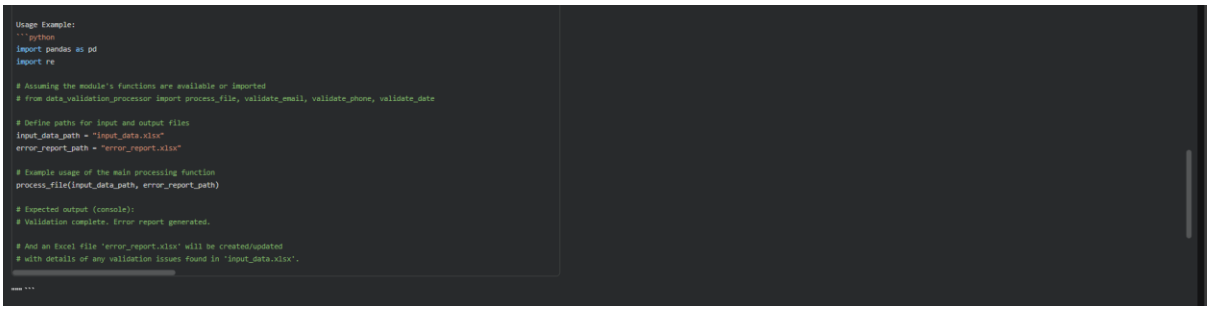
Analyze the following Python module and generate a comprehensive module-level docstring suitable for a production-grade software repository. The documentation must: - Provide a concise overview of the module's purpose - List external dependencies and installation instructions if applicable - Summarize all major functions and their responsibilities - Describe expected inputs and outputs at a high level - Include a short usage example - Maintain a professional and technical writing style

Here is a comprehensive module-level docstring for the provided Python module, which includes an overview, dependencies, function summaries, I/O descriptions, and a usage example. I will first display the docstring in a markdown cell and then update the code cell to include it.

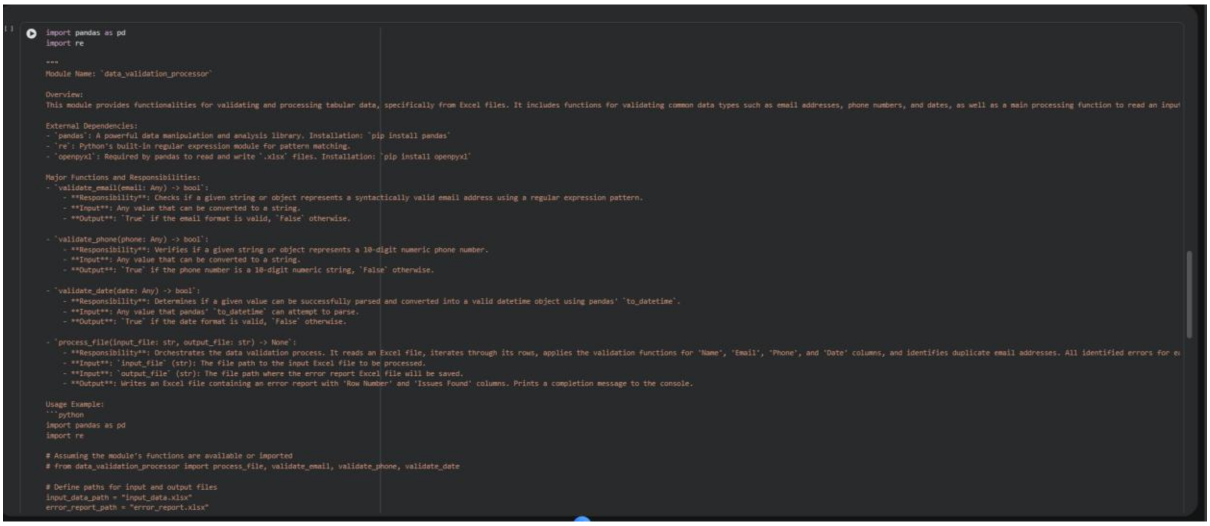
Display the generated module docstring
Explain the `process_file` function logic
How can I create a dummy `input_data.xlsx` file for testing
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more



