2891. Method Chaining

Easy

Leetcode Link

Problem Description

In this problem, we are given a `DataFrame` named `animals` that contains information about different animals, including their `name`, `species`, `age`, and `weight`. Our task is to write a Python function that uses Pandas to list the names of animals that have a weight strictly greater than 100 kilograms. After finding the relevant animals, we need to sort this list by the animals' weight in descending order so the heaviest animals appear first.

The DataFrame is structured with columns for each attribute of the animals, and each row corresponds to a distinct animal. We are interested in filtering the rows based on a particular column (`weight`) and then manipulating the DataFrame to return a specific subset of its data (the `name` column).

In the context of the problem, we are also asked to leverage method chaining in Pandas which allows us to execute multiple operations in a compact and readable one-liner. This is efficient and elegant, minimizing the need for creating temporary variables and making the code easier to understand at a glance.

Intuition

The intuition behind the solution involves two main steps, which we can implement in Pandas through method chaining:

1. **Filtering**: First, we need to filter the DataFrame to include only those rows where the animals' weight is more than 100 kilograms. In Pandas, this is achieved with a boolean indexing operation, where we compare the `weight` column against the value `100`. The comparison generates a boolean Series that we use to filter out rows that don't meet the condition.

2. **Sorting and Selecting Columns**: After filtering the rows, we should sort them by the `weight` column in descending order to meet the requirement of listing heavier animals first. The `sort_values` function in Pandas can be used for this purpose by specifying the `ascending=False` parameter. Once sorted, we need to select the `name` column as this is what we want to return. By indexing the DataFrame with a list of column names (`['name']`), we can select the required column(s).

The final solution combines filtering, sorting, and column selection in a single expression using method chaining. Each operation returns a DataFrame or Series that is immediately used as the input for the next operation in the chain, resulting in concise and efficient code.

Solution Approach

The solution is implemented using a Python function that expects a Pandas DataFrame as an input and returns a DataFrame as an output. Here is a step-by-step breakdown of the one-liner solution within the function:

1. **Filtering with Boolean Indexing**:

   In `animals[animals['weight'] > 100]`, we perform a boolean indexing operation. This creates a boolean Series by comparing each value in the `weight` column to the number `100`. This Series is then used to filter the DataFrame, keeping only the rows where the condition (greater than 100) is `True`.

2. **Sorting Values**:

   The `.sort_values('weight', ascending=False)` method is chained after the boolean indexing. This call sorts the filtered DataFrame by the `weight` column in descending order (`ascending=False`). The resulting DataFrame maintains only the filtered rows, now sorted so that the heaviest animals are at the top.

3. **Selecting Columns**:

   The last part of the chain `[['name']]` selects only the `name` column of the sorted DataFrame. This indexing operation constrains the output to contain only the names of the heavy animals, as requested.

By following these steps, the function returns the names of animals that weigh more than 100 kilograms, sorted by their weight in descending order. The entire process is a demonstration of method chaining in Pandas and showcases how expressive and efficient this approach can be for data manipulation tasks.

The algorithm's complexity essentially depends on the filtering and sorting operations. The filtering runs in $O(n)$ time, where n is the number of rows in the DataFrame, as it involves checking each weight once. Sorting can be expected to run on average in $O(n \log n)$ time. Consequently, the overall complexity of the operation would be dominated by the sorting step, resulting in an average time complexity of $O(n \log n)$.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have the following `DataFrame` named `animals`:

name	species	age	weight
Daisy	cow	5	200
Bubbles	fish	1	22
Boomer	kangaroo	3	85
Zeus	elephant	10	500
Fluffy	rabbit	2	4

We want to extract the names of animals weighing more than 100 kilograms, sorted by their weight in descending order.

Step-by-Step Walkthrough

1. **Filtering with Boolean Indexing**:

   We apply the boolean indexing operation `animals['weight'] > 100` to create the following boolean Series:

   ```
   0  Daisy    True
   1  Bubbles  False
   2  Boomer   False
   3  Zeus     True
   4  Fluffy   False
   ```

   Using this Series to filter the DataFrame, we get:

name	species	age	weight
Daisy	cow	5	200
Zeus	elephant	10	500

2. **Sorting Values**:

   We then sort the filtered results by the `weight` column in descending order:

name	species	age	weight
Zeus	elephant	10	500
Daisy	cow	5	200

3. **Selecting Columns**:

   Finally, we select just the `name` column:

name
Zeus
Daisy

The Code

```
def heavy_animals(df):
    return df.loc['weight'] > 100].sort_values('weight', ascending=False)[['name']]

# # Now, let's use our `animals` DataFrame as an input to our function
result = heavy_animals(animals)
print(result)
```

Expected Output:

```
   name
2  Zeus
0  Daisy
```

This output matches our criteria, listing the names of the animals that weigh more than 100 kilograms, sorted in descending order by weight. With the above approach, we are able to efficiently filter, sort, and select the necessary data using method chaining in Pandas.

