Chapter 9: Tabular data

The **pandas** package is a high-level data manipulation tool for Python.

The name *pandas* is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals.

How to install pandas?

If you have Anaconda installed, then pandas was already installed together with it.

If you have a standalone Python3 and Jupyter Notebook installation, open a command prompt / terminal and type in:

```
pip3 install pandas xlrd openpyxl
```

The `xlrd` and `openpyxl` packages are required for managing Microsoft Excel files.
If you have Anaconda installed, these packages were also included in the default installation.

Support OpenDocument format

The `xlrd` package is used for older format MS Excel files (`.xls`), while `openpyxl` is used for newer format MS Excel files (`.xlsx`).

In case support for *OpenDocument* format is required (`.ods` files), e.g. for compatibility with OpenOffice or LibreOffice, the `odf` package can be used:

- Install with Anaconda Command Prompt: `conda install odf`
- Install with Python Package Installer: `pip3 install odf`

How to use pandas?

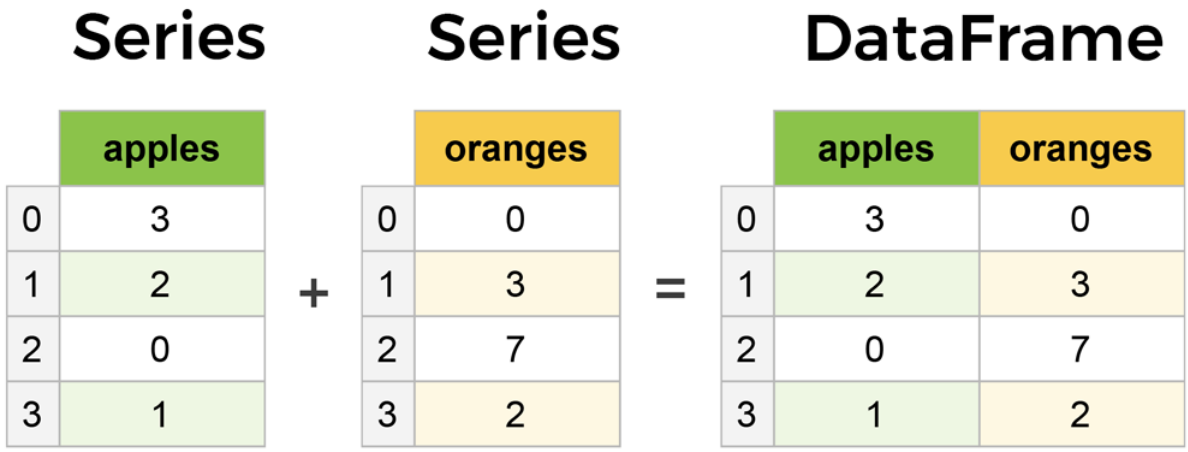
The pandas package is a module which you can simply import. It is usually aliased with the `pd` abbreviation:

```
import pandas as pd
```

Series and Dataframes

The primary two components of pandas are the **Series** and **DataFrame**.

A *Series* is essentially a column, and a *DataFrame* is a multi-dimensional table made up of a collection of *Series*.



We can work with DataFrames and Series in a similar way, since many operations support both of them. (E.g. calculating the mean value of a Series or a DataFrame.)

Create a DataFrame from scratch

A Python *dictionary* can easily be converted to a pandas *DataFrame*. Each `(key, value)` tuple in the dictionary corresponds to a column in the resulting DataFrame. The values in the rows for each column are given as lists.

In [1]:

```
import pandas as pd

data = {
    'apples': [3, 2, 0, 1, 5, 0, 4],
    'oranges': [0, 3, 7, 2, 1, 6, 2]
}

df = pd.DataFrame(data)
display(df)
```

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2
4	5	1
5	0	6
6	4	2

By default the rows in the DataFrame are *indexed* numerically from 0, but we can set a custom **Index**:

In [2]:

```
df = pd.DataFrame(data, index = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday'])
display(df)
```

	apples	oranges
Monday	3	0
Tuesday	2	3
Wednesday	0	7
Thursday	1	2
Friday	5	1
Saturday	0	6
Sunday	4	2

Remark: the `display()` function is a special Jupyter Notebook function to provide "prettier" output. Where `display()` is used, `print()` could also be used.

