pandas

November 14, 2018

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

```
In [2]: 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
```

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

In [4]: series_a.index

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

In [5]: series_a.values

Out[5]: 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.

```
In [6]: # a list
        list_a = ['one','two','three']

        series_b = pd.Series(list_a)
        series_b

Out[6]: 0      one
        1      two
        2    three
        dtype: object

In [7]: # 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

Out[7]: index_one        one
        index_two        two
        index_three    three
        dtype: object

In [8]: # creating Series with same value
        pd.Series(5, index=[100, 200, 300])

Out[8]: 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:

```
In [12]: 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
```

```
Out[12]: apples     2
         bananas    7
         lemons     3
         oranges    0
         dtype: int64
```

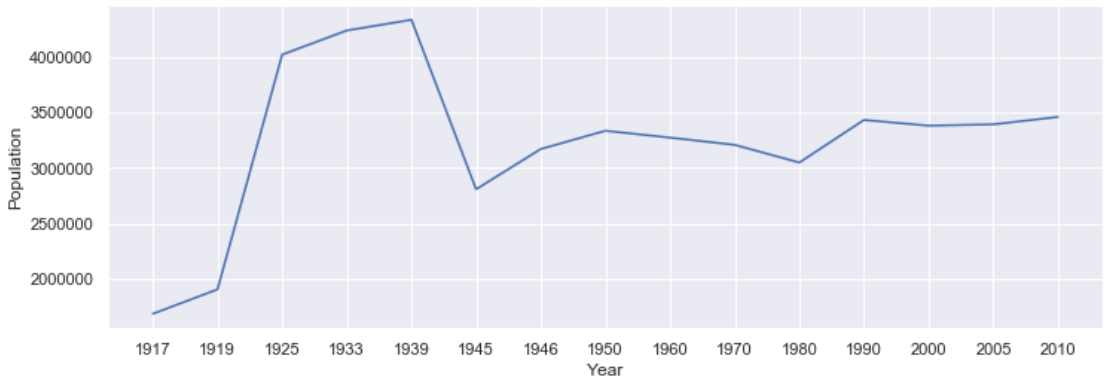
4.3 Some More Examples

```
In [32]: 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)
```

```
In [38]: %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")
```

```
Out[38]: Text(0.5,0,'Year')
```



4.4 Indexing Pandas Series

```
In [30]: population.name = 'Population'
         population.index.name = 'Year'
         # note that the indices are strings
         population

Out[30]: 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

In [16]: population['1917']

Out[16]: 1681916

In [17]: population[['1917','1925','1945','2010']]

Out[17]: Year
         1917    1681916
         1925    4024286
```

```
1945    2807405
2010    3460725
Name: Population, dtype: int64
```

```
In [18]: population['1925':'1950']
```

```
Out[18]: Year
1925    4024286
1933    4242501
1939    4338756
1945    2807405
1946    3170832
1950    3336026
Name: Population, dtype: int64
```

```
In [19]: population > 3e6
```

```
Out[19]: 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
```

```
In [20]: population[population > 3e6]
```

```
Out[20]: 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
```

```
In [21]: population_ = population.copy() # copy is important here
         population_[population_ > 3e6] = "more than three million"
         population_
```

```
Out[21]: 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
```

```
In [22]: population[population==1681916]
```

```
Out[22]: Year
         1917    1681916
         Name: Population, dtype: int64
```

```
In [23]: population[(population==1681916) | (population==3460725)]
```

```
Out[23]: 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)

```
In [17]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         data
```

```
Out[17]: 1    a
         3    b
         5    c
         dtype: object

In [18]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         # explicit index when indexing
         data[1]

Out[18]: 'a'

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

Out[5]: 1    a
        3    b
        dtype: object

In [19]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         # positional index when slicing
         data[1:3]

Out[19]: 3    b
         5    c
         dtype: object

In [31]: data = pd.Series(['a', 'b', 'c'], index=['1', '3', '5'])
         # positional index when actual index is not integer valued
         data[1]

Out[31]: 'b'
```

4.5.1 Explicit Indexing with `loc`

```
In [20]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         data.loc[1]

Out[20]: 'a'

In [21]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         #  loc for explicit indexing
         data.loc[1:3]

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

4.5.2 Positional Indexing with `iloc`

```
In [22]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         data.iloc[1]
```

```
Out[22]: 'b'
```

```
In [23]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         data.iloc[1:3]
```

