Vilnius Gediminas Technical University

Project Report

Building a Calculator Application

Marc-Henry de Ponton d'Amécourt

supervised by

professors Liutauras Medžiūnas and Tomasz Szturo

April, 2024

Contents

1 Introduction

The application is a user-friendly login interface coupled with a calculator program developed using Python's tkinter library. To run the program, execute the Python script file named "main.py" using a Python interpreter, ensuring all necessary dependencies, such as the tkinter library, are installed. Once launched, users are prompted to log in with their username and password or create a new account if they are new users. After logging in, users are directed to the calculator interface, where they can input mathematical expressions and perform calculations using the provided buttons. The calculator interface consists of several buttons, each serving a specific function. The "Calculate" button evaluates the mathematical expression entered by the user and displays the result. Additionally, there are buttons for saving the calculation data, appending data to an existing file, displaying past calculation results, and logging out of the application. These buttons enhance the usability and functionality of the calculator program, providing users with convenient options for managing their calculations and interacting with the application. Moreover there is also a file named "test.py" which test the principal functions of the Calculator class.

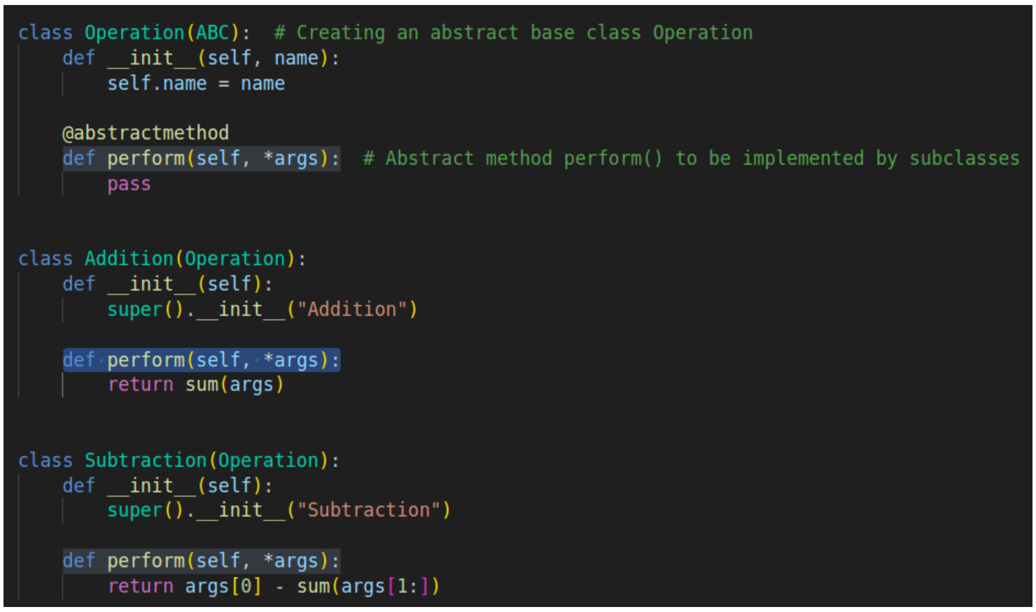
2 Body/Analysis

2.1 4 object-oriented programming pillars

In this project, I have applied the four pillars of object-oriented programming: encapsulation, inheritance, polymorphism, and abstraction.

Encapsulation is evident in the way I encapsulate the calculator's arithmetic operations within separate classes, ensuring that each operation is self-contained and encapsulated from the rest of the program.