In [3]:

```
print(df)
```

```
           apples  oranges
Monday          3        0
Tuesday         2        3
Wednesday       0        7
Thursday        1        2
Friday          5        1
Saturday        0        6
Sunday          4        2
```

Access the columns of a DataFrame

In [4]:

```
print('Apples purchased over the week:')
print(df['apples']) # df is a DataFrame, df['apples'] is a Series
```

```
Apples purchased over the week:
Monday       3
Tuesday      2
Wednesday    0
Thursday     1
Friday       5
Saturday     0
Sunday       4
Name: apples, dtype: int64
```

Values of single cells can also be accessed:

In [5]:

```
print('Apples purchased on Monday:')
print(df['apples']['Monday'])
print(df['apples'][0])
```

```
Apples purchased on Monday:
3
3
```

Access the rows of a DataFrame

Rows can be accessed through the `loc` property with their textual indexes:

In [6]:

```
print('Fruits purchased on Monday:')
print(df.loc['Monday'])
```

```
Fruits purchased on Monday:
apples     3
oranges    0
Name: Monday, dtype: int64
```

Rows can also be accessed through the `iloc` property with ther numerical indexes:

In [7]:

```
print('Fruits purchased on Monday:')
print(df.iloc[0])
```

```
Fruits purchased on Monday:
apples     3
oranges    0
Name: Monday, dtype: int64
```

Accessing single cells:

In [8]:

```
print('Apples purchased on Monday:')
print(df.loc['Monday']['apples'])
print(df.loc['Monday'][0])
print(df.iloc[0]['apples'])
print(df.iloc[0][0])
```

```
Apples purchased on Monday:
3
3
3
3
```

Iterate over the rows of a DataFrame

Manually iterating through all the rows with the `iterrows()` method:

In [9]:

```
for index, row in df.iterrows():
    print("Index: {0}, Apples: {1}, Oranges: {2}".format(index, row["apples"], r
ow["oranges"]))
```

```
Index: Monday, Apples: 3, Oranges: 0
Index: Tuesday, Apples: 2, Oranges: 3
Index: Wednesday, Apples: 0, Oranges: 7
Index: Thursday, Apples: 1, Oranges: 2
Index: Friday, Apples: 5, Oranges: 1
Index: Saturday, Apples: 0, Oranges: 6
Index: Sunday, Apples: 4, Oranges: 2
```

Alternatively, if only the index values are required, the `df.index` list can be iterated:

In [10]:

```
print(df.index)
```

```
Index(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Satu
rday',
       'Sunday'],
      dtype='object')
```

Reading external files

The *pandas* library has a great support for reading (and writing) external data files, like CSV files, Excel files, JSON files, etc.

Let's use the *European countries datatset*. The dataset contains the country name, capital city name, area (in km^2), population (in millions) and the region data for 43 European countries respectively.
Data source: [EuroStat (https://ec.europa.eu/eurostat/)](https://ec.europa.eu/eurostat/)

The dataset is given in the `data/countries_europe.csv` file, which we can read with th `read_csv()` method, passing the file path and the delimiter symbol (latter will be discussed soon).

In [11]:

```
import pandas as pd

countries = pd.read_csv('../data/countries_europe.csv', delimiter = ';')
display(countries)
```

	Country	Capital	Area (km2)	Population (millions)	Region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western
3	Belgium	Brussels	30519	10.00	Western
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern
...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern
39	Serbia	Belgrade	66577	7.20	Southern
40	Slovakia	Bratislava	49035	5.30	Central
41	Slovenia	Ljubljana	20250	2.00	Southern
42	Ukraine	Kiev	603700	51.80	Eastern

A **comma-separated values (CSV) file** is a delimited text file that uses a **comma** to separate values. The separator (also called the *delimiter*) can be another character than a comma, often a **semicolon** is used. The default delimiter is the comma, but we can easily configure it int the `read_csv()` method call.

A CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format.

For example:

```
Country;Capital;Area (km2);Population (millions);Region
Albania;Tirana;28748;3.2;Southern
Andorra;Andorra la Vella;468;0.07;Western
Austria;Vienna;83857;7.6;Western
Belgium;Brussels;30519;10;Western
Bosnia and Herzegovina;Sarajevo;51130;4.5;Southern
Bulgaria;Sofia;110912;9;Southern
Czech Republic;Prague;78864;10.4;Central
Denmark;Copenhagen;43077;5.1;Northern
United Kingdom;London;244100;57.2;Western
...
```

Overwrite the used column names:

In [12]:

```
countries.columns = ['country', 'capital', 'area', 'population', 'region']
display(countries)
```

	country	capital	area	population	region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western
3	Belgium	Brussels	30519	10.00	Western
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern
...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern
39	Serbia	Belgrade	66577	7.20	Southern
40	Slovakia	Bratislava	49035	5.30	Central
41	Slovenia	Ljubljana	20250	2.00	Southern
42	Ukraine	Kiev	603700	51.80	Eastern

Use further reading functions, like `read_excel`, `read_json` to easily load other file formats into a pandas DataFrame.

The same European country dataset is given in the `data/countries_europe.xls` and `data/countries_europe.xlsx` MS Excel files:

In [13]:

```
countries = pd.read_excel('../data/countries_europe.xls')
countries.columns = ['country', 'capital', 'area', 'population', 'region']
display(countries)
```

	country	capital	area	population	region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western
3	Belgium	Brussels	30519	10.00	Western
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern
...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern
39	Serbia	Belgrade	66577	7.20	Southern
40	Slovakia	Bratislava	49035	5.30	Central
41	Slovenia	Ljubljana	20250	2.00	Southern
42	Ukraine	Kiev	603700	51.80	Eastern

In [14]:

```
countries = pd.read_excel('../data/countries_europe.xlsx')
countries.columns = ['country', 'capital', 'area', 'population', 'region']
display(countries)
```

	country	capital	area	population	region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western
3	Belgium	Brussels	30519	10.00	Western
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern
...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern
39	Serbia	Belgrade	66577	7.20	Southern
40	Slovakia	Bratislava	49035	5.30	Central
41	Slovenia	Ljubljana	20250	2.00	Southern
42	Ukraine	Kiev	603700	51.80	Eastern

Working with DataFrames

Query the row and column count of a DataFrame:

In [15]:

```
print('Number of rows: {0}'.format(len(countries)))
print('Number of rows: {0}'.format(countries.shape[0]))
print('Number of columns: {0}'.format(countries.shape[1]))
```

```
Number of rows: 43
Number of rows: 43
Number of columns: 5
```

In [16]:

```
print('Number of rows by columns:')
print(countries.count())
```

```
Number of rows by columns:
country       43
capital       43
area          43
population    43
region        43
dtype: int64
```

Remark: if a column contains empty cells, we would see different numbers here.

The first or last few rows of a DataFrame can be fetched with the `head(n)` and `tail(n)` methods. They are especially useful with large DataFrames.

In [17]:

```
display(countries.head(3))
```

	country	capital	area	population	region
0	Albania	Tirana	28748	3.20	Southern
1	Andorra	Andorra la Vella	468	0.07	Western
2	Austria	Vienna	83857	7.60	Western

In [18]:

```
display(countries.tail(3))
```

	country	capital	area	population	region
40	Slovakia	Bratislava	49035	5.3	Central
41	Slovenia	Ljubljana	20250	2.0	Southern
42	Ukraine	Kiev	603700	51.8	Eastern

Add a new column to a DataFrame

Calculate the population density of each country and add it as a new column to the `countries` *DataFrame*.

