Untitled4

March 21, 2024

1 Week 7 Python

[1]:
```
import pandas as pd
```

1.1 Grouping, Aggregating and Filtering

Let's import `employee_datasetv2.csv` dataset

[2]:
```
df=pd.read_csv('employee_datasetv2.csv')
```

[3]:
```
df.head(10)
```

[3]:
```
   Employee_ID Department  Years_of_Experience  Full_Time Performance_Score  \
0            1    Finance             1.820000      False              Good
1            2         IT             2.090000      False              Poor
2            3         IT             7.047228      False         Excellent
3            4    Finance             1.237509      False         Excellent
4            5  Marketing             4.160000       True           Average
5            6    Finance             2.475017       True         Excellent
6            7  Marketing            10.420000       True              Good
7            8  Marketing             6.102669       True              Good
8            9  Marketing            10.000000       True              Poor
9           10         HR             7.479808      False           Average

     Salary First_Name Last_Name        Date_of_Birth  Gender  …  \
0   56550.0    Michael     Davis  2000-05-11 05:33:39   Other  …
1   55225.0      Karen     Brown  1984-12-18 14:13:34   Other  …
2   87600.0     Joseph   Johnson  1991-07-29 11:32:58    Male  …
3   53875.0      David    Garcia  1998-01-24 12:21:27   Other  …
4   61400.0      Linda  Martinez  1973-06-05 08:09:32   Other  …
5   70550.0    Michael     Brown  1973-03-07 10:44:36  Female  …
6   82050.0    Charles     Moore  1981-03-26 17:29:10    Male  …
7   84750.0      David     Lopez  1981-03-25 12:42:58  Female  …
8   80500.0      David   Johnson  1973-08-09 18:43:34  Female  …
9  101350.0   Patricia  Martinez  1967-11-27 08:26:50  Female  …

                Location Remote_Work            Hire_Date  \
```

```
0          Glasgow, Scotland        True  2022-05-16 07:30:27
1            London, England       False  2020-05-03 22:30:19
2  Belfast, Northern Ireland        True  2017-02-25 23:43:54
3  Belfast, Northern Ireland        True  2022-12-18 10:59:47
4        Manchester, England        True  2012-04-04 14:02:58
5          Glasgow, Scotland       False  2021-09-22 12:23:53
6          Glasgow, Scotland       False  2011-08-25 03:34:26
7  Belfast, Northern Ireland       False  2018-02-05 19:31:10
8          Glasgow, Scotland       False  2012-09-29 19:05:03
9             Cardiff, Wales        True  2016-09-20 00:26:27

  Last_Performance_Review_Date Project_Count  Bonus  \
0          2022-10-08 07:30:27             5      0
1          2020-08-15 22:30:19             6    500
2          2017-07-17 23:43:54             3   2000
3          2023-04-30 10:59:47             9    500
4          2013-01-12 14:02:58             5    500
5          2022-06-05 12:23:53             8      0
6          2012-01-13 03:34:26             1   1500
7          2018-06-13 19:31:10             5    500
8          2013-03-30 19:05:03             3   1000
9          2017-05-01 00:26:27             1   1500

   Employee_Satisfaction_Score  Department_Budget  Training_Hours_Last_Year  \
0                         2.68              75000                        46
1                         1.99             100000                        15
2                         2.42              50000                         4
3                         4.03             125000                        34
4                         1.06              75000                        11
5                         1.46             150000                        24
6                         1.18              75000                        20
7                         1.16             150000                        35
8                         4.42             100000                        22
9                         3.81             150000                        15

   Number_of_Direct_Reports
0                         6
1                         8
2                         3
3                         3
4                         5
5                         2
6                         5
7                         6
8                         9
9                         9
```

```
[10 rows x 21 columns]
```