Python Solution

```
import pandas as pd  # Importing the pandas library with the alias 'pd'

# Define a function that finds animals weighing more than 100 units
def find_heavy_animals(animals_df: pd.DataFrame) -> pd.DataFrame:
    """
    Identify and return a DataFrame with the names of animals that weigh more than 100 units.
    The result is sorted by weight in descending order.

    :param animals_df: A pandas DataFrame with columns including 'name' and 'weight'.
    :return: A DataFrame with the names of heavy animals, sorted by weight.
    """
    # Filter the DataFrame to include only animals weighing more than 100 units
    heavy_animals = animals_df[animals_df['weight'] > 100]

    # Sort the filtered DataFrame by weight in descending order and select only the 'name' column
    sorted_heavy_animals = heavy_animals.sort_values('weight', ascending=False)[['name']]

    return sorted_heavy_animals  # Return the sorted DataFrame with animal names
```

Java Solution

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

// Class to represent an animal with a name and weight
class Animal {
    String name;
    int weight;

    public Animal(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    // Getters...
    public String getName() {
        return name;
    }

    public int getWeight() {
        return weight;
    }

    // You might also want to add setters and other utility methods if needed.
}

public class AnimalWeightFinder {

    // Function to find animals weighing more than 100 units
    public static List<String> findHeavyAnimals(List<Animal> animals) {
        // Filter the list for animals only weighing more than 100 units
        List<Animal> heavyAnimals = animals.stream()
                                           .filter(animal -> animal.getWeight() > 100)
                                           .collect(Collectors.toList());

        // Sort the list of heavy animals by weight in descending order
        Collections.sort(heavyAnimals, new Comparator<Animal>() {
            public int compare(Animal a1, Animal a2) {
                return a2.getWeight() - a1.getWeight();
            }
        });

        // Extract just the names of the sorted heavy animals
        List<String> sortedHeavyAnimalNames = new ArrayList<>();
        for (Animal animal : heavyAnimals) {
            sortedHeavyAnimalNames.add(animal.getName());
        }

        // Return the list of sorted heavy animal names
        return sortedHeavyAnimalNames;
    }

    // Main method for demonstration purposes (Optional)
    public static void main(String[] args) {
        // List of animals (simulating a DataFrame)
        List<Animal> animals = new ArrayList<>();
        animals.add(new Animal("Elephant", 1200));
        animals.add(new Animal("Tiger", 150));
        animals.add(new Animal("Rabbit", 5));
        animals.add(new Animal("Bear", 600));

        // Find and print names of heavy animals
        List<String> heavyAnimalNames = findHeavyAnimals(animals);
        System.out.println("Heavy Animals: " + heavyAnimalNames);
    }
}
```

C++ Solution

```
#include <vector>
#include <algorithm>
#include <string>

// Assuming an Animal structure defined like this:
struct Animal {
    std::string name;
    double weight;
};

// Comparator function for sorting Animals by weight in descending order
bool compareByWeightDescending(const Animal &a, const Animal &b) {
    return a.weight > b.weight;
}

// Define a function that finds animals weighing more than 100 units
std::vector<std::string> findHeavyAnimals(const std::vector<Animal> &animals) {
    std::vector<std::string> heavy_animals_names; // Vector to keep names of heavy animals

    // Iterate over the input vector and select animals that weigh more than 100 units
    for (const auto &animal : animals) {
        if (animal.weight > 100) {
            heavy_animals_names.push_back(animal.name);
        }
    }

    // Sort the names of the heavy animals by their weights in descending order.
    // Since we only have the names in the vector, we would need to reference back to the original vector.
    // Therefore, this step might require either keeping weights in the pair with names OR having a map for weights
    // For simplicity, let's assume we have a map for weights:
    std::sort(heavy_animals_names.begin(), heavy_animals_names.end(), [&](const std::string &name1, const std::string &name2) {
        double weight1 = std::find_if(animals.begin(), animals.end(), [&](const Animal &animal) {
            return animal.name == name1;
        })->weight;
        double weight2 = std::find_if(animals.begin(), animals.end(), [&](const Animal &animal) {
            return animal.name == name2;
        })->weight;
        return weight1 > weight2;
    });

    return heavy_animals_names; // Return the vector containing sorted heavy animals names
}
```

Typescript Solution

```
import { DataFrame } from 'pandas-js'; // Importing the DataFrame class from 'pandas-js'

/**
 * Identify and return an array with the names of animals that weigh more than 100 units.
 * The result is sorted by weight in descending order.
 *
 * @param animalsDf A DataFrame with columns including 'name' and 'weight'.
 * @return An array with the names of heavy animals, sorted by weight.
 */
function findHeavyAnimals(animalsDf: DataFrame): string[] {

    // Filter the DataFrame to include only animals weighing more than 100 units
    const heavyAnimals = animalsDf.filter((row, any) => row.get('weight') > 100);

    // Sort the filtered DataFrame by weight in descending order
    const sortedHeavyAnimals = heavyAnimals.sort_values({ by: 'weight', ascending: false });

    // Select only the 'name' column and convert it to an array
    const heavyAnimalNames: string[] = sortedHeavyAnimals.get('name').to_json({ orient: 'records' });

    return heavyAnimalNames; // Return the array with animal names
}

// Note that pandas-js might not have exact one-to-one mapping with the Python pandas library.
// The provided functionality is based on typical usage of a JavaScript DataFrame library.
// It is assumed that the 'pandas-js' library has a similar API to that of Python's pandas.
```

Time and Space Complexity

The time complexity of the `findHeavyAnimals` function involves several steps. First, we filter the `animals` DataFrame, which requires $O(n)$ time, where n is the number of rows in `animals`. Then we sort this filtered DataFrame, which takes $O(m \log m)$ time where m is the number of rows with `weight` greater than 100. Finally, we slice the DataFrame to include only the `name` column, which is an $O(m)$ operation. Therefore, the overall time complexity is $O(n + m \log m)$.

The space complexity of the `findHeavyAnimals` function also involves several components. The filtering operation generates a new DataFrame which can be up to $O(n)$ space if all animals are heavier than 100 units. The sorting operation takes place in-place in pandas by default, so it does not change the space complexity, but if a copy was made during this process, it would require additional $O(m)$ space. Selecting a single column from the DataFrame does not require additional space as it provides a view on the existing DataFrame, not a copy. Hence, the overall space complexity is $O(n)$ if a copy is made during sorting, otherwise it remains $O(n)$ due to the initial filter result.