Marshall Wace Writeup: FramePro Series and DataFrame Implementation

Anaïs Killian[1]

[1]Harvard University.

1 Abstract

In this project, my goal goal was to create a subset of the Pandas Dataframe and create some enhanced modifications. The code I produced focuses on correctness but also on avoiding redundancy.

1.1 Programming Language Used

I decided to use Python to implement this code. Using another language like Java or C would be thus hard to evaluate when compared to the Python DataFrame functionality.

1.2 Outline of Code

I created and used two primary files: datastructures.py and tests.py. datastructures.py contains the two classes called `Series` and `DataFrame`.
This functionality was added to my `Series` objects:

- I raise an exception for any kind of operations between `Series` of different types.
- I raise an exception for any kind of operations between `Series` of different lengths.
- I perform type checks to determine that all values conform to the created `Series` type or are `None`.
- I implemented read access by overiding the square bracket access operator. The key can be an integer, in which that index of that `Series` is returned. The key can also be a or Series of booleans, in which a new Series with all rows and values where the boolean `Series` is True is returned.
- Many element-wise operations on `Series` were implemented, such as $+, -, *, /$.
- Additionally, inequalities such as $>, <, >=, <=, ! =, ==$ were added to compare elements of `Series` to another value or another `Series`.
- A string representation was implemented.

This functionality was added to the `DataFrame` objects:

- I implemented square bracket access, which can receive a key that is a string and return the respective `Series`. The key can also be a boolean `Series`, in which another DataFrame containing only the rows with True values is returned.
- A string representation was implemented.

2 Result Summary

2.1 Optimization and Performance

We begin by explaining the performance of the Series class: The Series `data` instance variable is constructed as a list, and essentially all Series class operations refer to the `data` instance variable and perform standard list access operations.
As for the DataFrame Class, the DataFrame constructor takes in a dictionary, where the keys are strings and the values are `Series` objects. The constructor adds the keys and values to an internal instance dictionary variable called `columns`. It then checks that all of the Series are of the same length.

To make the code more efficient and readable, using Python built-in functions like map() and filter() are great tools. map() especially could be used to apply the operations functions like addition and comparison to the Series rather than building this myself. Additionally, more efficient algorithms like binary search could be used to look for an element in a `Series`, given it is sorted, rather than just linear search.

2.2 Avoiding Redundancy

2.2.1 Verification

It is important to verify that two objects of the `Series` class have the same length when performing operations between them, and if necessary, if they have the same type. To avoid redundant code within all of my Class functions that perform operations on my Series objects, I implemented these two functions that will raise Exceptions in the proper cases:

```python
# check if a series and another series are of the same length
def check_length(self, other):
    if len(other.data) != len(self.data):
      raise ValueError("Series are of different lengths")

# check if a series and another series are of the same type
def check_types(self, other):
    if other.data_type != self.data_type:
      raise ValueError("Series are of different types")
```

This enables the proper exception message to be thrown while avoiding repeating this code within all operation-based functions.
In the case where both checks are needed, there is also the function `check_valid` that combines the two prior functions:

```python
# idea behind this function is that more checking can be added if needed
# avoid redundant code in all of the operation functions
def check_valid(self, other):
    self.check_length(other)
    self.check_types(other)
```

2.2.2 Operations

Rather than using the same code for all of the different operations that are performed on a `series`, I used this helper function to take in a function that acts as the arithmetic operation:

```python
# avoiding redudancy between all operation functions
def element_wise_operation(self, other, operation):
    if isinstance(other, (int, float)):
      return Series([operation(elm, other) for elm in self.data], self.data_type)
    elif isinstance(other, Series):
      # for operations between two series
      # if Series types or lengths are different, throw exception
      self.check_valid(other)
      return Series([operation(elm1, elm2) for elm1, elm2 in zip(self.data, other.data)],
    self.data_type)
    else:
      raise ValueError("Operation is not valid")
```

Additionally, I used this code for equality operations, since this must return a Boolean series:

```python
def equality_op(self, other, operation):
    if isinstance(other, (int, float)):
      return Series([operation(elm, other) for elm in self.data], bool)
    elif isinstance(other, Series):
      self.check_valid(other)
      return Series([operation(d, o) for d, o in zip(self.data, other.data)], bool)
```

These two helper functions prevent lots of redundant code throughout the operation-based functions.

2.3 Runtime Analysis

The runtime of a sequence of function calls is primarily dependent on two factors: the validation checks (if any), and the list accessing (if any).
For the `series` constructor, we iterate through the list, ensuring that all of the elements are of the indicated type. This is $O(n)$ where n is the numbers of elements. In general, $O(n)$ is the standard runtime for a function within the `datastructures.py` file that explicitly iterate through the entire list at hand. For functions that access an item in the `series` based on a given index, this is $O(1)$.

2.4 Tests

I implemented 21 tests into `texts.py` in order to check the full functionality of the two classes. In addition to printing out the results executed from all of the tests, I used assert statements when needed to ensure that the value output was as expected.

3 Conclusion

In conclusion, FramePro demonstrates how a programmer can implement their own functionality of a library, and make adjustments as wanted. Although Pandas is an extensive library with many different tools, some sequence of functions can result in unwanted behavior. In addition, the strategy of minimizing redundancy through writing utility functions not only simplifies the code but also facilitates easier maintenance and scalability.