pandas

November 2, 2019

1 Python for Data Science

`pandas`

2 Pandas

- Etymology: **pan**el **da**ta
- Written by Wes McKinney
- DataFrames for Python
- Read/Write for many formats
- Very efficient operations (much more efficient than plain python)
- Database-like API
- Very popular amongst Data Scientists

2.1 Pandas - Why Python is booming

Stackoverflow Blog: Why is Python Growing So Quickly?

3 Pandas Data Structures

- `Series`: one-dimensional array of indexed data (*think: column of a table/database*)
- `DataFrame`: table (*think: data base*)

4 The Pandas Series Object

Pandas `Series`: one-dimensional array of indexed data

[1]:
```
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings("ignore", message="numpy.dtype size changed")
warnings.filterwarnings("ignore", message="numpy.ufunc size changed")
```

```
# create a pandas Series
series_a = pd.Series([0.25, 0.5, 0.75, 1.0])
series_a
```

[1]:
```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

[2]:
```
series_a.index
```

[2]:
```
RangeIndex(start=0, stop=4, step=1)
```

[3]:
```
series_a.values
```

[3]:
```
array([0.25, 0.5 , 0.75, 1.  ])
```

4.1 Generating Pandas Series

Pandas `Series` can be created from most python collections.

This also means that they support all content types that python collections support.

[4]:
```
# a list
list_a = ['one','two','three']

series_b = pd.Series(list_a)
series_b
```

[4]:
```
0      one
1      two
2    three
dtype: object
```

[5]:
```
# a list with indices
list_a = ['one','two','three']
list_b = ['index_one','index_two','index_three']

series_c = pd.Series(data=list_a, index=list_b)
series_c
```

[5]:
```
index_one        one
index_two        two
index_three    three
dtype: object
```

[6]:
```
# creating Series with same value
pd.Series(5, index=[100, 200, 300])
```

[6]:
```
100    5
200    5
300    5
dtype: int64
```

4.2 Why is this helpful?

There are many reasons why these indexed structures are helpful. Most of them are related to speed. But many are related to convenience:

[7]:
```
fruits = pd.Series([1,0,2,2], index=['apples','oranges','bananas','lemons'])
more_fruits = pd.Series([1,0,1,5],␣
 ↪index=['lemons','oranges','apples','bananas'])

fruits + more_fruits
```

[7]:
```
apples     2
bananas    7
lemons     3
oranges    0
dtype: int64
```

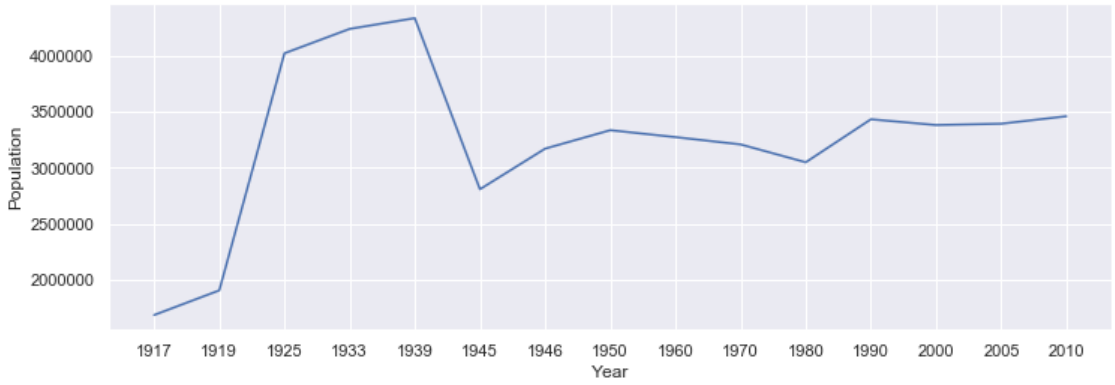
4.3 Some More Examples

[8]:
```
berlin_population_dict = {
 '1917': 1681916,
 '1919': 1902509,
 '1925': 4024286,
 '1933': 4242501,
 '1939': 4338756,
 '1945': 2807405,
 '1946': 3170832,
 '1950': 3336026,
 '1960': 3274016,
 '1970': 3208719,
 '1980': 3048759,
 '1990': 3433695,
 '2000': 3382169,
 '2005': 3394000,
 '2010': 3460725}

population = pd.Series(berlin_population_dict)
```

[9]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()  # set plot style
plt.figure(figsize=[12,4])
plt.plot(population)
plt.ylabel("Population")
plt.xlabel("Year")
```