```
Out[23]: 3    b
         5    c
         dtype: object
```

4.6 Operations on Pandas Series

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

```
In [24]: population_in_millions = population / 1e6
         population_in_millions
```

```
Out[24]: 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
```

```
In [27]: # histograms
         population_in_millions.value_counts()
```

```
Out[27]: 3    9
         4    3
         1    2
         2    1
         Name: Population, dtype: int64
```

```
In [28]: # sorting
         population.sort_values()
```

```
Out[28]: 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
```

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

```
Out[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

```
In [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)
```

```
Out[32]: 35
```

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

```
Out[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
```

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

```
1 loop, best of 3: 926 ms per loop
```

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

```
The slowest run took 10.75 times longer than the fastest. This could mean that an intermediate r
100 loops, best of 3: 10.1 ms per loop
```

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

```
In [37]: 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
```

```
Out[37]: 1945    2807405.0
         1950    3336026.0
```

```
1955          NaN
1960    3274016.0
1965          NaN
1970    3208719.0
dtype: float64
```

5.1 Dropping rows

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

Out[38]: 1945    False
         1950    False
         1955     True
         1960    False
         1965     True
         1970    False
         dtype: bool

In [39]: population_w_nans[~population_w_nans.isnull()]

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

In [40]: population_w_nans.dropna()

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

5.2 Filling with Placeholder

```
In [41]: population_w_nans.fillna(method='ffill')

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

In [42]: population_w_nans.fillna(value=population_w_nans.median())
```

```
Out[42]: 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

```
In [43]: some_numpy_array = np.arange(9).reshape((3, 3))
         some_numpy_array

Out[43]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])

In [44]: df = pd.DataFrame(some_numpy_array,
                           index=['a', 'c', 'd'],
                           columns=['Ohio', 'Texas', 'California'])
         df

Out[44]:    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

Type	Notes
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
List of lists or tuples	Treated as the “2D ndarray” case

Type	Notes
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

```
In [14]: # 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

,

```
<tr style="text-align: right;">      <th>Format Type</th>      <th>Data Description</th>      <t
```

6.2 Accessing Values in Pandas Data Frames

```
In [16]: # some data
         berlin_population_by_borough = {
           'Area in kmš': {
             '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}}
```

In [17]:
```
# a DataFrame from a dictionary of dictionaries
df = pd.DataFrame(berlin_population_by_borough)
df
```

Out[17]:

	Area in kmš	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

In [18]:
```
# like Series (and tables in DBs) DataFrames have indices
df.index
```

Out[18]:
```
Index(['Charlottenburg-Wilmersdorf', 'Friedrichshain-Kreuzberg', 'Lichtenberg',
       'Marzahn-Hellersdorf', 'Mitte', 'Neukölln', 'Pankow', 'Spandau',
       'Steglitz-Zehlendorf', 'Tempelhof-Schöneberg', 'Treptow-Köpenick'],
      dtype='object')
```

In [19]:
```
# Accessing by row and column index
df.loc['Friedrichshain-Kreuzberg','Area in kmš']
```

Out[19]:
```
20.16
```

In [20]:
```
# Accessing an entire column
df.loc[:,'Area in kmš']
```

Out[20]:
```
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 kmš, dtype: float64
```

```
In [21]: # Accessing an entire column
         df['Area in kmš']
```

```
Out[21]: 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 kmš, dtype: float64
```

```
In [22]: [b for b in df.index if 'berg' in b.lower()]
```

```
Out[22]: ['Friedrichshain-Kreuzberg', 'Lichtenberg', 'Tempelhof-Schöneberg']
```

```
In [23]: df.loc[['Friedrichshain-Kreuzberg', 'Lichtenberg', 'Tempelhof-Schöneberg'],:]
```

```
Out[23]:                          Area in kmš  Population 30 September 2010
         Friedrichshain-Kreuzberg       20.16                        268831
         Lichtenberg                    52.29                        259881
         Tempelhof-Schöneberg           53.09                        335060
```

```
In [25]: # a single column of a DataFrame is a Series
         df['Population 30 September 2010']
```

```
Out[25]: 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
```

```
In [26]: # boolean indexing
         df['Population 30 September 2010'] > 3e5
```

```
Out[26]: 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
```

```
In [27]: # boolean indexing
         df[df['Population 30 September 2010'] > 3e5]
```

