

**AssignmentNumber:10.4**

**K.Akshay**

**2303A51817**

<b>Q.No.</b>	<b>Question</b>	<b>Expected Time to complete</b>
1	<p><b>Lab 9 – Code Review and Quality: Using AI to Improve Code Quality and Readability</b></p> <p><b>Lab Objectives</b></p> <ul style="list-style-type: none"><li>• Use AI for automated code review and quality enhancement.</li><li>• Identify and fix syntax, logical, performance, and security issues in Python code.</li><li>• Improve readability and maintainability through structured refactoring and comments.</li><li>• Apply prompt engineering for targeted improvements.</li><li>• Evaluate AI-generated suggestions against PEP 8 standards and software engineering best practices</li></ul>	Week 5
	<p><b>Task 1: AI-Assisted Syntax and Code Quality Review</b></p> <p><b>Scenario</b></p> <p>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.</p> <p><b>Task Description</b></p> <p>You are given a Python script containing:</p> <ul style="list-style-type: none"><li>• Syntax errors</li><li>• Indentation issues</li><li>• Incorrect variable names</li><li>• Faulty function calls</li></ul> <p>Use an AI tool (GitHub Copilot / Cursor AI) to:</p> <ul style="list-style-type: none"><li>• Identify all syntactic and structural errors</li><li>• Correct them systematically</li><li>• Generate an explanation of each fix made</li></ul>	

## 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 this Python script for syntax errors, indentation issues, incorrect variable names, and faulty function calls. Correct the code so it runs properly, make it clean and readable, and provide a brief explanation of each fix (syntax, naming, and structure). Deliver the corrected executable script along with a clear explanation."

Code :

```
▶ def process_numbers(numbers_list):
    total = 0
    count = 0

    for num in numbers_list
        total += num
    count += 1 # Indentation error and incorrect operator
+   for num in numbers_list:
+       # Syntax Fix: Added a colon after the for loop header
+       # Logical Fix: Added a try-except block to handle non-numeric types
+       try:
+           total += num
+           # Syntax Fix: Corrected the operator from += to =
+           # Indentation Fix: Corrected the indentation to be inside the loop
+           count += 1
+       except TypeError:
+           print(f"Skipping non-numeric value: {num}")

        if count > 0:
            avergae = total / count # Typo in 'average'
+           # Naming Fix: Corrected the typo 'avergae' to 'average'
+           average = total / count
            print(f"The sum is: {total}")
            print(f"The average is: {avergae}")
        else:
            print("No numbers to process.")
+           print(f"The average is: {average}")

    else:
+       # Syntax Fix: Added a colon after the else statement
+       print("No numbers to process or all were non-numeric.")

# Example usage with mixed types and an empty list
my_data = [10, 20, 'hello', 40, 50]
another_data = []

+print("Processing my_data:")
process_numbers(my_data)
+print("\nProcessing another_data:")
process_numbers(another_data)

...
File "<tokenize>", line 7
    count += 1 # Indentation error and incorrect operator
    ^
IndentationError: unindent does not match any outer indentation level
```

Next steps: [Explain error](#)

```
[3] 0s  def process_numbers(numbers_list):
        total = 0
        count = 0

        for num in numbers_list:
            # Syntax Fix: Added a colon after the for loop header
            # Logical Fix: Added a try-except block to handle non-numeric types
            try:
                total += num
                # Syntax Fix: Corrected the operator from += to +=
                # Indentation Fix: Corrected the indentation to be inside the loop
                count += 1
            except TypeError:
                print(f"Skipping non-numeric value: {num}")

        if count > 0:
            # Naming Fix: Corrected the typo 'avergae' to 'average'
            average = total / count
            print(f"The sum is: {total}")
            print(f"The average is: {average}")
        else:
            # Syntax Fix: Added a colon after the else statement
            print("No numbers to process or all were non-numeric.")

    # Example usage with mixed types and an empty list
    my_data = [10, 20, 'hello', 40, 50]
    another_data = []
```

```
print("Processing my_data:")
process_numbers(my_data)
print("\nProcessing another_data:")
process_numbers(another_data)
```

```
• Processing my_data:
Skipping non-numeric value: hello
The sum is: 120
The average is: 30.0

Processing another_data:
No numbers to process or all were non-numeric.
```

Explanation : The corrected version transforms a non-functional script prone to crashes into a robust and reliable piece of code that adheres to Python best practices for syntax, readability, and error handling. The primary difference lies in moving from a state of multiple execution-blocking errors to a fully functional and resilient program.

