Advanced Python Calculator for Software Engineering Graduate Course(Midterm)

Project Description:
This is a Python-based modular application for arithmetic calculation. It also supports a dynamic plugin system feature. It includes the App class responsible for loading plugins and executing commands through a CommandHandler mechanism. The application contains different arithmetic operations like add, subtract, divide and multiply through its plugin system. The plugin folder contains all these arithmetic operations as commands and is loaded dynamically and registered using the CommandHandler. It also provides a calculation history management feature, allowing users to load, save, clear, and delete historical calculation records via a singleton instance of a CalculationHistory class. The calculation record is also saved in a .csv file using panda library. Additionally, the project employs a LoggingUtility class for comprehensive logging across the application. The project's structure enables easy scalability, allowing for adding more complex operations or functionalities by adding new plugins.

./app/__init__.py:
This file contains the App class which is the central component of an application that handles command execution based on user input, leveraging a plugin system for command extension.

- *Dynamic Plugin Loading*: The app dynamically loads plugins from specified paths (‘load_plugins’ method), allowing for the extension of the app's functionality without modifying its core code. Plugins are organized into categories such as calculations and history, located within the ‘app.plugins’ package.
- *Command Registration:* For each discovered plugin, the app attempts to import it and register its command instance with the command handler (load_plugin_commands). This process enables the app to recognize and execute a wide range of commands, determined by the available plugins.
- *REPL Loop:* The ‘start’ method initiates a Read-Eval-Print Loop (REPL), continuously accepting user input. It displays available commands, processes input commands, and handles execution or exits based on the user's input.
- *EAFP Principle:* In several places, the code follows the "Easier to Ask for Forgiveness than Permission" (EAFP) coding style, particularly when loading plugins and executing commands. It attempts operations directly and handles exceptions for unexpected cases (e.g., missing plugins or unknown commands), rather than checking preconditions upfront.

./app/commands/__init__.py:
This code defines a command handling system, consisting of an abstract Command class and a CommandHandler class. It's designed to manage and execute commands within an application, particularly useful in CLI (Command Line Interface) applications or systems with modular command structures.

- The ‘Command’ class defines a contract for creating commands by enforcing the implementation of an ‘execute’ method in subclasses.
- The ‘CommandHandler’ class manages a collection of commands, allowing them to be registered and later executed by name. It handles errors gracefully, logging them instead of causing the application to crash.

- This design follows the command pattern, allowing for a high degree of modularity and flexibility in extending the application with new commands. It separates the command execution logic from the command definition, facilitating easier maintenance and scalability.

./app/commands/base_command.py:
The 'BaseCommand' class acts as a common base for other command classes that are part of the application's command handling system. By providing a 'CalculationHistory' instance to each command, it encapsulates the functionality related to managing calculation history, ensuring that all commands can interact with this history without having to instantiate CalculationHistory themselves.

./app/calculation_history.py:
'CalculationHistory' class is designed to manage a history of calculations within an application.

- *Singleton*: The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. In this context, it ensures there's only one 'CalculationHistory' instance throughout the application, which centralizes the management of calculation history. The pattern is implemented in the ' __new__' method. Before creating a new instance, it checks whether an instance already exists. If not, it creates a new instance and initializes it. Otherwise, it returns the existing instance.
- *Initialization and Environment variable*: 'initialize' Initializes the instance by loading environment variables (with load_dotenv from the dotenv package), retrieving the path for the history file (with os.getenv), and ensuring the path is absolute (with os.path.abspath). It then attempts to load or initialize the history data. The 'Environment Variable HISTORY_FILE_PATH' variable specifies the file path for storing the calculation history. If not set, it defaults to 'calculation_history.csv'.
- *History Management:* -'load_or_initialize_history' Method checks if the history file exists. If so, it loads the history from the CSV file into a DataFrame. If not, it initializes a new DataFrame with a single column, 'Calculations'.
  -'add_record' Method adds a new record to the history DataFrame and saves the updated DataFrame to the CSV file.
  -'save_history' Method saves the current state of the history DataFrame to the CSV file, ensuring the directory exists first.
  -'load_history' Method loads the calculation history from the CSV file into the DataFrame if the file exists.
  -'clear_history' Method clears all records from the history by resetting the DataFrame to an empty one with the same column structure and saving this change.
  -'delete_history' Method deletes a specific record from the history identified by its index. It checks for the existence of the history file and whether the specified index is valid before attempting deletion.
- *Look Before You Leap (LBYL):* This coding style involves checking for preconditions before executing a code block. In this code, it's applied when accessing the history file and deleting records, checking for the file's existence and the validity of indices.