```
Out[27]:                            Area in kmš  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
```

```
In [28]: # boolean row indexing with column indexing
         df.loc[df['Population 30 September 2010'] > 3e5,'Population 30 September 2010']
```

```
Out[28]: 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

```
In [62]: df['Density'] = df['Population 30 September 2010'] / df['Area in kmš']
         df = df.sort_values(by=['Density'])
         df
```

```
Out[62]:                            Area in kmš  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

```
In [4]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)})
        df1

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

In [5]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'], 'data2': range(3)})
        df2

Out[5]:   key  data2
        0   a      0
        1   b      1
        2   d      2
```

6.4.1 Inner join

```
In [6]: pd.merge(df1, df2, on='key')
```

```
Out[6]:   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

```
In [7]: pd.merge(df1, df2, on='key', how='outer')
```

```
Out[7]:   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

```
In [8]: import numpy as np
        arr = np.arange(12).reshape((3, 4))
        arr
```

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

```
In [9]: np.concatenate([arr, arr], axis=1)
```

```
Out[9]: 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]])
```

```
In [10]: np.concatenate([arr, arr], axis=0)
```

```
Out[10]: 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

```
In [12]: s1 = pd.Series([0, 1], index=['a', 'b'])
         s1

Out[12]: a    0
         b    1
         dtype: int64

In [11]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
         s2

Out[11]: c    2
         d    3
         e    4
         dtype: int64

In [13]: s3 = pd.Series([5, 6], index=['f', 'g'])
         pd.concat([s1, s2, s3])

Out[13]: a    0
         b    1
         c    2
         d    3
         e    4
         f    5
         g    6
         dtype: int64

In [15]: pd.concat([s1, s2, s3], axis=1, sort=True)

Out[15]:      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

```
In [291]: 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
```

```
Out[291]:   key1 key2     data1     data2
          0    a  one  1.502392 -0.788310
          1    a  two -1.166386  0.727721
          2    b  one  0.304301  0.407109
          3    b  two -0.577516  0.121743
          4    a  one  0.550197  0.395730

In [292]: grouped = df['data1'].groupby(df['key1'])
          grouped

Out[292]: <pandas.core.groupby.groupby.SeriesGroupBy object at 0x1a1cba2cf8>

In [293]: grouped.mean()

Out[293]: key1
          a    0.295401
          b   -0.136607
          Name: data1, dtype: float64

In [294]: grouped.max()

Out[294]: key1
          a    1.502392
          b    0.304301
          Name: data1, dtype: float64

In [295]: grouped.quantile(.9)

Out[295]: key1
          a    1.311953
          b    0.216119
          Name: data1, dtype: float64

In [302]: df.groupby(['key1','key2'])['data1'].agg(['mean','sum']).rename(columns={'mean': 'my_m

Out[302]:             my_mean    my_sum
          key1 key2
          a    one   1.026294  2.052589
               two  -1.166386 -1.166386
          b    one   0.304301  0.304301
               two  -0.577516 -0.577516
```

6.7.1 Iterating over groups

```
In [22]: for name, group in df.groupby('key1'):
             print(name)
             print(group)
```

```
a
  key1 key2     data1     data2
0    a  one  1.163201 -2.187141
1    a  two -1.526825  0.310791
4    a  one  0.276677 -1.389537
b
  key1 key2     data1     data2
2    b  one -1.059584  1.204863
3    b  two -2.193576 -0.858221
```

```
In [23]: for (k1, k2), group in df.groupby(['key1', 'key2']):
             print((k1, k2))
             print(group)
```

```
('a', 'one')
  key1 key2     data1     data2
0    a  one  1.163201 -2.187141
4    a  one  0.276677 -1.389537
('a', 'two')
  key1 key2     data1     data2
1    a  two -1.526825  0.310791
('b', 'one')
  key1 key2     data1     data2
2    b  one -1.059584  1.204863
('b', 'two')
  key1 key2     data1     data2
3    b  two -2.193576 -0.858221
```

7 Some Experiments with Names in Berlin

Source data portal Berlin

```
In [63]: import urllib
         import os

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

         base_url = "https://www.berlin.de/daten/liste-der-vornamen-{}/{}.csv"

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

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

        years = range(2013,2018)
```

In [64]:
```
# download all name files from Berlin open data portal
all_names = []

for borough in boroughs:
    for year in years:
        try:
            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)
        except:
            print("File {} not found".format(url))