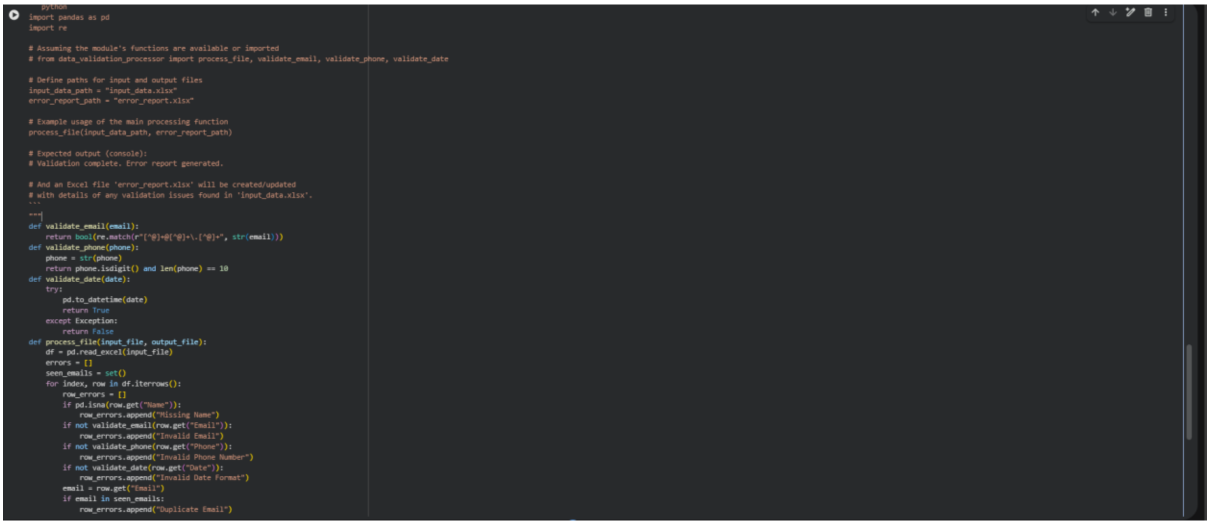
````
"""
Module Name: `data_validation_processor`

Overview:
This module provides functionalities for validating and processing tabular data, specifically from Excel files. It includes funct

External Dependencies:
- `pandas`: A powerful data manipulation and analysis library. Installation: `pip install pandas`
- `re`: Python's built-in regular expression module for pattern matching.
- `openpyxl`: Required by pandas to read and write `.xlsx` files. Installation: `pip install openpyxl`

Major Functions and Responsibilities:
- `validate_email(email: Any) -> bool`:
    - **Responsibility**: Checks if a given string or object represents a syntactically valid email address using a regular expre
    - **Input**: Any value that can be converted to a string.
    - **Output**: `True` if the email format is valid, `False` otherwise.

- `validate_phone(phone: Any) -> bool`:
    - **Responsibility**: Verifies if a given string or object represents a 10-digit numeric phone number.
    - **Input**: Any value that can be converted to a string.
    - **Output**: `True` if the phone number is a 10-digit numeric string, `False` otherwise.

- `validate_date(date: Any) -> bool`:
    - **Responsibility**: Determines if a given value can be successfully parsed and converted into a valid datetime object using
    - **Input**: Any value that pandas' `to_datetime` can attempt to parse.
    - **Output**: `True` if the date format is valid, `False` otherwise.

- `process_file(input_file: str, output_file: str) -> None`:
    - **Responsibility**: Orchestrates the data validation process. It reads an Excel file, iterates through its rows, applies th
    - **Input**: `input_file` (str): The file path to the input Excel file to be processed.
    - **Input**: `output_file` (str): The file path where the error report Excel file will be saved.
    - **Output**: Writes an Excel file containing an error report with 'Row Number' and 'Issues Found' columns. Prints a completi

Usage Example:
````

