Pandas - Data Manipulation

Data Manipulation - Merging

- `The merge()` function is used to combine data through the connection of the rows using one or more keys (`id` field in this example).

```
frame1 = pd.DataFrame({'id':['ball','pencil','pen','mug','ashtray'],
                       'price':[12.33,11.44,33.21,13.23,33.62]})
frame1
```

	id	price
0	ball	12.33
1	pencil	11.44
2	pen	33.21
3	mug	13.23
4	ashtray	33.62

```
frame2 = pd.DataFrame({'id':['pencil','pencil','ball','pen'],
                       'color':['white','red','red','black']})
frame2
```

	id	color
0	pencil	white
1	pencil	red
2	ball	red
3	pen	black

```
pd.merge(frame1,frame2)
```

Merged files without specifying any column

	id	price	color
0	ball	12.33	red
1	pencil	11.44	white
2	pencil	11.44	red
3	pen	33.21	black

The returned dataframe consists of all rows that have an ID in common

Data Manipulation - Merging

When two dataframes have columns with the same name, it is necessary to explicitly define the name of the key column in the `on` option.

```
frame3 = pd.DataFrame({'id':['ball','pending','pen','mug','ashtray'],
                       'color':['white','red','red','black','green'],
                       'brand':['OMG','ABC','ABC','POD','POD']})
frame3
```

	id	color	brand
0	ball	white	OMG
1	pending	red	ABC
2	pen	red	ABC
3	mug	black	POD
4	ashtray	green	POD

```
frame4 = pd.DataFrame({'id':['pencil','pencil','ball','pen'],
                       'brand': ['OMG','POD','ABC','POD']})
frame4
```

	id	brand
0	pencil	OMG
1	pencil	POD
2	ball	ABC
3	pen	POD

```
pd.merge(frame3,frame4,on='brand')
```

	id_x	color	brand	id_y
0	ball	white	OMG	pencil
1	pending	red	ABC	ball
2	pen	red	ABC	ball
3	mug	black	POD	pencil
4	mug	black	POD	pen
5	ashtray	green	POD	pencil
6	ashtray	green	POD	pen

Criteria for merging: ID

```
pd.merge(frame3,frame4, on='id')
```

	id	color	brand_x	brand_y
0	ball	white	OMG	ABC
1	pen	red	ABC	POD

Criteria for merging: brand

Data Manipulation - Merging

- Instead of considering the columns of a dataframe as keys, indexes could be used as keys for merging. Set `left_index` or `right_index` options to True to activate which indexes to consider.

```
pd.merge(frame3,frame4,right_index=True, left_index=True)
```

	id	color	brand_x	brand_y	sid
0	ball	white	OMG	pencil	OMG
1	pencil	red	ABC	pencil	POD
2	pen	red	ABC	ball	ABC
3	mug	black	POD	pen	POD

- The `join()` function is more convenient when you want to merge by indexes. When some columns in frame3 have the same name as the columns in frame4, rename the columns in frame4 before launching the `join()` function.

```
frame4.columns = ['brand2','id2']
```

```
frame3.join(frame4)
```

	id	color	brand	brand2	id2
0	ball	white	OMG	pencil	OMG
1	pencil	red	ABC	pencil	POD
2	pen	red	ABC	ball	ABC
3	mug	black	POD	pen	POD
4	ashtray	green	POD	NaN	NaN

Data Manapulation - Concatenating

- `Concatenation` is another type of data combination.
- `NumPy` provides a `concatenate()` function to combine arrays

```
array1 = np.arange(9).reshape((3,3))
```

```
array1
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
array2 = np.arange(9).reshape((3,3))+6
array2
```

```
array([[ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

```
np.concatenate([array1,array2],axis=1)
```

```
array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11],
       [ 6,  7,  8, 12, 13, 14]])
```

```
np.concatenate([array1,array2],axis=0)
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Data Manipulation - Concatenating Series

- With pandas library and its data structure like `series` and `dataframe`, having labelled axes allows further generalisation of the concatenation of arrays.