[4]:
```python
df.describe()
```

[4]:
```
       Employee_ID  Years_of_Experience         Salary  Project_Count  \
count     50.00000            50.000000      50.000000      50.000000   
mean      25.50000             6.552038   78774.000000       4.900000   
std       14.57738             3.777367   15041.086553       2.697315   
min        1.00000             1.120000   53875.000000       1.000000   
25%       13.25000             2.980262   65975.000000       2.250000   
50%       25.50000             6.184805   80150.000000       5.000000   
75%       37.75000            10.099589   89156.250000       7.000000   
max       50.00000            13.880903  105750.000000       9.000000   

             Bonus  Employee_Satisfaction_Score  Department_Budget  \
count    50.000000                    50.000000          50.000000   
mean   1090.000000                     2.684200       98000.000000   
std     760.571751                     1.188431       38412.370456   
min       0.000000                     1.060000       50000.000000   
25%     500.000000                     1.415000       50000.000000   
50%    1250.000000                     2.710000      100000.000000   
75%    2000.000000                     3.562500      125000.000000   
max    2000.000000                     4.840000      150000.000000   

       Training_Hours_Last_Year  Number_of_Direct_Reports  
count                 50.000000                 50.000000  
mean                  24.840000                  4.620000  
std                   14.945288                  2.834842  
min                    0.000000                  0.000000  
25%                   11.250000                  2.000000  
50%                   23.000000                  5.000000  
75%                   38.000000                  7.000000  
max                   49.000000                  9.000000  
```

1.2 Grouping

Concept: **Grouping** involves organizing data into groups based on some criteria, such as department or education level. Pandas uses the .groupby() method to achieve this, allowing for powerful and flexible data analysis.

Example: Group employees by their department to calculate the average salary within each department.

[5]:
```python
grouped_by_dept=df.groupby('Department')
```

This line of code performs a grouping operation on the DataFrame `df` based on the values in the 'Department' column. Here's a step-by-step explanation:

1. **`df`**: This is the DataFrame that contains the data from the `employee_datasetv2.csv` file

you've read into Python using **pandas**. It's assumed to have multiple columns, one of which is named 'Department'.

2. **.groupby('Department')**: This method is called on the DataFrame **df**. The **groupby** method is used to split the data into groups based on some criteria. In this case, the criteria is the 'Department' column. This means that the DataFrame will be divided into groups where each group contains rows that have the same value in the 'Department' column.

3. **grouped_by_dept**: This is the variable to which the result of the **groupby** operation is assigned. However, it's important to note that **grouped_by_dept** does not hold a simple DataFrame. Instead, it holds a **DataFrameGroupBy** object, which is a collection of groups awaiting further analysis or manipulation.

After this line of code is executed, **grouped_by_dept** can be used to perform operations on each group separately, such as calculating statistics (mean, median, max, min, etc.), applying functions, or aggregating data in various ways specific to each department. This is particularly useful for analyzing differences across departments or summarizing data at the department level.

[6]:
```
avg_salary_by_dept=grouped_by_dept['Salary'].mean()
```

This line of code calculates the average salary within each department from the previously grouped DataFrame (**grouped_by_dept**). Here's how it works:

1. **grouped_by_dept**: This is the **DataFrameGroupBy** object you created by grouping the original DataFrame **df** by the 'Department' column. It represents your dataset partitioned into groups, where each group consists of rows that have the same department name.

2. **['Salary']**: This part of the code selects the 'Salary' column from each group. Because **grouped_by_dept** is a group of data grouped by department, this operation is applied to each department group, effectively isolating the 'Salary' data for further computation within each department.

3. **.mean()**: This method computes the mean (average) of the 'Salary' column for each department group. The operation is performed on the 'Salary' column of each department, calculating the average salary for that department.

4. **avg_salary_by_dept**: The result of the mean calculation for each group is assigned to this variable. The resulting object is a Series where the index is the department names (from the 'Department' column used to group the DataFrame) and the values are the calculated average salaries for those departments.

The final outcome is that **avg_salary_by_dept** holds the average salary for each department, allowing for an easy comparison of how average salaries differ across departments in the dataset.

[7]:
```
avg_salary_by_dept
```

[7]:
```
Department
Finance      76141.666667
HR           78359.090909
IT           80515.625000
Marketing    80796.875000
Name: Salary, dtype: float64
```

[8]:
```
avg_salary_by_dept.reset_index()
```

[8]:
```
   Department         Salary
0     Finance   76141.666667
1          HR   78359.090909
2          IT   80515.625000
3   Marketing   80796.875000
```

The `avg_salary_by_dept.reset_index()` method is used to transform the Series `avg_salary_by_dept` into a DataFrame and reset its index. This operation is particularly useful after performing groupby calculations which often result in the grouping columns becoming the index of the resulting Series or DataFrame. Here's the detailed process:

1. **`avg_salary_by_dept`**: Before calling `reset_index()`, `avg_salary_by_dept` is a pandas Series with departments as its index and their corresponding average salaries as its values. This Series was obtained by grouping the original DataFrame `df` by 'Department', selecting the 'Salary' column, and then calculating the mean salary for each department.