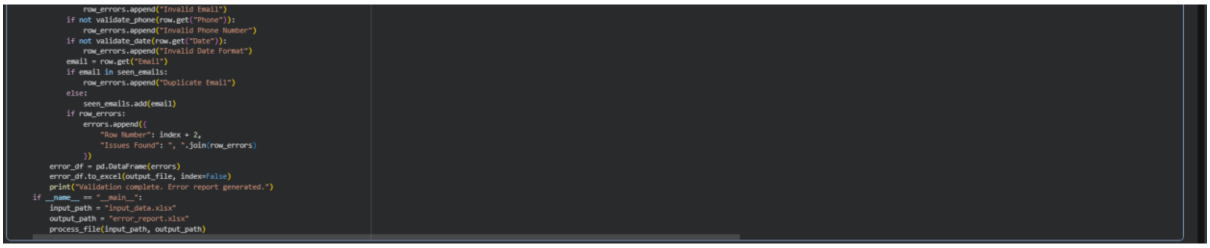


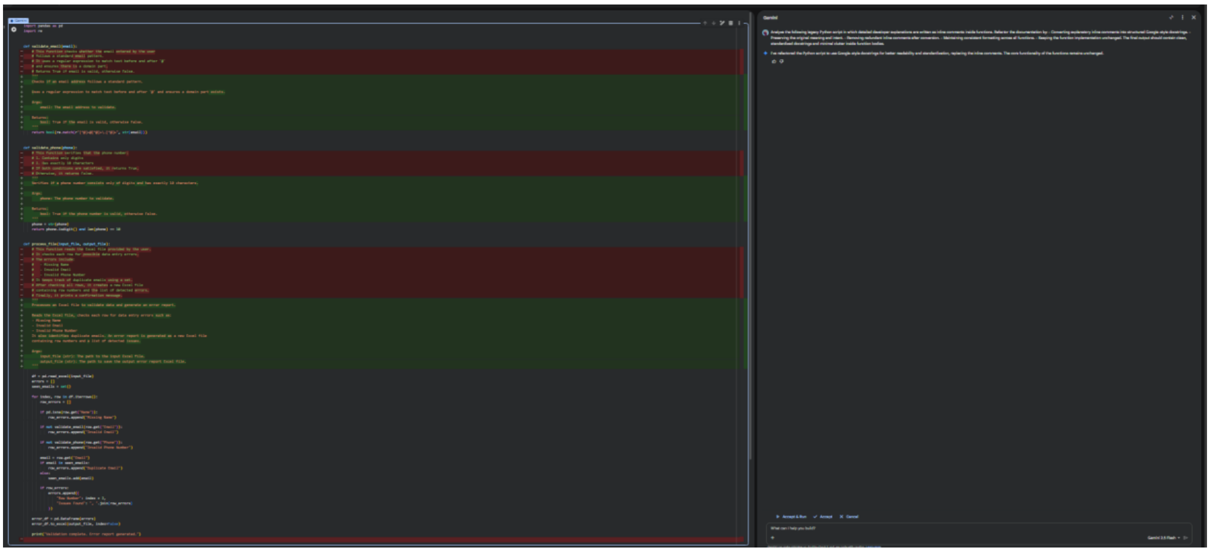
~~~
Usage Example:
```python
import pandas as pd
import re

# Assuming the module's functions are available or imported
# from data_validation_processor import process_file, validate_email, validate_phone, validate_date

# Define paths for input and output files
input_data_path = "input_data.xlsx"
error_report_path = "error_report.xlsx"

# Example usage of the main processing function
process_file(input_data_path, error_report_path)

# Expected output (console):
# Validation complete. Error report generated.

# And an Excel file 'error_report.xlsx' will be created/updated
# with details of any validation issues found in 'input_data.xlsx'.