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

In [65]:
```
all_names_df.sample(n=10)
```

Out[65]:

	anzahl	borough	geschlecht	position	vorname	year
604	1	marzahn-hellersdorf	m	NaN	Laurin	2014
766	1	treptow-koepenick	m	2.0	Otis	2017
633	2	mitte	w	NaN	Annabelle	2016
422	1	steglitz-zehlendorf	m	NaN	Darijan	2015
576	2	neukoelln	m	NaN	Albert	2014
69	12	lichtenberg	w	NaN	Luise	2016
825	1	lichtenberg	w	2.0	NaN	2017
661	2	mitte	w	NaN	Christina	2014
127	8	spandau	m	NaN	Peter	2013
2436	1	neukoelln	m	NaN	Siraç	2014

In [66]:
```
# 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)
```

Out[66]:

	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
12	35	friedrichshain-kreuzberg	m	NaN	Elias	2016
13	33	friedrichshain-kreuzberg	m	NaN	Karl	2016
14	33	friedrichshain-kreuzberg	m	NaN	Paul	2016
15	31	friedrichshain-kreuzberg	m	NaN	Oskar	2016
17	30	friedrichshain-kreuzberg	m	NaN	Felix	2016
19	29	friedrichshain-kreuzberg	m	NaN	Noah	2016
21	28	friedrichshain-kreuzberg	m	NaN	Henry	2016
22	27	friedrichshain-kreuzberg	m	NaN	Jonas	2016
24	26	friedrichshain-kreuzberg	m	NaN	Jonathan	2016
25	26	friedrichshain-kreuzberg	m	NaN	Maximilian	2016
29	24	friedrichshain-kreuzberg	m	NaN	Valentin	2016
28	24	friedrichshain-kreuzberg	m	NaN	Jakob	2016
32	23	friedrichshain-kreuzberg	m	NaN	Julius	2016
34	22	friedrichshain-kreuzberg	m	NaN	Moritz	2016
36	21	friedrichshain-kreuzberg	m	NaN	Adam	2016
37	21	friedrichshain-kreuzberg	m	NaN	Ben	2016
38	21	friedrichshain-kreuzberg	m	NaN	David	2016
39	21	friedrichshain-kreuzberg	m	NaN	Friedrich	2016
40	21	friedrichshain-kreuzberg	m	NaN	Louis	2016
41	21	friedrichshain-kreuzberg	m	NaN	Lukas	2016
48	19	friedrichshain-kreuzberg	m	NaN	Liam	2016
49	19	friedrichshain-kreuzberg	m	NaN	Luca	2016
52	18	friedrichshain-kreuzberg	m	NaN	Carl	2016
53	18	friedrichshain-kreuzberg	m	NaN	Levi	2016
54	18	friedrichshain-kreuzberg	m	NaN	Michael	2016
...	...	...	...	...	...	...
1936	1	friedrichshain-kreuzberg	m	NaN	Glenn	2016
1934	1	friedrichshain-kreuzberg	m	NaN	Glen	2016
1932	1	friedrichshain-kreuzberg	m	NaN	Giuseppe	2016
1930	1	friedrichshain-kreuzberg	m	NaN	Giovanni	2016
1928	1	friedrichshain-kreuzberg	m	NaN	Giocondo	2016
1926	1	friedrichshain-kreuzberg	m	NaN	Gino	2016
1924	1	friedrichshain-kreuzberg	m	NaN	Gilbert	2016
1922	1	friedrichshain-kreuzberg	m	NaN	Gharbi	2016
1920	1	friedrichshain-kreuzberg	m	NaN	Gero	2016
1918	1	friedrichshain-kreuzberg	m	NaN	Georges	2016
1948	1	friedrichshain-kreuzberg	m	NaN	Günter	2016
1952	1	friedrichshain-kreuzberg	m	NaN	Güven	2016
1986	1	friedrichshain-kreuzberg	m	NaN	Haydar	2016

```
1954       1  friedrichshain-kreuzberg          m       NaN       Hadar  2016
1984       1  friedrichshain-kreuzberg          m       NaN      Hayato  2016
1982       1  friedrichshain-kreuzberg          m       NaN       Hauke  2016
1980       1  friedrichshain-kreuzberg          m       NaN      Harris  2016
1978       1  friedrichshain-kreuzberg          m       NaN     Harouna  2016
1976       1  friedrichshain-kreuzberg          m       NaN    Hannibal  2016
1974       1  friedrichshain-kreuzberg          m       NaN      Hanifi  2016
1972       1  friedrichshain-kreuzberg          m       NaN       Hanad  2016
1970       1  friedrichshain-kreuzberg          m       NaN   Hamza-Can  2016
1968       1  friedrichshain-kreuzberg          m       NaN       Hamsi  2016
1966       1  friedrichshain-kreuzberg          m       NaN    Hamallah  2016
1964       1  friedrichshain-kreuzberg          m       NaN        Hakî  2016
1962       1  friedrichshain-kreuzberg          m       NaN       Haika  2016
1960       1  friedrichshain-kreuzberg          m       NaN       Hagen  2016
1958       1  friedrichshain-kreuzberg          m       NaN        Hadi  2016
1956       1  friedrichshain-kreuzberg          m       NaN      Haddou  2016
3520       1  friedrichshain-kreuzberg          m       NaN       ükrü  2016

[1807 rows x 6 columns]
```