## 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 :

Review the following Python function that detects duplicates in a list using inefficient nested loops. Analyze the logic for performance bottlenecks, refactor it for better time complexity while preserving correctness, and explain why the original approach was inefficient and how the optimized version improves performance. Provide a clean, readable implementation using sets or hash-based structures.

Code :

Original Inefficient Code

Subtask:

Provide the Python function that detects duplicates using an inefficient nested loop approach, as described.

**Reasoning:** The user wants a Python function that inefficiently detects duplicates using nested loops, along with example calls. I will provide a code block that defines the function and demonstrates its usage.

```
def has_duplicates_inefficient(input_list):
    n = len(input_list)
    for i in range(n):
        for j in range(n):
            # Check if elements are the same value but at different indices
            if i != j and input_list[i] == input_list[j]:
                return True
    return False

# Example usage
my_list_with_duplicates = [1, 2, 3, 2]
my_list_without_duplicates = [1, 2, 3, 4]

print(f"Does {my_list_with_duplicates} have duplicates? {has_duplicates_inefficient(my_list_with_duplicates)}")
print(f"Does {my_list_without_duplicates} have duplicates? {has_duplicates_inefficient(my_list_without_duplicates)}")
```

## Original Inefficient Code

### Subtask:

Provide the Python function that detects duplicates using an inefficient nested loop approach, as described.

**Reasoning:** The user wants a Python function that inefficiently detects duplicates using nested loops, along with example calls. I will provide a code block that defines the function and demonstrates its usage.

```
def has_duplicates_inefficient(input_list):
    n = len(input_list)
    for i in range(n):
        for j in range(n):
            # Check if elements are the same value but at different indices
            if i != j and input_list[i] == input_list[j]:
                return True
    return False

# Example usage
my_list_with_duplicates = [1, 2, 3, 2]
my_list_without_duplicates = [1, 2, 3, 4]

print(f"Does {my_list_with_duplicates} have duplicates? {has_duplicates_inefficient(my_list_with_duplicates)}")
print(f"Does {my_list_without_duplicates} have duplicates? {has_duplicates_inefficient(my_list_without_duplicates)})")
```

Does [1, 2, 3, 2] have duplicates? True  
Does [1, 2, 3, 4] have duplicates? False

### Analyze Original Performance

#### Subtask:

Explain the performance bottlenecks of the original nested loop approach, specifically discussing its time complexity ( $O(N^2)$ ) and why it is inefficient for large lists.

The `has_duplicates_inefficient` function uses a nested loop structure to detect duplicates. This design leads to a time complexity of  $O(N^2)$ , where 'N' is the number of elements in the `input_list`.

Explanation of  $O(N^2)$  Time Complexity:

1. **Outer Loop:** The outer loop iterates N times (once for each element `input_list[i]`).
2. **Inner Loop:** For each iteration of the outer loop, the inner loop also iterates N times (once for each element `input_list[j]`).

This means that for every element in the list, the code performs N comparisons. Therefore, the total number of comparisons is approximately  $N * N = N^2$ . As N grows, the number of operations increases quadratically.

Why it is Inefficient for Large Lists:

### Optimized Code Implementation

#### Subtask:

Implement a refactored version of the duplicate detection function using sets or hash-based structures to achieve better time complexity while maintaining correctness. Ensure the code is clean and readable.