""" ```
~~~

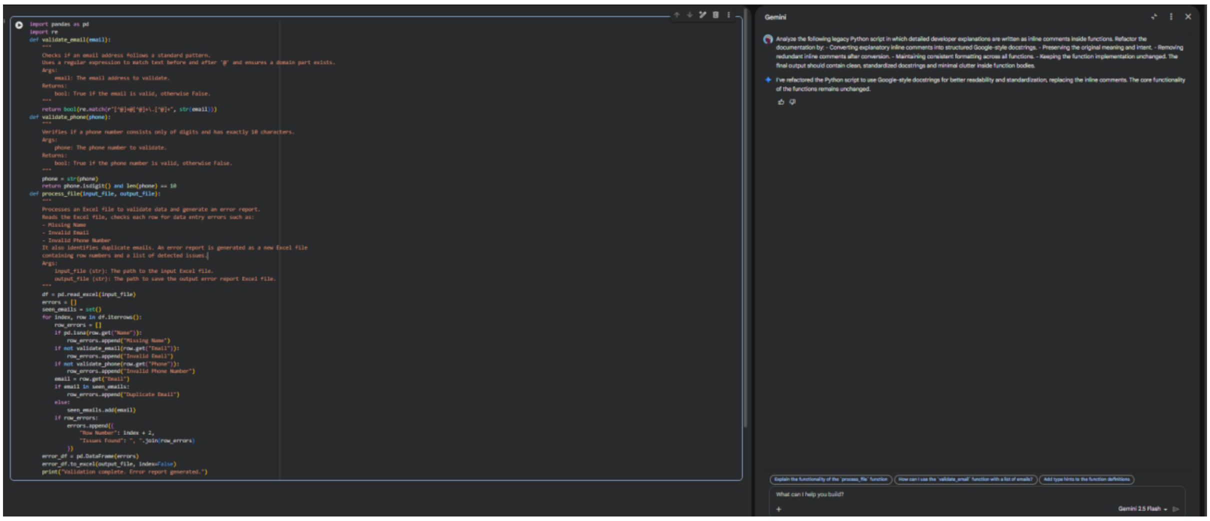


~~~
import pandas as pd
import re

"""
Module Name: `data_validation_processor`

Overview:
This module provides functionalities for validating and processing tabular data, specifically from Excel files. It includes functions for validating common data types such as email addresses, phone numbers, and dates, as well as a main processing function to read an inpu

External Dependencies:
- `pandas`: A powerful data manipulation and analysis library. Installation: `pip install pandas`
- `re`: Python's built-in regular expression module for pattern matching.
- `openpyxl`: Required by pandas to read and write `.xlsx` files. Installation: `pip install openpyxl`

Major Functions and Responsibilities:
- `validate_email(email: Any) -> bool`:
    - **Responsibility**: Checks if a given string or object represents a syntactically valid email address using a regular expression pattern.
    - **Input**: Any value that can be converted to a string.
    - **Output**: `True` if the email format is valid, `False` otherwise.

- `validate_phone(phone: Any) -> bool`:
    - **Responsibility**: Verifies if a given string or object represents a 10-digit numeric phone number.
    - **Input**: Any value that can be converted to a string.
    - **Output**: `True` if the phone number is a 10-digit numeric string, `False` otherwise.

- `validate_date(date: Any) -> bool`:
    - **Responsibility**: Determines if a given value can be successfully parsed and converted into a valid datetime object using pandas' `to_datetime`.
    - **Input**: Any value that pandas' `to_datetime` can attempt to parse.
    - **Output**: `True` if the date format is valid, `False` otherwise.