First, create a list of the density values:

In [19]:

```
density = []
for i in range(len(countries)):
    density.append(countries['population'][i] * 1e6 / countries['area'][i])

print(density)
```

```
[111.31209127591485, 149.57264957264957, 90.63047807577185, 327.6647
3344473934, 88.01095247408567, 81.14541257934218, 131.8725907892067
3, 118.39264572741834, 234.33019254403933, 35.47671840354767, 49.614
643545279385, 14.490824941962767, 103.3154706644729, 75.782262403661
8, 436.1535967936817, 83.1858407079646, 49.79867108689157, 2.9126213
59223301, 202.07587030403232, 120.88920728021671, 40.81632653061224
4, 187.5, 55.214723926380366, 154.67904098994586, 81.67075020417687,
111.78468549808676, 949.367088607595, 130.56379821958458, 15000.0, 4
3.440486533449175, 220.14216814828507, 12.967885956705786, 190.85426
36842507, 113.6498933855762, 97.6842105263158, 491.8032786885246, 7
6.86486443652903, 162.25510377061488, 18.890400120898562, 108.145455
63783288, 108.08606097685326, 98.76543209876543, 85.80420738777539]
```

Then add a new column to the DataFrame:

In [20]:

```
countries['density'] = density
display(countries)
```

	country	capital	area	population	region	density
0	Albania	Tirana	28748	3.20	Southern	111.312091
1	Andorra	Andorra la Vella	468	0.07	Western	149.572650
2	Austria	Vienna	83857	7.60	Western	90.630478
3	Belgium	Brussels	30519	10.00	Western	327.664733
4	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern	88.010952
...	...	...	...	...	...	...
38	Sweden	Stockholm	449964	8.50	Northern	18.890400
39	Serbia	Belgrade	66577	7.20	Southern	108.145456
40	Slovakia	Bratislava	49035	5.30	Central	108.086061
41	Slovenia	Ljubljana	20250	2.00	Southern	98.765432
42	Ukraine	Kiev	603700	51.80	Eastern	85.804207

Calculating aggregated values

Summation, mean and median values:

In [21]:

```
print("Sum population: {0:.2f} million".format(countries["population"].sum()))
print("Mean population: {0:.2f} million".format(countries["population"].mean()))
print("Median population: {0:.2f} million".format(countries["population"].median
()))
```

```
Sum population: 572.16 million
Mean population: 13.31 million
Median population: 5.10 million
```

Maximum and minimum values and their indexes:

In [22]:

```
print("Max population:{0:.2f} million".format(countries["population"].max()))
print("Max population index: {0}".format(countries["population"].idxmax()))
print("Min population:{0:.2f} million".format(countries["population"].min()))
print("Min population index: {0}".format(countries["population"].idxmin()))
```

```
Max population:78.60 million
Max population index: 30
Min population:0.03 million
Min population index: 21
```

Standard deviation:

In [23]:

```
print("Standard deviation of population: {0:.2f} million".format(countries["popu
lation"].std()))
```

```
Standard deviation of population: 19.42 million
```

Quantile calculation:

In [24]:

```
print("The 90% quantile for the European countries:")
print(countries.quantile(0.9))
```

```
The 90% quantile for the European countries:
area          353262.600000
population        49.200000
density          308.997825
Name: 0.9, dtype: float64
```

Which means that e.g. the top 5 countries (top 10%) has a higher population than 49.2 million.

Calculate all basic statistical data for all columns at once:

In [25]:

```
display(countries.describe())
```

	area	population	density
count	43.000000	43.000000	43.000000
mean	136551.162791	13.306047	489.478549
std	163143.708680	19.415679	2271.195151
min	2.000000	0.030000	2.912621
25%	29633.500000	2.150000	76.323563
50%	65200.000000	5.100000	108.086061
75%	222550.000000	10.400000	158.467072
max	603700.000000	78.600000	15000.000000

Sorting the DataFrame

A DataFrame can be sorted into a **new** DataFrame through the `sort_values` function. The original DataFrame remains intact.