[9]:
```
Text(0.5, 0, 'Year')
```



4.4 Indexing Pandas Series

[10]:
```
population.name = 'Population'
population.index.name = 'Year'
# note that the indices are strings
population
```

[10]:
```
Year
1917    1681916
1919    1902509
1925    4024286
1933    4242501
1939    4338756
1945    2807405
1946    3170832
1950    3336026
1960    3274016
1970    3208719
1980    3048759
1990    3433695
2000    3382169
2005    3394000
```

```
2010    3460725
Name: Population, dtype: int64
```

[11]:
```
population['1917']
```

[11]:
```
1681916
```

[12]:
```
population[['1917','1925','1945','2010']]
```

[12]:
```
Year
1917    1681916
1925    4024286
1945    2807405
2010    3460725
Name: Population, dtype: int64
```

[13]:
```
population['1925':'1950']
```

[13]:
```
Year
1925    4024286
1933    4242501
1939    4338756
1945    2807405
1946    3170832
1950    3336026
Name: Population, dtype: int64
```

[14]:
```
population > 3e6
```

[14]:
```
Year
1917    False
1919    False
1925     True
1933     True
1939     True
1945    False
1946     True
1950     True
1960     True
1970     True
1980     True
1990     True
2000     True
2005     True
2010     True
Name: Population, dtype: bool
```

[15]:
```
population[population > 3e6]
```

[15]:
```
Year
1925    4024286
1933    4242501
1939    4338756
1946    3170832
1950    3336026
1960    3274016
1970    3208719
1980    3048759
1990    3433695
2000    3382169
2005    3394000
2010    3460725
Name: Population, dtype: int64
```

[16]:
```
population_ = population.copy() # copy is important here
population_[population_ > 3e6] = "more than three million"
population_
```

[16]:
```
Year
1917                    1681916
1919                    1902509
1925    more than three million
1933    more than three million
1939    more than three million
1945                    2807405
1946    more than three million
1950    more than three million
1960    more than three million
1970    more than three million
1980    more than three million
1990    more than three million
2000    more than three million
2005    more than three million
2010    more than three million
Name: Population, dtype: object
```

[17]:
```
population[population==1681916]
```

[17]:
```
Year
1917    1681916
Name: Population, dtype: int64
```

[18]:
```
population[(population==1681916) | (population==3460725)]
```

[18]:
```
Year
1917    1681916
2010    3460725
Name: Population, dtype: int64
```

4.5 Explicit and Positional Indexing

- Pandas objects can be indexed in different ways:
- explicit indices: what you define as index
- positional index: the row number
- Depending on how you access values in a `pandas` object, explicit or positional indexing is used.
- To avoid confusion, specify the type of indexing with `loc` (explicit indexing) or `iloc` (positional indexing)

[19]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

[19]:
```
1    a
3    b
5    c
dtype: object
```

[20]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
# explicit index when indexing
data[1]
```

[20]:
```
'a'
```

[21]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
# explicit index when fancy-indexing
data[[1,3]]
```

[21]:
```
1    a
3    b
dtype: object
```

[22]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
# positional index when slicing
data[1:3]
```

[22]:
```
3    b
5    c
dtype: object
```

[23]:
```
data = pd.Series(['a', 'b', 'c'], index=['1', '3', '5'])
# positional index when actual index is not integer valued
data[1]
```

[23]:
```
'b'
```

4.5.1 Explicit Indexing with `loc`

[24]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data.loc[1]
```

[24]:
```
'a'
```

[25]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
#  loc for explicit indexing
data.loc[1:3]
```

[25]:
```
1    a
3    b
dtype: object
```

4.5.2 Positional Indexing with `iloc`

[26]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data.iloc[1]
```