In [70]:
```
# most popular names per year and borough
all_names_df.groupby(['borough','year'],as_index=False).\
    agg({"anzahl": "max",'vorname':'first'})
```

Out[70]:

	borough	year	anzahl	vorname
0	charlottenburg-wilmersdorf	2013	121	Marie
1	charlottenburg-wilmersdorf	2014	118	Marie
2	charlottenburg-wilmersdorf	2015	115	Marie
3	charlottenburg-wilmersdorf	2016	111	Marie
4	charlottenburg-wilmersdorf	2017	72	Marie
5	friedrichshain-kreuzberg	2013	71	Marie
6	friedrichshain-kreuzberg	2014	70	Sophie
7	friedrichshain-kreuzberg	2015	65	Marie
8	friedrichshain-kreuzberg	2016	64	Charlotte
9	friedrichshain-kreuzberg	2017	47	Marie
10	lichtenberg	2013	66	Marie
11	lichtenberg	2014	69	Sophie
12	lichtenberg	2015	69	Marie
13	lichtenberg	2016	70	Sophie
14	lichtenberg	2017	35	Marie
15	marzahn-hellersdorf	2013	23	Sophie
16	marzahn-hellersdorf	2014	25	Marie
17	marzahn-hellersdorf	2015	30	Marie
18	marzahn-hellersdorf	2016	27	Marie
19	marzahn-hellersdorf	2017	18	Marie
20	mitte	2013	64	Marie
21	mitte	2014	73	Sophie
22	mitte	2015	80	Marie

23	mitte	2016	69	Sophie
24	mitte	2017	50	Marie
25	neukoelln	2013	59	Sophie
26	neukoelln	2014	54	Marie
27	neukoelln	2015	55	Ali
28	neukoelln	2016	57	Sophie
29	neukoelln	2017	32	Sophie
30	pankow	2013	119	Marie
31	pankow	2014	122	Marie
32	pankow	2015	128	Marie
33	pankow	2016	112	Marie
34	pankow	2017	76	Marie
35	reinickendorf	2013	33	Marie
36	reinickendorf	2014	22	Marie
37	reinickendorf	2015	24	Marie
38	reinickendorf	2016	26	Marie
39	reinickendorf	2017	13	Marie
40	spandau	2013	71	Marie
41	spandau	2014	75	Sophie
42	spandau	2015	74	Marie
43	spandau	2016	64	Sophie
44	spandau	2017	47	Marie
45	steglitz-zehlendorf	2013	23	Sophie
46	steglitz-zehlendorf	2014	27	Sophie
47	steglitz-zehlendorf	2015	27	Marie
48	steglitz-zehlendorf	2016	23	Marie
49	steglitz-zehlendorf	2017	17	Sophie
50	tempelhof-schoeneberg	2013	93	Sophie
51	tempelhof-schoeneberg	2014	114	Marie
52	tempelhof-schoeneberg	2015	103	Sophie
53	tempelhof-schoeneberg	2016	94	Marie
54	tempelhof-schoeneberg	2017	62	Sophie
55	treptow-koepenick	2013	27	Sophie
56	treptow-koepenick	2014	25	Sophie
57	treptow-koepenick	2015	25	Marie
58	treptow-koepenick	2016	21	Marie
59	treptow-koepenick	2017	18	Marie

