UNIVERSITÄT
BIELEFELD
Faculty of Technology

Programming

Tabular Data Analays

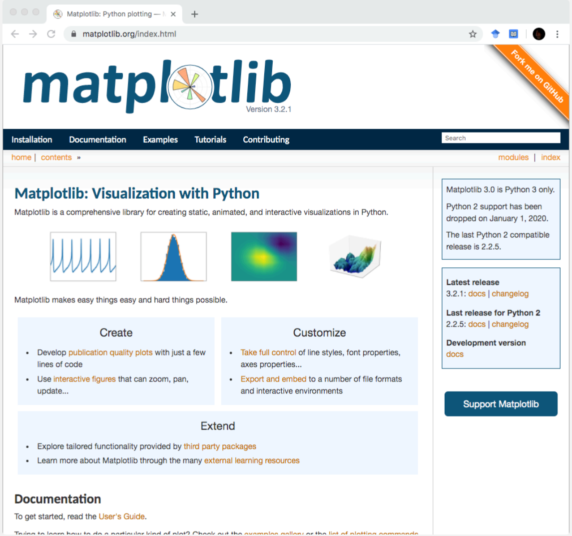
Daniel Dörr

Faculty of Technology, Bielefeld University

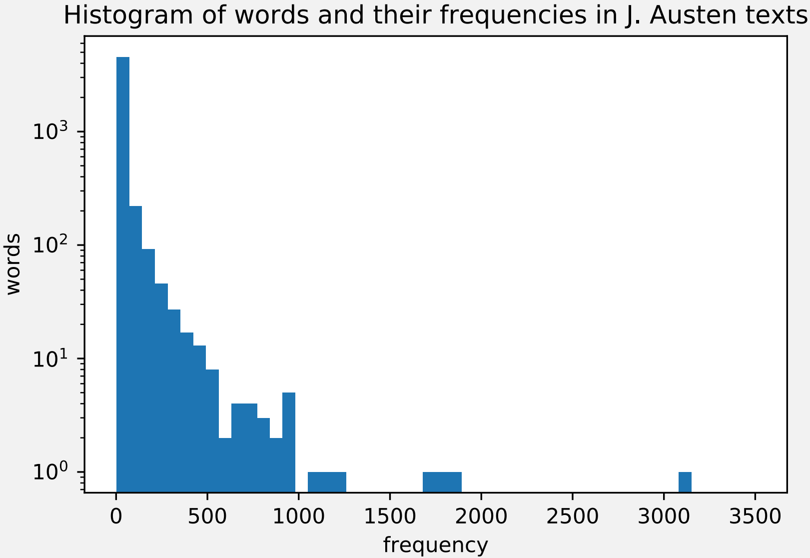
Recap

Matplotlib: Visualization with Python

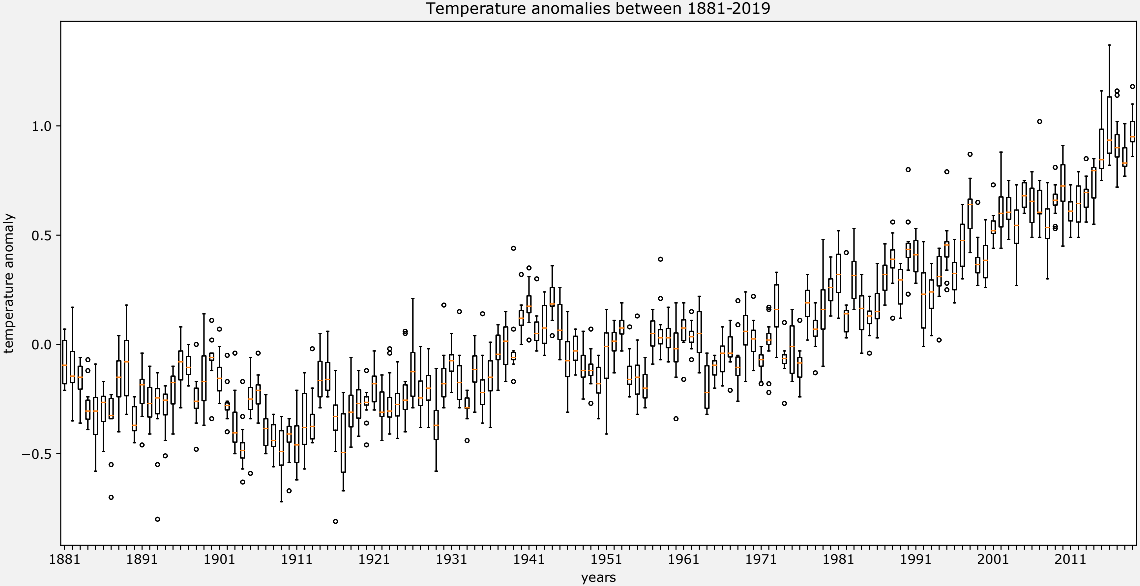
- de-facto standard library for scientific visualizations
- many third party packages built on top of Matplotlib
- comprehensive library for creating static, animated, and interactive visualizations



source: `https://matplotlib.org/`


Histogram of words and their frequencies in J. Austen texts
words
10^3
10^2
10^1
10^0
0
500
1000
1500
2000
2500
3000
3500
frequency