- `process_file(input_file: str, output_file: str) -> None`:
    - **Responsibility**: Orchestrates the data validation process. It reads an Excel file, iterates through its rows, applies the validation functions for 'Name', 'Email', 'Phone', and 'Date' columns, and identifies duplicate email addresses. All identified errors for e
    - **Input**: `input_file` (str): The file path to the input Excel file to be processed.
    - **Input**: `output_file` (str): The file path where the error report Excel file will be saved.
    - **Output**: Writes an Excel file containing an error report with 'Row Number' and 'Issues Found' columns. Prints a completion message to the console.

Usage Example:
```python
import pandas as pd
import re

# Assuming the module's functions are available or imported
# from data_validation_processor import process_file, validate_email, validate_phone, validate_date

# Define paths for input and output files
input_data_path = "input_data.xlsx"
error_report_path = "error_report.xlsx"
~~~



```
   python
import pandas as pd
import re

# Assuming the module's functions are available or imported
# from data_validation_processor import process_file, validate_email, validate_phone, validate_date

# Define paths for input and output files
input_data_path = "input_data.xlsx"
error_report_path = "error_report.xlsx"

# Example usage of the main processing function
process_file(input_data_path, error_report_path)

# Expected output (console):
# Validation complete. Error report generated.

# And an Excel file 'error_report.xlsx' will be created/updated
# with details of any validation issues found in 'input_data.xlsx'.
```
"""
def validate_email(email):
    return bool(re.match(r"[^@]+@[^@]+\.[^@]+", str(email)))
def validate_phone(phone):
    phone = str(phone)
    return phone.isdigit() and len(phone) == 10
def validate_date(date):
    try:
        pd.to_datetime(date)
        return True
    except Exception:
        return False
def process_file(input_file, output_file):
    df = pd.read_excel(input_file)
    errors = []
    seen_emails = set()
    for index, row in df.iterrows():
        row_errors = []
        if pd.isna(row.get("Name")):
            row_errors.append("Missing Name")
        if not validate_email(row.get("Email")):
            row_errors.append("Invalid Email")
        if not validate_phone(row.get("Phone")):
            row_errors.append("Invalid Phone Number")
        if not validate_date(row.get("Date")):
            row_errors.append("Invalid Date Format")
        email = row.get("Email")
        if email in seen_emails:
            row_errors.append("Duplicate Email")
```



```
            row_errors.append("Invalid Email")
        if not validate_phone(row.get("Phone")):
            row_errors.append("Invalid Phone Number")
        if not validate_date(row.get("Date")):
            row_errors.append("Invalid Date Format")
        email = row.get("Email")
        if email in seen_emails:
            row_errors.append("Duplicate Email")
        else:
            seen_emails.add(email)
        if row_errors:
            errors.append({
                "Row Number": index + 2,
                "Issues Found": ", ".join(row_errors)
            })
    error_df = pd.DataFrame(errors)
    error_df.to_excel(output_file, index=False)
    print("Validation complete. Error report generated.")
if __name__ == "__main__":
    input_path = "input_data.xlsx"
    output_path = "error_report.xlsx"
    process_file(input_path, output_path)
```

Task 4: Converting Developer Comments into Structured Docstrings

Scenario

In a legacy project, developers have written long explanatory comments inside functions instead of proper docstrings. The team now wants to standardize documentation.
```