2. **`.reset_index()`**: This method converts the Series into a DataFrame and resets its index. By default, the index of the Series (in this case, the department names) becomes a regular column in the resulting DataFrame, and a new numerical index is introduced, starting from 0 and incrementing by 1 for each row.

3. **Result**: The result of this operation is a DataFrame with two columns:

   - The first column (often named 'index' if no other arguments are provided to `reset_index()`) contains the department names that were previously the index in the Series.
   - The second column contains the average salaries for each department. The name of this column will be the same as the name of the Series, if it had one, or it might default to a generic name like '0' if the Series did not have a name.

This process makes the data structure more flexible for further analysis or for merging with other DataFrames, as it converts the Series back into a DataFrame and provides a standard, numerical index.

1.3 Grouping and Aggregating

Concept: **Aggregation** refers to any data transformation that produces scalar values from arrays. After grouping data, you can perform aggregation operations like calculating sums, means, and minimums or maximums.

Example: Find the total bonuses distributed in each department.

[9]:
```
total_bonus_by_dept= df.groupby('Department')['Bonus'].sum()
```

This line of code calculates the total sum of bonuses within each department from your DataFrame `df`. It's doing several things in a single, concise operation:

1. **`df.groupby('Department')`**: This groups the DataFrame `df` by the 'Department' column. Each group consists of all rows that have the same value in the 'Department' column, effectively segregating the data by department.

2. **`['Bonus']`**: After grouping, this selects the 'Bonus' column from each group. This means that for each department, only the 'Bonus' data is considered for the subsequent operation.

3. **`.sum()`**: This method is applied to the 'Bonus' column of each department group to calculate the sum of bonuses within that department. It aggregates the bonus values by adding them up for each department.

4. **`total_bonus_by_dept`**: The result of the sum operation is assigned to this variable. The resulting object is a pandas Series where the index is the department names (as determined by the 'Department' column used to group the DataFrame) and the values are the total sum of bonuses for those departments.

Thus, `total_bonus_by_dept` holds the total bonus amount for each department, enabling an analysis of how bonuses are distributed across different departments within the organization. This can be useful for understanding departmental rewards or for budgeting and financial planning related to employee compensation.

[10]:
```
total_bonus_by_dept.reset_index()
```

[10]:

	Department	Bonus
0	Finance	14500
1	HR	12500
2	IT	20000
3	Marketing	7500

The `total_bonus_by_dept.reset_index()` method is used to convert the `total_bonus_by_dept` Series into a DataFrame and reset its index. This transformation is especially useful when you have a Series with a non-default index (in this case, department names as the index) and you want to turn it back into a column, making the data structure resemble a traditional table with a default integer index. Here's what happens during this operation:

1. **`total_bonus_by_dept`**: Initially, this is a pandas Series resulting from summing up the 'Bonus' values for each department in the DataFrame **`df`**. The index of this Series is the unique values from the 'Department' column, representing different departments.

2. **`.reset_index()`**: This method performs two main actions:

   - It converts the Series into a DataFrame. The values of the Series (total bonuses) become one column in the new DataFrame.
   - It resets the index of this DataFrame to a default integer index, starting from 0. The original index (department names) is added as a new column in the DataFrame.

3. **Result**: The outcome is a DataFrame with two columns:

   - The first column, often named 'Department' if the original Series had a name or defaulting to 'index' if not specified, contains the names of the departments. These were previously the index of the Series.
   - The second column contains the corresponding total bonus amounts for each department. If the Series had a name (in this case, 'Bonus'), that name would be used for this column. Otherwise, a default name would be applied, typically a numerical label like '0'.

This operation is particularly useful for preparing the data for further analysis, reporting, or visualization, as it presents the information in a more conventional tabular format with a simple numerical index.