- The `concat()` function

```
ser1
```

```
1    0.326100
2    0.983239
3    0.306811
4    0.149875
dtype: float64
```

```
ser2 = pd.Series(np.random.rand(4), index=[5,6,7,8])
```

```
ser2
```

```
5    0.221997
6    0.687002
7    0.499663
8    0.857193
dtype: float64
```

```
pd.concat([ser1,ser2])
```

```
1    0.326100
2    0.983239
3    0.306811
4    0.149875
5    0.221997
6    0.687002
7    0.499663
8    0.857193
dtype: float64
```

Data Manipulation - Concatenating Series

- To create a hierarchical index on the concatenation, use the `keys` option.

```
pd.concat([ser1,ser2], keys=[1,2])
```

```
1  1    0.326100
   2    0.983239
   3    0.306811
   4    0.149875
2  5    0.221997
   6    0.687002
   7    0.499663
   8    0.857193
dtype: float64
```

```
pd.concat([ser1,ser2], axis = 1, keys=[1,2])
```

- In the case of combinations between series along the axis = 1 the keys become the column headers of the dataframe.

	1	2
1	0.326100	NaN
2	0.983239	NaN
3	0.306811	NaN
4	0.149875	NaN
5	NaN	0.221997
6	NaN	0.687002
7	NaN	0.499663
8	NaN	0.857193

Data manipulation - Concatenating Dataframes

- The same logic of concatenation applied to the series can be applied to the dataframe

```
frame2 = pd.DataFrame(np.random.rand(9).reshape(3,3), index=[4,5,6],
                      columns=['A','B','C'])
```

```
pd.concat([frame1,frame2])
```

	A	B	C
1	0.976314	0.748882	0.955794
2	0.046396	0.449692	0.867622
3	0.433338	0.986343	0.323115
4	0.802874	0.773448	0.922387
5	0.580696	0.584984	0.276520
6	0.725205	0.017955	0.974704

```
pd.concat([frame1,frame2], axis=1)
```

	A	B	C	A	B	C
1	0.976314	0.748882	0.955794	NaN	NaN	NaN
2	0.046396	0.449692	0.867622	NaN	NaN	NaN
3	0.433338	0.986343	0.323115	NaN	NaN	NaN
4	NaN	NaN	NaN	0.802874	0.773448	0.922387
5	NaN	NaN	NaN	0.580696	0.584984	0.276520
6	NaN	NaN	NaN	0.725205	0.017955	0.974704

Combining

- Sometimes, a combination of data cannot be obtained either with merging or with concatenation.
- This can be applied to series using the combine_first() function
- For example we want the two datasets to have indexes that overlap partially or entirely.

```
ser1 = pd.Series(np.random.rand(5), index=[1,2,3,4,5])
ser1
```

```
1    0.598546
2    0.172542
3    0.738250
4    0.682647
5    0.013372
dtype: float64
```

```
ser2 = pd.Series(np.random.rand(4), index=[2,4,5,6])
ser2
```

```
2    0.504086
4    0.421815
5    0.970975
6    0.107031
dtype: float64
```

```
ser1.combine_first(ser2)
```

```
1    0.598546
2    0.172542
3    0.738250
4    0.682647
5    0.013372
6    0.107031
dtype: float64
```

```
ser2.combine_first(ser1)
```

```
1    0.598546
2    0.504086
3    0.738250
4    0.421815
5    0.970975
6    0.107031
dtype: float64
```

Data Manipulation - Pivoting

- After putting together the data in order to unify the values collected from different sources, the values can be arranged and rearranged by column values on rows or vice versa. This is operation is know as `pivoting`.

- In pivoting you have two basic operations:
  - `Stacking` – rotates or pivots the data structure converting columns to rows
  - `Unstacking` – converts row to columns

Data Preparation - Pivoting

```
frame1 = pd.DataFrame(np.arange(9).reshape(3,3),
                      index=['white','black','red'],
                      columns=['ball','pen','pencil'])
```

```
frame1
```

	ball	pen	pencil
white	0	1	2
black	3	4	5
red	6	7	8

Initial dataframe

```
ser5 = frame1.stack()
ser5
```

```
white  ball      0
       pen       1
       pencil    2
black  ball      3
       pen       4
       pencil    5
red    ball      6
       pen       7
       pencil    8
dtype: int32
```

Using the stack() function on the dataframe, you get the pivoting of columns in rows, thus producing a series