In [71]:
```
# least popular names per year and borough
all_names_df.groupby(['borough','year'],as_index=False).\
    agg({"anzahl": "min",'vorname':'last'})
```

Out[71]:

	borough	year	anzahl	vorname
0	charlottenburg-wilmersdorf	2013	1	elila
1	charlottenburg-wilmersdorf	2014	1	erife
2	charlottenburg-wilmersdorf	2015	1	evket
3	charlottenburg-wilmersdorf	2016	1	irin
4	charlottenburg-wilmersdorf	2017	1	ervan

5	friedrichshain-kreuzberg	2013	1	emsi
6	friedrichshain-kreuzberg	2014	1	ura
7	friedrichshain-kreuzberg	2015	1	efik
8	friedrichshain-kreuzberg	2016	1	ükrü
9	friedrichshain-kreuzberg	2017	1	iyar
10	lichtenberg	2013	1	Öne
11	lichtenberg	2014	1	Ziyu
12	lichtenberg	2015	1	ukasz
13	lichtenberg	2016	1	ình
14	lichtenberg	2017	1	imon
15	marzahn-hellersdorf	2013	1	Yusuf
16	marzahn-hellersdorf	2014	1	Zacharias
17	marzahn-hellersdorf	2015	1	Zvezdelina
18	marzahn-hellersdorf	2016	1	ình
19	marzahn-hellersdorf	2017	1	Zeon
20	mitte	2013	1	erif
21	mitte	2014	1	irin
22	mitte	2015	1	hirin
23	mitte	2016	1	ilan
24	mitte	2017	1	tefanija
25	neukoelln	2013	1	irin
26	neukoelln	2014	1	efika
27	neukoelln	2015	1	ivomir
28	neukoelln	2016	1	tefania
29	neukoelln	2017	1	Özgür
30	pankow	2013	1	c
31	pankow	2014	1	Éloise
32	pankow	2015	1	Évangéline
33	pankow	2016	1	Émie
34	pankow	2017	1	ukasz
35	reinickendorf	2013	1	eljko
36	reinickendorf	2014	1	ahin
37	reinickendorf	2015	1	ukasz
38	reinickendorf	2016	1	Çaan
39	reinickendorf	2017	1	Zoè
40	spandau	2013	1	eripovna
41	spandau	2014	1	ura
42	spandau	2015	1	evki
43	spandau	2016	1	ehida
44	spandau	2017	1	ura
45	steglitz-zehlendorf	2013	1	Zimin
46	steglitz-zehlendorf	2014	1	Zuri
47	steglitz-zehlendorf	2015	1	Zephyr
48	steglitz-zehlendorf	2016	1	
49	steglitz-zehlendorf	2017	1	Ömer
50	tempelhof-schoeneberg	2013	1	ucja
51	tempelhof-schoeneberg	2014	1	ura
52	tempelhof-schoeneberg	2015	1	afak

```
53       tempelhof-schoeneberg  2016       1       irin
54       tempelhof-schoeneberg  2017       1       ükrü
55           treptow-koepenick  2013       1       Yusup
56           treptow-koepenick  2014       1       ucja
57           treptow-koepenick  2015       1       Yorin
58           treptow-koepenick  2016       1          a
59           treptow-koepenick  2017       1        ifa
```

8 Exercises

We will use pandas to get some insights into what the Berlin Senat spends money on

```
In [122]: df_ausgaben_berlin = pd.read_csv("data/zuwendungen-berlin.csv.gz")
          df_ausgaben_berlin.sample(n=10)
```