1.4 Combining Grouping and Aggregation

Concept: By **combining grouping and aggregation**, you can perform more complex analyses that involve summarizing grouped data using various aggregation functions.

Example: Calculate the average number of training hours last year, grouped by department and full-time/part-time status.

[11]:
```
avg_training_hours=df.
↪groupby(['Department','Full_Time'])['Training_Hours_Last_Year'].mean()
```

This line of code calculates the average training hours in the last year for each department, further segmented by whether the employee is full-time or not. Here's a breakdown of what each part does:

1. **`df.groupby(['Department','Full_Time'])`**: This groups the DataFrame **`df`** by two columns: 'Department' and 'Full_Time'. It creates groups based on the unique combinations of department names and full-time status. This means that for each department, there will be separate groups for full-time employees and for those who are not full-time, allowing for a more nuanced analysis.
2. **`['Training_Hours_Last_Year']`**: From each group created in the step above, this selects the 'Training_Hours_Last_Year' column. This column presumably contains numerical values representing the number of training hours each employee completed in the last year.
3. **`.mean()`**: This computes the mean (average) of the 'Training_Hours_Last_Year' values within each group. Since the groups are defined by both department and full-time status, this calculates the average training hours for each category of employee within each department.
4. **`avg_training_hours`**: The result of the mean calculation is assigned to this variable. The resulting object is likely a pandas Series with a MultiIndex. The first level of the index is the 'Department', and the second level is the 'Full_Time' status (which could be a boolean or some other indicator of whether an employee is considered full-time). The values of this Series are the average training hours for each combination of department and full-time status.

This operation allows you to understand not just how training hours are distributed across different departments, but also how this distribution might vary between full-time and part-time (or equivalent distinctions) employees within those departments. It's a powerful way to analyze the data for insights into training practices across different segments of the workforce.

[12]:
```
avg_training_hours
```

[12]:
```
Department  Full_Time
Finance     False        30.000000
            True         27.888889
HR          False        22.000000
            True         25.285714
```

```
IT          False        21.142857
            True         23.222222
Marketing   False        26.500000
            True         22.666667
Name: Training_Hours_Last_Year, dtype: float64
```

To illustrate the differences between `avg_training_hours.reset_index()` and `avg_training_hours.unstack()`, let's first clarify what each method does, especially in the context of a pandas Series with a MultiIndex created from grouping by multiple columns, as is the case with `avg_training_hours`.

1.4.1 `avg_training_hours.reset_index()`

- **What it does**: Converts the Series with a MultiIndex (in this case, 'Department' and 'Full_Time') into a DataFrame. The indices ('Department' and 'Full_Time') become columns in the resulting DataFrame, making it a "flat" structure with a default integer index.
- **Use case**: Useful when you want a simple DataFrame format where each group's identifying information ('Department' and 'Full_Time') is moved to columns alongside the values (average training hours).

1.4.2 `avg_training_hours.unstack()`

- **What it does**: Pivots the level of the specified index columns (if no level is specified, the last level is unstacked) to the columns, creating a new DataFrame where each unique value of the unstacked level becomes a column, and the values are the data points corresponding to each combination of indices.
- **Use case**: Useful for creating a pivot table-like structure where one of the indices ('Department' or 'Full_Time') is used to create columns, making it easier to compare across one of the categorical dimensions.

To understand the practical difference, let's apply both methods to the `avg_training_hours` Series. This will give us a direct comparison of the outputs.

Let's proceed by loading the data, performing the operations to create `avg_training_hours`, and then applying both `reset_index()` and `unstack()` to see the differences.

[13]:
```python
avg_training_hours.reset_index()
```

[13]:

	Department	Full_Time	Training_Hours_Last_Year
0	Finance	False	30.000000
1	Finance	True	27.888889
2	HR	False	22.000000
3	HR	True	25.285714
4	IT	False	21.142857
5	IT	True	23.222222
6	Marketing	False	26.500000
7	Marketing	True	22.666667

[14]:
```python
avg_training_hours.unstack()
```

[14]:
```
Full_Time        False       True
Department
Finance      30.000000  27.888889
HR           22.000000  25.285714
IT           21.142857  23.222222
Marketing    26.500000  22.666667
```

Based on the operations performed, here's a comparison of the results:

1.4.3 Using `reset_index()`

The `reset_index()` method transformed `avg_training_hours` into a DataFrame with three columns: 'Department', 'Full_Time', and 'Training_Hours_Last_Year'. Each row represents a unique combination of 'Department' and 'Full_Time', with the corresponding average training hours listed. For example, the Finance department has separate rows for full-time (True) and not full-time (False) employees, with average training hours of approximately 27.89 and 30.00 hours, respectively.