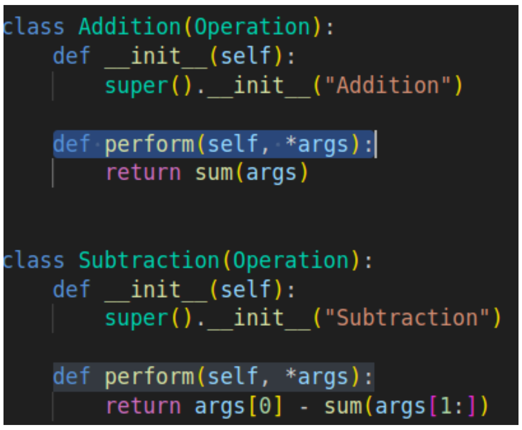
Inheritance is utilised through the creation of subclasses for each arithmetic operation, inheriting common functionality from the abstract base class.



```python
class Operation(ABC):  # Creating an abstract base class Operation
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def perform(self, *args):  # Abstract method perform() to be implemented by subclasses
        pass


class Addition(Operation):
    def __init__(self):
        super().__init__("Addition")

    def perform(self, *args):
        return sum(args)


class Subtraction(Operation):
    def __init__(self):
        super().__init__("Subtraction")

    def perform(self, *args):
        return args[0] - sum(args[1:])
```

Polymorphism is demonstrated by the ability of different arithmetic operation objects to be treated interchangeably, allowing for flexibility and extensibility in the calculator application.

We can see that all the operations class have the same method: `def perform(self, *args):` but it acts differently for each one.

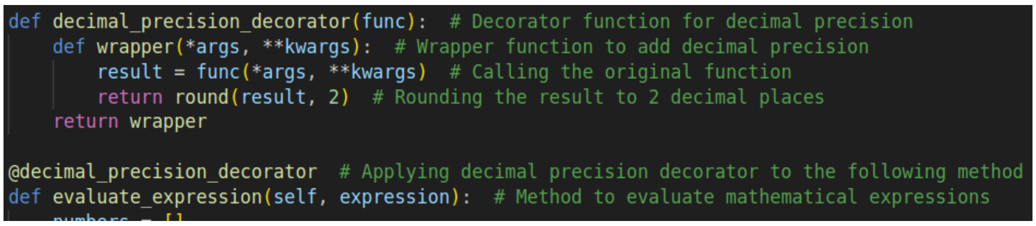


```python
class Addition(Operation):
    def __init__(self):
        super().__init__("Addition")

    def perform(self, *args):
        return sum(args)


class Subtraction(Operation):
    def __init__(self):
        super().__init__("Subtraction")

    def perform(self, *args):
        return args[0] - sum(args[1:])
```

Lastly, abstraction is employed to hide the complex implementation details of each arithmetic operation, providing users with a simplified interface for performing calculations. Overall, these object-oriented programming principles contribute to the clarity, modularity, and extensibility of the codebase.

2.2 Decorator

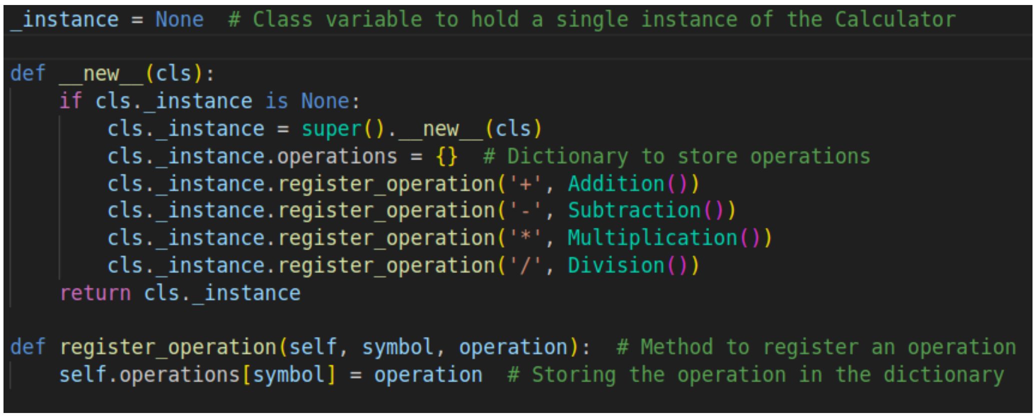
In this project, I have used a decorator to add functionality to the calculator application, specifically for rounding the result to two decimal places. The `decimal_precision_decorator` function is defined as a decorator, which takes another function as input and returns a modified version of that function. This decorator is applied to the `evaluate_expression` method of the `Calculator` class. Whenever `evaluate_expression` is called, the result is passed through the `decimal_precision_decorator`, which rounds it to two decimal places before returning it. This approach allows for the addition of new functionality, such as rounding, without modifying the original code, thereby enhancing the maintainability and extensibility of the application. Unlike other patterns, such as strategy or command patterns, decorators excel in dynamically extending object functionality, making them ideal for this application.



```python
def decimal_precision_decorator(func):  # Decorator function for decimal precision
    def wrapper(*args, **kwargs):  # Wrapper function to add decimal precision
        result = func(*args, **kwargs)  # Calling the original function
        return round(result, 2)  # Rounding the result to 2 decimal places
    return wrapper

@decimal_precision_decorator  # Applying decimal precision decorator to the following method
def evaluate_expression(self, expression):  # Method to evaluate mathematical expressions
    numbers = []
```