[26]:
```
'b'
```

[27]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data.iloc[1:3]
```

[27]:
```
3    b
5    c
dtype: object
```

4.6 Operations on Pandas Series

- Efficient arithmetic and aggregation operations
- Compatible with `numpy`

[28]:
```
population_in_millions = population / 1e6
population_in_millions
```

[28]:
```
Year
1917    1.681916
1919    1.902509
1925    4.024286
1933    4.242501
1939    4.338756
1945    2.807405
1946    3.170832
1950    3.336026
1960    3.274016
1970    3.208719
1980    3.048759
1990    3.433695
2000    3.382169
2005    3.394000
2010    3.460725
Name: Population, dtype: float64
```

[29]:
```
# histograms
population_in_millions.value_counts()
```

[29]:
```
4.338756    1
3.433695    1
3.170832    1
3.208719    1
4.242501    1
3.048759    1
3.274016    1
1.681916    1
3.382169    1
2.807405    1
3.460725    1
4.024286    1
3.394000    1
1.902509    1
3.336026    1
Name: Population, dtype: int64
```

[30]:
```
# sorting
population.sort_values()
```

[30]:
```
Year
1917    1681916
1919    1902509
1945    2807405
1980    3048759
1946    3170832
```

```
1970    3208719
1960    3274016
1950    3336026
2000    3382169
2005    3394000
1990    3433695
2010    3460725
1925    4024286
1933    4242501
1939    4338756
Name: Population, dtype: int64
```

[31]:
```
# basic stats
population.describe()
```

[31]:
```
count    1.500000e+01
mean     3.247088e+06
std      7.276748e+05
min      1.681916e+06
25%      3.109796e+06
50%      3.336026e+06
75%      3.447210e+06
max      4.338756e+06
Name: Population, dtype: float64
```

4.7 Applying Arbitrary Functions

[32]:
```
# apply custom functions

def some_function(v):
    '''
    Divides number by a million and returns its integer representation
    '''
    return int(v / 1e6)

some_function(35 * 1e6)
```

[32]:
```
35
```

[33]:
```
population.apply(some_function)
```

[33]:
```
Year
1917    1
1919    1
1925    4
1933    4
```

```
1939    4
1945    2
1946    3
1950    3
1960    3
1970    3
1980    3
1990    3
2000    3
2005    3
2010    3
Name: Population, dtype: int64
```

[34]:
```
s = pd.Series(range(int(1e6)))
%timeit s.apply(some_function)
```

```
370 ms ± 5.05 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[35]:
```
%timeit (s / 1e6).astype(int)
```

```
3.72 ms ± 13.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

5 Missing Values

There are three main options how to deal with missing values:

- drop rows with missing values
- replace missing values with placeholder symbol
- impute missing values with some ML model

[36]:
```
berlin_population_dict = {
 '1945': 2807405,
 '1950': 3336026,
 '1955': None,
 '1960': 3274016,
 '1965': np.nan,
 '1970': 3208719}

population_w_nans = pd.Series(berlin_population_dict)
population_w_nans
```

[36]:
```
1945    2807405.0
1950    3336026.0
1955          NaN
1960    3274016.0
1965          NaN
1970    3208719.0
```

```
dtype: float64
```

5.1 Dropping rows

[37]:
```
population_w_nans.isnull()
```

[37]:
```
1945    False
1950    False
1955     True
1960    False
1965     True
1970    False
dtype: bool
```

[38]:
```
population_w_nans[~population_w_nans.isnull()]
```

[38]:
```
1945    2807405.0
1950    3336026.0
1960    3274016.0
1970    3208719.0
dtype: float64
```

[39]:
```
population_w_nans.dropna()
```

[39]:
```
1945    2807405.0
1950    3336026.0
1960    3274016.0
1970    3208719.0
dtype: float64
```

5.2 Filling with Placeholder

[40]:
```
population_w_nans.fillna(method='ffill')
```

[40]:
```
1945    2807405.0
1950    3336026.0
1955    3336026.0
1960    3274016.0
1965    3274016.0
1970    3208719.0
dtype: float64
```

[41]:
```
population_w_nans.fillna(value=population_w_nans.median())
```

[41]:
```
1945    2807405.0
1950    3336026.0
1955    3241367.5
1960    3274016.0
1965    3241367.5
1970    3208719.0
dtype: float64
```

6 The Pandas DataFrame Object

Pandas `Dataframe`: **a table** (or DataBase - i.e. one or more Series concatenated)

6.1 Generating Pandas DataFrames

[42]:
```
some_numpy_array = np.arange(9).reshape((3, 3))
some_numpy_array
```

[42]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

[43]:
```
df = pd.DataFrame(some_numpy_array,
                  index=['a', 'c', 'd'],
                  columns=['Ohio', 'Texas', 'California'])
df
```

[43]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

6.1.1 Pandas DataFrame Constructors

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record array	Treated as the “dict of arrays” case
dict of Series	Each value becomes a column
dict of dicts	Each inner dict becomes a column
List of dicts or Series	Each item becomes a row in the DataFrame

Type	Notes
List of lists or tuples	Treated as the “2D ndarray” case
Another DataFrame	The DataFrame’s indexes are used unless different ones are passed
NumPy MaskedArray	Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result

[44]:
```
# a DataFrame from a csv file
import os
df_from_csv = pd.read_csv(os.path.join("data","berlin_population.csv"))
```

6.1.2 Pandas DataFrame IO

,

Format Type

