2881. Create a New Column

Easy

Leetcode Link

Problem Description

A company has a DataFrame `employees` that holds two columns: `name` and `salary`. The `name` column is of object type and contains the names of the employees, while the `salary` column is of integer type and contains each employee's salary. The task is to write a piece of code that adds a new column to the `employees` DataFrame. This new column, named `bonus`, is supposed to contain the doubled values of the `salary` column for each employee. It is essentially a bonus calculation where each employee's bonus is twice their current salary.

Intuition

The intuition behind the solution is to take advantage of the functionality provided by the pandas library in Python to manipulate DataFrame objects. Pandas allows us to perform vectorized operations on columns, which means operations can be applied to each element of a column without the need for explicit iteration over rows.

Given that the objective is to double the salary for each employee, we can simply select the `salary` column of the DataFrame and multiply its values by 2. The resulting Series (a one-dimensional array in pandas) can then be assigned to a new column called `bonus` within the same DataFrame.

This is a straightforward operation, and it involves the following steps:

1. Multiply the `salary` column by 2 using the `*` (multiplication) operator. This operation is inherently element-wise when using pandas Series.
2. Assign the result of this multiplication to a new column in the DataFrame named `bonus`. This is done by setting `employees['bonus']`—which creates the new column—to the result of the multiplication.

The solution is efficient because it does not involve any explicit loops and makes full use of the features of pandas for vectorized operations on DataFrames.

Solution Approach

The solution approach for this problem is quite straightforward, thanks to Python's pandas library. Given the goal is to generate a new column named `bonus` derived from the existing `salary` column, we follow these steps:

1. Select the `salary` column from the `employees` DataFrame. This can be done with `employees['salary']`.
2. Multiply the selected column by 2 to calculate the bonus. In pandas, this operation will automatically apply to each element (i.e., each salary) in the column, resulting in a new Series where each value is double the original.
3. Assign this new Series to a new column in the `employees` DataFrame named `bonus`. This column is created on the fly with the assignment operation: `employees['bonus'] = new_series`.

Here's an explanation of the elements and concepts used in the implementation:

- **DataFrame**: A pandas DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). In this case, `employees` is a DataFrame representing a table of employees with columns for `name` and `salary`.
- **Series**: A Series is a one-dimensional array capable of holding any data type. When we select a single column from a DataFrame (like `employees['salary']`), we're working with a Series.
- **Element-wise Multiplication**: When multiplying the `salary` Series by 2 (`employees['salary'] * 2`), pandas applies the multiplication to each element of the Series, doubling every individual salary. This is an efficient vectorized operation that avoids the need for explicit looping.
- **Column Assignment**: By setting `employees['bonus']` equal to the doubled salaries Series, we're assigning the results of our calculation to a new column in the DataFrame called `bonus`.

In summary, the implementation uses basic pandas operations to create and calculate a new column in an existing DataFrame, demonstrating simple and effective manipulation of tabular data.

Example Walkthrough

Let's say we have the following `employees` DataFrame:

name	salary
Alice	70000
Bob	60000
Charlie	50000

We want to add a `bonus` column where each employee's bonus equals twice their salary. Here's how we apply our solution approach:

1. We select the `salary` column from the `employees` DataFrame using `employees['salary']`.
2. We calculate the bonus by multiplying every salary by 2. For our example, this would be:
   - Alice's bonus: 70000 * 2 = 140000
   - Bob's bonus: 60000 * 2 = 120000
   - Charlie's bonus: 50000 * 2 = 100000

   This step is performed using `employees['salary'] * 2`, resulting in a Series that looks like:

Series Index	*Value (bonus)*
Alice	140000
Bob	120000
Charlie	100000

3. We assign this Series to the new `bonus` column in the `employees` DataFrame. This assignment operation is `employees['bonus'] = employees['salary'] * 2`.

After executing these steps, our `employees` DataFrame will be updated to include the new `bonus` column, resulting in:

name	salary	bonus
Alice	70000	140000
Bob	60000	120000
Charlie	50000	100000

The `bonus` column reflects the doubled salary for each employee, effectively showing the desired calculation. Each step of this process leverages the power of pandas to efficiently handle and compute data in a vectorized manner without the need for explicit looping constructs.