Whisker plot of GISS data

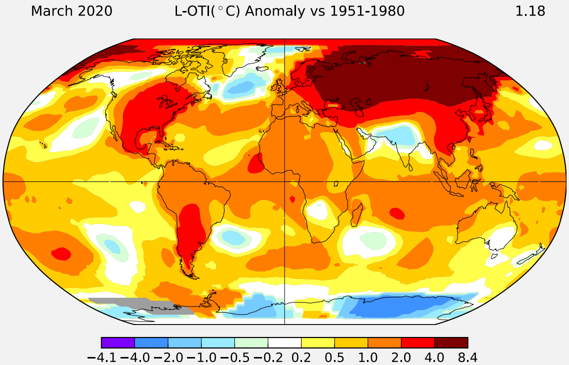


N-dimensional array: `numpy.ndarray`

Array data structure

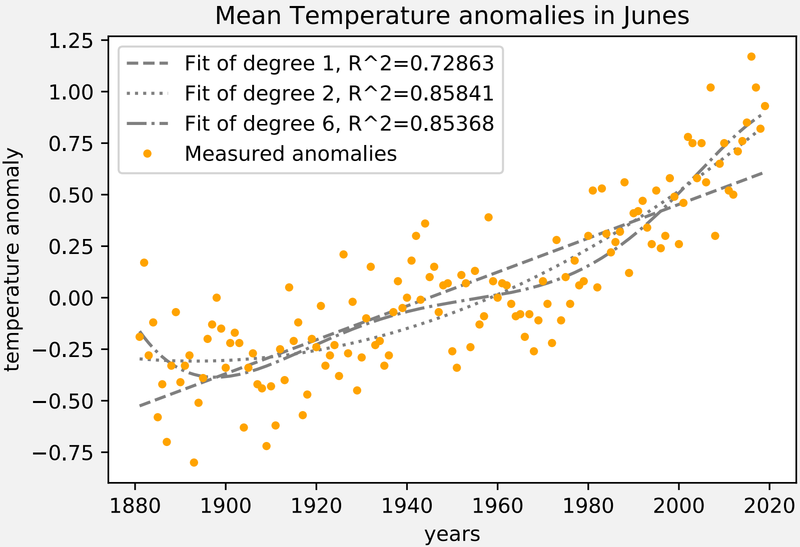
- immutable
- n-dimensional
- very storage efficient
- can store only data of same type

NASA's GISS Surface Temperature Analysis



- `https://data.giss.nasa.gov/gistemp`
- Collection of temperature data from thousands of meteorological stations
- Data represents *anomalies*, i.e., deviations from mean temperature measured in 1951-1980

Linear Regression with `numpy.polyfit()`



Pandas Series

Pandas DataFrame

Multi-Indexing DataFrames

Pandas data structures

Series

- Container for scalar values
- 1D array
- More powerful than a “1D NumPy array”
- Allows to freely set index
- Size immutable

Data Frame

- Container for Series
- 2D array / table
- Mutability
  - Rows are immutable
  - Allows insertion of new columns

Lecture 06: Tabular Data Analysis with Pandas

The Pandas package is a toolkit for processing and analyzing tabular data. Tables can contain numerical data, but also categorical/ordinal data.

Main literature: https://jakevdp.github.io/PythonDataScienceHandbook (https://jakevdp.github.io/PythonDataScienceHandbook)

Series

`pandas.Series` are size-immutable 1D arrays. They are more powerful than NumPy arrays in handling such data, because they enable free choice of the Series' index.

Creating Series

The simplest way to create a Series is by supplying any kind of collection:

In [1]:
```
import pandas as pd

x = pd.Series((1, 5, 8, 1, 9))
print(x)

print(f'the value of element with index 2 in x is {x[2]}')
```

```
0     1
1     5
2     8
3     1
4     9
dtype: int64
the value of element with index 2 in x is 8
```

Changing values

Values of a Series instance can be updated, just as in a NumPy array:

In [2]:
```
x[2] = 6

print(f'the updated value of element with index 2 in x is {x[2]}')
```

```
the updated value of element with index 2 in x is 6
```

The print otuput shows two columns: index (left), values (right). Just as NumPy arrays, Series are typed according to the *values* that they store.