```
  <th>Data Description</th>      <th>Reader</th>      <th>Writer</th>    </tr>  </thead>  <tbo
```

6.1.3 Example: Getting Berlin Population Statistics from Wikipedia

see “https://en.wikipedia.org/wiki/Berlin_population_statistics”

[45]:
```
df_berlin_population = pd.read_html(
    "https://en.wikipedia.org/wiki/Berlin_population_statistics",
    header=0)

df_berlin_population[0]
```

[45]:

	Borough	Population 30 September 2010	Area in km²	\
0	Mitte	332100	39.47	
1	Friedrichshain-Kreuzberg	268831	20.16	
2	Pankow	368956	103.01	
3	Charlottenburg-Wilmersdorf	320014	64.72	
4	Spandau	225420	91.91	
5	Steglitz-Zehlendorf	293989	102.50	
6	Tempelhof-Schöneberg	335060	53.09	
7	Neukölln	310283	44.93	
8	Treptow-Köpenick	241335	168.42	
9	Marzahn-Hellersdorf	248264	61.74	
10	Lichtenberg	259881	52.29	
11	Reinickendorf	240454	89.46	
12	Total Berlin	3450889	891.82	

Largest Non-German ethnic groups

```
0   Turks, Arabs, Kurds, many Asians, Africans and…
1               Turks, Arabs, African, Kurds, Chinese
2   Poles, Italians, French, Americans, Vietnamese…
3           Turks, Africans, Russians, Arabs, others.
4           Turks, Africans, Russians, Arabs, others.
5                Poles, Turks, Croats, Serbs, Koreans
6             Turks, Croats, Serbs, Koreans, Africans
7      Arabs, Turks, Kurds, Russians, Africans, Poles
8             Russians, Poles, Ukrainians, Vietnamese
9   Russians, Vietnamese, several other Eastern Eu…
10   Vietnamese, Russians, Ukrainians, Poles, Chinese
11       Turks, Poles, Serbs, Croats, Arabs, Italians
12  Turks, Arabs, Russians, Vietnamese, Poles, Afr…
```

[46]:
```
df_berlin_population[0].copy()[:11].set_index('Borough')
```

[46]:

Borough	Population 30 September 2010	Area in km²	Largest Non-German ethnic groups
Mitte	332100	39.47	Turks, Arabs, Kurds, many Asians, Africans and…
Friedrichshain-Kreuzberg	268831	20.16	Turks, Arabs, African, Kurds, Chinese
Pankow	368956	103.01	Poles, Italians, French, Americans, Vietnamese…
Charlottenburg-Wilmersdorf	320014	64.72	Turks, Africans, Russians, Arabs, others.
Spandau	225420	91.91	Turks, Africans, Russians, Arabs, others.
Steglitz-Zehlendorf	293989	102.50	Poles, Turks, Croats, Serbs, Koreans
Tempelhof-Schöneberg	335060	53.09	Turks, Croats, Serbs, Koreans, Africans
Neukölln	310283	44.93	Arabs, Turks, Kurds, Russians, Africans, Poles
Treptow-Köpenick	241335	168.42	Russians, Poles, Ukrainians, Vietnamese
Marzahn-Hellersdorf	248264	61.74	Russians, Vietnamese, several other Eastern Eu…
Lichtenberg	259881	52.29	Vietnamese, Russians, Ukrainians, Poles, Chinese

[47]:
```
df_population_over_time = df_berlin_population[4]
df_population_over_time
```

[47]:

	Year	Population
0	December 5, 1917 [1]	1681916
1	October 8, 1919 [1]	1902509
2	June 16, 1925 [1]	4024286
3	June 16, 1933 [1]	4242501
4	May 17, 1939 [1]	4338756
5	August 12, 1945 [1]	2807405
6	October 29, 1946 [1]	3170832
7	December 31, 1950	3336026
8	December 31, 1960	3274016
9	December 31, 1970	3208719
10	December 31, 1980	3048759
11	December 31, 1990	3433695
12	December 31, 2000	3382169
13	September 30, 2005	3394000
14	September 30, 2010	3450889
15	December 31, 2010	3460725

[48]:
```
df_population_over_time.Year = df_population_over_time.Year\
                                .str.replace(".*, ","")
df_population_over_time
```

[48]:

	Year	Population
0	1917 [1]	1681916
1	1919 [1]	1902509
2	1925 [1]	4024286
3	1933 [1]	4242501
4	1939 [1]	4338756
5	1945 [1]	2807405
6	1946 [1]	3170832
7	1950	3336026
8	1960	3274016
9	1970	3208719
10	1980	3048759
11	1990	3433695
12	2000	3382169
13	2005	3394000
14	2010	3450889
15	2010	3460725

6.2 Accessing Values in Pandas Data Frames

[49]:
```
# some data
berlin_population_by_borough = {
  'Area in km²': {
    'Charlottenburg-Wilmersdorf': 64.72,
```

