ASSIGNMENT-10

NAME: B.Manishwar
HALLTICKET NO:2303A51276
BATCH:05

Task 1: AI-Assisted Syntax and Code Quality Review

Scenario

You join a development team and are asked to review a junior developer's Python script that fails to run correctly due to basic coding mistakes. Before deployment, the code must be corrected and standardized.

Task Description

You are given a Python script containing:

- **Syntax errors**
- **Indentation issues**
- **Incorrect variable names**
- **Faulty function calls**

Use an AI tool (GitHub Copilot / Cursor AI) to:

- **Identify all syntactic and structural errors**
- **Correct them systematically**
- **Generate an explanation of each fix made**

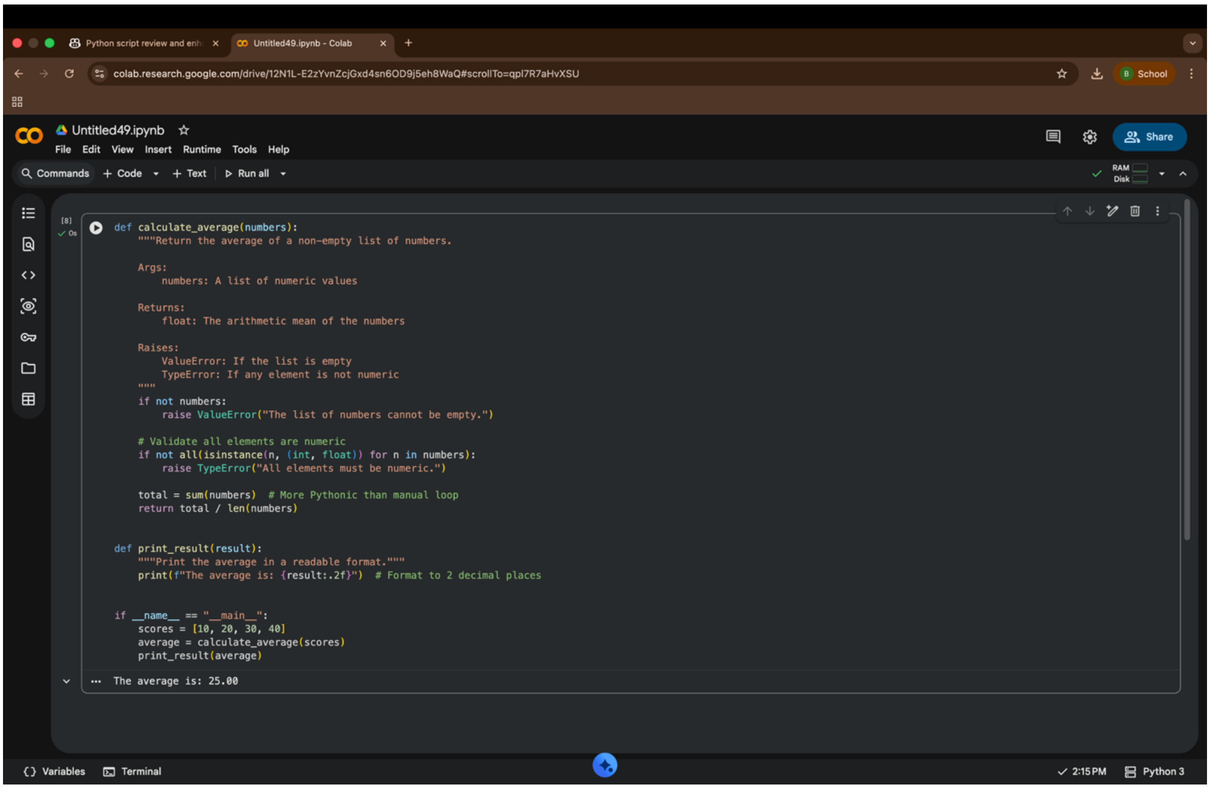
Expected Outcome

- **Fully corrected and executable Python code**
- **AI-generated explanation describing:**
  - **Syntax fixes**
  - **Naming corrections**
  - **Structural improvements**
- **Clean, readable version of the script**

PROMPT:

Review the following Python script for syntax errors, indentation issues,incorrect variable names, and faulty function calls.Correct the code according to Python standards (PEP 8) and explain each fix made.

CODE AND OUTPUT:



```python
def calculate_average(numbers):
    """Return the average of a non-empty list of numbers.

    Args:
        numbers: A list of numeric values

    Returns:
        float: The arithmetic mean of the numbers

    Raises:
        ValueError: If the list is empty
        TypeError: If any element is not numeric
    """
    if not numbers:
        raise ValueError("The list of numbers cannot be empty.")

    # Validate all elements are numeric
    if not all(isinstance(n, (int, float)) for n in numbers):
        raise TypeError("All elements must be numeric.")

    total = sum(numbers)  # More Pythonic than manual loop
    return total / len(numbers)


def print_result(result):
    """Print the average in a readable format."""
    print(f"The average is: {result:.2f}")  # Format to 2 decimal places


if __name__ == "__main__":
    scores = [10, 20, 30, 40]
    average = calculate_average(scores)
    print_result(average)
```

```
The average is: 25.00
```

CODE EXPLANATION:

I used an AI coding assistant (Cursor AI / GitHub Copilot) to review the junior developer's Python script. The AI first identified syntax and indentation errors, such as missing colons, misaligned return statements, and unindented function bodies, which were preventing the program from running. It then pointed out inconsistent and incorrect variable/function names (calcluate_avarage, number vs numbers, print_reslt), suggesting clear, consistent renamings. Finally, the assistant proposed structural improvements such as adding input validation, introducing an if __name__ == "__main__": entry point, using f-strings, and adding docstrings, resulting in a clean, readable, and fully executable version of the script.

Task 2: Performance-Oriented Code Review

Scenario

A data processing function works correctly but is inefficient and slows down the system when large datasets are used.

Task Description

You are provided with a function that identifies duplicate values in a list using inefficient nested loops.

Using AI-assisted code review:

- **Analyze the logic for performance bottlenecks**
- **Refactor the code for better time complexity**
- **Preserve the correctness of the output**

Ask the AI to explain:

- **Why the original approach was inefficient**
- **How the optimized version improves performance**

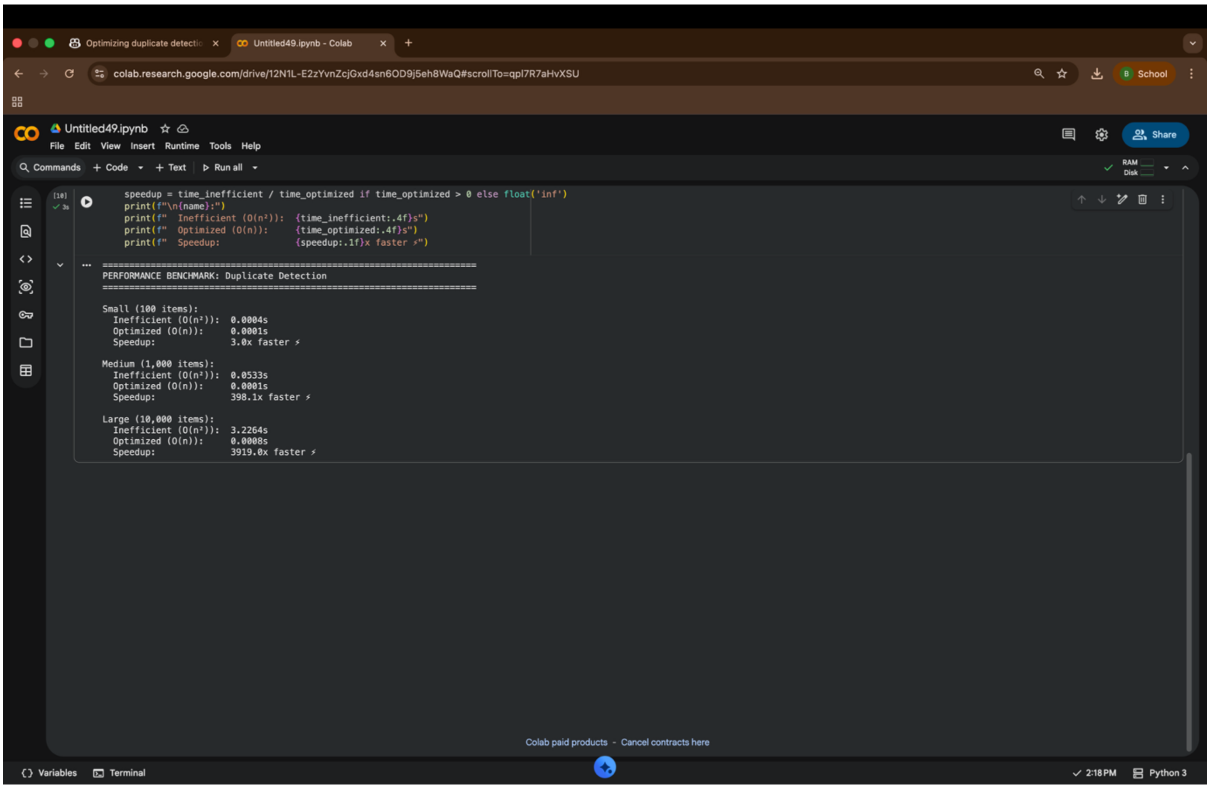
Expected Outcome

- **Optimized duplicate-detection logic (e.g., using sets or hash-based structures)**
- **Improved time complexity**
- **AI explanation of performance improvement**
- **Clean, readable implementation**

PROMPT:

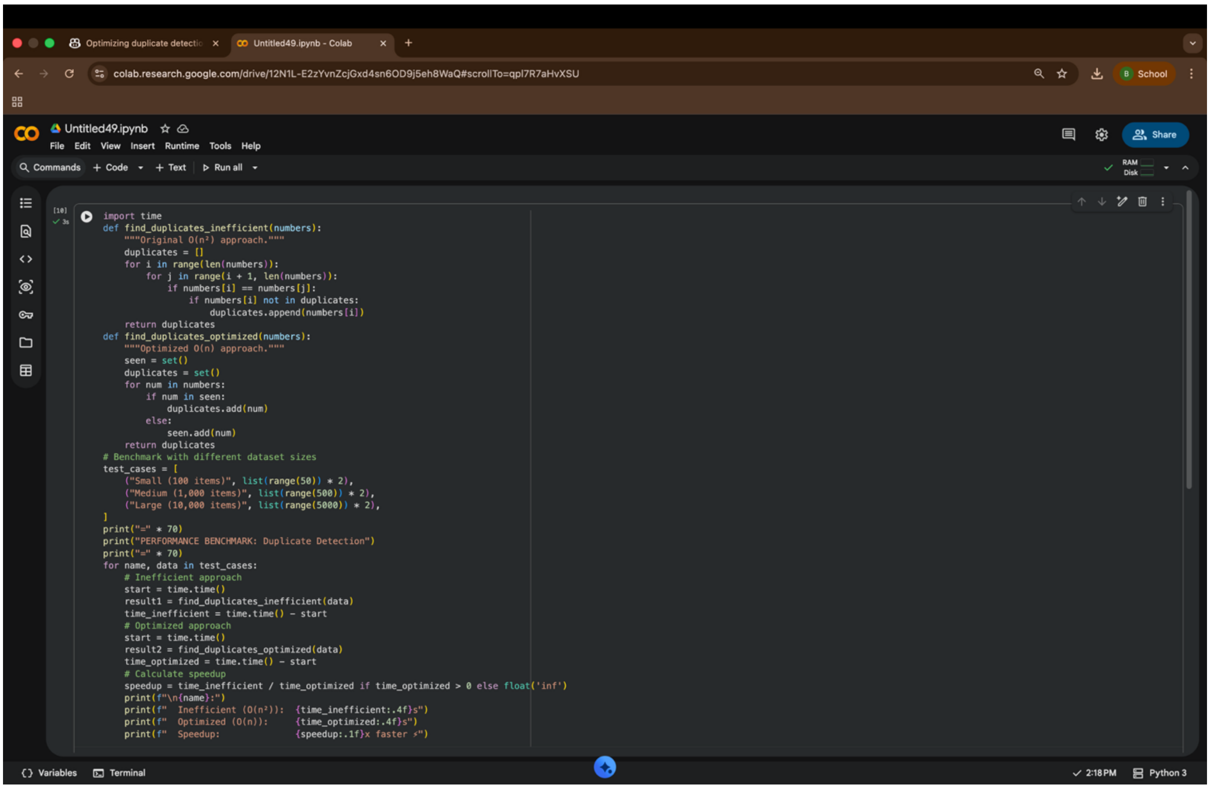
Analyze the performance bottleneck, refactor the code to improve time complexity,and preserve the correctness of the output.Use an optimized approach such as sets or hash-based structures.

CODE AND OUTPUT:


Optimizing duplicate detectio
Untitled49.ipynb - Colab
colab.research.google.com/drive/12N1L-E2zYvnZcjGxd4sn6OD9j5eh8WaQ#scrollTo=qpl7R7aHvXSU
School
Untitled49.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all
Share
RAM
Disk
[10]
3s
Colab paid products - Cancel contracts here
Variables
Terminal
2:18 PM
Python 3

```python
    speedup = time_inefficient / time_optimized if time_optimized > 0 else float('inf')
    print(f"\n{name}:")
    print(f"  Inefficient (O(n²)):  {time_inefficient:.4f}s")
    print(f"  Optimized (O(n)):     {time_optimized:.4f}s")
    print(f"  Speedup:              {speedup:.1f}x faster ⚡")
```

```
======================================================================
PERFORMANCE BENCHMARK: Duplicate Detection
======================================================================

Small (100 items):
  Inefficient (O(n²)):  0.0004s
  Optimized (O(n)):     0.0001s
  Speedup:              3.0x faster ⚡

Medium (1,000 items):
  Inefficient (O(n²)):  0.0533s
  Optimized (O(n)):     0.0001s
  Speedup:              398.1x faster ⚡

Large (10,000 items):
  Inefficient (O(n²)):  3.2264s
  Optimized (O(n)):     0.0008s
  Speedup:              3919.0x faster ⚡
```


Optimizing duplicate detectio
Untitled49.ipynb - Colab
colab.research.google.com/drive/12N1L-E2zYvnZcjGxd4sn6OD9j5eh8WaQ#scrollTo=qpl7R7aHvXSU
School
Untitled49.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all
Share
RAM
Disk
[10]
3s
Variables
Terminal
2:18 PM
Python 3

```python
import time
def find_duplicates_inefficient(numbers):
    """Original O(n²) approach."""
    duplicates = []
    for i in range(len(numbers)):
        for j in range(i + 1, len(numbers)):
            if numbers[i] == numbers[j]:
                if numbers[i] not in duplicates:
                    duplicates.append(numbers[i])
    return duplicates
def find_duplicates_optimized(numbers):
    """Optimized O(n) approach."""
    seen = set()
    duplicates = set()
    for num in numbers:
        if num in seen:
            duplicates.add(num)
        else:
            seen.add(num)
    return duplicates
# Benchmark with different dataset sizes
test_cases = [
    ("Small (100 items)", list(range(50)) * 2),
    ("Medium (1,000 items)", list(range(500)) * 2),
    ("Large (10,000 items)", list(range(5000)) * 2),
]
print("=" * 70)
print("PERFORMANCE BENCHMARK: Duplicate Detection")
print("=" * 70)
for name, data in test_cases:
    # Inefficient approach
    start = time.time()
    result1 = find_duplicates_inefficient(data)
    time_inefficient = time.time() - start
    # Optimized approach
    start = time.time()
    result2 = find_duplicates_optimized(data)
    time_optimized = time.time() - start
    # Calculate speedup
    speedup = time_inefficient / time_optimized if time_optimized > 0 else float('inf')
    print(f"\n{name}:")
    print(f"  Inefficient (O(n²)):  {time_inefficient:.4f}s")
    print(f"  Optimized (O(n)):     {time_optimized:.4f}s")
    print(f"  Speedup:              {speedup:.1f}x faster ⚡")
```

CODE EXPLANATION:

The original implementation uses nested loops to compare each element with every other element in the list, resulting in $O(n^2)$ time complexity. This approach becomes inefficient and slow when handling large datasets because the number of comparisons increases significantly as the list grows. The AI identifies this performance bottleneck and replaces the nested loops with a set-based approach, which allows constant-time lookups. By scanning the list only once and storing seen elements in a set, duplicates can be detected in $O(n)$ time. This optimization improves efficiency while preserving the correctness of the output and producing cleaner, more readable code.

Task 3: Readability and Maintainability Refactoring

Scenario

A working script exists in a project, but it is difficult to understand due to poor naming, formatting, and structure. The team wants it rewritten for long-term maintainability.

Task Description

You are given a poorly structured Python function with:

- **Cryptic function names**
- **Poor indentation**
- **Unclear variable naming**
- **No documentation**

Use AI-assisted review to:

- **Refactor the code for clarity**
- **Apply PEP 8 formatting standards**
- **Improve naming conventions**
- **Add meaningful documentation**

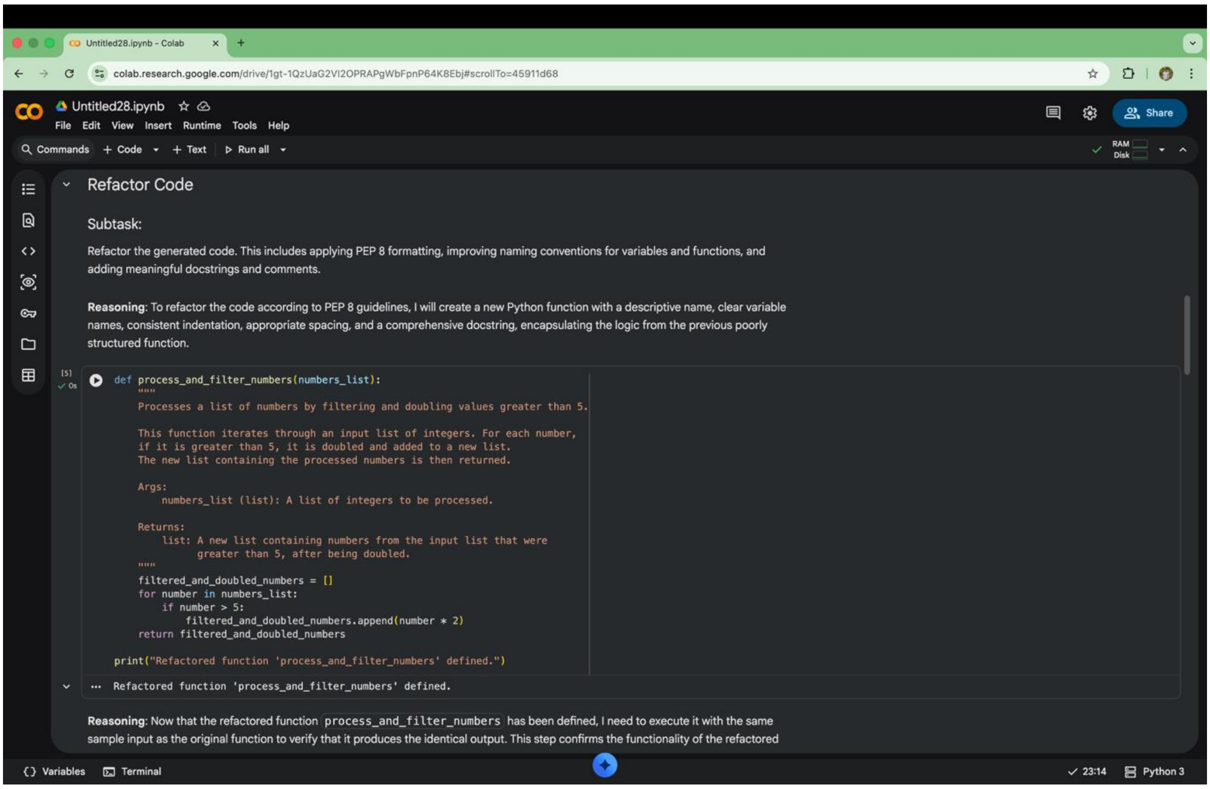
Expected Outcome

- **Clean, well-structured code**
- **Descriptive function and variable names**
- **Proper indentation and formatting**
- **Docstrings explaining the function purpose**
- **AI explanation of readability improvements**

PROMPT:

Improve naming conventions, apply PEP 8 formatting,fix indentation, and add a meaningful docstring.Explain the readability and maintainability improvements made.