In [3]:
```
x.values
```

Out[3]:
```
array([1, 5, 6, 1, 9])
```

In [4]:
```
x.values.dtype
```

Out[4]:
```
dtype('int64')
```

Explicit Indexes

Series allows explicit declaration of the index. The supplied index does not need to correspond to *consecutive* index positions, as shown below:

In [5]:
```
x = pd.Series((1, 5, 8, 1, 9), index=(0, 4, 6, 2, 1))
print(x)

print(f'the value of element with index 4 in x is {x[4]}')
```

```
0     1
4     5
6     8
2     1
1     9
dtype: int64
the value of element with index 4 in x is 5
```

In [6]:
```
x.keys()
```

Out[6]:
```
Int64Index([0, 4, 6, 2, 1], dtype='int64')
```

It must not even be an integer index

In [7]:
```
x = pd.Series((1, 5, 8, 1, 9), index=(0.0, 0.6, 0.3, 0.2, 0.1))
print(x)

print(f'the value of element with index 0.3 in x is {x[0.3]}')
```

```
0.0    1
0.6    5
0.3    8
0.2    1
0.1    9
dtype: int64
the value of element with index 0.3 in x is 8
```

In [8]:
```
x.keys()
```

Out[8]:
```
Float64Index([0.0, 0.6, 0.3, 0.2, 0.1], dtype='float64')
```

Any kind of data type can be used for indexing:

In [9]:
```
x = pd.Series((1, 5, 8, 1, 9), index=(1, 'text', 0.3, (1, 2), 1+1j))
print(x)

print(f'the value of element with index (1,2) in x is {x[(1,2)]}')
```

```
1         1
text      5
0.3       8
(1, 2)    1
(1+1j)    9
dtype: int64
the value of element with index (1,2) in x is 1
```

In [10]:
```
x.keys()
```

Out[10]:
```
Index([1, 'text', 0.3, (1, 2), (1+1j)], dtype='object')
```

Pandas can create Series instances directly from dictionaries:

In [11]:
```
my_dict = {
    1: 1,
    'text': 5,
    0.3: 8,
    (1, 2): 1,
    1+1j : 9}

x = pd.Series(my_dict)
print(x)
```

```
1          1
text       5
0.3        8
(1, 2)     1
(1+1j)     9
dtype: int64
```

In [12]:
```
list(my_dict.items())
```

Out[12]:
```
[(1, 1), ('text', 5), (0.3, 8), ((1, 2), 1), ((1+1j), 9)]
```

In [13]:
```
list(x.items())
```

Out[13]:
```
[(1, 1), ('text', 5), (0.3, 8), ((1, 2), 1), ((1+1j), 9)]
```

Also the 'in' operator works as expected:

In [14]:
```
if 'text' in x:
    print('\'text\' is a key of Series x')
```

```
'text' is a key of Series x
```

The Pandas Series acts similar to the builtin Python dictionary, but it is more flexible, because it allows mutable data types as keys (indices) as well as duplicate keys:

In [15]:
```
y = pd.Series((1, 5, 8, 1, 9), index=(0.3, 'text', 1, 0.3, 1))
print(y)

print(f'the value of element with index 1 in y is: \n{y[1]}\n' + \
      f'and is of type {type(y[1])}')
```

```
0.3     1
text    5
1       8
0.3     1
1       9
dtype: int64
the value of element with index 1 in y is:
1    8
1    9
dtype: int64
and is of type <class 'pandas.core.series.Series'>
```

Slicing

The true power of Pandas' Series data type is the broad support for slicing or the user-defined index:

In [16]:
```
x['text':(1, 2)]
```

Out[16]:
```
text      5
0.3       8
(1, 2)    1
dtype: int64
```

This example reveals that the user-defined index is *ordered*. This order corresponds the order in which the index is constructed:

In [17]:
```
my_dict.keys()
```

Out[17]:
```
dict_keys([1, 'text', 0.3, (1, 2), (1+1j)])
```

In [18]:
```
x.keys()
```

Out[18]:
```
Index([1, 'text', 0.3, (1, 2), (1+1j)], dtype='object')
```

At the same time, elements can also be accessed by using the index position of this order:

In [19]:
```
x[1:4]
```

Out[19]:
```
text      5
0.3       8
(1, 2)    1
dtype: int64
```

Here is another example to highlight the order of the index:

In [20]:
```
my_dict = {
     'Cypress': 1,
     'Russia': 5,
     'Alaska': 8,
     'India': 1,
     'Australia' : 9}
x = pd.Series(my_dict)
print(x)
```

```
Cypress      1
Russia       5
Alaska       8
India        1
Australia    9
dtype: int64
```

In [21]:
```
my_dict.keys()
```
Out[21]:
```
dict_keys(['Cypress', 'Russia', 'Alaska', 'India', 'Australia'])
```
In [22]:
```
x.keys()
```
Out[22]:
```
Index(['Cypress', 'Russia', 'Alaska', 'India', 'Australia'], dtype='object')
```
In [23]:
```
x['Russia':'India']
```
Out[23]:
```
Russia     5
Alaska     8
India      1
dtype: int64
```

Sorted index

In [24]:
```
y = x.sort_index()
print(y)
```

```
Alaska       8
Australia    9
Cypress      1
India        1
Russia       5
dtype: int64
```

In [25]:
```
y['Australia':'India']
```

Out[25]:
```
Australia    9
Cypress      1
India        1
dtype: int64
```

Confusion!