```
    'Friedrichshain-Kreuzberg': 20.16,
    'Lichtenberg': 52.29,
    'Marzahn-Hellersdorf': 61.74,
    'Mitte': 39.47,
    'Neukölln': 44.93,
    'Pankow': 103.01,
    'Spandau': 91.91,
    'Steglitz-Zehlendorf': 102.5,
    'Tempelhof-Schöneberg': 53.09,
    'Treptow-Köpenick': 168.42},
 'Population 30 September 2010': {
    'Charlottenburg-Wilmersdorf': 320014,
    'Friedrichshain-Kreuzberg': 268831,
    'Lichtenberg': 259881,
    'Marzahn-Hellersdorf': 248264,
    'Mitte': 332100,
    'Neukölln': 310283,
    'Pankow': 368956,
    'Spandau': 225420,
    'Steglitz-Zehlendorf': 293989,
    'Tempelhof-Schöneberg': 335060,
    'Treptow-Köpenick': 241335}}
```

[50]:
```
# a DataFrame from a dictionary of dictionaries
df = pd.DataFrame(berlin_population_by_borough)
df
```

[50]:

	Area in km²	Population 30 September 2010
Charlottenburg-Wilmersdorf	64.72	320014
Friedrichshain-Kreuzberg	20.16	268831
Lichtenberg	52.29	259881
Marzahn-Hellersdorf	61.74	248264
Mitte	39.47	332100
Neukölln	44.93	310283
Pankow	103.01	368956
Spandau	91.91	225420
Steglitz-Zehlendorf	102.50	293989
Tempelhof-Schöneberg	53.09	335060
Treptow-Köpenick	168.42	241335

[51]:
```
# like Series (and tables in DBs) DataFrames have indices
df.index
```

[51]:
```
Index(['Charlottenburg-Wilmersdorf', 'Friedrichshain-Kreuzberg', 'Lichtenberg',
       'Marzahn-Hellersdorf', 'Mitte', 'Neukölln', 'Pankow', 'Spandau',
       'Steglitz-Zehlendorf', 'Tempelhof-Schöneberg', 'Treptow-Köpenick'],
      dtype='object')
```

[52]:
```
# Accessing by row and column index
df.loc['Friedrichshain-Kreuzberg','Area in km²']
```

[52]:
```
20.16
```

[53]:
```
# Accessing an entire column
df.loc[:,'Area in km²']
```

[53]:
```
Charlottenburg-Wilmersdorf     64.72
Friedrichshain-Kreuzberg       20.16
Lichtenberg                    52.29
Marzahn-Hellersdorf            61.74
Mitte                          39.47
Neukölln                       44.93
Pankow                        103.01
Spandau                        91.91
Steglitz-Zehlendorf           102.50
Tempelhof-Schöneberg           53.09
Treptow-Köpenick              168.42
Name: Area in km², dtype: float64
```

[54]:
```
# Accessing an entire column
df['Area in km²']
```

[54]:
```
Charlottenburg-Wilmersdorf     64.72
Friedrichshain-Kreuzberg       20.16
Lichtenberg                    52.29
Marzahn-Hellersdorf            61.74
Mitte                          39.47
Neukölln                       44.93
Pankow                        103.01
Spandau                        91.91
Steglitz-Zehlendorf           102.50
Tempelhof-Schöneberg           53.09
Treptow-Köpenick              168.42
Name: Area in km², dtype: float64
```

[55]:
```
[b for b in df.index if 'berg' in b.lower()]
```

[55]:
```
['Friedrichshain-Kreuzberg', 'Lichtenberg', 'Tempelhof-Schöneberg']
```

[56]:
```
df.loc[['Friedrichshain-Kreuzberg', 'Lichtenberg', 'Tempelhof-Schöneberg'],:]
```

[56]:

	Area in km²	Population 30 September 2010
Friedrichshain-Kreuzberg	20.16	268831
Lichtenberg	52.29	259881
Tempelhof-Schöneberg	53.09	335060

[57]:
```
# a single column of a DataFrame is a Series
df['Population 30 September 2010']
```

[57]:
```
Charlottenburg-Wilmersdorf    320014
Friedrichshain-Kreuzberg      268831
Lichtenberg                   259881
Marzahn-Hellersdorf           248264
Mitte                         332100
Neukölln                      310283
Pankow                        368956
Spandau                       225420
Steglitz-Zehlendorf           293989
Tempelhof-Schöneberg          335060
Treptow-Köpenick              241335
Name: Population 30 September 2010, dtype: int64
```

[58]:
```
# boolean indexing
df['Population 30 September 2010'] > 3e5
```

[58]:
```
Charlottenburg-Wilmersdorf     True
Friedrichshain-Kreuzberg      False
Lichtenberg                   False
Marzahn-Hellersdorf           False
Mitte                          True
Neukölln                       True
Pankow                         True
Spandau                       False
Steglitz-Zehlendorf           False
Tempelhof-Schöneberg           True
Treptow-Köpenick              False
Name: Population 30 September 2010, dtype: bool
```

[59]:
```
# boolean indexing
df[df['Population 30 September 2010'] > 3e5]
```

[59]:

	Area in km²	Population 30 September 2010
Charlottenburg-Wilmersdorf	64.72	320014
Mitte	39.47	332100
Neukölln	44.93	310283
Pankow	103.01	368956
Tempelhof-Schöneberg	53.09	335060

[60]:
```
# boolean row indexing with column indexing
df.loc[df['Population 30 September 2010'] > 3e5,'Population 30 September 2010']
```

[60]:
```
Charlottenburg-Wilmersdorf    320014
Mitte                         332100
```