In [26]:

```
bypopulation = countries.sort_values(by = 'population', ascending = False)
display(bypopulation)
```

	country	capital	area	population	region	density
30	Germany	Berlin	357042	78.60	Western	220.142168
32	Italy	Rome	301277	57.50	Southern	190.854264
8	United Kingdom	London	244100	57.20	Western	234.330193
12	France	Paris	543965	56.20	Western	103.315471
42	Ukraine	Kiev	603700	51.80	Eastern	85.804207
36	Spain	Madrid	504782	38.80	Southern	76.864864
...	...	...	...	...	...	...
17	Iceland	Reykjavik	103000	0.30	Northern	2.912621
1	Andorra	Andorra la Vella	468	0.07	Western	149.572650
28	Monaco	Monaco	2	0.03	Southern	15000.000000
35	San Marino	San Marino	61	0.03	Southern	491.803279
21	Liechtenstein	Vaduz	160	0.03	Western	187.500000

Note that the row indices remained intact.
Task: which countries will display the following code cell?

In [27]:

```
print(bypopulation.loc[0])
print()
print(bypopulation.iloc[0])
```

```
country          Albania
capital           Tirana
area               28748
population           3.2
region           Southern
density       111.312091
Name: 0, dtype: object

country          Germany
capital           Berlin
area              357042
population          78.6
region           Western
density       220.142168
Name: 30, dtype: object
```

Now we can e.g. verify that the top 5 countries (top 10%) has a higher population than 49.2 million:

In [28]:

```
print("Population of the 4th country: {0:.1f} million".format(bypopulation.iloc[
4]["population"]))
print("Population of the 5th country: {0:.1f} million".format(bypopulation.iloc[
5]["population"]))
```

```
Population of the 4th country: 51.8 million
Population of the 5th country: 38.8 million
```

Note: a DataFrame can be sorted by modifying it (and without creating a new one) by passing the `inplace = True` argument to the `sort_values()` method:

```
countries.sort_values(by = 'population', ascending = False, inplace = Tru
e)
```

Sorting by multiple columns

A *DataFrame* can be sorted using multiple columns, by passing a list of columns to the `by` parameter. (And a list of boolean values to the `ascending` parameter.)

In [29]:

```
byregion = countries.sort_values(by = ['region', 'population'], ascending = [True, False])
display(byregion)
```

	country	capital	area	population	region	density
19	Poland	Warsaw	312683	37.80	Central	120.889207
6	Czech Republic	Prague	78864	10.40	Central	131.872591
25	Hungary	Budapest	93036	10.40	Central	111.784685
40	Slovakia	Bratislava	49035	5.30	Central	108.086061
42	Ukraine	Kiev	603700	51.80	Eastern	85.804207
...	...	...	...	...	...	...
37	Switzerland	Berne	41293	6.70	Western	162.255104
16	Ireland	Dublin	70283	3.50	Western	49.798671
23	Luxembourg	Luxembourg	2586	0.40	Western	154.679041
1	Andorra	Andorra la Vella	468	0.07	Western	149.572650
21	Liechtenstein	Vaduz	160	0.03	Western	187.500000

Indexing the DataFrame

We can assign one of the columns as the index column. The indexer column must contain unique values.