CODE AND OUPUT:


Untitled28.ipynb - Colab
colab.research.google.com/drive/1gt-1QzUaG2VI2OPRAPgWbFpnP64K8Ebj#scrollTo=45911d68
Untitled28.ipynb
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text ▷ Run all
RAM
Disk
Variables
Terminal
✓ 23:14
Python 3

Refactor Code

Subtask:

Refactor the generated code. This includes applying PEP 8 formatting, improving naming conventions for variables and functions, and adding meaningful docstrings and comments.

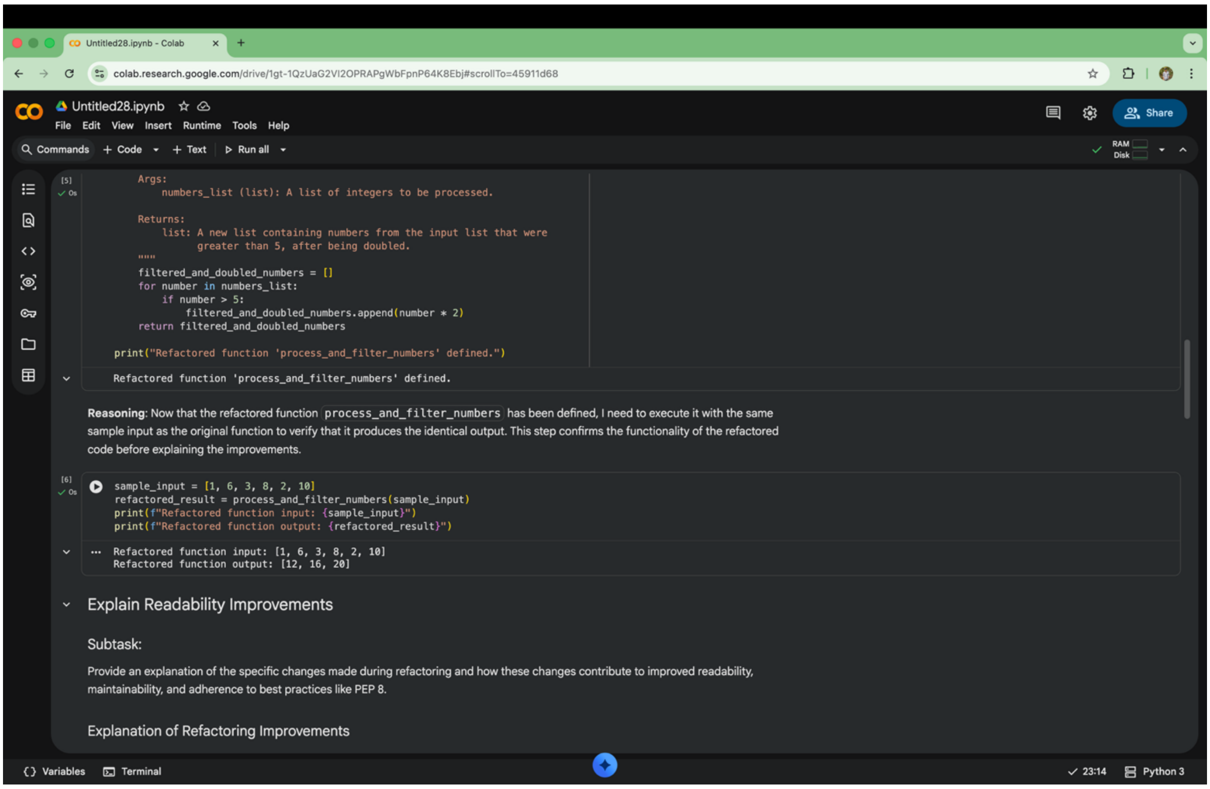
Reasoning: To refactor the code according to PEP 8 guidelines, I will create a new Python function with a descriptive name, clear variable names, consistent indentation, appropriate spacing, and a comprehensive docstring, encapsulating the logic from the previous poorly structured function.

[5]

```
def process_and_filter_numbers(numbers_list):
    """
    Processes a list of numbers by filtering and doubling values greater than 5.

    This function iterates through an input list of integers. For each number,
    if it is greater than 5, it is doubled and added to a new list.
    The new list containing the processed numbers is then returned.

    Args:
        numbers_list (list): A list of integers to be processed.

    Returns:
        list: A new list containing numbers from the input list that were
              greater than 5, after being doubled.
    """
    filtered_and_doubled_numbers = []
    for number in numbers_list:
        if number > 5:
            filtered_and_doubled_numbers.append(number * 2)
    return filtered_and_doubled_numbers

print("Refactored function 'process_and_filter_numbers' defined.")
```

```
Refactored function 'process_and_filter_numbers' defined.
```

Reasoning: Now that the refactored function `process_and_filter_numbers` has been defined, I need to execute it with the same sample input as the original function to verify that it produces the identical output. This step confirms the functionality of the refactored


Untitled28.ipynb - Colab
colab.research.google.com/drive/1gt-1QzUaG2VI2OPRAPgWbFpnP64K8Ebj#scrollTo=45911d68
Untitled28.ipynb
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text ▷ Run all
RAM
Disk
Variables
Terminal
✓ 23:14
Python 3

[5]

```
    Args:
        numbers_list (list): A list of integers to be processed.

    Returns:
        list: A new list containing numbers from the input list that were
              greater than 5, after being doubled.
    """
    filtered_and_doubled_numbers = []
    for number in numbers_list:
        if number > 5:
            filtered_and_doubled_numbers.append(number * 2)
    return filtered_and_doubled_numbers

print("Refactored function 'process_and_filter_numbers' defined.")
```

```
Refactored function 'process_and_filter_numbers' defined.
```

Reasoning: Now that the refactored function `process_and_filter_numbers` has been defined, I need to execute it with the same sample input as the original function to verify that it produces the identical output. This step confirms the functionality of the refactored code before explaining the improvements.

[6]

```
sample_input = [1, 6, 3, 8, 2, 10]
refactored_result = process_and_filter_numbers(sample_input)
print(f"Refactored function input: {sample_input}")
print(f"Refactored function output: {refactored_result}")
```

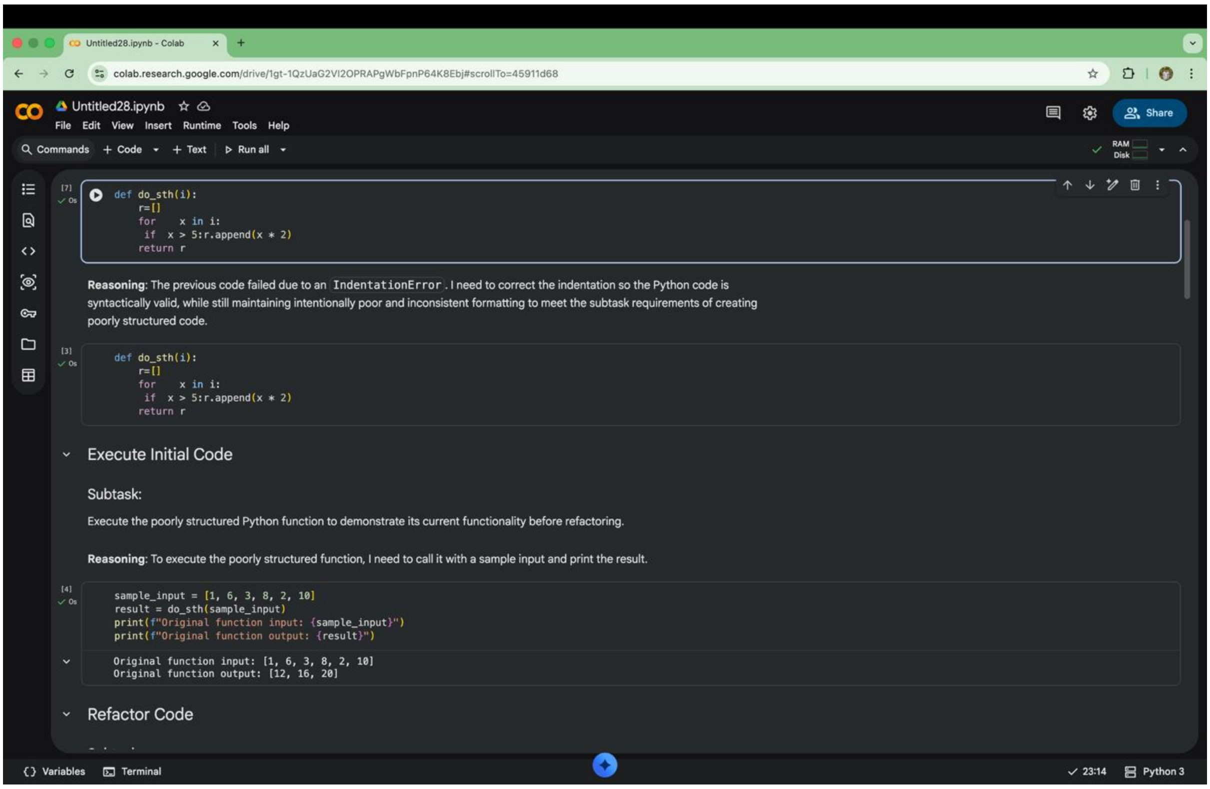
```
Refactored function input: [1, 6, 3, 8, 2, 10]
Refactored function output: [12, 16, 20]
```

Explain Readability Improvements

Subtask:

Provide an explanation of the specific changes made during refactoring and how these changes contribute to improved readability, maintainability, and adherence to best practices like PEP 8.