The possiblity to access elements by both indexes, the explicit index, and the implicit/position-based index can lead to erratic behaviour of the "bracket" operation:

In [26]:
```
x = pd.Series((1, 5, 8, 1, 9), index=(1, 'text', 0.3, (1, 2), 1+1j))
print(f'the value of element with index 1 in x is {x[1]}')

print('all elments from 1 to the end of the series:')
x[1:]
```

```
the value of element with index 1 in x is 1
all elments from 1 to the end of the series:
```

Out[26]:
```
text      5
0.3       8
(1, 2)    1
(1+1j)    9
dtype: int64
```

It is not clear, whether the programmer wants to refer to all elements from *index 1 onwards* or *position 1 onwards*.

Direct access to explicit and positional index

Another example:

In [27]:
```
x = pd.Series((1, 5, 8, 1, 9), index = (1, 4, 2, 0, 3))
print(x)

x[1:3]
```

```
1    1
4    5
2    8
0    1
3    9
dtype: int64
```

Out[27]:
```
4    5
2    8
dtype: int64
```

Also, here, it is not clear if the programmer intended to access the explicit or positional index. Pandas computed the results according to the positional index.

Pandas provides a way to directly access the explicit index and the positional index.

The `loc` attribute provides access of the *explicit* index:

In [28]:
```
x.loc[1:3]  # inclusive
```

Out[28]:
```
1    1
4    5
2    8
0    1
3    9
dtype: int64
```

The `iloc` attribute refers to the the *positional index*:

In [29]:
```
x.iloc[1:3] # exclusive
```

Out[29]:
```
4    5
2    8
dtype: int64
```

Just as slicing Python lists or tuples, the stop position of the slice is exclusive.

Quiz

- Given the following series `x = pd.Series((1, 5, 8, 1, 9), index = (1, 4, 2, 0, 3))`, what is the result of the following expressions?
  - `x[1:3]`
  - `x[2]`
  - `x.loc[1:3]`
  - `x.iloc[1:3]`

- *True* or *false?*
  - Pandas Series allows indexes to be of any type, but just as `dict()` prohibits duplicate keys
  - Every Pandas Series instance maintains two indexes: the *explicit* index, and the *positional* index.
  - Indexes in Pandas Series are always *sorted*.
  - The terms *ordered* and *sorted* are synonyms.

Quiz

- Given the following series `x = pd.Series((1, 5, 8, 1, 9), index = (1, 4, 2, 0, 3))`, what is the result of the following expressions?
  - `x[1:3]` `pd.Series({4:5, 2:8})`
  - `x[2]` `8`
  - `x.loc[1:3]` `pd.Series({1:1,4:5, 2:8, 0:1, 3:9})`
  - `x.iloc[1:3]` `pd.Series({4:5, 2:8})`

- *True* or *false?*
  - Pandas Series allows indexes to be of any type, but just as `dict()` prohibits duplicate keys false
  - Every Pandas Series instance maintains two indexes: the *explicit* index, and the *positional* index. true
  - Indexes in Pandas Series are always *sorted*. false
  - The terms *ordered* and *sorted* are synonyms. false

Pandas Series

Pandas DataFrame

Multi-Indexing DataFrames

DataFrame

In [30]:

```
federal_states = ['Schleswig-Holstein', 'Mecklenburg-Vorpommern',
                  'Rheinland-Pfalz', 'Hessen', 'Nordrhein-Westfalen',
                  'Brandenburg', 'Hamburg', 'Bremen', 'Saarland',
                  'Baden-Wuerttemberg', 'Sachsen-Anhalt', 'Thueringen',
                  'Bayern', 'Niedersachsen', 'Berlin', 'Sachsen']

population_female = [1360484,  793140, 1950352, 2913862, 8517934, 1208327,
                      825451,  316102,  485050, 5132555, 1117016, 1076074,
                     6062701, 3803776, 1599653, 1977567]
population_male   = [1439635,  816841, 2039456, 3057954, 9020318, 1247453,
                      881245,  334761,  514573, 5354105, 1170024, 1112515,
                     6334913, 3974216, 1692712, 2079232]

demogrphx = pd.DataFrame(
    zip(federal_states, population_female, population_male),
    columns=('FederalState', 'PopulationFemale', 'PopulationMale'))

demogrphx.head()
```

Out[30]:

	FederalState	PopulationFemale	PopulationMale
0	Schleswig-Holstein	1360484	1439635
1	Mecklenburg-Vorpommern	793140	816841
2	Rheinland-Pfalz	1950352	2039456
3	Hessen	2913862	3057954
4	Nordrhein-Westfalen	8517934	9020318