```
Neukölln                      310283
Pankow                        368956
Tempelhof-Schöneberg          335060
Name: Population 30 September 2010, dtype: int64
```

6.3 Operations on Pandas Data Frames

- All Series operations work on DataFrame columns
- DataFrames support all standard DB operations and more

[61]:
```
df['Density'] = df['Population 30 September 2010'] / df['Area in km²']
df = df.sort_values(by=['Density'])
df
```

[61]:

	Area in km²	Population 30 September 2010	\
Treptow-Köpenick	168.42	241335	
Spandau	91.91	225420	
Steglitz-Zehlendorf	102.50	293989	
Pankow	103.01	368956	
Marzahn-Hellersdorf	61.74	248264	
Charlottenburg-Wilmersdorf	64.72	320014	
Lichtenberg	52.29	259881	
Tempelhof-Schöneberg	53.09	335060	
Neukölln	44.93	310283	
Mitte	39.47	332100	
Friedrichshain-Kreuzberg	20.16	268831	

	Density
Treptow-Köpenick	1432.935518
Spandau	2452.616690
Steglitz-Zehlendorf	2868.185366
Pankow	3581.749345
Marzahn-Hellersdorf	4021.120829
Charlottenburg-Wilmersdorf	4944.592089
Lichtenberg	4969.994263
Tempelhof-Schöneberg	6311.169712
Neukölln	6905.920320
Mitte	8413.985305
Friedrichshain-Kreuzberg	13334.871032

6.4 Database style joins with pandas

[62]:
```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1':␣
↪range(7)})
df1
```

[62]:
```
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   a      5
6   b      6
```

[63]:
```
df2 = pd.DataFrame({'key': ['a', 'b', 'd'], 'data2': range(3)})
df2
```

[63]:
```
  key  data2
0   a      0
1   b      1
2   d      2
```

6.4.1 Inner join

[64]:
```
pd.merge(df1, df2, on='key')
```

[64]:
```
  key  data1  data2
0   b      0      1
1   b      1      1
2   b      6      1
3   a      2      0
4   a      4      0
5   a      5      0
```

6.4.2 Outer join

[65]:
```
pd.merge(df1, df2, on='key', how='outer')
```

[65]:
```
  key  data1  data2
0   b    0.0    1.0
1   b    1.0    1.0
2   b    6.0    1.0
3   a    2.0    0.0
```

