PANDAS

(in depth - 2)

Data Manipulation

Comprises of following three stages:

- Data preparation : we looked at various functions such as merge(), concat, combine, pivot etc for data preparation.

This lecture, we will look at

- Data transformation
- Data aggregation

Dropping duplicates

Detecting duplicates rows in huge datasets can be problematic. Pandas provides tools for handling duplicate values.

- The **duplicated()** function applied to a DataFrame can detect the rows which appear to be duplicated.
- It returns a Series of Booleans where each element corresponds to a row, with **True** if the row is duplicated (i.e., only the other occurrences, not the first), and with **False** if there are no duplicates in the previous elements.

Creating a Dataframe with duplicate rows

- dframe = pd.DataFrame(’color’: [’white’,’white’,’red’,’red’,’white’],’value’: [2,1,3,3,2])

print(dframe)

	color	value
0	white	2
1	white	1
2	red	3
3	red	3
4	white	2

Detecting duplicates

```
>>> dframe.duplicated()
0     False
1     False
2     False
3     True
4     True
dtype: bool
```

Boolean returns and removing duplicates

- We can make use of the fact that the result of this operation is a boolean series to filter rows:
- To find the duplicate rows, just type:

```
>>> dframe[dframe.duplicated()]

   color  value
3  red    3
4  white  2
```

- The **drop_duplicates()** function, returns the DataFrame without duplicate rows.

Replace

Often in the data structure that you have assembled, there are values that do not meet your needs.

- For instance, some of the text may be in a foreign language,
- may contain unwanted synonyms,
- may be in the wrong shape etc.

In such cases, we can use the replace function.

```
>>> frame = pd.DataFrame({ 'item':['ball','mug','pen','pencil','ashtray'],
                          'color':['white','rosso','verde','black','yellow'],
                          'price':[5.56,4.20,1.30,0.56,2.75]})
```

```
>>> frame
     color     item
0    white     ball
1    rosso     mug
2    verde     pen
3    black     pencil
4    yellow    ashtray
```

Creating a mapping

- First, we create a mapping as follows:

```
>>> newcolors = {
       'rosso': 'red’,
       'verde': 'green’
}
```

- Now we use replace using the mapping as an argument:

```
>>> frame.replace(newcolors)
```

	color	item	price
0	white	ball	5.56
1	red	mug	4.20
2	green	pen	1.30
3	black	pencil	0.56
4	yellow	ashtray	2.75

Replacing instances of NaN

For example with 0s:

```
>>> ser = pd.Series([1,3,np.nan,4,6,np.nan,3])
>>> ser.replace(np.nan,0)
0    1
1    3
2    0
3    4
4    6
5    0
6    3
dtype: float64
```

Using mapping to add values into a column

- The mapping is always defined separately. First defining the dataframe:

```
>>> frame = pd.DataFrame({ 'item':['ball','mug','pen','pencil','ashtray’],
                          'color':['white','red','green','black','yellow']})

>>> print(frame)

    color     item
0   white     ball
1   red       mug
2   green     pen
3   black     pencil
4   yellow    ashtray
```

The mapping

- Let's suppose you want to add a column to indicate the price of the item shown in the DataFrame ‘frame’. Assume you have a price list available somewhere, in which the price for each type of item is described. Then, define a dict object that contains a list of prices for each type of item.

```
>>> price = {'ball' : 5.56, 'mug' : 4.20, 'bottle' : 1.30, 'scissors' : 3.41, 'pen' : 1.30, 'pencil' : 0.56,   'ashtray’ :
                    2.75}
```

Applying the mapping

- The **map()** function applied to a Series or to a column of a DataFrame accepts a function or an object containing a dict with mapping. So in your case you can apply the mapping of the prices on the column item, making sure to add a column to the price data frame.

```
>>> frame['price'] = frame['item'].map(prices)

>>> frame    # print the altered dataFrame
```

	color	item	price
0	white	ball	5.56
1	red	mug	4.20
2	green	pen	1.30
3	black	pencil	0.56
4	yellow	ashtray	2.75

Discretization and Binning

- Supposing we have readings of an experimental value between 0 and 100. These data are collected in a list.

```
>>> results = [12,34,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]
```

- You know that the experimental values have a range from 0 to 100; therefore you can uniformly divide this interval, for example, into four equal parts, i.e., bins. The first contains the values between 0 and 25, the second between 26 and 50, the third between 51 and 75, and the last between 76 and 100.