./app/logging_utility.py:
This Python code defines a class named ‘LoggingUtility’ that provides a *static* utility for configuring and using logging throughout an application. The class contains methods to set up logging and log messages at different severity levels (info, warning, and error).

./app/plugins/calculations/add/__init__.py:
-‘AddCommand’ class is responsible for executing an addition operation on a set of numbers. It inherits from ‘BaseCommand’, leveraging shared functionality for command execution, especially managing the calculation history. The execute method is where the primary logic for addition is implemented.
-*Error Handling (EAFP):* Embraces Python's "Easier to Ask for Forgiveness than Permission" philosophy by trying to operate and handling exceptions if they occur.
-Calls ‘self.history_instance.add_record(operation, result)’ to save the addition operation and its result to the calculation history. This demonstrates how command classes can interact with other parts of the application, such as recording history.

./app/plugins/calculations/subtract/__init__.py:
-The ‘SubtractCommand’ class in this Python code is designed to perform subtraction on a set of numbers provided as arguments. It inherits from ‘BaseCommand’, using the shared functionality for executing commands and managing calculation history.
-*Error Handling (EAFP):* It follows the "Easier to Ask for Forgiveness than Permission" principle. The method attempts to perform the subtraction by converting arguments to floats. If a ValueError is encountered (indicating that at least one argument cannot be converted to a float), it catches the exception and logs an appropriate error message.
-Uses ‘self.history_instance.add_record(operation, result)’ to save the details of the subtraction operation to the application's calculation history.

./app/plugins/calculations/multiply/__init__.py:
-The ‘MultiplyCommand’ class in this Python code is designed to perform multiplication on a set of numbers provided as arguments. It inherits from ‘BaseCommand’, using the shared functionality for executing commands and managing calculation history.
-*Error Handling (EAFP):* It follows the "Easier to Ask for Forgiveness than Permission" principle. The method attempts to perform the subtraction by converting arguments to floats. If a ValueError is encountered (indicating that at least one argument cannot be converted to a float), it catches the exception and logs an appropriate error message.
-Uses ‘self.history_instance.add_record(operation, result)’ to save the details of the multiplication operation to the application's calculation history.

./app/plugins/calculations/divide/__init__.py:
-The ‘DivideCommand’ class is designed to perform division between two numbers. It inherits from ‘BaseCommand’, allowing it to use shared functionalities, notably the history_instance for recording calculations.
-Adopts the "Easier to Ask for Forgiveness than Permission" principle, using a try-except block to handle errors(*ValueError*: Catches and logs an error if any argument is not a number. And *ZeroDivisionError*: Catches and logs an error if an attempt is made to divide by zero.) rather than checking for conditions upfront.
-Records the operation in the history using ‘self.history_instance.add_record(operation, result)’.

./app/plugins/history/load/__init__.py:
-The ‘LoadCommand’ implements the functionality to load and display the calculation history stored in a CSV file.
-Calls ‘self.history_instance.load_history()’, which attempts to load the calculation history from a CSV file. The method returns a boolean indicating success (True) or failure (False).

./app/plugins/history/clear/__init__.py:
-The ‘ClearCommand’ provides the functionality to clear the calculation history.
-If no arguments are passed, the method proceeds to clear the calculation history by calling ‘self.history_instance.clear_history()’.
-Through the use of LBYL, it prevents incorrect command usage and informs the user accordingly.

./app/plugins/history/delete/__init__.py:
-The ‘DeleteCommand’ class demonstrates careful user input handling through a combination of LBYL and EAFP programming styles. This class encapsulates the functionality to delete a specific entry from the calculation history, based on an index provided by the user.
-It attempts to delete the entry at the provided index (adjusted to be 0-based) by calling ‘self.history_instance.delete_history(index - 1)’.