Explanation of Refactoring Improvements



```python
def do_sth(i):
    r=[]
    for    x in i:
     if  x > 5:r.append(x * 2)
    return r
```

Reasoning: The previous code failed due to an `IndentationError`. I need to correct the indentation so the Python code is syntactically valid, while still maintaining intentionally poor and inconsistent formatting to meet the subtask requirements of creating poorly structured code.

```python
def do_sth(i):
    r=[]
    for    x in i:
     if  x > 5:r.append(x * 2)
    return r
```

Execute Initial Code

Subtask:

Execute the poorly structured Python function to demonstrate its current functionality before refactoring.

Reasoning: To execute the poorly structured function, I need to call it with a sample input and print the result.

```python
sample_input = [1, 6, 3, 8, 2, 10]
result = do_sth(sample_input)
print(f"Original function input: {sample_input}")
print(f"Original function output: {result}")
```

```
Original function input: [1, 6, 3, 8, 2, 10]
Original function output: [12, 16, 20]
```

Refactor Code

CODE EXPLANATION:

Original Issue: The code had cryptic names, poor formatting, and no documentation, making it hard to understand.
Refactoring Process: We applied PEP 8, giving functions and variables clear, descriptive names.
Formatting: Indentation was standardized to 4 spaces, and proper spacing was added for visual clarity.
Documentation: A comprehensive docstring was added to explain the function's purpose, arguments, and return values.
Outcome: The refactored code is now highly readable, maintainable, and adheres to Python best practices while retaining its original functionality**.**

Task 4: Secure Coding and Reliability Review

Scenario

A backend function retrieves user data from a database but has security vulnerabilities and poor error handling, making it unsafe for production deployment.

Task Description

You are given a Python script that:

- **Uses unsafe SQL query construction**
- **Has no input validation**
- **Lacks exception handling**

Use AI tools to:

- **Identify security vulnerabilities**

- **Refactor the code using safe coding practices**
- **Add proper exception handling**
- **Improve robustness and reliability**

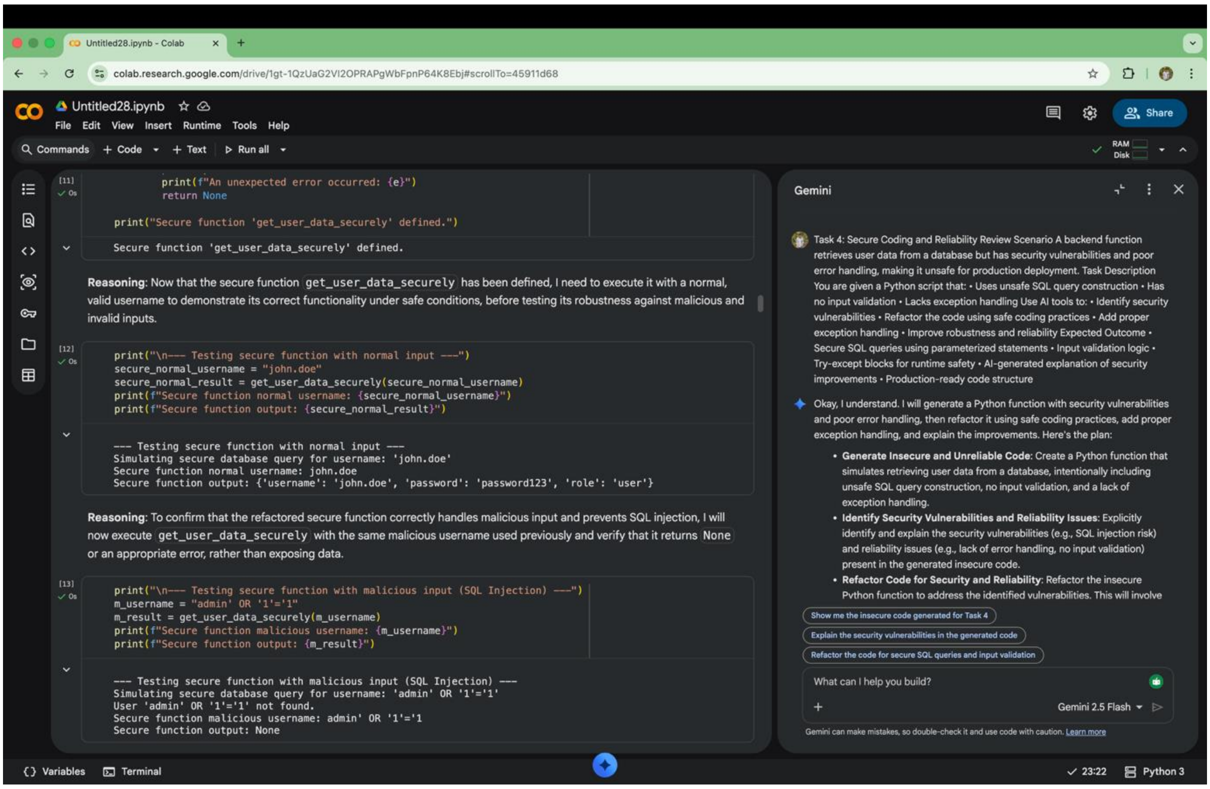
Expected Outcome

- **Secure SQL queries using parameterized statements**
- **Input validation logic**
- **Try-except blocks for runtime safety**
- **AI-generated explanation of security improvements**
- **Production-ready code structure**

PROMPT:

Review the following Python database function for security vulnerabilities. Identify unsafe SQL query construction, missing input validation,and lack of exception handling.

CODE AND OUTPUT:


Untitled28.ipynb - Colab
colab.research.google.com/drive/1gt-1QzUaG2VI2OPRAPgWbFpnP64K8Ebj#scrollTo=45911d68
Untitled28.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all
Share
RAM
Disk
Gemini
Variables
Terminal
23:22
Python 3

[11]

```
        print(f"An unexpected error occurred: {e}")
        return None

print("Secure function 'get_user_data_securely' defined.")
```

```
Secure function 'get_user_data_securely' defined.
```

Reasoning: Now that the secure function `get_user_data_securely` has been defined, I need to execute it with a normal, valid username to demonstrate its correct functionality under safe conditions, before testing its robustness against malicious and invalid inputs.

[12]

```
print("\n--- Testing secure function with normal input ---")
secure_normal_username = "john.doe"
secure_normal_result = get_user_data_securely(secure_normal_username)
print(f"Secure function normal username: {secure_normal_username}")
print(f"Secure function output: {secure_normal_result}")
```

```
--- Testing secure function with normal input ---
Simulating secure database query for username: 'john.doe'
Secure function normal username: john.doe
Secure function output: {'username': 'john.doe', 'password': 'password123', 'role': 'user'}
```

Reasoning: To confirm that the refactored secure function correctly handles malicious input and prevents SQL injection, I will now execute `get_user_data_securely` with the same malicious username used previously and verify that it returns `None` or an appropriate error, rather than exposing data.

[13]

```
print("\n--- Testing secure function with malicious input (SQL Injection) ---")
m_username = "admin' OR '1'='1"
m_result = get_user_data_securely(m_username)
print(f"Secure function malicious username: {m_username}")
print(f"Secure function output: {m_result}")
```

```
--- Testing secure function with malicious input (SQL Injection) ---
Simulating secure database query for username: 'admin' OR '1'='1'
User 'admin' OR '1'='1' not found.
Secure function malicious username: admin' OR '1'='1
Secure function output: None
```

Task 4: Secure Coding and Reliability Review Scenario A backend function retrieves user data from a database but has security vulnerabilities and poor error handling, making it unsafe for production deployment. Task Description You are given a Python script that: • Uses unsafe SQL query construction • Has no input validation • Lacks exception handling Use AI tools to: • Identify security vulnerabilities • Refactor the code using safe coding practices • Add proper exception handling • Improve robustness and reliability Expected Outcome • Secure SQL queries using parameterized statements • Input validation logic • Try-except blocks for runtime safety • AI-generated explanation of security improvements • Production-ready code structure

Okay, I understand. I will generate a Python function with security vulnerabilities and poor error handling, then refactor it using safe coding practices, add proper exception handling, and explain the improvements. Here's the plan:

- **Generate Insecure and Unreliable Code**: Create a Python function that simulates retrieving user data from a database, intentionally including unsafe SQL query construction, no input validation, and a lack of exception handling.
- **Identify Security Vulnerabilities and Reliability Issues**: Explicitly identify and explain the security vulnerabilities (e.g., SQL injection risk) and reliability issues (e.g., lack of error handling, no input validation) present in the generated insecure code.
- **Refactor Code for Security and Reliability**: Refactor the insecure Python function to address the identified vulnerabilities. This will involve