Out[122]:

```
                                                    Name  \
4885   Modul e. V., Förderverein Modernes Lehren und ...
16451              Alevitische Gemeinde zu Berlin e. V.
15497  Verein für ambulante Versorgung Hohenschönhaus...
17859                                              BVG
16441                          Albatros-Lebensnetz gGmbH
39643  Tauwetter - vereint gegen sexualisierte Gewalt...
17573                                  Bouledozer e. V.
31121                                  Stiftung Synanon
1536                          BTB Bildungszentrum GmbH
27400             Frauenzentrum Schokoladenfabrik e. V.

                                                  Geber                Art  \
4885   Senatsverwaltung für Arbeit, Integration und F...  Projektförderung
16451  Senatsverwaltung für Arbeit, Integration und F...  Projektförderung
15497  Senatsverwaltung für Arbeit, Integration und F...  Projektförderung
17859   Senatsverwaltung für Stadtentwicklung und Umwelt  Projektförderung
16441       Senatsverwaltung für Gesundheit und Soziales  Projektförderung
39643       Senatsverwaltung für Gesundheit und Soziales  Projektförderung
17573             Senatsverwaltung für Inneres und Sport  Projektförderung
31121       Senatsverwaltung für Gesundheit und Soziales  Projektförderung
1536   Senatsverwaltung für Arbeit, Integration und F...  Projektförderung
27400  Senatsverwaltung für Arbeit, Integration und F...  Projektförderung

       Jahr                                 Anschrift Politikbereich  \
4885   2012       Grüntaler Straße 62, 13359 Berlin         Arbeit
16451  2014         Waldemarstraße 20, 10999 Berlin         Arbeit
15497  2013      Ribnitzer Straße 1 B, 13051 Berlin         Arbeit
17859  2014   Holzmarktstraße 15 - 17, 10179 Berlin        Verkehr
16441  2014        Berliner Straße 14, 13507 Berlin     Gesundheit
39643  2016        Gneisenaustraße 2a, 10961 Berlin     Gesundheit
17573  2014             Kruppstraße 5, 10557 Berlin          Sport
31121  2015              Dorfstraße 9, 13051 Berlin     Gesundheit
```

```
1536   2012  Straßburger Straße 6 - 9, 10405 Berlin         Arbeit
27400  2015           Naunynstraße 72, 10997 Berlin         Frauen

                                                    Zweck   Betrag
4885   BVBO / Rheingau-Gymnasium / Jahrgangsstufe Kl...     4139
16451  FAV - Interkulturelle Helfer für Flüchtlinge, ...     7259
15497  Unterstützung der Stadtteilarbeit des Vereins ...    45072
17859    U5; PB I; Projektlos 3.0, Rohbau sdl. Bundestag  6237250
16441  Albatros- Lebensnetz gGmbH Schwangerschafts-un...   251727
39643  Tauwetter, Informations- und Beratungsstelle f...   142532
17573                  Teilhabeprogramm"Boule ist Cool"    16000
31121  Suchtselbsthilfe Synanon - Sicherung der Aufna...   290048
1536     Modulare Qualifizierung "Lager & Logistik 1-12"     3110
27400  Förderung der Gleichstellung von Frauen und Mä...   175808
```

8.1 Assignment 01

Extract some summary statistics of the money spent by the Senat of Berlin

Write a function `assignment_04_01` that takes the data frame of spendings and returns

- the count
- the mean
- the standard deviation
- the minimum
- the 25% percentile
- the 50% percentile (median)
- the 75% percentile
- the maximum

of all spendings in a list. The data is in the subdirectory `data` and can be loaded by `df = pd.read_csv("data/zuwendungen-berlin.csv.gz")`. For convenient compuation of the summary statistics check the pandas Series API for `describe()`

```
def assignment_04_01(df):
    spending_statistics = df. ...
    ...
    return spending_statistics
```

8.2 Assignment 02

How much is each recipient of a spending receiving in total?

Write a function `assignment_04_02` that takes the data frame of spendings and groups by recipient (column `'Name'`) and then sums all money received for each recipient. Return the names of the recipients that received in total 143 Euros.

```
def assignment_04_01(df):
    money_received = df.groupby(['Name']). ...
```

```
    ...
    return names_of_recipients
```

8.3 Assignment 03

How much is Berlin spending on each political ressort?

Write a function `assignment_04_03` that takes the data frame of spendings (spending is the column 'Betrag'), groups by political ressort (in german 'Politikbereich') and computes the

- minimum
- median
- maximum

of the spendings on each political ressort. Return the aggregates in the political ressort ('Politikbereich') 'sciences' ('Wissenschaft')

```
def assignment_04_03(df):
    spending_per_ressort = df.groupby(['Politikbereich']). ...
    ...
    return
```

8.4 Assignment 04

How much is Berlin spending on each U-Bahn?

Write a function `assignment_04_04` that takes the data frame of spendings, filters for transportation (german 'Verkehr'), groups by the specific ubahn and sums up the spendings. For the ubahn grouping you can extract the ubahn with the regular expression `'U[1-9]'`. The function should return the ubahn names ordered from most (first element) to least expensive (last element).

```
def assignment_04_04(df):
    df['ubahn'] = df['Zweck'].str.extract('(U[1-9])') ...
    ...
    return
```