In [30]:

```
countries_indexed = countries.set_index('country')
display(countries_indexed)
```

country	capital	area	population	region	density
Albania	Tirana	28748	3.20	Southern	111.312091
Andorra	Andorra la Vella	468	0.07	Western	149.572650
Austria	Vienna	83857	7.60	Western	90.630478
Belgium	Brussels	30519	10.00	Western	327.664733
Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern	88.010952
...	...	...	...	...	...
Sweden	Stockholm	449964	8.50	Northern	18.890400
Serbia	Belgrade	66577	7.20	Southern	108.145456
Slovakia	Bratislava	49035	5.30	Central	108.086061
Slovenia	Ljubljana	20250	2.00	Southern	98.765432
Ukraine	Kiev	603700	51.80	Eastern	85.804207

In the indexed *DataFrame*, rows can be accessed through the values of the index column with the already used `loc` property:

In [31]:

```
print(countries_indexed.loc["Hungary"])
```

```
capital         Budapest
area               93036
population          10.4
region           Central
density       111.784685
Name: Hungary, dtype: object
```

Remark: by default the `set_index()` method call removes the indexer column. We can keep the indexer column also as a "normal" column through setting the `drop` parameter to `False` .

In [32]:

```
countries_indexed = countries.set_index('country', drop = False)
display(countries_indexed)
```

country	country	capital	area	population	region	density
Albania	Albania	Tirana	28748	3.20	Southern	111.312091
Andorra	Andorra	Andorra la Vella	468	0.07	Western	149.572650
Austria	Austria	Vienna	83857	7.60	Western	90.630478
Belgium	Belgium	Brussels	30519	10.00	Western	327.664733
Bosnia and Herzegovina	Bosnia and Herzegovina	Sarajevo	51130	4.50	Southern	88.010952
...	...	...	...	...	...	...
Sweden	Sweden	Stockholm	449964	8.50	Northern	18.890400
Serbia	Serbia	Belgrade	66577	7.20	Southern	108.145456
Slovakia	Slovakia	Bratislava	49035	5.30	Central	108.086061
Slovenia	Slovenia	Ljubljana	20250	2.00	Southern	98.765432
Ukraine	Ukraine	Kiev	603700	51.80	Eastern	85.804207

Filtering the DataFrame

Configure a condition as a boolean expression:

In [33]:

```
condition = countries["region"] == "Central"
print(type(condition))
```

```
<class 'pandas.core.series.Series'>
```

As we have seen, the result is a *Series*, so a new column! Evaluate it:

In [34]:

```
print(condition)
```

```
0     False
1     False
2     False
3     False
4     False
5     False
6      True
7     False
...
39    False
40     True
41    False
42    False
Name: region, dtype: bool
```

The series in the `condition` variable stores the boolean `True` / `False` value for each row, whether the profit was positive for that row or not.

Now filter the DataFrame by the condition. As we can observe, only the rows with the logical `True` value in the `condition` series remained:

In [35]:

```
centralEuropean = countries[condition]
display(centralEuropean)
```

	country	capital	area	population	region	density
6	Czech Republic	Prague	78864	10.4	Central	131.872591
19	Poland	Warsaw	312683	37.8	Central	120.889207
25	Hungary	Budapest	93036	10.4	Central	111.784685
40	Slovakia	Bratislava	49035	5.3	Central	108.086061

The whole workflow can be achieved in a single statement:

In [36]:

```
display(countries[countries["region"] == "Central"])
```

	country	capital	area	population	region	density
6	Czech Republic	Prague	78864	10.4	Central	131.872591
19	Poland	Warsaw	312683	37.8	Central	120.889207
25	Hungary	Budapest	93036	10.4	Central	111.784685
40	Slovakia	Bratislava	49035	5.3	Central	108.086061

Summary exercise on DataFrames

Task 1

Display the name of the countries which have a population of less than 1 million.

In [37]:

```
small_countries = countries[countries["population"] < 1]
print(small_countries["country"])
```

```
1           Andorra
17          Iceland
21    Liechtenstein
23       Luxembourg
26            Malta
28           Monaco
29       Montenegro
35       San Marino
Name: country, dtype: object
```

Task 2

Write a program that calculates which Western-European country has the largest area?

In [38]:

```
max_index = countries[countries["region"] == "Western"]["area"].idxmax()
print(countries.iloc[max_index])
```

```
country           France
capital            Paris
area              543965
population          56.2
region           Western
density       103.315471
Name: 12, dtype: object
```