Reassemble the table with unstack() function

```
ser5.unstack()
```

	ball	pen	pencil
white	0	1	2
black	3	4	5
red	6	7	8

```
ser5.unstack(0)
```

	white	black	red
ball	0	3	6
pen	1	4	7
pencil	2	5	8

You can also unstack on a different level

Pivoting from ‘Long’ to ‘Wide’ format

- Pivoting a type of dataset that have entries on various columns, sometimes duplicated lines – e.g.. A log file that is accumulating data.
- This can be difficult to read and in fully understanding the relationship between the key values and the rest of the columns.
- Instead of long format, the data can be arranged in a table that is *wide* using the `pivot()` function.

```
longframe = pd.DataFrame({'color':['white','white','white',
                                   'red','red','red',
                                   'black','black','black'],
                          'item':['ball','pen','mug',
                                  'ball','pen','mug',
                                  'ball','pen','mug'],
                          'value':np.random.rand(9)})
```

```
longframe
```

	color	item	value
0	white	ball	0.587818
1	white	pen	0.490479
2	white	mug	0.912572
3	red	ball	0.423560
4	red	pen	0.446265
5	red	mug	0.711930
6	black	ball	0.524044
7	black	pen	0.812680
8	black	mug	0.541409

Long type: difficult to read and understanding in more complex data

Pivot() function allows you to transform a dataframe from the *long* to the *wide* type. More efficient and takes less space

```
widetable = longframe.pivot('color','item')
widetable
```

	value		
item	ball	mug	pen
color			
black	0.524044	0.541409	0.812680
red	0.423560	0.711930	0.446265
white	0.587818	0.912572	0.490479

Data preparation - Removing

- The last stage of data preparation is the removal of columns and rows.
- In order to remove a `column`, use `del` command applied to the dataframe with the column name specified.
- To remove a `row`, use the `drop()` function with the label of the corresponding index as argument

```
frame1 = pd.DataFrame(np.arange(9).reshape(3,3),
                      index=['white','black','red'],
                      columns=['ball','pen','pencil'])
```

```
frame1
```

	ball	pen	pencil
white	0	1	2
black	3	4	5
red	6	7	8

Initial dataframe

Removing a column called *ball*

```
del frame1['ball']
```

```
frame1
```

	pen	pencil
white	1	2
black	4	5
red	7	8

Removing a row called *white*

```
frame1.drop('white')
```

	pen	pencil
black	4	5
red	7	8

Data Transformation – Removing Duplicates

- The `duplicated()` function detect the rows that are duplicated. It returns `True` if a row is duplicate and `False` if it is not.
- The `drop_duplicates()` function will return the dataframe without duplicate rows

```
dframe = pd.DataFrame({'color':['white','white',
                                'red','red','white']})
```

```
dframe
```

	color
0	white
1	white
2	red
3	red
4	white

Initial dataframe

```
dframe.duplicated()
```

```
0    False
1     True
2    False
3     True
4     True
dtype: bool
```

Detect duplicated rows

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

	color
1	white
3	red
4	white

Identify the duplicate rows and in this case these are rows 1, 3 and 4

```
dframe.drop_duplicates()
```

	color
0	white
2	red

delete duplicated rows

- Mapping is the creation of a list of matches between two different values, with the ability to bind a value to a particular label or string

```
map = {
    'label1': 'value1',
    'label2': 'value2',
    'label3': 'value3'
}
```

```
map
```

```
{'label1': 'value1', 'label2': 'value2', 'label3': 'value3'}
```

- The `replace()` function replaces values
- The `map()` function creates a new column.
- The `rename()` function replaces the index values.

Replacing values via mapping

```
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
```

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

Initial dataframe with incorrect values: rosso and verde

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

To replace the incorrect values, *first* define a mapping of correspondences containing as a key the new value.

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

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

Use the *replace()* function with the mapping as argument to replace the incorrect values

Replacing values via mapping

```
ser = pd.Series([1,3,np.nan,4,6,np.nan,3])
ser
```

```
0    1.0
1    3.0
2    NaN
3    4.0
4    6.0
5    NaN
6    3.0
dtype: float64
```

```
ser.replace(np.nan,0)
```

```
0    1.0
1    3.0
2    0.0
3    4.0
4    6.0
5    0.0
6    3.0
dtype: float64
```

Adding Values via 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.