DataFrames can also be constructed from Series that share the same index:

In [31]:
```
popFemaleSeries = pd.Series(population_male, index=federal_states)
popMaleSeries = pd.Series(population_male, index=federal_states)

# create DataFrame from Series with same Index
demogrphx = pd.DataFrame({'PopulationFemale': popFemaleSeries,
                          'PopulationMale': popMaleSeries})
demogrphx.head()
```

Out[31]:

	PopulationFemale	PopulationMale
Schleswig-Holstein	1439635	1439635
Mecklenburg-Vorpommern	816841	816841
Rheinland-Pfalz	2039456	2039456
Hessen	3057954	3057954
Nordrhein-Westfalen	9020318	9020318

Each column corresponds to a Series instance.

Accessing columns, rows, and values

In [32]:
```
demogrphx['PopulationFemale'].head()
```

Out[32]:
```
Schleswig-Holstein        1439635
Mecklenburg-Vorpommern     816841
Rheinland-Pfalz           2039456
Hessen                    3057954
Nordrhein-Westfalen       9020318
Name: PopulationFemale, dtype: int64
```

In [33]:
```
type(demogrphx['PopulationFemale'])
```

Out[33]:
```
pandas.core.series.Series
```

In [34]:
```
demogrphx.PopulationFemale.head()
```

Out[34]:
```
Schleswig-Holstein        1439635
Mecklenburg-Vorpommern     816841
Rheinland-Pfalz           2039456
Hessen                    3057954
Nordrhein-Westfalen       9020318
Name: PopulationFemale, dtype: int64
```

Slicing: just like Series...

Attention: Slicing operates on rows!

In [35]:
```
demogrphx.loc['Mecklenburg-Vorpommern':'Hessen']
```

Out[35]:

	PopulationFemale	PopulationMale
Mecklenburg-Vorpommern	816841	816841
Rheinland-Pfalz	2039456	2039456
Hessen	3057954	3057954

In [36]:
```
demogrphx.iloc[1:4]
```

Out[36]:

	PopulationFemale	PopulationMale
Mecklenburg-Vorpommern	816841	816841
Rheinland-Pfalz	2039456	2039456
Hessen	3057954	3057954

Orientation of DataFrame

`DataFrame` stores values in a column-first fashion, whereas NumPy arrays are rows-first oriented:

In [37]:
```
demogrphx['PopulationFemale'][0] #first column, then row
```

Out[37]:
```
1439635
```

In [38]:
```
print(demogrphx.values)
print(f'first element of NumpyArray demographx.values is ' + \
      f'{demogrphx.values[0]}')
```

```
[[1439635 1439635]
 [ 816841  816841]
 [2039456 2039456]
 [3057954 3057954]
 [9020318 9020318]
 [1247453 1247453]
 [ 881245  881245]
 [ 334761  334761]
 [ 514573  514573]
 [5354105 5354105]
 [1170024 1170024]
 [1112515 1112515]
 [6334913 6334913]
 [3974216 3974216]
 [1692712 1692712]
 [2079232 2079232]]
first element of NumpyArray demographx.values is [1439635 1439635]
```

Transposing a Pandas Dataframe is analog to tranposing a NumPy Array!

In [39]:

```
demogrphx.T
```

Out[39]:

	Schleswig-Holstein	Mecklenburg-Vorpommern	Rheinland-Pfalz	Hessen	Nordrhein-Westfalen	Brandenburg	Hamburg	Bremen	Saarland	Ba Wuertten
PopulationFemale	1439635	816841	2039456	3057954	9020318	1247453	881245	334761	514573	5354105
PopulationMale	1439635	816841	2039456	3057954	9020318	1247453	881245	334761	514573	5354105

Selecting subsets of columns

In [40]:
```
sub_lst = ['Hessen', 'Brandenburg']
demogrphx.T[sub_lst]
```

Out[40]:

	Hessen	Brandenburg
PopulationFemale	3057954	1247453
PopulationMale	3057954	1247453

In [41]:
```
demogrphx.T[['Hessen', 'Brandenburg']]
```

Out[41]:

	Hessen	Brandenburg
PopulationFemale	3057954	1247453
PopulationMale	3057954	1247453

Broadcasting with DataFrame

In [42]:

```
demo_mil = demogrphx / 1_000_000 # better readible than 1000000
demo_mil.columns = ('PopulationMale (M)', 'PopulationFemale (M)')

demo_mil.head()
```

Out[42]:

	PopulationMale (M)	PopulationFemale (M)
Schleswig-Holstein	1.439635	1.439635
Mecklenburg-Vorpommern	0.816841	0.816841
Rheinland-Pfalz	2.039456	2.039456
Hessen	3.057954	3.057954
Nordrhein-Westfalen	9.020318	9.020318