./tests/conftest.py
-It defines a fixture named ‘app_instance’ that sets up an environment for testing the App class.
-By using the ‘app_instance’ fixture, each test function that requires an ‘App’ object starts with a fresh instance, ensuring that tests are isolated from each other and more reproducible since they do not share a state.

./tests/test_app.py

-test_app_start_exit_command

- Tests if the application's REPL (Read-Eval-Print Loop) exits correctly when the 'exit' command is entered by the user.
- monkeypatch.setattr('builtins.input', lambda _: 'exit'): This line replaces the built-in input function with a mock that always returns 'exit', simulating user input.
- pytest.raises(SystemExit): Checks if the SystemExit exception is raised, which is expected when the application exits.

-test_app_initialization_and_env_loading

- Verifies that the application correctly loads environment variables during initialization.
- Asserts that calling get_environment_variable('ENVIRONMENT') on an app_instance fixture returns a non-None value, indicating that the environment variable is loaded.

-test_execute_known_command

- Tests the application's ability to execute a known command.
- patch.object(app_instance.command_handler, 'execute_command'): Temporarily replaces the execute_command method of the app's command handler with a mock for the duration of the test.
- mock_execute.assert_called_with('add', '2', '3'): Asserts that the mock execute_command method was called with the expected arguments.

test_handle_unknown_command

- Ensures the application logs an error when an unknown command is executed.
- patch('app.logging_utility.LoggingUtility.error'): Replaces the LoggingUtility.error method with a mock.
- mock_log_error.assert_called(): Checks that the mock error method was called, indicating that an error log was generated for the unknown command.

test_repl_exit

- Tests the application's REPL loop exits correctly after executing an unknown command and then the 'exit' command.
- @patch('builtins.input', side_effect=['unknown_command', 'exit']): A decorator that replaces the input function with a mock that first returns 'unknown_command' and then 'exit', simulating user inputs.
- assert mock_input.call_count == 2: Verifies that the mock input function was called twice, matching the number of inputs provided in side_effect.

./tests/test_calculation_history.py

- *'mock_env' Fixture:* Creates a temporary CSV file path using pytest's tmp_path fixture, which provides a temporary directory unique to the test function. Uses 'monkeypatch.setenv' to temporarily set the 'HISTORY_FILE_PATH' environment variable to the path of this temporary CSV file for the duration of the test.
- *'calculation_history_instance' Fixture:* Resets the singleton instance of 'CalculationHistory' to ensure each test starts with a fresh instance. Returns a new instance of 'CalculationHistory'.
- *test_singleton_pattern:* Verifies that 'CalculationHistory' follows the singleton pattern by asserting that two instances are actually the same object.
- *test_load_history_exists:* Checks that the 'load_history' method correctly loads data from an existing CSV file and updates the internal DataFrame.

- *test_load_history_not_exists:* Ensures that load_history correctly handles cases where the CSV file does not exist.
- *test_add_record:* Validates that add_record correctly adds a new calculation record to the internal DataFrame and, implicitly, to the CSV file upon saving.
- *test_clear_history:* Tests that clear_history effectively clears all records from the internal DataFrame.
- *test_delete_history_valid_index:* Confirms that delete_history successfully deletes a record at a valid index.
- *test_delete_history_invalid_index:* Ensures that delete_history handles invalid indices correctly by not modifying the internal DataFrame and returning False.

./tests/test_add_command.py:

- *'mock_history_instance' Fixture*: A pytest fixture that mocks the 'CalculationHistory' class to prevent actual file I/O or other side effects during testing. It specifically mocks the '__init__' method of 'CalculationHistory' to do nothing (preventing actual initialization) and sets 'add_record' to a 'MagicMock', enabling you to monitor interactions with this method. This mocked instance is then returned for use in tests.
- *'add_command' Fixture:* Another pytest fixture that patches 'CalculationHistory' within 'AddCommand' to use the mocked 'CalculationHistory' instance from the 'mock_history_instance' fixture. This ensures that the 'AddCommand' class uses a mock history instance instead of interacting with the actual history management logic. Returns an instance of 'AddCommand' for testing.
- *Test Functions:* 'test_add_command_success' tests the 'execute' method of 'AddCommand' with valid numeric arguments. It mocks 'LoggingUtility.info' to assert that it was called with the expected result of adding the numbers. It also verifies that 'add_record' was called once, indicating that the operation was logged to the calculation history. 'test_add_command_value_error' tests the 'execute' method with at least one non-numeric argument to verify that it handles errors correctly. It mocks 'LoggingUtility.error' to assert that an appropriate error message is logged. It also checks that 'add_record' was not called, reflecting that the operation was not added to the history due to the error.