- To do this binning with pandas, first you have to define an array containing the values for the separation of the bins:

```
>>> bins = [0,25,50,75,100]
```

- Then there is a special function called **cut()** which is applied to the array of results, passing the bins.

```
>>> cat = pd.cut(results, bins)
```

Discretization and Binning ..

```
print(cat)      # gives the following output

 (0, 25]                      # value 12 belongs to this bin
 (25, 50]
 (50, 75]
 (50, 75]
 (25, 50]
 (75, 100]
 (75, 100]
 (0, 25]
 (0, 25]
 (50, 75]
 (50, 75]
 (25, 50]
 (75, 100]
 (0, 25]
 (25, 50]
 (75, 100]
 (75, 100]
Levels (4): Index(['(0, 25]', '(25, 50]', '(50, 75]', '(75, 100]'], dtype=object)
```

Discretisation and binning ..

- The object returned by the **cut()** function is a special object of **Categorical** type. You can consider it as an array of strings indicating the name of the bin. Internally it contains a **levels** array indicating the names of the different internal categories and a **labels** array that contains a list of numbers equal to the elements of **results** (i.e., the array subjected to binning).
- The number corresponds to the bin to which the corresponding element of **results** is assigned.

```
>>> cat.levels
Index(['(0, 25]', '(25, 50]', '(50, 75]', '(75, 100]'], dtype='object')
```

Discretisation and binning ..

```
>>> cat.labels
array([0, 1, 2, 2, 1, 3, 3, 0, 0, 2, 2, 1, 3, 0, 1, 3, 3], dtype=int64)
```

- Finally to know the occurrences for each bin, that is, how many results fall into each category, you have to use the value_**counts()** function.

```
>>> pd.value_counts(cat)

 (75, 100]     5
 (0, 25]       4
 (25, 50]      4
(50, 75]       4
dtype: int64
```

Detecting and filtering outliers

- We often wish to detect and remove outlying datapoints.
- By way of example, create a DataFrame with three columns from 1,000 completely random values:

```
>>> randframe = pd.DataFrame(np.random.randn(1000,3))
```

- With the **describe()** function you can see the statistics for each column.

```
>>> randframe.describe()
```

	0	1	2
count	1000.000000	1000.000000	1000.000000
mean	0.021609	-0.022926	-0.019577
std	1.045777	0.998493	1.056961
min	-2.981600	-2.828229	-3.735046
25%	-0.675005	-0.729834	-0.737677
50%	0.003857	-0.016940	-0.031886
75%	0.738968	0.619175	0.718702
max	3.104202	2.942778	3.458472

Detecting and removing outliers ..

- For example, you might consider outliers those that have a value greater than three times the standard deviation.
- To have only the standard deviation of each column of the DataFrame, use the **std()** function:

```
>>> randframe.std()
0    1.045777
1    0.998493
2    1.056961
dtype: float64
```

Detecting and removing outliers ..

- Now we apply the filter to all the values of the DataFrame, applying the corresponding standard deviation for each column.
- The **any()** function, enables easy application of the filter to each column.

```
>>> randframe[(np.abs(randframe) > (3*randframe.std())).any(1)]    # displays following
```

	0	1	2
69	-0.442411	-1.099404	3.206832
576	-0.154413	-1.108671	3.458472
907	2.296649	1.129156	-3.735046

Permutation

- Permutation operations (the random reordering) of a Series or the rows of a DataFrame are easy to do using the **numpy.random.permutation()** function.

```
>>> nframe = pd.DataFrame(np.arange(25).reshape(5,5))

print(nframe)     # produces following

0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
3  15  16  17  18  19
4  20  21  22  23  24
```

Permutation ..

- Now create an array of five integers from 0 to 4 arranged in random order with the **permutation()** function. This will be the new order in which to determine the order of the rows in the DataFrame.
- >>> new_order = np.random.permutation(5)

```
print(new_order)        # gives the following output
array([2, 3, 0, 1, 4])
```

- Now apply it to all of the rows of the DataFrame, using the **take()** function:

```
>>> nframe.take(new_order)
```

	0	1	2	3	4
2	10	11	12	13	14
3	15	16	17	18	19
0	0	1	2	3	4
1	5	6	7	8	9
4	20	21	22	23	24

- Now the indices follow the same order as indicated in the **new_order** array.

Permutation ..

- You can submit just a portion of the entire DataFrame to a permutation. It generates an array that has a sequence limited to a certain range, for example, in our case from 2 to 4.

```
>>> new_order = [3,4,2]
>>> nframe.take(new_order)

      0   1   2   3   4
3    15  16  17  18  19
4    20  21  22  23  24
2    10  11  12  13  14
```

Random sampling

- Sometimes, when you have a huge DataFrame, you may have the need to sample it randomly, and the quickest way to do this is by using the **np.random.randint()** function.

```
>>> sample = np.random.randint(0, len(nframe), size=3)
>>> sample
array([1, 4, 4])

# take random samples

>>> nframe.take(sample)
    0   1   2   3   4
1   5   6   7   8   9
4  20  21  22  23  24
4  20  21  22  23  24
```

Data aggregation

- The last stage of data manipulation is data aggregation.
- By data aggregation we often mean a transformation that produces a single integer from an array. We have already seen examples using sum, mean, count etc.
- A major function for aggregation in Pandas is GroupBy.