Show me the insecure code generated for Task 4

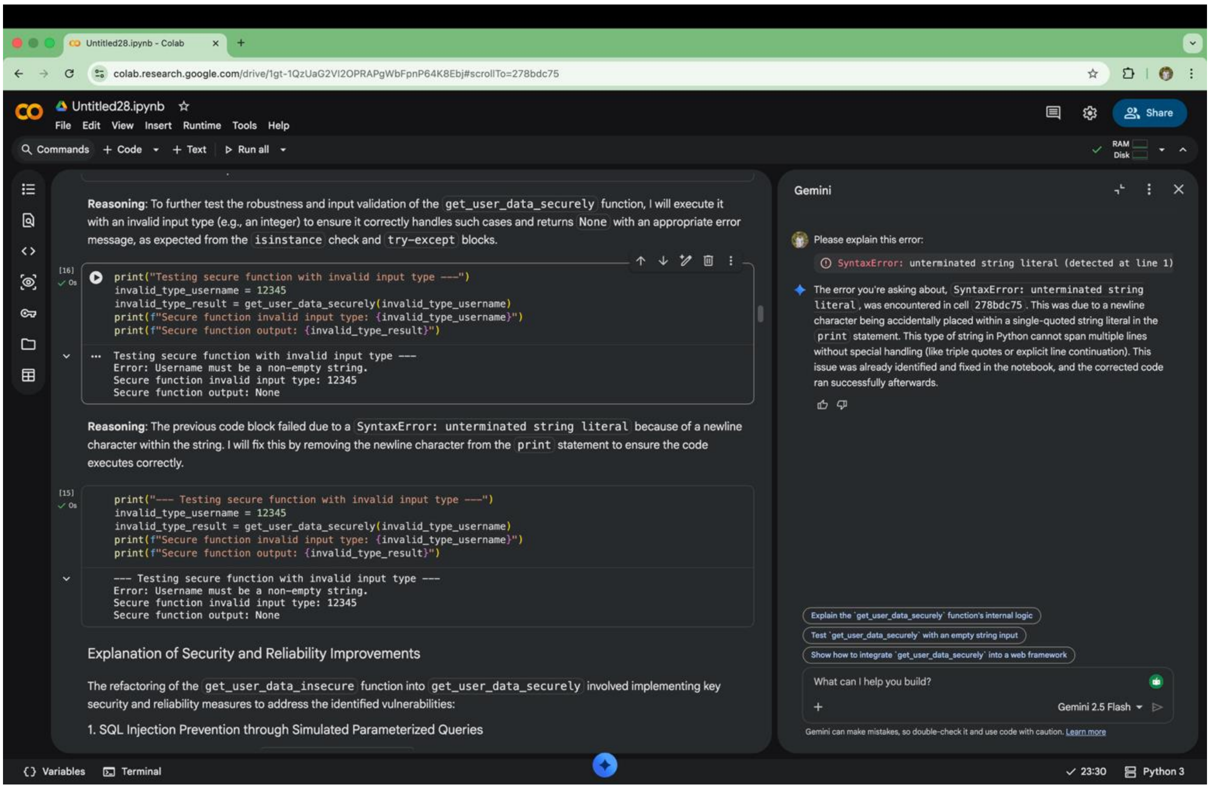
Explain the security vulnerabilities in the generated code

Refactor the code for secure SQL queries and input validation

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


Untitled28.ipynb - Colab
colab.research.google.com/drive/1gt-1QzUaG2VI2OPRAPgWbFpnP64K8Ebj#scrollTo=278bdc75
Untitled28.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text ▷ Run all
Share
RAM
Disk
Variables
Terminal
23:30
Python 3

Reasoning: To further test the robustness and input validation of the `get_user_data_securely` function, I will execute it with an invalid input type (e.g., an integer) to ensure it correctly handles such cases and returns `None` with an appropriate error message, as expected from the `isinstance` check and `try-except` blocks.

[16]
```
print("Testing secure function with invalid input type ---")
invalid_type_username = 12345
invalid_type_result = get_user_data_securely(invalid_type_username)
print(f"Secure function invalid input type: {invalid_type_username}")
print(f"Secure function output: {invalid_type_result}")
```

```
Testing secure function with invalid input type ---
Error: Username must be a non-empty string.
Secure function invalid input type: 12345
Secure function output: None
```

Reasoning: The previous code block failed due to a `SyntaxError: unterminated string literal` because of a newline character within the string. I will fix this by removing the newline character from the `print` statement to ensure the code executes correctly.

[15]
```
print("--- Testing secure function with invalid input type ---")
invalid_type_username = 12345
invalid_type_result = get_user_data_securely(invalid_type_username)
print(f"Secure function invalid input type: {invalid_type_username}")
print(f"Secure function output: {invalid_type_result}")
```

```
--- Testing secure function with invalid input type ---
Error: Username must be a non-empty string.
Secure function invalid input type: 12345
Secure function output: None
```

Explanation of Security and Reliability Improvements

The refactoring of the `get_user_data_insecure` function into `get_user_data_securely` involved implementing key security and reliability measures to address the identified vulnerabilities:

1. SQL Injection Prevention through Simulated Parameterized Queries

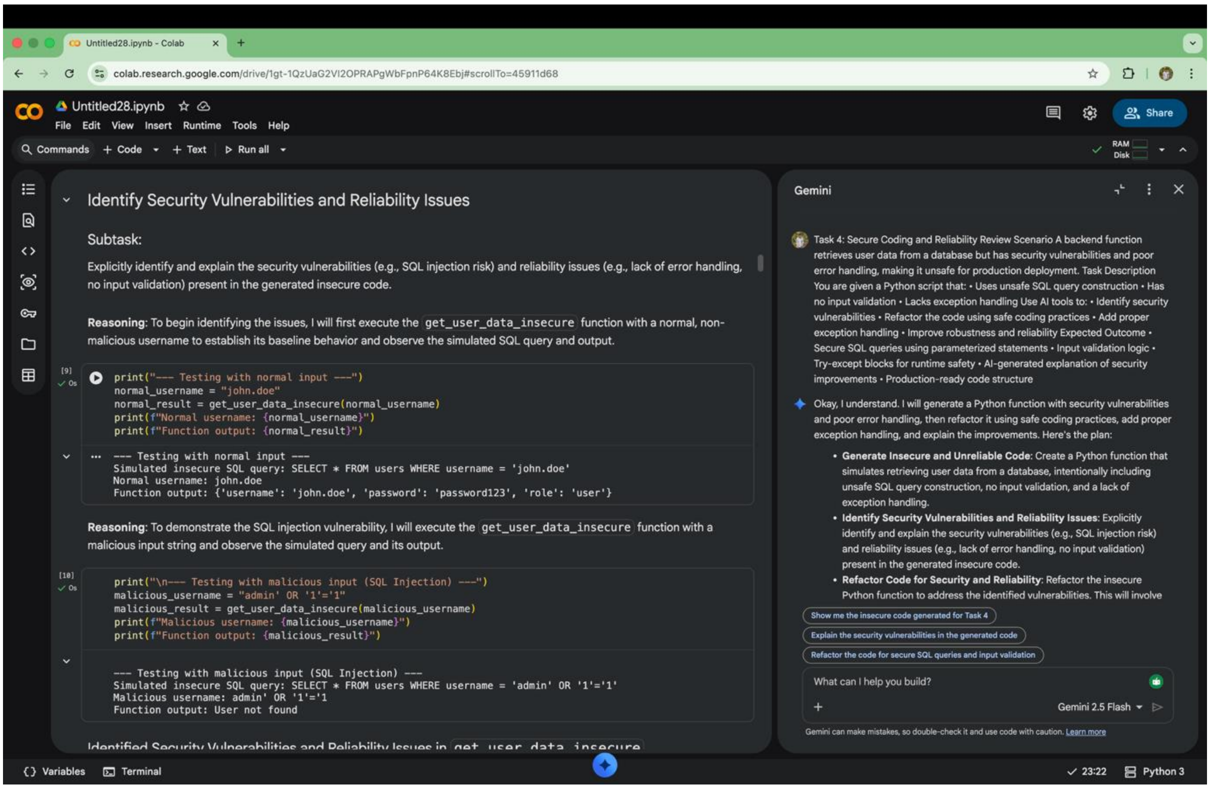
Gemini

Please explain this error:

```
SyntaxError: unterminated string literal (detected at line 1)
```

The error you're asking about, `SyntaxError: unterminated string literal`, was encountered in cell `278bdc75`. This was due to a newline character being accidentally placed within a single-quoted string literal in the `print` statement. This type of string in Python cannot span multiple lines without special handling (like triple quotes or explicit line continuation). This issue was already identified and fixed in the notebook, and the corrected code ran successfully afterwards.

Explain the `get_user_data_securely` function's internal logic
Test `get_user_data_securely` with an empty string input
Show how to integrate `get_user_data_securely` into a web framework
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


Untitled28.ipynb - Colab
colab.research.google.com/drive/1gt-1QzUaG2VI2OPRAPgWbFpnP64K8Ebj#scrollTo=45911d68
Untitled28.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text ▷ Run all
Share
RAM
Disk
Variables
Terminal
23:22
Python 3

Identify Security Vulnerabilities and Reliability Issues

Subtask:

Explicitly identify and explain the security vulnerabilities (e.g., SQL injection risk) and reliability issues (e.g., lack of error handling, no input validation) present in the generated insecure code.

Reasoning: To begin identifying the issues, I will first execute the `get_user_data_insecure` function with a normal, non-malicious username to establish its baseline behavior and observe the simulated SQL query and output.

[9]
```
print("--- Testing with normal input ---")
normal_username = "john.doe"
normal_result = get_user_data_insecure(normal_username)
print(f"Normal username: {normal_username}")
print(f"Function output: {normal_result}")
```

```
--- Testing with normal input ---
Simulated insecure SQL query: SELECT * FROM users WHERE username = 'john.doe'
Normal username: john.doe
Function output: {'username': 'john.doe', 'password': 'password123', 'role': 'user'}
```

Reasoning: To demonstrate the SQL injection vulnerability, I will execute the `get_user_data_insecure` function with a malicious input string and observe the simulated query and its output.