1.4.4 Using `unstack()`

The `unstack()` method, on the other hand, pivoted the 'Full_Time' level of the index to the columns, creating a DataFrame where each row represents a department, and there are separate columns for full-time (True) and not full-time (False) employees' average training hours. This structure makes it easy to compare the average training hours between full-time and not full-time employees within each department at a glance. For instance, in the Finance department, full-time employees have an average of approximately 27.89 training hours, while not full-time employees have an average of 30.00 hours.

Both methods are useful for data analysis, but they serve different purposes depending on your needs. `reset_index()` gives a "flattened" DataFrame that can be useful for further analysis or merging with other data, while `unstack()` provides a pivot table-like structure that is great for comparing subgroups side by side.

1.5 Apply Method

Concept: Pandas also allows for more advanced group operations such as filtering, transforming, and **applying** custom functions to groups.

Scenario: Suppose we want to categorize employees based on their years of experience into 'Junior', 'Mid-Level', and 'Senior'.

[15]:
```python
def cat_exp(years):
    if years<3:
        return 'Junior'
    elif years<7:
        return 'Mid-Level'
    else:
        return 'Senior'
```

[16]:
```
df['Experience_Level']=df['Years_of_Experience'].apply(cat_exp)
```

This block of code defines a function named `cat_exp` that categorizes years of experience into three levels: 'Junior', 'Mid-Level', and 'Senior'. It then applies this function to the 'Years_of_Experience' column of the DataFrame `df` to create a new column named 'Experience_Level' with the categorized experience levels. Here's a step-by-step explanation:

1. **Define the `cat_exp` function:**
   - The function takes a single argument `years`, which represents the number of years of experience.
   - It contains a series of conditional statements (`if`, `elif`, `else`) to categorize the experience based on the number of years:
     - If `years` is less than 3, it returns 'Junior'.
     - If `years` is 3 or more but less than 7, it returns 'Mid-Level'.
     - Otherwise, it returns 'Senior'.
2. **Apply the function to the DataFrame:**
   - `df['Years_of_Experience'].apply(cat_exp)` applies the `cat_exp` function to each element in the 'Years_of_Experience' column of `df`. The `.apply()` method is used to apply a function along an axis of the DataFrame or on values of a Series. Here, it's used on a Series to transform each year value into a categorical label according to the logic defined in `cat_exp`.
3. **Create a new column 'Experience_Level':**
   - The result of the `.apply()` operation, which is a Series of categorized experience levels, is assigned to a new column in `df` named 'Experience_Level'. This operation adds the 'Experience_Level' column to the DataFrame, where each row's value corresponds to the categorized experience level of the respective 'Years_of_Experience'.

By executing this code, `df` now includes a new column 'Experience_Level' that categorizes each employee's experience based on the number of years they have worked, providing a straightforward way to segment the workforce into experience-based groups.