**Reasoning:** The subtask requires implementing an optimized duplicate detection function using sets and demonstrating its usage. I will define the function, convert the list to a set, compare lengths, and then test it with the provided example lists.

```
def has_duplicates_optimized(input_list):
    # Convert the list to a set to remove duplicates
    unique_elements = set(input_list)
    # If the length of the original list is different from the set, duplicates were present
    return len(input_list) != len(unique_elements)

# Example usage with the same test lists
# These lists were defined in a previous step
# my_list_with_duplicates = [1, 2, 3, 2]
# my_list_without_duplicates = [1, 2, 3, 4]

print(f"Does {my_list_with_duplicates} have duplicates? {has_duplicates_optimized(my_list_with_duplicates)}")
print(f"Does {my_list_without_duplicates} have duplicates? {has_duplicates_optimized(my_list_without_duplicates)})")
```

... Does [1, 2, 3, 2] have duplicates? True

Explanation : The task successfully demonstrated how to optimize a Python function for detecting duplicates in a list by evolving from an inefficient  $O(N^2)$  nested loop approach to an optimized  $O(N)$  set-based

	<p>approach. The performance difference is substantial, highlighting the critical role of data structure selection in algorithmic efficiency.</p>	
	<p><b>Task 3: Readability and Maintainability Refactoring</b></p> <p><b>Scenario</b></p> <p>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.</p> <p><b>Task Description</b></p> <p>You are given a poorly structured Python function with:</p> <ul style="list-style-type: none"> <li>• Cryptic function names</li> <li>• Poor indentation</li> <li>• Unclear variable naming</li> <li>• No documentation</li> </ul> <p>Use AI-assisted review to:</p> <ul style="list-style-type: none"> <li>• Refactor the code for clarity</li> <li>• Apply PEP 8 formatting standards</li> <li>• Improve naming conventions</li> <li>• Add meaningful documentation</li> </ul> <p><b>Expected Outcome</b></p> <ul style="list-style-type: none"> <li>• Clean, well-structured code</li> <li>• Descriptive function and variable names</li> <li>• Proper indentation and formatting</li> <li>• Docstrings explaining the function purpose</li> <li>• AI explanation of readability improvements</li> </ul> <p>Prompt :</p> <div style="background-color: black; color: white; padding: 10px;"> <p>Review the following Python function that suffers from cryptic names, poor indentation, unclear variable naming, and no documentation. Refactor the code for clarity, apply PEP 8 formatting standards, improve naming conventions, and add meaningful docstrings. Provide a clean, well-structured version of the function and explain how readability and maintainability were improved</p> </div>	

Code :

```
◆ Gemini
11] -def pc(l):
-    r = 0
-    for i in l:
-        if i % 2 == 0:
-            r += 1
-    return r
+def count_even_numbers(number_list):
+    """
+        Counts the number of even integers in a given list.
+
+        Args:
+            number_list (list): A list of integers.
+
+        Returns:
+            int: The total count of even numbers in the list.
+    """
+    even_count = 0
+    for number in number_list:
+        if number % 2 == 0:
+            even_count += 1
+    return even_count
+
```

```
def count_even_numbers(number_list):
    """
    Counts the number of even integers in a given list.

    Args:
        number_list (list): A list of integers.

    Returns:
        int: The total count of even numbers in the list.
    """
    even_count = 0
    for number in number_list:
        if number % 2 == 0:
            even_count += 1
    return even_count
```

Explanation : Better names and documentation drastically reduce the cognitive load for anyone reading or modifying the code. This leads to fewer errors during development, easier debugging, and quicker onboarding for new team members. It exemplifies what clean, maintainable code should look like.

#### 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 script that retrieves user data from a database but uses unsafe SQL query construction, lacks input validation, and has no exception handling. Refactor the code using secure coding practices, add proper input validation, and implement try-except blocks for runtime safety. Provide a production-ready version of the script and explain the security vulnerabilities in the original code, how they were fixed, and how the

Code :

Original Vulnerable Code

Subtask:

Provide the Python script that retrieves user data from a database but uses unsafe SQL query construction, lacks input validation, and has no exception handling.