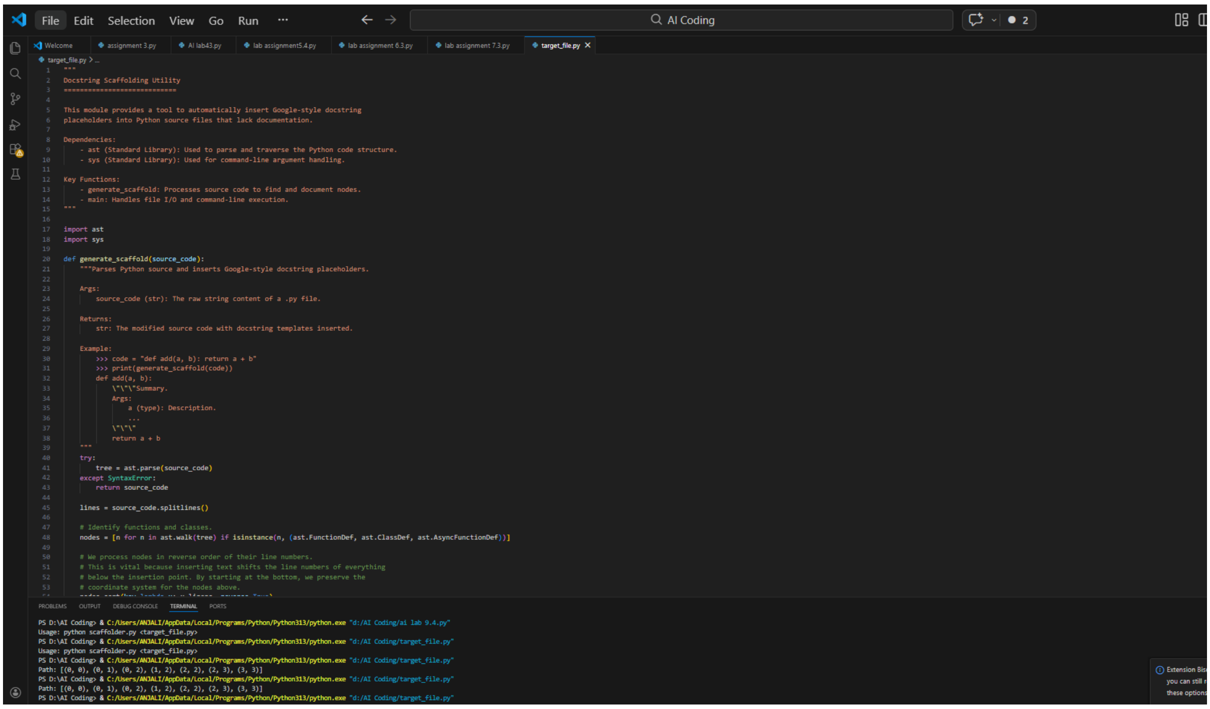

Gemini
Explain the functionality of the `process_file` function
How can I use the `validate_email` function with a list of emails?
Add type hints to the function definitions
What can I help you build?
Gemini 2.5 Flash

```
import pandas as pd
import re
def validate_email(email):
    """
    Checks if an email address follows a standard pattern.
    Uses a regular expression to match text before and after '@' and ensures a domain part exists.
    Args:
        email: The email address to validate.
    Returns:
        bool: True if the email is valid, otherwise False.
    """
    return bool(re.match(r"[^@]+@[^@]+\.[^@]+", str(email)))
def validate_phone(phone):
    """
    Verifies if a phone number consists only of digits and has exactly 10 characters.
    Args:
        phone: The phone number to validate.
    Returns:
        bool: True if the phone number is valid, otherwise False.
    """
    phone = str(phone)
    return phone.isdigit() and len(phone) == 10
def process_file(input_file, output_file):
    """
    Processes an Excel file to validate data and generate an error report.
    Reads the Excel file, checks each row for data entry errors such as:
    - Missing Name
    - Invalid Email
    - Invalid Phone Number
    It also identifies duplicate emails. An error report is generated as a new Excel file
    containing row numbers and a list of detected issues.
    Args:
        input_file (str): The path to the input Excel file.
        output_file (str): The path to save the output error report Excel file.
    """
    df = pd.read_excel(input_file)
    errors = []
    seen_emails = set()
    for index, row in df.iterrows():
        row_errors = []
        if pd.isna(row.get("Name")):
            row_errors.append("Missing Name")
        if not validate_email(row.get("Email")):
            row_errors.append("Invalid Email")
        if not validate_phone(row.get("Phone")):
            row_errors.append("Invalid Phone Number")
        email = row.get("Email")
        if email in seen_emails:
            row_errors.append("Duplicate Email")
        else:
            seen_emails.add(email)
        if row_errors:
            errors.append({
                "Row Number": index + 2,
                "Issues Found": ", ".join(row_errors)
            })
    error_df = pd.DataFrame(errors)
    error_df.to_excel(output_file, index=False)
    print("Validation complete. Error report generated.")
```