Python Solution

```python
import pandas as pd

def create_bonus_column(employees_df: pd.DataFrame) -> pd.DataFrame:
    # Create a new column 'bonus' in the dataframe
    # The 'bonus' is calculated as double the employee's salary
    employees_df['bonus'] = employees_df['salary'] * 2

    # Return the modified dataframe with the new 'bonus' column
    return employees_df
```

Java Solution

```java
import java.util.List;
import java.util.stream.Collectors;

// Assuming Employee is a predefined class with at least two fields: name and salary.
class Employee {
    private String name;
    private double salary;
    private double bonus;

    // Getter and setter methods for name, salary, and bonus
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public double getBonus() {
        return bonus;
    }

    public void setBonus(double bonus) {
        this.bonus = bonus;
    }

    // Constructor
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
        this.bonus = 0; // bonus initialized to 0
    }
}

public class EmployeeBonusCalculator {

    /**
     * Creates a bonus field for each employee and sets it to double their salary.
     *
     * @param employees List of Employee objects
     * @return The list with updated Employee objects including the bonus
     */
    public List<Employee> createBonusColumn(List<Employee> employees) {
        // Loop through each employee in the list and calculate their bonus
        List<Employee> updatedEmployees = employees.stream().map(employee -> {
            // Calculate the bonus as double the employee's salary
            double bonus = employee.getSalary() * 2;

            // Set the bonus to the employee's record
            employee.setBonus(bonus);
            return employee;
        }).collect(Collectors.toList());

        // Return the list with updated employees
        return updatedEmployees;
    }
}
```

C++ Solution

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <string>

typedef std::map<std::string, std::string> EmployeeRow;
typedef std::vector<EmployeeRow> DataFrame;

DataFrame CreateBonusColumn(DataFrame employees_df) {
    // Iterate through each employee entry in the dataframe
    for (auto& employee : employees_df) {
        // Assume 'salary' is stored as a string, convert it to a double to perform the calculation
        double salary = std::stod(employee["salary"]);

        // Double the salary to determine the bonus
        double bonus = salary * 2;

        // Store the bonus back in the row, converting it to a string
        employee["bonus"] = std::to_string(bonus);
    }

    // Return the modified dataframe with the new 'bonus' column
    return employees_df;
}

int main() {
    // Example usage:
    // Create a sample dataframe with employee salaries
    DataFrame employees_df = {
        {{"name", "Alice"}, {"salary", "50000"}},
        {{"name", "Bob"}, {"salary", "60000"}},
        {{"name", "Charlie"}, {"salary", "75000"}}
    };

    // Add bonus column to the dataframe
    employees_df = CreateBonusColumn(employees_df);

    // Print the result to check the 'bonus' column
    for (const auto& employee : employees_df) {
        std::cout << "Name: " << employee.at("name")
                  << ", Salary: " << employee.at("salary")
                  << ", Bonus: " << employee.at("bonus") << std::endl;
    }

    return 0;
}
```

Typescript Solution

```typescript
// Define an interface to represent the structure of an employee object
interface Employee {
    salary: number;
    // Additional properties can be defined here if they exist
    // for example, name, id, department, etc.
    // ...
    bonus?: number;  // The bonus property is optional because it will be added later
}

// Function to create a bonus property for each employee
function createBonusColumn(employees: Employee[]): Employee[] {
    // Iterate over the array of employee objects
    employees.forEach(employee => {
        // Calculate the bonus as double the employee's salary and assign it to the 'bonus' property
        employee.bonus = employee.salary * 2;
    });

    // Return the modified array of employee objects with the new 'bonus' property added
    return employees;
}

// Example usage:
// const employees: Employee[] = [{ salary: 30000 }, { salary: 40000 }];
// const updatedEmployees = createBonusColumn(employees);
// console.log(updatedEmployees);
```

Time and Space Complexity

The time complexity of the function is $O(n)$, where n is the number of rows in the `employees` DataFrame. This is because the operation `employees['salary'] * 2` is applied to each row to calculate the bonus.

The space complexity of the function is $O(n)$, assuming that the creation of an additional column represents new memory allocation proportional to the number of rows in the DataFrame. If the bonus values are stored as a separate array before being assigned to the DataFrame, this would still entail $O(n)$ additional space.