**Reasoning:** The subtask requires providing a Python script that demonstrates vulnerable SQL query construction, lacks input validation, and has no exception handling. I will create a code block that simulates a database, defines a vulnerable function, and then calls it with both benign and malicious inputs.

```
▶ import re

# 1. Simulate a database as a list of dictionaries
simulated_db = [
    {'id': 1, 'username': 'admin', 'password': 'admin_password123'},
    {'id': 2, 'username': 'alice', 'password': 'alice_secure_password'},
    {'id': 3, 'username': 'bob', 'password': 'bob_pass'}
]

def get_user_data(username):
    """
    Retrieves user data from a simulated database using an unsafe SQL-like query.
    Lacks input validation and exception handling.
    """
    for user in simulated_db:
        if user['username'] == username:
            return user
```

```
Lacks input validation and exception handling.  
***  
# 3. Construct a simulated SQL query string by directly embedding the username  
# This is highly vulnerable to SQL injection  
sql_query = f"SELECT * FROM users WHERE username = '{username}'"  
print(f"\nSimulated Query: {sql_query}")  
  
found_users = []  
# Simulate parsing the WHERE clause for demonstration  
# In a real scenario, this would be handled by the database engine  
match_username = username.replace("'", "") # Simplistic way to extract username part for simulation  
  
# 4. Simulate the execution of this query against the 'database'  
# The vulnerability lies in how 'match_username' is derived and used without proper sanitization  
if "' OR '1='1" in username: # This condition specifically demonstrates the injection  
    print("SQL Injection detected! Bypassing username check due to '1='1' clause.")  
    found_users = simulated_db # In an injected scenario, all data might be returned  
elif "' OR '1='1" not in username:  
    for user in simulated_db:  
        # Simulating the WHERE clause 'username = 'value''  
        if user['username'] == match_username:  
            found_users.append(user)  
  
# 5. Print the user data if a match is found  
if found_users:  
    print("Found user(s):")  
    for user in found_users:  
        print(f" ID: {user['id']}, Username: {user['username']}, Password: {user['password']}")  
  
else:  
    print("No user found for username: '{username}'")  
  
# 6. Call the get_user_data function with a normal username  
print("--- Testing with normal input ---")  
get_user_data('admin')  
get_user_data('alice')  
get_user_data('nonexistent_user')  
  
# 7. Call the get_user_data function with a malicious input designed to demonstrate SQL injection  
print("\n--- Testing with malicious input (SQL Injection) ---")  
malicious_username = "' OR '1='1"  
get_user_data(malicious_username)  
  
--- Testing with normal input ---  
  
Simulated Query: SELECT * FROM users WHERE username = 'admin'  
Found user(s):  
ID: 1, Username: admin, Password: admin_password123  
  
Simulated Query: SELECT * FROM users WHERE username = 'alice'  
Found user(s):  
ID: 2, Username: alice, Password: alice_secure_password  
  
Simulated Query: SELECT * FROM users WHERE username = 'nonexistent_user'  
No user found for username: 'nonexistent_user'  
  
--- Testing with malicious input (SQL Injection) ---  
  
Simulated Query: SELECT * FROM users WHERE username = '' OR '1='1'  
SQL Injection detected! Bypassing username check due to '1='1' clause.  
Found user(s):  
ID: 1, Username: admin, Password: admin_password123  
ID: 2, Username: alice, Password: alice_secure_password  
ID: 3, Username: bob, Password: bob_pass
```