```
4   a    4.0    0.0
5   a    5.0    0.0
6   c    3.0    NaN
7   d    NaN    2.0
```

6.5 Concatenation

Remember numpy array concatenation

[66]:
```
import numpy as np
arr = np.arange(12).reshape((3, 4))
arr
```

[66]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[67]:
```
np.concatenate([arr, arr], axis=1)
```

[67]:
```
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

[68]:
```
np.concatenate([arr, arr], axis=0)
```

[68]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

6.6 Concatenation with Pandas

[69]:
```
s1 = pd.Series([0, 1], index=['a', 'b'])
s1
```

[69]:
```
a    0
b    1
dtype: int64
```

[70]:
```
s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
s2
```

[70]:
```
c    2
d    3
e    4
dtype: int64
```

[71]:
```
s3 = pd.Series([5, 6], index=['f', 'g'])
pd.concat([s1, s2, s3])
```

[71]:
```
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

[72]:
```
pd.concat([s1, s2, s3], axis=1, sort=True)
```

[72]:

	0	1	2
a	0.0	NaN	NaN
b	1.0	NaN	NaN
c	NaN	2.0	NaN
d	NaN	3.0	NaN
e	NaN	4.0	NaN
f	NaN	NaN	5.0
g	NaN	NaN	6.0

6.7 Group-by and Aggregations

Aka split-apply-combine

[73]:
```
df = pd.DataFrame({ 'key1' : ['a', 'a', 'b', 'b', 'a'],
                    'key2' : ['one', 'two', 'one', 'two', 'one'],
                    'data1' : np.random.randn(5),
                    'data2' : np.random.randn(5)})
df
```

[73]:

	key1	key2	data1	data2
0	a	one	-0.102319	-0.591012
1	a	two	-0.572835	-0.762974
2	b	one	0.046125	-0.283270
3	b	two	-1.270324	-3.176505
4	a	one	0.414838	-0.259414

[74]:
```
grouped = df['data1'].groupby(df['key1'])
grouped
```

[74]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x1a1d1dac10>
```

[75]:
```
grouped.mean()
```

[75]:
```
key1
a   -0.086772
b   -0.612100
Name: data1, dtype: float64
```

[76]:
```
grouped.max()
```

[76]:
```
key1
a    0.414838
b    0.046125
Name: data1, dtype: float64
```

[77]:
```
grouped.quantile(.9)
```

[77]:
```
key1
a    0.311407
b   -0.085520
Name: data1, dtype: float64
```

[78]:
```
df.groupby(['key1','key2'])['data1'].agg(['mean','sum']).rename(columns={'mean':
 ↪ 'my_mean', 'sum':'my_sum'})
```

[78]:
```
             my_mean    my_sum
key1 key2
a    one    0.156260  0.312519
     two   -0.572835 -0.572835
b    one    0.046125  0.046125
     two   -1.270324 -1.270324
```

6.7.1 Iterating over groups

[79]:
```
for name, group in df.groupby('key1'):
    print(name)
    print(group)
```

```
a
  key1 key2     data1     data2
0    a  one -0.102319 -0.591012
```

```
1    a  two -0.572835 -0.762974
4    a  one  0.414838 -0.259414
b
  key1 key2     data1     data2
2    b  one  0.046125 -0.283270
3    b  two -1.270324 -3.176505
```

[80]:
```
for (k1, k2), group in df.groupby(['key1', 'key2']):
    print((k1, k2))
    print(group)
```

```
('a', 'one')
  key1 key2     data1     data2
0    a  one -0.102319 -0.591012
4    a  one  0.414838 -0.259414
('a', 'two')
  key1 key2     data1     data2
1    a  two -0.572835 -0.762974
('b', 'one')
  key1 key2     data1    data2
2    b  one  0.046125 -0.28327
('b', 'two')
  key1 key2     data1     data2
3    b  two -1.270324 -3.176505
```