Broadcasting on DataFrames assumes row-wise processing

In [43]:
```
demo_mil = demogrphx / (1_000_000, 1_000_000)
demo_mil.head()
```

Out[43]:

	PopulationFemale	PopulationMale
Schleswig-Holstein	1.439635	1.439635
Mecklenburg-Vorpommern	0.816841	0.816841
Rheinland-Pfalz	2.039456	2.039456
Hessen	3.057954	3.057954
Nordrhein-Westfalen	9.020318	9.020318

Column-wise broadcasting can be achieved by explicit function calls

In [44]:
```
import numpy as np

# create an array of 16 elements, one for each federal state
one_mil = np.ones(len(demogrphx)) * 1_000_000

# divide columnwise
demogrphx.divide(one_mil, axis=0).head()
```

Out[44]:

	PopulationFemale	PopulationMale
Schleswig-Holstein	1.439635	1.439635
Mecklenburg-Vorpommern	0.816841	0.816841
Rheinland-Pfalz	2.039456	2.039456
Hessen	3.057954	3.057954
Nordrhein-Westfalen	9.020318	9.020318

Adding new column to existing DataFrame

Same as creating new entries in a dictionary:

In [45]:

```
demogrphx['PopulationTotal'] = demogrphx.PopulationFemale + \
        demogrphx.PopulationMale
demogrphx.head()
```

Out[45]:

	PopulationFemale	PopulationMale	PopulationTotal
Schleswig-Holstein	1439635	1439635	2879270
Mecklenburg-Vorpommern	816841	816841	1633682
Rheinland-Pfalz	2039456	2039456	4078912
Hessen	3057954	3057954	6115908
Nordrhein-Westfalen	9020318	9020318	18040636

Multi-level Columns

In [46]:

```
demogrphx.columns = pd.MultiIndex.from_tuples(
    (('Population', 'Female'),
     ('Population', 'Male'),
     ('Population', 'Total')))

demogrphx.head()
```

Out[46]:

	Population		
	Female	Male	Total
Schleswig-Holstein	1439635	1439635	2879270
Mecklenburg-Vorpommern	816841	816841	1633682
Rheinland-Pfalz	2039456	2039456	4078912
Hessen	3057954	3057954	6115908
Nordrhein-Westfalen	9020318	9020318	18040636

Acccess of sublevel columns

In [47]:
```
demogrphx[('Population', 'Female')].head()
```

Out[47]:
```
Schleswig-Holstein        1439635
Mecklenburg-Vorpommern     816841
Rheinland-Pfalz           2039456
Hessen                    3057954
Nordrhein-Westfalen       9020318
Name: (Population, Female), dtype: int64
```

In [48]:
```
demogrphx.Population.Female.head()
```

Out[48]:
```
Schleswig-Holstein        1439635
Mecklenburg-Vorpommern     816841
Rheinland-Pfalz           2039456
Hessen                    3057954
Nordrhein-Westfalen       9020318
Name: Female, dtype: int64
```

Also the index can be named:

In [49]:

```
demogrphx.index.name = 'FederalState'
demogrphx.head()
```

Out[49]:

	Population		
	Female	Male	Total
FederalState			
Schleswig-Holstein	1439635	1439635	2879270
Mecklenburg-Vorpommern	816841	816841	1633682
Rheinland-Pfalz	2039456	2039456	4078912
Hessen	3057954	3057954	6115908
Nordrhein-Westfalen	9020318	9020318	18040636

Reading and writing data from files with Pandas

Pandas provides functions to read in tabular data of various formats, such as CSV, Excel, JSON, SPSS, etc.

In [50]:
```
demogrphx = pd.read_table('12111-04-01-4-B_processed2.tsv', index_col = 0,
                          header=[0, 1])
demogrphx
```

Out[50]:

	Age	Population		
	Age	Male	Female	Total
FederalState				
Schleswig-Holstein	0	11132	10400	21532
Schleswig-Holstein	1	11504	10360	21864
Schleswig-Holstein	2	11733	11067	22800
Schleswig-Holstein	3	12214	11147	23361
Schleswig-Holstein	4	12142	10945	23087
...	...	...	...	...
Thueringen	96	149	599	748
Thueringen	97	75	476	551
Thueringen	98	47	275	322
Thueringen	99	30	181	211
Thueringen	100	32	248	280

1616 rows × 4 columns

Export table to file

In [51]:

```
out_file = open('demographics.tsv', 'w')
demogrphx.to_csv(out_file, sep='\t')
```

Quiz

- *True* or *false?*
  - Columns of a Pandas DataFrame share all the same two (explicit+positional) indexes.
  - Columns can be added a Pandas DataFrame also after instantiation
  - Columns of a Pandas DataFrame must be all of same type
  - Columns in a Pandas DataFrame are essentially Series instances