## Improved :

```
import re

# Re-using the simulated database from the previous step
# simulated_db = [
#     {'id': 1, 'username': 'admin', 'password': 'admin_password123'},
#     {'id': 2, 'username': 'alice', 'password': 'alice_secure_password'},
#     {'id': 3, 'username': 'bob', 'password': 'bob_pass'}
# ]

def get_user_data_secure(username):
    """
    Retrieves user data from a simulated database using secure coding practices:
    input validation, simulated parameterized queries, and exception handling.
    """
    print(f"\nAttempting to retrieve data for username: '{username}'")

    try:
        # 1. Input Validation
        if not isinstance(username, str):
            print("Validation Error: Username must be a string.")
            return []
        if not username.strip():
            print("Validation Error: Username cannot be empty.")
            return []
        # Example pattern: alphanumeric characters, underscores, hyphens, min 3, max 20 chars
        if not re.fullmatch(r"^[a-zA-Z0-9_-]{3,20}$", username):
            print(f"Validation Error: Username '{username}' contains invalid characters or has invalid length.")
            print(" Allowed: alphanumeric, underscores, hyphens. Length: 3-20 characters.")

        # In a real app, sensitive data like passwords wouldn't be returned directly
        print(f" ID: {user['id']}, Username: {user['username']}, Password: [REDACTED]") # Password redacted for secure display
        found_users = []
        else:
            print(f"No user found for username: '{username}'")
            return []

        except Exception as e:
            # 4. Comprehensive Error Handling
            print(f"An unexpected error occurred during data retrieval: {e}")
            # In a production system, this error would be logged securely
            return []

    # Example usage with various inputs to demonstrate robustness
    print("--- Testing get_user_data_secure with various inputs ---")

    # Valid inputs
    get_user_data_secure('admin')
    get_user_data_secure('alice')

    # Invalid inputs (validation failure)
    get_user_data_secure('')
    get_user_data_secure(' ')
    get_user_data_secure('user with spaces')
    get_user_data_secure('invalid$char')
    get_user_data_secure('sh') # Too short
    get_user_data_secure('thisisaverylongusernameexceedingthelimit') # Too long
    get_user_data_secure(123) # Not a string

    # Non-existent user
    get_user_data_secure('nonexistent_user')

    # Malicious inputs (previously vulnerable, now handled by validation)
    # Note: These will now be rejected by input validation, not cause injection.
    get_user_data_secure(" OR '1='1")
    get_user_data_secure("admin'--")
    get_user_data_secure(" OR 1=1 --")

    ...
    --- Testing get_user_data_secure with various inputs ---

    Attempting to retrieve data for username: 'admin'
    Found user(s):
        ID: 1, Username: admin, Password: [REDACTED]

    Attempting to retrieve data for username: 'alice'
    Found user(s):
        ID: 2, Username: alice, Password: [REDACTED]

    Attempting to retrieve data for username: ''
    Validation Error: Username cannot be empty.

    Attempting to retrieve data for username: ' '
    Validation Error: Username cannot be empty.

    Attempting to retrieve data for username: 'user with spaces'
    Validation Error: Username 'user with spaces' contains invalid characters or has invalid length.
    Allowed: alphanumeric, underscores, hyphens. Length: 3-20 characters.
```

```

Attempting to retrieve data for username: 'user with spaces'
Validation Error: Username 'user with spaces' contains invalid characters or has invalid length.
    Allowed: alphanumeric, underscores, hyphens. Length: 3-20 characters.

...
Attempting to retrieve data for username: 'invalid$char'
Validation Error: Username 'invalid$char' contains invalid characters or has invalid length.
    Allowed: alphanumeric, underscores, hyphens. Length: 3-20 characters.

Attempting to retrieve data for username: 'sh'
Validation Error: Username 'sh' contains invalid characters or has invalid length.
    Allowed: alphanumeric, underscores, hyphens. Length: 3-20 characters.

Attempting to retrieve data for username: 'thisisaverylongusernameexceedingthelimit'
Validation Error: Username 'thisisaverylongusernameexceedingthelimit' contains invalid characters or has invalid length.
    Allowed: alphanumeric, underscores, hyphens. Length: 3-20 characters.

Attempting to retrieve data for username: '123'
Validation Error: Username must be a string.

Attempting to retrieve data for username: 'nonexistent_user'
No user found for username: 'nonexistent_user'.

Attempting to retrieve data for username: '' OR '1'='1'
Validation Error: Username '' OR '1'='1' contains invalid characters or has invalid length.
    Allowed: alphanumeric, underscores, hyphens. Length: 3-20 characters.

Attempting to retrieve data for username: 'admin--'
Validation Error: Username 'admin--' contains invalid characters or has invalid length.
    Allowed: alphanumeric, underscores, hyphens. Length: 3-20 characters.

Attempting to retrieve data for username: '' OR 1=1 --

```

**Explanation :** The combination of **simulated parameterized queries**, rigorous **input validation**, and comprehensive **exception handling** fundamentally transforms the script from a vulnerable piece of code to a secure and reliable component. The system is now resilient against SQL injection attacks, protected from various forms of malicious input, and capable of handling unexpected errors gracefully, thereby significantly enhancing both its security posture and operational stability.