2.3 Singleton

I have also used a singleton design pattern to ensure that only one instance of the `Calculator` class is created throughout the application's lifecycle. The `Calculator` class defines a private class attribute `_instance`, initialised as `None`. The `__new__` method of the class is overridden to control the creation of new instances. If `_instance` is `None`, indicating that no instance has been created yet, a new instance is created and assigned to `_instance`. Subsequent calls to `__new__` return the existing instance. This ensures that all components of the application interact with the same instance of the `Calculator` class, providing a centralised and consistent behaviour. Compared to alternative patterns like factory or builder patterns, the singleton pattern excels in enforcing a single instance constraint, making it the most suitable choice for the calculator application.



```python
_instance = None  # Class variable to hold a single instance of the Calculator

def __new__(cls):
    if cls._instance is None:
        cls._instance = super().__new__(cls)
        cls._instance.operations = {}  # Dictionary to store operations
        cls._instance.register_operation('+', Addition())
        cls._instance.register_operation('-', Subtraction())
        cls._instance.register_operation('*', Multiplication())
        cls._instance.register_operation('/', Division())
    return cls._instance

def register_operation(self, symbol, operation):  # Method to register an operation
    self.operations[symbol] = operation  # Storing the operation in the dictionary
```

2.4 Data storage

In this project, user authentication details are stored in JSON format for efficient management and retrieval. When users create an account or log in, their credentials, including usernames and hashed passwords, are stored securely in a JSON file. This JSON file serves as a database for user authentication, allowing the application to validate user credentials during the login process.

On the other hand, calculation history is recorded in a text file to maintain a log of users' past calculations. Each calculation entry is appended to the text file, providing users with a chronological record of their arithmetic operations. By storing calculation history in a text file, the application ensures simplicity and ease of access to past calculations for users.

3 Results

- Successfully implemented a user-friendly login interface coupled with a calculator application.
- Encountered challenges in handling user authentication and data persistence, which were overcome through rigorous testing and debugging.
- The application demonstrates effective use of object-oriented programming principles and design patterns, resulting in a modular and extensible codebase.
- Future prospects include enhancing the user interface, adding support for more advanced mathematical functions, and improving error handling mechanisms.

4 Conclusion

In conclusion, this coursework has achieved the development of a robust and user-friendly calculator application. The program successfully meets the defined objectives and functional requirements, providing users with an efficient tool for performing arithmetic calculations securely. The result of this work is a well-designed application that offers essential mathematical functionalities and data persistence features. Moving forward, there are several future prospects for the program, including enhancements to the user interface, addition of advanced mathematical functions, and further refinement of error handling mechanisms. With continued development and refinement, the calculator application has the potential to become a versatile tool for users across various domains.