Analyze the following legacy Python script in which detailed developer explanations are written as inline comments inside functions. Refactor the documentation by: - Converting explanatory inline comments into structured Google-style docstrings. - Preserving the original meaning and intent. - Removing redundant inline comments after conversion. - Maintaining consistent formatting across all functions. - Keeping the function implementation unchanged. The final output should contain clean, standardized docstrings and minimal clutter inside function bodies.

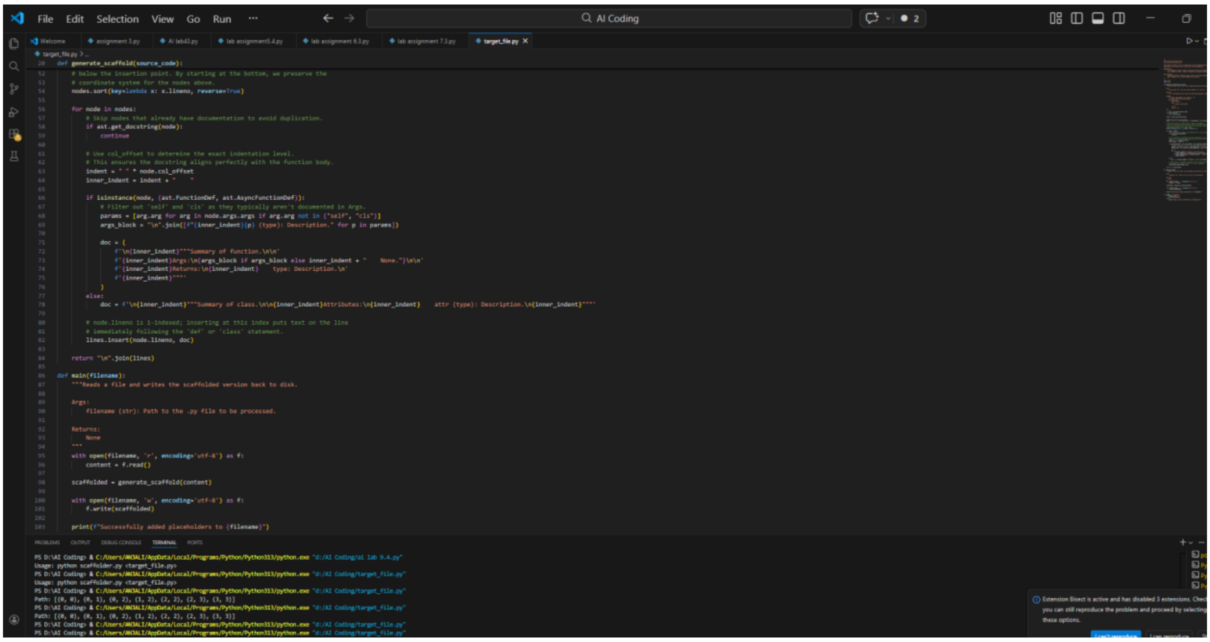
I've refactored the Python script to use Google-style docstrings for better readability and standardization, replacing the inline comments. The core functionality of the functions remains unchanged.