[10]
```
print("\n--- Testing with malicious input (SQL Injection) ---")
malicious_username = "admin' OR '1'='1"
malicious_result = get_user_data_insecure(malicious_username)
print(f"Malicious username: {malicious_username}")
print(f"Function output: {malicious_result}")
```

```

--- Testing with malicious input (SQL Injection) ---
Simulated insecure SQL query: SELECT * FROM users WHERE username = 'admin' OR '1'='1'
Malicious username: admin' OR '1'='1
Function output: User not found
```

Identified Security Vulnerabilities and Reliability Issues in `get_user_data_insecure`

Gemini

Task 4: Secure Coding and Reliability Review Scenario A backend function retrieves user data from a database but has security vulnerabilities and poor error handling, making it unsafe for production deployment. Task Description You are given a Python script that: • Uses unsafe SQL query construction • Has no input validation • Lacks exception handling Use AI tools to: • Identify security vulnerabilities • Refactor the code using safe coding practices • Add proper exception handling • Improve robustness and reliability Expected Outcome • Secure SQL queries using parameterized statements • Input validation logic • Try-except blocks for runtime safety • AI-generated explanation of security improvements • Production-ready code structure

Okay, I understand. I will generate a Python function with security vulnerabilities and poor error handling, then refactor it using safe coding practices, add proper exception handling, and explain the improvements. Here's the plan:

- **Generate Insecure and Unreliable Code**: Create a Python function that simulates retrieving user data from a database, intentionally including unsafe SQL query construction, no input validation, and a lack of exception handling.
- **Identify Security Vulnerabilities and Reliability Issues**: Explicitly identify and explain the security vulnerabilities (e.g., SQL injection risk) and reliability issues (e.g., lack of error handling, no input validation) present in the generated insecure code.
- **Refactor Code for Security and Reliability**: Refactor the insecure Python function to address the identified vulnerabilities. This will involve

Show me the insecure code generated for Task 4
Explain the security vulnerabilities in the generated code
Refactor the code for secure SQL queries and input validation
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

CODE EXPLANATION:

SQL Injection Prevention: User input is now treated as data, preventing malicious code injection by simulating parameterized queries.

Robust Input Validation: Explicit checks ensure the username is a non-empty

string, rejecting invalid inputs early to enhance security and prevent errors.
Comprehensive Exception Handling: try-except blocks gracefully manage errors, preventing crashes and providing predictable behavior with informative messages.
Enhanced Reliability: The function is more stable and predictable, operating only with valid data and handling unexpected situations gracefully.
Production-Ready: These changes transform the function into a secure and robust component suitable for real-world application environments.

Task 5: AI-Based Automated Code Review Report

Scenario

Your team uses AI tools to perform automated preliminary code reviews before human review, to improve code quality and consistency across projects.

Task Description

You are provided with a poorly written Python script.

Using AI-assisted review:

- **Generate a structured code review report that evaluates:**
  - **Code readability**
  - **Naming conventions**
  - **Formatting and style consistency**
  - **Error handling**
  - **Documentation quality**
  - **Maintainability**

The task is not just to fix the code, but to analyze and report on quality issues.

Expected Outcome

- **AI-generated review report including:**
  - **Identified quality issues**
  - **Risk areas**
  - **Code smell detection**
  - **Improvement suggestions**
- **Optional improved version of the code**
- **Demonstration of AI as a code reviewer, not just a code generator**

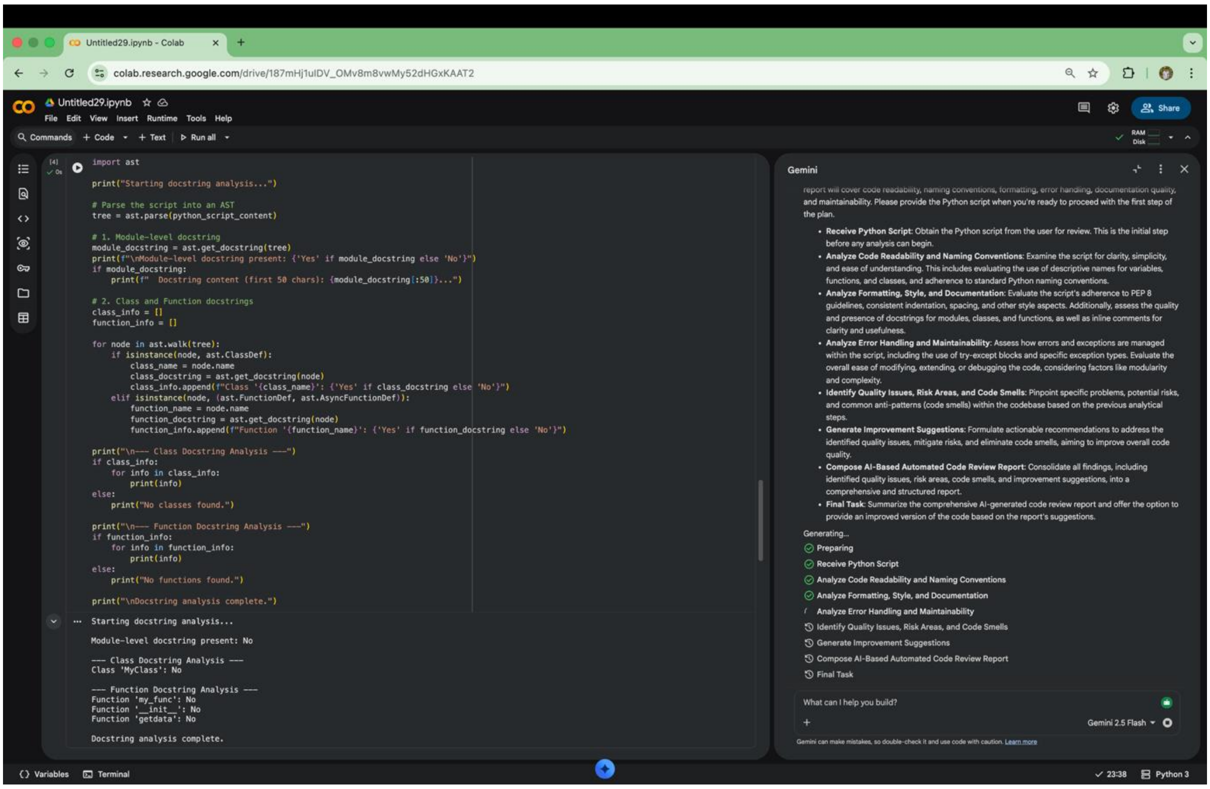
PROMPT:

Perform a structured code review of the following Python script.
Do not just fix the code — generate a detailed review report covering:
- Readability
- Naming conventions
- Formatting and style consistency

- Error handling
- Documentation quality
- Maintainability
- Code smells and risk areas

Provide improvement suggestions and optionally include a refactored version.

CODE AND OUTPUT:


Untitled29.ipynb - Colab
colab.research.google.com/drive/187mHj1uIDV_OMv8m8vwMy52dHGxKAAT2
Untitled29.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all
Share
RAM
Disk
[4]
0s
Gemini

```python
import ast

print("Starting docstring analysis...")

# Parse the script into an AST
tree = ast.parse(python_script_content)

# 1. Module-level docstring
module_docstring = ast.get_docstring(tree)
print(f"\nModule-level docstring present: {'Yes' if module_docstring else 'No'}")
if module_docstring:
    print(f"  Docstring content (first 50 chars): {module_docstring[:50]}...")

# 2. Class and Function docstrings
class_info = []
function_info = []

for node in ast.walk(tree):
    if isinstance(node, ast.ClassDef):
        class_name = node.name
        class_docstring = ast.get_docstring(node)
        class_info.append(f"Class '{class_name}': {'Yes' if class_docstring else 'No'}")
    elif isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
        function_name = node.name
        function_docstring = ast.get_docstring(node)
        function_info.append(f"Function '{function_name}': {'Yes' if function_docstring else 'No'}")

print("\n--- Class Docstring Analysis ---")
if class_info:
    for info in class_info:
        print(info)
else:
    print("No classes found.")

print("\n--- Function Docstring Analysis ---")
if function_info:
    for info in function_info:
        print(info)
else:
    print("No functions found.")

print("\nDocstring analysis complete.")
```

```
Starting docstring analysis...

Module-level docstring present: No

--- Class Docstring Analysis ---
Class 'MyClass': No

--- Function Docstring Analysis ---
Function 'my_func': No
Function '__init__': No
Function 'getdata': No

Docstring analysis complete.
```

report will cover code readability, naming conventions, formatting, error handling, documentation quality, and maintainability. Please provide the Python script when you're ready to proceed with the first step of the plan.

