Data Transformation

Indexing DataFrames in pandas

While working with lists in Python, we can fetch the data present at any location using the index-based approach. Let's say a list of employee names **emp_name**.

emp_name = *[a, b, c, d, e]*

Now if we want to fetch the employee name present at the 3rd location in the list then we will use **emp_name[2]** to fetch the employee name. This technique is called the index-based approach because we are using the index of that location to fetch the data.

Similarly, we also need to fetch the data present at a particular location in pandas. For these various techniques are present, here we are going to learn a different kind of indexing that exists in pandas data frames.

You are going to learn the following indexing techniques:

1. **DataFrame.loc[]**
2. **DataFrame.iloc[]**
3. **DataFrame.at[]**
4. **DataFrame.iat[]**

pandas DataFrame.loc[]

A pandas dataframe is a two-dimensional tabular data structure which contains labelled axes (rows and columns). The dataFrame.loc[] is a label-based indexing method which can fetch the data from a group of columns and rows by labels or a boolean array.

Let's take a look at some of the examples of using dataframe.loc[] using the Bigmart sales data.

- **Fetching all the details of a single row.**

```
1  df.loc[4]
```

```
Item_Identifier                 NCD19
Item_Weight                      8.93
Item_Fat_Content              Low Fat
Item_Visibility                  0.0
Item_Type                   Household
Item_MRP                      53.8614
Outlet_Identifier              OUT013
Outlet_Establishment_Year        1987
Outlet_Size                      High
Outlet_Location_Type           Tier 3
Outlet_Type          Supermarket Type1
Item_Outlet_Sales             994.7052
Name: 4, dtype: object
```

Here we only need to pass the index of the row or if the label for the row is present then pass that label to the dataframe.loc[] as shown in the example.

- **Fetching multiple rows using index or labels of the rows.**

```
1  df.loc[[4, 5, 7]]
```

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type
4	NCD19	8.930	Low Fat	0.00000	Household
5	FDP36	10.395	Regular	0.00000	Baking Goods
7	FDP10	NaN	Low Fat	0.12747	Snack Foods

We need to pass the index or label of rows as a list.

- **Fetching all the rows of a single column.**

```
1  df.loc[:, 'Item_Type']
```

```
0                    Dairy
1              Soft Drinks
2                     Meat
3       Fruits and Vegetables
4                Household
               ...
8518           Snack Foods
8519          Baking Goods
8520       Health and Hygiene
8521           Snack Foods
8522           Soft Drinks
Name: Item_Type, Length: 8523, dtype: object
```

To fetch all the rows of a single column we need to use the following syntax:
df.loc[:, “column_name”]. Here “:” is used to fetch all the rows.

- **Fetch a range of rows for particular columns.**

```
df.loc[5:10, ['Item_Fat_Content', 'Item_Type']]
```

	Item_Fat_Content	Item_Type
5	Regular	Baking Goods
6	Regular	Snack Foods
7	Low Fat	Snack Foods
8	Regular	Frozen Foods
9	Regular	Frozen Foods
10	Low Fat	Fruits and Vegetables

Here we are fetching the values of **Item_Fat_Content** and **Item_Type** for the rows ranging from 5 to 10.

- **Fetching data using Boolean indexing**

```
df.loc[df['Item_Fat_Content'] == "Regular", ['Item_Fat_Content', 'Item_Type']]
```

	Item_Fat_Content	Item_Type
1	Regular	Soft Drinks
3	Regular	Fruits and Vegetables
5	Regular	Baking Goods
6	Regular	Snack Foods
8	Regular	Frozen Foods
...	...	...
8510	Regular	Snack Foods
8513	Regular	Meat
8514	Regular	Canned
8519	Regular	Baking Goods
8521	Regular	Snack Foods

2889 rows × 2 columns

Here, all the rows which have **Regular** as **Item_Fat_Content** have been fetched and only ['Item_Fat_Content', 'Item_Type'] are displayed.

For further reading on loc method, you can access this link -> pandas.DataFrame.loc

pandas DataFrame.iloc[]

The disadvantage of the DataFrame.loc[] method is that we cannot use integer-based indexing while selecting the columns. If we want to fetch the columns from 2 to 6 in a dataframe then it is not possible with DataFrame.loc[].

```
1  df.loc[5:10, 2:4]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-468a3e459491> in <module>
----> 1 df.loc[5:10, 2:4]

                                  8 frames
/usr/local/lib/python3.8/dist-packages/pandas/core/indexes/base.py in _maybe_cast_slice_bound(self, label,
side, kind)
   5747         # reject them, if index does not contain label
   5748         if (is_float(label) or is_integer(label)) and label not in self._values:
-> 5749             raise self._invalid_indexer("slice", label)
   5750
   5751         return label

TypeError: cannot do slice indexing on Index with these indexers [2] of type int
```

For integer-based indexing, there is one other method provided by pandas called DataFrame.iloc[]. This works in the same way as DataFrame.loc[] just that it only accepts integer-based indexing.

```
1  df.iloc[5:10, 2:4]
```

	Item_Fat_Content	Item_Visibility
5	Regular	0.000000
6	Regular	0.012741
7	Low Fat	0.127470
8	Regular	0.016687
9	Regular	0.094450

For further reading on iloc method, you can access this link -> pandas.DataFrame.iloc

Till now we have learned about DataFrame.loc[] and DataFrame.iloc[]. These are very helpful while fetching data from a particular location from a dataframe. We can also use these methods to fetch a single value for a row/column pair. But they are not designed to fetch a single value they are mostly used to fetch a range of values. To fetch single values pandas provide two other methods similar to loc[] and iloc[], these are at[] and iat[]. The main difference between these is the computation time. The at[] and iat[] methods are specifically designed for fetching a single value and thus work faster than loc[] and iloc[].

Let's take a look at some of the examples of using dataframe.at[] and dataframe.iat[] on the Bigmart sales data.

pandas DataFrame.at[]

Fetching the value present in the 6th row for the "Item_Type" column using the df.at[].

```
1   df.at[5, 'Item_Type']
```

```
'Baking Goods'
```

The df.at[] is similar to the df.loc[], it only takes label-based indexing for the columns.

Comparison of df.at[] and df.loc[]

The below code is the comparison of fetching a single value using df.at[] and df.loc[].

```
1   %timeit df.loc[5, 'Item_Type']
```

```
8.17 µs ± 192 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
1   %timeit df.at[5, 'Item_Type']
```

```
4.18 µs ± 146 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

As it is clearly seen that df.at[] is taking nearly half the time compared to df.loc[] after running 100000 times. This is a substantial improvement over time while working with a huge volume of datasets.

For further reading on at method, you can access this link -> pandas.DataFrame.at

pandas DataFrame.iat[]

The dataframe.iat[] method is similar to dataframe.iloc[], it takes integer-based indexing for the columns. Following is an example showing the usage of this method.

```
1  df.iat[5, 4]
```

```
'Baking Goods'
```

Here, we are fetching the value of the cell present in the sixth row and 5th column.

Comparison of df.iat[] and df.iloc[]

```
1  %timeit df.iloc[5, 4]
```

```
27 µs ± 1.93 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
1  %timeit df.iat[5, 4]
```

```
21.3 µs ± 496 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

As we can see there is an improvement in the time taken by the df.iat[] method in comparison to the df.iloc[] method.
So, it is recommended to use df.at[] and df.iat[] instead of using df.loc[] and df.iloc[] while fetching only a single value or assigning a new value to a particular cell.

For further reading on iat method, you can access this link: pandas.DataFrame.iat