- Which of the following is the correct way to import the CSV file `demogrphx.csv` for reading and using the 'Name' column as the index row?
  - `pd.read_csv('demogrphx.csv', index_col='Name')`
  - `pd.read_csv('demogrphx.csv', index='Name')`
  - `pd.read_csv('demogrphx.csv', index=0, index_col_name='Name')`
  - `pd.read_csv('demogrphx.csv', index_col=0)`

source (in part): `https://realpython.com/quizzes`

Quiz

- *True* or *false?*
  - Columns of a Pandas DataFrame share all the same two (explicit+positional) indexes. true
  - Columns can be added a Pandas DataFrame also after instantiation true
  - Columns of a Pandas DataFrame must be all of same type false
  - Columns in a Pandas DataFrame are essentially Series instances true

- Which of the following is the correct way to import the CSV file `demogrphx.csv` for reading and using the 'Name' column as the index row?
  - `pd.read_csv('demogrphx.csv', index_col='Name')` ✔
  - `pd.read_csv('demogrphx.csv', index='Name')`
  - `pd.read_csv('demogrphx.csv', index=0, index_col_name='Name')`
  - `pd.read_csv('demogrphx.csv', index_col=0)` ✔

source (in part): `https://realpython.com/quizzes`

Pandas Series

Pandas DataFrame

Multi-Indexing DataFrames

Multi-Indexing

In [52]:

```
multi_idx = pd.MultiIndex.from_tuples(zip(demogrphx.index, demogrphx.Age))

# reset table to default index
demogrphx.reset_index(inplace=True)

# create index from columns 'FederalState' and ('Age', 'Age')
demogrphx.set_index(['FederalState', ('Age', 'Age')], inplace=True)
# change the name of index column ('Age', 'Age') to 'Age'
demogrphx.index.names = ('FederalState', 'Age')
demogrphx
```

Out[52]:

		Population		
		Male	Female	Total
FederalState	Age			
Schleswig-Holstein	0	11132	10400	21532
	1	11504	10360	21864
	2	11733	11067	22800
	3	12214	11147	23361
	4	12142	10945	23087
...	...	...	...	...
Thueringen	96	149	599	748
	97	75	476	551
	98	47	275	322
	99	30	181	211
	100	32	248	280

1616 rows × 3 columns

Reading file with multiple index columns

In [53]:

```
demogrphx = pd.read_table('12111-04-01-4-B_processed3.tsv', index_col = [0, 1],
                          header=[0, 1])
demogrphx
```

Out[53]:

		Population		
		Male	Female	Total
FederalState	Age			
Schleswig-Holstein	0	11132	10400	21532
	1	11504	10360	21864
	2	11733	11067	22800
	3	12214	11147	23361
	4	12142	10945	23087
...	...	...	...	...
Thueringen	96	149	599	748
	97	75	476	551
	98	47	275	322
	99	30	181	211
	100	32	248	280

1616 rows × 3 columns

Accessing elements of a multi-indexed table

In [54]:
```
demogrphx.loc['Schleswig-Holstein', 0]
```

Out[54]:
```
Population  Male      11132
            Female    10400
            Total     21532
Name: (Schleswig-Holstein, 0), dtype: int64
```

Slicing multi-indexed tables

In [55]:
```
# make sure to sort index prior to analysis!
demogrphx.sort_index(inplace=True)
```

Slicing based on first level

In [56]:
```
demogrphx['Hessen':'Nordrhein-Westfalen']
```

Out[56]:

		Population		
		Male	Female	Total
FederalState	Age			
Hessen	0	25784	24393	50177
	1	25935	24265	50200
	2	26300	24978	51278
	3	27072	25291	52363
	4	26638	24793	51431
...	...	...	...	...
Nordrhein-Westfalen	96	1114	5809	6923
	97	701	4050	4751
	98	422	2734	3156
	99	243	1743	1986
	100	317	2534	2851

404 rows × 3 columns

Slicing based on first & second level

In [57]:

```
demogrphx[('Hessen', 99):('Nordrhein-Westfalen', 1)]
```

Out[57]:

		Population		
		Male	Female	Total
FederalState	Age			
Hessen	**99**	117	593	710
	100	125	906	1031
Mecklenburg-Vorpommern	**0**	6486	6323	12809
	1	6421	6481	12902
	2	6498	6453	12951
...	**...**	...	...	...
Niedersachsen	**98**	203	1255	1458
	99	143	862	1005
	100	163	1145	1308
Nordrhein-Westfalen	**0**	72606	68444	141050
	1	72730	69066	141796

206 rows × 3 columns

Slicing with IndexSlice

In [58]:

```
# instance of IndexSlice can be used universally
idx = pd.IndexSlice
demogrphx.loc[idx[:, 20:25], idx[:, 'Total']]
```

Out[58]:

		Population
		Total
FederalState	**Age**	
Baden-Wuerttemberg	**20**	132022
	21	131097
	22	132252
	23	131513
	24	128160
...	**...**	...
Thueringen	**21**	27372
	22	28285
	23	29341
	24	28365
	25	27623