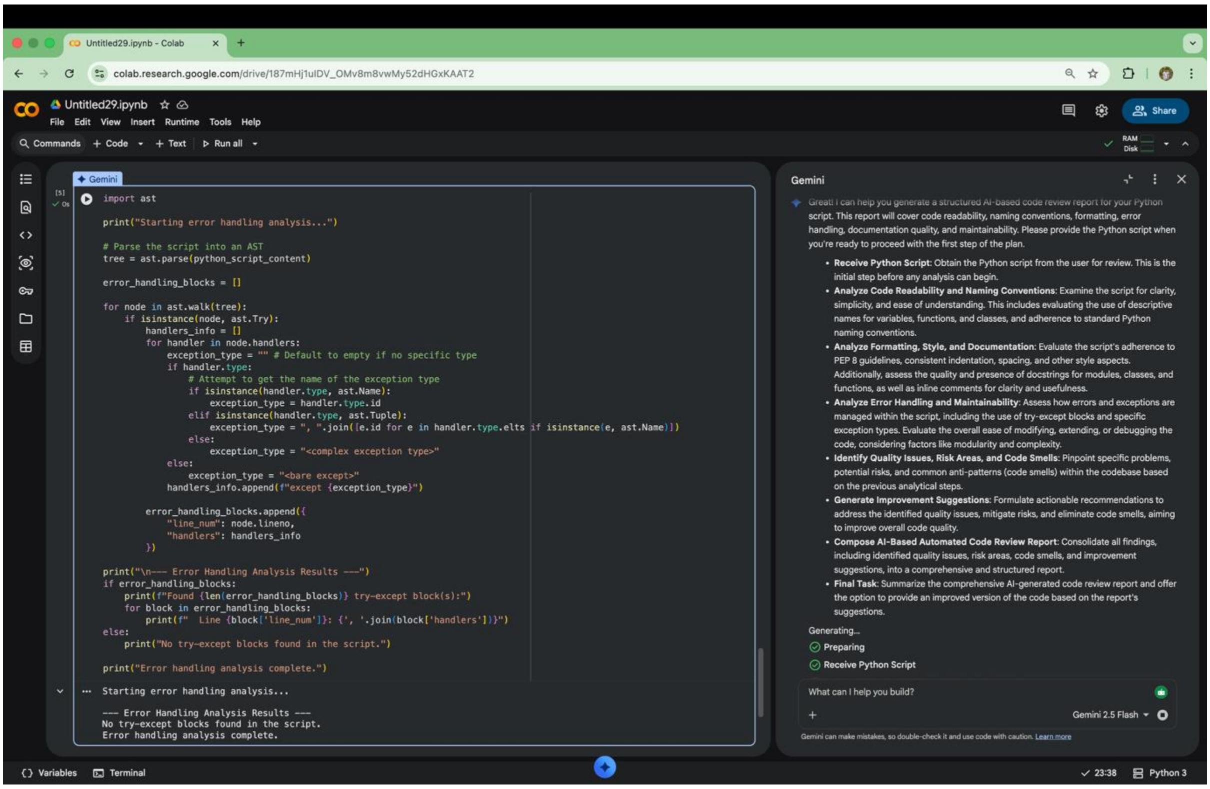
- **Receive Python Script**: Obtain the Python script from the user for review. This is the initial step before any analysis can begin.
- **Analyze Code Readability and Naming Conventions**: Examine the script for clarity, simplicity, and ease of understanding. This includes evaluating the use of descriptive names for variables, functions, and classes, and adherence to standard Python naming conventions.
- **Analyze Formatting, Style, and Documentation**: Evaluate the script's adherence to PEP 8 guidelines, consistent indentation, spacing, and other style aspects. Additionally, assess the quality and presence of docstrings for modules, classes, and functions, as well as inline comments for clarity and usefulness.
- **Analyze Error Handling and Maintainability**: Assess how errors and exceptions are managed within the script, including the use of try-except blocks and specific exception types. Evaluate the overall ease of modifying, extending, or debugging the code, considering factors like modularity and complexity.
- **Identify Quality Issues, Risk Areas, and Code Smells**: Pinpoint specific problems, potential risks, and common anti-patterns (code smells) within the codebase based on the previous analytical steps.
- **Generate Improvement Suggestions**: Formulate actionable recommendations to address the identified quality issues, mitigate risks, and eliminate code smells, aiming to improve overall code quality.
- **Compose AI-Based Automated Code Review Report**: Consolidate all findings, including identified quality issues, risk areas, code smells, and improvement suggestions, into a comprehensive and structured report.
- **Final Task**: Summarize the comprehensive AI-generated code review report and offer the option to provide an improved version of the code based on the report's suggestions.

Generating...

✓ Preparing
✓ Receive Python Script
✓ Analyze Code Readability and Naming Conventions
✓ Analyze Formatting, Style, and Documentation
Analyze Error Handling and Maintainability
Identify Quality Issues, Risk Areas, and Code Smells
Generate Improvement Suggestions
Compose AI-Based Automated Code Review Report
Final Task

What can I help you build?

Gemini can make mistakes, so double-check it and use code with caution. Learn more



```python
import ast

print("Starting error handling analysis...")

# Parse the script into an AST
tree = ast.parse(python_script_content)

error_handling_blocks = []

for node in ast.walk(tree):
    if isinstance(node, ast.Try):
        handlers_info = []
        for handler in node.handlers:
            exception_type = "" # Default to empty if no specific type
            if handler.type:
                # Attempt to get the name of the exception type
                if isinstance(handler.type, ast.Name):
                    exception_type = handler.type.id
                elif isinstance(handler.type, ast.Tuple):
                    exception_type = ", ".join([e.id for e in handler.type.elts if isinstance(e, ast.Name)])
                else:
                    exception_type = "<complex exception type>"
            else:
                exception_type = "<bare except>"
            handlers_info.append(f"except {exception_type}")

        error_handling_blocks.append({
            "line_num": node.lineno,
            "handlers": handlers_info
        })

print("\n--- Error Handling Analysis Results ---")
if error_handling_blocks:
    print(f"Found {len(error_handling_blocks)} try-except block(s):")
    for block in error_handling_blocks:
        print(f"  Line {block['line_num']}: {', '.join(block['handlers'])}")
else:
    print("No try-except blocks found in the script.")

print("Error handling analysis complete.")
```

```
Starting error handling analysis...

--- Error Handling Analysis Results ---
No try-except blocks found in the script.
Error handling analysis complete.
```

Great! I can help you generate a structured AI-based code review report for your Python script. This report will cover code readability, naming conventions, formatting, error handling, documentation quality, and maintainability. Please provide the Python script when you're ready to proceed with the first step of the plan.

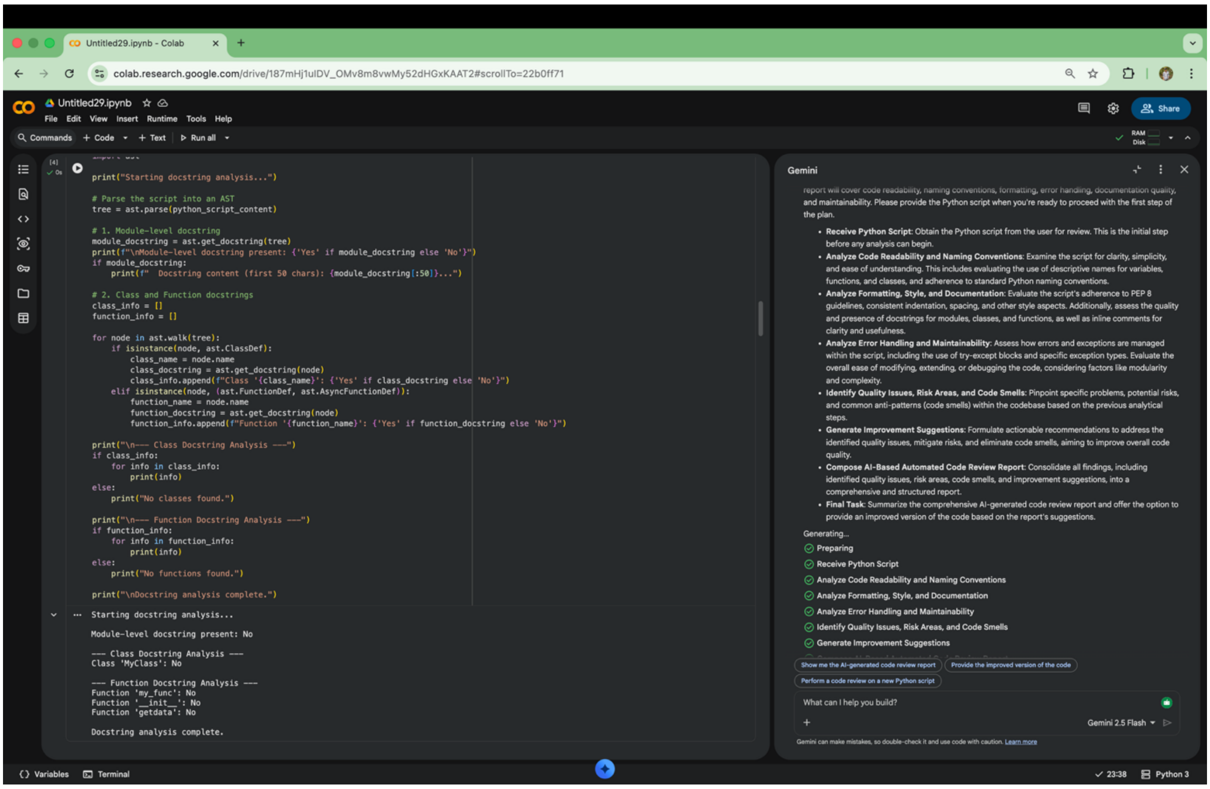
- **Receive Python Script**: Obtain the Python script from the user for review. This is the initial step before any analysis can begin.
- **Analyze Code Readability and Naming Conventions**: Examine the script for clarity, simplicity, and ease of understanding. This includes evaluating the use of descriptive names for variables, functions, and classes, and adherence to standard Python naming conventions.
- **Analyze Formatting, Style, and Documentation**: Evaluate the script's adherence to PEP 8 guidelines, consistent indentation, spacing, and other style aspects. Additionally, assess the quality and presence of docstrings for modules, classes, and functions, as well as inline comments for clarity and usefulness.
- **Analyze Error Handling and Maintainability**: Assess how errors and exceptions are managed within the script, including the use of try-except blocks and specific exception types. Evaluate the overall ease of modifying, extending, or debugging the code, considering factors like modularity and complexity.
- **Identify Quality Issues, Risk Areas, and Code Smells**: Pinpoint specific problems, potential risks, and common anti-patterns (code smells) within the codebase based on the previous analytical steps.
- **Generate Improvement Suggestions**: Formulate actionable recommendations to address the identified quality issues, mitigate risks, and eliminate code smells, aiming to improve overall code quality.
- **Compose AI-Based Automated Code Review Report**: Consolidate all findings, including identified quality issues, risk areas, code smells, and improvement suggestions, into a comprehensive and structured report.
- **Final Task**: Summarize the comprehensive AI-generated code review report and offer the option to provide an improved version of the code based on the report's suggestions.

Generating...