GroupBy

We can think of the GroupBy process as comprising of 3 stages: Splitting, applying and combining.

- Splitting: The initial splitting into groups is usually done on the basis of a common index or data value.
- Applying: The second phase, that of applying, consists in applying a function, or better a calculation, which will produce a new and single value per group.
- Combining: The last phase, that of combining, will collect all the results obtained from each group and combine them together to form a new object.

GroupBy ..

We define a DataFrame containing both numeric and string values as:

```
>>> frame = pd.DataFrame({ 'color': ['white','red','green','red','green'],
                  'object': ['pen','pencil','pencil','ashtray','pen'],
                  'price1' : [5.56,4.20,1.30,0.56,2.75],
                  'price2' : [4.75,4.12,1.60,0.75,3.15]})
```

```
>>> print(frame)     # prints the frame contents
```

	color	object	price1	price2
0	white	pen	5.56	4.75
1	red	pencil	4.20	4.12
2	green	pencil	1.30	1.60
3	red	ashtray	0.56	0.75
4	green	pen	2.75	3.15

GroupBy ..

- Suppose you want to calculate the average **price1** column using group labels listed in the column color. There are several ways to do this. You can for example access the **price1** column and call the **groupby()** function with the column color.

```
>>> group = frame['price1'].groupby(frame['color'])
```

```
>>> print(group)      # will print the following
```

```
<pandas.core.groupby.SeriesGroupBy object at 0x00000000098A2A20>
```

- The object that we got is a **GroupBy** object.
- In the operation that you just did there was not really any calculation; there was just a collection of all the information needed to go into the calculation.
- What you have done is in fact a process of grouping, in which all rows having the same value of color are grouped into a single item.

GroupBy ..

- To analyse in detail how the division into groups of rows of the DataFrame was made, you call the attribute groups of the GroupBy object.

```
>>> group.groups
{'white': [0L], 'green': [2L, 4L], 'red': [1L, 3L]}
```

- Each group is listed explicitly specifying the rows of the data frame assigned to each of them.

GroupBy ..

- Now we can apply the operation to obtain the results for each individual group:

```
>>> group.sum()

color
green     4.05
red       4.76
white     5.56
Name: price1, dtype: float64
```

```
>>> group.mean()

color
green     2.025
red       2.380
white     5.560
Name: price1, dtype: float64
```

Hierarchical grouping

- The same thing can be extended to multiple columns, i.e., make a grouping of multiple keys:

```
>>> ggroup = frame['price1'].groupby([frame['color'],frame['object']])

>>> ggroup.groups
{('red', 'ashtray'): [3L], ('red', 'pencil'): [1L], ('green', 'pen'): [4L], ('green', ‘pencil’:[2L], ('white', 'pen'): [0L]}

>>> ggroup.sum()

color   object
green   pen         2.75
        pencil      1.30
red     ashtray     0.56
        pencil      4.20
white   pen         5.56
Name: price1, dtype: float64
```

Hierarchical grouping ..

- So far we have applied the grouping to a single column of data. It can be extended to multiple columns or the entire data frame.
- Also if you do not need to reuse the object GroupBy several times, it is convenient to combine into a single pass all of the groupings and calculations to be done, without defining any intermediate variable.

```
>>> frame[['price1','price2']].groupby(frame['color']).mean()

        price1  price2
color
green   2.025   2.375
red     2.380   2.435
white   5.560   4.750
```

Contents of frame from previous slide:

```
>>> frame
   color   object   price1  price2
0  white   pen      5.56    4.75
1  red     pencil   4.20    4.12
2  green   pencil   1.30    1.60
3  red     ashtray  0.56    0.75
4  green   pen      2.75    3.15
```

Group iteration

- The **GroupBy** object supports the operation of an iteration for generating a sequence of 2-tuples containing the name of the group together with the data portion.

```
>>> for name, group in frame.groupby('color'):
        print name
        print group
```

Will output the following:

green

	color	object	price1	price2
2	green	pencil	1.30	1.60
4	green	pen	2.75	3.15

red

	color	object	price1	price2
1	red	pencil	4.20	4.12
3	red	ashtray	0.56	0.75

white

	color	object	price1	price2
0	white	pen	5.56	4.75

- ✓ In this example, we only applied the print function for illustration.
- ✓ In practice, you replace the printing operation of a variable with the function to be applied.