[17]:
```
df.head(10)
```

[17]:
```
   Employee_ID Department  Years_of_Experience  Full_Time Performance_Score  \
0            1    Finance             1.820000      False              Good
1            2         IT             2.090000      False              Poor
2            3         IT             7.047228      False         Excellent
3            4    Finance             1.237509      False         Excellent
4            5  Marketing             4.160000       True           Average
5            6    Finance             2.475017       True         Excellent
6            7  Marketing            10.420000       True              Good
7            8  Marketing             6.102669       True              Good
8            9  Marketing            10.000000       True              Poor
9           10         HR             7.479808      False           Average

    Salary First_Name Last_Name        Date_of_Birth  Gender  …  \
0  56550.0    Michael     Davis  2000-05-11 05:33:39   Other  …
1  55225.0      Karen     Brown  1984-12-18 14:13:34   Other  …
```

```
2   87600.0     Joseph   Johnson  1991-07-29 11:32:58    Male  …
3   53875.0      David    Garcia  1998-01-24 12:21:27   Other  …
4   61400.0      Linda  Martinez  1973-06-05 08:09:32   Other  …
5   70550.0    Michael     Brown  1973-03-07 10:44:36  Female  …
6   82050.0    Charles     Moore  1981-03-26 17:29:10    Male  …
7   84750.0      David     Lopez  1981-03-25 12:42:58  Female  …
8   80500.0      David   Johnson  1973-08-09 18:43:34  Female  …
9  101350.0   Patricia  Martinez  1967-11-27 08:26:50  Female  …

  Remote_Work           Hire_Date  Last_Performance_Review_Date  \
0        True  2022-05-16 07:30:27           2022-10-08 07:30:27
1       False  2020-05-03 22:30:19           2020-08-15 22:30:19
2        True  2017-02-25 23:43:54           2017-07-17 23:43:54
3        True  2022-12-18 10:59:47           2023-04-30 10:59:47
4        True  2012-04-04 14:02:58           2013-01-12 14:02:58
5       False  2021-09-22 12:23:53           2022-06-05 12:23:53
6       False  2011-08-25 03:34:26           2012-01-13 03:34:26
7       False  2018-02-05 19:31:10           2018-06-13 19:31:10
8       False  2012-09-29 19:05:03           2013-03-30 19:05:03
9        True  2016-09-20 00:26:27           2017-05-01 00:26:27

  Project_Count Bonus  Employee_Satisfaction_Score  Department_Budget  \
0             5     0                         2.68              75000
1             6   500                         1.99             100000
2             3  2000                         2.42              50000
3             9   500                         4.03             125000
4             5   500                         1.06              75000
5             8     0                         1.46             150000
6             1  1500                         1.18              75000
7             5   500                         1.16             150000
8             3  1000                         4.42             100000
9             1  1500                         3.81             150000

   Training_Hours_Last_Year  Number_of_Direct_Reports  Experience_Level
0                        46                         6            Junior
1                        15                         8            Junior
2                         4                         3            Senior
3                        34                         3            Junior
4                        11                         5         Mid-Level
5                        24                         2            Junior
6                        20                         5            Senior
7                        35                         6         Mid-Level
8                        22                         9            Senior
9                        15                         9            Senior

[10 rows x 22 columns]
```

The `.apply()` method in pandas is a powerful and flexible tool used to apply a function along an axis of a DataFrame or to elements of a Series. It allows for both row-wise and column-wise operations in a DataFrame, and element-wise operations in a Series. Here's a more detailed explanation:

1.5.1 Applying to a DataFrame

When used on a DataFrame, `.apply()` can operate across rows or columns:

- **Column-wise operation (`axis=0`)**: By default, or when `axis=0` is specified, `.apply()` applies the given function to each column, treating each column as an array-like structure. This is useful for performing operations that need to consider each value in a column, like summing values or finding maximum values.
- **Row-wise operation (`axis=1`)**: When `axis=1` is specified, `.apply()` applies the function to each row. This mode is handy for operations that need to consider values across columns for each row, such as calculating a sum of certain columns for each row.

1.5.2 Applying to a Series

When `.apply()` is used on a Series, it applies a given function to each element in the Series. This is similar to mapping a function over an iterable, transforming each element in the Series according to the logic defined in the function.

1.5.3 General Use Cases

- **Data transformation**: You can use `.apply()` to perform complex data transformations, such as converting data types, applying conditional logic (like categorizing data based on values), or combining data from multiple columns.
- **Aggregation and summary statistics**: While pandas provides built-in methods for common operations (like `.sum()`, `.mean()`, etc.), `.apply()` allows for custom aggregation or summary statistics that aren't directly supported.
- **Applying custom functions**: `.apply()` shines when you have a specific, possibly complex operation that needs to be performed on a dataset, which is not covered by pandas' built-in methods. This includes applying mathematical formulas, data cleaning operations, or any user-defined function.