✓ Preparing

✓ Receive Python Script

Gemini can make mistakes, so double-check it and use code with caution. Learn more


Untitled29.ipynb - Colab
colab.research.google.com/drive/187mHj1ulDV_OMv8m8vwMy52dHGxKAAT2#scrollTo=22b0ff71
Untitled29.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all
Share
RAM
Disk
Gemini
[4]
0s
Variables
Terminal
23:38
Python 3

```python
print("Starting docstring analysis...")

# Parse the script into an AST
tree = ast.parse(python_script_content)

# 1. Module-level docstring
module_docstring = ast.get_docstring(tree)
print(f"\nModule-level docstring present: {'Yes' if module_docstring else 'No'}")
if module_docstring:
    print(f"  Docstring content (first 50 chars): {module_docstring[:50]}...")

# 2. Class and Function docstrings
class_info = []
function_info = []

for node in ast.walk(tree):
    if isinstance(node, ast.ClassDef):
        class_name = node.name
        class_docstring = ast.get_docstring(node)
        class_info.append(f"Class '{class_name}': {'Yes' if class_docstring else 'No'}")
    elif isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
        function_name = node.name
        function_docstring = ast.get_docstring(node)
        function_info.append(f"Function '{function_name}': {'Yes' if function_docstring else 'No'}")

print("\n--- Class Docstring Analysis ---")
if class_info:
    for info in class_info:
        print(info)
else:
    print("No classes found.")

print("\n--- Function Docstring Analysis ---")
if function_info:
    for info in function_info:
        print(info)
else:
    print("No functions found.")

print("\nDocstring analysis complete.")
```

```
Starting docstring analysis...

Module-level docstring present: No

--- Class Docstring Analysis ---
Class 'MyClass': No

--- Function Docstring Analysis ---
Function 'my_func': No
Function '__init__': No
Function 'getdata': No

Docstring analysis complete.
```

report will cover code readability, naming conventions, formatting, error handling, documentation quality, and maintainability. Please provide the Python script when you're ready to proceed with the first step of the plan.

- **Receive Python Script**: Obtain the Python script from the user for review. This is the initial step before any analysis can begin.
- **Analyze Code Readability and Naming Conventions**: Examine the script for clarity, simplicity, and ease of understanding. This includes evaluating the use of descriptive names for variables, functions, and classes, and adherence to standard Python naming conventions.
- **Analyze Formatting, Style, and Documentation**: Evaluate the script's adherence to PEP 8 guidelines, consistent indentation, spacing, and other style aspects. Additionally, assess the quality and presence of docstrings for modules, classes, and functions, as well as inline comments for clarity and usefulness.
- **Analyze Error Handling and Maintainability**: Assess how errors and exceptions are managed within the script, including the use of try-except blocks and specific exception types. Evaluate the overall ease of modifying, extending, or debugging the code, considering factors like modularity and complexity.
- **Identify Quality Issues, Risk Areas, and Code Smells**: Pinpoint specific problems, potential risks, and common anti-patterns (code smells) within the codebase based on the previous analytical steps.
- **Generate Improvement Suggestions**: Formulate actionable recommendations to address the identified quality issues, mitigate risks, and eliminate code smells, aiming to improve overall code quality.
- **Compose AI-Based Automated Code Review Report**: Consolidate all findings, including identified quality issues, risk areas, code smells, and improvement suggestions, into a comprehensive and structured report.
- **Final Task**: Summarize the comprehensive AI-generated code review report and offer the option to provide an improved version of the code based on the report's suggestions.

Generating...

✓ Preparing

✓ Receive Python Script

✓ Analyze Code Readability and Naming Conventions

✓ Analyze Formatting, Style, and Documentation

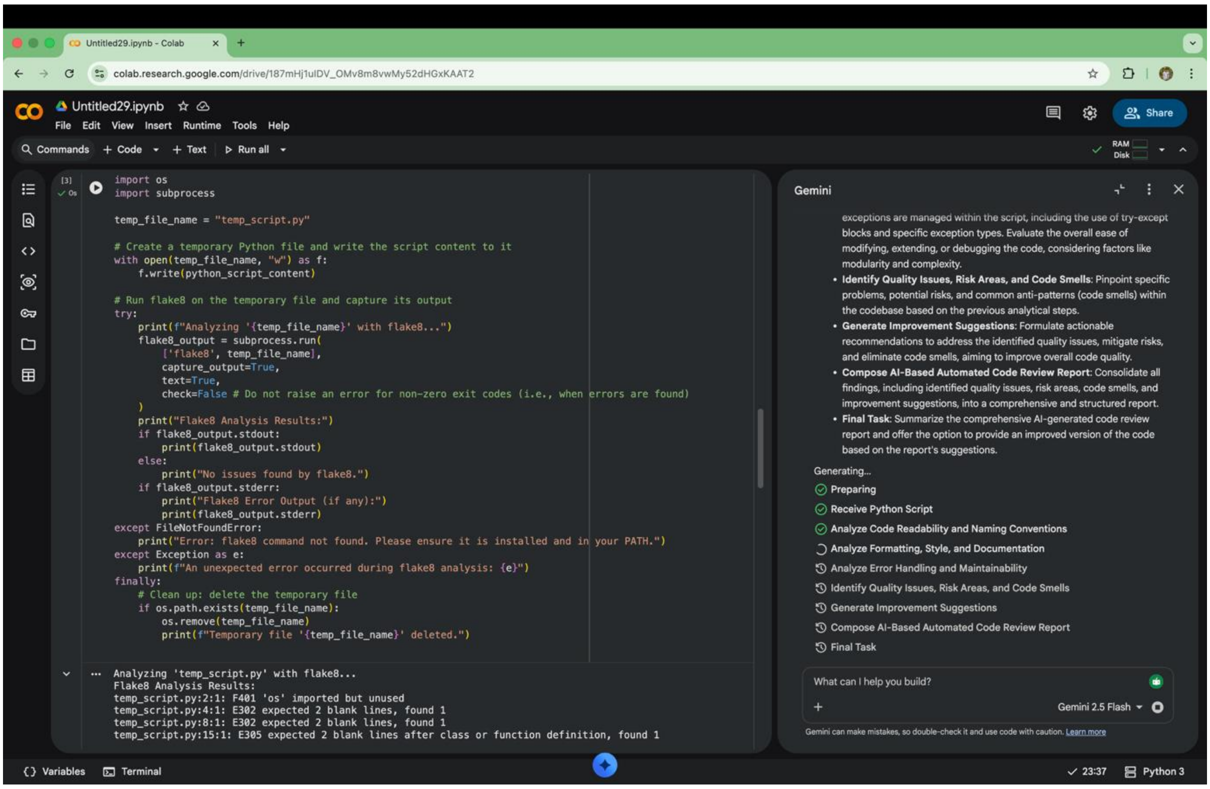
✓ Analyze Error Handling and Maintainability

✓ Identify Quality Issues, Risk Areas, and Code Smells

✓ Generate Improvement Suggestions

Show me the AI-generated code review report
Provide the improved version of the code
Perform a code review on a new Python script
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



```
import os
import subprocess

temp_file_name = "temp_script.py"

# Create a temporary Python file and write the script content to it
with open(temp_file_name, "w") as f:
    f.write(python_script_content)

# Run flake8 on the temporary file and capture its output
try:
    print(f"Analyzing '{temp_file_name}' with flake8...")
    flake8_output = subprocess.run(
        ['flake8', temp_file_name],
        capture_output=True,
        text=True,
        check=False # Do not raise an error for non-zero exit codes (i.e., when errors are found)
    )
    print("Flake8 Analysis Results:")
    if flake8_output.stdout:
        print(flake8_output.stdout)
    else:
        print("No issues found by flake8.")
    if flake8_output.stderr:
        print("Flake8 Error Output (if any):")
        print(flake8_output.stderr)
except FileNotFoundError:
    print("Error: flake8 command not found. Please ensure it is installed and in your PATH.")
except Exception as e:
    print(f"An unexpected error occurred during flake8 analysis: {e}")
finally:
    # Clean up: delete the temporary file
    if os.path.exists(temp_file_name):
        os.remove(temp_file_name)
        print(f"Temporary file '{temp_file_name}' deleted.")
```

```
Analyzing 'temp_script.py' with flake8...
Flake8 Analysis Results:
temp_script.py:2:1: F401 'os' imported but unused
temp_script.py:4:1: E302 expected 2 blank lines, found 1
temp_script.py:8:1: E302 expected 2 blank lines, found 1
temp_script.py:15:1: E305 expected 2 blank lines after class or function definition, found 1
```

exceptions are managed within the script, including the use of try-except blocks and specific exception types. Evaluate the overall ease of modifying, extending, or debugging the code, considering factors like modularity and complexity.

- **Identify Quality Issues, Risk Areas, and Code Smells**: Pinpoint specific problems, potential risks, and common anti-patterns (code smells) within the codebase based on the previous analytical steps.
- **Generate Improvement Suggestions**: Formulate actionable recommendations to address the identified quality issues, mitigate risks, and eliminate code smells, aiming to improve overall code quality.
- **Compose AI-Based Automated Code Review Report**: Consolidate all findings, including identified quality issues, risk areas, code smells, and improvement suggestions, into a comprehensive and structured report.
- **Final Task**: Summarize the comprehensive AI-generated code review report and offer the option to provide an improved version of the code based on the report's suggestions.

Generating...
✓ Preparing
✓ Receive Python Script
✓ Analyze Code Readability and Naming Conventions
Analyze Formatting, Style, and Documentation
Analyze Error Handling and Maintainability
Identify Quality Issues, Risk Areas, and Code Smells
Generate Improvement Suggestions
Compose AI-Based Automated Code Review Report
Final Task

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

CODE EXPLANATION:

Documentation & Style: The script lacks docstrings and violates PEP 8 blank line conventions, hindering readability.
Naming & Imports: ARG_TWO uses an inconsistent naming convention, and the os module is imported but unused.
Error Handling: There is no explicit error handling, making the script fragile to unexpected inputs.
Maintainability Risk: These issues collectively reduce maintainability, increase cognitive load, and risk runtime crashes.
Improvements: Add docstrings, fix naming and formatting, remove unused imports, and implement robust error handling.