**Features to distinguish:**

Function Name: The name of the function, which is essential for identifying and calling the function.

Function Parameters: The input arguments or parameters the function accepts. This includes the parameter names and their data types.

Function Return Type: The data type that the function returns (e.g., int, str, list, etc.).

Function Docstring: A docstring that provides a description of what the function does, its input parameters, and any additional information about its usage. Proper documentation is essential for code understanding and maintainability.

Function Body: The actual code that makes up the function, including all statements, loops, conditionals, and expressions that define its behavior.

Function Annotations: Optional type annotations that specify the expected data types of parameters and return values. These can be useful for static analysis or automated documentation generation.

Local Variables: Variables defined within the function scope, which can affect the function's behavior.

Function Calls: Any other functions that the function calls within its body. This is important for understanding dependencies and program flow.

Control Flow: The control flow structure of the function, including conditionals (if statements), loops (for and while loops), and other control flow constructs.

Error Handling: Any exception handling mechanisms used in the function, such as try-except blocks to handle errors gracefully.

Function Dependencies: External libraries or modules imported within the function, which are necessary for its execution.

Function Complexity: Metrics like cyclomatic complexity, which quantify the complexity of the function's control flow. High complexity might indicate that the function is hard to understand or maintain.

Function Calls Inside Loops: Identification of any function calls made inside loops, which can impact performance.

Function Comments and TODOs: Any inline comments or TODO markers within the function code.

-NLP basics started

**Python Unit Testing frameworks:**

1. **unittest** is a testing framework in Python that provides a set of tools for writing and running test cases. It is often referred to as the "standard library" for testing in Python because it is part of Python's standard library.

The key features and concepts of the unittest framework include:

Test Cases: Organize tests as classes inheriting from unittest.TestCase, each containing test methods.

Test Methods: Functions within test case classes, named with "test," make assertions about code behavior.

Test Discovery: unittest automatically discovers and runs tests in your codebase.

Assertions: Use assertion methods (e.g., assertEqual, assertTrue, assertRaises) to verify expected outcomes.

Test Fixtures: Support for test fixtures to set up and tear down test environments using methods like setUp and tearDown.

Test Discovery with Test Loaders: Test loaders locate and load test cases and methods for execution.

Test Suites: Group multiple test cases into test suites with unittest.TestSuite.

Test Runners: Built-in test runners (e.g., TextTestRunner, HTMLTestRunner) execute tests and provide results.

1. **pytest:**

pytest is a popular and user-friendly testing framework that extends Python's built-in unittest. It offers concise and expressive test cases, advanced fixtures, and detailed test reports. Many Python developers prefer pytest for its simplicity and extensibility.

1. **nose:**

nose is an older but still widely used testing framework. It provides test discovery, test case execution, and test report generation. It's known for its test discovery capabilities and support for test attributes.

1. **doctest:**

doctest is a testing framework that extracts test cases from docstrings within Python modules. It's a lightweight approach to test documentation and code examples. Test cases are embedded directly within module documentation.

1. **Hypothesis:**

Hypothesis is a property-based testing framework. Instead of writing specific test cases, you define properties that the tested code should satisfy, and Hypothesis generates a wide range of input data to validate these properties.

1. **nose2:**

nose2 is a successor to the original nose testing framework, providing test discovery, test execution, and an extensible plugin system. It's designed to improve on nose and address some of its limitations.

1. **Robot Framework:**

Robot Framework is an external library for keyword-driven test automation. It allows you to define tests using a high-level, tabular syntax and is often used for acceptance testing and robotic process automation (RPA) scenarios.

1. **Behave:**

Behave is a behavior-driven development (BDD) framework that uses the Gherkin language to define tests. It allows you to write tests in a natural language style, making it more accessible to non-technical stakeholders.

1. **Selenium:**

Selenium is primarily used for web application testing and automation. It allows you to script interactions with web pages and perform various types of tests, including unit tests for web applications.

1. **Tox:**

Tox is a tool for running test environments and test suites on different versions of Python and different configurations. It's useful for ensuring cross-compatibility and validating your code on multiple Python environments.

**Difference between Unit test cases and System test cases:**

**Scope:**

**Unit Test Cases:**

Focus on testing individual components, such as functions, methods, or classes in isolation.

Isolate the specific unit of code being tested from the rest of the system.

**Normal (System) Test Cases:**

Test the entire system or a significant part of it.

Evaluate how different components, subsystems, or the entire system work together in real-world scenarios.

**Purpose:**

**Unit Test Cases:**

Aim to verify the correctness of a small, specific piece of code.

Help ensure that individual units of code function as intended.

Facilitate early bug detection and regression testing during development.

**Normal (System) Test Cases:**

Verify that the entire system or a substantial part of it meets the specified requirements.

Evaluate the system's overall functionality, performance, and user experience.

Detects issues related to integration, data flow, and system behavior.

**Dependency:**

**Unit Test Cases:**

Typically run in isolation from other units or components.

Mock or stub external dependencies to isolate the unit being tested.

**Normal (System) Test Cases:**

Rely on the full system and its dependencies, including external resources, databases, and user interfaces.

**Granularity:**

**Unit Test Cases:**

Have fine granularity, focusing on small code units like functions and methods.

Test individual code paths within a unit.

**Normal (System) Test Cases:**

Have coarse granularity, examining the system as a whole or specific subsystems.

Cover higher-level scenarios and interactions between components.

**Frequency:**

**Unit Test Cases:**

Executed frequently during development and integrated into the development workflow (e.g., Continuous Integration).

Provide rapid feedback to developers.

**Normal (System) Test Cases:**

Usually executed less frequently, often as part of a formal testing phase.

May include various types of tests, such as functional, performance, security, and acceptance tests.

**Development Phase:**

**Unit Test Cases:**

Primarily created and maintained by developers during the coding phase.

Help identify and resolve issues early in the development process.

**Normal (System) Test Cases:**

Typically created and maintained by dedicated QA or testing teams during the testing phase.

Evaluate the overall system's compliance with requirements and user expectations.

Some of the datasets searched:

<https://huggingface.co/datasets/vikp/python_functions_filtered>