96 rows × 1 columns

Data aggregation

Total number of 20-to-25 year-olds in Germany:

In [59]:
```
demogrphx.loc[idx[:, 20:25], idx[:, 'Total']].sum()
```

Out[59]:
```
Population  Total    5789320
dtype: int64
```

Column/Index-based grouping

Population count per federal state

In [60]:

```
demogrphx.groupby('FederalState').sum()
```

Out[60]:

	Population		
	Male	Female	Total
FederalState			
Baden-Wuerttemberg	5132555	5354105	10486660
Bayern	6062701	6334913	12397614
Berlin	1599653	1692712	3292365
Brandenburg	1208327	1247453	2455780
Bremen	316102	334761	650863
Hamburg	825451	881245	1706696
Hessen	2913862	3057954	5971816
Mecklenburg-Vorpommern	793140	816841	1609981
Niedersachsen	3803776	3974216	7777992
Nordrhein-Westfalen	8517934	9020318	17538252
Rheinland-Pfalz	1950352	2039456	3989808
Saarland	485050	514573	999623
Sachsen	1977567	2079232	4056799
Sachsen-Anhalt	1117016	1170024	2287040
Schleswig-Holstein	1360484	1439635	2800119
Thueringen	1076074	1112515	2188589

Population count per age group

In [61]:
```
demogrphx.groupby('Age').sum()
```

Out[61]:

	Population		
	Male	Female	Total
Age			
0	336808	319265	656073
1	338210	320570	658780
2	343332	326040	669372
3	351954	333415	685369
4	344300	324494	668794
...	...	...	...
96	5261	26132	31393
97	3533	18625	22158
98	2175	12575	14750
99	1354	8143	9497
100	1679	11766	13445

101 rows × 3 columns

Population grouped on 'total population' column

In [62]:
```
demogrphx.groupby(('Population', 'Total')).sum()
```

Out[62]:

	Population	
	Male	**Female**
(Population, Total)		
113	21	92
126	13	113
141	17	124
150	26	124
158	34	124
...	...	...
302302	152403	149899
305652	154389	151263
309822	156763	153059
313219	158875	154344
313362	158873	154489

1602 rows × 2 columns

Masking

Using a Boolean array to select rows of a table is called *masking*:

In [63]:
```
demogrphx.Population.Total < 150
```

Out[63]:
```
FederalState          Age
Baden-Wuerttemberg    0        False
                      1        False
                      2        False
                      3        False
                      4        False
                               ...
Thueringen            96       False
                      97       False
                      98       False
                      99       False
                      100      False
Name: Total, Length: 1616, dtype: bool
```

In [64]:
```
demogrphx[demogrphx.Population.Total < 150]
```

Out[64]:

		Population		
		Male	Female	Total
FederalState	**Age**			
Bremen	**99**	13	113	126
Saarland	**99**	21	92	113
	100	17	124	141

Two or more Boolean arrays can be integrated with Boolean operation `&` (AND / *conjunction*) or `|` (OR / *disjunction*).

In [65]:
```
demogrphx[(demogrphx.Population.Total > 1000) &
          (demogrphx.Population.Total < 1200)]
```

Out[65]:

		Population		
		Male	**Female**	**Total**
FederalState	**Age**			
Baden-Wuerttemberg	**99**	169	957	1126
Berlin	**97**	145	968	1113
	100	125	900	1025
Hessen	**98**	151	941	1092
	100	125	906	1031
Niedersachsen	**99**	143	862	1005
Rheinland-Pfalz	**97**	173	924	1097
Sachsen-Anhalt	**93**	172	900	1072
Schleswig-Holstein	**95**	217	925	1142

Quiz

- *True* or *false*?
  - Pandas multi-indexes are *hierarchical* indexes
  - Multi-indexes can be used to index columns and rows of a DataFrame
  - DataFrame supports only grouping for columns that are indexes
  - Masking is a fast way to access columns of a DataFrame

Quiz

- *True* or *false*?
  - Pandas multi-indexes are *hierarchical* indexes — true
  - Multi-indexes can be used to index columns and rows of a DataFrame — true
  - DataFrame supports only grouping for columns that are indexes — false
  - Masking is a fast way to access columns of a DataFrame — false

Recap

Summary

Pandas:

- Series
  - Creating & indexing
  - Accessing elements and subsets
- DataFrame
  - Creating & indexing
  - Accessing columns, rows, and elements
  - Broadcasting and vectorized operations
  - Reading & writing tables
- Multi-Indexing
  - Creating multi-indexes
  - Slicing
  - Grouping & Masking

What comes next?

- Have a look at the Jupyter Notebook of this lecture
- Play with the Census data set using Pandas
- Further reading about Pandas: Chapter 3 of the “Python Data Science Handbook”:
  `https://jakevdp.github.io/PythonDataScienceHandbook/`