./tests/test_multiply_command.py and **./tests/test_subtract_command.py** runs the tests similarly to *test_add_command.py.*

./tests/test_divide_command.py:

- *mock_history_instance Fixture* works similarly to *mock_history_instance Fixture* in *test_add_command.py.*
- *divide_command Fixture* works similarly to *'add_command' Fixture.*
- *test_divide_command_success*: Tests successful division, mocking LoggingUtility.info to verify the correct logging of the division result. Also checks that the operation was recorded in history.
- *test_divide_command_invalid_input*: Tests the command's response to non-numeric input by mocking 'LoggingUtility.error' and verifying the appropriate error message is logged. It confirms no record is added to the history for invalid inputs.

- *test_divide_command_division_by_zero*: Specifically tests division by zero, ensuring that the appropriate error message is logged via 'LoggingUtility.error' and that no erroneous record is added to the history.
- *test_divide_command_incorrect_argument_count*: Tests the handling of incorrect numbers of arguments by checking for a warning message when not exactly two arguments are provided. This scenario also should not result in a history record.

./tests/test_load_command.py:

- *mock_history_instance Fixture* works similarly to *mock_history_instance Fixture* in *test_add_command.py.*
- *load_command Fixture* works similarly to *'add_command' Fixture.*
- *test_load_command_with_data*: Tests the behaviour of *'LoadCommand'* when the history contains data. It sets up a mock DataFrame with calculation history, simulates a successful history load (*load_history.return_value = True*), and verifies that *LoggingUtility.info* is called with the correct string representation of the DataFrame.
- *test_load_command_empty*: Tests the behaviour *'LoadCommand'* when the history is empty but the load operation is successful. It ensures that a warning message is logged stating "No calculations in history."
- *test_load_command_no_csv*: Tests the behaviour LoadCommand' when the CSV file is not present or cannot be loaded (*load_history.return_value = False*). It checks that the appropriate warning message "Unable to load history. No CSV file present." is logged.

./tests/test_clear_command.py:

- *mock_history_instance Fixture* works similarly to *mock_history_instance Fixture* in *test_add_command.py.*
- *clear_command Fixture* works similarly to *'add_command' Fixture.*
- *test_clear_command_success*: Tests that the *'ClearCommand'* successfully clears the calculation history. It mocks the '*LoggingUtility.info*' method to verify that the correct success message is logged. The test ensures that the 'clear_history' method of the mock *'CalculationHistory'* is called exactly once.
- *test_clear_command_with_arguments*: Tests that the *'ClearCommand'* handles unexpected arguments correctly by not proceeding with clearing the history and instead logging a warning. It mocks *'LoggingUtility.warning'* to check that the correct warning message is logged when an argument is passed to the 'execute' method. The test also verifies that the 'clear_history' method is not called in this scenario.

./tests/test_delete_command.py:

- *mock_history_instance Fixture* works similarly to *mock_history_instance Fixture* in *test_add_command.py.*
- *delete_command Fixture* works similarly to *'add_command' Fixture.*
- *test_delete_command_no_index_provided*: Tests that the *'DeleteCommand'* class handles the scenario where no index is provided by the user. It's expected to issue a warning, indicating that an index must be specified.

- *test_delete_command_multiple_indices_provided*: Tests that the *'DeleteCommand'* responds correctly when multiple indices are provided, which is not supported. The expected behavior is to issue a warning indicating that only one index can be declared.
- *test_delete_command_invalid_index*: Tests the behavior when a non-integer index is provided. Since the index must be an integer, the command should log an error message indicating the invalid index.
- *test_delete_command_success*: Tests the successful deletion of a record by providing a valid index. It checks that the 'delete_history' method of *'CalculationHistory'* is called with the correct, 0-based adjusted index and that a success message is logged.
- '*mocker.patch.object*' and *'patch'* are used to replace certain parts of the code with mock objects during testing. This approach allows the tests to control and verify the interactions with these parts without relying on their real implementations.