1.5.4 Important Considerations

- **Performance**: While `.apply()` is very flexible, it's not always the fastest option for large datasets, especially with complex operations. Vectorized operations using pandas' built-in methods or accessing the underlying NumPy arrays can be more efficient.
- **Broadcasting**: The function used with `.apply()` should return a value that pandas can incorporate into a Series or DataFrame. Depending on the operation, you might need to ensure that the return values are of a consistent format or type.

In summary, `.apply()` is a versatile method that can significantly enhance the data manipulation capabilities of pandas, allowing for custom, row-wise, column-wise, or element-wise operations on data structures.

1.6 Task 1:

calculate total salary: this should include the existing bonus; and added bonus for individuals that has a score of 4.00 or more. Added bounus is 1000 pounds

Here's how you can do it:

1. Define a function that takes an employee's satisfaction score as its input.
2. Inside the function, use an `if` statement to check if the score is 4.00 or higher.
3. Return 1000 if the condition is met, indicating the employee receives the added bonus.
4. Return 0 otherwise, indicating no added bonus for that employee.
5. Apply this function to the `Employee_Satisfaction_Score` column to create the `Added_Bonus` column.

Let's define the function and apply it:

[18]:
```python
# Define the function to determine the added bonus
def calculate_added_bonus(satisfaction_score):
    if satisfaction_score >= 4.00:
        return 1000
    else:
        return 0

# Apply the function to the Employee_Satisfaction_Score column
df['Added_Bonus'] = df['Employee_Satisfaction_Score'].
 ↪apply(calculate_added_bonus)

# Recalculate the Total_Salary to reflect any changes
df['Total_Salary'] = df['Salary'] + df['Bonus'] + df['Added_Bonus']

# Display the first few rows to verify the calculation
df[['Employee_Satisfaction_Score', 'Added_Bonus', 'Total_Salary']].head()
```

[18]:

	Employee_Satisfaction_Score	Added_Bonus	Total_Salary
0	2.68	0	56550.0
1	1.99	0	55725.0
2	2.42	0	89600.0
3	4.03	1000	55375.0
4	1.06	0	61900.0

The function to determine the added bonus based on the `Employee_Satisfaction_Score` has been successfully defined and applied. The `Added_Bonus` column now reflects the additional 1000 pounds for employees with a satisfaction score of 4.00 or higher, as calculated by the custom function. The `Total_Salary` column has been recalculated to include this `Added_Bonus`. Here are the results for the first few rows:

- The `Employee_Satisfaction_Score` column shows the satisfaction scores of the employees.
- The `Added_Bonus` column indicates the additional bonus, where only employees with a score of 4.00 or more receive the 1000 pounds bonus. For instance, the employee at index 3, with a satisfaction score of 4.03, receives the added bonus, resulting in an `Added_Bonus` of 1000

pounds.
- The `Total_Salary` column has been updated to include any added bonuses alongside the original salary and bonus, accurately reflecting the total compensation for each employee.

1.7 Task 2

Split the Location column into City and Country. For example `Glasgow, Scotland` should be split into two `Glasgow` and `Scotland`

To split the `Location` column in your DataFrame into two separate columns, `City` and `Country`, you can use the `.str.split()` method on the `Location` column with the appropriate separator. Assuming the format of the `Location` values is consistent and uses a comma as the separator (as in "Glasgow, Scotland"), here's how you can do it:

1. Use the `.str.split()` method on the `Location` column, specifying `,` as the separator and `expand=True` to return a DataFrame.
2. Assign the result to two new columns in your existing DataFrame, `City` and `Country`. Let's implement this approach.

[19]:
```python
# Split the Location column into City and Country
df[['City', 'Country']] = df['Location'].str.split(',', expand=True)

# Display the first few rows to verify the split
df[['Location', 'City', 'Country']].head()
```

[19]:

	Location	City	Country
0	Glasgow, Scotland	Glasgow	Scotland
1	London, England	London	England
2	Belfast, Northern Ireland	Belfast	Northern Ireland
3	Belfast, Northern Ireland	Belfast	Northern Ireland
4	Manchester, England	Manchester	England

The `Location` column has been successfully split into two new columns, `City` and `Country`. For example, "Glasgow, Scotland" has been split into "Glasgow" for the city and "Scotland" for the country, as seen in the first row. This process has been applied across the DataFrame, allowing for more specific geographic analysis or filtering based on either city or country.

[ ]:
```python
```