7 Some Experiments with Names in Berlin

Source data portal Berlin

[81]:
```
import urllib
import os

basedir = os.path.join("data","vornamen")
os.makedirs(basedir, exist_ok=True)

base_url = "https://raw.githubusercontent.com/berlinonline/
 ↪haeufige-vornamen-berlin/master/data/cleaned/{}/{}.csv"

boroughs = [
"charlottenburg-wilmersdorf",
"friedrichshain-kreuzberg",
"lichtenberg",
"marzahn-hellersdorf",
"mitte",
"neukoelln",
```

```
"pankow",
"reinickendorf",
"spandau",
"steglitz-zehlendorf",
"tempelhof-schoeneberg",
"treptow-koepenick"
]

years = range(2012,2019)
```

[82]:
```
# download all name files from Berlin open data portal
all_names = []

for borough in boroughs:
    for year in years:
        url = base_url.format(year, borough)
        filename = os.path.join(basedir, "{}-{}.csv".format(year,borough))
        urllib.request.urlretrieve(url, filename)
        df_vornamen_stadtteil = pd.
 ↪read_csv(filename,sep=',',error_bad_lines=False)
        df_vornamen_stadtteil['borough'] = borough
        df_vornamen_stadtteil['year'] = year
        all_names.append(df_vornamen_stadtteil)

# concatenate DataFrames
all_names_df = pd.concat(all_names, sort=True)
```

[83]:
```
all_names_df.sample(n=10)
```

[83]:

	anzahl	borough	geschlecht	position	vorname	year
177	5	neukoelln	w	NaN	Amina	2016
1334	1	lichtenberg	m	NaN	José	2014
552	3	charlottenburg-wilmersdorf	m	NaN	Ezra	2015
1664	1	friedrichshain-kreuzberg	m	NaN	Huzayer	2013
43	15	neukoelln	m	NaN	Jonathan	2016
2882	1	charlottenburg-wilmersdorf	w	1.0	Annelotte	2018
1156	1	charlottenburg-wilmersdorf	m	NaN	Andrés	2012
223	5	spandau	m	NaN	Franz	2015
2655	1	charlottenburg-wilmersdorf	w	NaN	Nursevim	2013
1053	1	reinickendorf	w	NaN	Sofie	2012

[84]:
```
# names for boys in friedrichshain in 2016 sorted by popularity
all_names_df.loc[
    (all_names_df['borough']=="friedrichshain-kreuzberg")
    & (all_names_df['geschlecht']=='m')
    & (all_names_df['year']==2016)].sort_values(by='anzahl', ascending=False)
```

[84]:

	anzahl	borough	geschlecht	position	vorname	year
3	42	friedrichshain-kreuzberg	m	NaN	Anton	2016
6	41	friedrichshain-kreuzberg	m	NaN	Emil	2016
7	40	friedrichshain-kreuzberg	m	NaN	Ali	2016
8	39	friedrichshain-kreuzberg	m	NaN	Alexander	2016
10	37	friedrichshain-kreuzberg	m	NaN	Leon	2016
…	…	…	…	…	…	…
1964	1	friedrichshain-kreuzberg	m	NaN	Hamallah	2016
1962	1	friedrichshain-kreuzberg	m	NaN	Hakî	2016
1960	1	friedrichshain-kreuzberg	m	NaN	Haika	2016
1958	1	friedrichshain-kreuzberg	m	NaN	Hagen	2016
3518	1	friedrichshain-kreuzberg	m	NaN	Şükrü	2016

[1805 rows x 6 columns]

[85]:
```
# least popular names per year and borough
all_names_df.groupby(['borough','year'],as_index=False).\
    agg({"anzahl": "min",'vorname':'last'})
```

[85]:

	borough	year	anzahl	vorname
0	charlottenburg-wilmersdorf	2012	1	Şeniz
1	charlottenburg-wilmersdorf	2013	1	Đželila
2	charlottenburg-wilmersdorf	2014	1	Şerife
3	charlottenburg-wilmersdorf	2015	1	Şevket
4	charlottenburg-wilmersdorf	2016	1	Şirin
..	…	…	…	…
79	treptow-koepenick	2014	1	Łucja
80	treptow-koepenick	2015	1	Yorin
81	treptow-koepenick	2016	1	Đa
82	treptow-koepenick	2017	1	Şifa
83	treptow-koepenick	2018	1	Đuć

[84 rows x 4 columns]