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

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

Initial dataframe without the price column

Define a dictionary object that contains a list of prices for each type of item.

```
prices = {
    'ball': 5.26,
    'mug': 4.20,
    'bottle': 1.30,
    'scissors' : 3.41,
    'pen' : 1.30,
    'pencil': 0.56,
    'ashtray': 2.75
}
```

Add the price column to the frame using *map()*

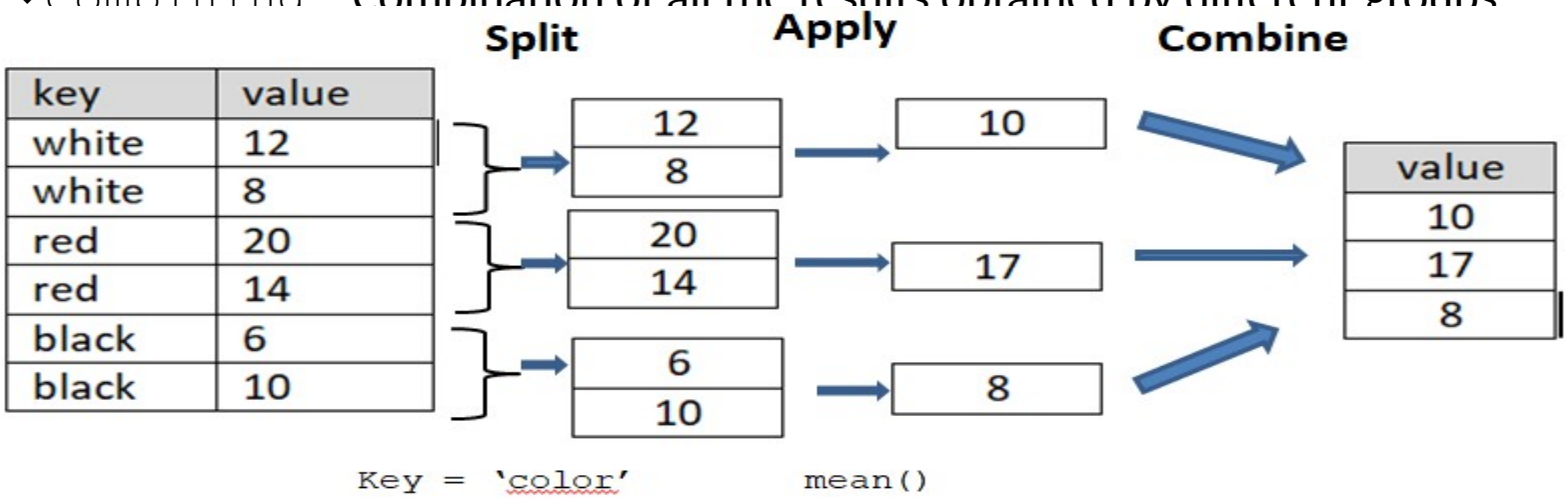
```
frame['price'] = frame['item'].map(prices)
frame
```

	item	color	price
0	ball	white	5.26
1	mug	red	4.20
2	pen	gree	1.30
3	pencil	black	0.56
4	ashtray	yellow	2.75

Data Aggregation

- Last stage of data manipulation.
- Involves a transformation that produce a single integer from an array.
- Pandas provide a flexible tool for data aggregation: `GroupBy`
- The process of GroupBy is divided into various phases:

  - ❖`Splitting` – Division into groups
  - ❖`Applying` – Application of a function on each group
  - ❖`Combining` – Combination of all the results obtained by different groups



Data Aggregation – Grouping to a single column of data

```
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]})
```

Define a dataframe containing numeric and string values

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

```
<pandas.core.groupby.groupby.SeriesGroupBy object at 0x000002084C71B240>
```

Access the price1 column and call the groupby() function with the color

```
group.groups
```

```
{'green': Int64Index([2, 4], dtype='int64'),
 'red': Int64Index([1, 3], dtype='int64'),
 'white': Int64Index([0], dtype='int64')}
```

Dataframe divided into groups of rows

```
group.mean()
```

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

mean of groups in price1

```
group.sum()
```

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

Sum of groups in price1

Data Aggregation – Hierarchical Grouping – using various columns

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

```
ggroup.groups
```

```
{('green', 'pen'): Int64Index([4], dtype='int64'),
 ('green', 'pencil'): Int64Index([2], dtype='int64'),
 ('red', 'ashtray'): Int64Index([3], dtype='int64'),
 ('red', 'pencil'): Int64Index([1], dtype='int64'),
 ('white', 'pen'): Int64Index([0], dtype='int64')}
```

```
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
```

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

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

```
frame.groupby(frame['color']).mean()
```

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