## 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 an AI-assisted code review of the following Python script. Generate a structured review report that evaluates code readability, naming conventions, formatting and style consistency, error handling, documentation quality, and maintainability. Identify quality issues, risk areas, and code smells, and provide improvement suggestions. Optionally, include a refactored version of the code. The goal is to demonstrate AI as a code reviewer, not just a code generator

Code :

```

▶ import json
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(message)s')

# Define a constant for the calculation threshold
VALUE_THRESHOLD = 10

def process_and_filter_records(input_records):
    """
    Processes a list of records, performs a calculation on the 'val' field,
    and filters records where the calculated value exceeds a predefined threshold.

    Each record in the input list is expected to be a dictionary with 'id' and 'val' keys.
    Non-numeric 'val' entries are skipped, and records without a 'val' key are ignored.

    Args:
        input_records (list): A list of dictionaries, where each dictionary
                             represents a record and may contain 'id' and 'val' keys.

    Returns:
        list: A list of dictionaries, each containing 'id' (as 'k') and the
              calculated 'value' (as 'v') for records that met the criteria.
    """
    processed_results = []
    for record in input_records:
        record_id = record.get('id', 'N/A') # Safely get 'id' for logging

```

```
processed_results = []
for record in input_records:
    record_id = record.get('id', 'N/A') # Safely get 'id' for logging

    if 'val' in record:
        try:
            # Perform the calculation: double the value and add 5
            calculated_value = record['val'] * 2 + 5

            # Filter based on the calculated value exceeding the threshold
            if calculated_value > VALUE_THRESHOLD:
                processed_results.append({'k': record_id, 'v': calculated_value})
        except TypeError:
            logging.warning(f"Skipping non-numeric 'val' for id: {record_id}")
        except Exception as e:
            logging.error(f"An unexpected error occurred processing record id {record_id}: {e}")
    # Records without 'val' key are implicitly ignored as per original logic
return processed_results

# --- Example Usage (similar to original) ---

# Some sample data
DATA_SET = [
    {'id': 1, 'val': 3},
    {'id': 2, 'val': 7},
    {'id': 3, 'val': 'bad_value'},
    {'id': 4, 'val': 0},
    {'id': 5, 'val': 10}

# Execute and print result
logging.info("\n--- Processing DATA_SET ---")
results = process_and_filter_records(DATA_SET)
print(json.dumps(results, indent=2))

# Test with empty data
logging.info("\n--- Processing empty data ---")
empty_data_results = process_and_filter_records([])
print(f"\nEmpty data result: {empty_data_results}")

# Test with missing 'val' key
logging.info("\n--- Processing data with missing 'val' key ---")
missing_val_data = [
    {'id': 6, 'val': 1},
    {'id': 7, 'other_key': 5}
]
missing_val_results = process_and_filter_records(missing_val_data)
print(f"\nMissing 'val' data result: {missing_val_results}")

...
WARNING:root:Skipping non-numeric 'val' for id: 3
[
    {
        "k": 1,
        "v": 11
    },
    {
        "k": 2,
        "v": 19
    }
]
```

```
        v : 11
    },
    {
...
    "k": 2,
    "v": 19
},
{
    "k": 5,
    "v": 25
}
]

Empty data result: []

Missing 'val' data result: []
```

Explanation : In the refactored code example, using `logging.warning` and `logging.error` meant that instead of just printing messages to `stdout` (which might be ignored or lost), these messages would be tagged with their severity, could be directed to a log file, and would stand out for anyone reviewing the application's behavior. This provides a much clearer audit trail of what happened during execution, especially when unexpected data ('bad\_value') or other exceptions occur, making the application more resilient and easier to maintain.