Task 5: Building a Mini Automatic Documentation Generator



```python
"""
Docstring Scaffolding Utility
============================

This module provides a tool to automatically insert Google-style docstring
placeholders into Python source files that lack documentation.

Dependencies:
    - ast (Standard Library): Used to parse and traverse the Python code structure.
    - sys (Standard Library): Used for command-line argument handling.

Key Functions:
    - generate_scaffold: Processes source code to find and document nodes.
    - main: Handles file I/O and command-line execution.
"""

import ast
import sys

def generate_scaffold(source_code):
    """Parses Python source and inserts Google-style docstring placeholders.

    Args:
        source_code (str): The raw string content of a .py file.

    Returns:
        str: The modified source code with docstring templates inserted.

    Example:
        >>> code = "def add(a, b): return a + b"
        >>> print(generate_scaffold(code))
        def add(a, b):
            \"\"\"Summary.
            Args:
                a (type): Description.
                ...
            \"\"\"
            return a + b
    """
    try:
        tree = ast.parse(source_code)
    except SyntaxError:
        return source_code

    lines = source_code.splitlines()

    # Identify functions and classes.
    nodes = [n for n in ast.walk(tree) if isinstance(n, (ast.FunctionDef, ast.ClassDef, ast.AsyncFunctionDef))]

    # We process nodes in reverse order of their line numbers.
    # This is vital because inserting text shifts the line numbers of everything
    # below the insertion point. By starting at the bottom, we preserve the
    # coordinate system for the nodes above.
```

```
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/ai lab 9.4.py"
Usage: python scaffolder.py <target_file.py>
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/target_file.py"
Usage: python scaffolder.py <target_file.py>
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/target_file.py"
Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (3, 3)]
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/target_file.py"
Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (3, 3)]
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/target_file.py"
```



```python
def generate_scaffold(source_code):
    # below the insertion point. By starting at the bottom, we preserve the
    # coordinate system for the nodes above.
    nodes.sort(key=lambda x: x.lineno, reverse=True)

    for node in nodes:
        # Skip nodes that already have documentation to avoid duplication.
        if ast.get_docstring(node):
            continue

        # Use col_offset to determine the exact indentation level.
        # This ensures the docstring aligns perfectly with the function body.
        indent = " " * node.col_offset
        inner_indent = indent + "    "

        if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
            # Filter out 'self' and 'cls' as they typically aren't documented in Args.
            params = [arg.arg for arg in node.args.args if arg.arg not in ("self", "cls")]
            args_block = "\n".join([f"{inner_indent}{p} (type): Description." for p in params])

            doc = (
                f'\n{inner_indent}"""Summary of function.\n\n'
                f'{inner_indent}Args:\n{args_block if args_block else inner_indent + "    None."}\n\n'
                f'{inner_indent}Returns:\n{inner_indent}    type: Description.\n'
                f'{inner_indent}"""'
            )
        else:
            doc = f'\n{inner_indent}"""Summary of class.\n\n{inner_indent}Attributes:\n{inner_indent}    attr (type): Description.\n{inner_indent}"""'

        # node.lineno is 1-indexed; inserting at this index puts text on the line
        # immediately following the 'def' or 'class' statement.
        lines.insert(node.lineno, doc)

    return "\n".join(lines)

def main(filename):
    """Reads a file and writes the scaffolded version back to disk.

    Args:
        filename (str): Path to the .py file to be processed.

    Returns:
        None
    """
    with open(filename, 'r', encoding='utf-8') as f:
        content = f.read()

    scaffolded = generate_scaffold(content)

    with open(filename, 'w', encoding='utf-8') as f:
        f.write(scaffolded)

    print(f"Successfully added placeholders to {filename}")
```

```
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/ai lab 9.4.py"
Usage: python scaffolder.py <target_file.py>
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/target_file.py"
Usage: python scaffolder.py <target_file.py>
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/target_file.py"
Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (3, 3)]
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/target_file.py"
Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (3, 3)]
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/target_file.py"
PS D:\AI Coding> & C:/Users/ANJALI/AppData/Local/Programs/Python/Python313/python.exe "d:/AI Coding/target_file.py"
```